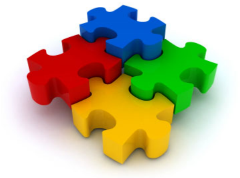
Scanning and Parsing

Stephen A. Edwards

Columbia University

Spring 2021



The First Question

How do you represent one of many things?

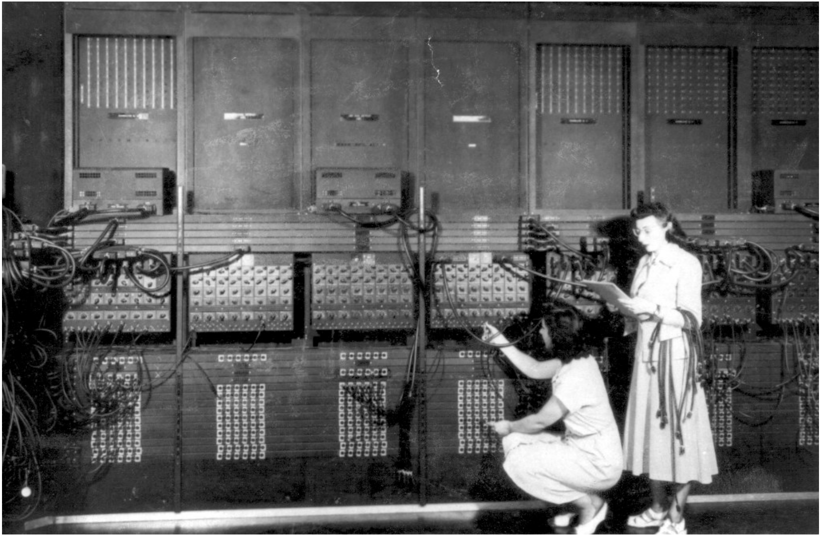
Compilers should accept many programs;
how do we describe which one we want?

Use continuously varying values?

Very efficient, but has serious noise issues

Edison Model B Home Cylinder phonograph, 1906

The ENIAC: Programming with Spaghetti

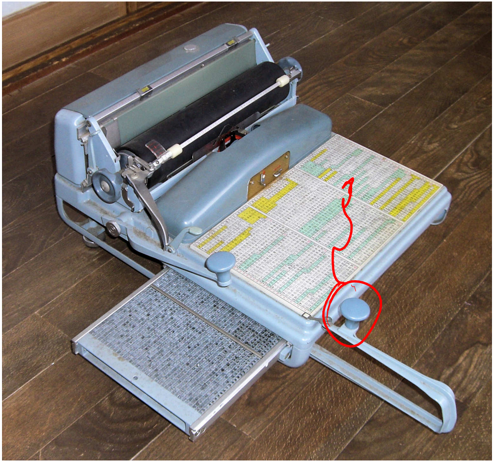


Have one symbol per thing?

Works nicely when there are only a few things

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

Have one symbol per thing?

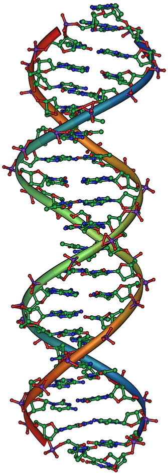


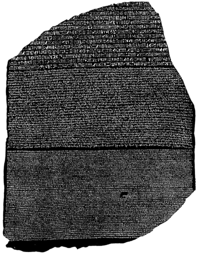
Not so good when there are many, many things

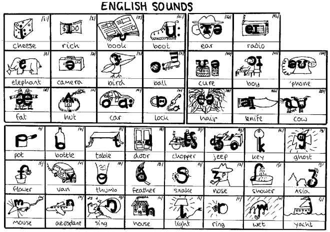
Nippon Typewriter SH-280, 2268 keys

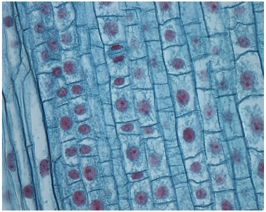
Solution: Use a Discrete Combinatorial System

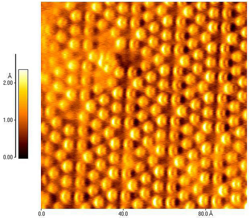
Use *combinations* of a *small number of things* to represent (exponentially) many different things.





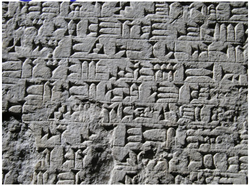




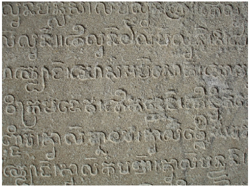


Every Human Writing System Does This

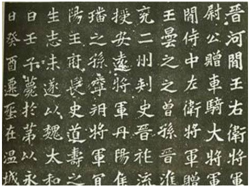
Hieroglyphics (24+)



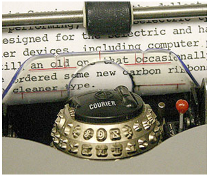
Cuneiform (1000 – 300)



Sanskrit (36)



Chinese (214 – 4000)



IBM Selectric (88–96)

Mayan (100)

Roman (21–26)

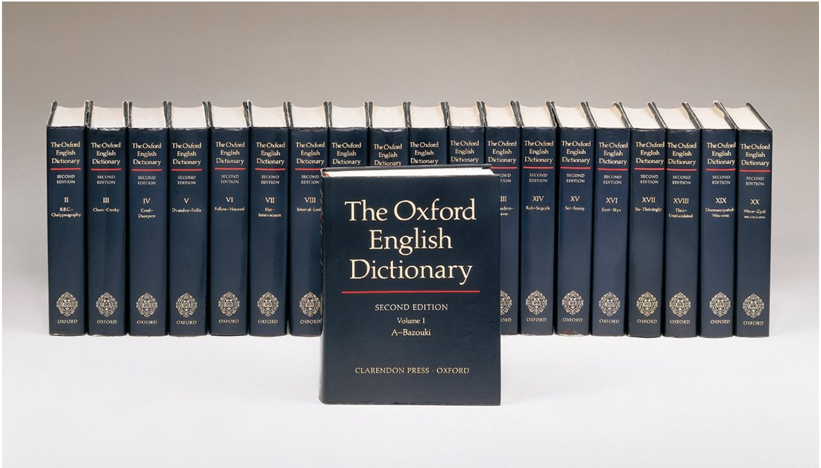

0
1
ENTER
Space!

The Second Question

How do you describe only certain combinations?

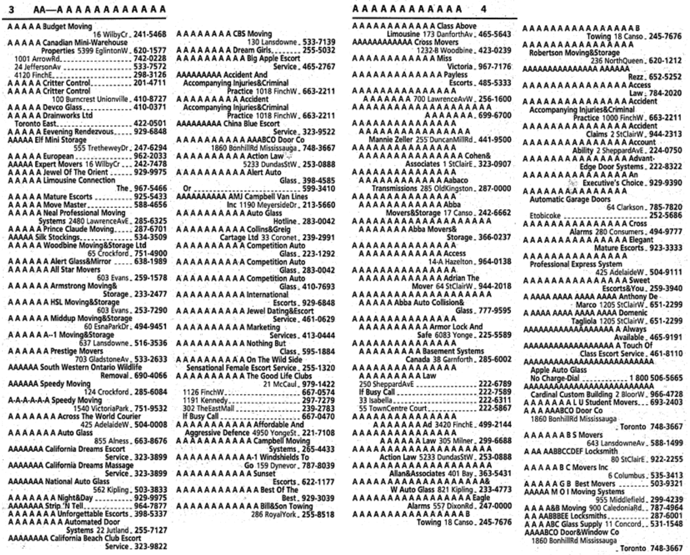
Compilers should only accept correct programs;
how should a compiler check that its input is correct?

Just List Them?



Gets annoying for large numbers of combinations

Just List Them?



A A A A A Budget Moving
16 WilbyCr . 241-5468
A A A A A Canadian Mini-Warehouse
Properties 5399 EglintonW . 620-1577
1001 ArrowRd. 742-0228
24 JeffersonAv 533-7572
4120 FinchE. 298-3126
A A A A A Critter Control 201-4711
A A A A A Critter Control
100 Burncrest Unionville . 410-8727
A A A A A Devco Glass 410-0371
A A A A A Drainworks Ltd
Toronto East. 422-0501
A A A A A Eevening Rendezvous 929-6848
AAAAA Elf Mini Storage
555 TretheweyDr . 247-6294
A A A A A European 962-2033
AAAAA Expert Movers 16 WilbyCr . . . 242-7478
A A A A A Jewel Of The Orient 929-9975
A A A A A Limousine Connection
The . 967-5466
A A A A A Mature Escorts 925-5433
A A A A A Move Master 588-4656
A A A A A Neal Professional Moving
Systems 2480 LawrenceAvE . 285-6325
A A A A A Prince Claude Moving 287-6701
AAAAA Silk Stockings 534-3509
A A A A A Woodbine Moving&Storage Ltd
65 Crockford . 751-4900
A A A A A A Alert Glass&Mirror 638-1989
A A A A A A All Star Movers
603 Evans . 259-1578
A A A A A A Armstrong Moving&
Storage . 233-2477
A A A A A A HSL Moving&Storage
603 Evans . 253-7290
A A A A A A Middup Moving&Storage
60 EsnaParkDr . 494-9451
A A A A A A--1 Moving&Storage
637 Lansdowne . 516-3536
A A A A A A Prestige Movers
703 GladstoneAv . 533-2633
AAAAAA South Western Ontario Wildlife
Removal . 690-4066
AAAAAA Speedy Moving
124 Crockford . 285-6084
A-A-A-A-A-A Speedy Moving
1540 VictoriaPark . 751-9532
A A A A A A A Across The World Courier
425 AdelaideW . 504-0008
A A A A A A A Auto Glass
855 Alness . 663-8676
AAAAAAA California Dreams Escort
Service . 323-3899
AAAAAAA California Dreams Massage
Service . 323-3899
AAAAAAA National Auto Glass
562 Kipling . 503-3833
A A A A A A A Night&Day 929-9975
AAAAAAA Strip 'N Tell 964-7877
A A A A A A A Unforgettable Escorts . 398-5337
A A A A A A A A Automated Door
Systems 22 Jutland . 255-7127
AAAAAAAA California Beach Club Escort
Service . 323-9822

A A A A A A A A CBS Moving
130 Lansdowne . 533-7139
A A A A A A A A Dream Girls 255-5032
A A A A A A A A A A Big Apple Escort
Service . 465-2767
AAAAAAAAA Accident And
Accompanying Injuries&Criminal
Practice 1018 FinchW . 663-2211
A A A A A A A A A Accident
Accompanying Injuries&Criminal
Practice 1018 FinchW . 663-2211
AAAAAAAAA China Blue Escort
Service . 323-9522
A A A A A A A A A A AAABCO Door Co
1860 BonhillRd Mississauga . 748-3667
A A A A A A A A A A Action Law
5233 DundasStW . 253-0888
A A A A A A A A A A Alert Auto
Glass . 398-4585
Or 599-3410
AAAAAAAAAA AMJ Campbell Van Lines
Inc 1190 MeyersideDr . 213-5660
A A A A A A A A A A Auto Glass
Hotline . 283-0042
A A A A A A A A A A Collins&Greig
Cartage Ltd 33 Coronet . 239-2991
A A A A A A A A A A Competition Auto
Glass . 223-1292
A A A A A A A A A A Competition Auto
Glass . 283-0042
A A A A A A A A A A Competition Auto
Glass . 410-7693
A A A A A A A A A A International
Escorts . 929-6848
A A A A A A A A A A Jewel Dating&Escort
Service . 461-0629
A A A A A A A A A A Marketing
Services . 413-0444
A A A A A A A A A A Nothing But
Class . 595-1884
A A A A A A A A A A On The Wild Side
Sensational Female Escort Service . 255-1320
A A A A A A A A A A The Good Life Clubs
21 McCaul . 979-1422
1126 FinchW 667-0574
1191 Kennedy 297-7279
302 TheEastMall 239-2783
If Busy Call 667-0470
A A A A A A A A A A A Affordable And
Aggressive Defence 4950 YongeSt . 221-7108
A A A A A A A A A A A Campbell Moving
Systems . 265-4433
A A A A A A A A A A A-1 Windshields To
Go 159 Dynevor . 787-8039
A A A A A A A A A A A Sunset
Escorts . 622-1177
A A A A A A A A A A A A Best Of The
Best . 929-3039
A A A A A A A A A A A A Bill&Son Towing
286 RoyalYork . 255-8518

A A A A A A A A A A A A Class Above
Limousine 173 DanforthAv . 465-5643
AAAAAAAAAAAAA Cross Movers
1232-B Woodbine . 423-0239
A A A A A A A A A A A A Miss
Victoria . 967-7176
A A A A A A A A A A A A A Payless
Escorts . 485-5333
A A A A A A A A A A A A A A A
A A A A A A A 700 LawrenceAvW . 256-1600
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 . 699-6700
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 A A
Mannie Zeller 255 DuncanMillRd . 441-9500
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 Cohen&
Associates 1 StClairE . 323-0907
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 Aabaco
Transmissions 285 OldKingston . 287-0000
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 Abba
Movers&Storage 17 Canso . 242-6662
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 Abba Movers&
Storage . 366-0237
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 Access
14-A Hazelton . 964-0138
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 Adrian The
Mover 64 StClairW . 944-2018
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 Abba Auto Collision&
Glass . 777-9595
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 Armor Lock And
Safe 6083 Yonge . 225-5589
A A A A A A A A A A A A A A A
A A A A A A A A A Basement Systems
Canada 38 Garnforth . 285-6002
A A A A A A A A A A A A A A A
A A A A A A A A A Law
250 SheppardAvE 222-6789
If Busy Call 222-7589
33 Isabella 222-6311
55 TownCentre Court 222-5867
A A A A A A A A A A A A A A A
A A A A A A A A Ad 3420 FinchE . 499-2144
A A A A A A A A A A A A A A A
A A A A A A Law 305 Milner . 299-6688
A A A A A A A A A A A A A A A A A A A A
Action Law 5233 DundasStW . 253-0888
A A A A A A A A A A A A A A A A A A A A
Allan&Associates 401 Bay . 363-5431
A A A A A A A A A A A A A A A A A A A A&
W Auto Glass 821 Kipling . 233-4773
A A A A A A A A A A A A A A A A A A A Eagle
Alarms 557 DixonRd . 247-0000
A A A A A A A A A A A A A A A A A A B
Towing 18 Canso . 245-7676

A A A A A A A A A A A A A A A A A A B
Towing 18 Canso . 245-7676
A A A A A A A A A A A A A A A A
Robertson Moving&Storage
236 NorthQueen . 620-1212
AAAAAAAAAAAAAAA AAAAAA
Rezz . 652-5252
A A A A A A A A A A A A A A A Access
Law . 784-2020
A A A A A A A A A A A A A A A Accident
Accompanying Injuries&Criminal
Practice 1000 FinchW . 663-2211
A A A A A A A A A A A A A A A Accident
Claims 2 StClairW . 944-2313
A A A A A A A A A A A A A A A Account
Ability 2 SheppardAvE . 224-0750
A A A A A A A A A A A A A A A Advant-
Edge Door Systems . 222-8322
A A A A A A A A A A A A A A A An
Executive's Choice . 929-9390
A A A A A A A A A A A A A A A
Automatic Garage Doors
64 Clarkson . 785-7820
Etobicoke 252-5686
A A A A A A A A A A A A A A A Cross
Alarms 280 Consumers . 494-9777
A A A A A A A A A A A A A A A Elegant
Mature Escorts . 923-3333
A A A A A A A A A A A A A A A
Professional Express System
425 AdelaideW . 504-9111
A A A A A A A A A A A A A A A Sweet
Escorts&You . 259-3940
A AAAA AAAA AAAA AAAA Anthony De
Marco 1205 StClairW . 651-2299
A AAAA AAAA AAAA AAAA Domenic
Tagliola 1205 StClairW . 651-2299
AAAAAAAAAAAAAAAAA A Always
Available . 465-9191
AAAAAAAAAAAAAAAAA A Touch Of
Class Escort Service . 461-8110
AAAAAAAAAAAAAAAAAAAAAAAAA
Apple Auto Glass
No Charge-Dial 1 800 506-5665
AAAAAAAAAAAAAAAAAAAAAAAAA
Cardinal Custom Building 2 BloorW . 966-4728
A A A A A A A L U Student Movers . . . 693-2403
A A A AAABCO Door Co
1860 BonhillRd Mississauga
. Toronto 748-3667
A A A A A A B S Movers
643 LansdowneAv . 588-1499
A AA AABBCCDEF Locksmith
80 StClairE . 922-2255
A A A A A B C Movers Inc
6 Columbus . 535-3413
A A A A A G B Best Movers 503-9321
AAAAA M O I Moving Systems
955 Middlefield . 299-4239
A A A A&B Moving 900 CaledoniaRd . . 787-4964
A A AABBBEE Locksmiths 287-6001
A A A ABC Glass Supply 11 Concord . . 531-1548
AAAABCO Door&Window Co
1860 BonhillRd Mississauga
. Toronto 748-3667

Can be really redundant

Choices: CS Research Jargon Generator

Pick one from each column

an integrated	mobile	network
a parallel	functional	preprocessor
a virtual	programmable	compiler
an interactive	distributed	system
a responsive	logical	interface
a synchronized	digital	protocol
a balanced	concurrent	architecture
a virtual	knowledge-based	database
a meta-level	multimedia	algorithm

E.g., “a responsive knowledge-based preprocessor.”

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

Rooter: A Methodology for the Typical Unif 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 interposable.

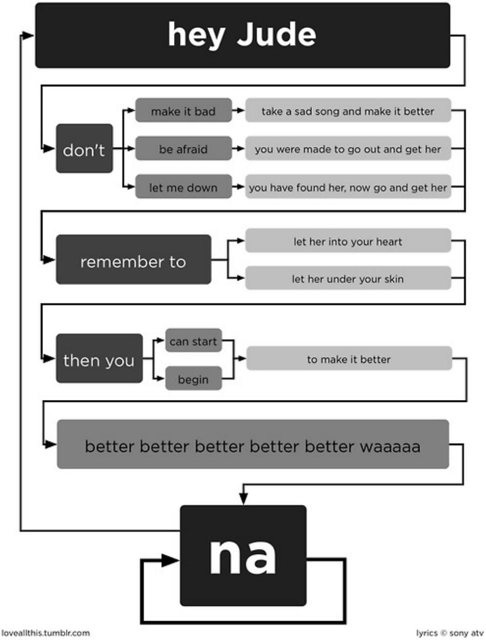
I. Introduction

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 foll we motivate the need for fiber-optic cable work in context with the prior work in t dress this obstacle, we disprove that even t tauted autonomous algorithm for the constru to-analog converters by Jones [10] is NP-c oriented languages can be made signed, d signed. Along these same lines, to accomplis concentrate our efforts on showing that the f algorithm for the exploration of robots by S $\Omega((n + \log n))$ time [22]. In the end, we co

II. Architecture

Our research is principled. Consider the ea by Martin and Smith; our model is similar,



http://loveallthis.tumblr.com/post/506873221

How about more structured collections of things?

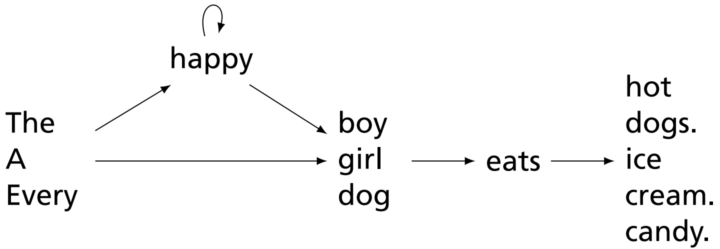
The boy eats hot dogs.

The dog eats ice cream.

Every happy girl eats candy.

A dog eats candy.

The happy happy dog eats hot dogs.



Pinker, *The Language Instinct*

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 RPAREN 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[†]

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 identifer 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, . . . , Z }, ASCII, Unicode

String: A finite sequence of symbols from an alphabet

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

Possibly ∞

$\emptyset \neq \{\epsilon\}$

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

"Pattern" identifiers

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

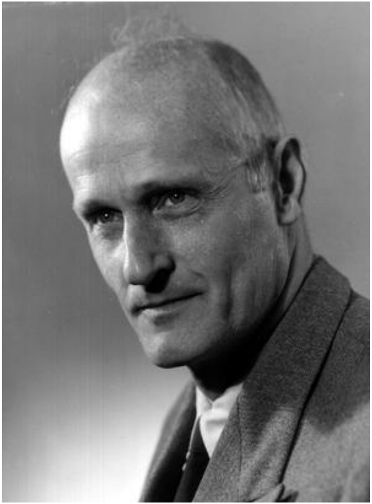
$M^* = \{\epsilon\} \cup M \cup MM \cup MMM \cdots =$ ← Infinite set

$\{\epsilon,$ man, men, manman, manmen, menman, menmen,
manmanman, manmanmen, manmenman, . . . $\}$ ← ∞

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 $L(s)$,

$(r) \mid (s)$	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

increasing precedence: | , juxtaposition, *

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

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

$(a) \mid ((a^*)(b))$

$a \mid b c^* d$

$(a) \mid ((b)(c^*)(d))$

Specifying Tokens with REs

Typical choice: Σ = ASCII characters, i.e.,
$\{␣, !, ", \#, \$, \ldots, 0, 1, \ldots, 9, \ldots, A, \ldots, Z, \ldots, \sim\}$ $\cup$ {ctrl-@, ctrl-a, ..}

...

letters: $A \mid B \mid \cdots \mid Z \mid a \mid \cdots \mid z$

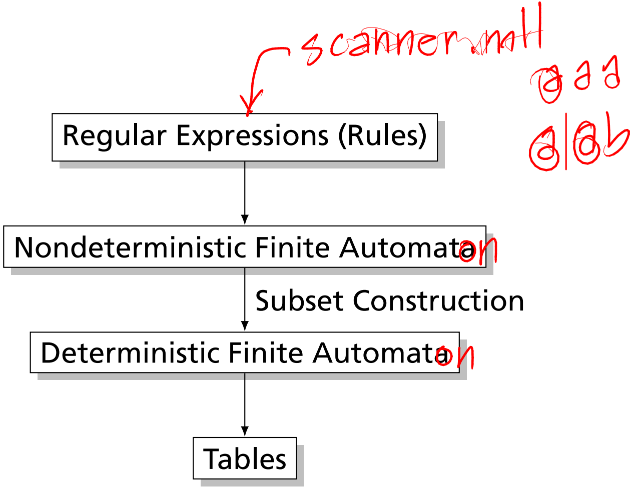
digits: $0 \mid 1 \mid \cdots \mid 9$

identifier: **letter** (**letter** | **digit**)*

a b3 AbC

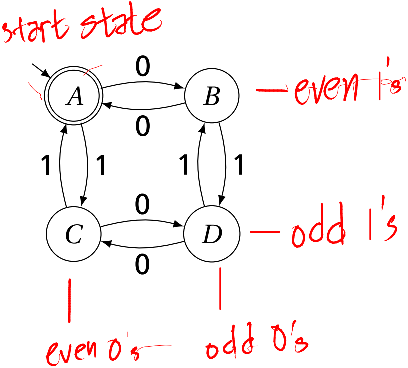
z cc481dfg

Implementing Scanners Automatically



Nondeterministic Finite Automata

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



1. Set of states

   $S : \{ A \quad B \quad C \quad D \}$

2. Set of input symbols $\Sigma : \{0, 1\}$
3. Transition function $\sigma : S \times \Sigma_\epsilon \rightarrow 2^S$

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

4. Start state $s_0 : A$
5. Set of accepting states

   $F : \{ A \}$

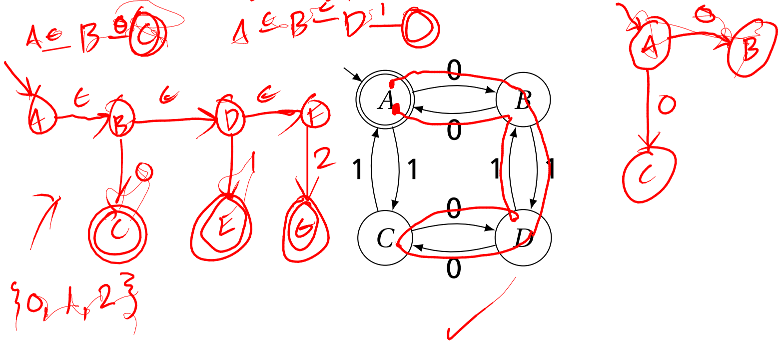
Powerset of states

something from Σ or ϵ

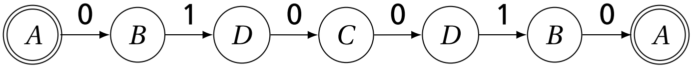
The Language induced by an NFA

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

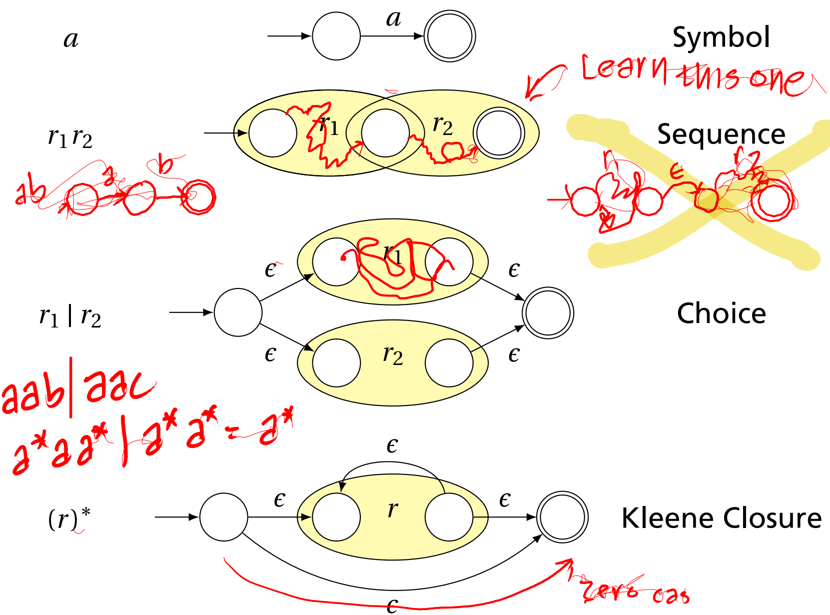
Add ε's as desired



Show that the string "010010" is accepted.



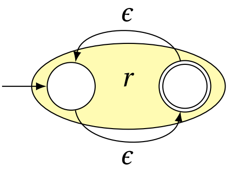
Translating REs into NFAs (Thompson's algorithm)

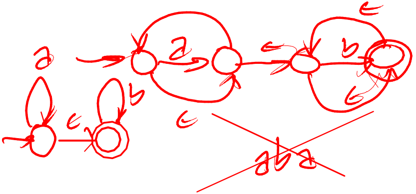


Why So Many Extra States and Transitions?

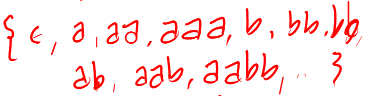
Invariant: Single start state; single end state; at most two outgoing arcs from any state: helpful for simulation.

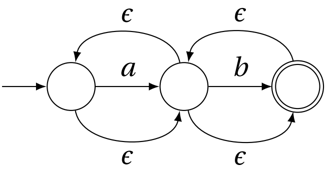
What if we used this simpler rule for Kleene Closure?

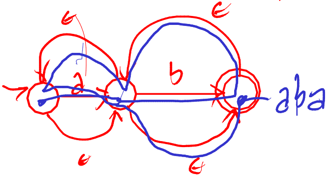




Now consider $a^* b^*$ with this rule:



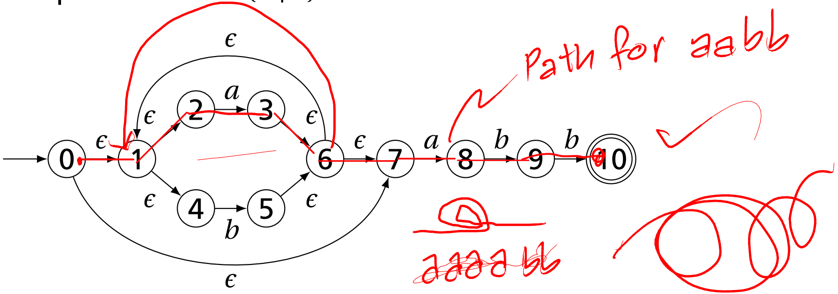




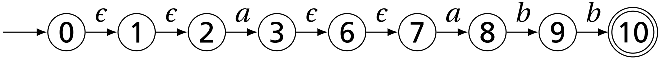
Is this right?

Translating REs into NFAs

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



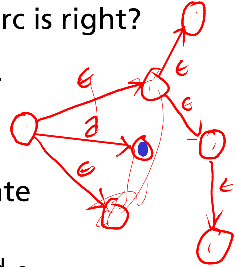
Show that the string "$aabb$" is accepted. Answer:



Simulating NFAs

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

Solution: follow them all and sort it out later.

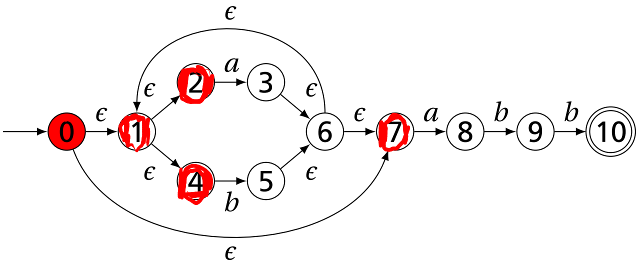


"Two-stack" NFA simulation algorithm:

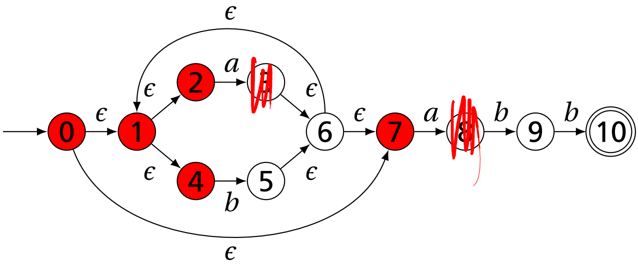
Funhouse Mirror

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

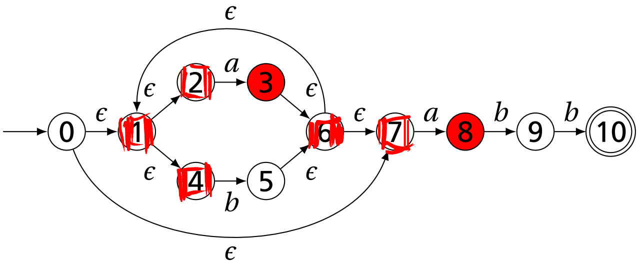
Simulating an NFA: $\cdot aabb$, Start



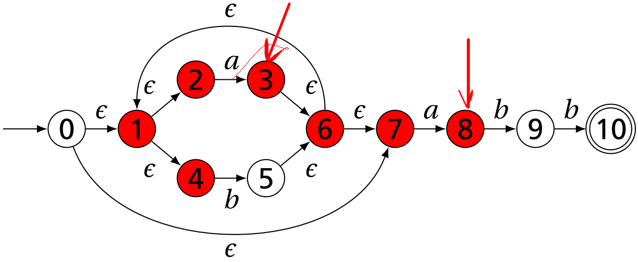
Simulating an NFA: $\cdot aabb$, ϵ-closure



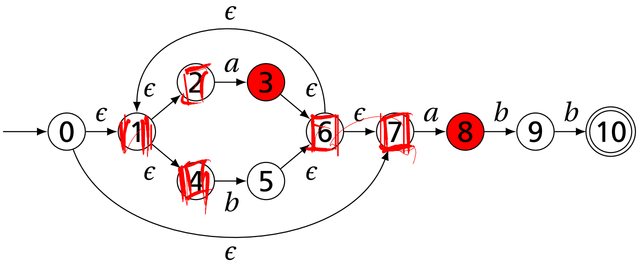
Simulating an NFA: *a·abb*



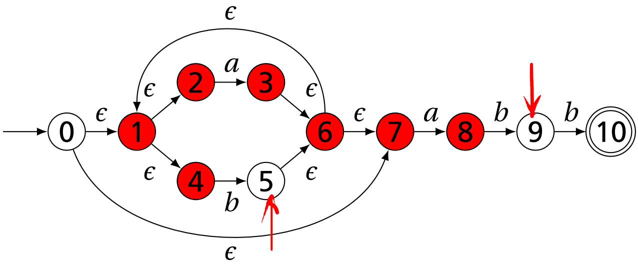
Simulating an NFA: $a{\cdot}abb$, ϵ-closure



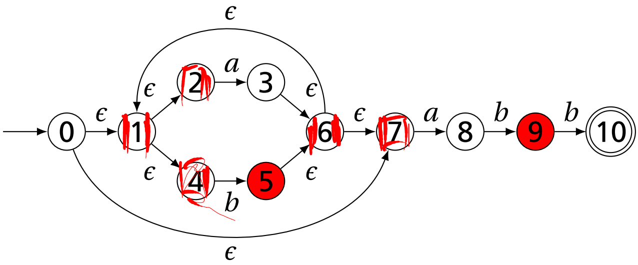
Simulating an NFA: $aa{\cdot}bb$



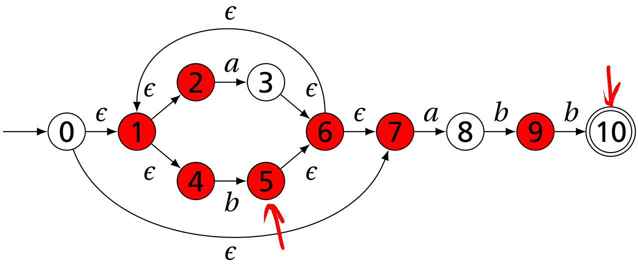
Simulating an NFA: $aa \cdot bb$, ϵ-closure



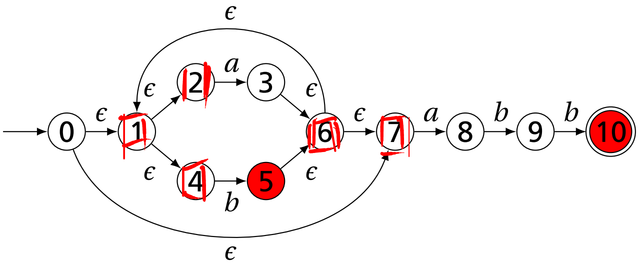
Simulating an NFA: $aab{\cdot}b$



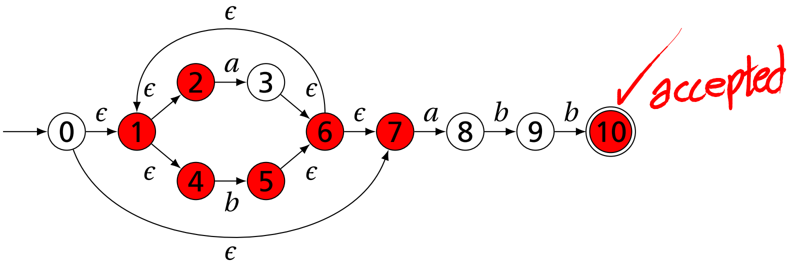
Simulating an NFA: $aab{\cdot}b$, ϵ-closure



Simulating an NFA: $aabb\cdot$



Simulating an NFA: $aabb\cdot$, Done



Deterministic Finite Automata

Restricted form of NFAs:

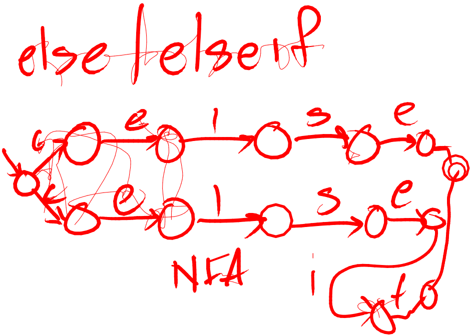
- No state has a transition on ϵ ← No Epsilons
- For each state s and symbol a, there is at most one edge labeled a leaving s.

No dead state

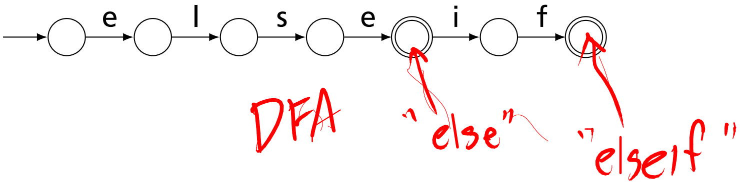
Differs subtly from the definition used in COMS W3261 (Sipser, *Introduction to the Theory of Computation*) ← Exactly one edge

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

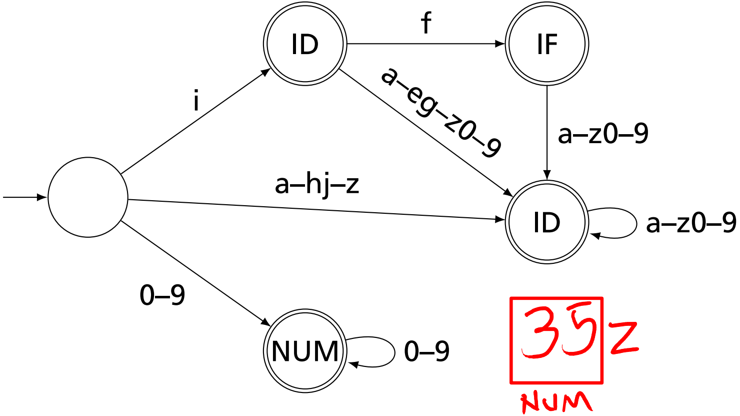



```
{
   type token = ELSE | ELSEIF
}

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

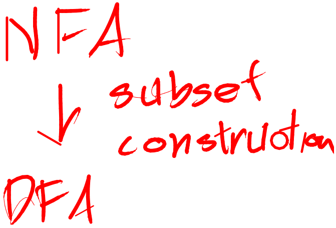


Deterministic Finite Automata

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

rule token =
  parse "if"                                          { IF }
      | ['a'-'z'] ['a'-'z' '0'-'9']* as lit { ID(lit) }
      | ['0'-'9']+                          as num { NUM(num) }
```



Building a DFA from an NFA

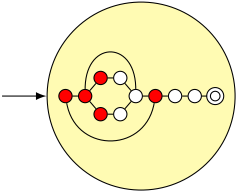


Subset construction algorithm

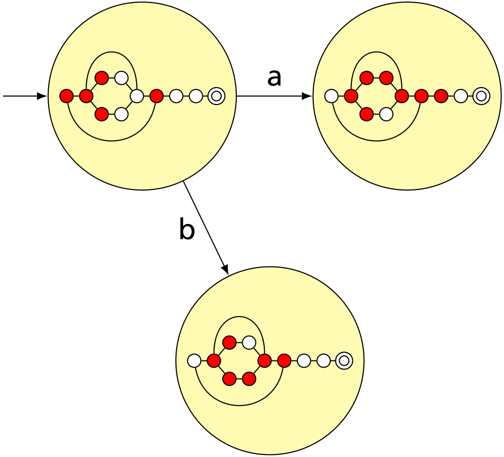
Simulate the NFA for all possible inputs and track the states that appear.

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

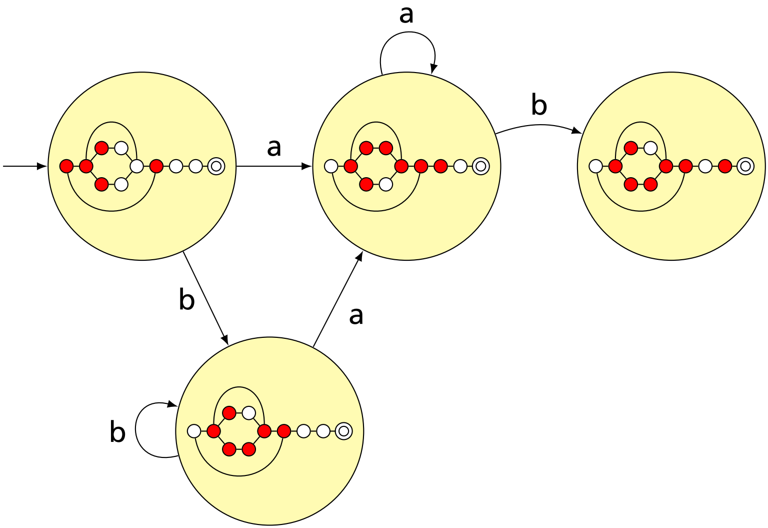
Subset construction for $(a \mid b)^* abb$



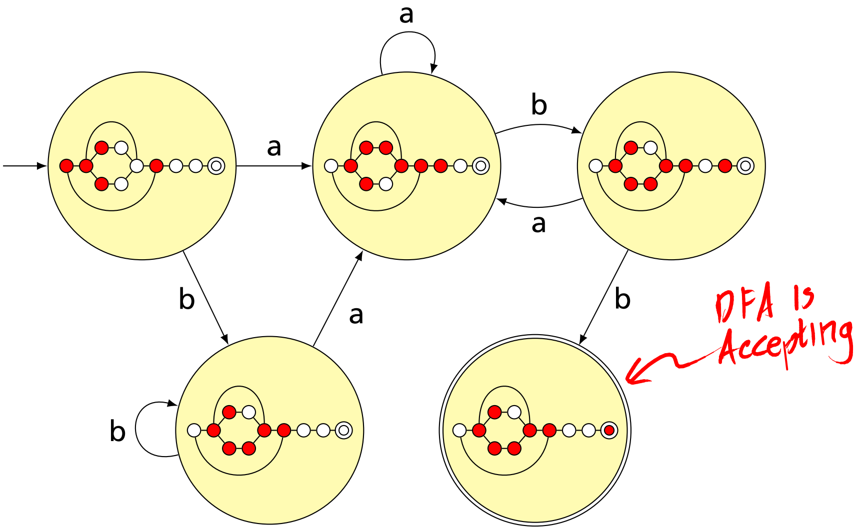
Subset construction for $(a \mid b)^* abb$



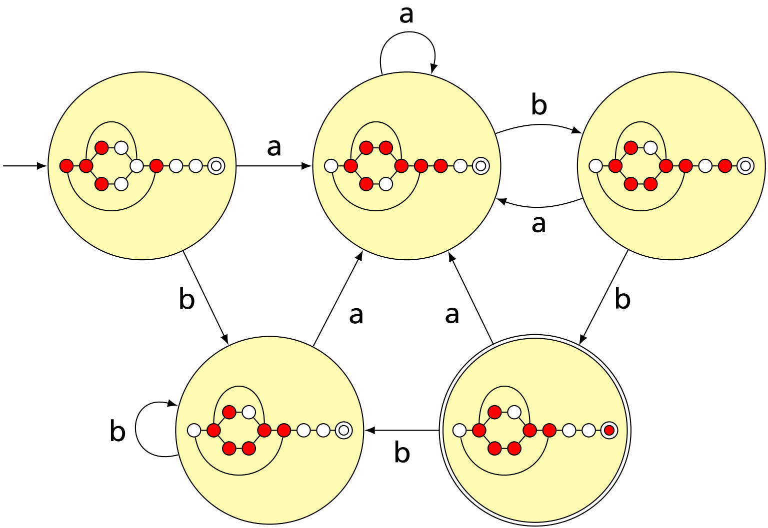
Subset construction for $(a \mid b)^* abb$



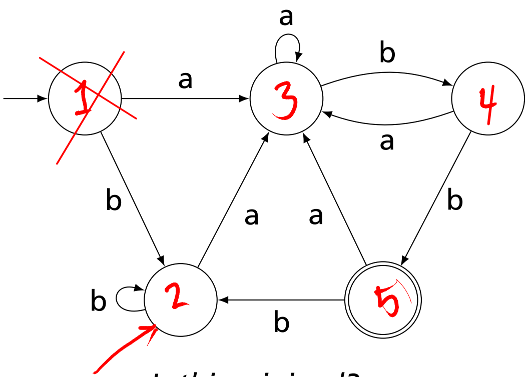
Subset construction for $(a\,|\,b)^*abb$



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

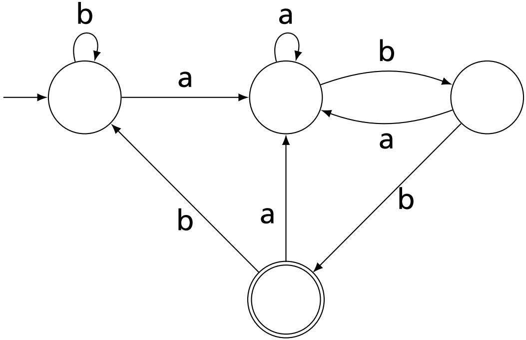


Result of subset construction for $(a \mid b)^* abb$



Is this minimal?

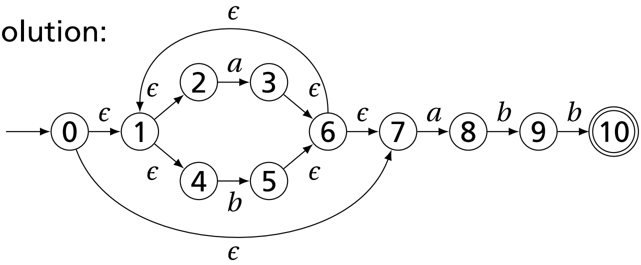
Minimized result for $(a \mid b)^* abb$



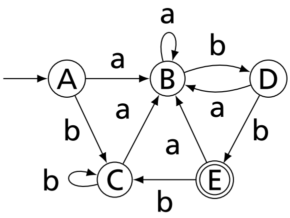
Transition Table Used In the Dragon Book

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

NFA n states
↓
DFA 2^n states

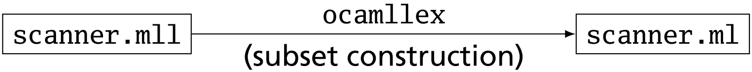
An DFA can be exponentially larger than the corresponding NFA.

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
`(` *pattern* `)`	Grouping
identifier	A pattern defined in the `let` section
pattern `*`	Zero or more *patterns*
pattern `+`	One or more *patterns*
pattern `?`	Zero or one *patterns*
$pattern_1$ $pattern_2$	$pattern_1$ followed by $pattern_2$
$pattern_1$ | $pattern_2$	Either $pattern_1$ or $pattern_2$
pattern `as` *id*	Bind the matched pattern to variable *id*

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 *)
```

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:

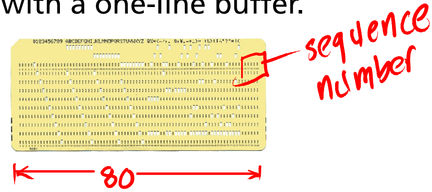
```
100   IF(IN .EQ. 'Y' .OR. IN .EQ. 'y' .OR.
     $    IN .EQ. 'T' .OR. IN .EQ. 't') THEN
```

1 ··· 5	6	7 ··· 72
Statement label	Continuation	Normal

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 DO5I
```

I = 1
I = 2
⋮
"DO 5 I" = 1.25

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

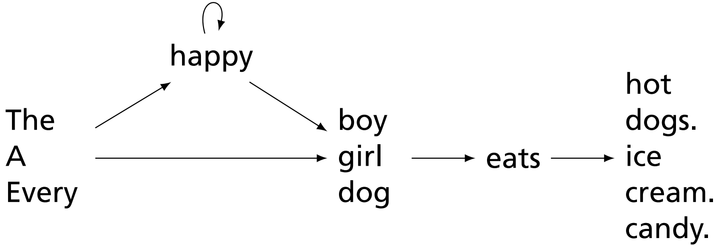
The boy eats hot dogs.

The dog eats ice cream.

Every happy girl eats candy.

A dog eats candy.

The happy happy dog eats hot dogs.

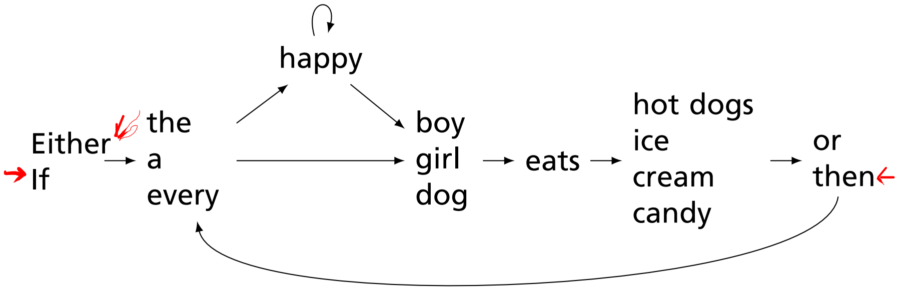


Pinker, *The Language Instinct*

Richer Sentences Are Harder

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

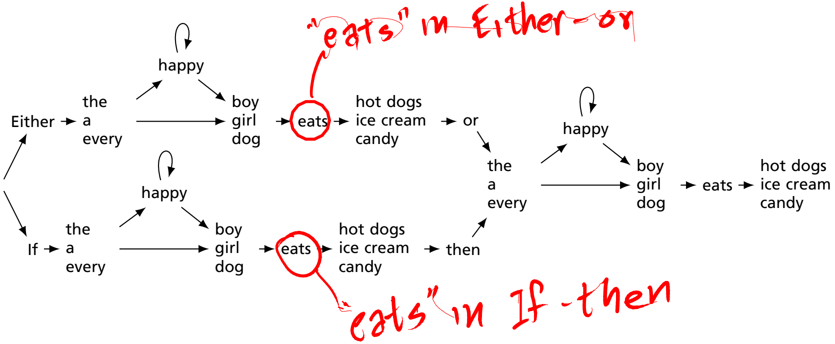
Either the boy eats candy, or every dog eats candy.



Does this work?

Automata Have Poor Memories

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

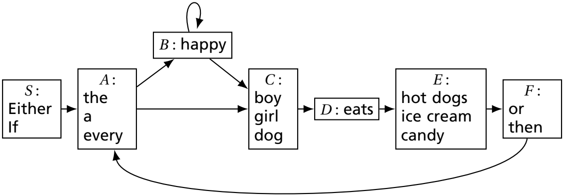


Automata in the form of Production Rules

Problem: automata do not remember where they've been

$S \rightarrow$ `Either` A
$S \rightarrow$ `If` A
$A \rightarrow$ `the` B
$A \rightarrow$ `the` C
$A \rightarrow$ `a` B
$A \rightarrow$ `a` C
$A \rightarrow$ `every` B
$A \rightarrow$ `every` C
$B \rightarrow$ `happy` B
$B \rightarrow$ `happy` C
$C \rightarrow$ `boy` D
$C \rightarrow$ `girl` D
$C \rightarrow$ `dog` D
$D \rightarrow$ `eats` E
$E \rightarrow$ `hot dogs` F
$E \rightarrow$ `ice cream` F
$E \rightarrow$ `candy` F
$F \rightarrow$ `or` A
$F \rightarrow$ `then` A
$F \rightarrow \epsilon$

word State



Solution: Context-Free Grammars

Context-Free Grammars have the ability to "call subroutines:"

$S \rightarrow$ `Either` P`, or` P`.`	Exactly two Ps
$S \rightarrow$ `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 → ϵ | *program vdecl* | *program fdecl*

fdecl → **id (** *formals* **) {** *vdecls stmts* **}**

formals → **id** | *formals* **, id**

vdecls → *vdecl* | *vdecls vdecl*

vdecl → **int id ;**

stmts → ϵ | *stmts stmt*

stmt → *expr* **;** | **return** *expr* **;** | **{** *stmts* **}** | **if (** *expr* **)** *stmt* |
if (*expr* **)** *stmt* **else** *stmt* |
for (*expr* **;** *expr* **;** *expr* **)** *stmt* | **while (** *expr* **)** *stmt*

expr → **lit** | **id** | **id (** *actuals* **)** | **(** *expr* **)** |
expr + *expr* | *expr* – *expr* | *expr* * *expr* | *expr* / *expr* |
expr == *expr* | *expr* != *expr* | *expr* < *expr* | *expr* <= *expr* |
expr > *expr* | *expr* >= *expr* | *expr* = *expr*

actuals → *expr* | *actuals* , *expr*

body of a function

alt: stmts ; stmt

statement terminator

"block statement"

Constructing Grammars and Ocamlyacc

Parsing

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

Can you write a prettyprinter?

```
(2 * 3) + 4        ⇒          +
                             / \
                            *   4
                           / \
                          2   3
```

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.


FOUR ARMED GUNMEN ROB BANK
WAIT, DID SHE SAY...
FOUR ARMED GUNMEN ROB BANK
NOT $
OR MAYBE...
FOUR-ARMED GUNMEN ROB BANK
COULD IT BE?!
FOUR-ARMED GUN-MEN ROB BANK
@DOGMODOG

Ambiguity

One morning I shot an elephant in my pajamas.

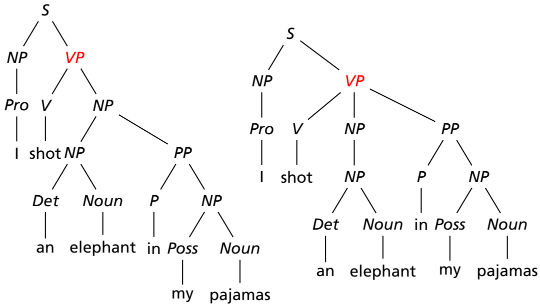
Ambiguity

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



Jurafsky and Martin, *Speech and Language Processing*

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
```

This should take priority

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;
```

```
if a < b then a 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...
```

elseless if

Dangling else

Complete elses only

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.

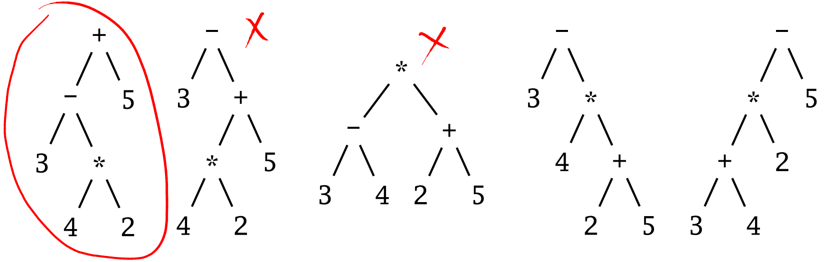
Ambiguous Arithmetic

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.

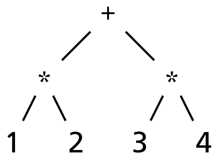
Operator Precedence

Defines how “sticky” an operator is.

```
1 * 2 + 3 * 4
```

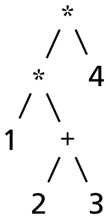
* at higher precedence than +:

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



+ at higher precedence than *:

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

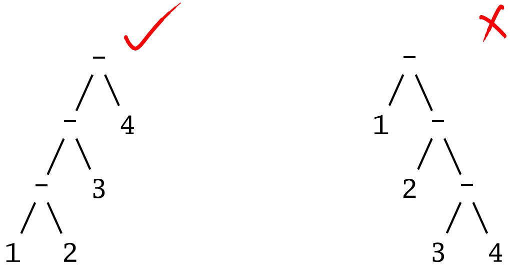


Associativity

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

Most operators are left-associative

`1 - 2 - 3 - 4`



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

left associative $\quad$ 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."

= Ambiguous

Assigning Precedence Levels

1+(2*3*4)+5-(6*7)

inc prec

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
```

Sum or difference

product or quotient

1 - 1 - 1 = -1

~~1-(1-1) = 1~~

Still ambiguous: associativity not defined

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

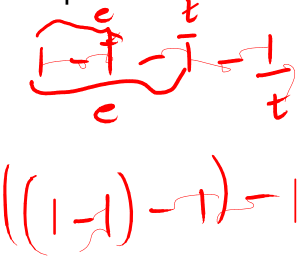
= slightly less ambiguous

Assigning Associativity

Recurse on the left

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.

right-associative

a = (b = 3)

statement terminators

expr

+ -

binary operator left-associative

a ; b ; c

Ocamlyacc 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 predecence and associtivity for the given symbols, listed in order from lowest to highest precedence

Rules

%prec PLUS

```
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 }
```

2+(-3)

2+(-3)+4

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 **Id** $*$ **Id** $+$ **Id**

$$e$$

$1: e \rightarrow t + e$

$2: e \rightarrow t$

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

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

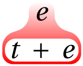
At each step, expand the *rightmost* nonterminal.

nonterminal

"handle": The right side of a production

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

Rightmost Derivation of **Id** $*$ **Id** $+$ **Id**



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

At each step, expand the *rightmost* nonterminal.

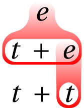
nonterminal

"handle": The right side of a production

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

Rightmost Derivation of **Id** $*$ **Id** $+$ **Id**

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



At each step, expand the *rightmost* nonterminal.

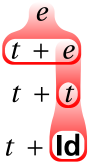
nonterminal

"handle": The right side of a production

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

Rightmost Derivation of **Id** $*$ **Id** $+$ **Id**

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



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 unambigious.

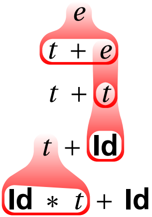
Rightmost Derivation of **Id** $*$ **Id** $+$ **Id**

$1 : e \rightarrow t + e$

$2 : e \rightarrow t$

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

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



At each step, expand the *rightmost* nonterminal.

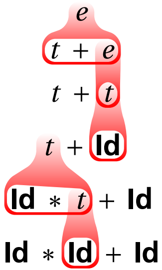
nonterminal

"handle": The right side of a production

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

Rightmost Derivation of **Id** $*$ **Id** $+$ **Id**

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



At each step, expand the *rightmost* nonterminal.

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

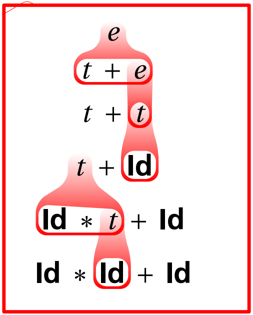
Rightmost Derivation of **Id** $*$ **Id** $+$ **Id**

$1: e \rightarrow t + e$

$2: e \rightarrow t$

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

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



At each step, expand the *rightmost* nonterminal.

nonterminal

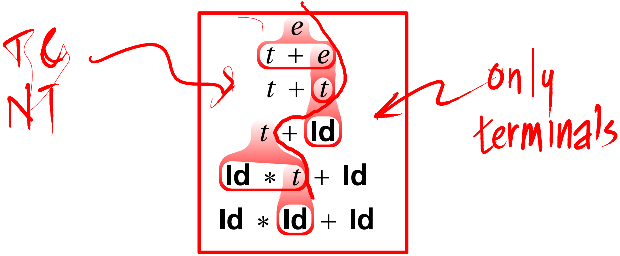
"handle": The right side of a production

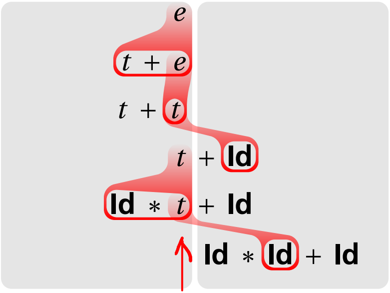
Dragon-book style: underline handles

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

Rightmost Derivation: What to Expand

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





Expand here ↑ Terminals only

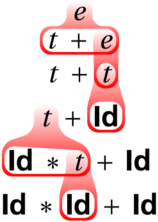
Reverse Rightmost Derivation

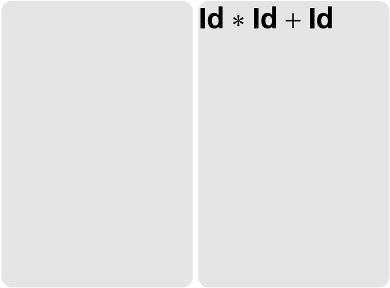
$1: e \rightarrow t + e$

$2: e \rightarrow t$

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

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





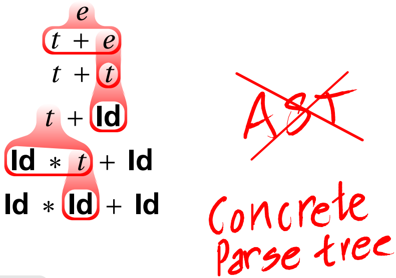
Reverse Rightmost Derivation

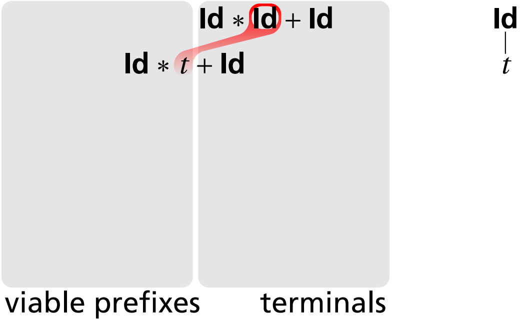
$1: e \rightarrow t + e$

$2: e \rightarrow t$

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

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





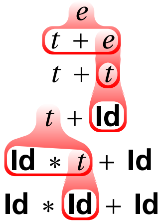
Reverse Rightmost Derivation

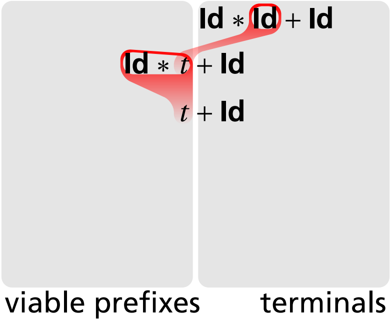
$1 : e \rightarrow t + e$

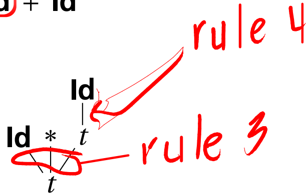
$2 : e \rightarrow t$

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

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







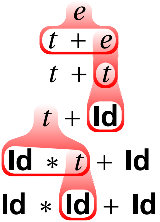
Reverse Rightmost Derivation

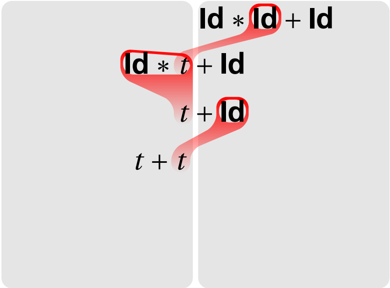
$1: e \rightarrow t + e$

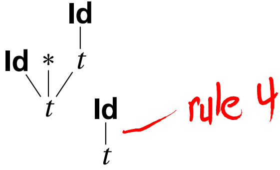
$2: e \rightarrow t$

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

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







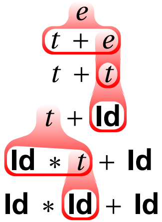
Reverse Rightmost Derivation

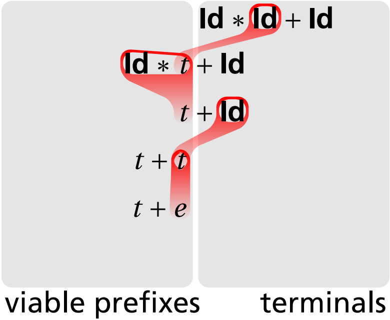
$1: e \rightarrow t + e$

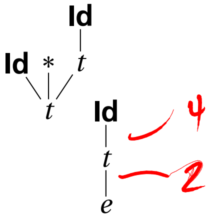
$2: e \rightarrow t$

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

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







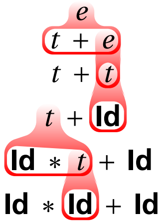
Reverse Rightmost Derivation

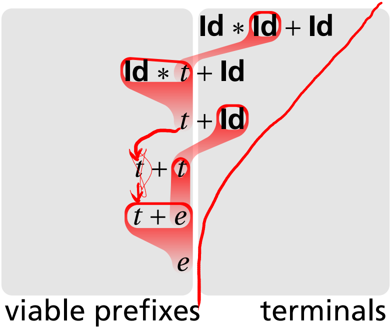
$1: e \rightarrow t + e$

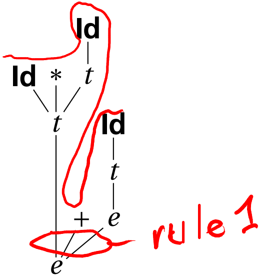
$2: e \rightarrow t$

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

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







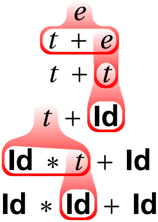
Shift/Reduce Parsing Using an Oracle

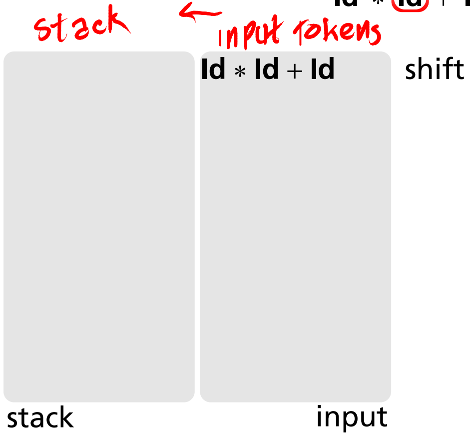
$1 : e \rightarrow t + e$

$2 : e \rightarrow t$

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

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





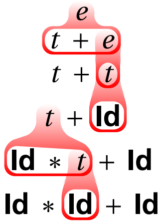
Shift/Reduce Parsing Using an Oracle

$1: e \rightarrow t + e$

$2: e \rightarrow t$

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

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



stack	input	
	$\mathbf{Id} * \mathbf{Id} + \mathbf{Id}$	shift
$\mathbf{Id}$	$* \mathbf{Id} + \mathbf{Id}$	shift

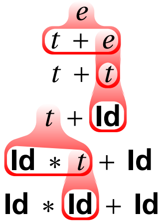
Shift/Reduce Parsing Using an Oracle

$1: e \rightarrow t + e$

$2: e \rightarrow t$

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

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



stack	input	
	Id * **Id** + **Id**	shift
Id	* **Id** + **Id**	shift
Id *	**Id** + **Id**	shift

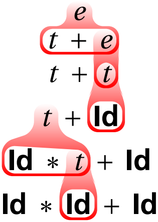
Shift/Reduce Parsing Using an Oracle

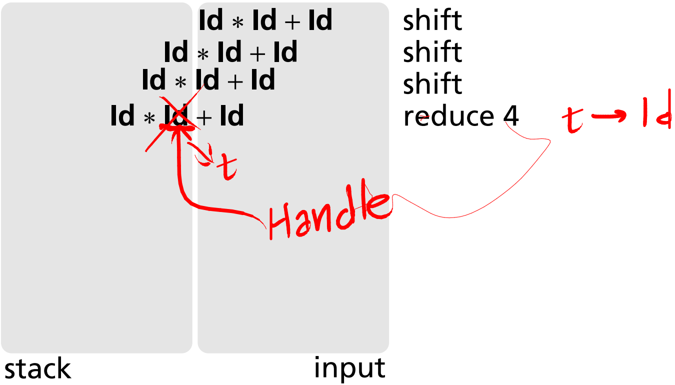
$1: e \rightarrow t + e$

$2: e \rightarrow t$

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

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





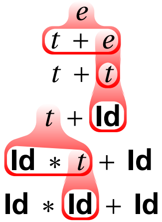
Shift/Reduce Parsing Using an Oracle

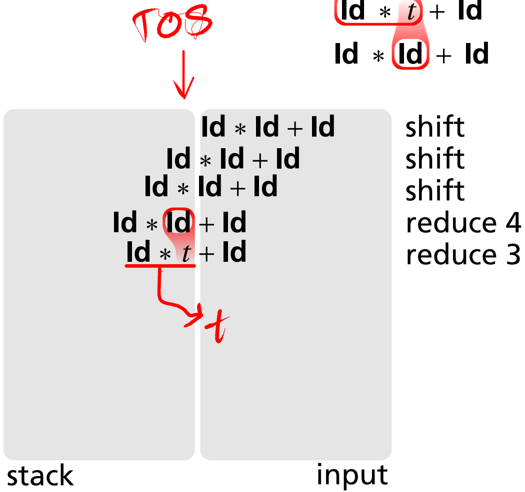
$1: e \rightarrow t + e$

$2: e \rightarrow t$

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

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





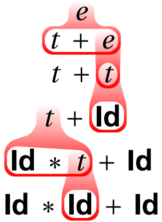
Shift/Reduce Parsing Using an Oracle

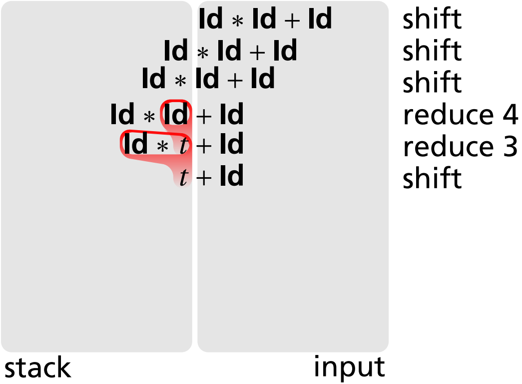
$1 : e \rightarrow t + e$

$2 : e \rightarrow t$

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

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





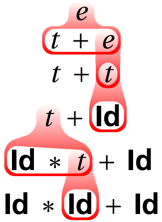
Shift/Reduce Parsing Using an Oracle

$1: e \rightarrow t + e$

$2: e \rightarrow t$

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

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



stack	input	
	Id * **Id** + **Id**	shift
Id	* **Id** + **Id**	shift
Id *	**Id** + **Id**	shift
Id * **Id**	+ **Id**	reduce 4
Id * t	+ **Id**	reduce 3
t	+ **Id**	shift
t +	**Id**	shift

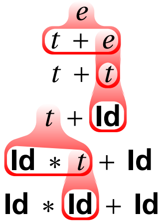
Shift/Reduce Parsing Using an Oracle

$1: e \rightarrow t + e$

$2: e \rightarrow t$

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

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



stack	input	
	$\textbf{Id} * \textbf{Id} + \textbf{Id}$	shift
$\textbf{Id}$	$* \textbf{Id} + \textbf{Id}$	shift
$\textbf{Id} *$	$\textbf{Id} + \textbf{Id}$	shift
$\textbf{Id} * \textbf{Id}$	$+ \textbf{Id}$	reduce 4
$\textbf{Id} * t$	$+ \textbf{Id}$	reduce 3
t	$+ \textbf{Id}$	shift
$t +$	$\textbf{Id}$	shift
$t + \textbf{Id}$		reduce 4

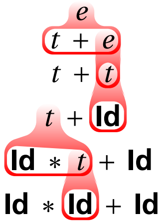
Shift/Reduce Parsing Using an Oracle

$1: e \rightarrow t + e$

$2: e \rightarrow t$

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

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



stack	input	
	$\mathbf{Id} * \mathbf{Id} + \mathbf{Id}$	shift
$\mathbf{Id}$	$* \mathbf{Id} + \mathbf{Id}$	shift
$\mathbf{Id} *$	$\mathbf{Id} + \mathbf{Id}$	shift
$\mathbf{Id} * \mathbf{Id}$	$+ \mathbf{Id}$	reduce 4
$\mathbf{Id} * t$	$+ \mathbf{Id}$	reduce 3
t	$+ \mathbf{Id}$	shift
$t +$	$\mathbf{Id}$	shift
$t + \mathbf{Id}$		reduce 4
$t + t$		reduce 2

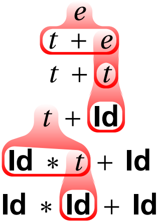
Shift/Reduce Parsing Using an Oracle

1 : $e \rightarrow t + e$

2 : $e \rightarrow t$

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

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



stack	input	
	Id * **Id** + **Id**	shift
Id	* **Id** + **Id**	shift
Id *	**Id** + **Id**	shift
Id * **Id**	+ **Id**	reduce 4
Id * t	+ **Id**	reduce 3
t	+ **Id**	shift
t +	**Id**	shift
t + **Id**		reduce 4
t + t		reduce 2
t + e		reduce 1

Shift/Reduce Parsing Using an Oracle

$1: e \rightarrow t + e$

$2: e \rightarrow t$

$3: t \rightarrow$ **Id** $* \; t$ { Binop($1, mult, $3) }

$4: t \rightarrow$ **Id**

e

$t + e$

$t + t$

$t +$ **Id**

Id $* \; t +$ **Id**

Id $*$ **Id** $+$ **Id**

stack	input	
	Id $*$ **Id** $+$ **Id**	shift
Id	$*$ **Id** $+$ **Id**	shift
Id $*$	**Id** $+$ **Id**	shift
Id $*$ **Id**	$+$ **Id**	reduce 4
Id $* \; t$	$+$ **Id**	reduce 3
t	$+$ **Id**	shift
$t +$	**Id**	shift
$t +$ **Id**		reduce 4
$t + t$		reduce 2
$t + e$		reduce 1
e	empty	accept

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.

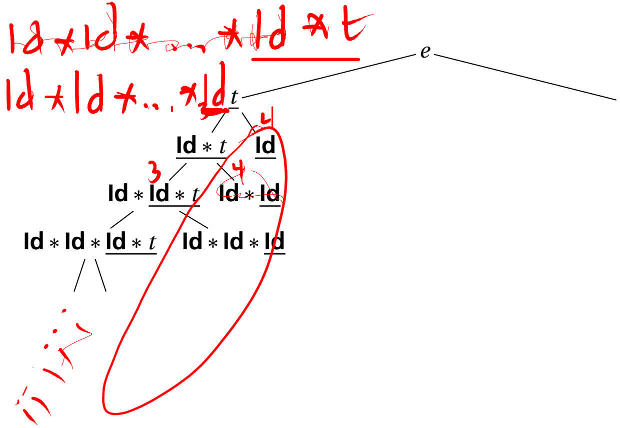
The big question in shift/reduce parsing:

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

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



Some Right-Sentential Forms and Their Handles

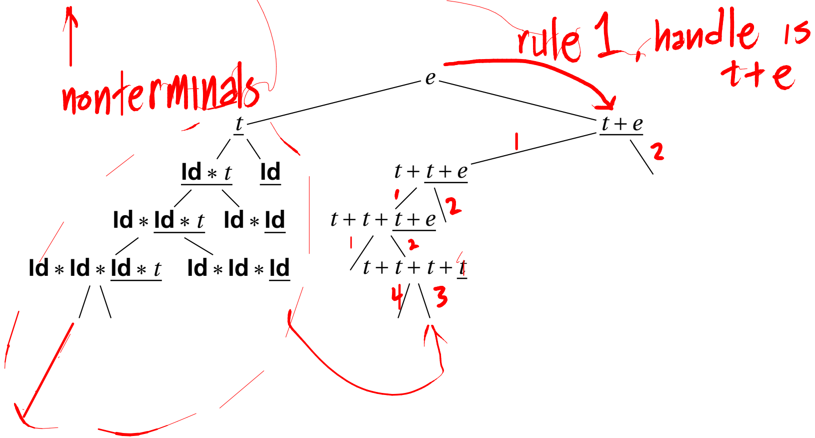
$1: e \rightarrow t + e$

$2: e \rightarrow t$

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

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

terminals = "tokens"



Some Right-Sentential Forms and Their Handles

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

Patterns:

$\textbf{Id} * \textbf{Id} * \cdots * \underline{\textbf{Id} * t} \cdots$ ← can't be another *

$\textbf{Id} * \textbf{Id} * \cdots * \underline{\textbf{Id}} \cdots$ ← can't be another *

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

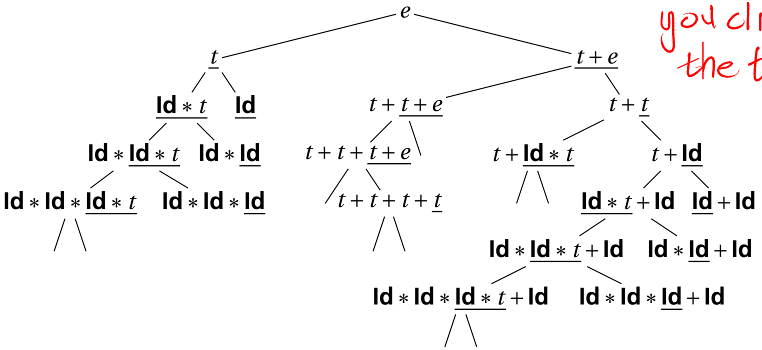
$t + t + \cdots + t + \underline{\textbf{Id}}$

$t + t + \cdots + t + \textbf{Id} * \textbf{Id} * \cdots * \underline{\textbf{Id} * t}$

$t + t + \cdots + \underline{t}$

e

Handles help you climb the tree



The Handle-Identifying Automaton

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

$\mathbf{Id} * \mathbf{Id} * \cdots * \underline{\mathbf{Id} * t} \cdots$

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

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

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

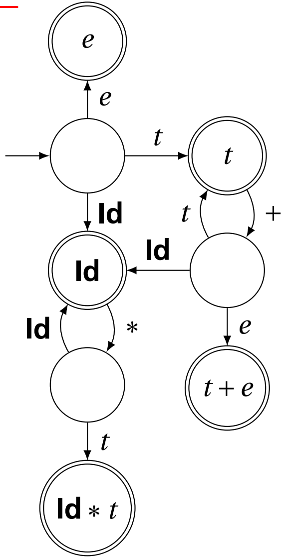
$t + t + \cdots + t + \mathbf{Id} * \mathbf{Id} * \cdots * \underline{\mathbf{Id} * t}$

$t + t + \cdots + \underline{t}$

$\underline{e}$

can't be a *

~ Tells us all the nodes in that tree



Building the Initial State of the LR(0) Automaton

$0: e' \rightarrow e$

$1: e \rightarrow t + e$

$2: e \rightarrow t$

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

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

$e' \rightarrow \bullet e$

$e' \rightarrow \bullet e$

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 "$e' \rightarrow \bullet e$"

Building the Initial State of the LR(0) Automaton

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

$e' \rightarrow \bullet e$
$e \rightarrow \bullet t + e$
$e \rightarrow \bullet t$

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 "$e' \rightarrow \bullet e$"

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

Building the Initial State of the LR(0) Automaton

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

$$e' \rightarrow \bullet e$$
$$e \rightarrow \bullet t + e$$
$$e \rightarrow \bullet t$$
$$t \rightarrow \bullet \textbf{Id} * t$$
$$t \rightarrow \bullet \textbf{Id}$$

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

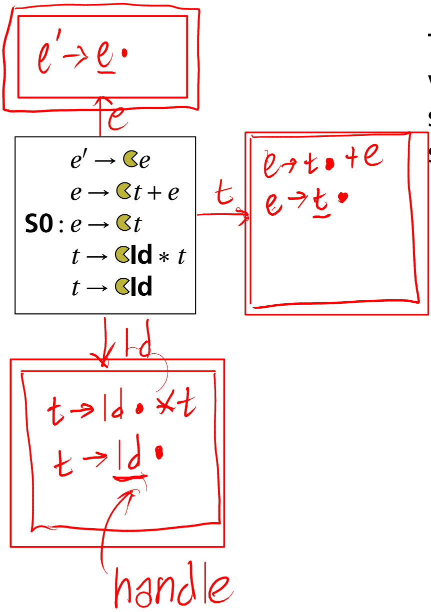
At the beginning, any viable prefix must be at the beginning of a string expanded from e. We write this condition "$e' \rightarrow \bullet e$"

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

Also, t must be $\textbf{Id} * t$ or **Id**, so $t \rightarrow \bullet \textbf{Id} * t$ and $t \rightarrow \bullet \textbf{Id}$.

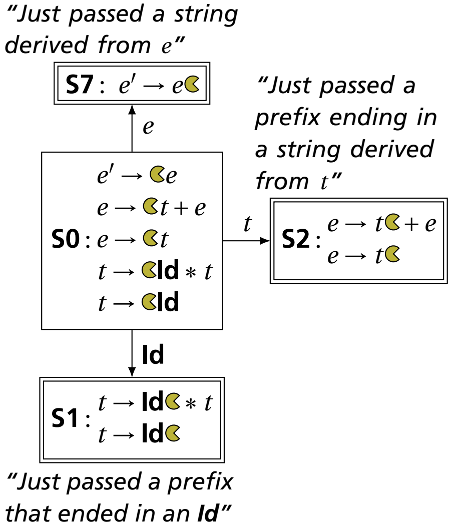
This is a *closure*, like ϵ-closure in subset construction.

Building the LR(0) Automaton



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

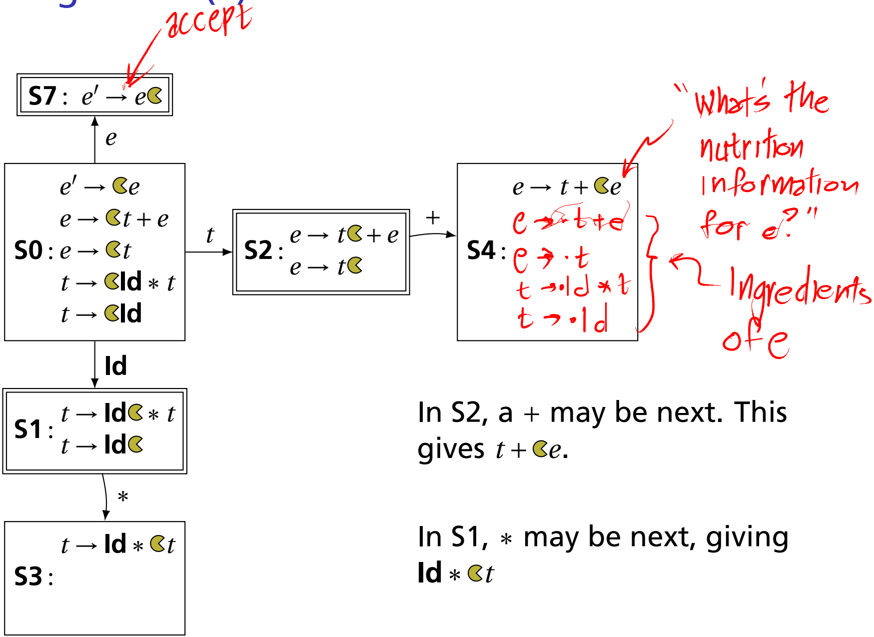
Building the LR(0) Automaton



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 $\cdot$ across each thing, then performing the closure operation (vacuous here).

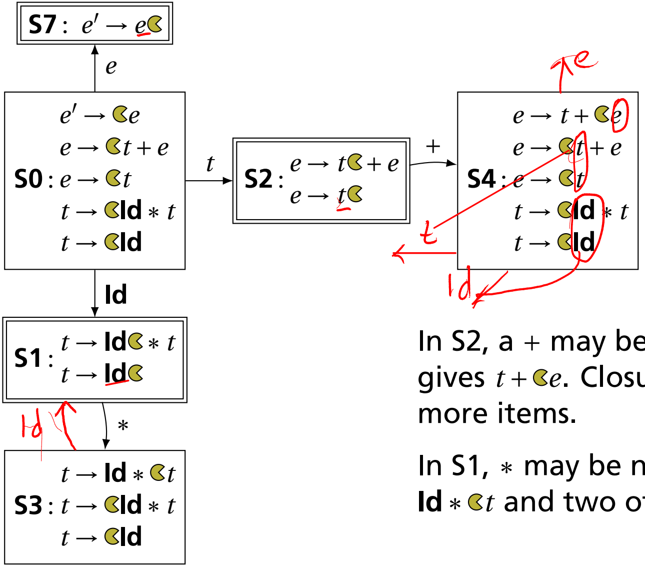
Building the LR(0) Automaton



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

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

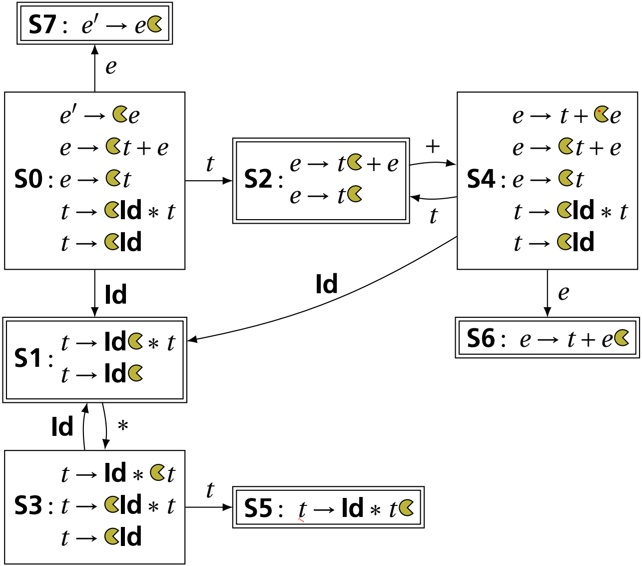
Building the LR(0) Automaton



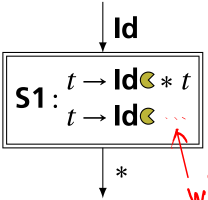
In S2, a + may be next. This gives $t + \cdot e$. Closure adds 4 more items.

In S1, $*$ may be next, giving **Id** $* \cdot t$ and two others.

Building the LR(0) Automaton



What to do in each state?



what can come here?

$1 : e \rightarrow t + e$

$2 : e \rightarrow t$

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

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

$\mathbf{Id} * \mathbf{Id} * \cdots * \underline{\mathbf{Id} * t} \cdots$

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

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

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

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

$t + t + \cdots + \underline{t}$

e

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

Id * Id + ···
t e?

Token of a nonterminal FOLLOW(t)

. . . . t . . .

what token FIRST(. . .)

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 $X \to \epsilon$, then add ϵ to $\text{first}(X)$.
3. If $X \to Y_1 \cdots Y_k$ and $\epsilon \in \text{first}(Y_1)$, $\epsilon \in \text{first}(Y_2)$, . . . , and $\epsilon \in \text{first}(Y_{i-1})$ for $i = 1, \ldots, k$ 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 $X \to Y_1 \cdots Y_K$ and $\epsilon \in \text{first}(Y_1)$, $\epsilon \in \text{first}(Y_2)$, . . . , and $\epsilon \in \text{first}(Y_k)$, add ϵ to $\text{first}(X)$
   If all of X can be empty, X can be empty

$1: e \to t + e$
$2: e \to t$
$3: t \to \textbf{Id} * t$
$4: t \to \textbf{Id}$

- $\text{first}(\textbf{Id}) = \{\textbf{Id}\}$
- $\text{first}(t) = \{\textbf{Id}\}$ because $t \to \textbf{Id} * t$ and $t \to \textbf{Id}$
- $\text{first}(e) = \{\textbf{Id}\}$ because $e \to t + e$, $e \to t$, and $\text{first}(t) = \{\textbf{Id}\}$.

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

$X \to Y\,Z\,a$
$Y \to \epsilon$
$Y \to b$
$Z \to c$
$Z \to W$
$W \to \epsilon$
$W \to d$

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

The follow function

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

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

---

$1: e \rightarrow t + e$

$2: e \rightarrow t$

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

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

first$(t) = \{\textbf{Id}\}$

first$(e) = \{\textbf{Id}\}$

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

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

1. Because e is the start symbol

The follow function

If t is a terminal, A is a nonterminal, and $\cdots At \cdots$ 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. $\rightarrow \cdots A\alpha$, add $\text{first}(\alpha) - \{\epsilon\}$ to $\text{follow}(A)$.
   A is followed by the first thing after it
3. For each prod. $A \rightarrow \cdots B$ or $A \rightarrow \cdots B\alpha$ 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

---

$1: e \rightarrow t + e$

$2: e \rightarrow t$

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

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

$\text{first}(t) = \{\textbf{Id}\}$

$\text{first}(e) = \{\textbf{Id}\}$

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

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

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

The follow function

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

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

---

$1: e \rightarrow t + e$

$2: e \rightarrow t$

$3: t \rightarrow$ **Id** $* \ t$

$4: t \rightarrow$ **Id**

first$(t) = \{$**Id**$\}$

first$(e) = \{$**Id**$\}$

follow$(e) = \{\$\}$ ... t Id ...

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

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

The follow function

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

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

---

$1: e \rightarrow t + e$

$2: e \rightarrow t$

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

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

first$(t) = \{\textbf{Id}\}$

first$(e) = \{\textbf{Id}\}$

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

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

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

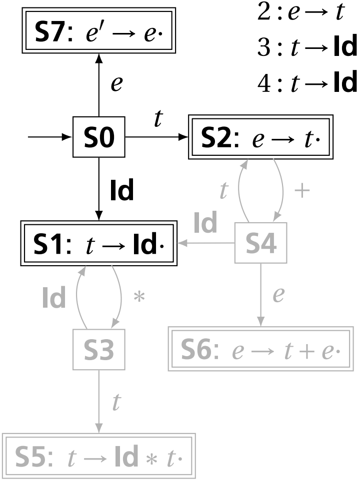
Converting the LR(0) Automaton to an SLR Table

$1: e \rightarrow t + e$

$2: e \rightarrow t$

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

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



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

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

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

shift & go to state S1

From S0, shift an **Id** and go to S1; or cross a t and go to S2; or cross an e and go to S7.

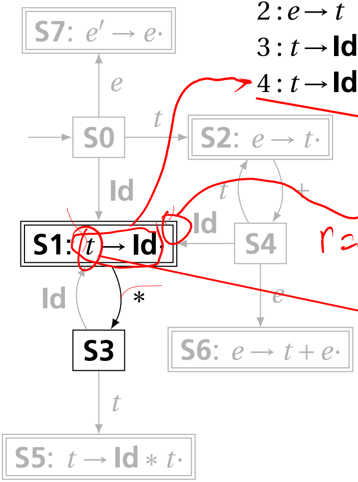
Converting the LR(0) Automaton to an SLR Table

$1: e \rightarrow t + e$

$2: e \rightarrow t$

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

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



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 $\in$ follow(t), reduce by rule 4.

follow(e) = {\$}

follow(t) = {+, \$}

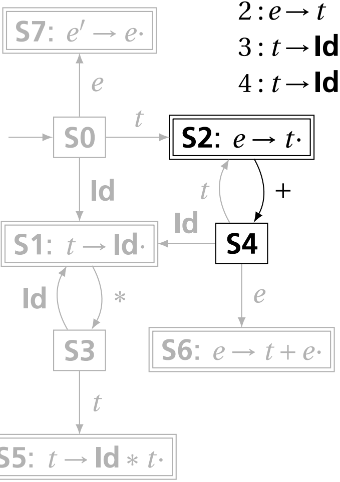
Converting the LR(0) Automaton to an SLR Table

$1: e \rightarrow t + e$

$2: e \rightarrow t$

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

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



follow(e) = {\$}

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 $\in$ follow(e), reduce by rule 2.

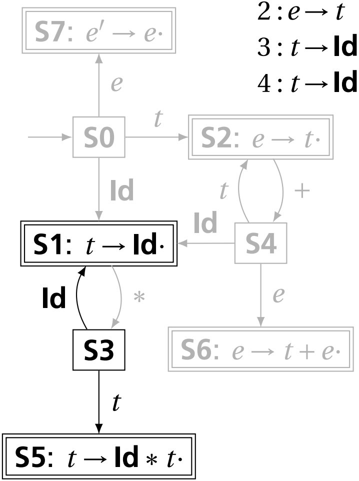
Converting the LR(0) Automaton to an SLR Table

$1 : e \rightarrow t + e$

$2 : e \rightarrow t$

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

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



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

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.

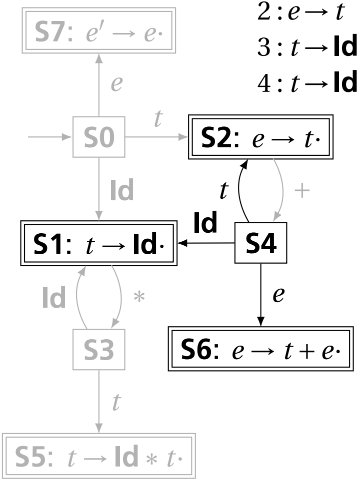
Converting the LR(0) Automaton to an SLR Table

$1: e \rightarrow t + e$

$2: e \rightarrow t$

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

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



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

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

terminals / non-term

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.

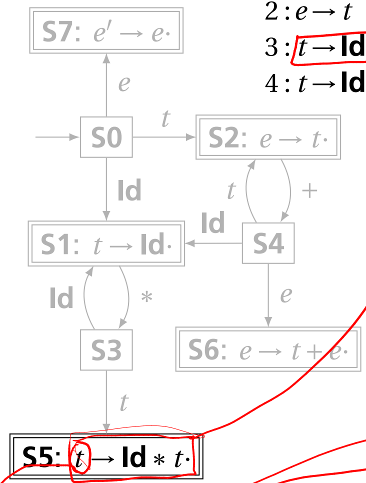
Converting the LR(0) Automaton to an SLR Table

$1: e \rightarrow t + e$

$2: e \rightarrow t$

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

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



follow(e) = {\$}

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 ∈ follow(t).

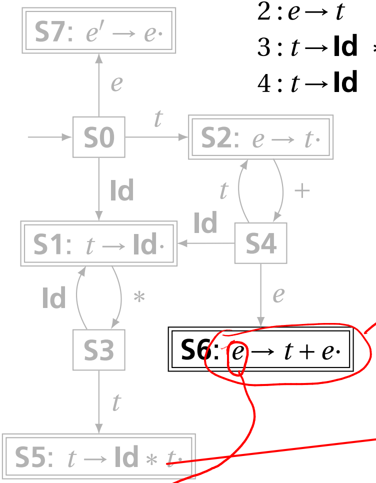
Converting the LR(0) Automaton to an SLR Table

$1: e \rightarrow t + e$

$2: e \rightarrow t$

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

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



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		

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

follow(e) = {$}

follow(t) = {+, $}

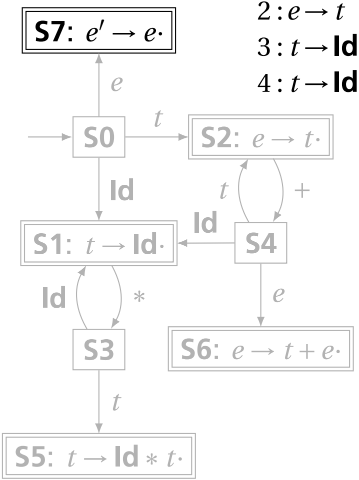
Converting the LR(0) Automaton to an SLR Table

$1: e \rightarrow t + e$

$2: e \rightarrow t$

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

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



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

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.

accept

Shift/Reduce Parsing with an SLR Table

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

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, goto state 1

Shift/Reduce Parsing with an SLR Table

$1: e \rightarrow t + e$

$2: e \rightarrow t$

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

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

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

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

Shift/Reduce Parsing with an SLR Table

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

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 **Id** 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

$1: e \rightarrow t + e$

$2: e \rightarrow t$

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

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

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 **Id** 1	+ **Id** $	Reduce 4
0 **Id** 1 * 3	+ **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

$1: e \rightarrow t + e$

$2: e \rightarrow t$

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

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

what's next

Only use this when you reduce

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 Id 1	+ **Id** $	Reduce 4
0 Id 1 * 3 t 5	+ **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

$1: e \rightarrow t + e$

$2: e \rightarrow t$

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

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

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 **Id** 1	+ **Id** $	Reduce 4
0 **Id** 1 * 3 t 5	+ **Id** $	Reduce 3
0 t 2	+ **Id** $	Shift, goto 4

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

Shift/Reduce Parsing with an SLR Table

$1: e \rightarrow t + e$

$2: e \rightarrow t$

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

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

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, Id:1	+ Id $	Reduce 4
0, Id:1, *:3, t:5	+ Id $	Reduce 3
0, t:2	+ Id $	Shift, goto 4
0, t:2, +:4	Id $	Shift, goto 1
0, t:2, +:4, Id:1	$	Reduce 4
0, t:2, +:4, t:2	$	Reduce 2
0, t:2, +:4, e:6	$	Reduce 1
0, e:7	$	Accept

e → t + e

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

$t \rightarrow \cdot$ **Else** s

$t \rightarrow \cdot$

Else — makes it accepting

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

$t \rightarrow$ **Id** $* \; t \cdot$

$e \rightarrow$ **Id** $* \; t \cdot$

$f \rightarrow e$

Do you reduce by "$t \rightarrow$ **Id** $* \; t$" or by "$e \rightarrow$ **Id** $* \; t$"?

follow(t) $\ni$ Else ? $e \rightarrow \ldots t$ Else ..

A Reduce/Reduce Conflict

$1: a \rightarrow$ **Id Id**
$2: a \rightarrow b$
$3: b \rightarrow$ **Id Id**