Parser II

Ronghui Gu

Spring 2019

Columbia University

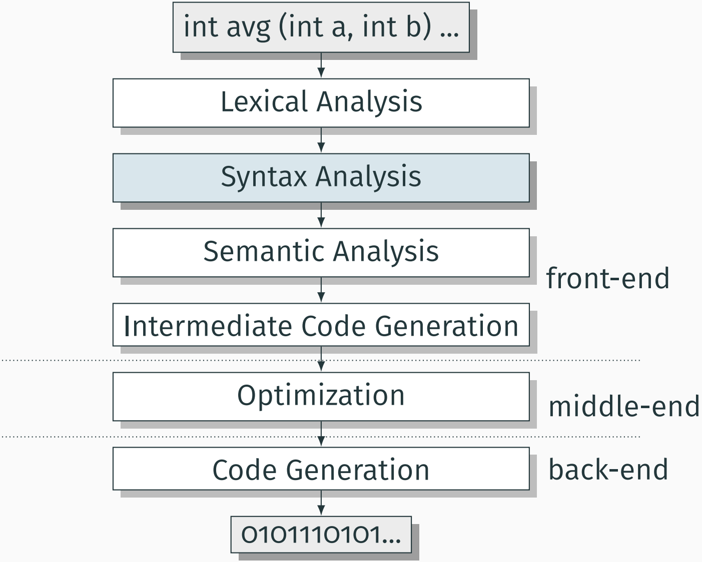
* Course website: https://www.cs.columbia.edu/ rgu/courses/4115/spring2019

** These slides are borrowed from Prof. Edwards.

Parsing

How do we combine words into sentences?

Parsing


int avg (int a, int b) ...
Lexical Analysis
Syntax Analysis
Semantic Analysis
front-end
Intermediate Code Generation
Optimization
middle-end
Code Generation
back-end
0101110101...

Solution: Context-Free Grammars

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

$S \rightarrow \text{Either } P\text{, or } P.$	Exactly two Ps
$S \rightarrow \text{If } P\text{, then } P.$	
$P \rightarrow A\ H\ N \text{ eats } O$	One each of A, H, N, and O
$A \rightarrow \text{the}$	
$A \rightarrow \text{a}$	
$A \rightarrow \text{every}$	
$H \rightarrow \text{happy } H$	H is "happy" zero or more times
$H \rightarrow \epsilon$	
$N \rightarrow \text{boy}$	
$N \rightarrow \text{girl}$	
$N \rightarrow \text{dog}$	
$O \rightarrow \text{hot dogs}$	
$O \rightarrow \text{ice cream}$	
$O \rightarrow \text{candy}$	

An Example

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

$$S \to 0\ S\ 1.$$
$$S \to \epsilon.$$

Constructing Grammars and Ocamlyacc

The Dangling Else Problem

Who owns the *else*?

$$\text{if (a) if (b) c(); else d();}$$

```
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

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 ...
```

$$\text{if (a) } \underbrace{\text{if (b) c();}}_{\text{cstmt?}} \text{ else d();}$$

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 ...
```

$$\text{if (a) } \underbrace{\text{if (b) c();}}_{\text{cstmt?}} \text{ else d();}$$

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.

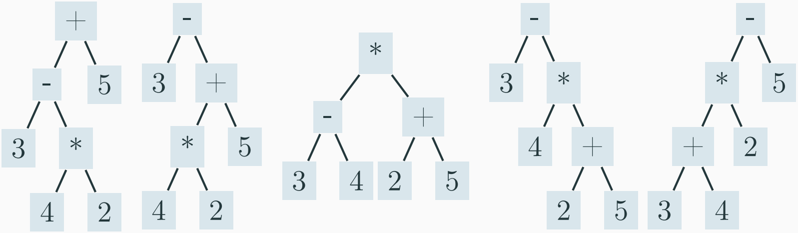
Ambiguous Arithmetic

Ambiguity can be a problem in expressions. Consider parsing

$$3 \text{ - } 4 * 2 + 5$$

with the grammar

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



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 $\underline{\text{expr PLUS}}$ *shift*?

$\underline{\text{expr TIMES expr}}$ 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 <u>atom TIMES</u> *cannot shift*!

term TIMES <u>atom</u> TIMES *cannot reduce*!

<u>term TIMES atom</u> 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 predecence and associtivity 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*, $\$1, \$2, \ldots$ 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 $\mathbf{Id} * \mathbf{Id} + \mathbf{Id}$

$$e$$

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

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

$1: e \rightarrow t + e$

$2: e \rightarrow t$

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

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

e

$\boxed{t \; + \; e}$

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 unambigious.

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

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

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

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 unambigious.

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

$1 : e \rightarrow t + e$

$2 : e \rightarrow t$

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

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

$$e$$
$$\boxed{t + e}$$
$$t + t$$
$$t + \boxed{\textbf{Id}}$$

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

$1 : e \rightarrow t + e$

$2 : e \rightarrow t$

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

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

$$e$$

$$\boxed{t + e}$$

$$t + \boxed{t}$$

$$t + \boxed{\textbf{Id}}$$

$$\boxed{\textbf{Id} * t} + \textbf{Id}$$

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 $\textbf{Id} * \textbf{Id} + \textbf{Id}$

$1 : e \rightarrow t + e$

$2 : e \rightarrow t$

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

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

$$e$$
$$\boxed{t + e}$$
$$t + \boxed{t}$$
$$t + \boxed{\textbf{Id}}$$
$$\boxed{\textbf{Id} * t} + \textbf{Id}$$
$$\textbf{Id} * \boxed{\textbf{Id}} + \textbf{Id}$$

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

$1 : e \to t + e$

$2 : e \to t$

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

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

$$
\begin{array}{c}
e \\
\boxed{t + e} \\
t + \boxed{t} \\
t + \boxed{\mathbf{Id}} \\
\boxed{\mathbf{Id} * t} + \mathbf{Id} \\
\mathbf{Id} * \boxed{\mathbf{Id}} + \mathbf{Id}
\end{array}
$$

At each step, expand the *rightmost* nonterminal.

nonterminal

“handle”: The right side of a production

$$e \to \underline{t + e} \to t + \underline{t} \to t + \underline{\mathbf{Id}} \to \underline{\mathbf{Id} * t} + \mathbf{Id} \to \mathbf{Id} * \underline{\mathbf{Id}} + \mathbf{Id}$$

Rightmost Derivation: What to Expand

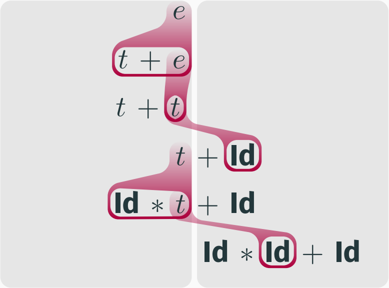
$1 : e \rightarrow t + e$

$2 : e \rightarrow t$

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

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

$$
\begin{array}{c}
e \\
t + e \\
t + t \\
t + \textbf{Id} \\
\textbf{Id} * t + \textbf{Id} \\
\textbf{Id} * \textbf{Id} + \textbf{Id}
\end{array}
$$



Expand here ↑ Terminals only

Reverse Rightmost Derivation

$1: e \rightarrow t + e$

$2: e \rightarrow t$

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

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

$$
\begin{array}{c}
e \\
\boxed{t + e} \\
t + \boxed{t} \\
t + \boxed{\textbf{Id}} \\
\boxed{\textbf{Id} * t} + \textbf{Id} \\
\textbf{Id} * \boxed{\textbf{Id}} + \textbf{Id}
\end{array}
$$

viable prefixes	terminals
	$\textbf{Id} * \textbf{Id} + \textbf{Id}$

Reverse Rightmost Derivation

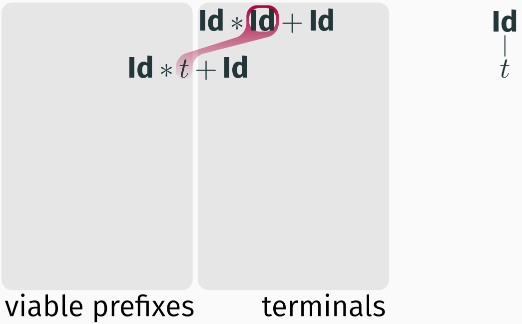
$1 : e \rightarrow t + e$

$2 : e \rightarrow t$

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

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

$$
\begin{array}{c}
e \\
t + e \\
t + t \\
t + \textbf{Id} \\
\textbf{Id} * t + \textbf{Id} \\
\textbf{Id} * \textbf{Id} + \textbf{Id}
\end{array}
$$



Reverse Rightmost Derivation

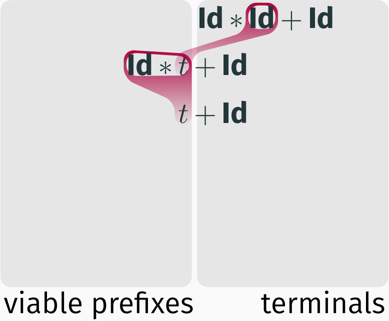
$1 : e \rightarrow t + e$

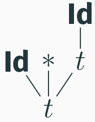
$2 : e \rightarrow t$

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

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

$$
\begin{array}{c}
e \\
\boxed{t + e} \\
t + \boxed{t} \\
t + \boxed{\textbf{Id}} \\
\boxed{\textbf{Id} * t} + \textbf{Id} \\
\textbf{Id} * \boxed{\textbf{Id}} + \textbf{Id}
\end{array}
$$





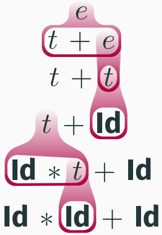
Reverse Rightmost Derivation

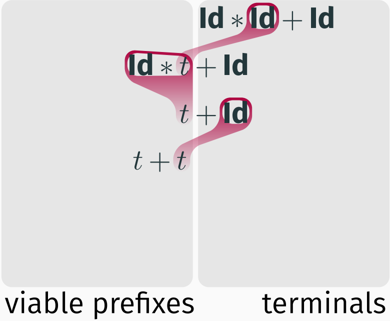
$1: e \rightarrow t + e$

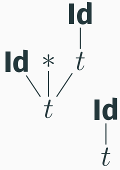
$2: e \rightarrow t$

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

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







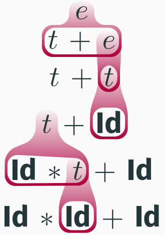
Reverse Rightmost Derivation

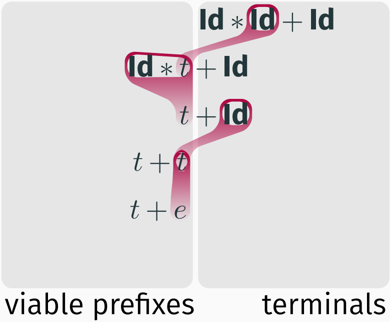
$1 : e \rightarrow t + e$

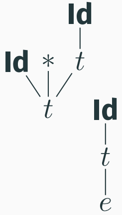
$2 : e \rightarrow t$

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

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







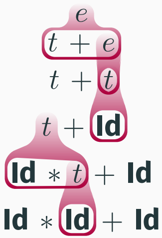
Reverse Rightmost Derivation

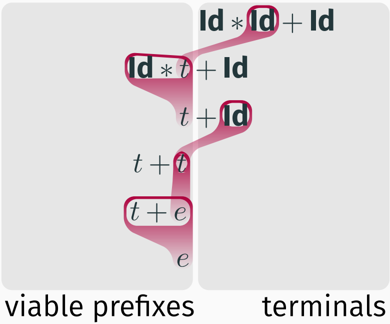
$1 : e \rightarrow t + e$

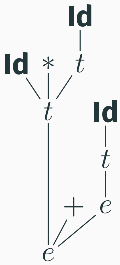
$2 : e \rightarrow t$

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

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







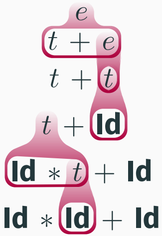
Shift/Reduce Parsing Using an Oracle

$1 : e \rightarrow t + e$

$2 : e \rightarrow t$

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

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



$\textbf{Id} * \textbf{Id} + \textbf{Id}$ shift

stack

input

Shift/Reduce Parsing Using an Oracle

$1: e \rightarrow t + e$

$2: e \rightarrow t$

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

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

e

$t + e$

$t + t$

$t + \mathbf{Id}$

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

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

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

Shift/Reduce Parsing Using an Oracle

$1 : e \to t + e$

$2 : e \to t$

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

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

$$e$$
$$\boxed{t + e}$$
$$t + \boxed{t}$$
$$t + \boxed{\textbf{Id}}$$
$$\boxed{\textbf{Id} * t} + \textbf{Id}$$
$$\textbf{Id} * \boxed{\textbf{Id}} + \textbf{Id}$$

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

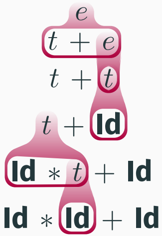
Shift/Reduce Parsing Using an Oracle

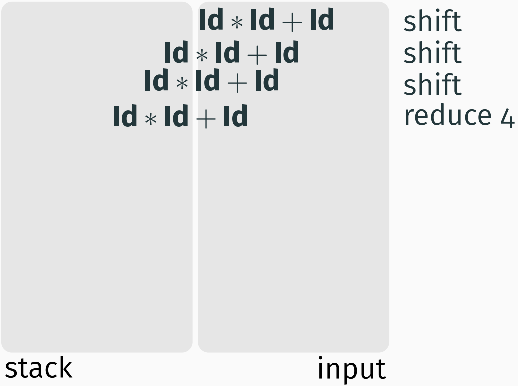
$1: e \rightarrow t + e$

$2: e \rightarrow t$

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

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





Shift/Reduce Parsing Using an Oracle

$1 : e \to t + e$

$2 : e \to t$

$3 : t \to \textbf{Id} \; * t$

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

e

$\boxed{t + e}$

$t + \boxed{t}$

$t + \boxed{\textbf{Id}}$

$\boxed{\textbf{Id} \; * \; t} + \textbf{Id}$

$\textbf{Id} \; * \boxed{\textbf{Id}} + \textbf{Id}$

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

Shift/Reduce Parsing Using an Oracle

$1: e \rightarrow t + e$

$2: e \rightarrow t$

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

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

$$
\begin{array}{c}
e \\
t + e \\
t + t \\
t + \textbf{Id} \\
\textbf{Id} * t + \textbf{Id} \\
\textbf{Id} * \textbf{Id} + \textbf{Id}
\end{array}
$$

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

Shift/Reduce Parsing Using an Oracle

$1: e \rightarrow t + e$

$2: e \rightarrow t$

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

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

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

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

Shift/Reduce Parsing Using an Oracle

$1 : e \rightarrow t + e$

$2 : e \rightarrow t$

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

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

$$
\begin{array}{c}
e \\
t + e \\
t + t \\
t + \textbf{Id} \\
\textbf{Id} * t + \textbf{Id} \\
\textbf{Id} * \textbf{Id} + \textbf{Id}
\end{array}
$$

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

$1 : e \rightarrow t + e$

$2 : e \rightarrow t$

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

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

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

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

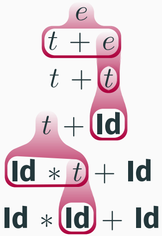
Shift/Reduce Parsing Using an Oracle

$1 : e \rightarrow t + e$

$2 : e \rightarrow t$

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

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



stack	input	
	$\textbf{Id} * \textbf{Id} + \textbf{Id}$	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
$t + t$		reduce 2
$t + e$		reduce 1

Shift/Reduce Parsing Using an Oracle

$1 : e \rightarrow t + e$

$2 : e \rightarrow t$

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

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

$$e$$
$$\boxed{t + e}$$
$$t + \boxed{t}$$
$$t + \boxed{\textbf{Id}}$$
$$\boxed{\textbf{Id} * t} + \textbf{Id}$$
$$\textbf{Id} * \boxed{\textbf{Id}} + \textbf{Id}$$

stack	input	
	$\textbf{Id} * \textbf{Id} + \textbf{Id}$	shift
$\textbf{Id}$	$* \textbf{Id} + \textbf{Id}$	shift
$\textbf{Id} *$	$\textbf{Id} + \textbf{Id}$	shift
$\textbf{Id} * \boxed{\textbf{Id}}$	$+ \textbf{Id}$	reduce 4
$\boxed{\textbf{Id} * t}$	$+ \textbf{Id}$	reduce 3
t	$+ \textbf{Id}$	shift
$t +$	$\textbf{Id}$	shift
$t + \boxed{\textbf{Id}}$		reduce 4
$t + \boxed{t}$		reduce 2
$\boxed{t + e}$		reduce 1
e		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.*

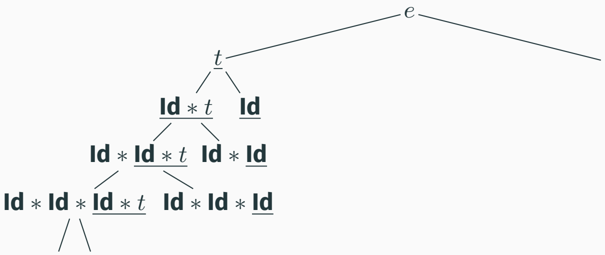
Some Right-Sentential Forms and Their Handles

$1 : e \rightarrow t + e$

$2 : e \rightarrow t$

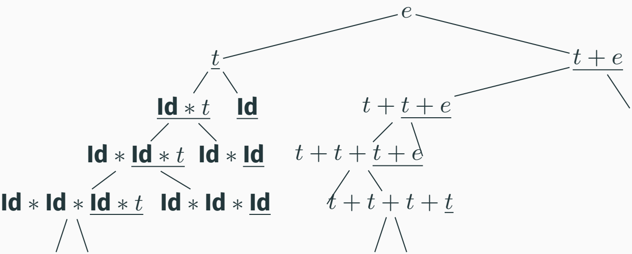
$3 : t \rightarrow \textbf{Id} \ * t$

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



Some Right-Sentential Forms and Their Handles

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



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:

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

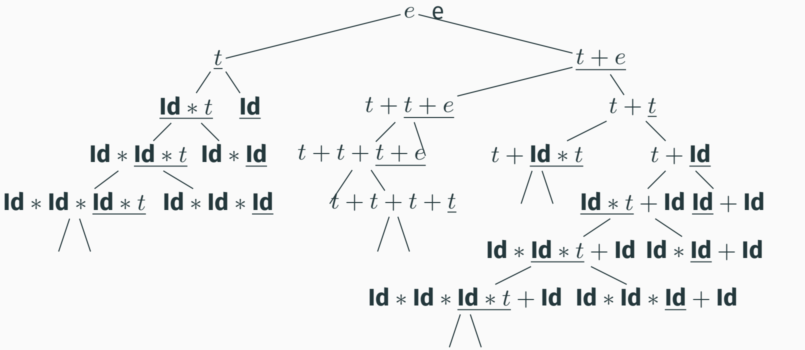
$\textbf{Id} * \textbf{Id} * \cdots * \underline{\textbf{Id}} \cdots$

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

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

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

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



The Handle-Identifying Automaton

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

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

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

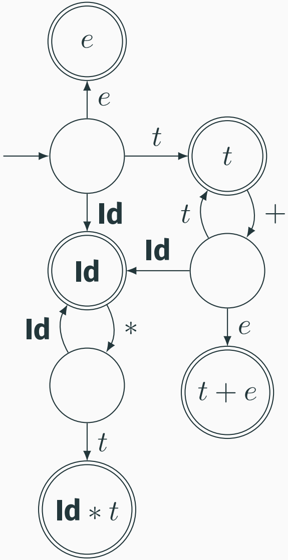
$t + t + \cdots + \underline{t + e}$

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

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

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

e



Building the Initial State of the LR(0) Automaton

$1 : e \to t + e$

$2 : e \to t$

$3 : t \to \textbf{Id} \; * \, t$

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

$$\boxed{e' \to \bullet e}$$

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 "$e' \to \bullet e$"

Building the Initial State of the LR(0) Automaton

$1 : e \rightarrow t + e$

$2 : e \rightarrow t$

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

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

$$\boxed{\begin{array}{l} e' \rightarrow \bullet e \\ e \rightarrow \bullet t + e \\ e \rightarrow \bullet t \\ \\ \\ \end{array}}$$

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

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

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

$1 : e \rightarrow t + e$

$2 : e \rightarrow t$

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

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

$$\boxed{\begin{array}{l} e' \rightarrow \bullet e \\ e \rightarrow \bullet t + e \\ e \rightarrow \bullet t \\ t \rightarrow \bullet \textbf{Id} * t \\ t \rightarrow \bullet \textbf{Id} \end{array}}$$

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

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

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.

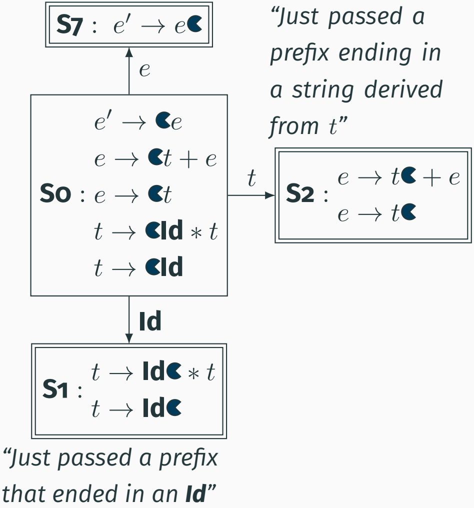
Also, t must be **Id** $* \, t$ or **Id**, so $t \rightarrow \bullet \textbf{Id} * t$ and $t \rightarrow \bullet \textbf{Id}$.

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

Building the LR(0) Automaton

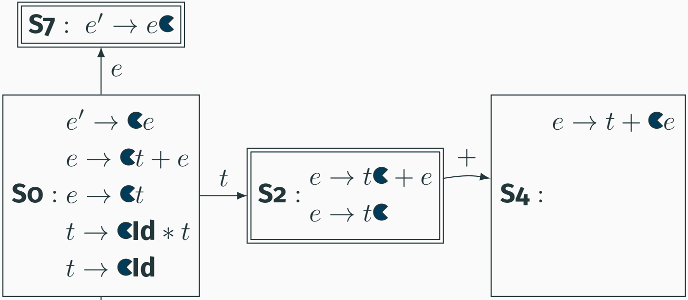
$$
\boxed{\begin{array}{rl}
 & e' \to \bullet e \\
 & e \to \bullet t + e \\
\textbf{S0} : & e \to \bullet t \\
 & t \to \bullet \textbf{Id} * t \\
 & t \to \bullet \textbf{Id}
\end{array}}
$$

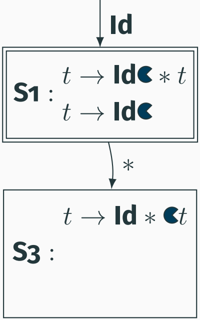
The first state suggests a viable prefix can start as any string derived from e, any string derived from t, or **Id**.



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

Building the LR(0) Automaton

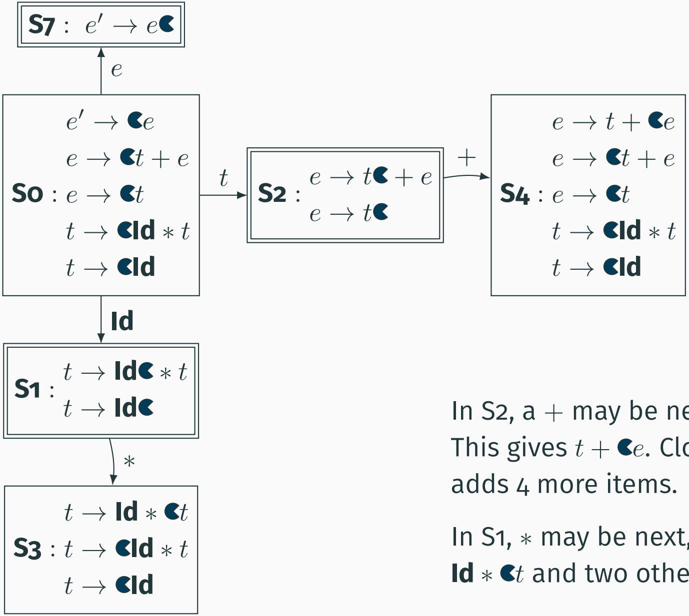




In S2, a $+$ may be next.
This gives $t + \bullet e$.

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

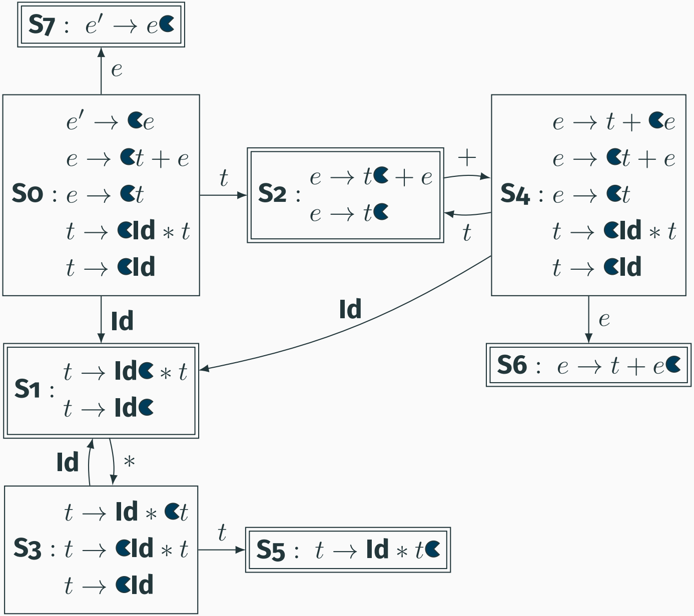
Building the LR(0) Automaton



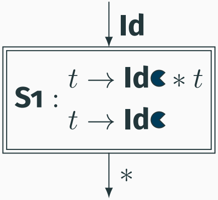
In S2, a $+$ may be next. This gives $t + \bullet e$. Closure adds 4 more items.

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

Building the LR(0) Automaton



What to do in each state?



$1 : e \to t + e$

$2 : e \to t$

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

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

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

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

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

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

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

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

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

The FIRST function

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

1. If X is a terminal, FIRST$(X) = \{X\}$.
2. If $X \to \epsilon$, then add ϵ to FIRST(X).
3. If $X \to Y_1 \cdots Y_k$ and $\epsilon \in$ FIRST(Y_1), $\epsilon \in$ FIRST(Y_2), …, and $\epsilon \in$ FIRST(Y_{i-1}) for $i = 1, \ldots, k$ for some k,
   add FIRST$(Y_i) - \{\epsilon\}$ to FIRST(X)
   X starts with anything that appears after skipping empty strings.
   Usually just FIRST$(Y_1) \subset$ FIRST(X)
4. If $X \to Y_1 \cdots Y_K$ and $\epsilon \in$ FIRST(Y_1), $\epsilon \in$ FIRST(Y_2), …, and $\epsilon \in$ FIRST(Y_k), add ϵ to FIRST(X)
   If all of X can be empty, X can be empty

$1 : e \to t + e$

$2 : e \to t$

$3 : t \to \textbf{Id} \; * t$

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

FIRST(**Id**) $= \{$**Id**$\}$

FIRST$(t) = \{$**Id**$\}$ because $t \to \textbf{Id} \; * t$ and $t \to \textbf{Id}$

FIRST$(e) = \{$**Id**$\}$ because $e \to t + e$, $e \to t$, and FIRST$(t) = \{$**Id**$\}$.

First and ϵ

$\epsilon \in$ FIRST(α) means α can derive the empty string.

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

$$\begin{aligned} X &\rightarrow YZa \\ Y &\rightarrow \\ Y &\rightarrow b \\ Z &\rightarrow c \\ Z &\rightarrow W \\ W &\rightarrow \\ W &\rightarrow d \end{aligned}$$

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

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

---

$1 : e \rightarrow t + e$

$2 : e \rightarrow t$

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

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

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

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

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

FOLLOW$(t) = \{ \qquad \}$

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 $t \in$ 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$(t) = \{ + \quad \}$

2. *Because* $e \rightarrow \underline{t} + e$ *and* 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

---

$1 : e \rightarrow t + e$

$2 : e \rightarrow t$

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

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

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

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

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

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 $t \in$ 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

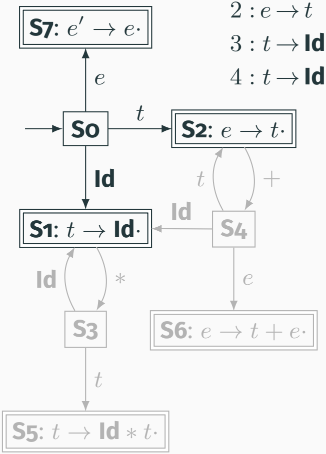
Converting the LR(0) Automaton to an SLR Table

$1: e \to t + e$

$2: e \to t$

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

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



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

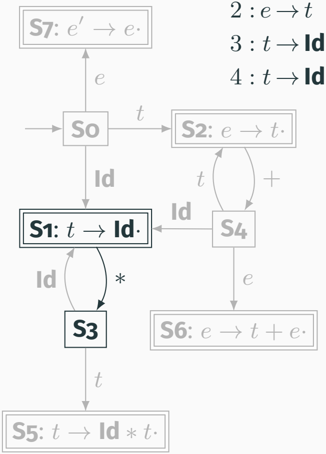
FOLLOW$(t) = \{+, \$\}$

State	Action				Goto	
	Id	$+$	$*$	$	e	t
0	S1				7	2

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

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



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

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

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

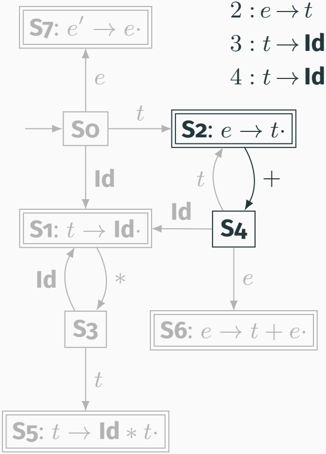
Converting the LR(0) Automaton to an SLR Table

$1 : e \rightarrow t + e$

$2 : e \rightarrow t$

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

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

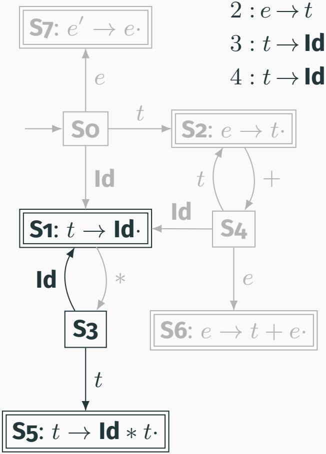


FOLLOW$(e) = \{\$\}$
FOLLOW$(t) = \{+, \$\}$

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



$1 : e \rightarrow t + e$

$2 : e \rightarrow t$

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

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

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

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

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

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

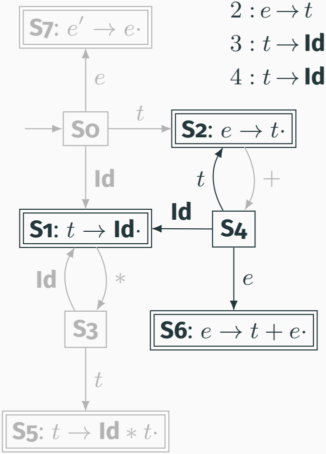
Converting the LR(0) Automaton to an SLR Table

$1: e \to t + e$

$2: e \to t$

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

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



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

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

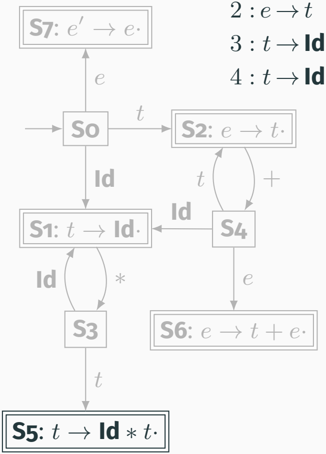
Converting the LR(0) Automaton to an SLR Table

$1 : e \rightarrow t + e$

$2 : e \rightarrow t$

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

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



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 $\in$ FOLLOW(t).

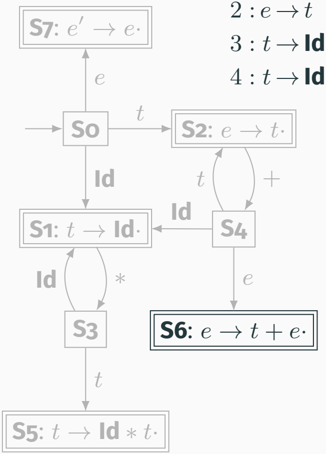
Converting the LR(0) Automaton to an SLR Table

$1 : e \rightarrow t + e$

$2 : e \rightarrow t$

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

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



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		

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

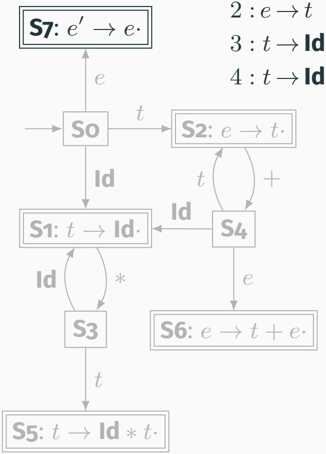
Converting the LR(0) Automaton to an SLR Table

$1 : e \rightarrow t + e$

$2 : e \rightarrow t$

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

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



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.

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/Reduce Parsing with an SLR Table

$1 : e \to t + e$

$2 : e \to t$

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

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

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

$1 : e \rightarrow t + e$

$2 : e \rightarrow t$

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

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

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	$+$ **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

$1 : e \rightarrow t + e$

$2 : e \rightarrow t$

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

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

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	$\textbf{Id} * \textbf{Id} + \textbf{Id}\ \$$	Shift, goto 1
0 **Id** 1	$* \textbf{Id} + \textbf{Id}\ \$$	Shift, goto 3
0 **Id** 1 $*$ 3	$\textbf{Id} + \textbf{Id}\ \$$	Shift, goto 1
0 **Id** 1 $*$ 3 **Id** 1	$+ \textbf{Id}\ \$$	Reduce 4
0 **Id** 1 $*$ 3 t 5	$+ \textbf{Id}\ \$$	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

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 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

$t \rightarrow \cdot$

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** $* \, t \, \cdot$

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

A Reduce/Reduce Conflict

$1 : a \rightarrow \textbf{Id Id}$

$2 : a \rightarrow b$

$3 : b \rightarrow \textbf{Id Id}$