

# Parser

---

Ronghui Gu

Spring 2022

Columbia University

\* Course website: <https://verigu.github.io/4115Spring2022/>

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

## The Big Picture

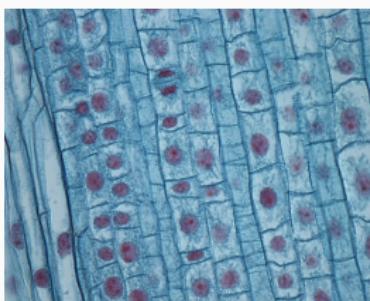
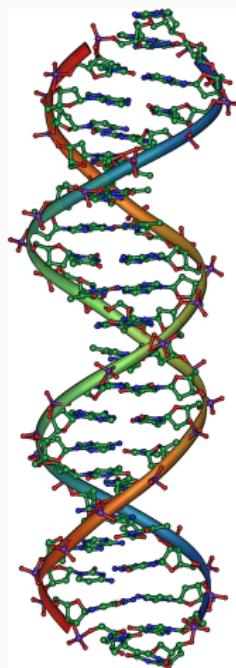
---

# The First Question

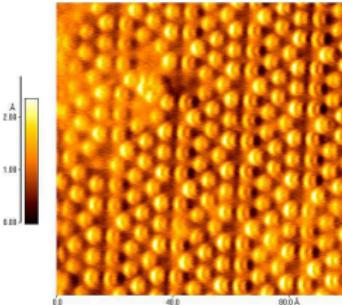
How do we describe/construct a program?

# Solution: Use a Discrete Combinatorial System

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



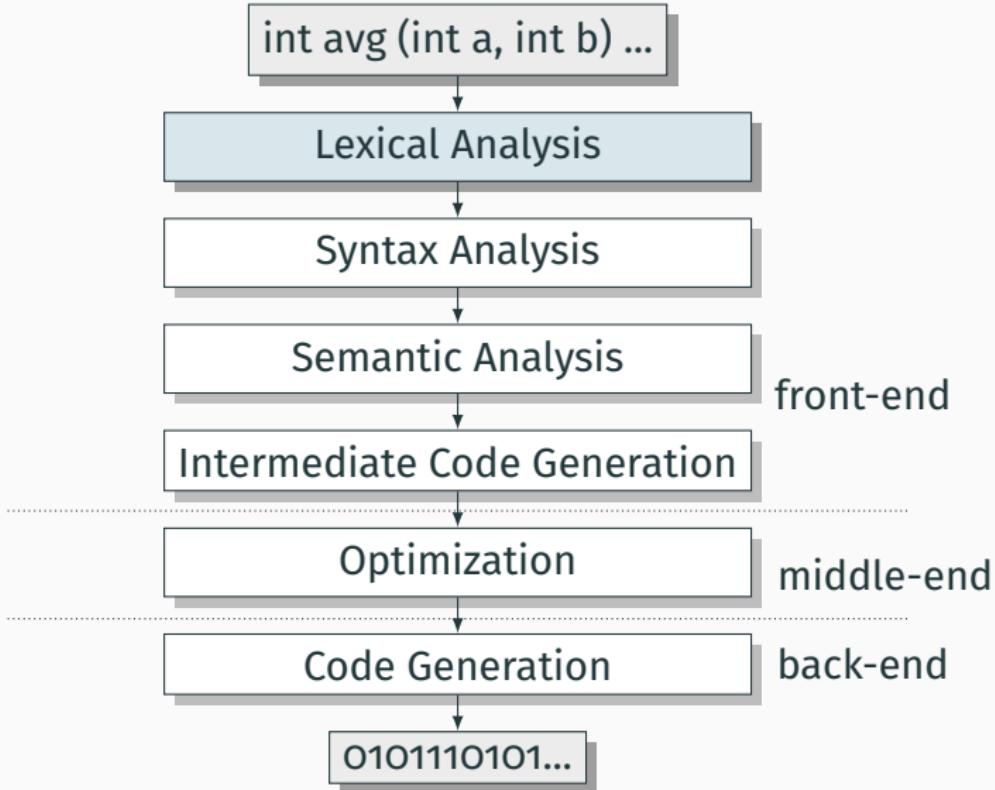
ENGLISH SOUNDS			
cheese	rich	vocal	vocal
elephant	camera	bird	bell
fat	last	car	lock
flower	van	feature	shovel
mouse	orange	house	light
	king	high	ring
		sun	water
		ring	young



## The Second Question

How do we combine **characters** into **words**?

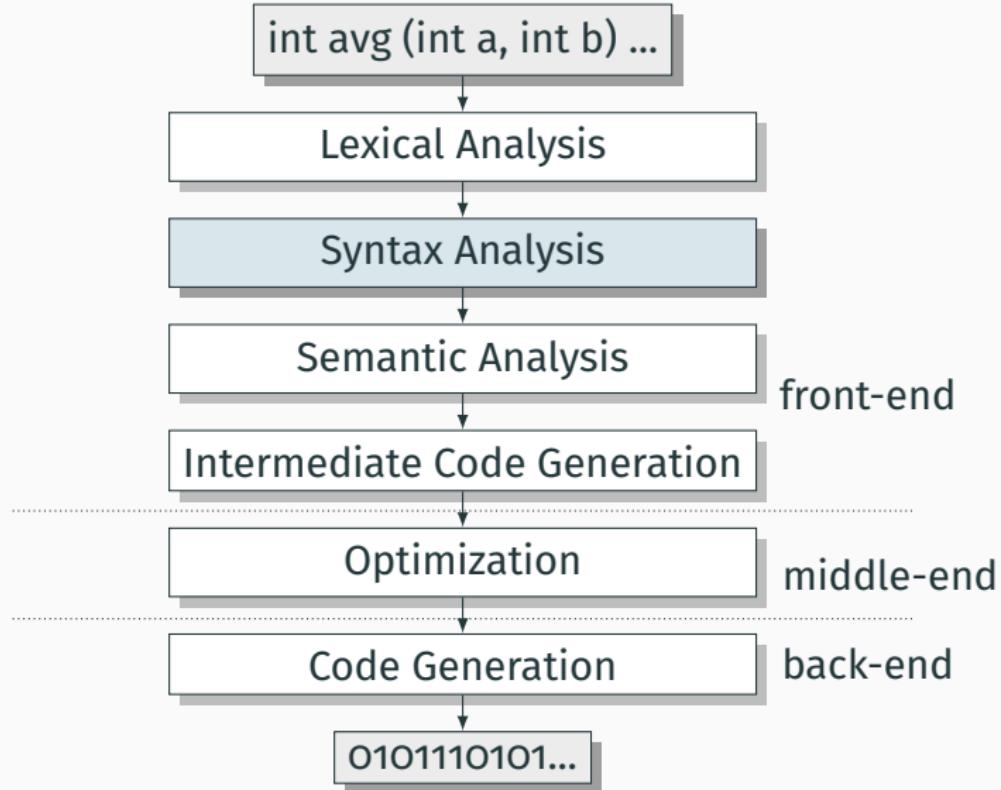
# Scanner



## The Third Question

How do we combine words into sentences?

# Parser



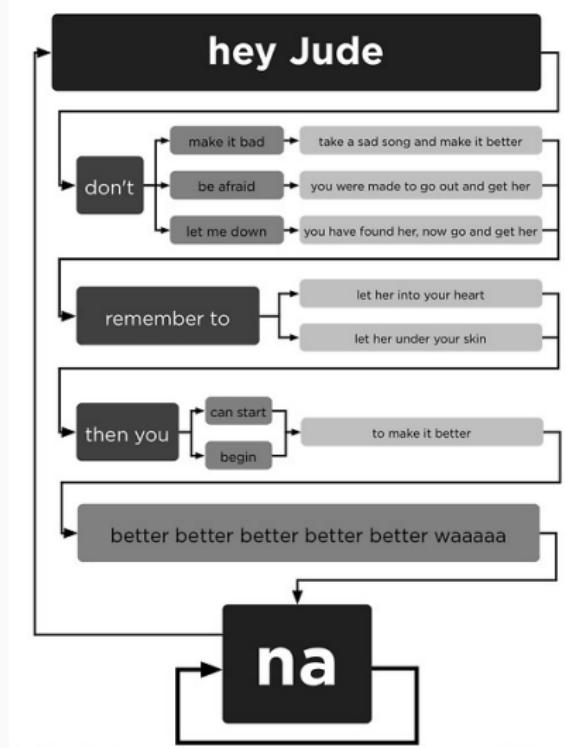
# Choices: CS Research Jargon Generator

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>



loveallthis.tumblr.com

lyrics © sony atv

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

## How about more structured collections of things?

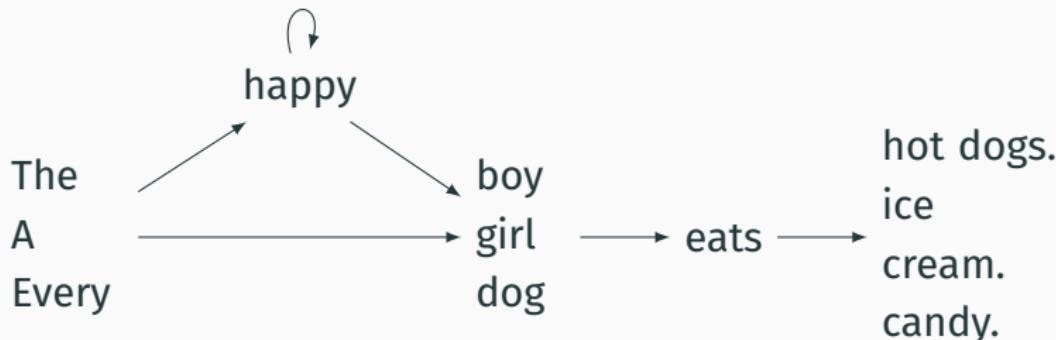
The boy eats hot dogs.

The dog eats ice cream.

Every happy girl eats candy.

A dog eats candy.

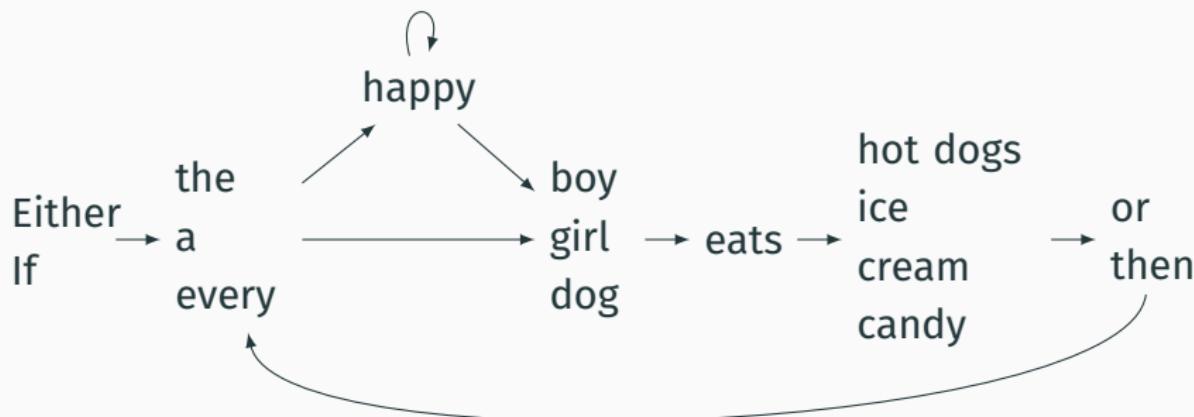
The happy happy dog eats hot dogs.



# Richer Sentences Are Harder

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

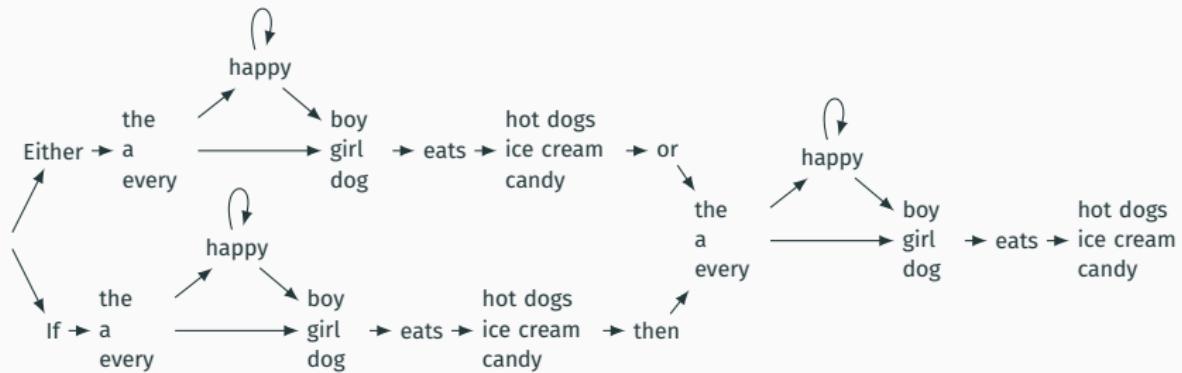
Either the boy eats candy, or every dog eats candy.



*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

$S \rightarrow \text{Either } A$

$S \rightarrow \text{If } A$

$A \rightarrow \text{the } B$

$A \rightarrow \text{the } C$

$A \rightarrow \text{a } B$

$A \rightarrow \text{a } C$

$A \rightarrow \text{every } B$

$A \rightarrow \text{every } C$

$B \rightarrow \text{happy } B$

$B \rightarrow \text{happy } C$

$C \rightarrow \text{boy } D$

$C \rightarrow \text{girl } D$

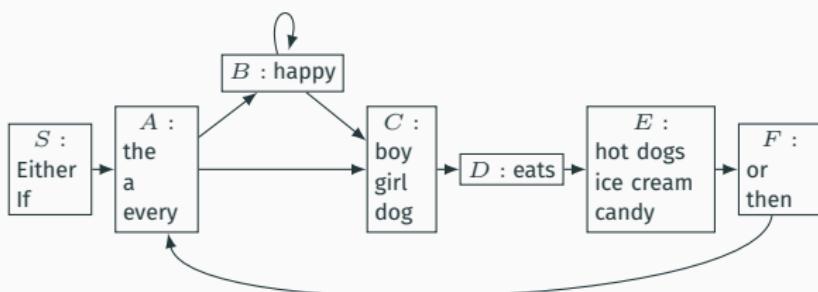
$C \rightarrow \text{dog } D$

$D \rightarrow \text{eats } E$

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

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

$E \rightarrow \text{candy } F$



## 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

## An Example

*n* 0's followed by *n* 1's, e.g., 000111, 01

$$S \rightarrow 0 \ S \ 1.$$

$$S \rightarrow \epsilon.$$

# **Constructing Grammars and Ocamlyacc**

---

# Parsing

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



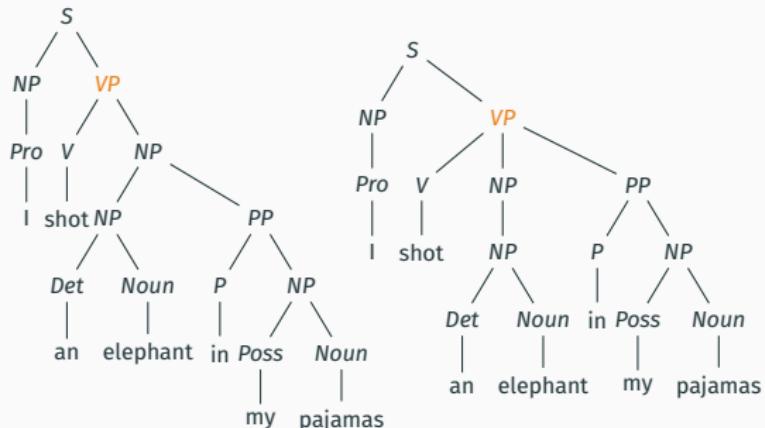
Goal: verify the syntax of the program, discard irrelevant information, and “understand” the structure of the program.

Parentheses and most other forms of punctuation removed.

# Ambiguity 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*?

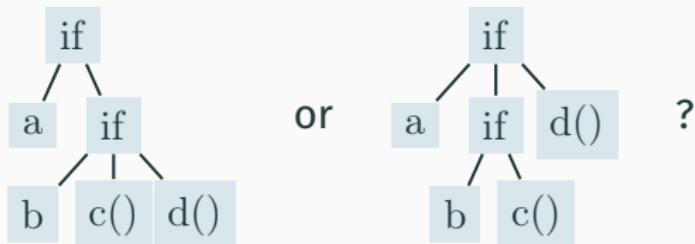
if (a) if (b) c(); else d();

```
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

Should this be



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

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...
```

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

# 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...
```

if (a) if (b) c(); else d();  
cstmt?

## Another Solution to the Dangling Else Problem

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.

## Another Solution to 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.

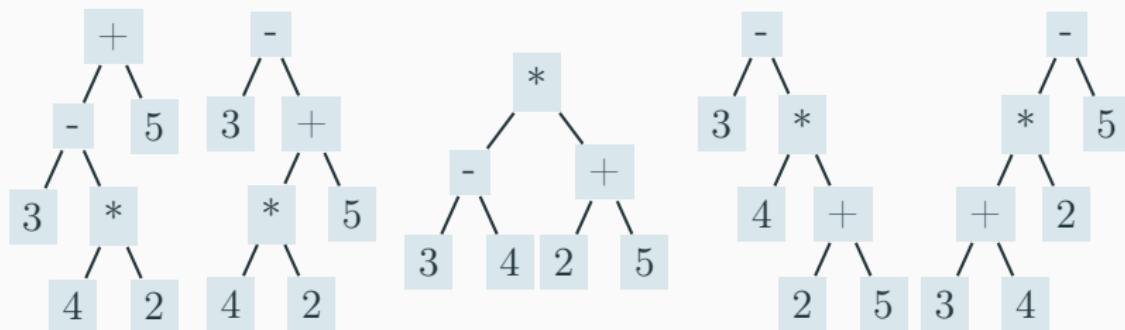
# Ambiguous Arithmetic

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.

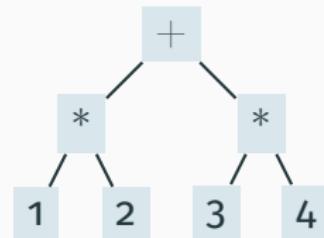
# Operator Precedence

Defines how “sticky” an operator is.

$$1 * 2 + 3 * 4$$

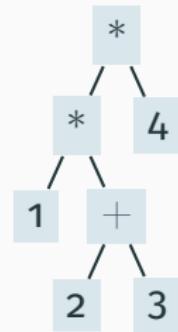
\* at higher precedence than +:

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



+ at higher precedence than \*:

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

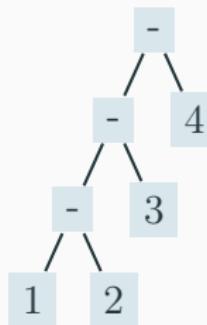


# Associativity

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

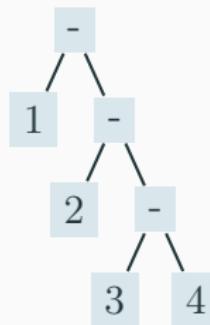
Most operators are left-associative

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.”

$1 * 2 + 3?$

expr TIMES expr PLUS *shift*?

expr TIMES expr PLUS *reduce*?

# 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."

$1 * 2 + 3?$

term TIMES term PLUS *cannot shift!*

term TIMES term PLUS *cannot reduce!*

term TIMES term PLUS *reduce!*

# 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."

$1 * 2 * 3 ?$

term TIMES term TIMES *shift*?

term TIMES term PLUS *reduce*?

# 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.

$1 * 2 * 3 ?$

term TIMES atom TIMES *cannot shift!*

term TIMES atom TIMES *cannot reduce!*

term TIMES atom TIMES *reduce!*

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

$e$   
 $t + e$   
 $t + t$

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.

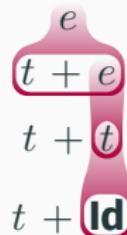
## Rightmost Derivation of $\text{Id} * \text{Id} + \text{Id}$

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.

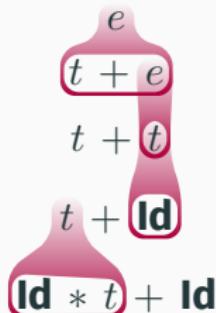
## Rightmost Derivation of $\text{Id} * \text{Id} + \text{Id}$

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.

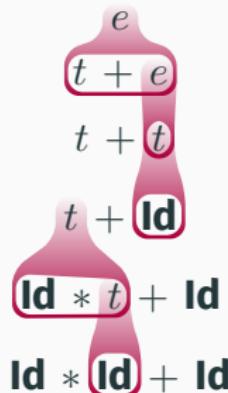
## Rightmost Derivation of $\text{Id} * \text{Id} + \text{Id}$

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.

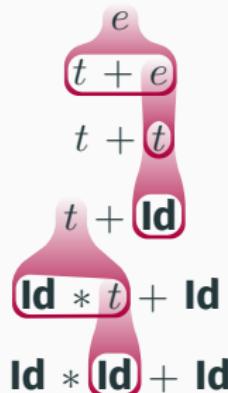
## Rightmost Derivation of $\text{Id} * \text{Id} + \text{Id}$

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

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

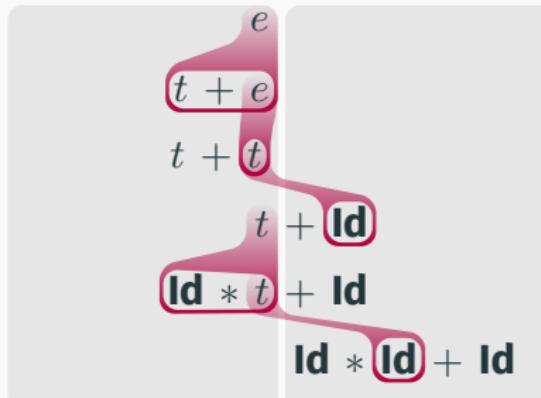
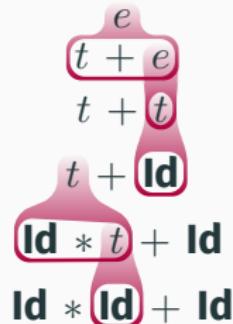
## Rightmost Derivation: What to Expand

1 :  $e \rightarrow t + e$

2 :  $e \rightarrow t$

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

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



Expand here ↑

Terminals only

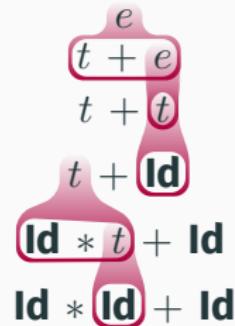
# Reverse Rightmost Derivation

1 :  $e \rightarrow t + e$

2 :  $e \rightarrow t$

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

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



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

viable prefixes

terminals

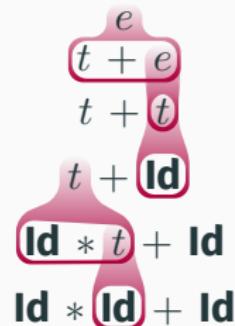
# Reverse Rightmost Derivation

1 :  $e \rightarrow t + e$

2 :  $e \rightarrow t$

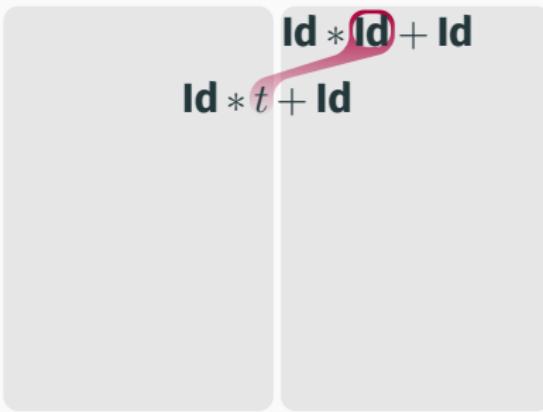
3 :  $t \rightarrow \mathbf{Id} * t$

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



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

$\mathbf{Id}$   
  |  
  t



viable prefixes

terminals

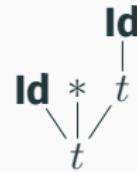
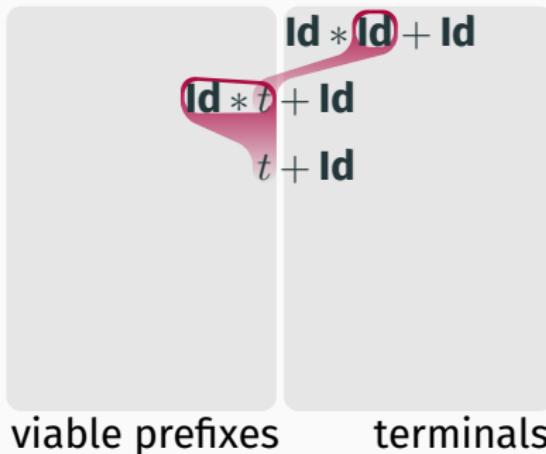
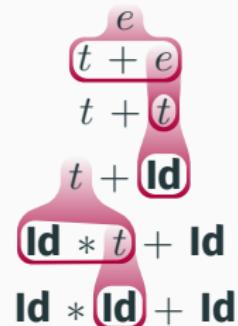
# Reverse Rightmost Derivation

1 :  $e \rightarrow t + e$

2 :  $e \rightarrow t$

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

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



terminals

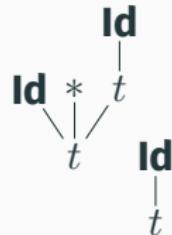
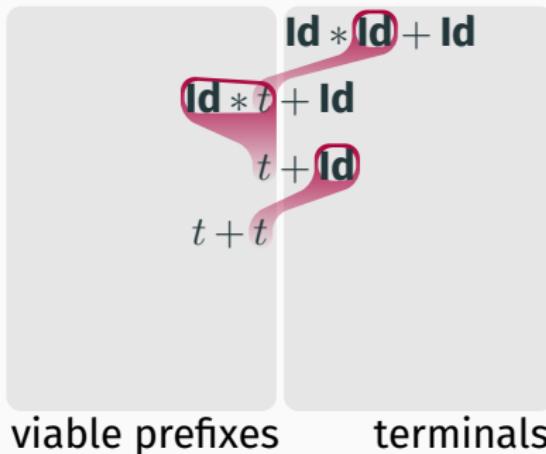
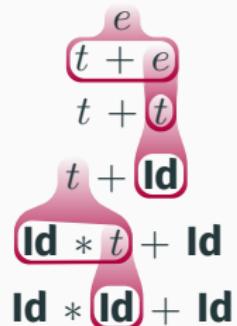
# Reverse Rightmost Derivation

1 :  $e \rightarrow t + e$

2 :  $e \rightarrow t$

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

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



terminals

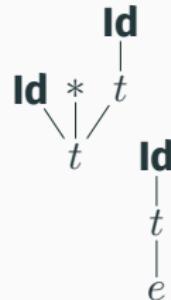
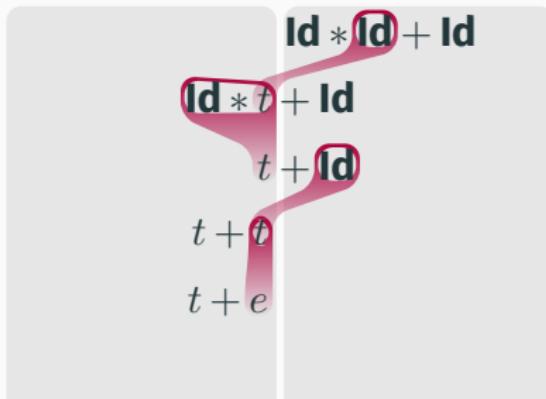
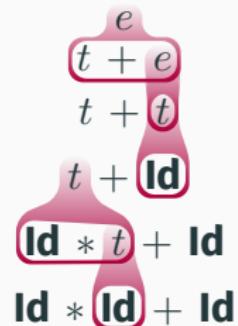
# Reverse Rightmost Derivation

1 :  $e \rightarrow t + e$

2 :  $e \rightarrow t$

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

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



viable prefixes

terminals

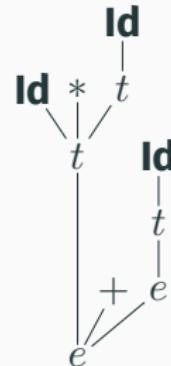
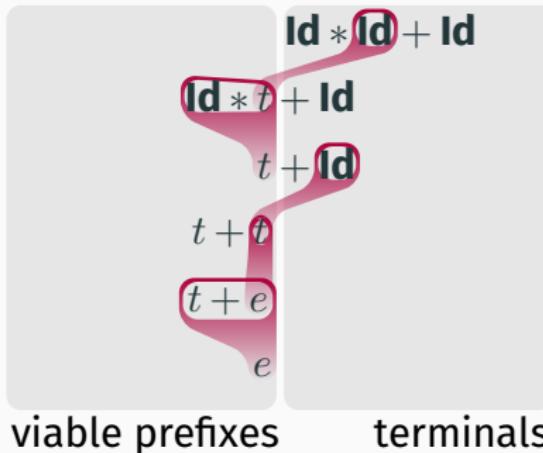
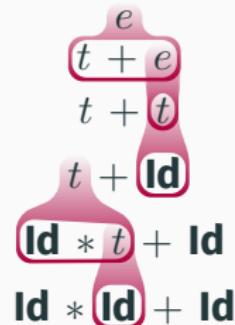
# Reverse Rightmost Derivation

1 :  $e \rightarrow t + e$

2 :  $e \rightarrow t$

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

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



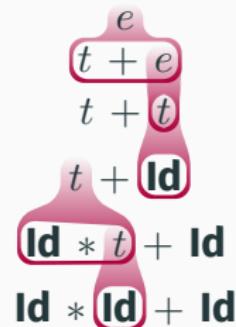
# Shift/Reduce Parsing Using an Oracle

1 :  $e \rightarrow t + e$

2 :  $e \rightarrow t$

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

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



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

shift

stack

input

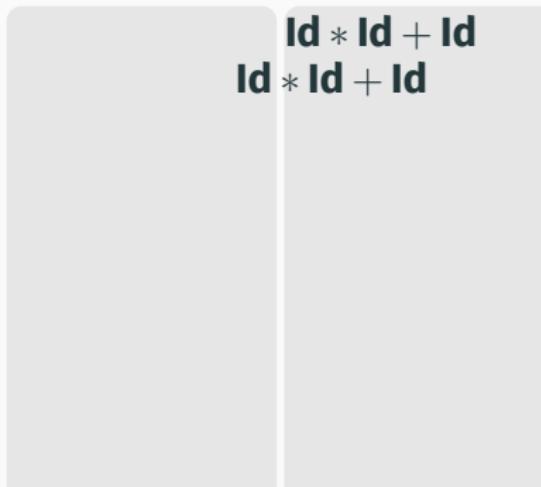
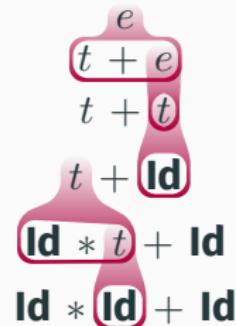
# Shift/Reduce Parsing Using an Oracle

1 :  $e \rightarrow t + e$

2 :  $e \rightarrow t$

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

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



shift  
shift

stack

input

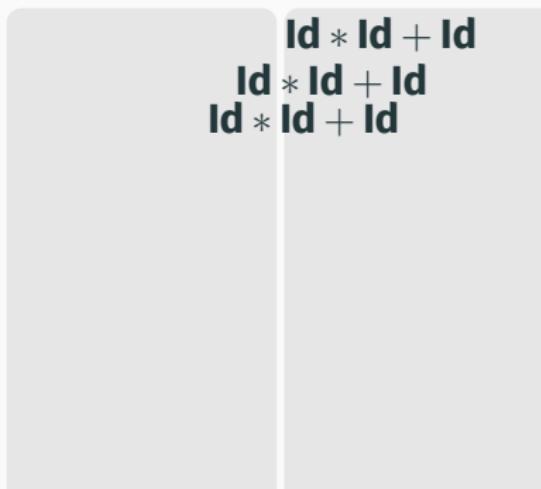
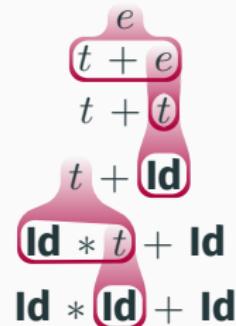
# Shift/Reduce Parsing Using an Oracle

1 :  $e \rightarrow t + e$

2 :  $e \rightarrow t$

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

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



input

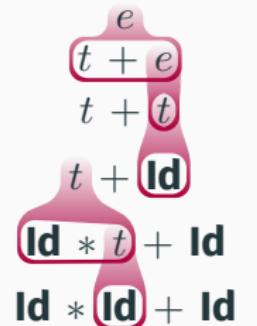
# Shift/Reduce Parsing Using an Oracle

1 :  $e \rightarrow t + e$

2 :  $e \rightarrow t$

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

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



$\text{Id} * \text{Id} + \text{Id}$	shift
$\text{Id} * \text{Id} + \text{Id}$	shift
$\text{Id} * \text{Id} + \text{Id}$	shift
$\text{Id} * \text{Id} + \text{Id}$	reduce 4

stack

input

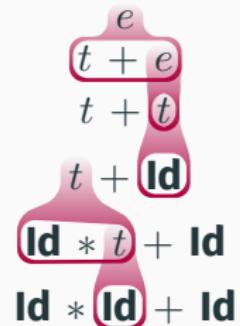
# Shift/Reduce Parsing Using an Oracle

1 :  $e \rightarrow t + e$

2 :  $e \rightarrow t$

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

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



stack	input
$\text{Id} * \text{Id} + \text{Id}$	
$\text{Id} * t + \text{Id}$	

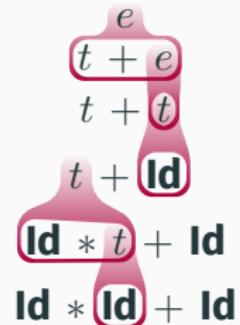
# Shift/Reduce Parsing Using an Oracle

1 :  $e \rightarrow t + e$

2 :  $e \rightarrow t$

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

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



$\text{Id} * \text{Id} + \text{Id}$

$\text{Id} * t + \text{Id}$

$t + \text{Id}$

shift

shift

shift

reduce 4

reduce 3

shift

stack

input

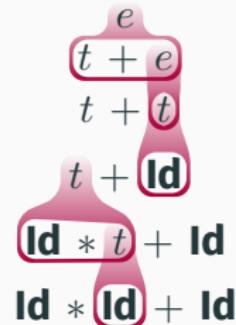
# Shift/Reduce Parsing Using an Oracle

1 :  $e \rightarrow t + e$

2 :  $e \rightarrow t$

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

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



$\text{Id} * \text{Id} + \text{Id}$	shift
$\text{Id} * \text{Id} + \text{Id}$	shift
$\text{Id} * \text{Id} + \text{Id}$	shift
$\text{Id} * \text{Id} + \text{Id}$	reduce 4
$\text{Id} * t + \text{Id}$	reduce 3
$t + \text{Id}$	shift
$t + \text{Id}$	shift

stack

input

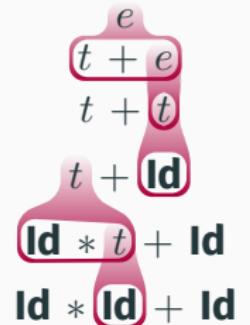
# Shift/Reduce Parsing Using an Oracle

1 :  $e \rightarrow t + e$

2 :  $e \rightarrow t$

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

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



$\text{Id} * \text{Id} + \text{Id}$	shift
$\text{Id} * \text{Id} + \text{Id}$	shift
$\text{Id} * \text{Id} + \text{Id}$	shift
$\text{Id} * \text{Id} + \text{Id}$	reduce 4
$\text{Id} * t + \text{Id}$	reduce 3
$t + \text{Id}$	shift
$t + \text{Id}$	shift
$t + \text{Id}$	reduce 4

stack

input

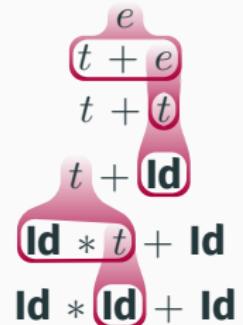
# Shift/Reduce Parsing Using an Oracle

1 :  $e \rightarrow t + e$

2 :  $e \rightarrow t$

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

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



$\text{Id} * \text{Id} + \text{Id}$	shift
$\text{Id} * \text{Id} + \text{Id}$	shift
$\text{Id} * \text{Id} + \text{Id}$	shift
$\text{Id} * \text{Id} + \text{Id}$	reduce 4
$\text{Id} * t + \text{Id}$	reduce 3
$t + \text{Id}$	shift
$t + \text{Id}$	shift
$t + \text{Id}$	reduce 4
$t + t$	reduce 2

stack

input

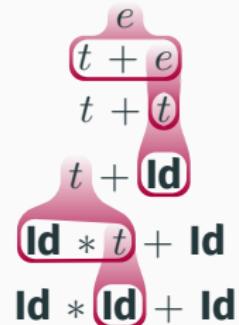
# Shift/Reduce Parsing Using an Oracle

1 :  $e \rightarrow t + e$

2 :  $e \rightarrow t$

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

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



stack	input
$\text{Id} * \text{Id} + \text{Id}$	
$\text{Id} * t + \text{Id}$	
$t + \text{Id}$	
$t + \text{Id}$	
$t + \text{Id}$	
$t + t$	
$t + e$	

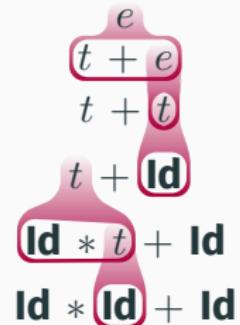
# Shift/Reduce Parsing Using an Oracle

1 :  $e \rightarrow t + e$

2 :  $e \rightarrow t$

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

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



stack	input
$\text{Id} * \text{Id} + \text{Id}$	
$\text{Id} * t + \text{Id}$	
$t + \text{Id}$	
$t + \text{Id}$	
$t + \text{Id}$	
$t + t$	
$t + e$	
$e$	

## 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.*

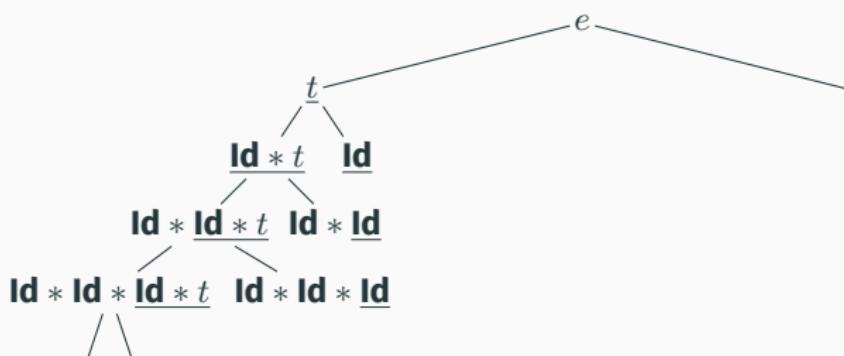
# Some Right-Sentential Forms and Their Handles

1 :  $e \rightarrow t + e$

2 :  $e \rightarrow t$

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

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



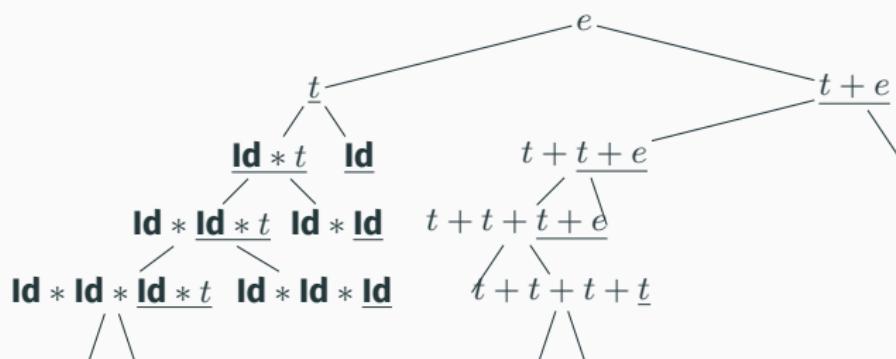
## **Some Right-Sentential Forms and Their Handles**

$$1 : e \rightarrow t + e$$

$2 : e \rightarrow t$

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

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



# 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: **Id** \* **Id** \* ... \* **Id** \* *t* ...

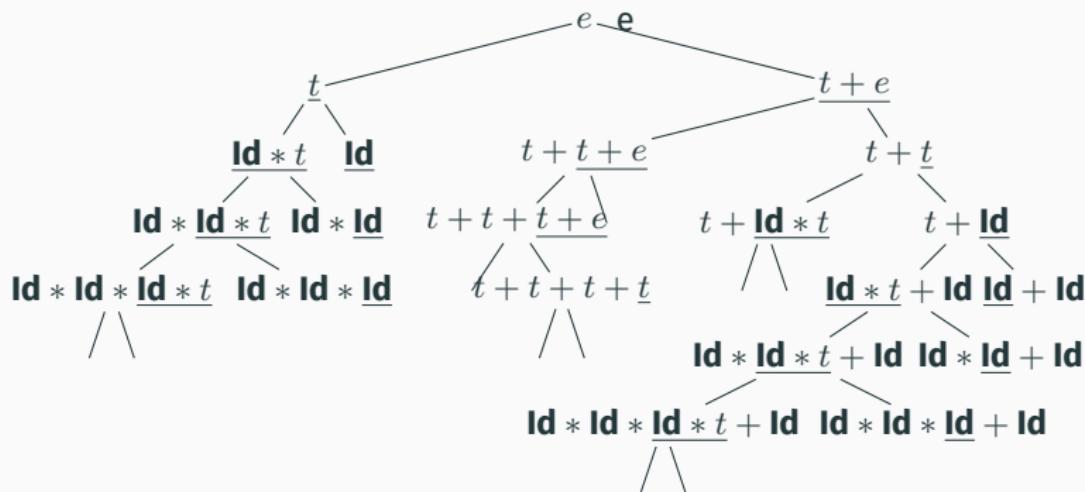
**Id** \* **Id** \* . . . \* **Id** . . .

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

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

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

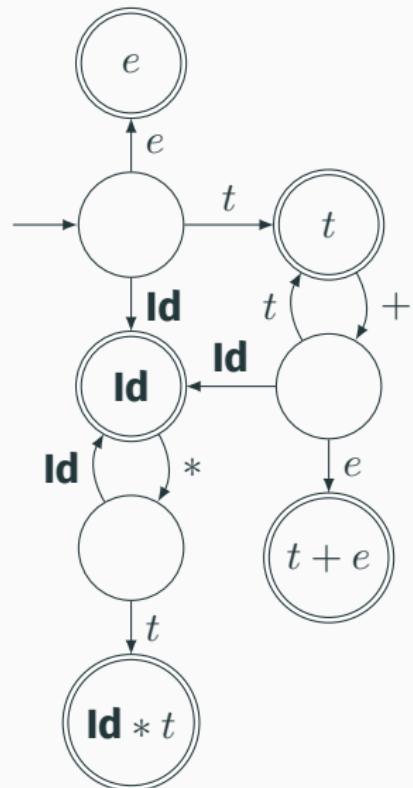
$$t + t + \cdots + t$$



# The Handle-Identifying Automaton

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

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



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

$e' \rightarrow \bullet e$

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

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

$e' \rightarrow \bullet e$

$e \rightarrow \bullet t + e$

$e \rightarrow \bullet t$

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

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

$e' \rightarrow \bullet e$

$e \rightarrow \bullet t + e$

$e \rightarrow \bullet t$

$t \rightarrow \bullet \text{Id} * t$

$t \rightarrow \bullet \text{Id}$

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  $\text{Id} * t$  or  $\text{Id}$ , so  $t \rightarrow \bullet \text{Id} * t$  and  $t \rightarrow \bullet \text{Id}$ .

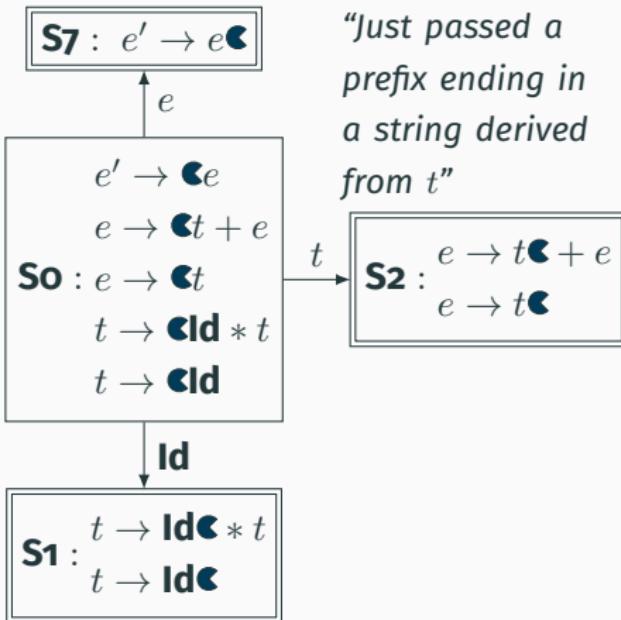
This is a *closure*, like  $\epsilon$ -closure in subset construction.

# Building the LR(0) Automaton

$e' \rightarrow \bullet e$
$e \rightarrow \bullet t + e$
<b>So :</b> $e \rightarrow \bullet t$
$t \rightarrow \bullet \text{Id} * t$
$t \rightarrow \bullet \text{Id}$

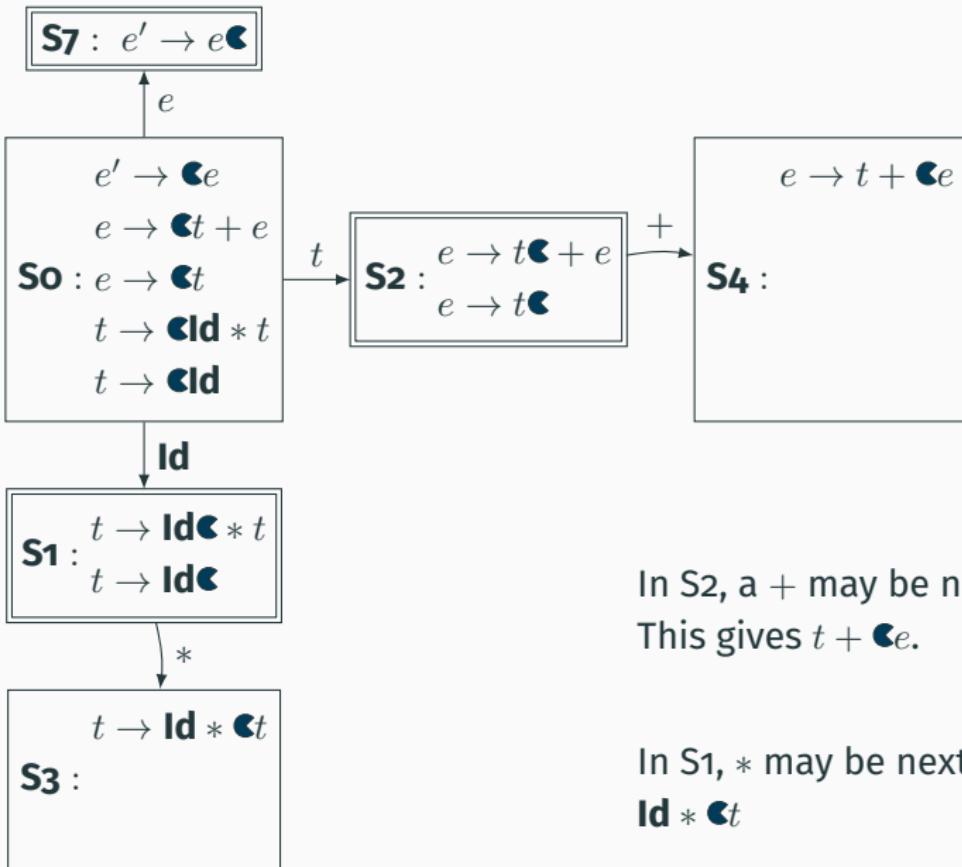
The first state suggests a viable prefix can start as any string derived from  $e$ , any string derived from  $t$ , or **Id**.

# Building the LR(0) Automaton

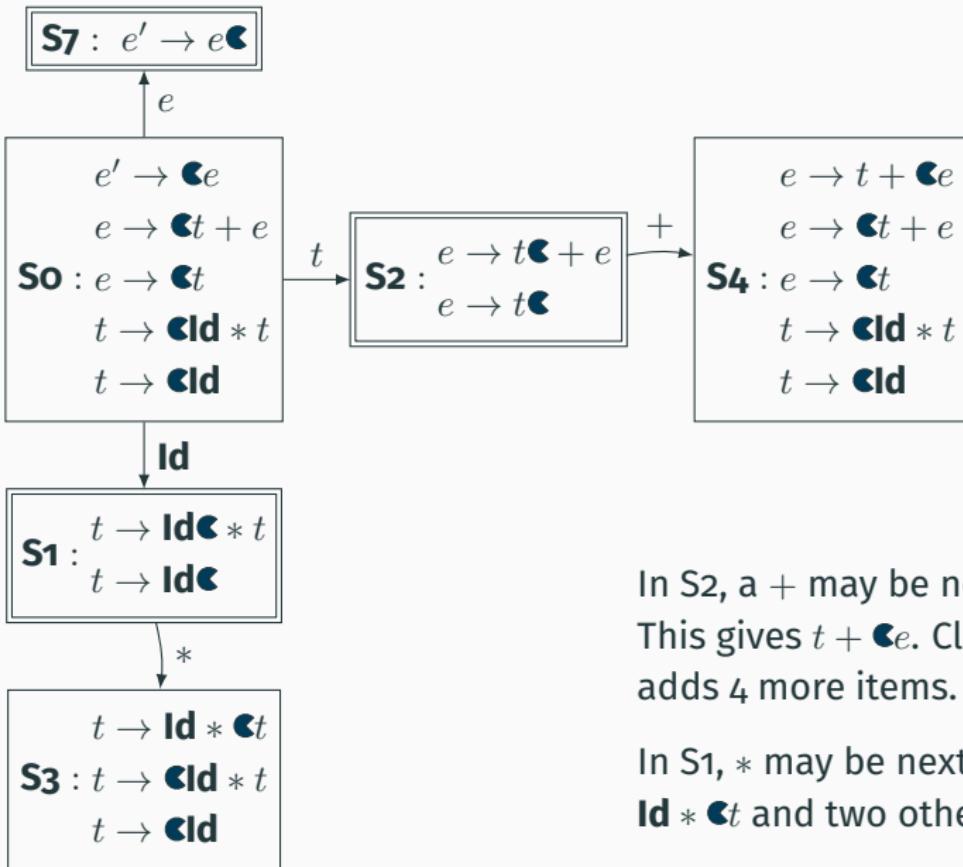


The first state suggests a viable prefix can start as any string derived from  $e$ , any string derived from  $t$ , or **Id**. The items for these three states come from advancing the  $\bullet$  across each thing, then performing the closure operation (vacuous here).

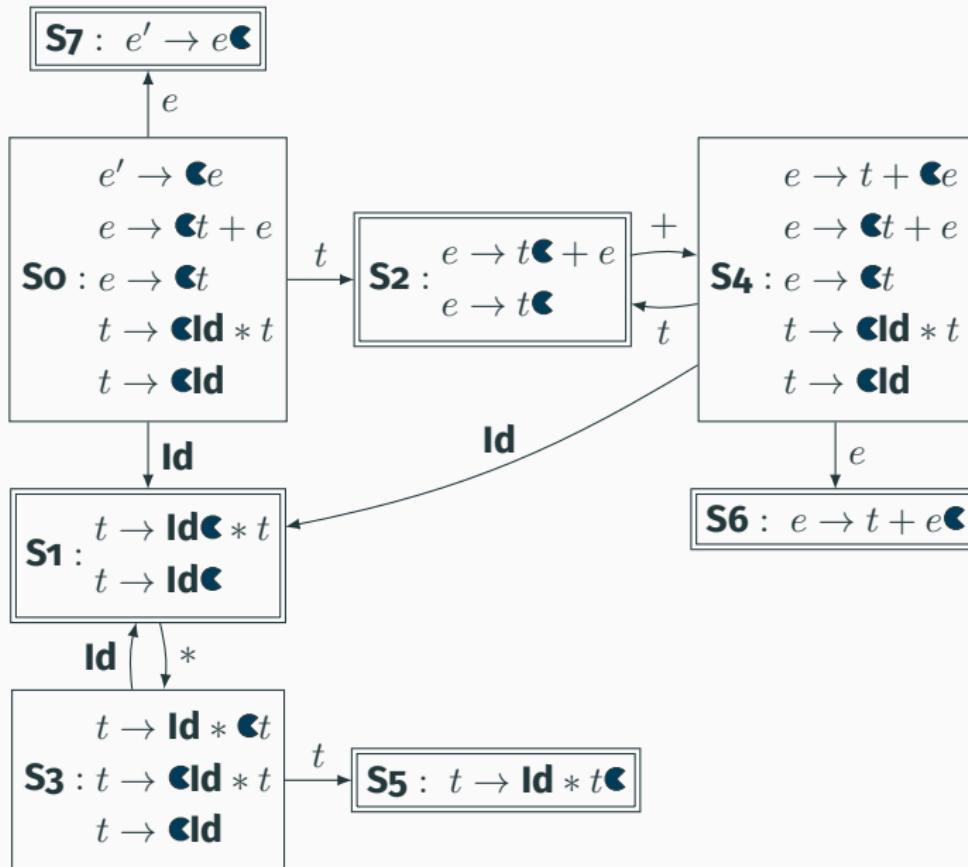
# Building the LR(0) Automaton



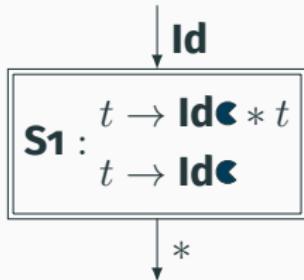
# Building the LR(0) Automaton



# Building the LR(0) Automaton



# What to do in each state?



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

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

---

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

---

# The FIRST function

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

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

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

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

---

$$1 : e \rightarrow t + e$$

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

$$2 : e \rightarrow t$$

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

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

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

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

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

## First and $\epsilon$

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

1. If  $X$  is a terminal,  $\text{FIRST}(X) = \{X\}$ .

2. If  $X \rightarrow \epsilon$ , then add  $\epsilon$  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  $\epsilon$  to  $\text{FIRST}(X)$

$$X \rightarrow Y Z a$$

$$\text{FIRST}(b) = \{b\} \quad \text{FIRST}(c) = \{c\} \quad \text{FIRST}(d) = \{d\} \quad (1)$$

$$Y \rightarrow$$

$$\text{FIRST}(W) = \{\epsilon\} \cup \text{FIRST}(d) = \{\epsilon, d\} \quad (2, 3)$$

$$Y \rightarrow b$$

$$\text{FIRST}(Z) = \text{FIRST}(c) \cup (\text{FIRST}(W) - \{\epsilon\}) \cup \{\epsilon\} = \{\epsilon, c, d\} \quad (3, 3, 4)$$

$$Z \rightarrow c$$

$$\text{FIRST}(Y) = \{\epsilon\} \cup \{b\} = \{\epsilon, b\} \quad (2, 3)$$

$$Z \rightarrow W$$

$$\text{FIRST}(X) = (\text{FIRST}(Y) - \{\epsilon\}) \cup (\text{FIRST}(Z) - \{\epsilon\}) \cup \quad (3, 3, 3)$$

$$W \rightarrow$$

$$\text{FIRST}(a) = \{a, b, c, d\} \quad 49$$

$$W \rightarrow d$$

## The FOLLOW function

If  $t$  is a terminal,  $A$  is a nonterminal, and  $\dots A t \dots$  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*

---

$$1 : e \rightarrow t + e$$

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

$$2 : e \rightarrow t$$

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

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

1. Because  $e$  is the start symbol

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

$$\text{FIRST}(t) = \{\mathbf{Id}\}$$

$$\text{FIRST}(e) = \{\mathbf{Id}\}$$

## The FOLLOW function

If  $t$  is a terminal,  $A$  is a nonterminal, and  $\dots A t \dots$  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*

---

$$1 : e \rightarrow t + e$$

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

$$2 : e \rightarrow t$$

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

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

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

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

$$\text{FIRST}(t) = \{\mathbf{Id}\}$$

$$\text{FIRST}(e) = \{\mathbf{Id}\}$$

## The FOLLOW function

If  $t$  is a terminal,  $A$  is a nonterminal, and  $\dots A t \dots$  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*

---

$$1 : e \rightarrow t + e$$

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

$$2 : e \rightarrow t$$

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

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

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

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

$$\text{FIRST}(t) = \{\mathbf{Id}\}$$

$$\text{FIRST}(e) = \{\mathbf{Id}\}$$

# The FOLLOW function

If  $t$  is a terminal,  $A$  is a nonterminal, and  $\dots A t \dots$  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*

---

$$1 : e \rightarrow t + e$$

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

$$2 : e \rightarrow t$$

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

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

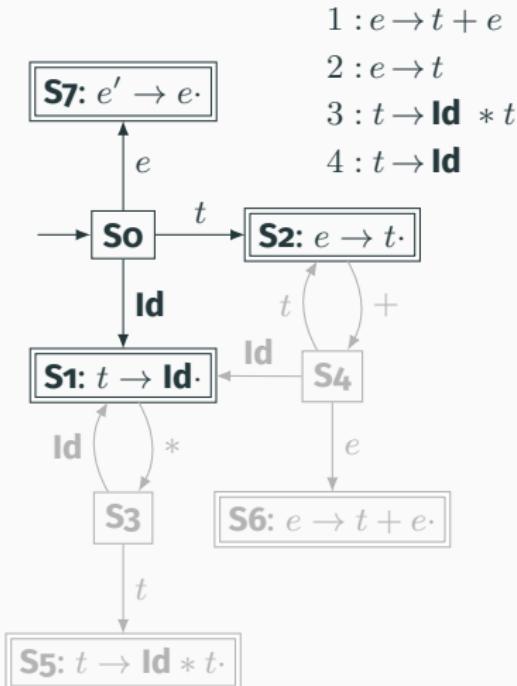
Fixed-point reached: applying any rule does not change any set

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

$$\text{FIRST}(t) = \{\mathbf{Id}\}$$

$$\text{FIRST}(e) = \{\mathbf{Id}\}$$

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

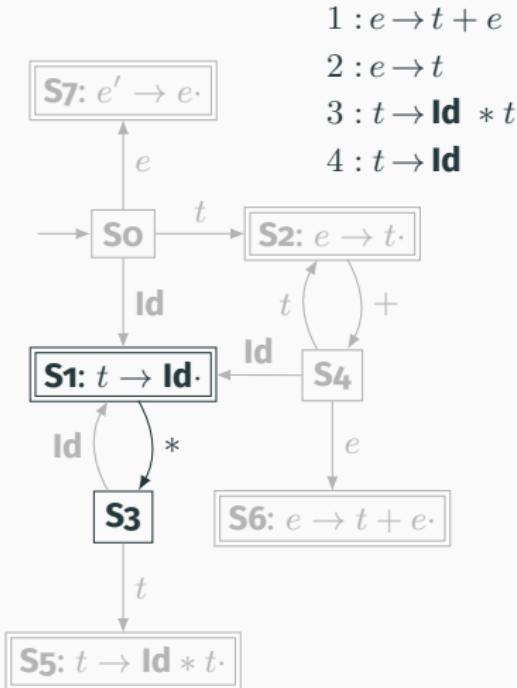
State	Action				Goto	
	$\text{Id}$	$+$	$*$	$\$$		
0	s1				7	2

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

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

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



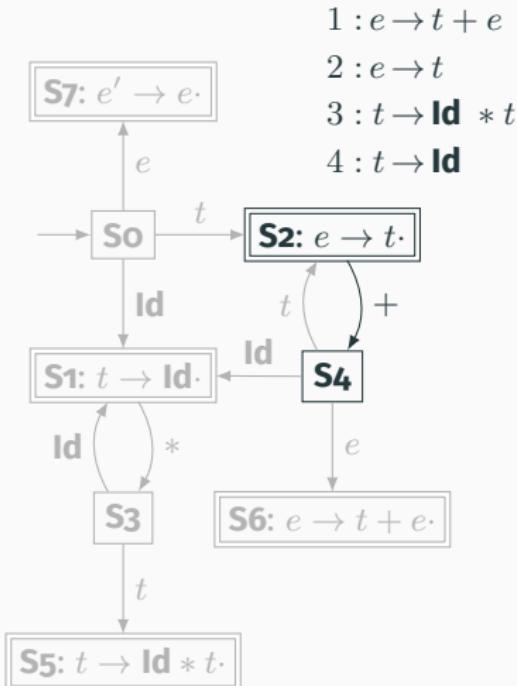
$\text{FOLLOW}(e) = \{\$\}$

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

State	Action				Goto	
	$\text{Id}$	$+$	$*$	$\$$		
0	$s_1$				7	2
1		$r_4$	$s_3$	$r_4$		

From  $S_1$ , shift a  $*$  and go to  $S_3$ ; or, if the next input  $\in \text{FOLLOW}(t)$ , reduce by rule 4.

# Converting the LR(0) Automaton to an SLR Table



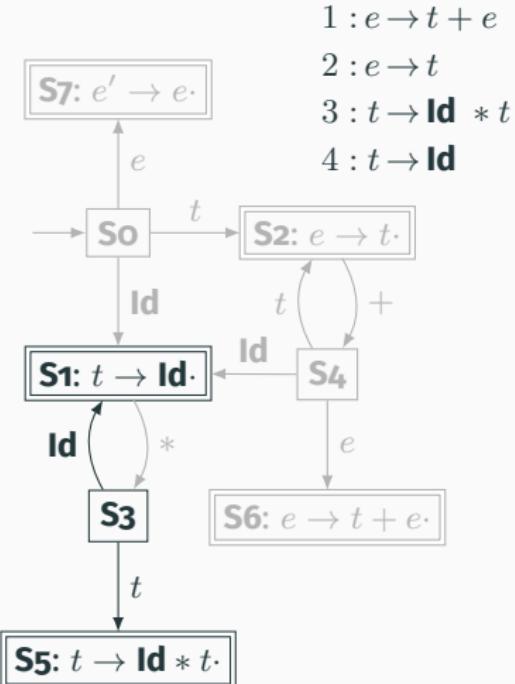
$\text{FOLLOW}(e) = \{\$\}$

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

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

From  $S_2$ , shift a  $+$  and go to  $S_4$ ; or, if the next input  $\in \text{FOLLOW}(e)$ , reduce by rule 2.

# Converting the LR(0) Automaton to an SLR Table

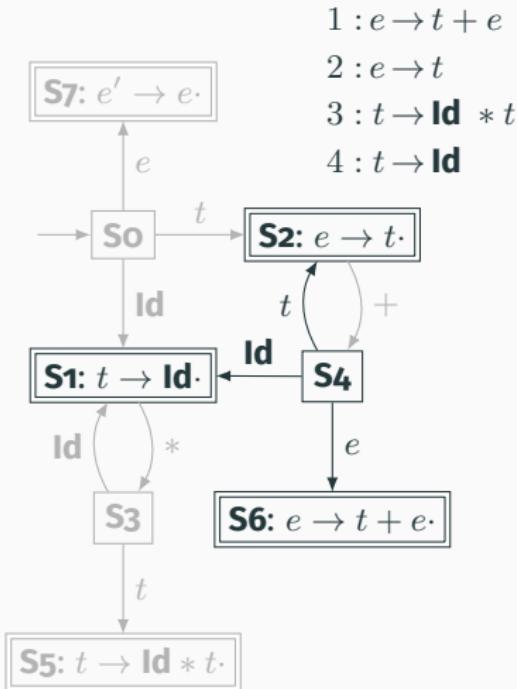


$\text{FOLLOW}(e) = \{\$\}$   
 $\text{FOLLOW}(t) = \{+, \$\}$

State	Action					Goto	
	<b>Id</b>	<b>+</b>	<b>*</b>	<b>\$</b>	<b>e</b>	<b>t</b>	
0	$s_1$						7 2
1		$r_4$	$s_3$	$r_4$			
2			$s_4$		$r_2$		
3	$s_1$						5

From  $S_3$ , shift an **Id** and go to  $S_1$ ; or cross a  $t$  and go to  $S_5$ .

# Converting the LR(0) Automaton to an SLR Table



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

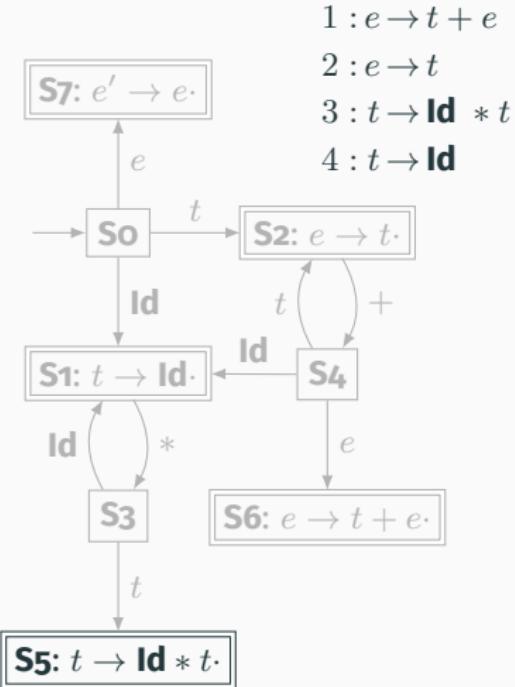
$\text{FOLLOW}(t) = \{+, \$\}$

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

State	Action				Goto	
	<b>Id</b>	<b>+</b>	<b>*</b>	<b>\$</b>	<b><math>e</math></b>	<b><math>t</math></b>
0	$s_1$				7	2
1		$r_4$	$s_3$	$r_4$		
2		$s_4$		$r_2$		
3	$s_1$					5
4	$s_1$				6	2

From  $S_4$ , shift an **Id** and go to  $S_1$ ; or cross an  $e$  or a  $t$ .

# Converting the LR(0) Automaton to an SLR Table

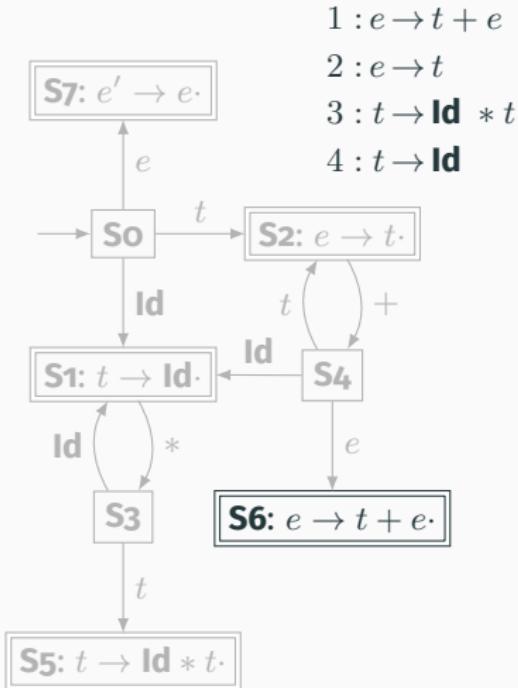


$\text{FOLLOW}(e) = \{\$\}$   
 $\text{FOLLOW}(t) = \{+, \$\}$

State	Action				Goto	
	$\text{Id}$	$+$	$*$	$\$$	$e$	$t$
0	$s_1$				7	2
1		$r_4$	$s_3$	$r_4$		
2		$s_4$		$r_2$		
3	$s_1$					5
4	$s_1$				6	2
5		$r_3$		$r_3$		

From  $S_5$ , reduce using rule 3 if the next symbol  $\in \text{FOLLOW}(t)$ .

# Converting the LR(0) Automaton to an SLR Table



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

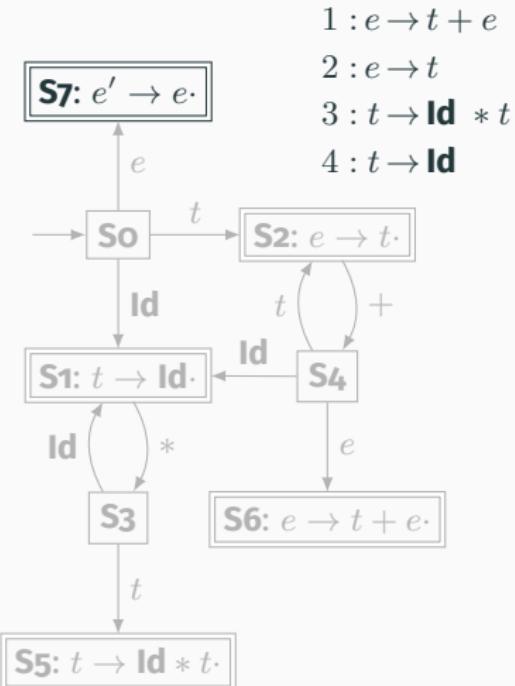
$\text{FOLLOW}(t) = \{+, \$\}$

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

State	Action				Goto	
	$\mathbf{Id}$	$+$	$*$	$\$$	$e$	$t$
0	$s_1$				7	2
1		$r_4$	$s_3$	$r_4$		
2		$s_4$		$r_2$		
3	$s_1$					5
4	$s_1$				6	2
5		$r_3$		$r_3$		
6				$r_1$		

From  $S_6$ , reduce using rule 1 if the next symbol  $\in \text{FOLLOW}(e)$ .

# Converting the LR(0) Automaton to an SLR Table



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

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

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

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				✓		

If, in  $S_7$ , 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 :  $t \rightarrow \text{Id} * t$

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

	Stack		Input		Action
	o		<b>Id * Id + Id \$</b>		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.

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

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

Stack	Input	Action
O	$\text{Id} * \text{Id} + \text{Id} \$$	Shift, goto 1
Id 1	$* \text{Id} + \text{Id} \$$	Shift, goto 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				✓		

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

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

State	Action				Goto	
	<b>Id</b>	<b>+</b>	<b>*</b>	<b>\$</b>	<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
O	<b>Id</b> * <b>Id</b> + <b>Id</b> \$	Shift, goto 1
O 1	* <b>Id</b> + <b>Id</b> \$	Shift, goto 3
O 1 3	<b>Id</b> + <b>Id</b> \$	Shift, goto 1
O 1 3 1	+ <b>Id</b> \$	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	
	$\mathbf{Id}$	+	*	\$	$e$	$t$
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
O	$\mathbf{Id} * \mathbf{Id} + \mathbf{Id} \$$	Shift, goto 1
O 1	* $\mathbf{Id} + \mathbf{Id} \$$	Shift, goto 3
O 1 3	$\mathbf{Id} + \mathbf{Id} \$$	Shift, goto 1
O 1 3 1	+ $\mathbf{Id} \$$	Reduce 4
O 1 3	+ $\mathbf{Id} \$$	

Remove the RHS of the rule (the handle: here, just  $\mathbf{Id}$ ), 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	
	$\text{Id}$	$+$	$*$	$\$$	$e$	$t$
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
O	$\text{Id} * \text{Id} + \text{Id} \$$	Shift, goto 1
O Id	$* \text{Id} + \text{Id} \$$	Shift, goto 3
O Id *	$\text{Id} + \text{Id} \$$	Shift, goto 1
O Id * Id	$+ \text{Id} \$$	Reduce 4
O Id * t	$+ \text{Id} \$$	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	+	*	\$	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
O	$\mathbf{Id} * \mathbf{Id} + \mathbf{Id} \$$	Shift, goto 1
O 1	$* \mathbf{Id} + \mathbf{Id} \$$	Shift, goto 3
O 1 3	$\mathbf{Id} + \mathbf{Id} \$$	Shift, goto 1
O 1 3 1	$+ \mathbf{Id} \$$	Reduce 4
O 1 3 5	$+ \mathbf{Id} \$$	Reduce 3
O 2	$+ \mathbf{Id} \$$	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	
	$\text{Id}$	$+$	$*$	$\$$	$e$	$t$
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
O	$\text{Id} * \text{Id} + \text{Id} \$$	Shift, goto 1
O 1	$* \text{Id} + \text{Id} \$$	Shift, goto 3
O 1 3	$\text{Id} + \text{Id} \$$	Shift, goto 1
O 1 3 1	$+ \text{Id} \$$	Reduce 4
O 1 3 5	$+ \text{Id} \$$	Reduce 3
O 2	$+ \text{Id} \$$	Shift, goto 4
O 2 4	$\text{Id} \$$	Shift, goto 1
O 2 4 1	$\$$	Reduce 4
O 2 4 2	$\$$	Reduce 2
O 2 4 6	$\$$	Reduce 1
O 7	$\$$	Accept

## L, R, and all that

LR parser: “Bottom-up parser”:

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

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 Ocamlacc 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 :  $b \rightarrow \text{Id Id}$

