Lexical Analysis
&
Parsing

15-411/15-611 Compiler Design

Seth Copen Goldstein

October 1, 2020

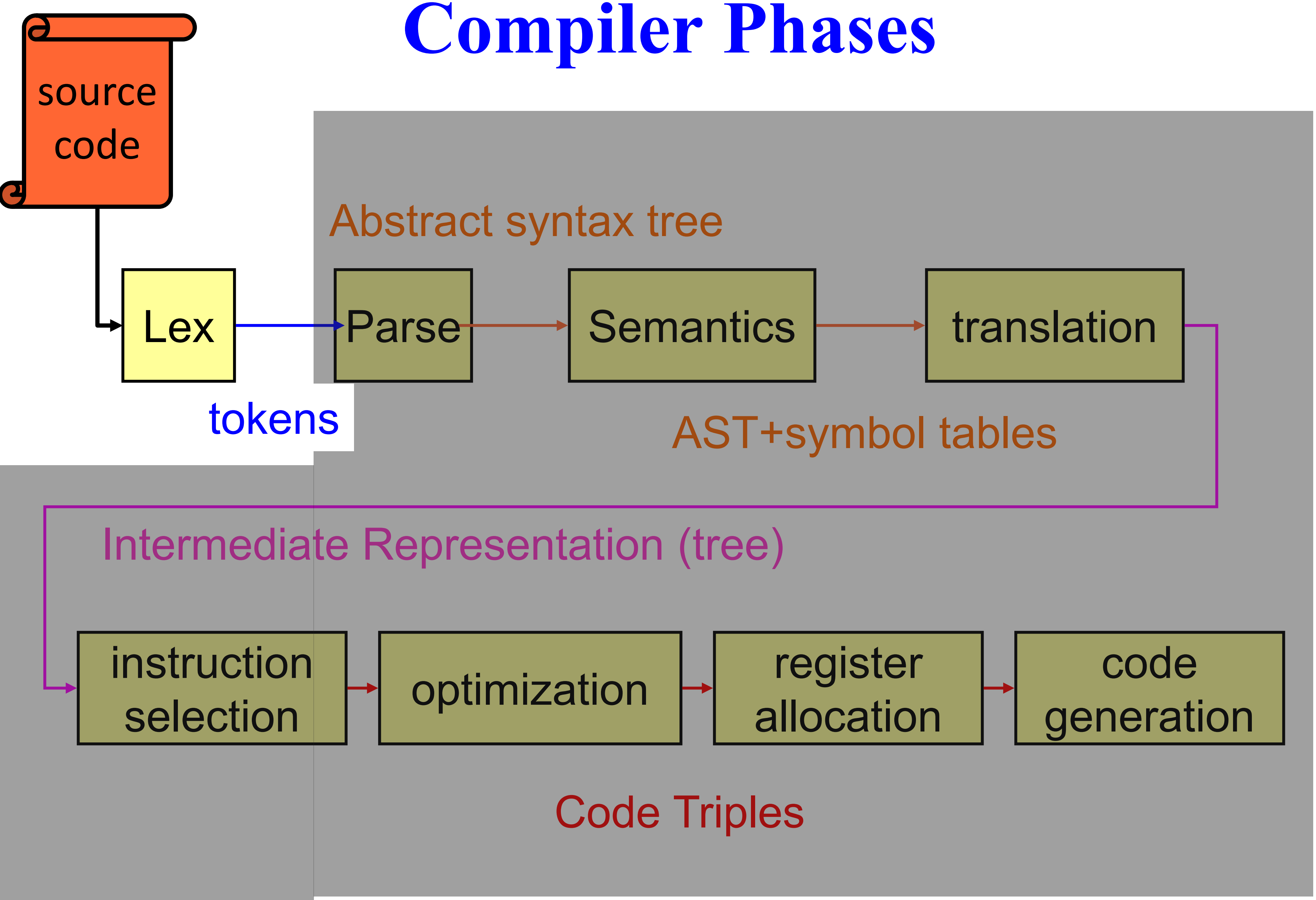
Today

- Lexing
- Parsing

Today – part 1

- Lexing
- Flex & other scanner generators
- Regular Expressions
- Finite Automata
- RE → NFA
- NFA → DFA
- DFA → Minimized DFA
- Limits of Regular Languages

Compiler Phases



source code

Lex

tokens

Abstract syntax tree

Parse

Semantics

translation

AST+symbol tables

Intermediate Representation (tree)

instruction selection

optimization

register allocation

code generation

Code Triples

The Lexer

- Turn stream of characters into a stream of tokens

```
// create a user friendly descriptor for this arg.
// if key is absent, then use it.  Otherwise use longkey

char*
ArgDesc::helpkey(WhichKey keytype, bool includebraks)
{
    static char buffer[128];  /* format buffer */
    char* p = buffer;
    …
```

```
CHAR STAR ID DOUBLE_COLON ID LPARIN ID ID COMMA BOOL ID
RPARIN LBRACE STATIC CHAR ID LBRAK INTCONST RBRAK SEMI
CHAR STAR ID EQ ID SEMI …
```

The Lexer

- Turn stream of characters into a stream of tokens
  - Strips out “unnecessary characters”
    - comments
    - whitespace
  - Classify tokens by type
    - keywords
    - numbers
    - punctuation
    - identifiers
  - Track location
  - Associate with syntactic information

The Lexer

- Turn stream of characters into a stream of tokens

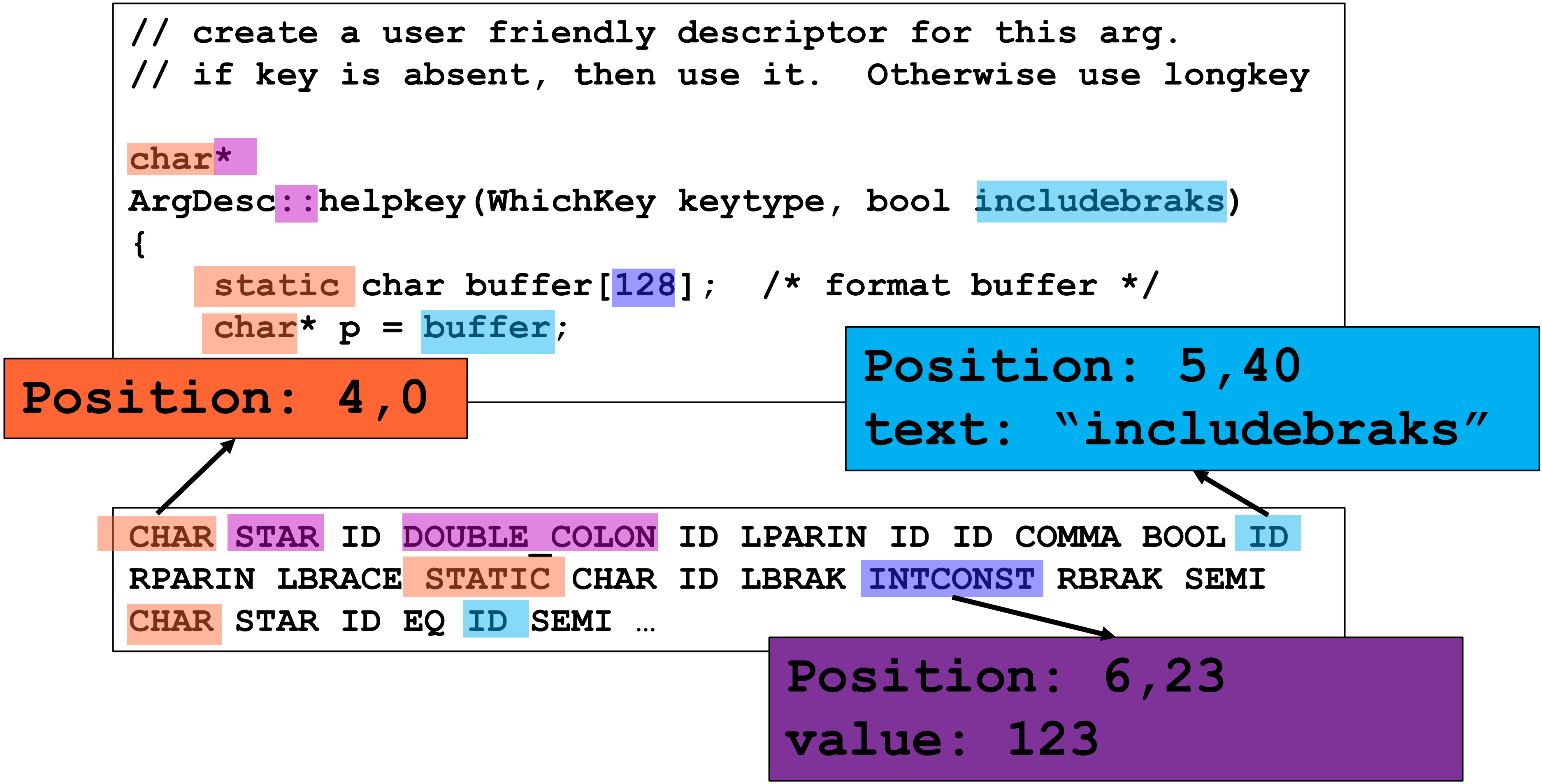
```
// create a user friendly descriptor for this arg.
// if key is absent, then use it.  Otherwise use longkey

char*
ArgDesc::helpkey(WhichKey keytype, bool includebraks)
{
    static char buffer[128];  /* format buffer */
    char* p = buffer;
    …
```

```
CHAR STAR ID DOUBLE_COLON ID LPARIN ID ID COMMA BOOL ID
RPARIN LBRACE STATIC CHAR ID LBRAK INTCONST RBRAK SEMI
CHAR STAR ID EQ ID SEMI …
```

The Lexer

- Turn stream of characters into a stream of tokens



```
// create a user friendly descriptor for this arg.
// if key is absent, then use it.  Otherwise use longkey

char*
ArgDesc::helpkey(WhichKey keytype, bool includebraks)
{
    static char buffer[128];  /* format buffer */
    char* p = buffer;
```

```
CHAR STAR ID DOUBLE_COLON ID LPARIN ID ID COMMA BOOL ID
RPARIN LBRACE STATIC CHAR ID LBRAK INTCONST RBRAK SEMI
CHAR STAR ID EQ ID SEMI …
```

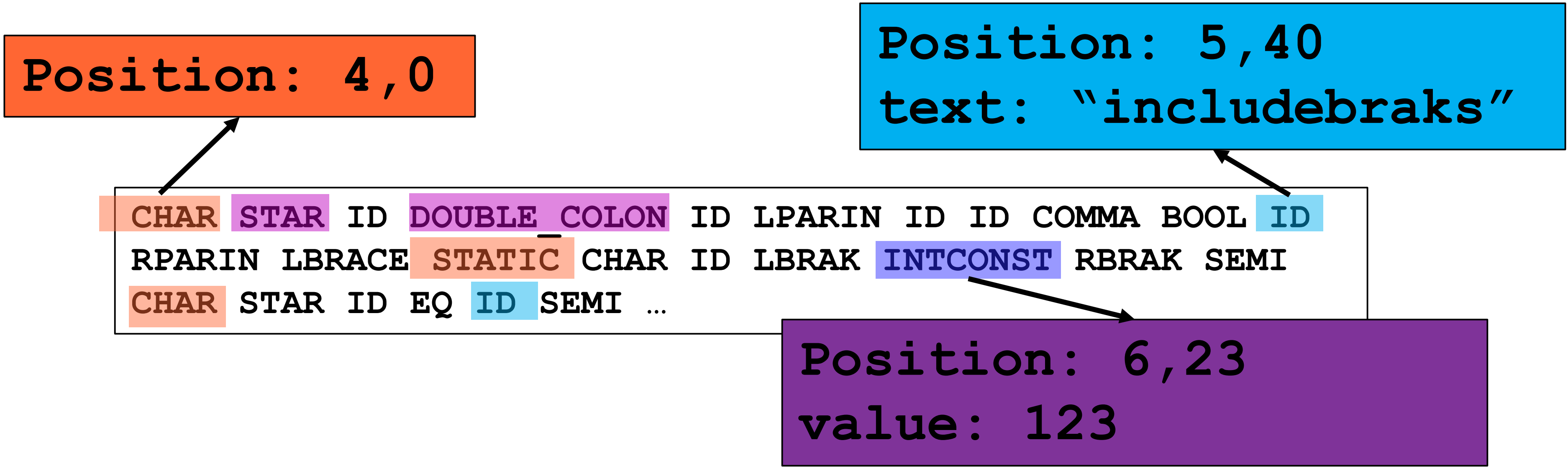
Position: 4,0

Position: 5,40
text: “includebraks”

Position: 6,23
value: 123

The Lexer

- Turn stream of characters into a stream of tokens
  - More concise
  - Easier to parse



Position: 4,0

Position: 5,40
text: “includebraks”

```
CHAR STAR ID DOUBLE_COLON ID LPARIN ID ID COMMA BOOL ID
RPARIN LBRACE STATIC CHAR ID LBRAK INTCONST RBRAK SEMI
CHAR STAR ID EQ ID SEMI …
```

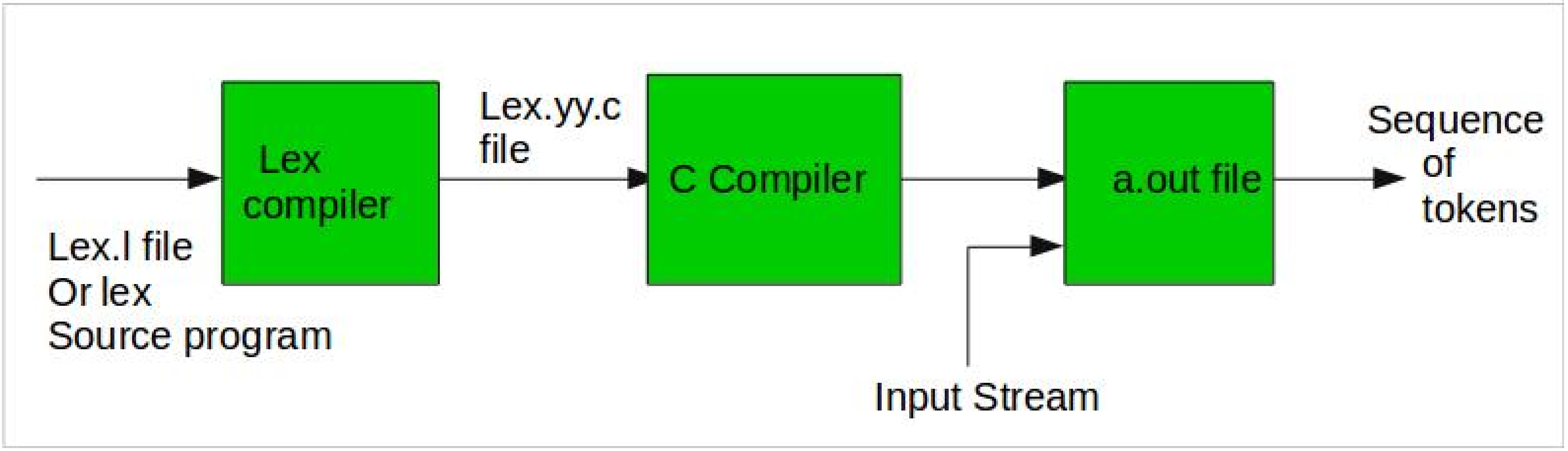
Position: 6,23
value: 123

Lexical Analyzers

- Input: stream of characters
- Output: stream of tokens (with information)
- How to build?
  - By hand is tedious
  - Use Lexical Analyzer Generator, e.g., flex
- Define tokens with regular expressions
- Flex turns REs into Deterministic Finite Automata (DFA) which recognizes and returns tokens.

FLEX

- Define tokens
- Generate scanner code
- Main interface: **`yylex()`** which reads from **yyin** and returns tokens til EOF



© 2019-20 Goldstein

2. Flex Program Format

- A flex program has three sections:

  Definitions
  %%
  RE rules & actions
  %%
  User code

wc As a Flex Program

```
%{
  int charCount=0, wordCount=0, lineCount=0;
%}
word   [^ \t\n]+
%%
{word} {wordCount++; charCount += yyleng; }
[\n]   {charCount++; lineCount++;}
.      {charCount++;}
%%
int main(void) {
   yylex();
   printf(“Chars %d, Words: %d, Lines: %d\n”,
      charCount,  wordCount,  lineCount);
   return 0;
}
```

A Flex Program

```
%{
   int charCount=0, wordCount=0, lineCount=0;
%}
word    [^ \t\n]+
```

1) Definitions

```
%%
{word} {wordCount++; charCount += yyleng; }
[\n]   {charCount++; lineCount++;}
.      {charCount++;}
```

2) Rules & Actions

```
%%
int main(void) {
   yylex();
   printf(“Chars %d, Words: %d, Lines: %d\n”,
      charCount,  wordCount,  lineCount);
   return 0;
}
```

3) User Code

skip

Section 1: RE Definitions

- Format:

```
          name      RE
```

- Examples:

```
digit     [0-9]
letter    [A-Za-z]
id        {letter} ({letter}|{digit})*

word      [^ \t\n]+
```

Regular Expressions in Flex

`x`	match the char **x**
`\.`	match the char **.**
`"string"`	match contents of string of chars
`.`	match any char except \n
`^`	match beginning of a line
`$`	match the end of a line
`[xyz]`	match one char **x**, **y**, or **z**
`[^xyz]`	match any char except **x**, **y**, and **z**
`[a-z]`	match one of **a** to **z**

Regular Expressions in Flex (cont)

`r*`	closure (match 0 or more r's)
`r+`	positive closure (match 1 or more r's)
`r?`	optional (match 0 or 1 r)
`r1 r2`	match *r*1 then *r*2 (concatenation)
`r1 | r2`	match *r*1 or *r*2 (union)
`( r )`	grouping
`r1 \ r2`	match *r*1 when followed by *r*2
{ *name* }	match the RE defined by name

Some number REs

`[0-9]`	A single digit.
`[0-9]+`	An integer.
`[0-9]+ (\.[0-9]+)?`	An integer or fp number.
`[+-]? [0-9]+ (\.[0-9]+)? ([eE][+-]?[0-9]+)?`	Integer, fp, or scientific notation.

Section 2: RE/Action Rule

- A rule has the form:

```
name       { action }
re         { action }
```

  - the name must be defined in section 1
  - the action is any C code

- If the named RE matches* an input character sequence, then the C code is executed.

* Some caveats here

Rule Matching

- Longest match rule.

```
“int”      { return INT; }
“integer”  { return INTEGER; }
```

- If rules can match same length input, first rule takes priority.

```
“int”      { return INT; }
[a-z]+     { return ID; }
[0-9]+     { return NUM; }
```

Section 3: C Functions

- Added to end of the lexical analyzer

Removing Whitespace

```
whitespace          [ \t\n]
%%

{whitespace}        ;

.                   { ECHO; }

%%


int main(void)
{
  yylex();
  return 0;
}
```

name

empty action

RE

ECHO macro

Printing Line Numbers

```
%{
   int lineno = 1;
%}
%%
^(.*)\n    { printf("%4d\t%s", lineno, yytext);
             lineno++;}
%%
int main(int argc, char *argv[])
{
   // appropriate arg processing & error
   handling, …
   yyin = fopen(argv[1], "r");
   yylex();
   return 0;
}
```

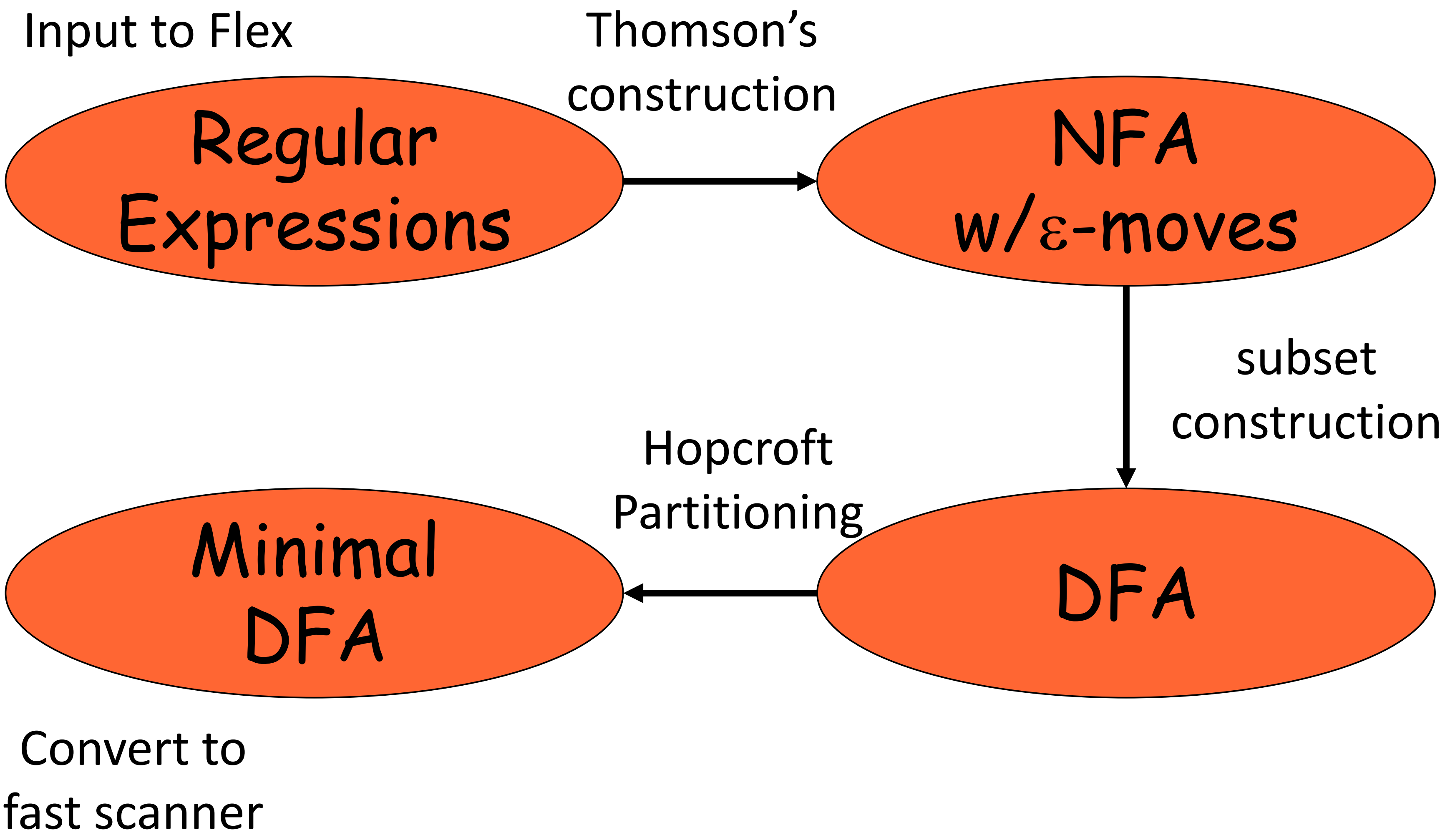
the matched text

Today – part 1

- Lexing
- Flex & other scanner generators
- **Regular Expressions**
- Finite Automata
- RE → NFA
- NFA → DFA
- DFA → Minimized DFA
- Limits of Regular Languages

Under The Covers

- How to go from REs to a working scanner?



Input to Flex

Regular Expressions

Thomson’s construction

NFA w/ε-moves

subset construction

DFA

Hopcroft Partitioning

Minimal DFA

Convert to fast scanner

Regular Languages

- Finite Alphabet, Σ, of symbols.
- word (or string), a finite sequence of symbols from Σ.
- Language over Σ is a set of words from Σ.
- Regular Expressions describe Regular Languages.
  - easy to write down, but hard to use directly
- The languages accepted by Finite Automata are also Regular.

Regular Expressions defined

- Base Cases:
  - A single character a
  - The empty string ε
- Recursive Rules:
  If R_1 and R_2 are regular expressions
  - Concatenation R_1R_2
  - Union $R_1|R_2$
  - Closure R_1*
  - Grouping (R_1)
- REs describe Regular Languages.

RE Examples

- even a’s

- odd b’s

- even a’s or odd b’s
- even a’s followed by odd b’s

RE Examples

- even a’s

  b* (a b* a b*)*

- odd b’s

  a* b a* (b a* b a*)*

- even a’s or odd b’s
- even a’s followed by odd b’s

RE Examples

- even a’s

$$R^A = b^* (a b^* a b^*)^*$$

- odd b’s

$$R^B = a^* b a^* (b a^* b a^*)^*$$

- even a’s or odd b’s

$$R^A | R^B$$

- even a’s followed by odd b’s

$$R^A R^B$$

Today – part 1

- Lexing
- Flex & other scanner generators
- Regular Expressions
- **Finite Automata**
- **RE → NFA**
- NFA → DFA
- DFA → Minimized DFA
- Limits of Regular Languages

Finite Automata

- finite set of states
- set of edges from states to states labeled by letter from Σ
- initial state
- set of accepting states
- How it works:
  - Start in initial state, on each character transition goto state using edge labeled for that character.
  - If at end of word we are in accepting state, the word is in language
  - Language accepted are strings that cause FA to end in an accepting state

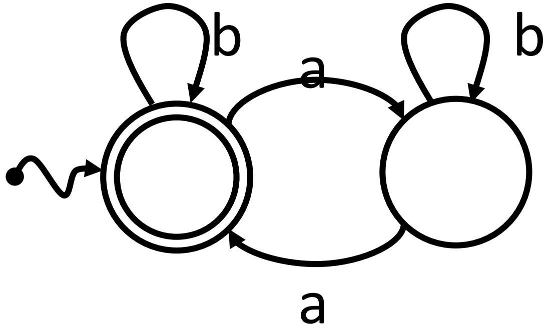
Example REs → FA

- even a’s b* (a b* a b*)*

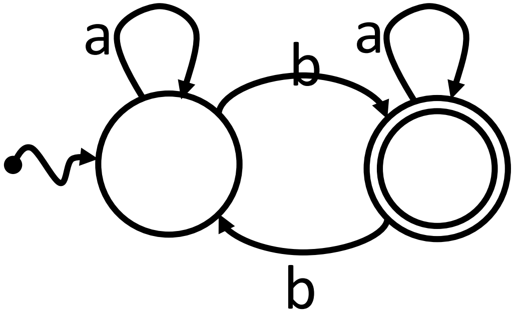
- odd b’s a* b a* (b a* b a*)*

Example REs → FA

- even a's b* (a b* a b*)*



- odd b's a* b a* (b a* b a*)*

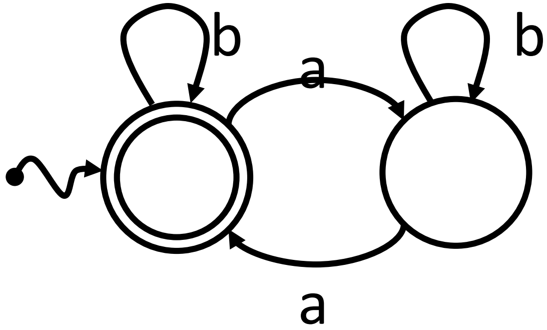


Deterministic Finite Automata
DFA

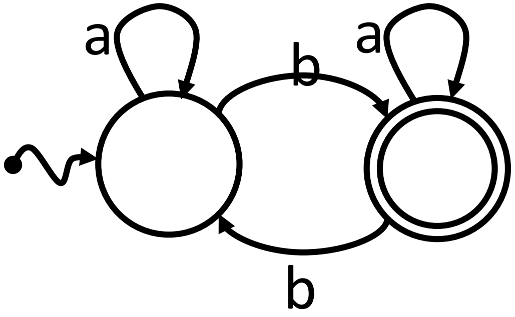
Ad Hoc

Example REs → FA

- even a’s b* (a b* a b*)*



- odd b’s a* b a* (b a* b a*)*



- even a’s or odd b’s

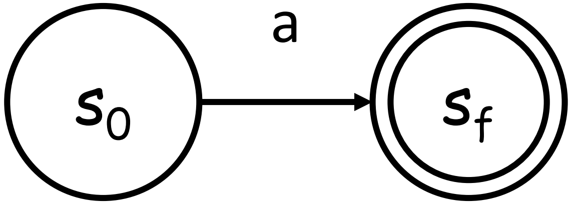
$R^A \mid R^B$

- even a’s followed by odd b’s

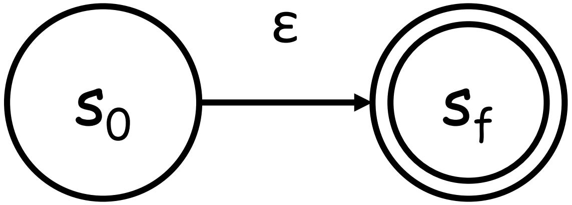
$R^A \; R^B$

Converting RE to NFA: Base Case

- for $a \in \Sigma$ the NFA $M_a = \{\Sigma, \{s_0, s_f\}, \delta, s_0, \{s_f\}\}$

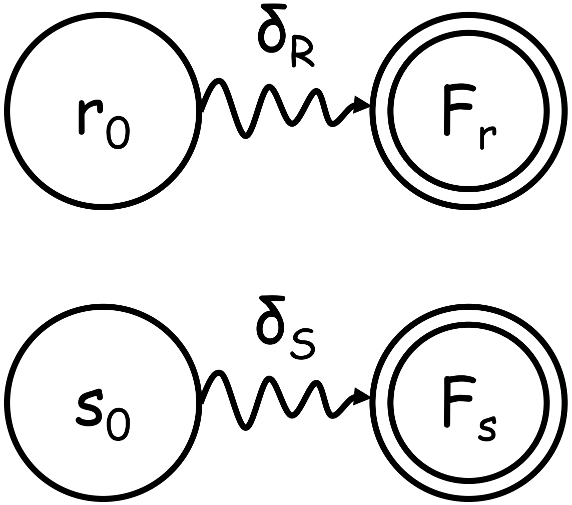


- for ε the NFA $M_\varepsilon = \{\Sigma, \{s_0, s_f\}, \delta, s_0, \{s_f\}\}$



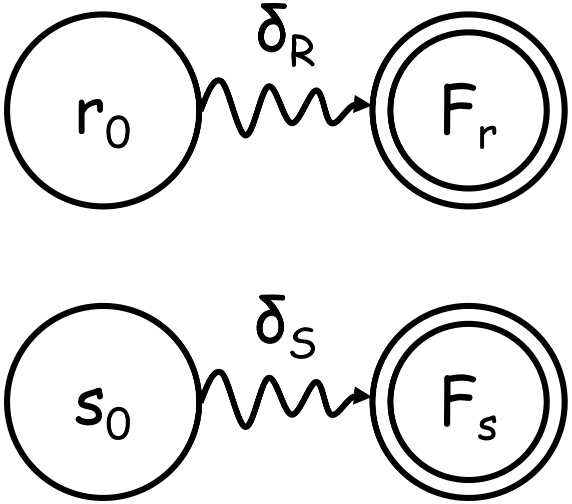
Recursive Case

- for RE R with $M_R = \{\Sigma, s_R, \delta_R, r_0, F_r\}$ and RE S with $M_s = \{\Sigma, s_S, \delta_S, s_0, F_s\}$



R|S

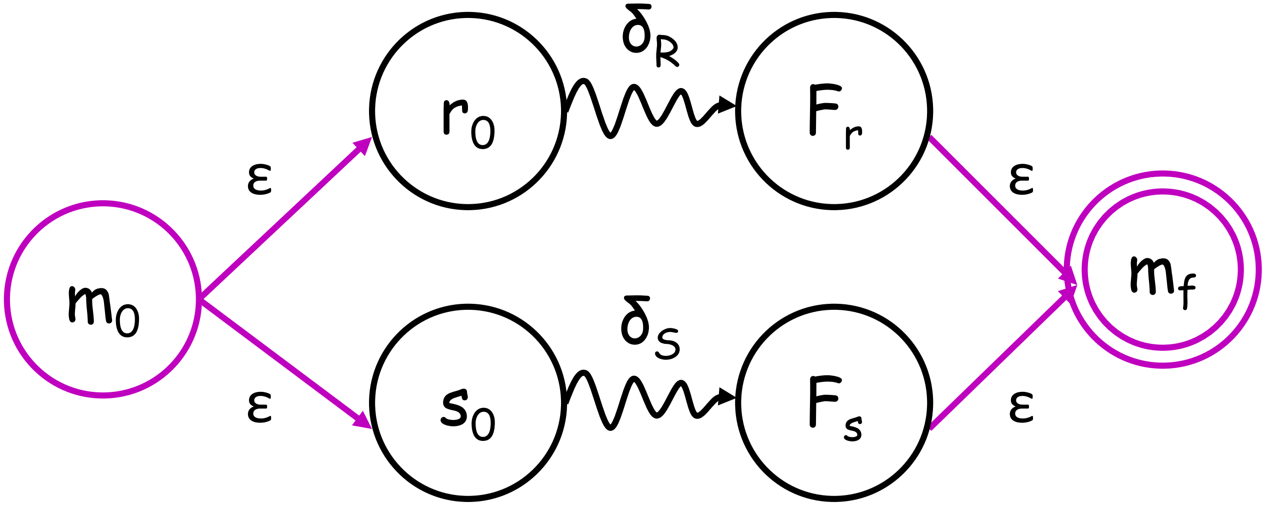
- for RE R with $M_R = \{\Sigma, s_R, \delta_R, r_0, F_r\}$ and RE S with $M_s = \{\Sigma, s_S, \delta_S, s_0, F_s\}$



- $M_{R|S} = \{\Sigma, s_R \cup s_s \cup \{m_0 , m_f\}, \delta_{R|S}, m_0, m_f\}$

R|S

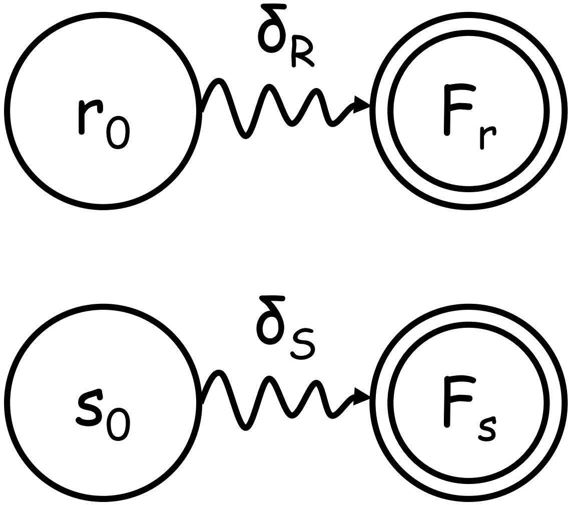
- for RE R with $M_R = \{\Sigma, s_R, \delta_R, r_0, F_r\}$ and RE S with $M_s = \{\Sigma, s_S, \delta_S, s_0, F_s\}$



- $M_{R|S} = \{\Sigma, s_R \cup s_R \cup \{m_0, m_f\}, \delta_{R|S}, m_0, m_f\}$

RS

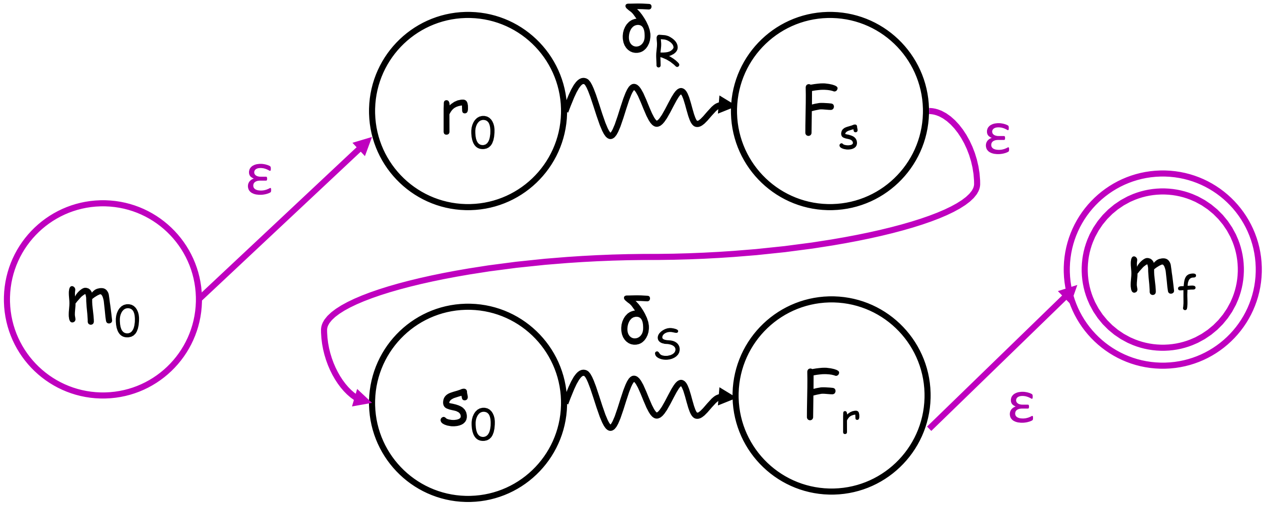
- for RE R with $M_R = \{\Sigma, s_R, \delta_R, r_0, F_r\}$ and RE S with $M_s = \{\Sigma, s_S, \delta_S, s_0, F_s\}$



- $M_{RS} = \{\Sigma, s_R \cup s_R \cup \{m_0 , m_f\}, \delta_{RS}, m_0, m_f\}$

RS

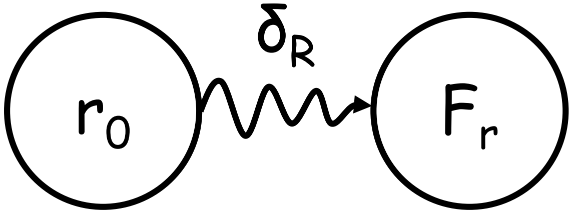
- for RE R with $M_R = \{\Sigma, s_R, \delta_R, r_0, F_r\}$ and RE S with $M_s = \{\Sigma, s_S, \delta_S, s_0, F_s\}$



- $M_{RS} = \{\Sigma, s_R \cup s_R \cup \{m_0 , m_f\}, \delta_{RS}, m_0, m_f\}$

R*

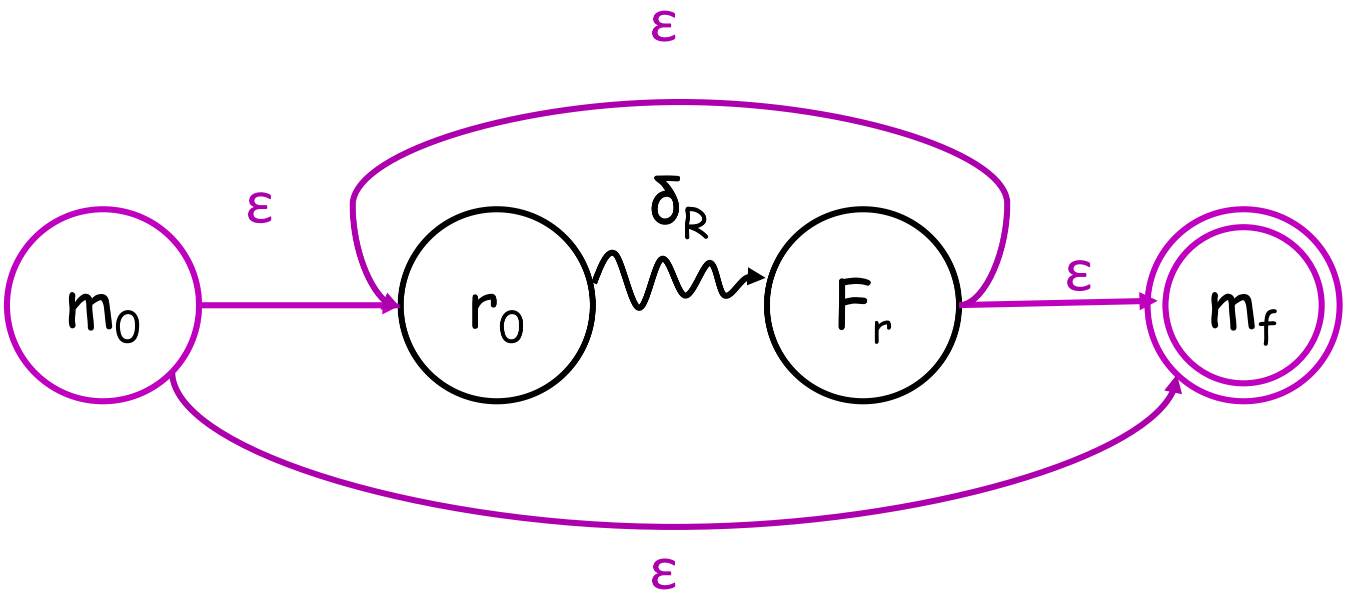
- for RE R with $M_R = \{\Sigma, s_R, \delta_R, r_0, F_r\}$



- $M_{R*} = \{\Sigma, s_R \cup \{m_0 , m_f\}, \delta_{R*}, m_0, m_f\}$

R*

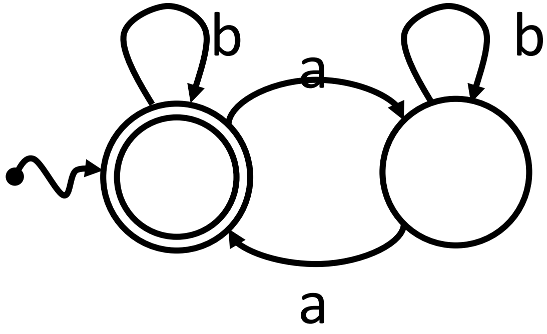
- for RE R with $M_R = \{\Sigma, s_R, \delta_R, r_0, F_r\}$



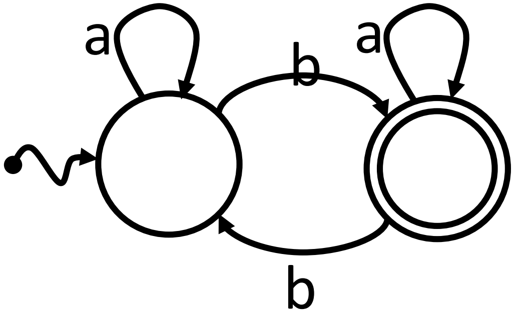
- $M_{R*} = \{\Sigma, s_R \cup \{m_0 , m_f\}, \delta_{R*}, m_0, m_f\}$

Example REs → FA

- even a’s   b* (a b* a b*)*



- odd b’s   a* b a* (b a* b a*)*



- even a’s or odd b’s

  R^A | R^B

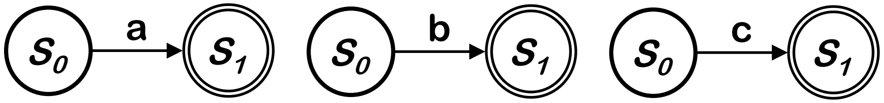
- even a’s followed by odd b’s

  $R^A\ R^B$

Example of Thompson’s Construction

Let’s try a (b | c)*

1. a, b, & c



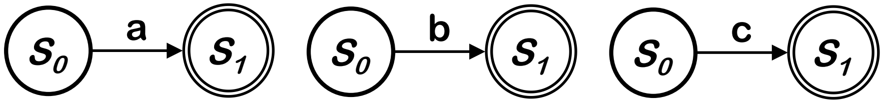
2. b | c

3. (b | c)*

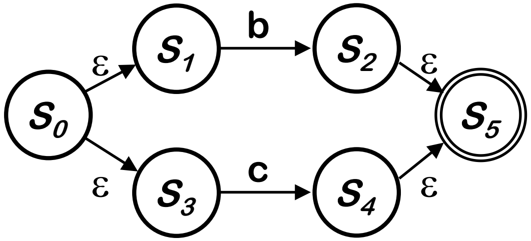
Example of Thompson's Construction

Let's try a (b | c)*

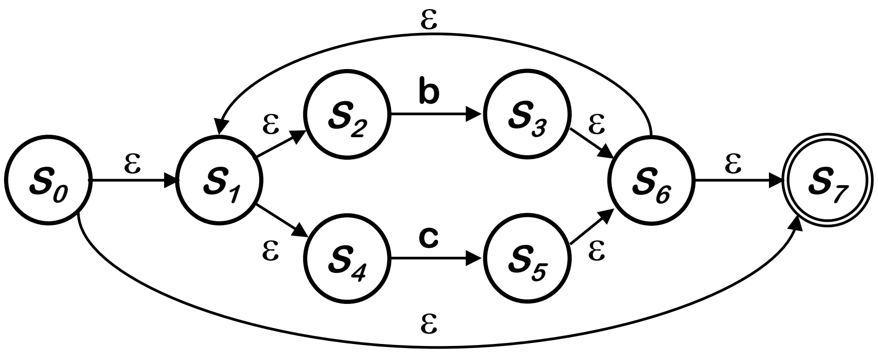
1. a, b, & c



2. b | c

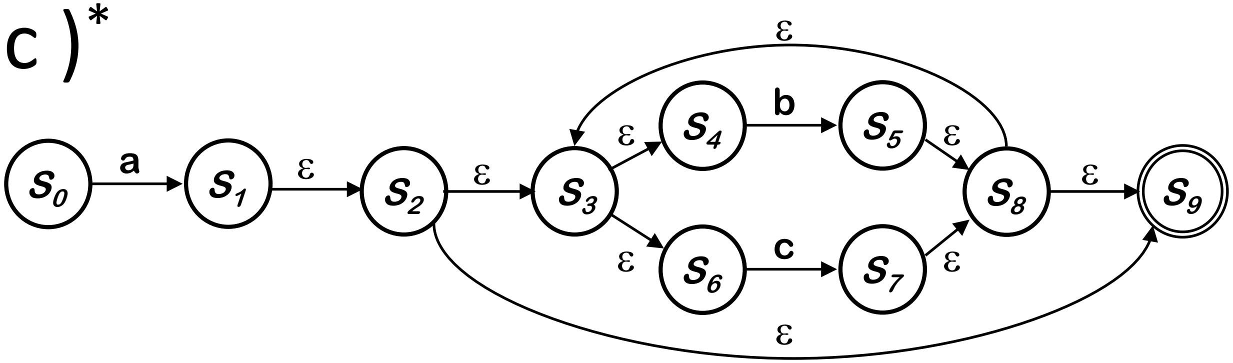


3. (b | c)*



Example of Thompson's Construction

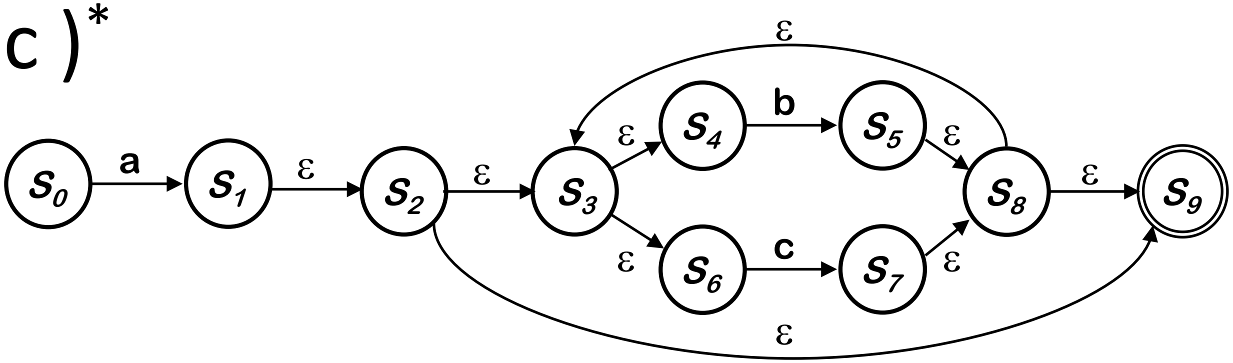
4. a (b | c)*



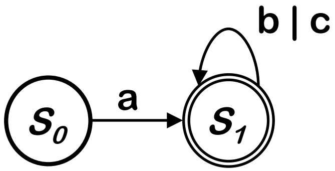
We could do a bit better. ☺

Example of Thompson’s Construction

4. a (b | c)*



We could do a bit better. ☺

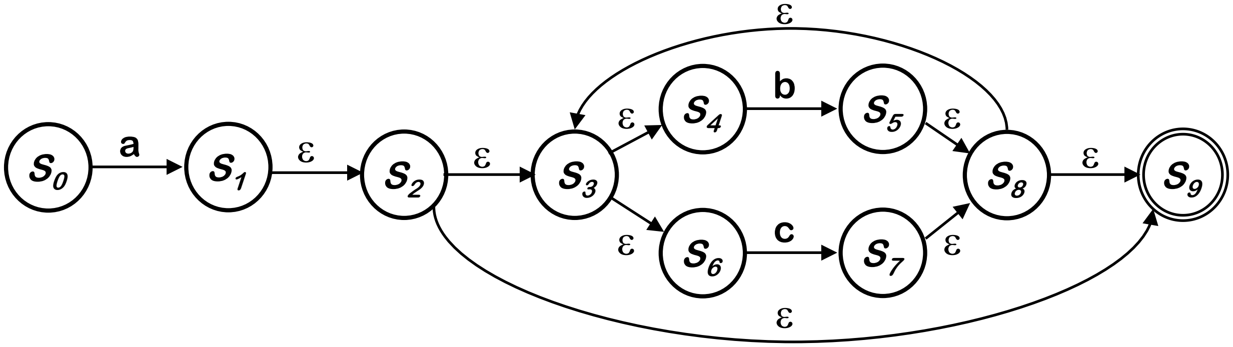


Today – part 1

- Lexing
- Flex & other scanner generators
- Regular Expressions
- Finite Automata
- RE → NFA
- **NFA → DFA**
- DFA → Minimized DFA
- Limits of Regular Languages

RE → NFA → DFA

- Can’t directly execute Non-deterministic FA
- Need to convert NFA to DFA
- Essentially, we will build a DFA that **simulates** the NFA

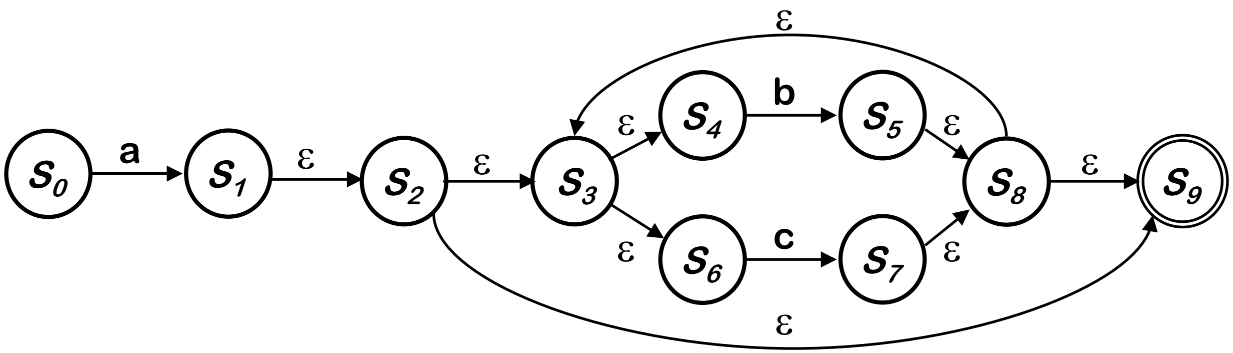


- Key idea: Keep track of all possible NFA states we could be in at each step:

  the **set of all possible NFA states** becomes the DFA state

Subset construction

- start in state { s_0 }.
- For each edge create a set of all states that can be reached. Continue until done.
- All sets that contain an NFA accepting state are accepting.



Lets first deal with ε edges

- ε-closure: all states that can be reached only along ε-edges:
- Computing ε-closure(s) for s∈S:

```
– initialize all ε-closure(s) = { s }
– while some ε-closure(s) changed
        foreach s∈S:
            foreach q ∈ ε-closure(s) :
                ε-closure(s) = ε-closure(s) ∪ δ(q, ε)
```

- Terminates?

Subset Construction

- NFA: $\{\Sigma, Q, \delta, q_0, F\} \rightarrow$ DFA: $\{\Sigma, S, \Delta, s_0, F'\}$

```
$s_0 \leftarrow \varepsilon$-closure($q_0$)
while $\exists$ unmarked s $\in$ S:
    mark s
    foreach a $\in \Sigma$
        t $\leftarrow$ $\varepsilon$-closure(Move(s, a))
        if t $\notin$ S:
            add t to S
        $\Delta$ (s,a) $\leftarrow$ t
```

Subset Construction

- NFA: $\{\Sigma, Q, \delta, q_0, F\} \rightarrow$ DFA: $\{\Sigma, S, \Delta, s_0, F'\}$

```
$s_0 \leftarrow \varepsilon$-closure($q_0$)
while $\exists$ unmarked $s \in S$:
    mark s
    foreach $a \in \Sigma$
        t $\leftarrow \varepsilon$-closure(Move(s, a))
        if $t \notin S$:
            add t to S
        $\Delta$ (s,a) $\leftarrow$ t
```

Move(s, a)
  the set of states
  reachable from s by a

Subset Construction

- NFA: $\{\Sigma, Q, \delta, q_0, F\} \rightarrow$ DFA: $\{\Sigma, S, \Delta, s_0, F'\}$

```
$s_0 \leftarrow \varepsilon$-closure($q_0$)
while $\exists$ unmarked s $\in$ S:
    mark s
    foreach a $\in \Sigma$
        t $\leftarrow \varepsilon$-closure(Move(s, a))
        if t $\notin$ S:
            add t to S
        $\Delta$ (s,a) $\leftarrow$ t
```

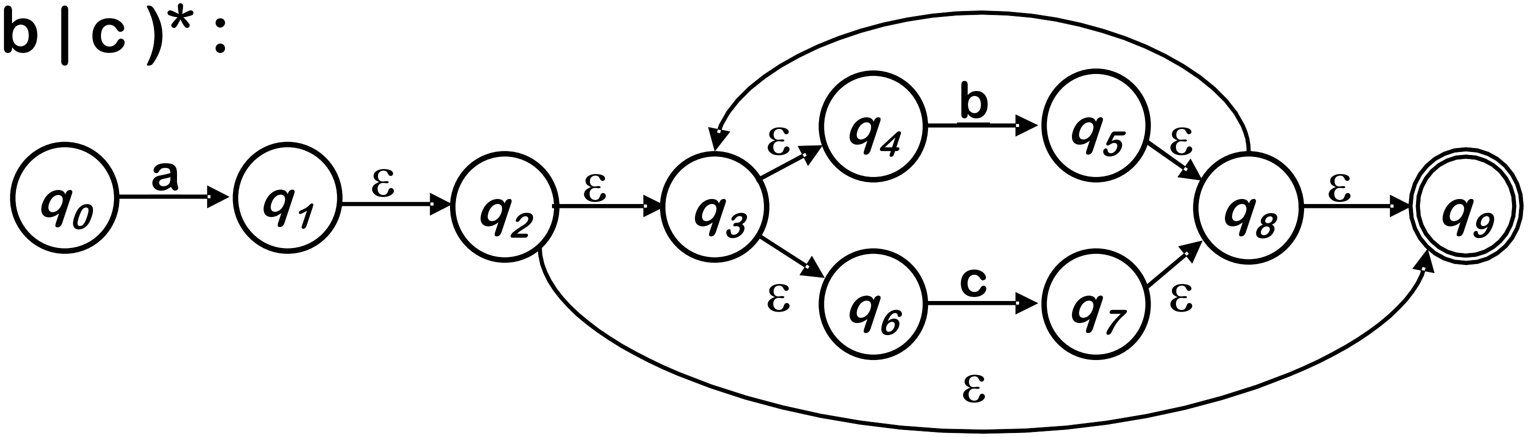
Why does this terminate?

Subset Construction

- NFA: $\{\Sigma, Q, \delta, q_0, F\} \rightarrow$ DFA: $\{\Sigma, S, \Delta, s_0, F'\}$
- Example of a fixed point computation
  - S is finite, at most ?
  - Always add to S, i.e., while loop is monotone
  - no duplicates in S
  - stop when S stops changing
- Other fixed point computations:
  - Constructing LR(1) items
  - Many Dataflow analysis (e.g., liveness)

example of subset construction

a (b | c)* :

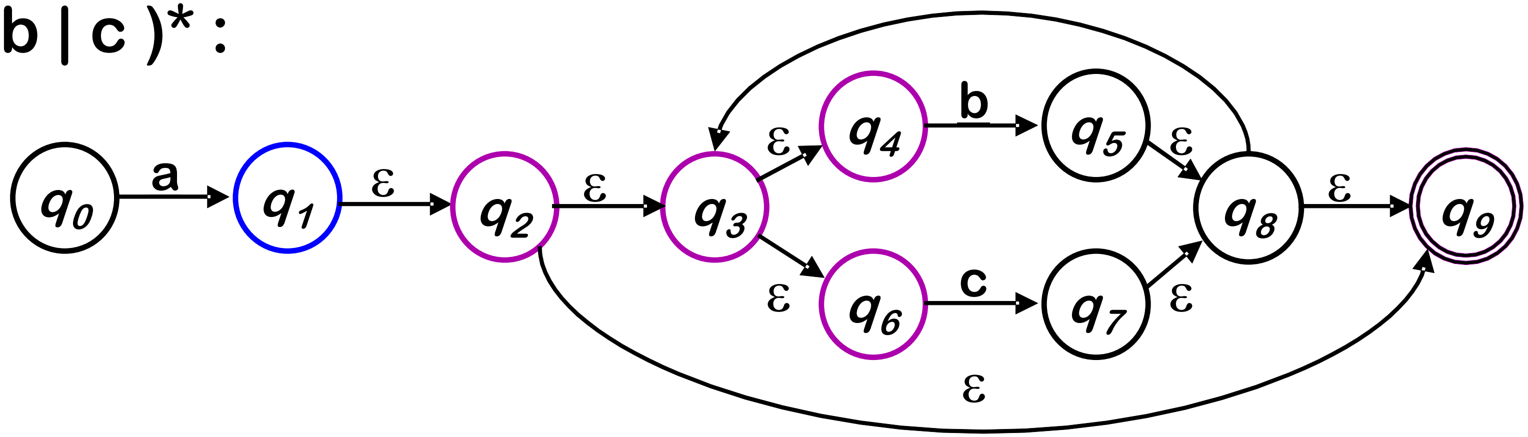


DFA States	NFA States	a	b	c
s_0	0			

Move(s_0,a)?

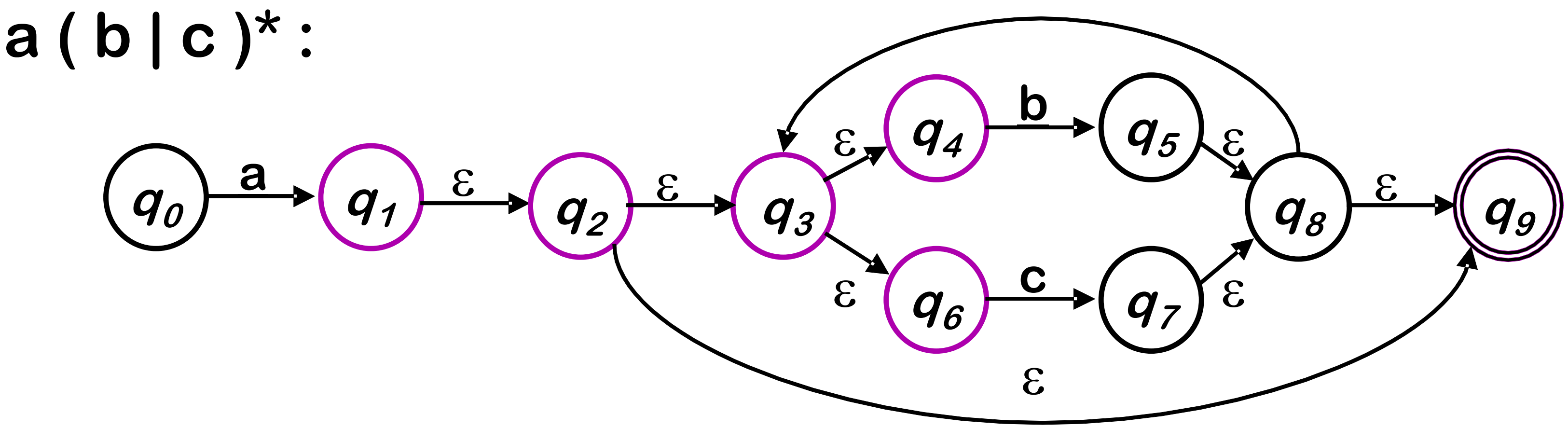
example of subset construction

a (b | c)* :



DFA States	NFA States	a	b	c
s_0	0	1, 2, 3, 4, 6, 9	-	-
s_1				

example of subset construction

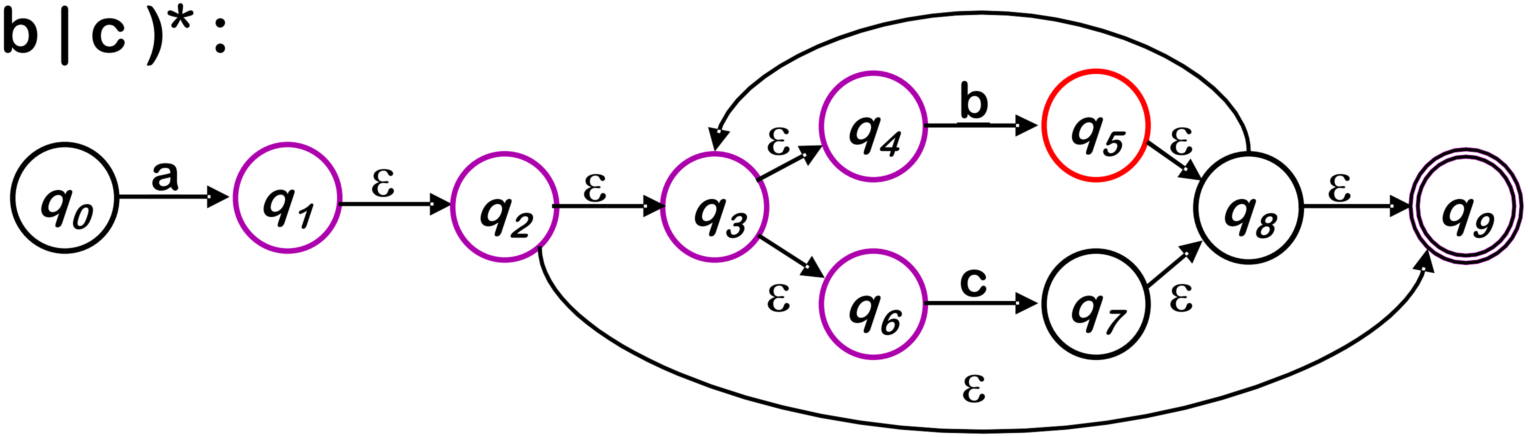


DFA States	NFA States	a	b	c
s_0	0	1, 2, 3, 4, 6, 9	-	-
s_1	1, 2, 3, 4, 6, 9	-		

b?

example of subset construction

a (b | c)* :

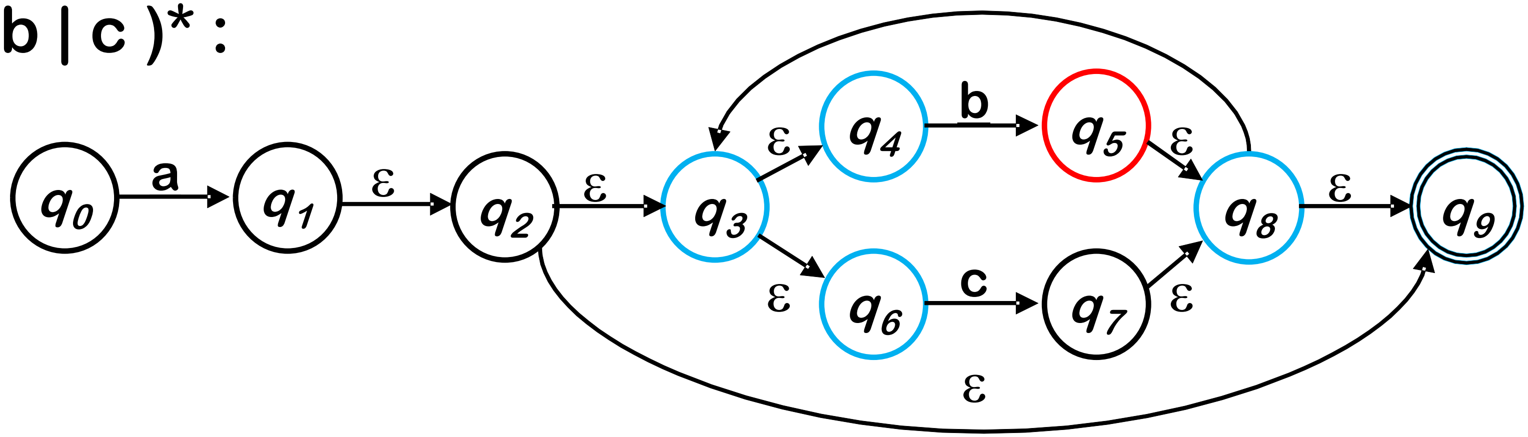


DFA States	NFA States	a	b	c
s_0	0	1, 2, 3, 4, 6, 9	-	-
s_1	1, 2, 3, 4, 6, 9	-	5	

ε-closure?

example of subset construction

a (b | c)* :

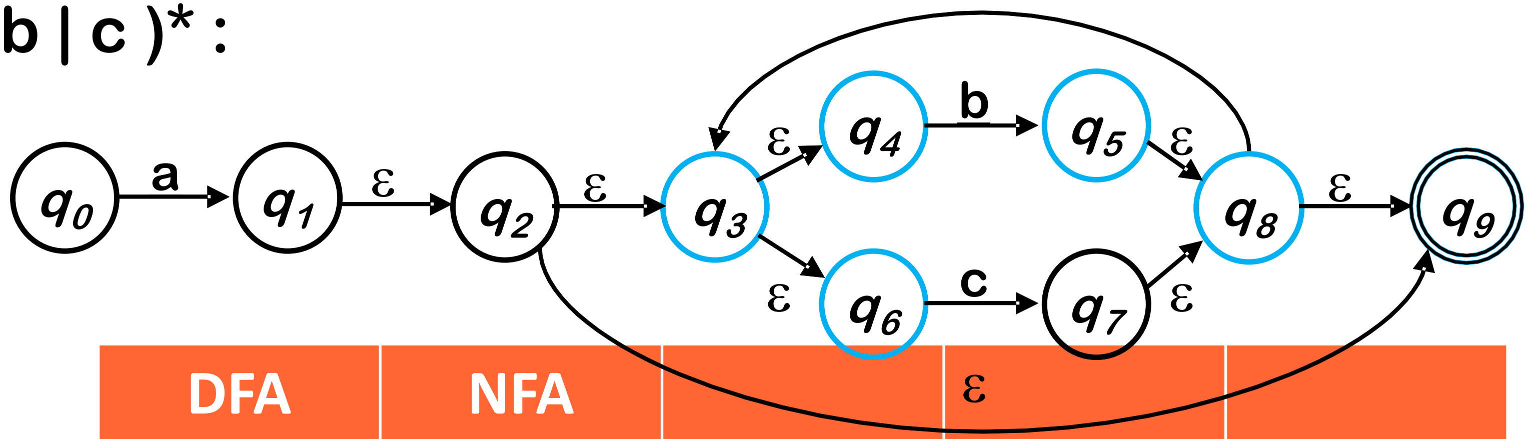


DFA States	NFA States	a	b	c
s_0	0	1, 2, 3, 4, 6, 9	-	-
s_1	1, 2, 3, 4, 6, 9	-	5, 3, 4, 6, 8, 9	

c?

example of subset construction

a (b | c)* :

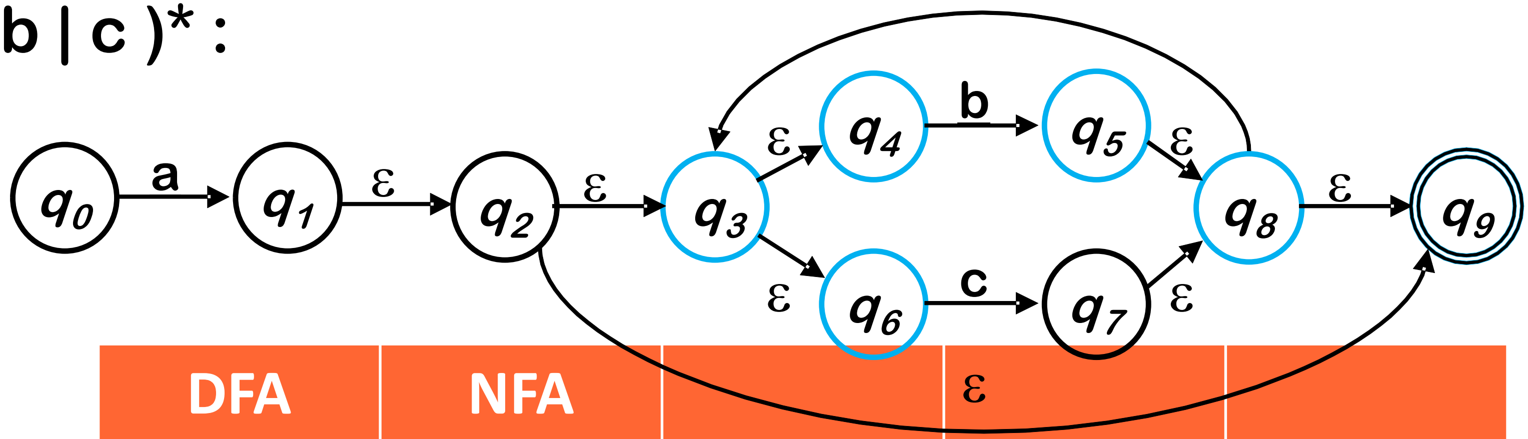


DFA States	NFA States	a	b	c
s_0	0	1, 2, 3, 4, 6, 9	-	-
s_1	1, 2, 3, 4, 6, 9	-	5, 3, 4, 6, 8, 9	7, 3, 4, 6, 8, 9
s_2	5, 3, 4, 6, 8, 9			
s_3	7, 3, 4, 6, 8, 9			

s_2,a?

example of subset construction

a (b | c)* :

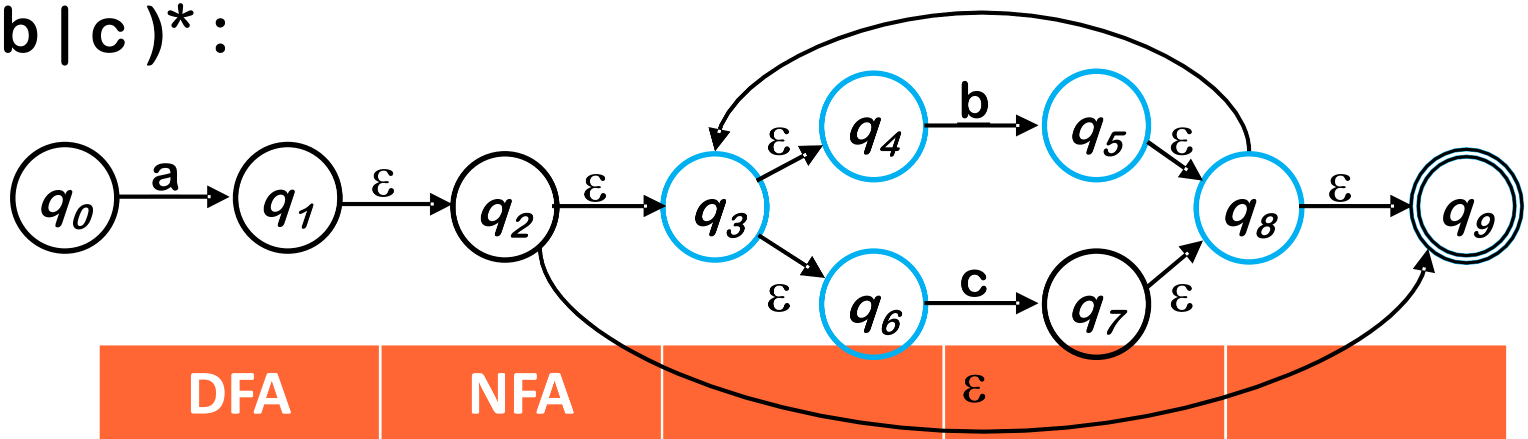


DFA States	NFA States	a	b	c
s_0	0	1, 2, 3, 4, 6, 9	-	-
s_1	1, 2, 3, 4, 6, 9	-	5, 3, 4, 6, 8, 9	7, 3, 4, 6, 8, 9
s_2	5, 3, 4, 6, 8, 9	-		
s_3	7, 3, 4, 6, 8, 9			

s_2,b?

example of subset construction

a (b | c)* :

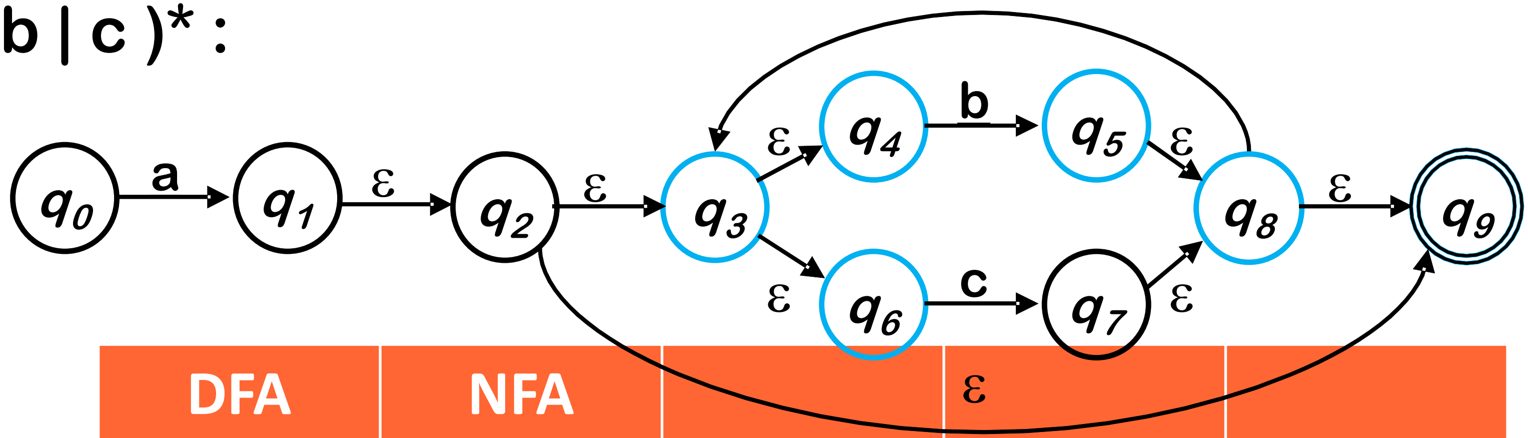


DFA States	NFA States	a	b	c
s_0	0	1, 2, 3, 4, 6, 9	-	-
s_1	1, 2, 3, 4, 6, 9	-	5, 3, 4, 6, 8, 9	7, 3, 4, 6, 8, 9
s_2	5, 3, 4, 6, 8, 9	-	s_2	
s_3	7, 3, 4, 6, 8, 9			

s_2,c?

example of subset construction

a (b | c)* :

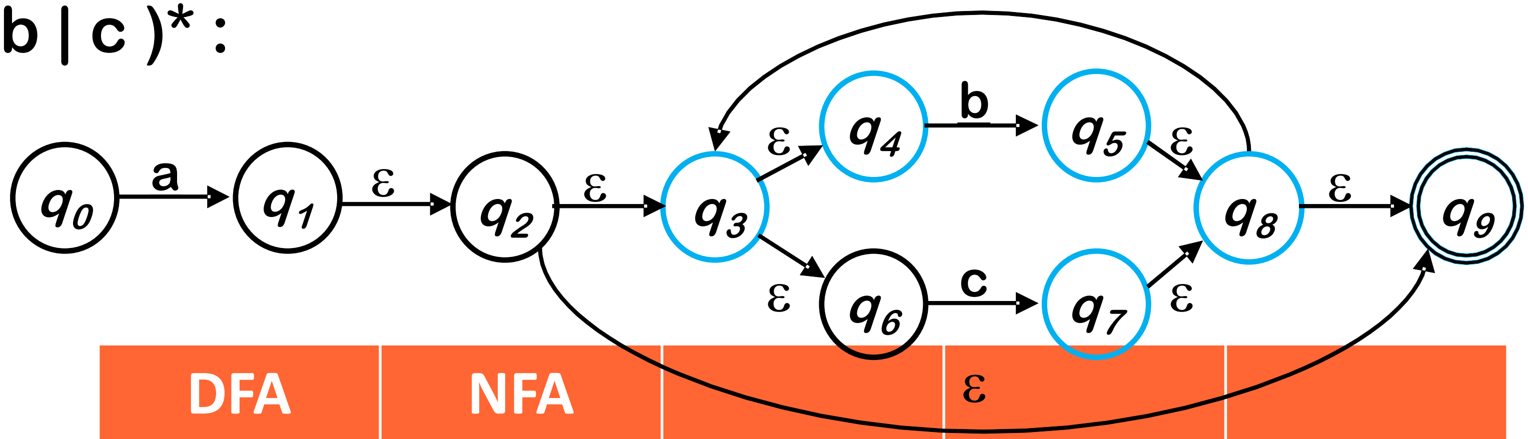


DFA States	NFA States	a	b	c
s_0	0	1, 2, 3, 4, 6, 9	-	-
s_1	1, 2, 3, 4, 6, 9	-	5, 3, 4, 6, 8, 9	7, 3, 4, 6, 8, 9
s_2	5, 3, 4, 6, 8, 9	-	s_2	s_3
s_3	7, 3, 4, 6, 8, 9			

rest?

example of subset construction

a (b | c)* :

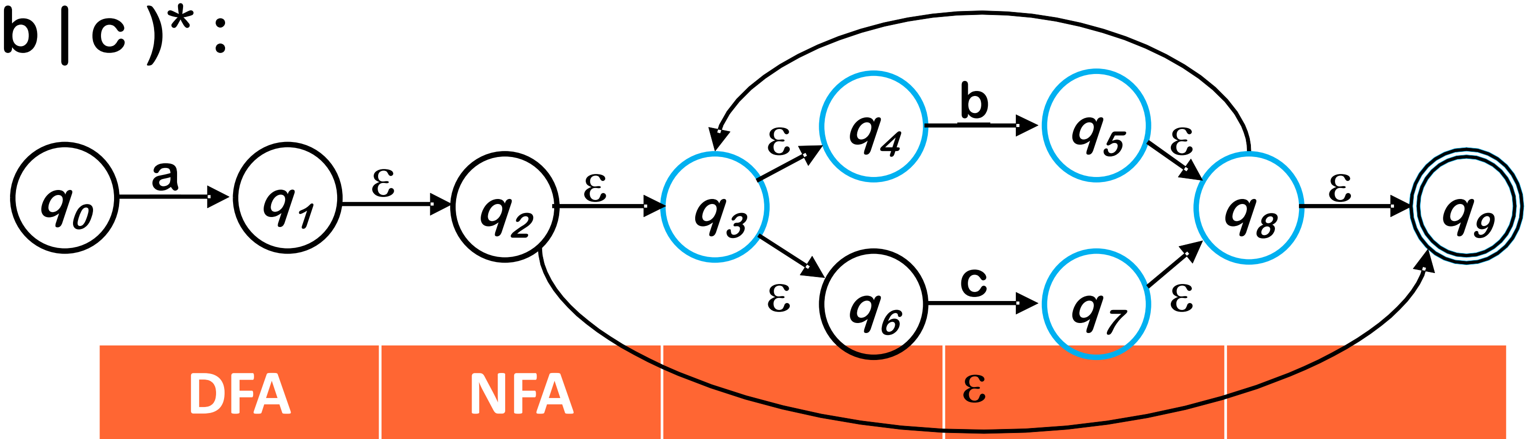


DFA States	NFA States	a	b	c
s_0	0	1, 2, 3, 4, 6, 9	-	-
s_1	1, 2, 3, 4, 6, 9	-	5, 3, 4, 6, 8, 9	7, 3, 4, 6, 8, 9
s_2	5, 3, 4, 6, 8, 9	-	s_2	s_3
s_3	7, 3, 4, 6, 8, 9	-	s_2	s_3

Final?

example of subset construction

a (b | c)* :

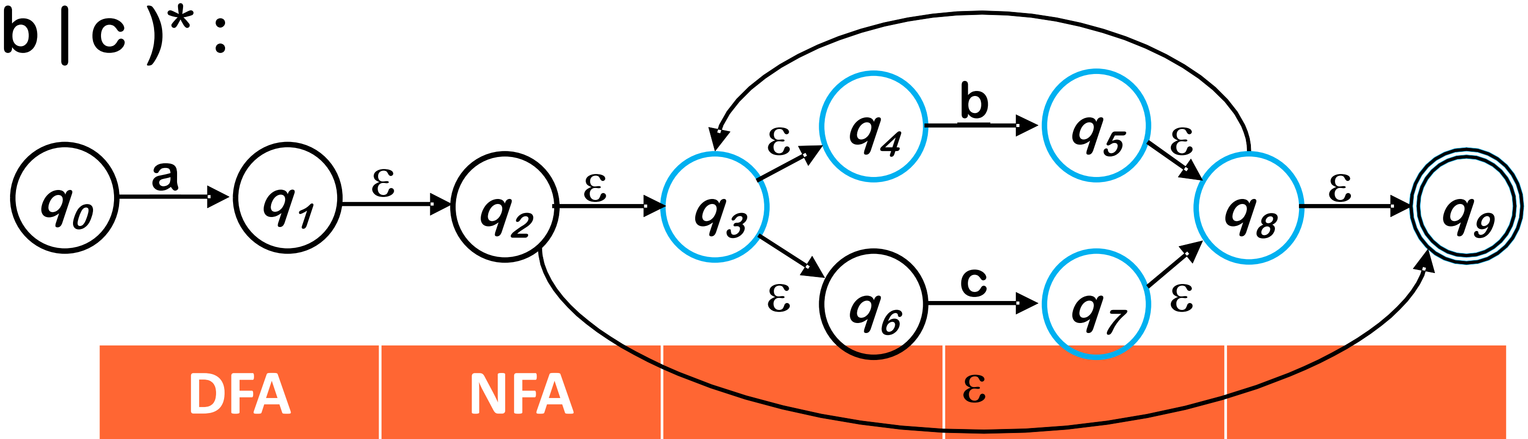


DFA States	NFA States	a	b	c
s_0	0	1, 2, 3, 4, 6, 9	-	-
s_1	1, 2, 3, 4, 6, **9**	-	5, 3, 4, 6, 8, 9	7, 3, 4, 6, 8, 9
s_2	5, 3, 4, 6, 8, **9**	-	s_2	s_3
s_3	7, 3, 4, 6, 8, **9**	-	s_2	s_3

Final?

example of subset construction

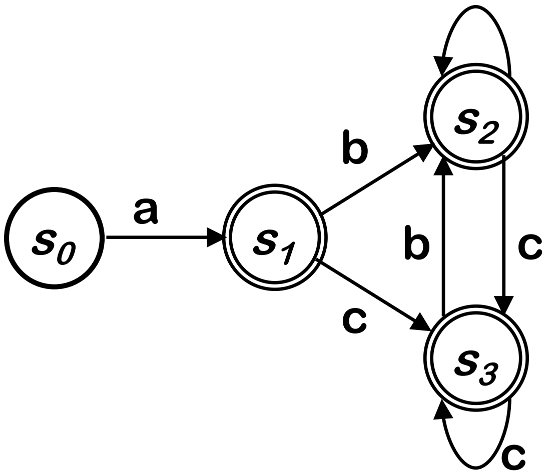
a (b | c)* :

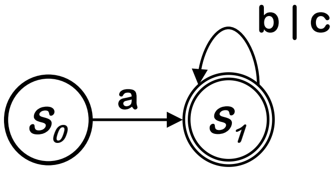


DFA States	NFA States	a	b	c
s_0	0	1, 2, 3, 4, 6, 9	-	-
s_1	1, 2, 3, 4, 6, **9**	-	5, 3, 4, 6, 8, 9	7, 3, 4, 6, 8, 9
s_2	5, 3, 4, 6, 8, **9**	-	s_2	s_3
s_3	7, 3, 4, 6, 8, **9**	-	s_2	s_3

example of subset construction

a (b | c)* :





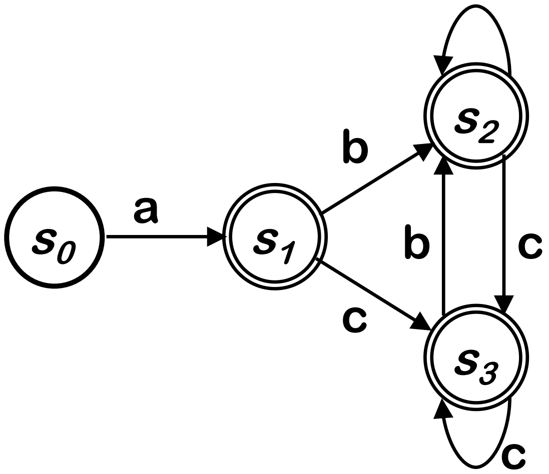
DFA States	NFA States	a	b	c
s_0	0	1, 2, 3, 4, 6, 9	-	-
s_1	1, 2, 3, 4, 6, **9**	-	5, 3, 4, 6, 8, 9	7, 3, 4, 6, 8, 9
s_2	5, 3, 4, 6, 8, **9**	-	s_2	s_3
s_3	7, 3, 4, 6, 8, **9**	-	s_2	s_3

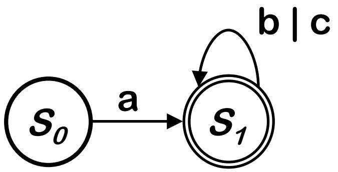
Today – part 1

- Lexing
- Flex & other scanner generators
- Regular Expressions
- Finite Automata
- RE → NFA
- NFA → DFA
- **DFA → Minimized DFA**
- Limits of Regular Languages

DFA Minimization

- Partition states into equivalent sets
- Two states are equivalent iff:
  - paths entering them are the same
  - ∀ a ∈Σ, transitions lead to equivalent states
- transition on a to different sets ⇒ different states.





DFA Minimization

- Plan:
  - start with maximal sets: { Q } and { Q – F }
  - partition sets for each a $\in\Sigma$ until no change
  - paritions become new states of minimized DFA

- Partitioning a set on "α"
  - Assume q_a, & $q_b \in$ s, and $\delta(q_a,\alpha) = q_x$ & $\delta(q_b, \alpha) = q_y$
  - If q_x & q_y are not in the same set, then s must be split (q_a has transition on α, q_b does not $\Rightarrow$ α splits s)

- One state in the final DFA cannot have two transitions on α

DFA Minimization

```
P ← { F, {Q-F}}
while ( P is still changing)
    T ← { }
    for each set S $\in$ P
        for each $\alpha \in \Sigma$
            partition S by $\alpha$ into $S_1, S_2, \ldots, S_k$
            $T \leftarrow T \cup S_1 \cup S_2 \cup \ldots \cup S_k$
    if $T \neq P$ then
        P ← T
```

DFA Minimization

```
$P \leftarrow \{ F, \{Q-F\}\}$
while ( P is still changing)
    $T \leftarrow \{ \}$
    for each set $S \in P$
        for each $\alpha \in \Sigma$
            partition S by $\alpha$ into $S_1, S_2, \ldots, S_k$
            $T \leftarrow T \cup S_1 \cup S_2 \cup \ldots \cup S_k$
    if $T \neq P$ then
        $P \leftarrow T$
```

Another Fixed Point Alg

Terminates:

- maximum of $2^{|Q|}$ sets
- Always adding to P
- Never combining sets in P

Initial partition ensures that final states remain final.

Hopcroft's worklist algorithm is efficient.

Today – part 1

- Lexing
- Flex & other scanner generators
- Regular Expressions
- Finite Automata
- RE → NFA
- NFA → DFA
- DFA → Minimized DFA
- **Limits of Regular Languages**

Regular Languages

- Regular Expressions are great
  - concise notation
  - automatic scanner generation
  - lots of useful languages
- But, …
  - Not all languages are regular
    - Context Free Languages
    - Context Sensitive Languages
  - Even simple things like balanced parenthesis, e.g., L = { A^kB^k } (or nested comments!)
  - RL can't count

Not all Scanning is easy

- Language design should start with lexemes
  - My favorite example from PL/I

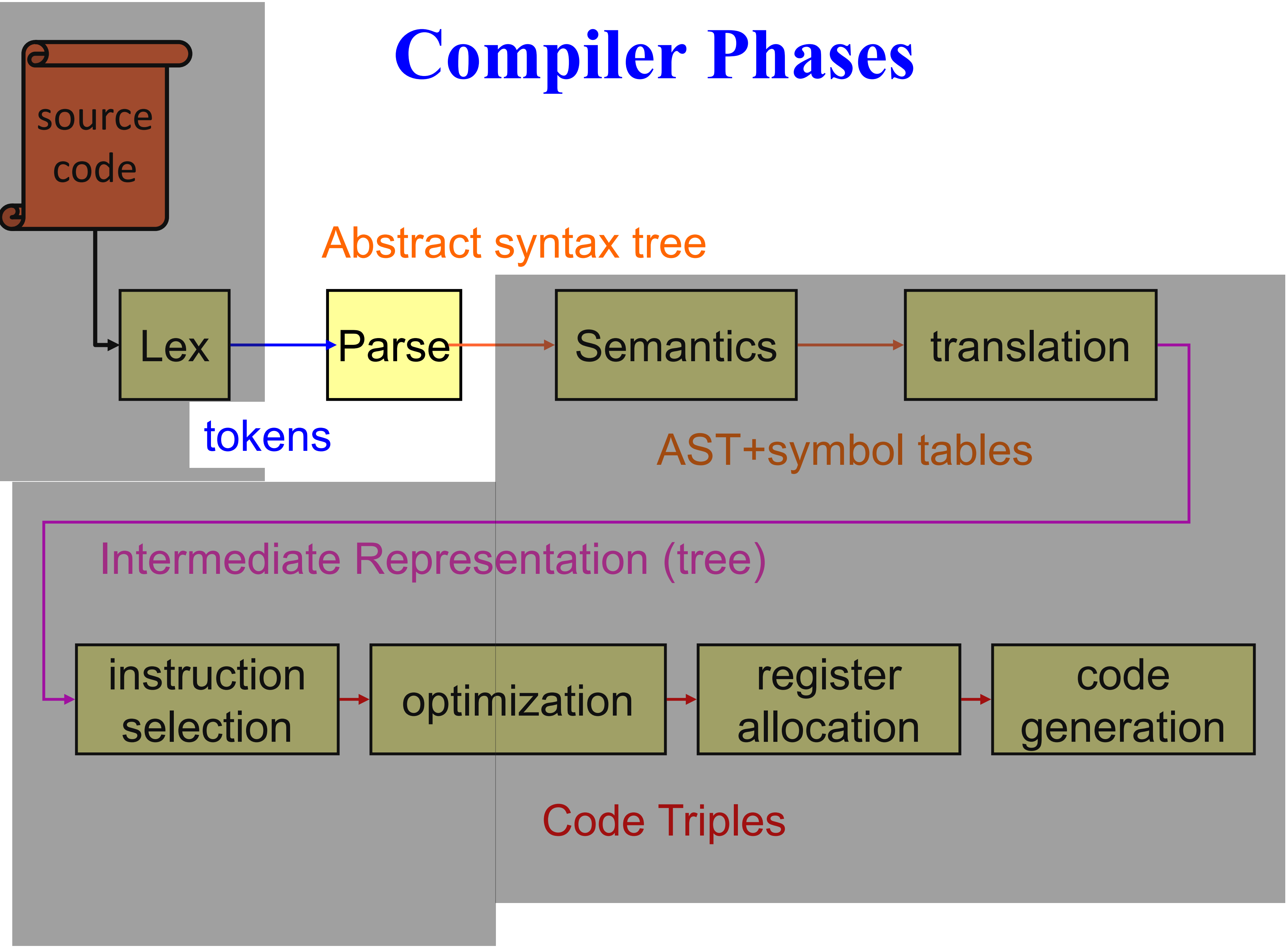
```
if then then then = else; else else = then
```

- blanks not important in Fortran
- nested comments in C
- limited identifier lengths in Fortran

Today – part 2

- Languages and Grammars
- Context Free Grammars
- Derivations & Parse Trees
- Ambiguity
- Top-down parsers
- FIRST, FOLLOW, and NULLABLE
- Bottom-up parsers

Compiler Phases



source code

Lex

tokens

Parse

Abstract syntax tree

Semantics

translation

AST+symbol tables

Intermediate Representation (tree)

instruction selection

optimization

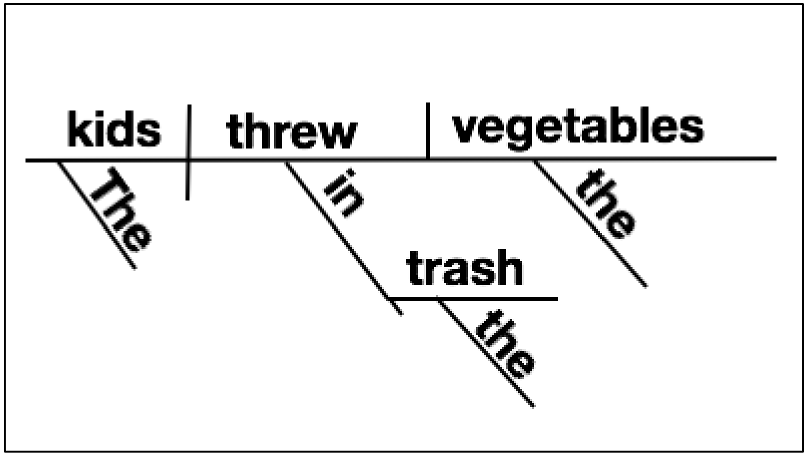
register allocation

code generation

Code Triples

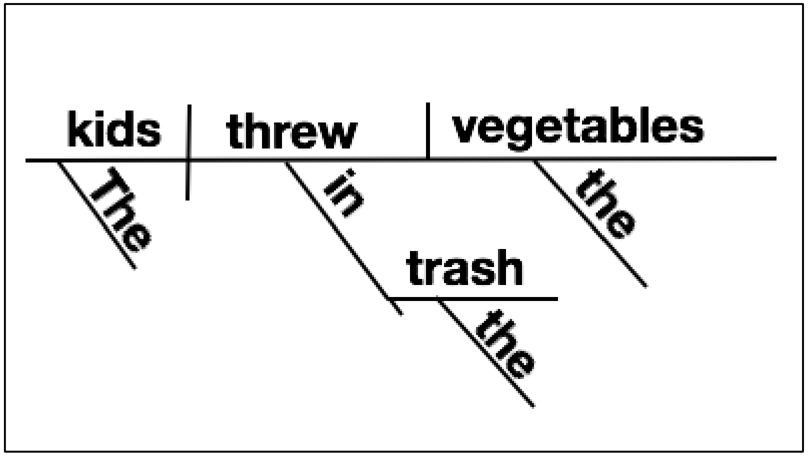
Languages

- Compiler translates from sequence of characters to an executable.
- A series of language transformations
- lexing: characters → tokens
- parsing: tokens → "sentences"



Languages

- Compiler translates from sequence of characters to an executable.
- A series of language transformations
- lexing: characters → tokens
- parsing: tokens → parse trees



Grammers and Languages

- A grammer, G, recognizes a language, L(G)
  - Σ set of terminal symbols
  - A set of non-terminals
  - S the start symbol, a non-terminal
  - P a set of productions
- Usually,
  - α, β, γ, … strings of terminals and/or non-terminals
  - A, B, C, … are non-terminals
  - a, b, c, … are terminals
- General form of a production is: $\alpha \rightarrow \beta$

Derivation

- A sequence of applying productions starting with S and ending with *w*

$$S \rightarrow \gamma_1 \rightarrow \gamma_2 \ldots \rightarrow \gamma_{n-1} \rightarrow w$$

$$S \rightarrow^* w$$

- *L(G) are all the w that can be derived from S*

→ ε

Regular Grammar (NFA)

- Regular expressions and NFAs can be described by a regular grammar
- E.G.,

  S → aA

  A → Sb

  S → ε
- An example derivation of aab:

→ ε

Regular Grammar (NFA)

- Regular expressions and NFAs can be described by a regular grammar
- E.G., a*bc*

  S → aS

  S → bA

  A → ε

  A → cA

- An example derivation of aabc:

  S → aS

→ ε

Regular Grammar (NFA)

- Regular expressions and NFAs can be described by a regular grammar
- E.G., a*bc*

S → aS
S → bA
A → ε
A → cA

- An example derivation of aabc:

S → aS→ aaS

→ ε

Regular Grammar (NFA)

- Regular expressions and NFAs can be described by a regular grammar
- E.G., a*bc*

S → aS
S → bA
A → ε
A → cA

- An example derivation of aabc:

S → aS→ aaS→ aabA

→ ε

Regular Grammar (NFA)

- Regular expressions and NFAs can be described by a regular grammar
- E.G., a*bc*

S → aS
S → bA
A → ε
A → cA

- An example derivation of aabc:

S → aS→ aaS→ aabA→ aabcA

→ ε

Regular Grammar (NFA)

- Regular expressions and NFAs can be described by a regular grammar
- E.G., a*bc*

  S → aS
  S → bA
  A → ε
  A → cA
- An example derivation of aabc:

  S → aS→ aaS→ aabA→ aabcA → aabc

→ ε

Regular Grammar (NFA)

- Regular expressions and NFAs can be described by a regular grammar
- E.G., a*bc*

S → aS
S → bA
A → ε
A → cA

- Above is a right-regular grammar
- All rules are of form:

A → a
A → aB
A → ε

→ ε

Regular Grammar (NFA)

- Regular expressions and NFAs can be described by a regular grammar
- right regular grammar:

  A → a
  A → aB
  A → ε
- left regular grammar:

  A → a
  A → Ba
  A → ε
- Regular grammars are either right-regular or left-regular.

Expressiveness

- Restrictions on production rules limit expressiveness of grammars.
- No restrictions allow a grammar to recognize all recursively enumerable languages
- A bit too expressive for our uses ☺
- Regular grammars cannot recognize a^nb^n
- We need something more expressive

Chomsky Hierarchy

Class	Language	Automaton	Form	"word" problem	Example
0	Recursively Enumerable	Turing Machine	any	undecidable	Post's Corresp. problem
1	Context Sensitive	Linear-Bounded TM	$\alpha A\beta \rightarrow \alpha\gamma\beta$	PSPACE-complete	$a^n b^n c^n$
2	Context Free	Pushdown Automata	$A \rightarrow \alpha$	cubic	$a^n b^n$
3	Regular	NFA	$A \rightarrow a$ $A \rightarrow aB$	linear	$a^* b^*$

Today – part 2

- Languages and Grammars
- Context Free Grammars
- Derivations & Parse Trees
- Ambiguity
- Top-down parsers
- FIRST, FOLLOW, and NULLABLE
- Bottom-up parsers

Context-Free Grammar

- A context-free grammar, G, is described by:
  - Σ, a set of terminals (which are just the set of possible tokens from the lexer)
    e.g., **`if`**, **`then`**, **`while`**, **`id`**, **`int`**, **`string`**, …
  - A, a set of non-terminals.
    Non-terminals are syntactic variables which define sets of strings in the language
    e.g., stmt, expr, term, factor, vardecl, …
  - S
  - P

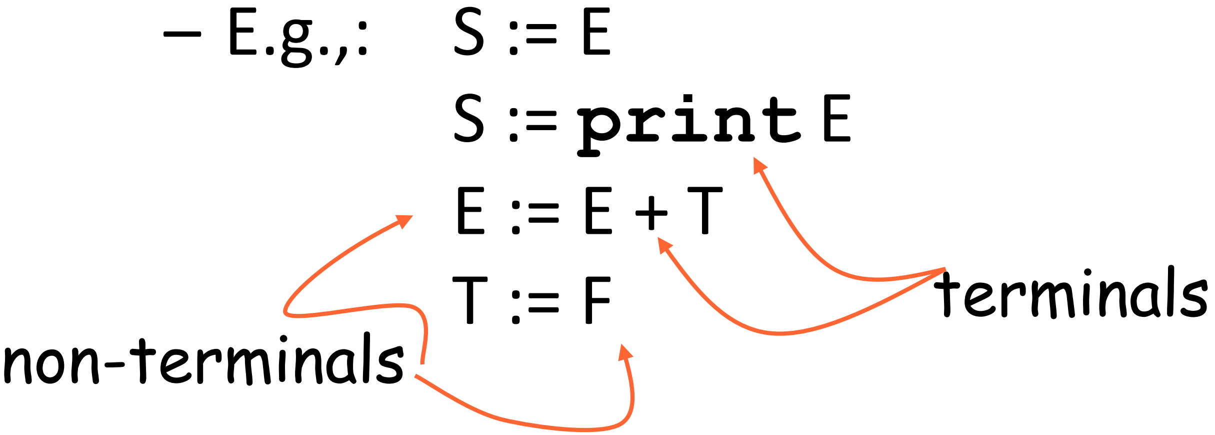
Context-Free Grammar

- A context-free grammar, G, is described by:
  - Σ, a set of terminals …
  - A, a set of non-terminals.
  - S, S $\in$ A, the start symbol
    The set of strings derived from S are the valid string in the language.
  - P, set of productions that specify how terminals and non-terminals combine to form strings in the language
    a production, p, has the form: $A \rightarrow \alpha$

Context-Free Grammar

- A context-free grammar, G, is described by:
  - Σ, a set of terminals …
  - A, a set of non-terminals.
  - S, S ∈ A, the start symbol
  - P, set of productions …

    a production, p, has the form: : A→ α
  - E.g.,:



S := E

S := **`print`** E

E := E + T

T := F

What makes a grammar CF?

- Only one NT on left-hand side → context-free
- What makes a grammar context-sensitive?
- $\alpha A \beta \rightarrow \alpha \gamma \beta$ where
  - α or β may be empty,
  - but γ is not-empty
- Are context-sensitive grammars useful for compiler writers?

Simple Grammar of Expressions

S := Exp

Exp := Exp + Exp

Exp := Exp - Exp

Exp := Exp * Exp

Exp := Exp / Exp

Exp := **id**

Exp := **int**

Describes a language of expressions. e.g.: 2+3*x

Derivations

- A sequence of steps in which a non-terminal is replaced by its right-hand side.

1 S := Exp

2 Ex

3 Ex

4 Exp:= Exp * Exp

5 Exp:= Exp / Exp

6 Exp:= id

7 Exp:= int

S

xp

by 6 ⇒ Exp * $\mathtt{id}_x$

by 2 ⇒ Exp + Exp * $\mathtt{id}_x$

by 7 ⇒ $\mathtt{int}_2$ + Exp * $\mathtt{id}_x$

by 7 ⇒ $\mathtt{int}_2$ + $\mathtt{int}_3$ * $\mathtt{id}_x$

There are possibly many derivations determined by the NT chosen to expand.

Leftmost Derivations

- Leftmost derivation: leftmost NT always chosen

1 S := Exp

2 Exp:= Exp + Exp

3 Exp:= Exp - Exp

4 Exp:= Exp * Exp

5 Exp:= Exp / Exp

6 Exp:= **id**

7 Exp:= **int**

S

by 1 ⇒ Exp

by 4 ⇒ Exp * Exp

by 2 ⇒ Exp + Exp * Exp

by 7 ⇒ $\mathbf{int}_2$ + Exp * Exp

by 7 ⇒ $\mathbf{int}_2$ + $\mathbf{int}_3$ * Exp

by 6 ⇒ $\mathbf{int}_2$ + $\mathbf{int}_3$ * $\mathbf{id}_x$

Rightmost Derivations

- Rightmost derivation: rightmost NT always chosen

1 S := Exp

2 Exp:= Exp + Exp

3 Exp:= Exp - Exp

4 Exp:= Exp * Exp

5 Exp:= Exp / Exp

6 Exp:= **id**

7 Exp:= **int**

S

by 1 ⇒ Exp

by 4 ⇒ Exp * Exp

by 6 ⇒ Exp * $\mathbf{id}_x$

by 2 ⇒ Exp + Exp * $\mathbf{id}_x$

by 7 ⇒ Exp + $\mathbf{int}_3$ * $\mathbf{id}_x$

by 7 ⇒ $\mathbf{int}_2$ + $\mathbf{int}_3$ * $\mathbf{id}_x$

Parse Trees

- symbols in rhs are children of NT being rewritten

S

by 1 ⇒ Exp

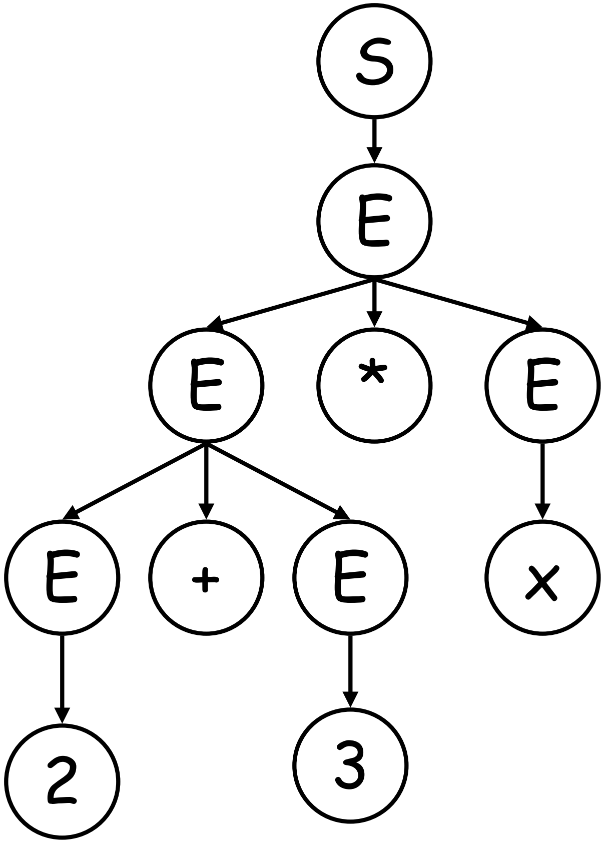
by 4 ⇒ Exp * Exp

by 2 ⇒ Exp + Exp * Exp

by 7 ⇒ $\mathtt{int}_2$ + Exp * Exp

by 7 ⇒ $\mathtt{int}_2$ + $\mathtt{int}_3$ * Exp

by 6 ⇒ $\mathtt{int}_2$ + $\mathtt{int}_3$ * $\mathtt{id}_x$



Parse Trees

- parse tree for rightmost derivation

S

by 1 $\Rightarrow$ Exp

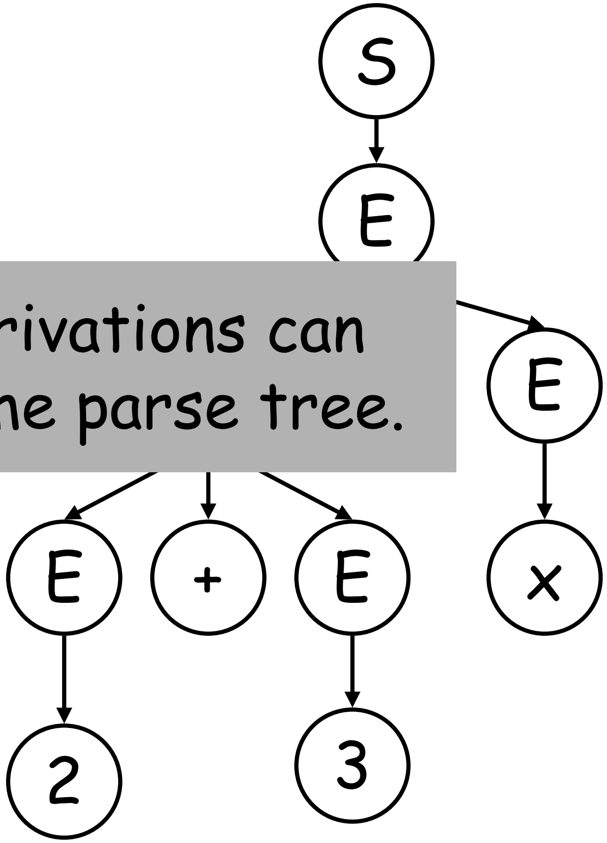
by 4 $\Rightarrow$ $\mathrm{Exp}\ *$

by 6 $\Rightarrow$ $\mathrm{Exp}\ *$

by 2 $\Rightarrow$ $\mathrm{Exp} + \mathrm{Exp} * \mathtt{id}_x$

by 7 $\Rightarrow$ $\mathrm{Exp} + \mathtt{int}_3 * \mathtt{id}_x$

by 7 $\Rightarrow$ $\mathtt{int}_2 + \mathtt{int}_3 * \mathtt{id}_x$



Different derivations can lead to the same parse tree.

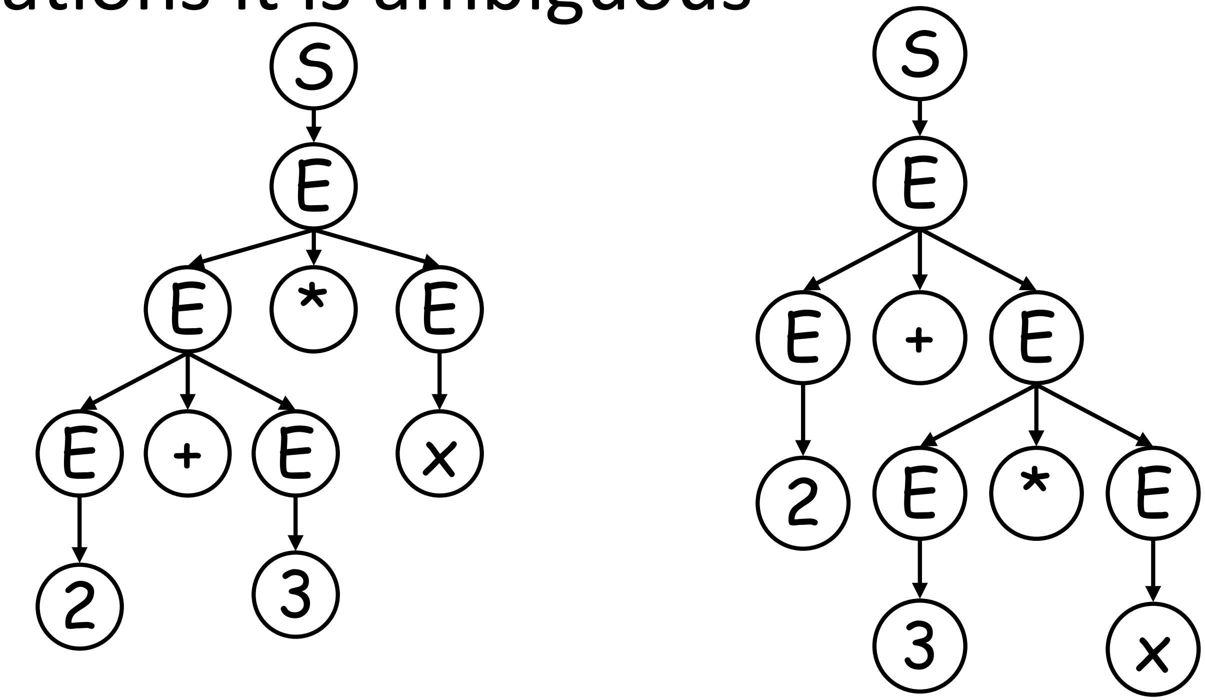
What about different parse trees for same sentence?

Ambiguous Grammars

- A gra a sentence with >1 parse trees. or,

What does ambiguity point out?

- If grammer has >1 leftmost (rightmost) derivations it is ambiguous



Converting Expression Grammar

- Adding precedence with more non-terminals
- One for each level of precedence:
  - (+, -) exp
  - (*, /) term
  - (**id**, **int**) factor
  - Make sure parse derives sentences that respect the precedence
  - Make sure that extra levels of precedence can be bypassed, i.e., “x” is still legal

A Better Exp Grammar

1	S	:= Exp
2	Exp	:= Exp + Term
3	Exp	:= Exp - Term
4	Exp	:= Term
5	Term	:= Term * Factor
6	Term	:= Term / Factor
7	Term	:= Factor
8	Factor	:= **id**
9	Factor	:= **int**

S

by 1 ⇒ Exp

by 2 ⇒ Exp + Term

by 4 ⇒ Term + Term

by 7 ⇒ Factor + Term

by 9 ⇒ **int**$_2$ + Term

by 5 ⇒ **int**$_2$ + Term * Factor

by 7 ⇒ **int**$_2$ + Factor * Factor

by 9 ⇒ **int**$_2$ + **int**$_3$ * Factor

by 8 ⇒ **int**$_2$ + **int**$_3$ * **id**$_x$

What is the parse tree?

Another Ambiguous Grammer

S := **if** E **then** S
  | **if** E **then** S **else** S
  | **other**

- What is the parse tree for:

  if E **then** **if** E **then** S **else** S?

- What is the language designers intention?
- Is there a context-free solution?

Dangling Else Grammar

S	:=	matchedS
	|	unmatchedS
unmatchedS	:=	**`if`** E **`then`** S
	|	if E then matchedS else unmatchedS
matchedS	:=	**`if`** E **`then`** matchedS **`else`** matchedS
	|	**`other`**

- Is this clearer?
- What is parse tree for: **`if`** E **`then`** **`if`** E **`then`** S **`else`** S?

Parser generators provide a better way

A primitive robot

```
Swing     := Back Swing Forward
           |
Back         :=   back-1-inch
Forward      :=   forward-2-inchs
```

- What is L(Swing)?

A primitive robot

```
S        :=  B S F
         |
B        :=  b
F        :=  f
```

- What is L(Swing)?
- What is the parse tree for “bbff”

Parsing a CFG

- Top-Down
  - start at root of parse-tree
  - pick a production and expand to match input
  - may require backtracking
  - if no backtracking required, predictive
- Bottom-up
  - start at leaves of tree
  - recognize valid prefixes of productions
  - consume input and change state to match
  - use stack to track state

Top-down Parsers

- Starts at root of parse tree and recursively expands children that match the input
- In general case, may require backtracking
- Such a parser uses recursive descent.
- When a grammar does not require backtracking a predictive parser can be built.

A Predictive Parser

```
S  :=  B S F
    |
B  :=  b
F  :=  f
```

Idea is for parser to do something besides recognize legal sentences.

```
S() {
        if match('b') -> B(); S(); F(); action();
        else return;
}
B() {   mustMatch('b');  action(); return;}
F() {   mustMatch('f'); action();  return;}
```

Top-Down parsing

- Start with root of tree, i.e., S
- Repeat until entire input matched:
  - pick a non-terminal, A, and pick a production $A \rightarrow \gamma$ that can match input, and expand tree
  - if no such rule applies, backtrack
- Key is obviously selecting the right production

Top-down for Exp Grammar

1 S := E
2 E := E + T
3 E := E - T
4 E := T
5 T := T * F
6 T := T / F
7 T := F
8 F := **id**
9 F := **int**

	S	▌ $\mathtt{int}_2$ - $\mathtt{int}_3$ * $\mathtt{id}_x$
by 1 ⇒	E	▌ $\mathtt{int}_2$ - $\mathtt{int}_3$ * $\mathtt{id}_x$

Top-down for Exp Grammar

1	S := E
2	E := E + T
3	E := E - T
4	E := T
5	T := T * F
6	T := T / F
7	T := F
8	F := id
9	F := int

	S	▮ $\texttt{int}_2$ - $\texttt{int}_3$ * $\texttt{id}_x$
by 1 ⇒	E	▮ $\texttt{int}_2$ - $\texttt{int}_3$ * $\texttt{id}_x$
by 2 ⇒	E + T	▮ $\texttt{int}_2$ - $\texttt{int}_3$ * $\texttt{id}_x$
by 4 ⇒	T + T	▮ $\texttt{int}_2$ - $\texttt{int}_3$ * $\texttt{id}_x$
by 7 ⇒	F + T	▮ $\texttt{int}_2$ - $\texttt{int}_3$ * $\texttt{id}_x$
by 9 ⇒	$\texttt{int}_2$ + T	$\texttt{int}_2$ ▮ - $\texttt{int}_3$ * $\texttt{id}_x$

Must backtrack here!

Top-down for Exp Grammar

1	S	::= E
2	E	::= E + T
3	E	::= E - T
4	E	::= T
5	T	::= T * F
6	T	::= T / F
7	T	::= F
8	F	::= id
9	F	::= int

	S	▌int$_2$ - int$_3$ * id$_x$
by 1 ⇒	E	▌int$_2$ - int$_3$ * id$_x$
by 2 ⇒	E + T	▌int$_2$ - int$_3$ * id$_x$
by 4 ⇒	T + T	▌int$_2$ - int$_3$ * id$_x$
by 7 ⇒	F + T	▌int$_2$ - int$_3$ * id$_x$
by 9 ⇒	int$_2$ + T	int$_2$▌- int$_3$ * id$_x$
by 3 ⇒	E - T	▌int$_2$ - int$_3$ * id$_x$
by 4 ⇒	T - T	▌int$_2$ - int$_3$ * id$_x$
by 7 ⇒	F - T	▌int$_2$ - int$_3$ * id$_x$
by 9 ⇒	int$_2$ - T	int$_2$▌- int$_3$ * id$_x$
by 5 ⇒	int$_2$ - T * F	int$_2$ - ▌int$_3$ * id$_x$

Top-down for Exp Grammar

1	S	:=	E	
2	E	:=	E + T	
3	E	:=	E - T	
4	E	:=	T	
5	T	:=	T * F	
6	T	:=	T / F	
7	T	:=	F	
8	F	:=	id	
9	F	:=	int	

S ▌int$_2$ - int$_3$ * id$_x$

by 1 ⇒ E ▌int$_2$ - int$_3$ * id$_x$

---

by 2 ⇒ E + T ▌int$_2$ - int$_3$ * id$_x$

by 4 ⇒ T + T ▌int$_2$ - int$_3$ * id$_x$

by 7 ⇒ F + T ▌int$_2$ - int$_3$ * id$_x$

by 9 ⇒ int$_2$ + T int$_2$▌- int$_3$ * id$_x$

---

by 3 ⇒ E - T ▌int$_2$ - int$_3$ * id$_x$

by 4 ⇒ T - T ▌int$_2$ - int$_3$ * id$_x$

by 7 ⇒ F - T ▌int$_2$ - int$_3$ * id$_x$

by 9 ⇒ int$_2$ - T int$_2$▌- int$_3$ * id$_x$

---

nt$_2$ -▌int$_3$ * id$_x$

What kind of derivation is this parsing?

Top-down for Exp Grammar

1	S := E
2	E := E + T
3	E := E - T
4	E := T
5	T := T * F
6	T := T / F
7	T := F
8	F := id
9	F := int

	S	▌int$_2$ - int$_3$ * id$_x$
by 1 ⇒	E	▌int$_2$ - int$_3$ * id$_x$
by 2 ⇒	E + T	▌int$_2$ - int$_3$ * id$_x$
by 2 ⇒	E + E + T	▌int$_2$ - int$_3$ * id$_x$
by 2 ⇒	E + E + E + T	▌int$_2$ - int$_3$ * id$_x$

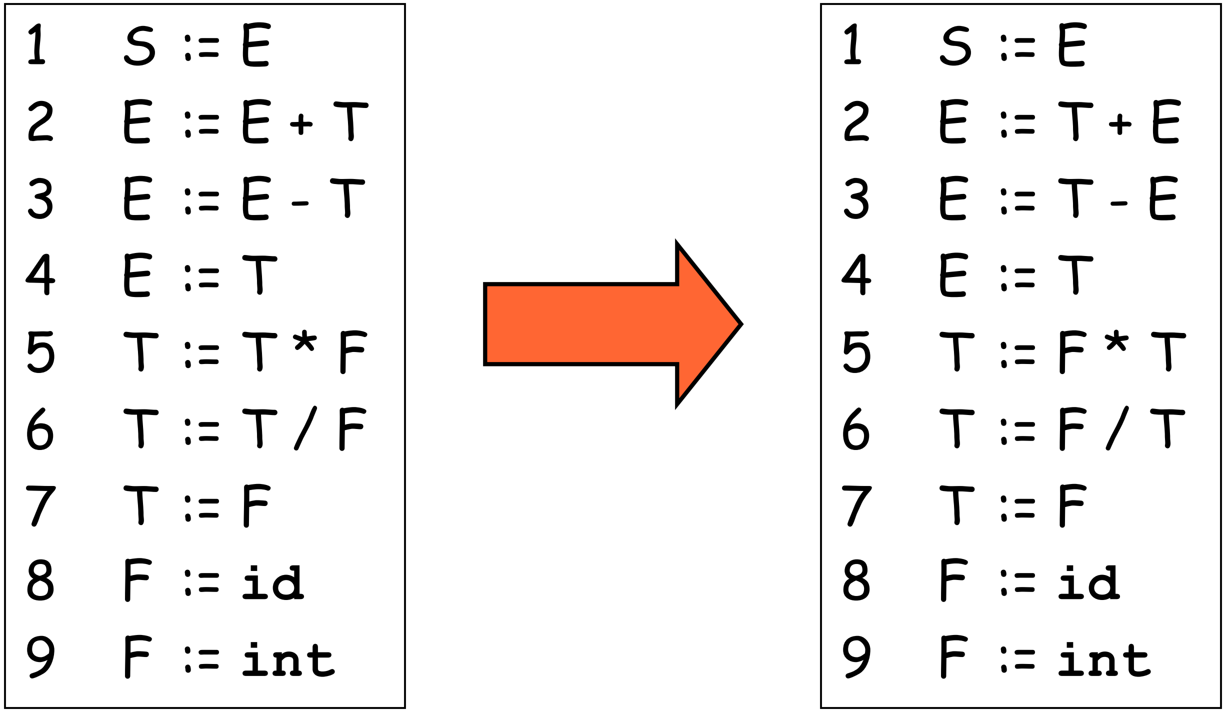
Will not terminate! Why?

grammar is left-recursive

What should we do about it?

Eliminate left-recursion

Does this work?



```
1    S := E
2    E := E + T
3    E := E - T
4    E := T
5    T := T * F
6    T := T / F
7    T := F
8    F := id
9    F := int
```

```
1    S := E
2    E := T + E
3    E := T - E
4    E := T
5    T := F * T
6    T := F / T
7    T := F
8    F := id
9    F := int
```

It is right recursive, but also right associative!

Eliminating Left-Recursion

- Given 2 productions:

  $A := A\ \alpha \mid \beta$

  Where neither α nor β start with A

  (e.g., For example, E := E + T | T)

  α β

- Make it right-recursive:

  $$\boxed{\begin{array}{l} A := \beta\ R \\ R := \alpha\ R \\ \quad \mid \end{array}}$$

  R is right recursive

- Extends to general case.

Rewriting Exp Grammar

1	S	:= E
2	E	:= E + T
3	E	:= E - T
4	E	:= T
5	T	:= T * F
6	T	:= T / F
7	T	:= F
8	F	:= **id**
9	F	:= **int**

1	S	:= E
2′	E′	:= + T E′
3′	E′	:= - T E′
4′	E′	:=
5′	T′	:= * F T′
6′	T′	:= / F T′
7′	T′	:=
8	F	:= **id**
9	F	:= **int**

2	E	:= T E′
5	T	:= F T′

Is this legible?

Try again

1	S := E
2	E := T E′
2′	E′ := + T E′
3′	E′ := - T E′
4′	E′ :=
5	T := F T′
5′	T′ := * F T′
6′	T′ := / F T′
7′	T′ :=
8	F := id
9	F := int

	S	●int$_2$ - int$_3$ * id$_x$
by 1 ⇒	E	●int$_2$ - int$_3$ * id$_x$
by 2 ⇒	T E′	●int$_2$ - int$_3$ * id$_x$
by 5 ⇒	F T′ E′	●int$_2$ - int$_3$ * id$_x$
by 9 ⇒	2 T′ E′	int$_2$ ●- int$_3$ * id$_x$
by 7′ ⇒	2 E′	int$_2$ ●- int$_3$ * id$_x$
by 3′ ⇒	2 - T E′	int$_2$ - ●int$_3$ * id$_x$
by 5 ⇒	2 - F T′ E′	int$_2$ - ●int$_3$ * id$_x$
by 9 ⇒	2 - 3 T′ E′	int$_2$ - int$_3$ ●* id$_x$
by 5′ ⇒	2 - 3 * F T′ E′	int$_2$ - int$_3$ * ●id$_x$
by 8 ⇒	2 - 3 * x T′ E′	int$_3$ * id$_x$●
		int$_3$ * id$_x$●
		int$_3$ * id$_x$●

Unlike previous time we tried this, it appears that only one production applies at a time. I.e., no backtracking needed. Why?

Lookahead

- How to pick right production?
- Lookahead in input stream for guidance
- General case: arbitrary lookahead required
- Luckily, many context-free grammers can be parsed with limited lookahead
- If we have $A \rightarrow \alpha \mid \beta$, then we want to correctly choose either $A \rightarrow \alpha$ or $A \rightarrow \beta$
- define FIRST(α) as the set of tokens that can be first symbol of α, i.e.,

  $a \in \text{FIRST}(\alpha)$ iff $\alpha \rightarrow^* a\gamma$ for some γ

Lookahead

- How to pick right production?
- If we have $A \rightarrow \alpha \mid \beta$, then we want to correctly choose either $A \rightarrow \alpha$ or $A \rightarrow \beta$
- define $\mathrm{FIRST}(\alpha)$ as the set of tokens that can be first symbol of α, i.e.,

  $a \in \mathrm{FIRST}(\alpha)$ iff $\alpha \rightarrow^* a\gamma$ for some γ
- If $A \rightarrow \alpha \mid \beta$ we want:

  $\mathrm{FIRST}(\alpha) \cap \mathrm{FIRST}(\beta) = \emptyset$
- If that is always true, we can build a predictive parser.

FIRST sets

- We use next k characters in input stream to guide the selection of the proper production.
- Given: $A := \alpha \mid \beta$ we want next input character to decide between α and β.
- FIRST(α) = set of terminals that can begin any string derived from α.
- IOW: $\mathbf{a} \in$ FIRST(α) iff $\alpha \Rightarrow^* \mathbf{a}\gamma$ for some γ

- FIRST(α) $\cap$ FIRST(β) = $\varnothing$ → no backtracking needed

Computing FIRST(α)

- Given X := A B C, FIRST(X) = FIRST(A B C)
- Can we ignore B or C?
- Consider:

```
A := a
   |
B := b
   | A
C := c
```

Computing FIRST(α)

- Given X := A B C, FIRST(X) = FIRST(A B C)
- Can we ignore B or C?
- Consider:

```
A := a
   |
B := b
   | A
C := c
```

- FIRST(X) must also include FIRST(C)
- IOW:
  - Must keep track of NTs that are nullable
  - For nullable NTs, determine FOLLOWS(NT)

nullable(A)

- nullable(A) is true if A can derive the empty string
- For example:

```
B := X Y b
X := x
   |  Y Y
Y :=
```

In this case, nullable(X) = nullable(Y) = true

nullable(B) = false

FOLLOW(A)

- FOLLOW(A) is the set of terminals that can immediately follow A in a sentential form.
- I.e.,
  a $\in$ FOLLOW(A) iff S $\Rightarrow^*$ αAaβ for some α and β

Building a Predictive Parser

- We want to know for each non-terminal which production to choose based on the next input character.
- Build a table with rows labeled by non-terminals, A, and columns labeled by terminals, a. We will put the production, A := α , in (A, a) iff
  - FIRST(α) contains a or
  - nullable(α) and FOLLOW(A) contains a

skip

The table for the robot

S := B S F

|

B := b

F := f

	FIRST	FOLLOW	nullable
S	b	$	yes
B	b	b,f	no
F	f	f,$	no

	b	f	$
S			
B			
F			

The table for the robot

S := B S F

|

B := b

F

FIRST(BSF) = b

	FIRST	FOLLOW	nullable
S	b	$	yes
B	b	b,f	no
F	f	f,$	no

nullable(ε)=true and FOLLOW(S) = $

	b	f	$
S	S:=BSF		S:=
B	B:=b		
F		F:=f	

Table f

1 S := E
2 E := T E′
2′ E′ := + T E′
3′ E′ := - T E′
4′ E′ :=
5 T := F T′
5′ T′ := * F T′
6′ T′ := / F T′
7′ T′ :=
8 F := id
9 F := int

	FIRST	FOLLOW	nullable
S	id, int	$	
E	id, int	$	
E′	+, -	$	yes
T	id, int	+,-,$	
T′	/, *	+,-,$	yes
F	id, int	/, *,$	

	+	-	*	/	id	int	$
S							
E							
E′							
T							
T′							
F							

Table f

1 S := E
2 E := T E′
2′ E′ := + T E′
3′ E′ := - T E′
4′ E′ :=
5 T := F T′
5′ T′ := * F T′
6′ T′ := / F T′
7′ T′ :=
8 F := `id`
9 F := `int`

	FIRST	FOLLOW	nullable
S	id, int	$	
E	id, int	$	
E′	+, -	$	yes
T	id, int	+,-,$	
T′	/, *	+,-,$	yes
F	id, int	/, *,$	

	+	-	*	/	id	int	$
S					:=E	:=E	
E					:=TE′	:=TE′	
E′	:=+TE′	:=-TE′					:=
T					:=FT′	:=FT′	
T′	:=	:=	:=*FT′	:=/FT′			:=
F					:=id	:=int	

Using the Table

- Each row in the table becomes a function
- For each input token with an entry: Create a series of invocations that implement the production, where
  - a non-terminal is eaten
  - a terminal becomes a recursive call
- For the blank cells implement errors

Example function

	+	-	*	/	id	int	$
S					:=E	:=E	
E					:=TE'	:=TE'	
E'	:=+TE'	:=-TE'			:=TE'	:=TE'	:=
T					[illegible]	[illegible]	
T'	:=	:=	:=*FT				
F					:=id	:=int	

How to handle errors?

```
Eprime() {
    switch (token) {
    case PLUS:     eat(PLUS); T(); Eprime(); break;
    case MINUS:    eat(MINUS); T(); Eprime(); break;
    case ID:       T(); Eprime();
    case INT:      T(); Eprime();
    default:       error();
}
```

Left-Factoring

- Predictive parsers need to make a choice based on the next terminal.
- Consider:

```
S := if E then S else S
   | if E then S
```

- When looking at **`if`**, can't decide
- so left-factor the grammar

```
S := if E then S X
X := else S
   |
```

Top-Down Parsing

- Can be constructed by hand
- LL(k) grammars can be parsed
  - Left-to-right
  - Leftmost-derivation
  - with k symbols lookahead
- Often requires
  - left-factoring
  - Elimination of left-recursion

Bottom-up parsers

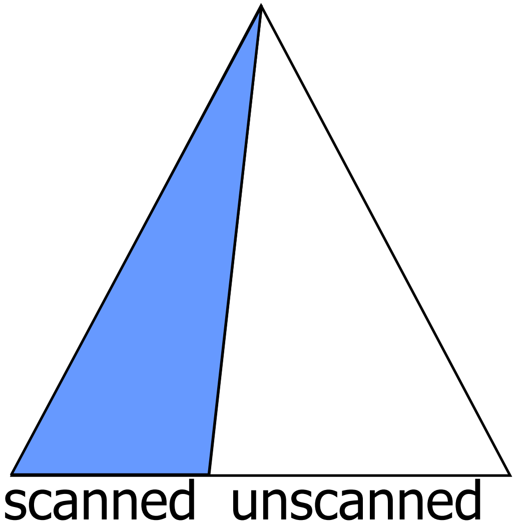
- What is the inherent restriction of top-down parsing, e.g., with LL(k) grammars?

Bottom-up parsers

- What is the inherent restriction of top-down parsing, e.g., with LL(k) grammars?
- Bottom-up parsers use the entire right-hand side of the production
- LR(k):
  – Left-to-right parse,
  – Rightmost derivation (in reverse),
  – k look ahead tokens

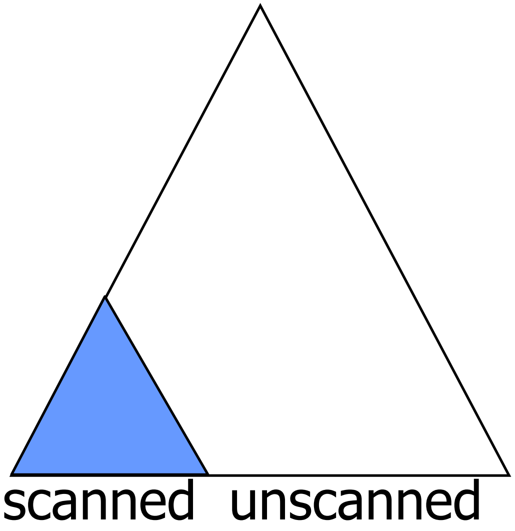
Top-down vs. Bottom-up

LL(k), recursive descent



Top-down

LR(k), shift-reduce



Bottom-up

Example - Top-down

```
S := X
X := X a
   | b
```

Is this grammar LL(k)?

How can we make it LL(k)?

```
S := X
X := b R
R := a R
   |
```

What about a bottom up parse?

Example - Bottom-up

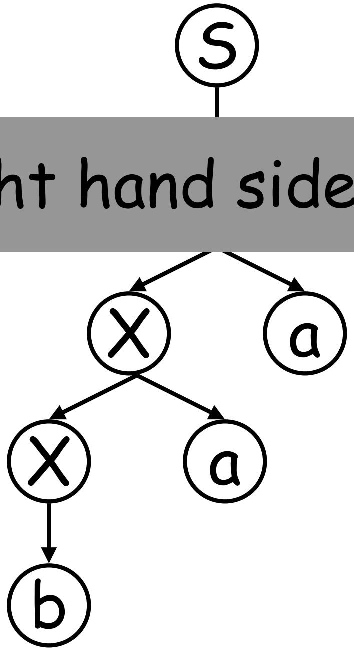
S := X
X := X a
  | b

right-most derivation:

LR parser gets to look at an entire right hand side.

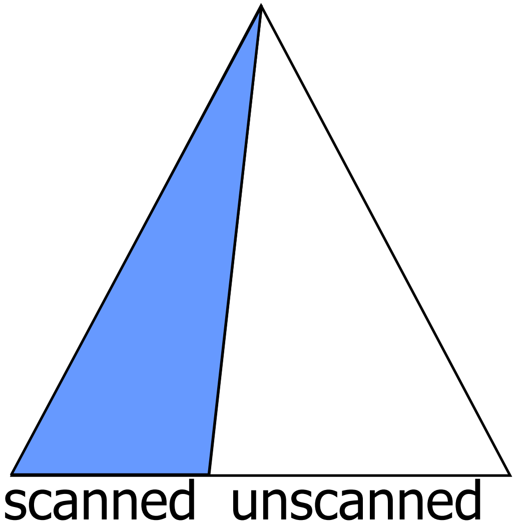
Left-to-Right, Rightmost in reverse

baa
Xaa
Xa
X
S



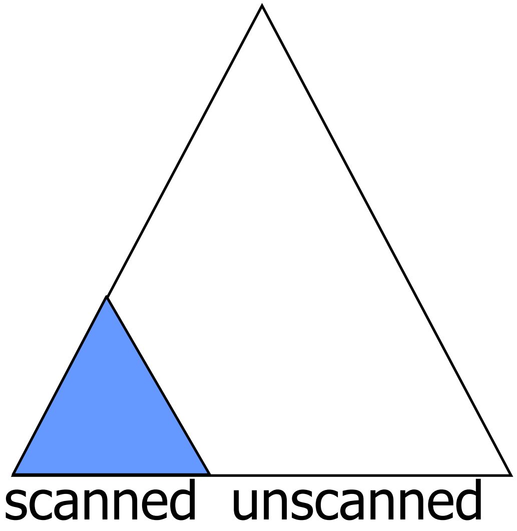
Top-down vs. Bottom-up

LL(k), recursive descent



Top-down

LR(k), shift-reduce



Bottom-up

A Rightmost Derivation

1 S := Exp

2 Exp := Exp + Term

3 Exp := Exp - Term

4 Exp := Term

5 Term := Term * Factor

6 Term := Term / Factor

7 Term := Factor

8 Factor := **id**

9 Factor := **int**

S

by 1 ⇒ Exp

by 2 ⇒ Exp + Term

by 5 ⇒ Exp + Term * Factor

by 8 ⇒ Exp + Term * $\mathtt{id}_x$

by 7 ⇒ Exp + Factor * $\mathtt{id}_x$

by 9 ⇒ Exp + $\mathtt{int}_3$ * $\mathtt{id}_x$

by 4 ⇒ Term + $\mathtt{int}_3$ * $\mathtt{id}_x$

by 7 ⇒ Factor + $\mathtt{int}_3$ * $\mathtt{id}_x$

by 9 ⇒ $\mathtt{int}_2$ + $\mathtt{int}_3$ * $\mathtt{id}_x$

A Rightmost Derivation In Reverse

$\texttt{int}_2 + \texttt{int}_3 * \texttt{id}_x$

$\text{Factor} + \texttt{int}_3 * \texttt{id}_x$

$\text{Term} + \texttt{int}_3 * \texttt{id}_x$

$\text{Exp} + \texttt{int}_3 * \texttt{id}_x$

Lets keep track of where we are in the input.

$\text{Exp} + \text{Factor} * \texttt{id}_x$

$\text{Exp} + \text{Term} * \texttt{id}_x$

$\text{Exp} + \text{Term} * \text{Factor}$

$\text{Exp} + \text{Term}$

Exp

S

A Rightmost Derivation In Reverse

$\texttt{int}_2 + \texttt{int}_3 * \texttt{id}_x$

$\text{Factor} + \texttt{int}_3 * \texttt{id}_x$

$\text{Term} + \texttt{int}_3 * \texttt{id}_x$

$\text{Exp} + \texttt{int}_3 * \texttt{id}_x$

$\text{Exp} + \text{Factor} * \texttt{id}_x$

$\text{Exp} + \text{Term} * \texttt{id}_x$

$\text{Exp} + \text{Term} * \text{Factor}$

$\text{Exp} + \text{Term}$

Exp

S

$\texttt{int}_2$ ● $+ \texttt{int}_3 * \texttt{id}_x$

Factor ● $+ \texttt{int}_3 * \texttt{id}_x$

Term ● $+ \texttt{int}_3 * \texttt{id}_x$

$\text{Exp} + \texttt{int}_3$ ● $* \texttt{id}_x$

$\text{Exp} + \text{Factor}$ ● $* \texttt{id}_x$

$\text{Exp} + \text{Term} * \texttt{id}_x$ ●

$\text{Exp} + \text{Term} * \text{Factor}$ ●

$\text{Exp} + \text{Term}$ ●

Exp ●

S ●

A Rightmost Derivation In Reverse

$\texttt{int}_2 + \texttt{int}_3 * \texttt{id}_x$

$\text{Factor} + \texttt{int}_3 * \texttt{id}_x$

$\text{Term} + \texttt{int}_3 * \texttt{id}_x$

$\text{Exp} + \texttt{int}_3 * \texttt{id}_x$

$\text{Exp} + \text{Factor} * \texttt{id}_x$

$\text{Exp} + \text{Term} * \texttt{id}$

$\text{Exp} + \text{Term} *$

$\text{Exp} + \text{Term}$

Exp

S

Lets format this differently,
<prefix of sentential form>   input

$\texttt{int}_2$ ● $+ \texttt{int}_3 * \texttt{id}_x$

Factor ● $+ \texttt{int}_3 * \texttt{id}_x$

Term ● $+ \texttt{int}_3 * \texttt{id}_x$

$\text{Exp} + \texttt{int}_3$ ● $* \texttt{id}_x$

$\text{Exp} + \text{Factor}$ ● $* \texttt{id}_x$

$\text{Exp} + \text{Term} * \texttt{id}_x$ ●

Factor ●

Exp ●

S ●

A Rightmost Derivation In Reverse

	$\texttt{int}_2 + \texttt{int}_3 * \texttt{id}_x \ \$$
$\texttt{int}_2$	$+ \texttt{int}_3 * \texttt{id}_x \ \$$
Factor	$+ \texttt{int}_3 * \texttt{id}_x \ \$$
Term	$+ \texttt{int}_3 * \texttt{id}_x \ \$$
Exp	$+ \texttt{int}_3 * \texttt{id}_x \ \$$
Exp +	$\texttt{int}_3 * \texttt{id}_x \ \$$
Exp + $\texttt{int}_3$	$* \texttt{id}_x \ \$$
Exp + Factor	$* \texttt{id}_x \ \$$
Exp + Term	$* \texttt{id}_x \ \$$
Exp + Term *	$\texttt{id}_x \ \$$
Exp + Term * $\texttt{id}_x$	$\$$
Exp + Term * Factor	$\$$
Exp + Term	$\$$
Exp	$\$$
S	$\$$

A Rightmost Derivation In Reverse

	$\texttt{int}_2 + \texttt{int}_3 * \texttt{id}_x \$$
$\texttt{int}_2$	$+ \texttt{int}_3 * \texttt{id}_x \$$
Factor	$+ \texttt{int}_3 * \texttt{id}_x \$$
Term	$+ \texttt{int}_3 * \texttt{id}_x \$$
Exp	$+ \texttt{int}_3 * \texttt{id}_x \$$
Exp +	$\texttt{int}_3 * \texttt{id}_x \$$
Exp + $\texttt{int}_3$	$* \texttt{id}_x \$$
Exp + Factor	$* \texttt{id}_x \$$
Exp + Term	$* \texttt{id}_x \$$
Exp + Term *	$\texttt{id}_x \$$
Exp + Term * $\texttt{id}_x$	\$
S	\$

LR-Parser either:

1. shifts a terminal or
2. reduces by a production.

A Rightmost Derivation In Reverse

	$\texttt{int}_2 + \texttt{int}_3 * \texttt{id}_x \ \$$	shift 2
$\texttt{int}_2$	$+ \texttt{int}_3 * \texttt{id}_x \ \$$	
Factor	$+ \texttt{int}_3 * \texttt{id}_x \ \$$	
Term	$+ \texttt{int}_3 * \texttt{id}_x \ \$$	
Exp	$+ \texttt{int}_3 * \texttt{id}_x \ \$$	
Exp +	$\texttt{int}_3 * \texttt{id}_x \ \$$	
Exp + $\texttt{int}_3$	$* \texttt{id}_x \ \$$	
Exp + Factor	$* \texttt{id}_x \ \$$	
Exp + Term	$* \texttt{id}_x \ \$$	
Exp + Term *	$\texttt{id}_x \ \$$	
Exp + Term * $\texttt{id}_x$	$\$$	
Exp + Term * Factor	$\$$	
Exp + Term	$\$$	
Exp	$\$$	
S	$\$$	

A Rightmost Derivation In Reverse

	$\texttt{int}_2 + \texttt{int}_3 * \texttt{id}_x$ \$	shift 2
$\texttt{int}_2$	$+ \texttt{int}_3 * \texttt{id}_x$ \$	reduce by F → int
Factor		
Term		
Exp		
Exp +		
Exp + $\texttt{int}_3$	$* \texttt{id}_x$ \$	
Exp + Factor	$* \texttt{id}_x$ \$	
Exp + Term	$* \texttt{id}_x$ \$	
Exp + Term *	$\texttt{id}_x$ \$	
Exp + Term * $\texttt{id}_x$	\$	
Exp + Term * Factor	\$	
Exp + Term	\$	
Exp	\$	
S	\$	

When we reduce by a production: A → β, β is on right side of sentential form.

E.g., here β is ‘int’ and production is F → int

A Rightmost Derivation In Reverse

	$\mathtt{int}_2 + \mathtt{int}_3 * \mathtt{id}_x \$$	shift 2
$\mathtt{int}_2$	$+ \mathtt{int}_3 * \mathtt{id}_x \$$	reduce by F → int
Factor	$+ \mathtt{int}_3 * \mathtt{id}_x \$$	reduce by T → F
Term	$+ \mathtt{int}_3 * \mathtt{id}_x \$$	
Exp	$+ \mathtt{int}_3 * \mathtt{id}_x \$$	
Exp +	$\mathtt{int}_3 * \mathtt{id}_x \$$	
Exp + $\mathtt{int}_3$	$* \mathtt{id}_x \$$	
Exp + Factor	$* \mathtt{id}_x \$$	
Exp + Term	$* \mathtt{id}_x \$$	
Exp + Term *	$\mathtt{id}_x \$$	
Exp + Term * $\mathtt{id}_x$	\$	
Exp + Term * Factor	\$	
Exp + Term	\$	
Exp	\$	
S	\$	

A Rightmost Derivation In Reverse

	$\mathtt{int}_2 + \mathtt{int}_3 * \mathtt{id}_x \ \$$	shift 2
$\mathtt{int}_2$	$+ \mathtt{int}_3 * \mathtt{id}_x \ \$$	reduce by F → int
Factor	$+ \mathtt{int}_3 * \mathtt{id}_x \ \$$	reduce by T → F
Term	$+ \mathtt{int}_3 * \mathtt{id}_x \ \$$	reduce by T → E
Exp	$+ \mathtt{int}_3 * \mathtt{id}_x \ \$$	
Exp +	$\mathtt{int}_3 * \mathtt{id}_x \ \$$	
Exp + $\mathtt{int}_3$	$* \mathtt{id}_x \ \$$	
Exp + Factor	$* \mathtt{id}_x \ \$$	
Exp + Term	$* \mathtt{id}_x \ \$$	
Exp + Term *	$\mathtt{id}_x \ \$$	
Exp + Term * $\mathtt{id}_x$	$\$$	
Exp + Term * Factor	$\$$	
Exp + Term	$\$$	
Exp	$\$$	
S	$\$$	

A Rightmost Derivation In Reverse

	$\mathtt{int}_2 + \mathtt{int}_3 * \mathtt{id}_x \ \$$	shift 2
$\mathtt{int}_2$	$+ \mathtt{int}_3 * \mathtt{id}_x \ \$$	reduce by F → int
Factor	$+ \mathtt{int}_3 * \mathtt{id}_x \ \$$	reduce by T → F
Term	$+ \mathtt{int}_3 * \mathtt{id}_x \ \$$	reduce by T → E
Exp	$+ \mathtt{int}_3 * \mathtt{id}_x \ \$$	shift +
Exp +	$\mathtt{int}_3 * \mathtt{id}_x \ \$$	
Exp + $\mathtt{int}_3$	$* \mathtt{id}_x \ \$$	
Exp + Factor	$* \mathtt{id}_x \ \$$	
Exp + Term	$* \mathtt{id}_x \ \$$	
Exp + Term *	$\mathtt{id}_x \ \$$	
Exp + Term * $\mathtt{id}_x$	$\$$	
Exp + Term * Factor	$\$$	
Exp + Term	$\$$	
Exp	$\$$	
S	$\$$	

A Rightmost Derivation In Reverse

	$\texttt{int}_2 + \texttt{int}_3 * \texttt{id}_x$ \$	shift 2
$\texttt{int}_2$	$+ \texttt{int}_3 * \texttt{id}_x$ \$	reduce by F → int
Factor	$+ \texttt{int}_3 * \texttt{id}_x$ \$	reduce by T → F
Term	$+ \texttt{int}_3 * \texttt{id}_x$ \$	reduce by T → E
Exp	$+ \texttt{int}_3 * \texttt{id}_x$ \$	shift +
Exp +	$\texttt{int}_3 * \texttt{id}_x$ \$	shift 3
Exp + $\texttt{int}_3$	$* \texttt{id}_x$ \$	
Exp + Factor	$* \texttt{id}_x$ \$	
Exp + Term	$* \texttt{id}_x$ \$	
Exp + Term *	$\texttt{id}_x$ \$	
Exp + Term * $\texttt{id}_x$	\$	
Exp + Term * Factor	\$	
Exp + Term	\$	
Exp	\$	
S	\$	

A Rightmost Derivation In Reverse

	$\texttt{int}_2 + \texttt{int}_3 * \texttt{id}_x \ \$$	shift 2
$\texttt{int}_2$	$+ \texttt{int}_3 * \texttt{id}_x \ \$$	reduce by F → int
Factor	$+ \texttt{int}_3 * \texttt{id}_x \ \$$	reduce by T → F
Term	$+ \texttt{int}_3 * \texttt{id}_x \ \$$	reduce by T → E
Exp	$+ \texttt{int}_3 * \texttt{id}_x \ \$$	shift +
Exp +	$\texttt{int}_3 * \texttt{id}_x \ \$$	shift 3
Exp + $\texttt{int}_3$	$* \texttt{id}_x \ \$$	reduce by F → int
Exp + Factor	$* \texttt{id}_x \ \$$	
Exp + Term	$* \texttt{id}_x \ \$$	
Exp + Term *	$\texttt{id}_x \ \$$	
Exp + Term * $\texttt{id}_x$	$	
Exp + Term * Factor	$	
Exp + Term	$	
Exp	$	
S	$	

A Rightmost Derivation In Reverse

	$\texttt{int}_2 + \texttt{int}_3 * \texttt{id}_x \ \$$	shift 2
$\texttt{int}_2$	$+ \texttt{int}_3 * \texttt{id}_x \ \$$	reduce by F → int
Factor	$+ \texttt{int}_3 * \texttt{id}_x \ \$$	reduce by T → F
Term	$+ \texttt{int}_3 * \texttt{id}_x \ \$$	reduce by T → E
Exp	$+ \texttt{int}_3 * \texttt{id}_x \ \$$	shift +
Exp +	$\texttt{int}_3 * \texttt{id}_x \ \$$	shift 3
Exp + $\texttt{int}_3$	$* \texttt{id}_x \ \$$	reduce by F → int
Exp + Factor	$* \texttt{id}_x \ \$$	reduce by F → T
Exp + Term	$* \texttt{id}_x \ \$$	
Exp + Term *	$\texttt{id}_x \ \$$	
Exp + Term * $\texttt{id}_x$	$\$$	
Exp + Term * Factor	$\$$	
Exp + Term	$\$$	
Exp	$\$$	
S	$\$$	

A Rightmost Derivation In Reverse

	$\texttt{int}_2 + \texttt{int}_3 * \texttt{id}_x \ \$$	shift 2
$\texttt{int}_2$	$+ \texttt{int}_3 * \texttt{id}_x \ \$$	reduce by F → int
Factor	$+ \texttt{int}_3 * \texttt{id}_x \ \$$	reduce by T → F
Term	$+ \texttt{int}_3 * \texttt{id}_x \ \$$	reduce by T → E
Exp	$+ \texttt{int}_3 * \texttt{id}_x \ \$$	shift +
Exp +	$\texttt{int}_3 * \texttt{id}_x \ \$$	shift 3
Exp + $\texttt{int}_3$	$* \texttt{id}_x \ \$$	reduce by F → int
Exp + Factor	$* \texttt{id}_x \ \$$	reduce by F → T
Exp + Term	$* \texttt{id}_x \ \$$	shift *
Exp + Term *	$\texttt{id}_x \ \$$	
Exp + Term * $\texttt{id}_x$	$\$$	
Exp + Term * Factor	$\$$	
Exp + Term	$\$$	
Exp	$\$$	
S	$\$$	

A Rightmost Derivation In Reverse

	$\texttt{int}_2 + \texttt{int}_3 * \texttt{id}_x \$$	shift 2
$\texttt{int}_2$	$+ \texttt{int}_3 * \texttt{id}_x \$$	reduce by F → int
Factor	$+ \texttt{int}_3 * \texttt{id}_x \$$	reduce by T → F
Term	$+ \texttt{int}_3 * \texttt{id}_x \$$	reduce by T → E
Exp	$+ \texttt{int}_3 * \texttt{id}_x \$$	shift +
Exp +	$\texttt{int}_3 * \texttt{id}_x \$$	shift 3
Exp + $\texttt{int}_3$	$* \texttt{id}_x \$$	reduce by F → int
Exp + Factor	$* \texttt{id}_x \$$	reduce by F → T
Exp + Term	$* \texttt{id}_x \$$	shift *
Exp + Term *	$\texttt{id}_x \$$	shift x
Exp + Term * $\texttt{id}_x$	$\$$	
Exp + Term * Factor	$\$$	
Exp + Term	$\$$	
Exp	$\$$	
S	$\$$	

A Rightmost Derivation In Reverse

	$\mathtt{int}_2 + \mathtt{int}_3 * \mathtt{id}_x \$$	shift 2
$\mathtt{int}_2$	$+ \mathtt{int}_3 * \mathtt{id}_x \$$	reduce by F → int
Factor	$+ \mathtt{int}_3 * \mathtt{id}_x \$$	reduce by T → F
Term	$+ \mathtt{int}_3 * \mathtt{id}_x \$$	reduce by T → E
Exp	$+ \mathtt{int}_3 * \mathtt{id}_x \$$	shift +
Exp +	$\mathtt{int}_3 * \mathtt{id}_x \$$	shift 3
Exp + $\mathtt{int}_3$	$* \mathtt{id}_x \$$	reduce by F → int
Exp + Factor	$* \mathtt{id}_x \$$	reduce by F → T
Exp + Term	$* \mathtt{id}_x \$$	shift *
Exp + Term *	$\mathtt{id}_x \$$	shift x
Exp + Term * $\mathtt{id}_x$	$\$$	reduce by F → id
Exp + Term * Factor	$\$$	
Exp + Term	$\$$	
Exp	$\$$	
S	$\$$	

A Rightmost Derivation In Reverse

	$\texttt{int}_2 + \texttt{int}_3 * \texttt{id}_x \$$	shift 2
$\texttt{int}_2$	$+ \texttt{int}_3 * \texttt{id}_x \$$	reduce by F → int
Factor	$+ \texttt{int}_3 * \texttt{id}_x \$$	reduce by T → F
Term	$+ \texttt{int}_3 * \texttt{id}_x \$$	reduce by T → E
Exp	$+ \texttt{int}_3 * \texttt{id}_x \$$	shift +
Exp +	$\texttt{int}_3 * \texttt{id}_x \$$	shift 3
Exp + $\texttt{int}_3$	$* \texttt{id}_x \$$	reduce by F → int
Exp + Factor	$* \texttt{id}_x \$$	reduce by F → T
Exp + Term	$* \texttt{id}_x \$$	shift *
Exp + Term *	$\texttt{id}_x \$$	shift x
Exp + Term * $\texttt{id}_x$	$\$$	reduce by F → id
Exp + Term * Factor	$\$$	reduce by T → T * F
Exp + Term	$\$$	
Exp	$\$$	
S	$\$$	

A Rightmost Derivation In Reverse

	$\mathtt{int}_2$ + $\mathtt{int}_3$ * $\mathtt{id}_x$ \$	shift 2
$\mathtt{int}_2$	+ $\mathtt{int}_3$ * $\mathtt{id}_x$ \$	reduce by F → int
Factor	+ $\mathtt{int}_3$ * $\mathtt{id}_x$ \$	reduce by T → F
Term	+ $\mathtt{int}_3$ * $\mathtt{id}_x$ \$	reduce by T → E
Exp	+ $\mathtt{int}_3$ * $\mathtt{id}_x$ \$	shift +
Exp +	$\mathtt{int}_3$ * $\mathtt{id}_x$ \$	shift 3
Exp + $\mathtt{int}_3$	* $\mathtt{id}_x$ \$	reduce by F → int
Exp + Factor	* $\mathtt{id}_x$ \$	reduce by F → T
Exp + Term	* $\mathtt{id}_x$ \$	shift *
Exp + Term *	$\mathtt{id}_x$ \$	shift x
Exp + Term * $\mathtt{id}_x$	\$	reduce by F → id
Exp + Term * Factor	\$	reduce by T → T * F
Exp + Term	\$	reduce by E → E + T
Exp	\$	
S	\$	

A Rightmost Derivation In Reverse

	$\mathtt{int}_2 + \mathtt{int}_3 * \mathtt{id}_x$ \$	shift 2
$\mathtt{int}_2$	$+ \mathtt{int}_3 * \mathtt{id}_x$ \$	reduce by F → int
Factor	$+ \mathtt{int}_3 * \mathtt{id}_x$ \$	reduce by T → F
Term	$+ \mathtt{int}_3 * \mathtt{id}_x$ \$	reduce by T → E
Exp	$+ \mathtt{int}_3 * \mathtt{id}_x$ \$	shift +
Exp +	$\mathtt{int}_3 * \mathtt{id}_x$ \$	shift 3
Exp + $\mathtt{int}_3$	$* \mathtt{id}_x$ \$	reduce by F → int
Exp + Factor	$* \mathtt{id}_x$ \$	reduce by F → T
Exp + Term	$* \mathtt{id}_x$ \$	shift *
Exp + Term *	$\mathtt{id}_x$ \$	shift x
Exp + Term * $\mathtt{id}_x$	\$	reduce by F → id
Exp + Term * Factor	\$	reduce by T → T * F
Exp + Term	\$	reduce by E → E + T
Exp	\$	reduce by S → E
S	\$	

A Rightmost Derivation In Reverse

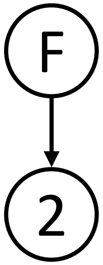
	int$_2$ + **int**$_3$ * **id**$_x$ \$	shift 2
int$_2$	+ **int**$_3$ * **id**$_x$ \$	reduce by F → int
Factor	+ **int**$_3$ * **id**$_x$ \$	reduce by T → F
Term	+ **int**$_3$ * **id**$_x$ \$	reduce by T → E
Exp	+ **int**$_3$ * **id**$_x$ \$	shift +
Exp +	**int**$_3$ * **id**$_x$ \$	shift 3
Exp + **int**$_3$	* **id**$_x$ \$	reduce by F → int
Exp + Factor	* **id**$_x$ \$	reduce by F → T
Exp + Term	* **id**$_x$ \$	shift *
Exp + Term *	**id**$_x$ \$	shift x
Exp + Term * **id**$_x$	\$	reduce by F → id
Exp + Term * Factor	\$	reduce by T → T * F
Exp + Term	\$	reduce by E → E + T
Exp	\$	reduce by S → E
S	\$	accept!

A Rightmost Derivation In Reverse

	$\texttt{int}_2 + \texttt{int}_3 * \texttt{id}_x \$$	shift 2
$\texttt{int}_2$	$+ \texttt{int}_3 * \texttt{id}_x \$$	
Factor	$+ \texttt{int}_3 * \texttt{id}_x \$$	
Term	$+ \texttt{int}_3 * \texttt{id}_x \$$	
Exp	$+ \texttt{int}_3 * \texttt{id}_x \$$	
Exp +	$\texttt{int}_3 * \texttt{id}_x \$$	
Exp + $\texttt{int}_3$	$* \texttt{id}_x \$$	
Exp + Factor	$* \texttt{id}_x \$$	
Exp + Term	$* \texttt{id}_x \$$	
Exp + Term *	$\texttt{id}_x \$$	
Exp + Term * $\texttt{id}_x$	$\$$	
Exp + Term * Factor	$\$$	
Exp + Term	$\$$	
Exp	$\$$	②
S	$\$$	

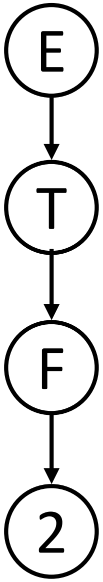
A Rightmost Derivation In Reverse

	$\texttt{int}_2 + \texttt{int}_3 * \texttt{id}_x \$$	shift 2
$\texttt{int}_2$	$+ \texttt{int}_3 * \texttt{id}_x \$$	reduce by F → int
Factor	$+ \texttt{int}_3 * \texttt{id}_x \$$	
Term	$+ \texttt{int}_3 * \texttt{id}_x \$$	
Exp	$+ \texttt{int}_3 * \texttt{id}_x \$$	
Exp +	$\texttt{int}_3 * \texttt{id}_x \$$	
Exp + $\texttt{int}_3$	$* \texttt{id}_x \$$	
Exp + Factor	$* \texttt{id}_x \$$	
Exp + Term	$* \texttt{id}_x \$$	
Exp + Term *	$\texttt{id}_x \$$	
Exp + Term * $\texttt{id}_x$	$\$$	
Exp + Term * Factor	$\$$	
Exp + Term	$\$$	
Exp	$\$$	
S	$\$$	



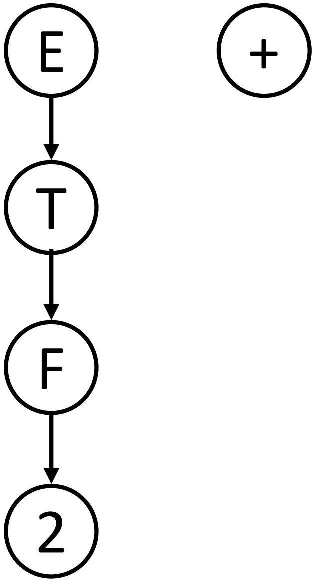
A Rightmost Derivation In Reverse

	$\mathtt{int}_2 + \mathtt{int}_3 * \mathtt{id}_x \$$	shift 2
$\mathtt{int}_2$	$+ \mathtt{int}_3 * \mathtt{id}_x \$$	reduce by F → int
Factor	$+ \mathtt{int}_3 * \mathtt{id}_x \$$	reduce by T → F
Term	$+ \mathtt{int}_3 * \mathtt{id}_x \$$	reduce by T → E
Exp	$+ \mathtt{int}_3 * \mathtt{id}_x \$$	
Exp +	$\mathtt{int}_3 * \mathtt{id}_x \$$	
Exp + $\mathtt{int}_3$	$* \mathtt{id}_x \$$	
Exp + Factor	$* \mathtt{id}_x \$$	
Exp + Term	$* \mathtt{id}_x \$$	
Exp + Term *	$\mathtt{id}_x \$$	
Exp + Term * $\mathtt{id}_x$	\$	
Exp + Term * Factor	\$	
Exp + Term	\$	
Exp	\$	
S	\$	



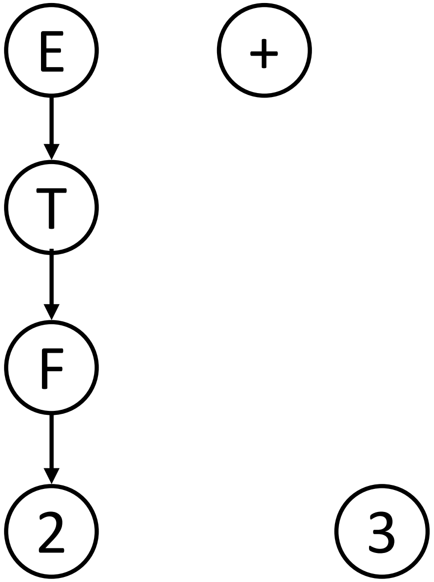
A Rightmost Derivation In Reverse

	$\texttt{int}_2 + \texttt{int}_3 * \texttt{id}_x \$$	shift 2
$\texttt{int}_2$	$+ \texttt{int}_3 * \texttt{id}_x \$$	reduce by F → int
Factor	$+ \texttt{int}_3 * \texttt{id}_x \$$	reduce by T → F
Term	$+ \texttt{int}_3 * \texttt{id}_x \$$	reduce by T → E
Exp	$+ \texttt{int}_3 * \texttt{id}_x \$$	shift +
Exp +	$\texttt{int}_3 * \texttt{id}_x \$$	
Exp + $\texttt{int}_3$	$* \texttt{id}_x \$$	
Exp + Factor	$* \texttt{id}_x \$$	
Exp + Term	$* \texttt{id}_x \$$	
Exp + Term *	$\texttt{id}_x \$$	
Exp + Term * $\texttt{id}_x$	$\$$	
Exp + Term * Factor	$\$$	
Exp + Term	$\$$	
Exp	$\$$	
S	$\$$	



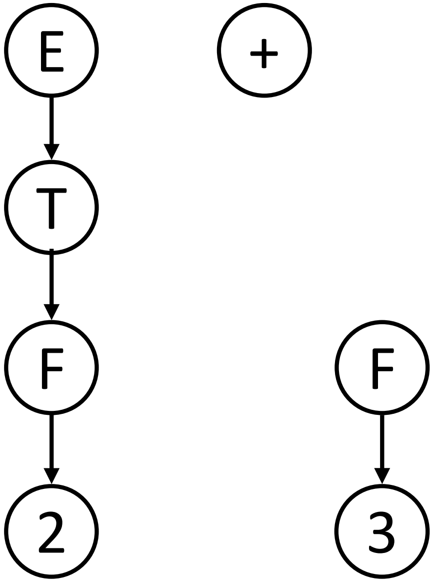
A Rightmost Derivation In Reverse

	int$_{2}$ + **int**$_{3}$ * **id**$_{x}$ \$	shift 2
int$_{2}$	+ **int**$_{3}$ * **id**$_{x}$ \$	reduce by F → int
Factor	+ **int**$_{3}$ * **id**$_{x}$ \$	reduce by T → F
Term	+ **int**$_{3}$ * **id**$_{x}$ \$	reduce by T → E
Exp	+ **int**$_{3}$ * **id**$_{x}$ \$	shift +
Exp +	**int**$_{3}$ * **id**$_{x}$ \$	shift 3
Exp + **int**$_{3}$	* **id**$_{x}$ \$	
Exp + Factor	* **id**$_{x}$ \$	
Exp + Term	* **id**$_{x}$ \$	
Exp + Term *	**id**$_{x}$ \$	
Exp + Term * **id**$_{x}$	\$	
Exp + Term * Factor	\$	
Exp + Term	\$	
Exp	\$	
S	\$	



A Rightmost Derivation In Reverse

	$\mathtt{int}_2 + \mathtt{int}_3 * \mathtt{id}_x$ \$	shift 2
$\mathtt{int}_2$	$+ \mathtt{int}_3 * \mathtt{id}_x$ \$	reduce by F → int
Factor	$+ \mathtt{int}_3 * \mathtt{id}_x$ \$	reduce by T → F
Term	$+ \mathtt{int}_3 * \mathtt{id}_x$ \$	reduce by T → E
Exp	$+ \mathtt{int}_3 * \mathtt{id}_x$ \$	shift +
Exp +	$\mathtt{int}_3 * \mathtt{id}_x$ \$	shift 3
Exp + $\mathtt{int}_3$	$* \mathtt{id}_x$ \$	reduce by F → int
Exp + Factor	$* \mathtt{id}_x$ \$	
Exp + Term	$* \mathtt{id}_x$ \$	
Exp + Term *	$\mathtt{id}_x$ \$	
Exp + Term * $\mathtt{id}_x$	\$	
Exp + Term * Factor	\$	
Exp + Term	\$	
Exp	\$	
S	\$	



Handles

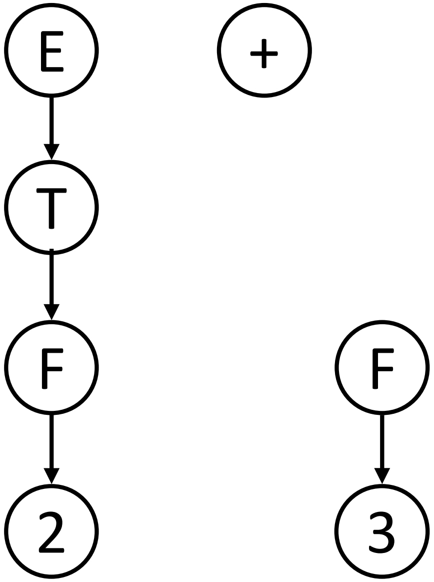
- LR parsing is handle pruning
- LR parsing finds a rightmost derivation (in reverse)
- A handle in γ, a right-hand sentential form, is
  - a position in γ matching β
  - a production $A \rightarrow \beta$

$$S \rightarrow^* \alpha A w \rightarrow \alpha \beta w$$

- if a grammar is unambiguous, then every γ has exactly 1 handle

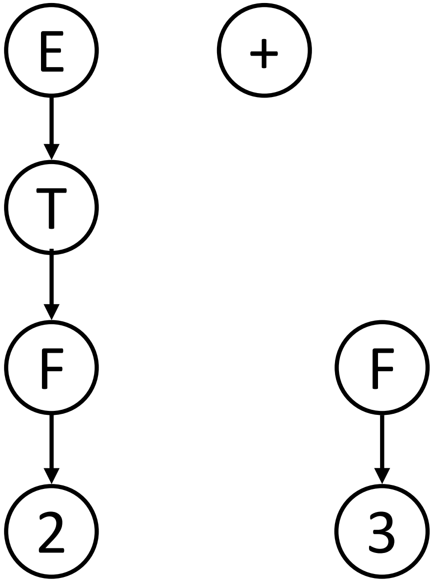
A Rightmost Derivation In Reverse

	$\texttt{int}_2$ + $\texttt{int}_3$ * $\texttt{id}_x$ $	shift 2
$\texttt{int}_2$	+ $\texttt{int}_3$ * $\texttt{id}_x$ $	reduce by F → int
Factor	+ $\texttt{int}_3$ * $\texttt{id}_x$ $	reduce by T → F
Term	+ $\texttt{int}_3$ * $\texttt{id}_x$ $	reduce by T → E
Exp	+ $\texttt{int}_3$ * $\texttt{id}_x$ $	shift +
Exp +	$\texttt{int}_3$ * $\texttt{id}_x$ $	shift 3
Exp + $\texttt{int}_3$	* $\texttt{id}_x$ $	reduce by F → int
Exp + Factor	* $\texttt{id}_x$ $	
Exp + Term	* $\texttt{id}_x$ $	
Exp + Term *	$\texttt{id}_x$ $	
Exp + Term * $\texttt{id}_x$	$	
Exp + Term * Factor	$	
Exp + Term	$	
Exp	$	
S	$	



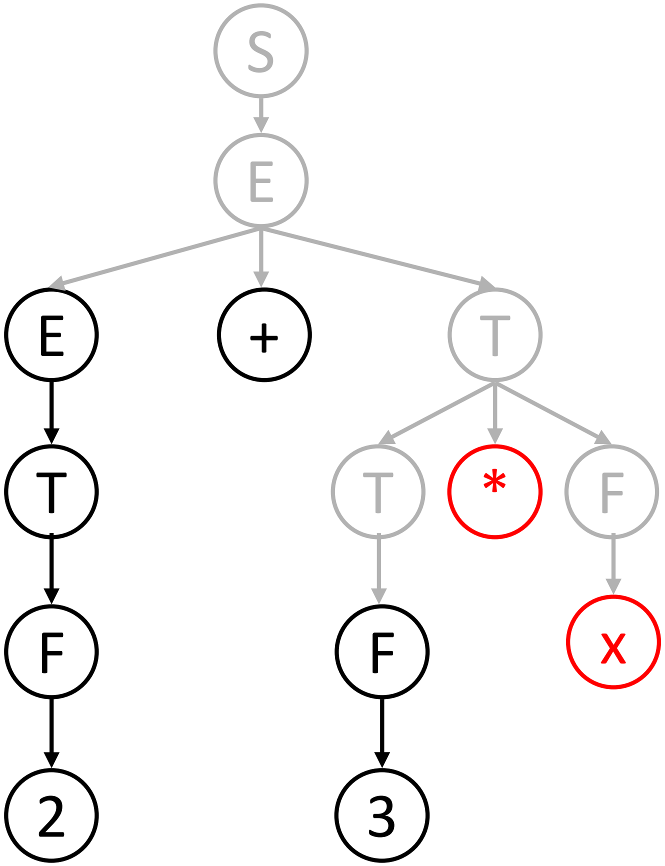
A Rightmost Derivation In Reverse

Where is next handle?	$\texttt{int}_2 + \texttt{int}_3 * \texttt{id}_x \$$	shift 2
$\texttt{int}_2$	$+ \texttt{int}_3 * \texttt{id}_x \$$	reduce by F → int
Factor	$+ \texttt{int}_3 * \texttt{id}_x \$$	reduce by T → F
Term	$+ \texttt{int}_3 * \texttt{id}_x \$$	reduce by T → E
Exp	$+ \texttt{int}_3 * \texttt{id}_x \$$	shift +
Exp +	$\texttt{int}_3 * \texttt{id}_x \$$	shift 3
Exp + $\texttt{int}_3$	$* \texttt{id}_x \$$	reduce by F → int
Exp + Factor	$* \texttt{id}_x \$$	
Exp + Term	$* \texttt{id}_x \$$	
Exp + Term *	$\texttt{id}_x \$$	
Exp + Term * $\texttt{id}_x$	$\$$	
Exp + Term * Factor	$\$$	
Exp + Term	$\$$	
Exp	$\$$	
S	$\$$	



A Rightmost Derivation In Reverse

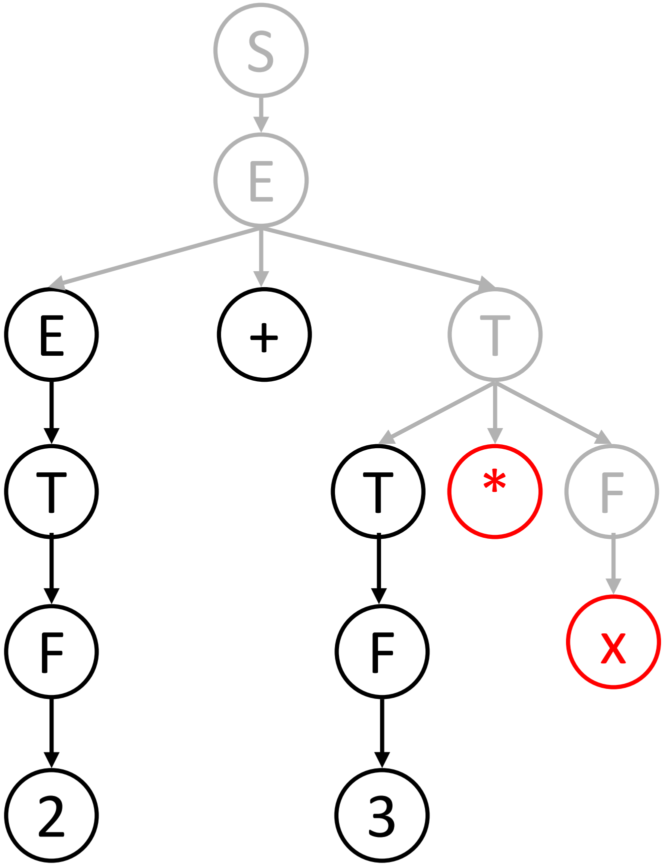
Where is next handle?	$\texttt{int}_2 + \texttt{int}_3 * \texttt{id}_x\ \$$
$\texttt{int}_2$	$+ \texttt{int}_3 * \texttt{id}_x\ \$$
Factor	$+ \texttt{int}_3 * \texttt{id}_x\ \$$
Term	$+ \texttt{int}_3 * \texttt{id}_x\ \$$
Exp	$+ \texttt{int}_3 * \texttt{id}_x\ \$$
Exp +	$\texttt{int}_3 * \texttt{id}_x\ \$$
Exp + $\texttt{int}_3$	$* \texttt{id}_x\ \$$
Exp + Factor	$* \texttt{id}_x\ \$$
Exp + Term	$* \texttt{id}_x\ \$$
Exp + Term *	$\texttt{id}_x\ \$$
Exp + Term * $\texttt{id}_x$	$\$$
Exp + Term * Factor	$\$$
Exp + Term	$\$$
Exp	$\$$
S	$\$$



A Rightmost Derivation In Reverse

Where is next handle? E+F*x and T→ F $_x$ \$

$\texttt{int}_2$	$+\ \texttt{int}_3 * \texttt{id}_x\ \$$
Factor	$+\ \texttt{int}_3 * \texttt{id}_x\ \$$
Term	$+\ \texttt{int}_3 * \texttt{id}_x\ \$$
Exp	$+\ \texttt{int}_3 * \texttt{id}_x\ \$$
Exp +	$\texttt{int}_3 * \texttt{id}_x\ \$$
Exp + $\texttt{int}_3$	$* \texttt{id}_x\ \$$
Exp + Factor	$* \texttt{id}_x\ \$$
Exp + Term	$* \texttt{id}_x\ \$$
Exp + Term *	$\texttt{id}_x\ \$$
Exp + Term * $\texttt{id}_x$	\$
Exp + Term * Factor	\$
Exp + Term	\$
Exp	\$
S	\$



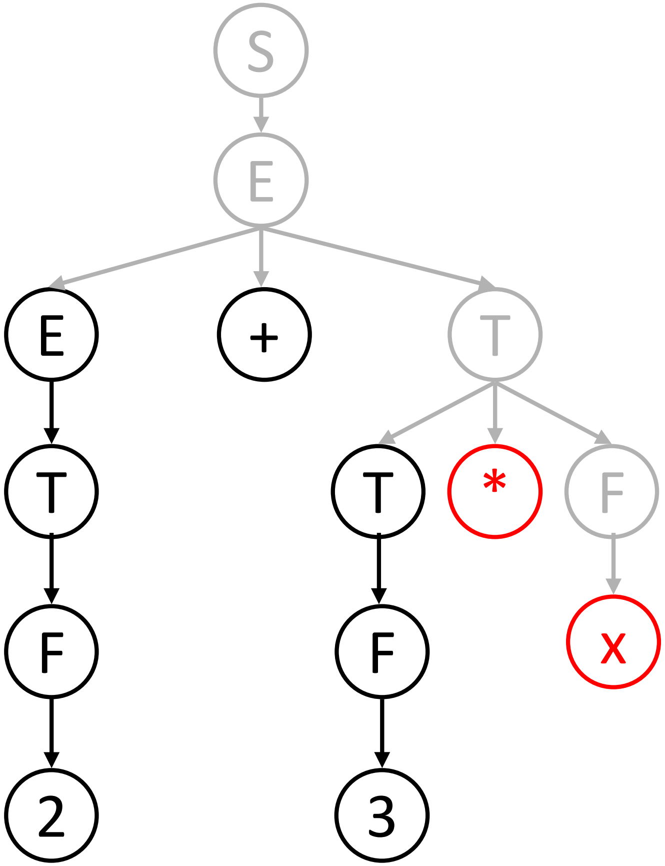
Handle Pruning

- LR parsing consists of
  - shifting til there is a handle on the top of the stack
  - reducing handle
- Key is handle is always on top of stack, i.e., if β is a handle with $A \rightarrow \beta$, then β can be found on top of stack.

A Rightmost Derivation In Reverse

	$\mathtt{int}_2 + \mathtt{int}_3 * \mathtt{id}_x \$$
$\mathtt{int}_2$	$+ \mathtt{int}_3 * \mathtt{id}_x \$$
Factor	$+ \mathtt{int}_3 * \mathtt{id}_x \$$
Term	$+ \mathtt{int}_3 * \mathtt{id}_x \$$
Exp	$+ \mathtt{int}_3 * \mathtt{id}_x \$$
Exp +	$\mathtt{int}_3 * \mathtt{id}_x \$$
Exp + $\mathtt{int}_3$	$* \mathtt{id}_x \$$
Exp + Factor	$* \mathtt{id}_x \$$
Exp + Term	$* \mathtt{id}_x \$$
Exp + Term *	$\mathtt{id}_x \$$
Exp + Term * $\mathtt{id}_x$	$\$$
Exp + Term * Factor	$\$$
Exp + Term	$\$$
Exp	$\$$
S	$\$$

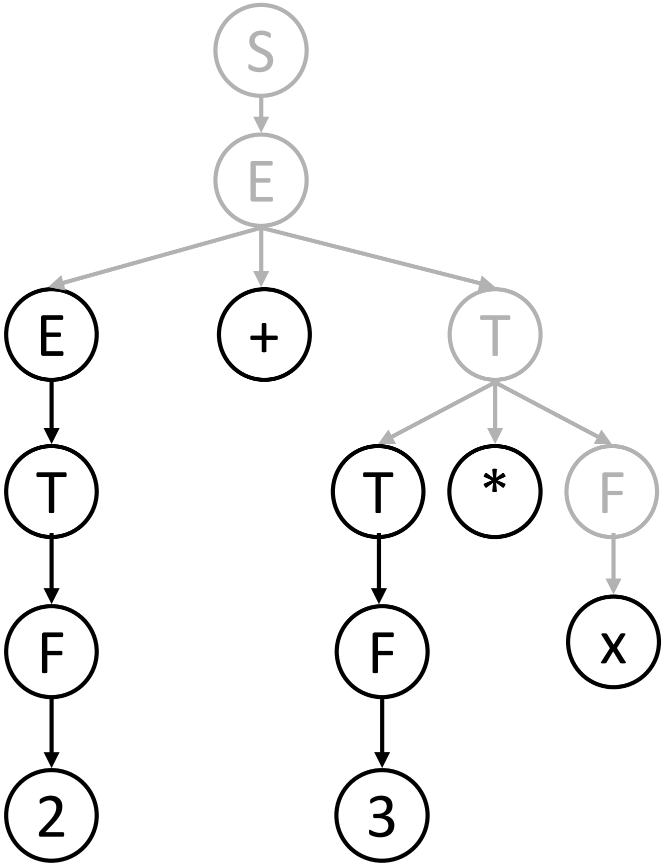
top of stack does not have a handle, so must shift.



A Rightmost Derivation In Reverse

	$\texttt{int}_2 + \texttt{int}_3 * \texttt{id}_x \ \$$
$\texttt{int}_2$	$+ \texttt{int}_3 * \texttt{id}_x \ \$$
Factor	$+ \texttt{int}_3 * \texttt{id}_x \ \$$
Term	$+ \texttt{int}_3 * \texttt{id}_x \ \$$
Exp	$+ \texttt{int}_3 * \texttt{id}_x \ \$$
Exp +	$\texttt{int}_3 * \texttt{id}_x \ \$$
Exp + $\texttt{int}_3$	$* \texttt{id}_x \ \$$
Exp + Factor	$* \texttt{id}_x \ \$$
Exp + Term	$* \texttt{id}_x \ \$$
Exp + Term *	$\texttt{id}_x \ \$$
Exp + Term * $\texttt{id}_x$	$\$$
Exp + Term * Factor	$\$$
Exp + Term	$\$$
Exp	$\$$
S	$\$$

Now, x is a handle.

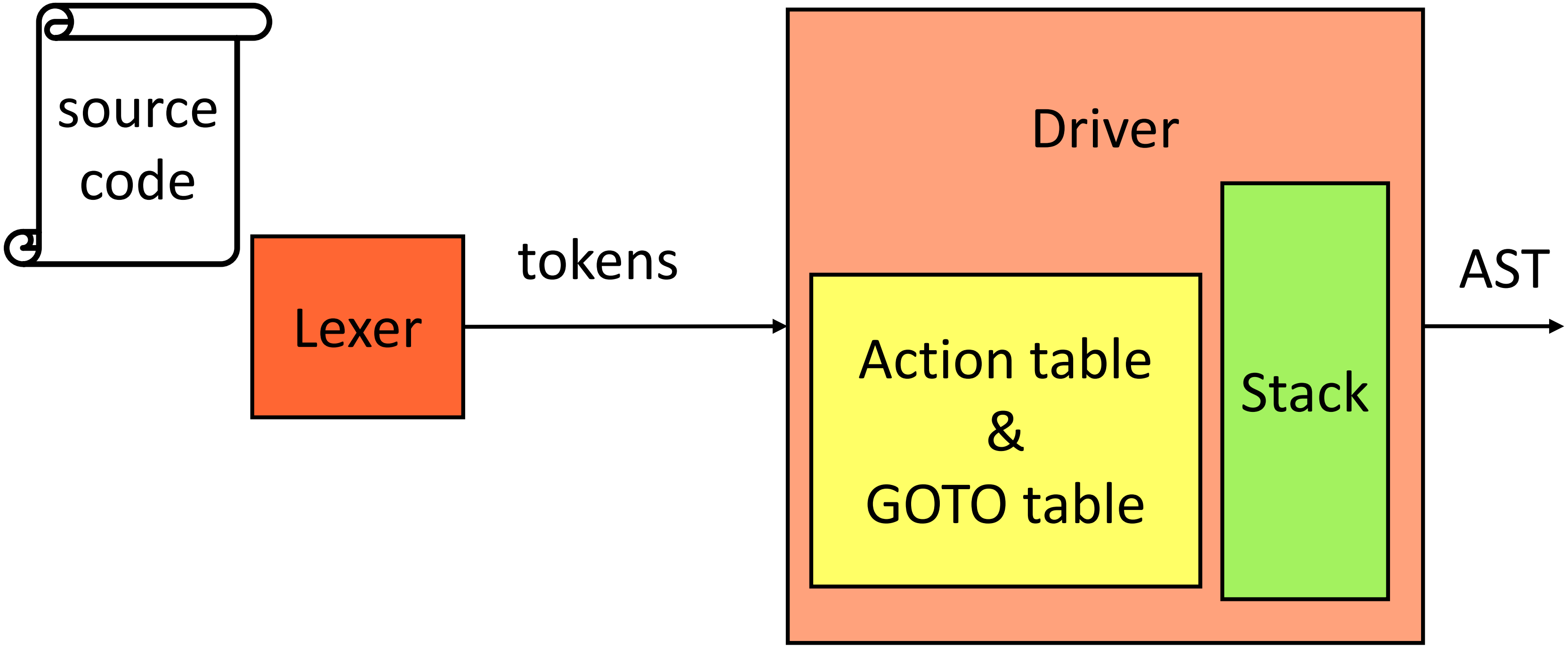


A Shift-Reduce Parser

- Stack holds the viable prefixes.
- input stream holds remaining source
- Four actions:
  - shift: push token from input stream onto stack
  - reduce: right-end of a handle (β of $A \rightarrow \beta$) is at top of stack, pop handle (β), push A
  - accept: success
  - error: syntax error discovered

Key is recognizing handles efficiently

Table-driven LR(k) parsers



Push down automata: FSM with stack

Parser Loop

Driver

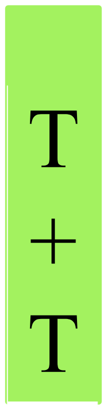
- Same code regardless of grammar
  - only tables change
- (Very) General Algorithm:
  - Based on table contents, top of stack, and current input character either
    - **shift**: pushes onto stack, reads next token
    - **reduce**: manipulate stack to simplify representation of already scanned input
    - **accept**: successfully scanned entire input
    - **error**: input not in language

Stack

Stack

- Represents the scanned input
- Contents?
  - Reduced nonterminals not enough
  - Must store previously seen *states*
    - the context of the current position
  - In fact, nonterminals unnecessary
    - include for readability

x + y• + z


T

+

T

Parser Tables

Action table
&
GOTO table

Action table

- given state *s* and **terminal** *a* tells parser loop what action (shift, reduce, accept, reject) to perform

Goto table

- used when performing reduction; given a state *s* and **nonterminal** *X* says what state to transition to

Parser Tables

Action table
&
GOTO table

s*N*	push state *N* onto stack
r*R*	reduce by rule *R*
g*N*	goto state *N*
a	accept
	error

	action			goto	
state	*ident*	+	$	E	T
0	s3			g1	g2
1			a		
2		s4	r2		
3		r3	r3		
4	s3			g5	g2
5			r1		

Parser Loop Revisited

Driver

```
while(true)
   s = state on top of stack
   a = current input token
   if(action[s][a] == sN)                shift
      push N
      read next input token
   else if(action[s][a] == rR)           reduce
      pop rhs of rule R from stack
      X = lhs of rule R
      N = state on top of stack
      push goto[N][X]
   else if(action[s][a] == a)            accept
      return success
   else                                  error
      return failure
```

Example

	action			goto	
state	*ident*	+	$	E	T
0	s3			g1	g2
1			a		
2		s4	r2		
3		r3	r3		
4	s3			g5	g2
5			r1		

0 S → E$

1 E → T + E

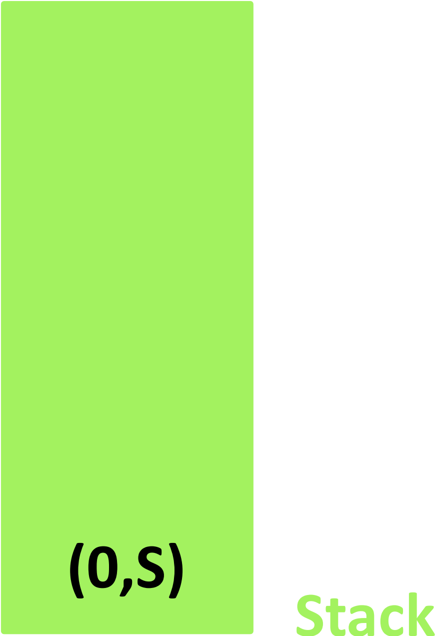
2 E → T

3 T → *identifier*

Current input token = **x**

State on top of the stack = **0**

x + y$



Example

state	action			goto	
	ident	+	$	E	T
0	s3			g1	g2
1			a		
2		s4	r2		
3		r3	r3		
4	s3			g5	g2
5			r1		

0 S → E$

1 E → T + E

2 E → T

3 T → *identifier*

Current input token = **+**

State on top of the stack = **3**

(3,x)
(0,S)

x + y$

Example

	action			goto	
state	*ident*	+	$	E	T
0	s3			g1	g2
1			a		
2		s4	r2		
3		r3	r3		
4	s3			g5	g2
5			r1		

0 S → E$

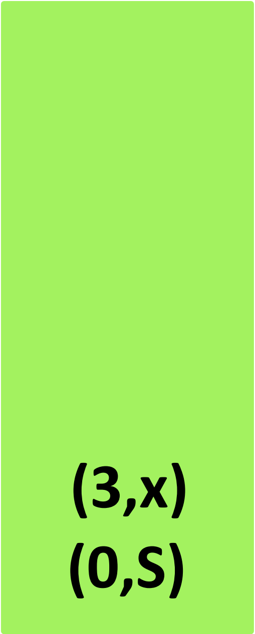
1 E → T + E

2 E → T

3 T → *identifier*

Current input token = **+**

State on top of the stack = **3**


(3,x)
(0,S)

x + y$

Example

	action			goto	
state	*ident*	+	$	E	T
0	s3			g1	g2
1			a		
2		s4	r2		
3		r3	r3		
4	s3			g5	g2
5			r1		

0 S → E$

1 E → T + E

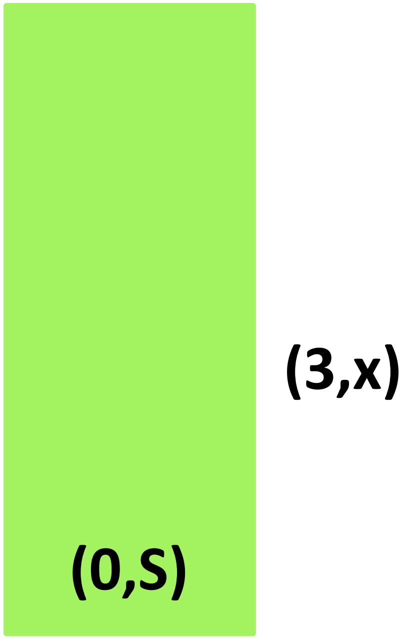
2 E → T

3 T → *identifier*

Current input token = **+**

State on top of the stack = **3**

x + y$



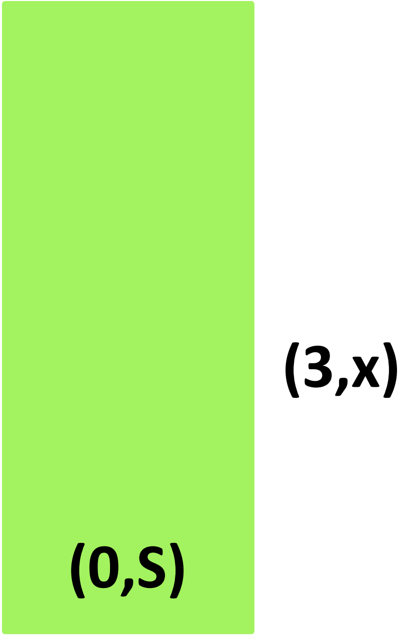
Example

	action			goto	
state	*ident*	+	$	E	T
0	s3			g1	g2
1			a		
2		s4	r2		
3		r3	r3		
4	s3			g5	g2
5			r1		

0 S → E$
1 E → T + E
2 E → T
3 T → *identifier*

Current input token = **+**
State on top of the stack = **0**

x + y$



Example

state	action			goto	
	ident	+	$	E	T
0	s3			g1	g2
1			a		
2		s4	r2		
3		r3	r3		
4	s3			g5	g2
5			r1		

[0] S → E$

[1] E → T + E

[2] E → T

[3] T → *identifier*

Current input token = **+**

State on top of the stack = **2**

(2,T)

(0,S)

x + y$

Example

	action			goto	
state	*ident*	+	$	E	T
0	s3			g1	g2
1			a		
2		s4	r2		
3		r3	r3		
4	s3			g5	g2
5			r1		

0 S → E$

1 E → T + E

2 E → T

3 T → *identifier*

Current input token = **+**

State on top of the stack = **2**

(2,T)
(0,S)

x + y$

Example

state	action			goto	
	ident	+	$	E	T
0	s3			g1	g2
1			a		
2		s4	r2		
3		r3	r3		
4	s3			g5	g2
5			r1		

0 S → E$

1 E → T + E

2 E → T

3 T → *identifier*

Current input token = **y**

State on top of the stack = **4**

(4,+)

(2,T)

(0,S)

x + y$

Example

state	action			goto	
	ident	+	$	E	T
0	s3			**g1**	**g2**
1			**a**		
2		s4	r2		
3		r3	r3		
4	s3			**g5**	**g2**
5			r1		

0 S → E$

1 E → T + E

2 E → T

3 T → *identifier*

Current input token = **y**

State on top of the stack = **4**

(4,+)

(2,T)

(0,S)

x + y$

Example

	action			goto	
state	*ident*	+	$	E	T
0	s3			g1	g2
1			a		
2		s4	r2		
3		r3	r3		
4	s3			g5	g2
5			r1		

0 S → E$

1 E → T + E

2 E → T

3 T → *identifier*

Current input token = **$**

State on top of the stack = **3**

(3,y)
(4,+)
(2,T)
(0,S)

x + y$

Example

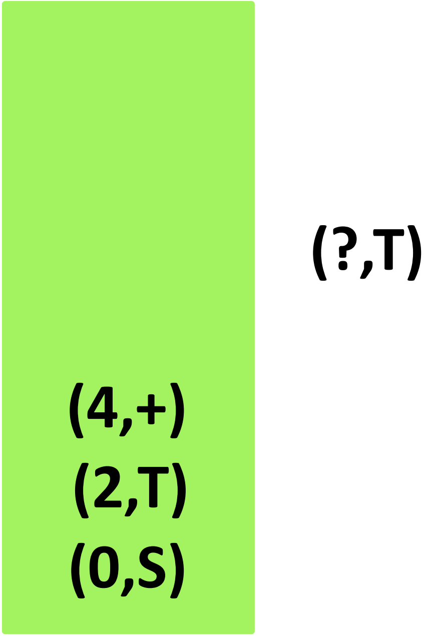
state	action			goto	
	ident	+	$	E	T
0	s3			g1	g2
1			a		
2		s4	r2		
3		r3	r3		
4	s3			g5	g2
5			r1		

0 S → E$

1 E → T + E

2 E → T

3 T → *identifier*


(?,T)

(4,+)

(2,T)

(0,S)

Current input token = $

State on top of the stack = 3

x + y$

Example

state	action			goto	
	ident	+	$	E	T
0	s3			g1	g2
1			a		
2		s4	r2		
3		r3	r3		
4	s3			g5	g2
5			r1		

0 S → E$

1 E → T + E

2 E → T

3 T → *identifier*

Current input token = $

State on top of the stack = 2

(2,T)
(4,+)
(2,T)
(0,S)

x + y$

Example

	action			goto	
state	*ident*	+	$	E	T
0	s3			g1	g2
1			a		
2		s4	r2		
3		r3	r3		
4	s3			g5	g2
5			r1		

0 S → E$

1 E → T + E

2 E → T

3 T → *identifier*

Current input token = **$**

State on top of the stack = **2**

(2,T)
(4,+)
(2,T)
(0,S)

x + y$

Example

	action			goto	
state	*ident*	+	$	E	T
0	s3			g1	g2
1			a		
2		s4	r2		
3		r3	r3		
4	s3			g5	g2
5			r1		

0 S → E$

1 E → T + E

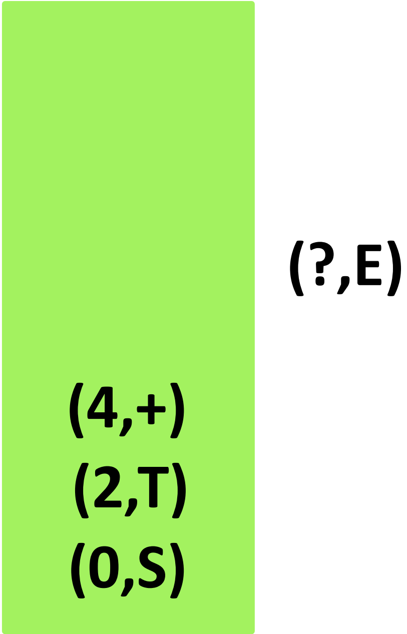
2 E → T

3 T → *identifier*

Current input token = **$**

State on top of the stack = **2**

x + y$



(?,E)

(4,+)

(2,T)

(0,S)

Example

	action			goto	
state	*ident*	+	$	E	T
0	s3			g1	g2
1			a		
2		s4	r2		
3		r3	r3		
4	s3			g5	g2
5			r1		

0 S → E$

1 E → T + E

2 E → T

3 T → *identifier*

Current input token = $

State on top of the stack = 5

(5,E)
(4,+)
(2,T)
(0,S)

x + y$

Example

state	action			goto	
	ident	+	$	E	T
0	s3			g1	g2
1			a		
2		s4	r2		
3		r3	r3		
4	s3			g5	g2
5			r1		

0 S → E$

1 E → T + E

2 E → T

3 T → *identifier*

Current input token = **$**

State on top of the stack = **5**

(5,E)
(4,+)
(2,T)
(0,S)

x + y$

Example

state	action			goto	
	ident	+	$	E	T
0	s3			g1	g2
1			a		
2		s4	r2		
3		r3	r3		
4	s3			g5	g2
5			r1		

0 S → E$

1 E → T + E

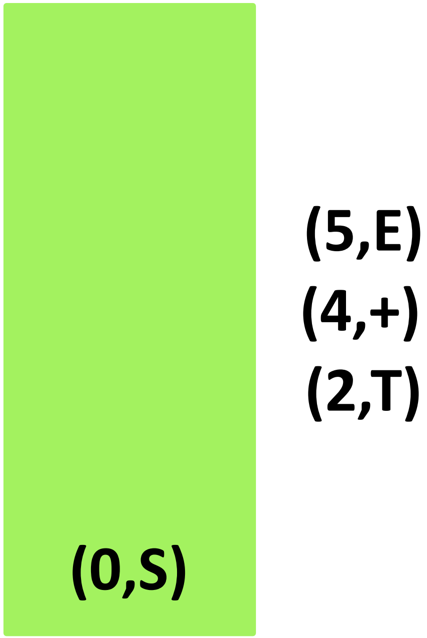
2 E → T

3 T → *identifier*

Current input token = **$**

State on top of the stack = **5**

x + y$



(5,E)

(4,+)

(2,T)

(0,S)

Example

state	action			goto	
	ident	+	$	E	T
0	s3			g1	g2
1			a		
2		s4	r2		
3		r3	r3		
4	s3			g5	g2
5			r1		

0 S → E$

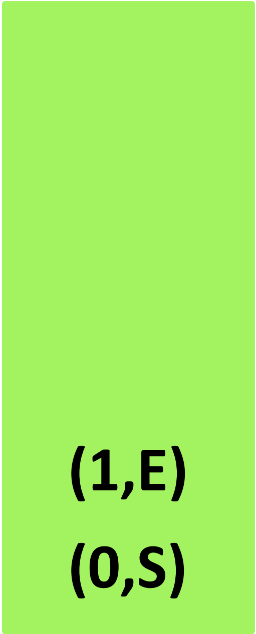
1 E → T + E

2 E → T

3 T → *identifier*

Current input token = **$**

State on top of the stack = **1**


(1,E)

(0,S)

x + y$

Example

	action			goto	
state	*ident*	+	$	E	T
0	s3			g1	g2
1			a		
2		s4	r2		
3		r3	r3		
4	s3			g5	g2
5			r1		

Accept!

0 S → E$

1 E → T + E

2 E → T

3 T → *identifier*

Current input token = **$**

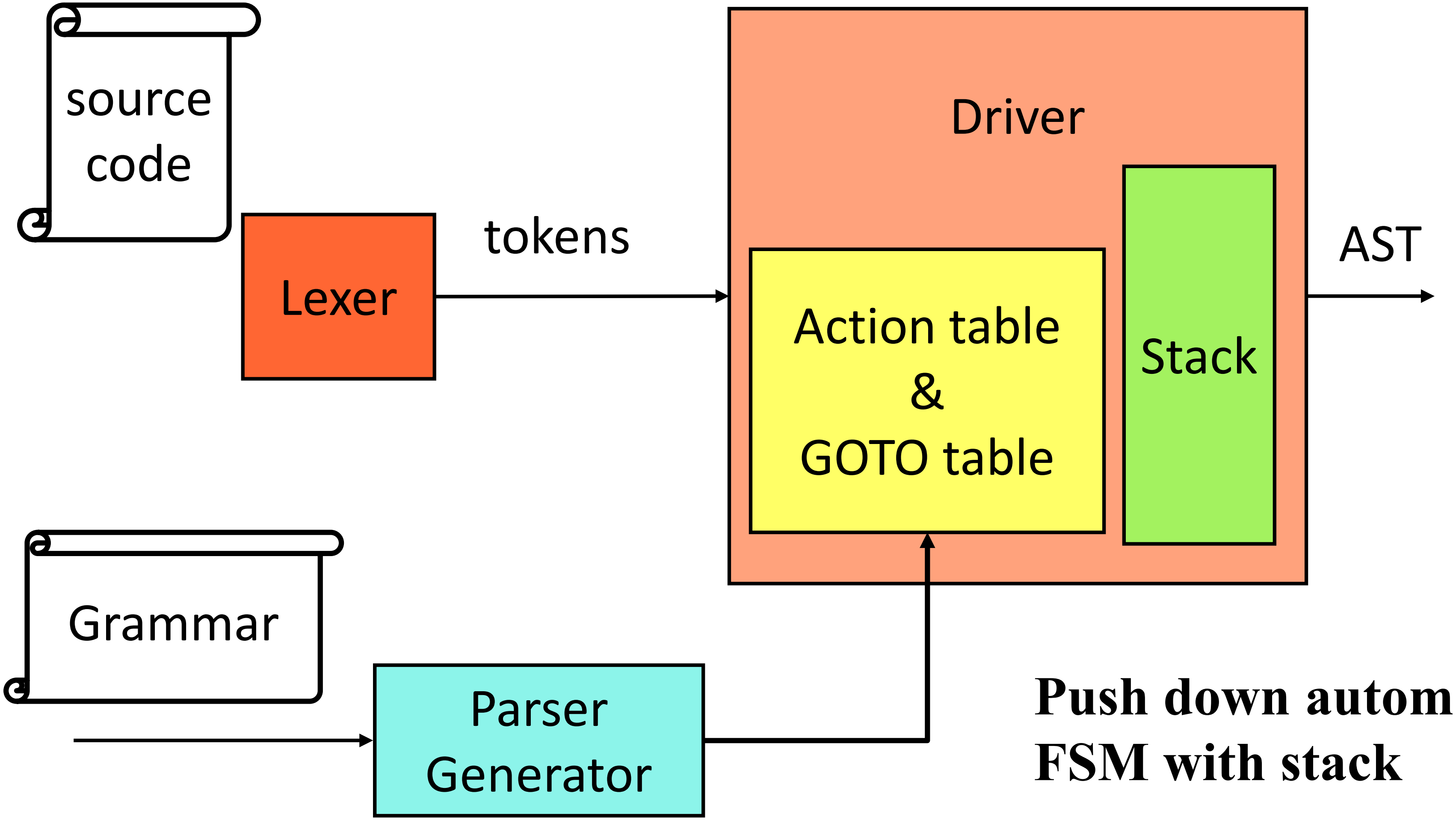
State on top of the stack = **1**

(1,E)

(0,S)

x + y$

Table-driven LR(k) parsers



Push down automata: FSM with stack

The parser generator

Parser Generator

- Finds handles
- Creates the action and GOTO tables.
- Creates the states
  - Each state indicates how much of a handle we have seen
  - each state is a set of *items*

Items

- Items are used to identify handles.
- LR(k) items have the form:

  [production-with-dot, lookahead]
- For example, A → a X b has 4 LR(0) items
  - [A → • a X b]
  - [A → a • X b]
  - [A → a X • b]
  - [A → a X b •]

The • indicates how much of the handle we have recognized.

What LR(0) Items Mean

- [X → • α β γ]
  input is consistent with X → α β γ
- [X → α • β γ]
  input is consistent with X → α β γ and we have already recognized α
- [X → α β • γ]
  input is consistent with X → α β γ and we have already recognized α β
- [X → α β γ •]
  input is consistent with X → α β γ and we can reduce to X

Generating the States

- Start with start production.
- In this case, “S → E$”

S → •E$

0 S → E$
1 E → T + E
2 E → T
3 T → *identifier*

- Each state is consistent with what we have already shifted from the input and what is possible to reduce. So, what other items should be in this state?

Completing a state

- For each item in a state, add in all other consistent items.

$S \rightarrow \bullet E\$$
$E \rightarrow \bullet T + E$
$E \rightarrow \bullet T$
$T \rightarrow \bullet \mathit{identifier}$

0 $S \rightarrow E\$$
1 $E \rightarrow T + E$
2 $E \rightarrow T$
3 $T \rightarrow \mathit{identifier}$

- This is called, taking the closure of the state.

Closure*

```
closure(state)
   repeat
      foreach item A → a•Xb in state
         foreach production X → w
            state.add(X → •w)
   until state does not change
   return state
```

Intuitively:

Given a set of items, add all production rules that could produce the nonterminal(s) at the current position in each item

*: for LR(0) items

What about the other states?

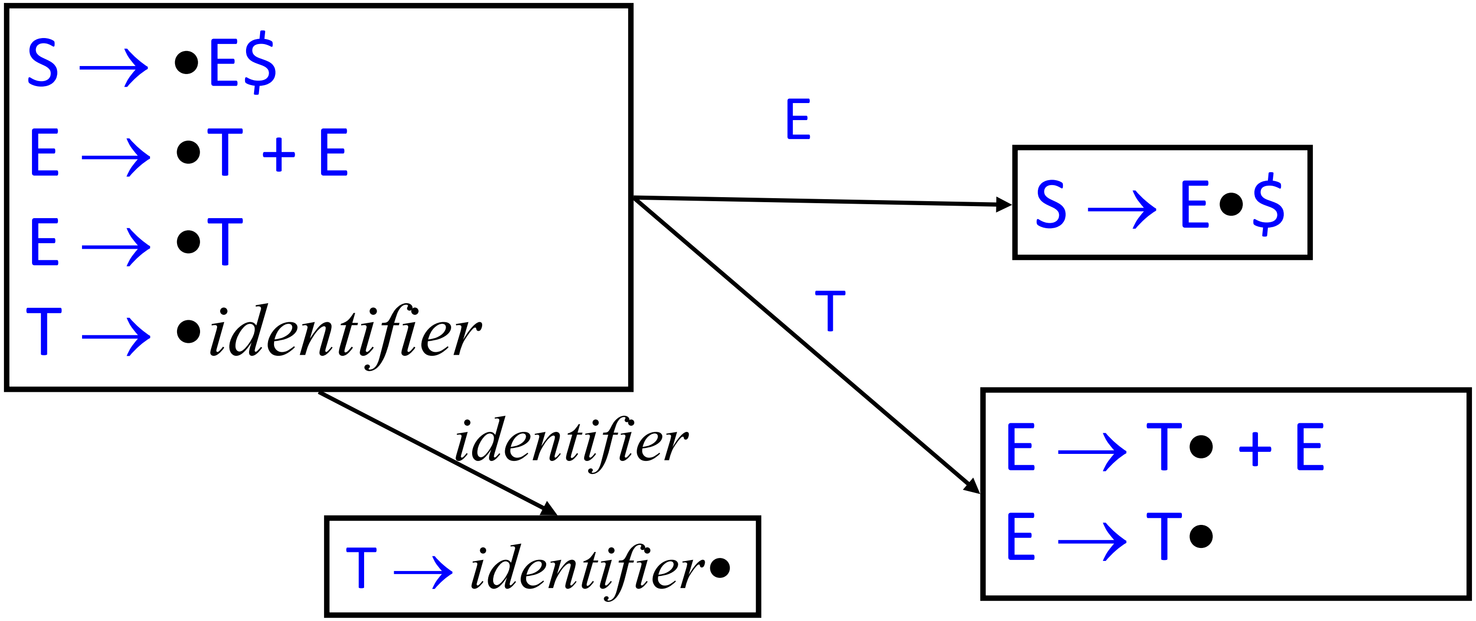
- How do we decide what the other states are?
- How do we decide what the transitions between states are?

0 S → E$

1 E → T + E

2 E → T

3 T → *identifier*



Next(state, sym)

- Next function determines what state to goto based on current state and symbol being recognized.
- For Non-terminal, this is used to determine the GOTO table.
- For terminal, this is used to determine the shift action.

Constructing states

```
initial_state = closure({start production})
state_set.add(initial_state)
state_queue.push(initial_state)

while(!state_queue.empty())
   s = state_queue.pop()
   foreach item $A \to a \bullet Xb$ in s
      n = closure(next(s, X))
      if(!state_set.contains(n))
         state_set.add(n)
         state_queue.push(n)
```

A state is a set of LR(0) items

get "next" state

Closure*

closure({S → •E$}) =

S → •E$

0 S → E$

1 E → T + E

2 E → T

3 T → *identifier*

*: for LR(0) items

Closure*

closure({$S \rightarrow \bullet E\$$}) =

$S \rightarrow \bullet E\$$

$E \rightarrow \bullet T + E$

$E \rightarrow \bullet T$

$T \rightarrow \bullet \mathit{identifier}$

0 $S \rightarrow E\$$

1 $E \rightarrow T + E$

2 $E \rightarrow T$

3 $T \rightarrow \mathit{identifier}$

*: for LR(0) items

Next

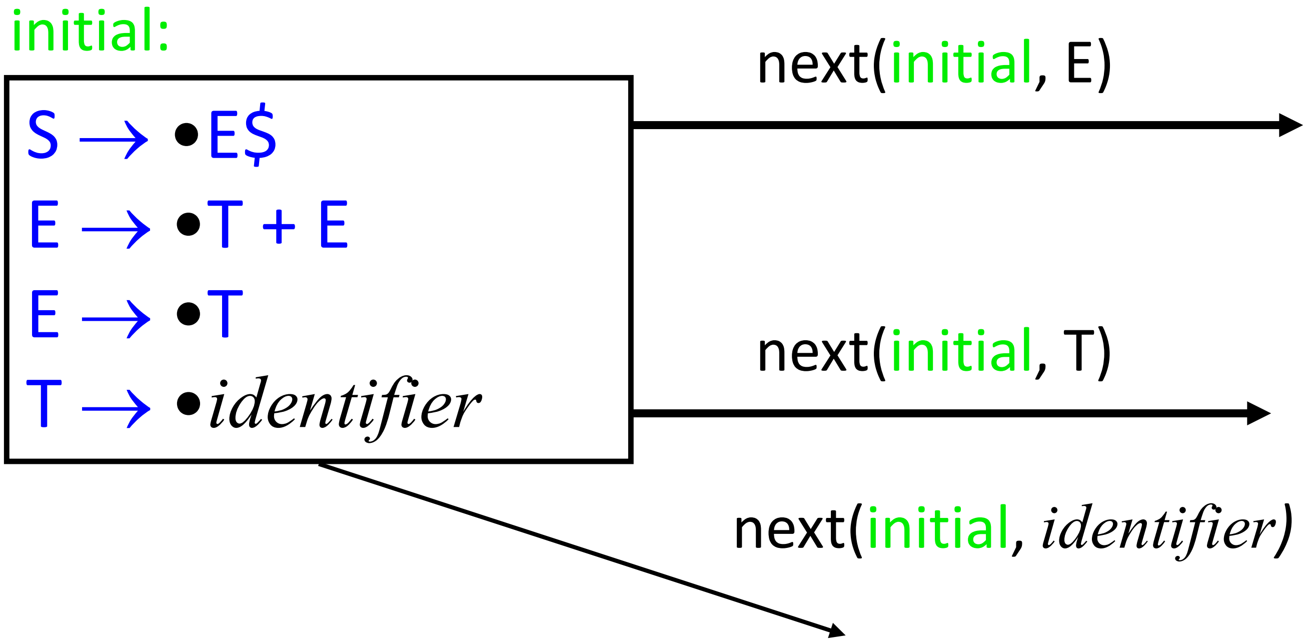
```
next(state, X)
    ret = empty
    foreach item A → a•Xb in state
        ret.add(A → aX•b)
    return ret
```

0 $S \rightarrow E\$$

1 $E \rightarrow T + E$

2 $E \rightarrow T$

3 $T \rightarrow \mathit{identifier}$



initial:

$S \rightarrow \bullet E\$$

$E \rightarrow \bullet T + E$

$E \rightarrow \bullet T$

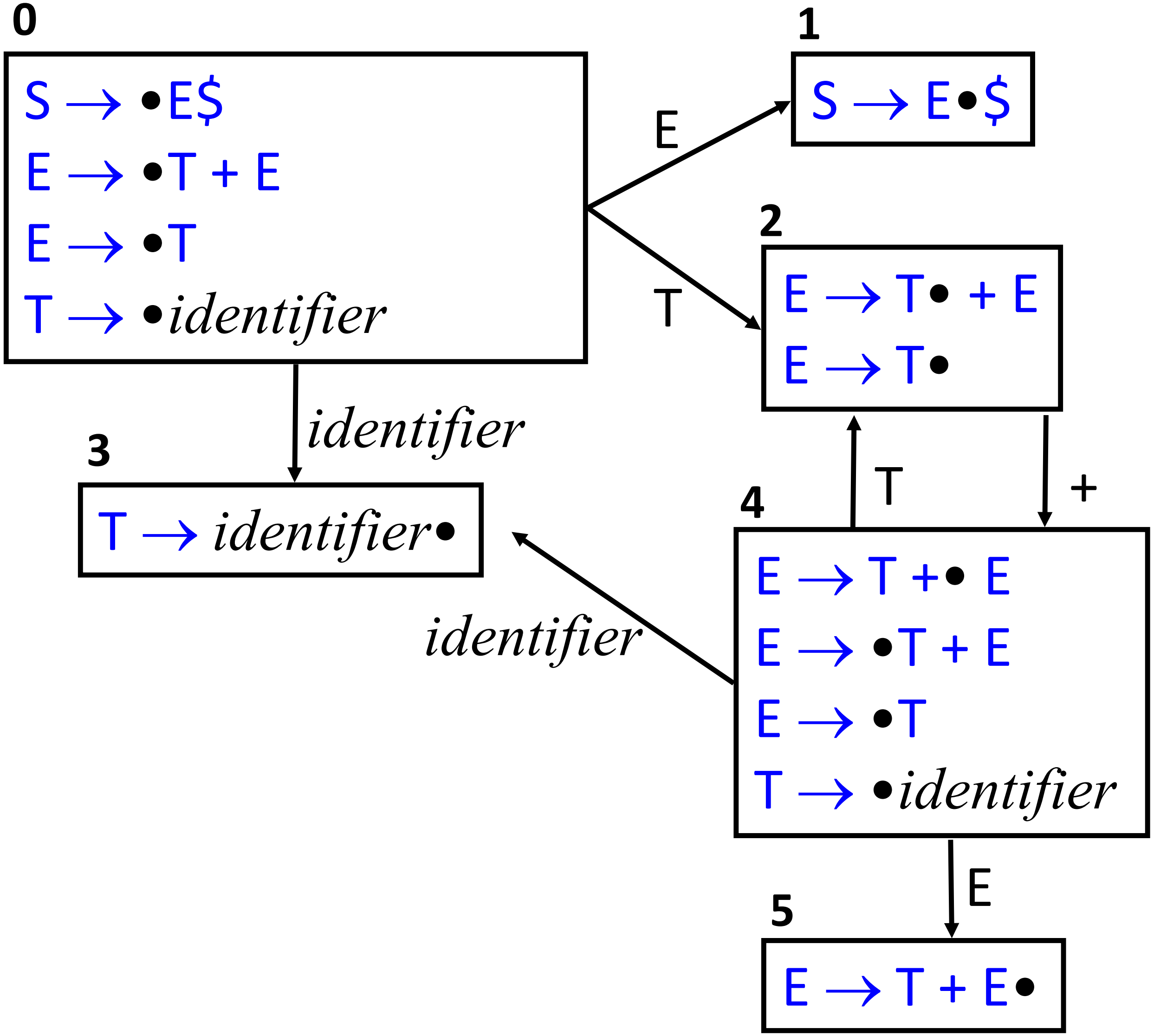
$T \rightarrow \bullet \mathit{identifier}$

next(initial, E)

next(initial, T)

next(initial, *identifier*)

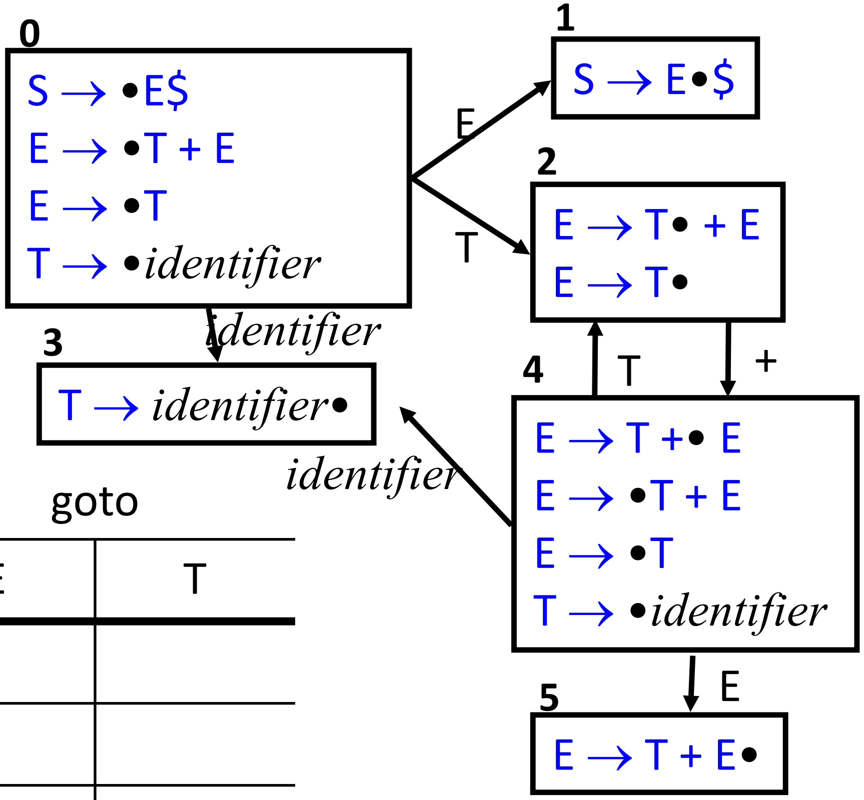
Example



0 S → E$
1 E → T + E
2 E → T
3 T → *identifier*

Parse Tables for LR(0) parser

What can we fill out?



	action			goto	
state	*ident*	+	$	E	T
0					
1					
2					
3					
4					
5					

0 S → E$

1 E → T + E

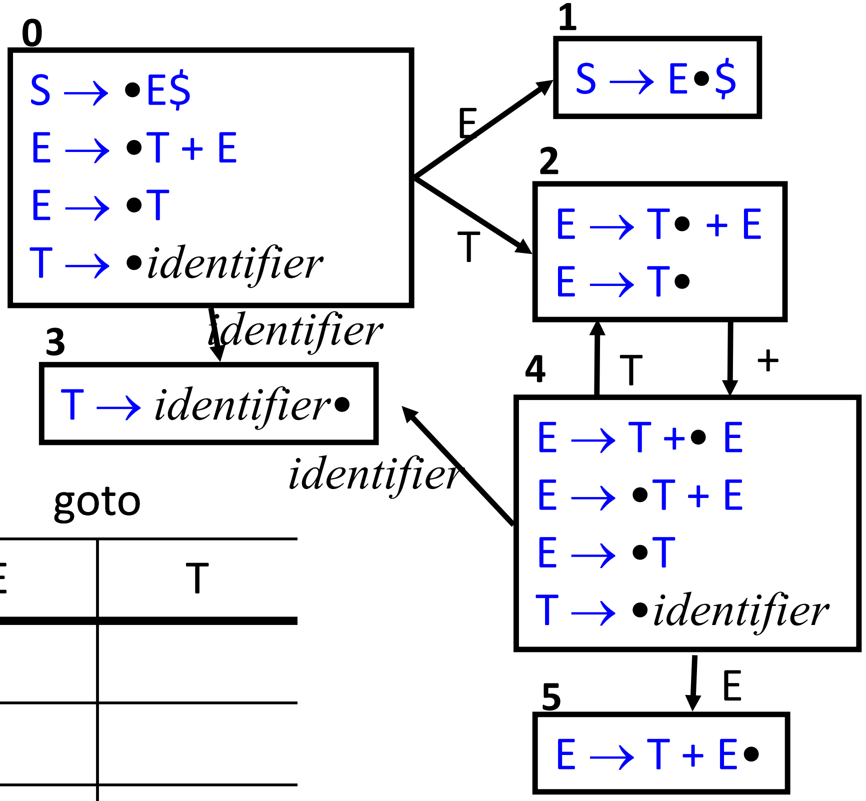
2 E → T

3 T → *identifier*

Parse Tables for LR(0) parser

shift

transition on terminal



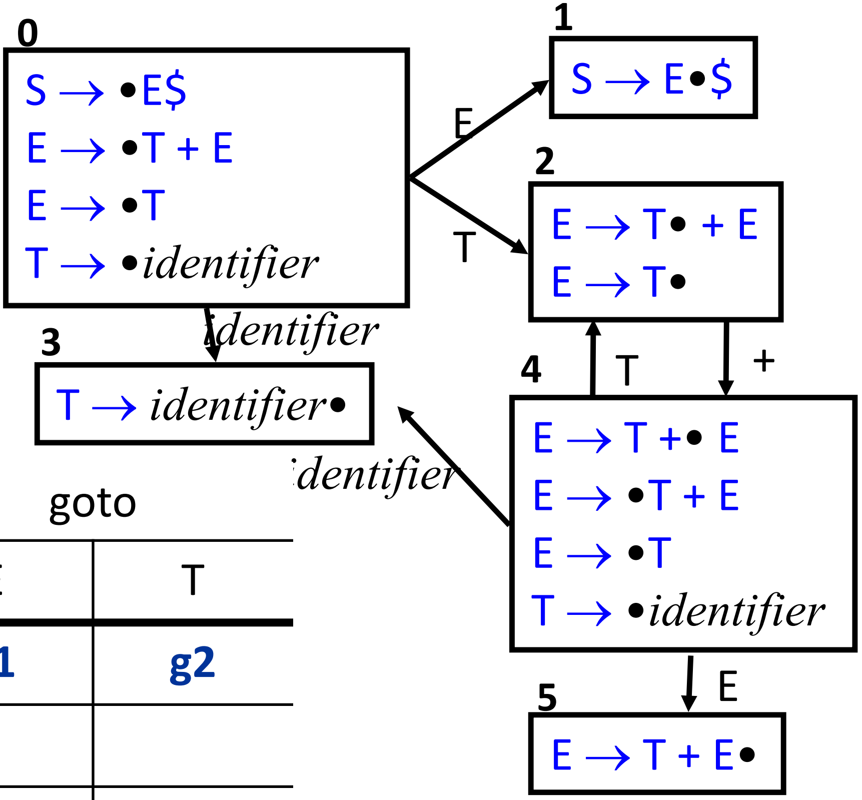
state	action			goto	
	ident	+	$	E	T
0	s3				
1					
2		s4			
3					
4	s3				
5					

0 S → E$
1 E → T + E
2 E → T
3 T → *identifier*

Parse Tables for LR(0) parser

goto

transition on nonterminal



	action			goto	
state	*ident*	+	$	E	T
0	s3			g1	g2
1					
2		s4			
3					
4	s3			g5	g2
5					

0 S → E$

1 E → T + E

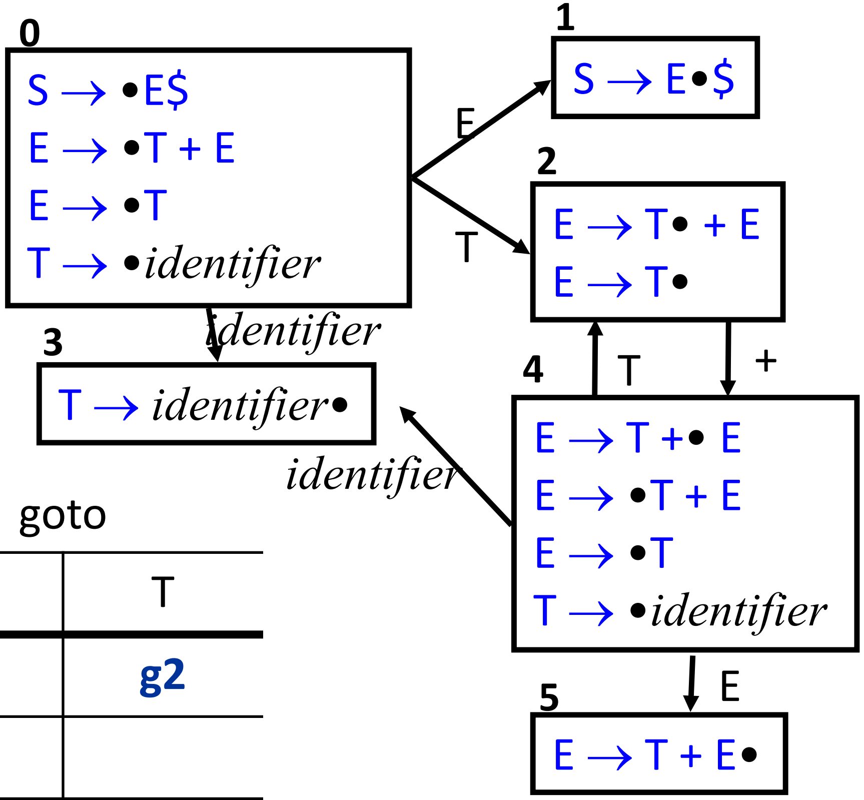
2 E → T

3 T → *identifier*

Parse Tables for LR(0) parser

accept

about to shift $



0
S → •E$
E → •T + E
E → •T
T → •*identifier*

1
S → E•$

2
E → T• + E
E → T•

3
T → *identifier*•

4
E → T +• E
E → •T + E
E → •T
T → •*identifier*

5
E → T + E•

	action			goto	
state	*ident*	+	$	E	T
0	s3			g1	g2
1			a		
2		s4			
3					
4	s3			g5	g2
5					

0 S → E$

1 E → T + E

2 E → T

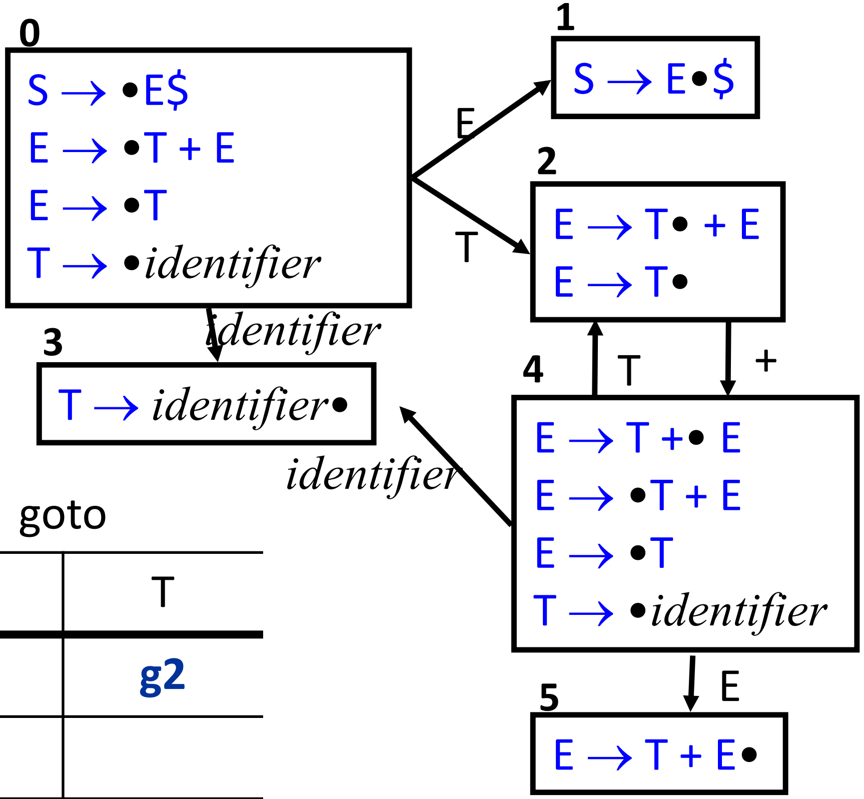
3 T → *identifier*

Parse Tables for LR(0) parser

reduce

item has dot at end

$A \rightarrow w\bullet$



	action			goto	
state	*ident*	+	$	E	T
0	s3			g1	g2
1			a		
2		s4			
3					
4	s3			g5	g2
5					

0 S → E$

1 E → T + E

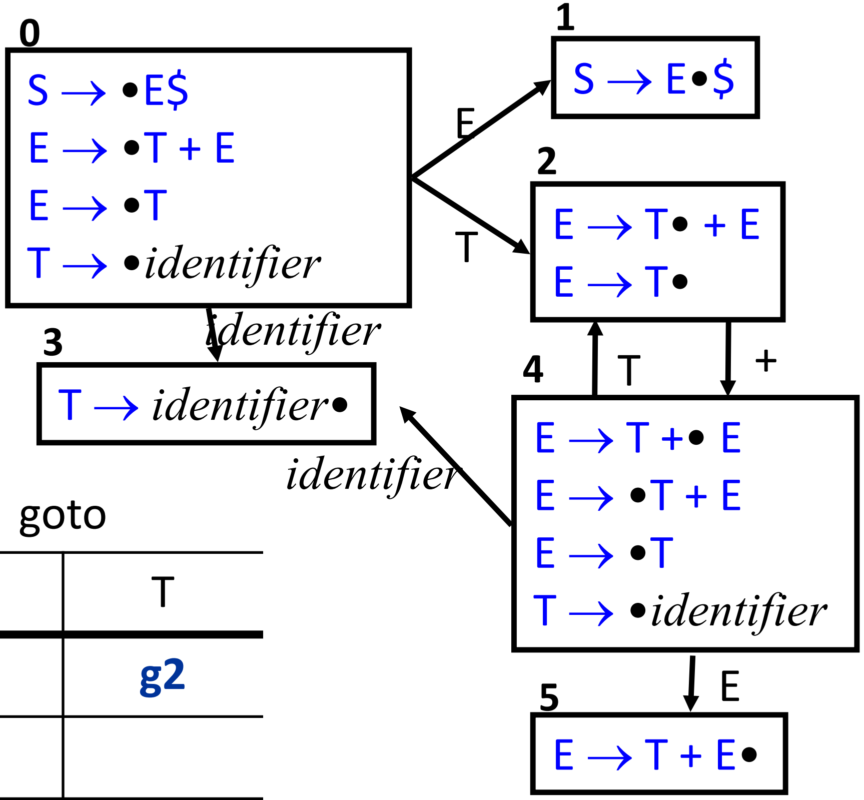
2 E → T

3 T → *identifier*

LR(0)

No lookahead

reduce state for *all* nonterminals



0
S → •E$
E → •T + E
E → •T
T → •*identifier*

1
S → E•$

2
E → T• + E
E → T•

3
T → *identifier*•

4
E → T +• E
E → •T + E
E → •T
T → •*identifier*

5
E → T + E•

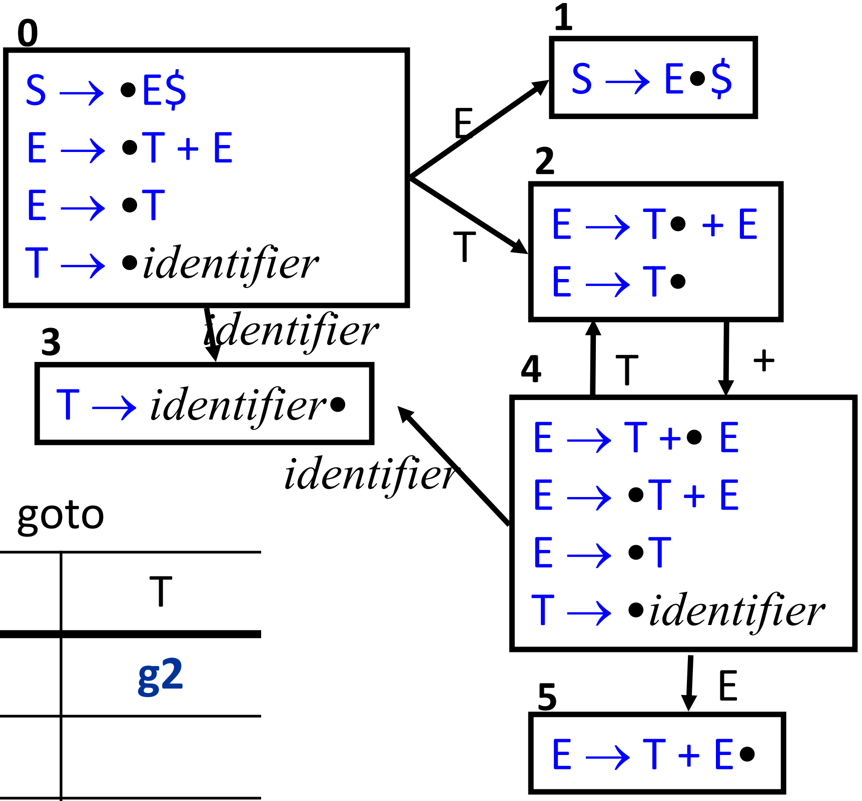
state	action			goto	
	ident	+	$	E	T
0	s3			g1	g2
1			a		
2	r2	r2/s4	r2		
3	r3	r3	r3		
4	s3			g5	g2
5	r1	r1	r1		

0 S → E$
1 E → T + E
2 E → T
3 T → *identifier*

LR(0)

shift/reduce conflict

need to be pickier about when we reduce

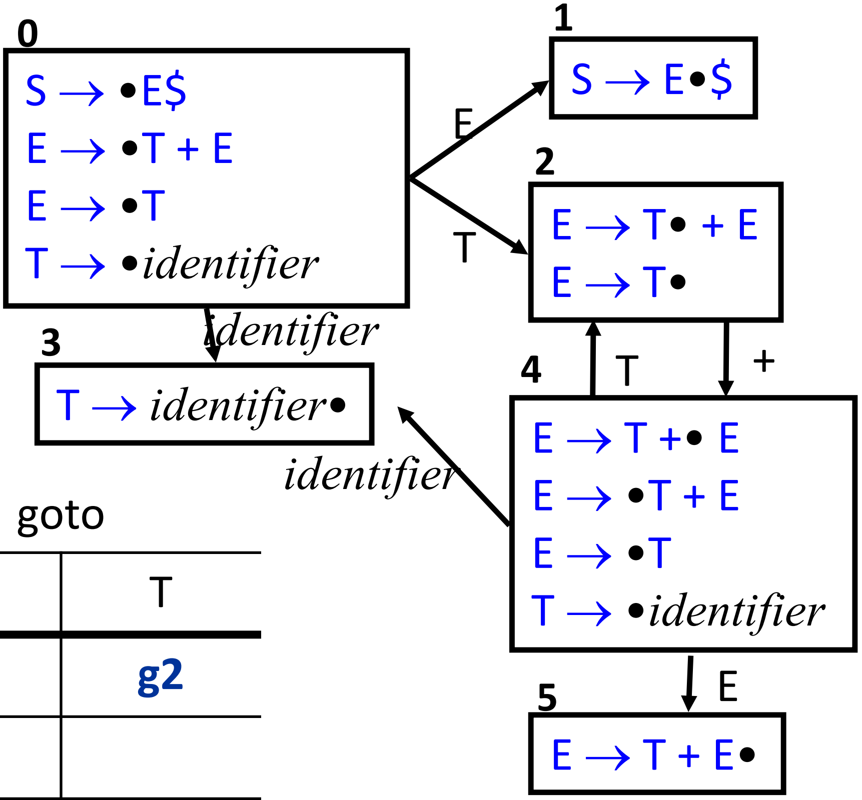


	action			goto	
state	*ident*	+	$	E	T
0	s3			g1	g2
1			a		
2	r2	r2/s4	r2		
3	r3	r3	r3		
4	s3			g5	g2
5	r1	r1	r1		

0 S → E$
1 E → T + E
2 E → T
3 T → *identifier*

SLR - Simple LR

Only reduce in position (**s**,**a**) by rule R:A → *w* if **a** is in the *follow set* of A



	action			goto	
state	*ident*	+	$	E	T
0	s3			g1	g2
1			**a**		
2		s4			
3					
4	s3			g5	g2
5					

0 S → E$
1 E → T + E
2 E → T
3 T → *identifier*

Reminder: Follow sets

follow(X)

set of terminals that can appear immediately after the nonterminal X in some sentential form

0 S → E$

1 E → T + E

2 E → T

3 T → *identifier*

I.e., t $\in$ FOLLOW(X) iff S $\Rightarrow^*$ αXtβ for some α and β

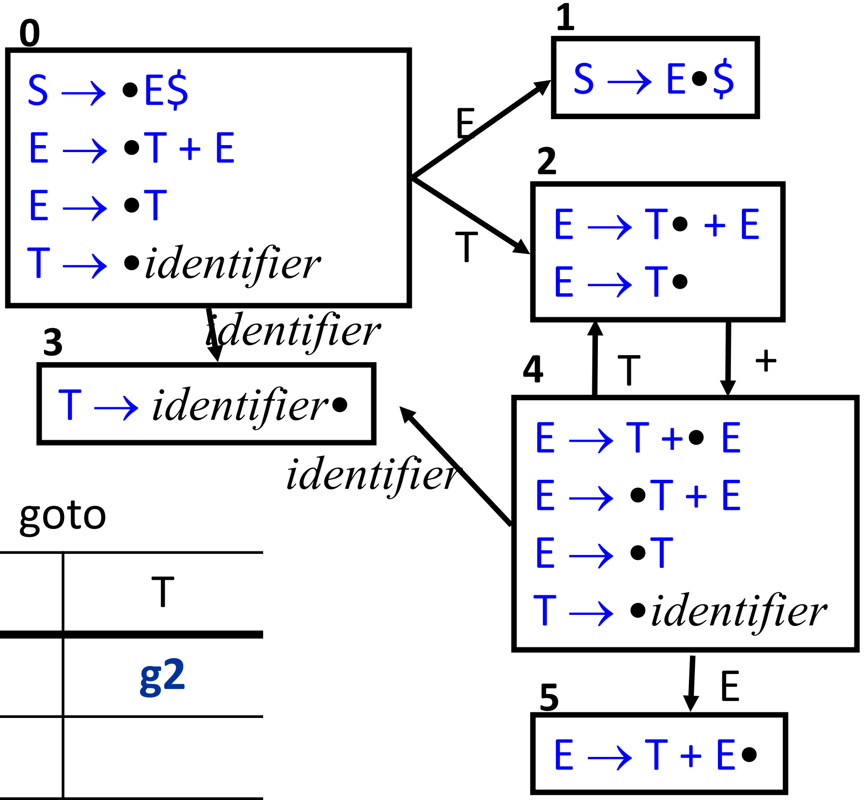
follow(E**) = {$}**

follow(T**) = {+,$}**

SLR - Reduce using follow sets

follow(E) = {$}

follow(T) = {+,$}



0

S → •E$

E → •T + E

E → •T

T → •*identifier*

1

S → E•$

2

E → T• + E

E → T•

3

T → *identifier*•

4

E → T +• E

E → •T + E

E → •T

T → •*identifier*

5

E → T + E•

state	action			goto	
	ident	+	$	E	T
0	s3			g1	g2
1			a		
2		s4	r2		
3		r3	r3		
4	s3			g5	g2
5			r1		

0 S → E$

1 E → T + E

2 E → T

3 T → *identifier*

SLR Limitations

- SLR uses LR(0) item sets
- Can remove some (but not all) shift/reduce conflicts using follow set
- Consider

0 S → E$

1 E → L = R

2 E → R

3 L → *id*

4 L → *R

5 R → L

Example

0 S → E$

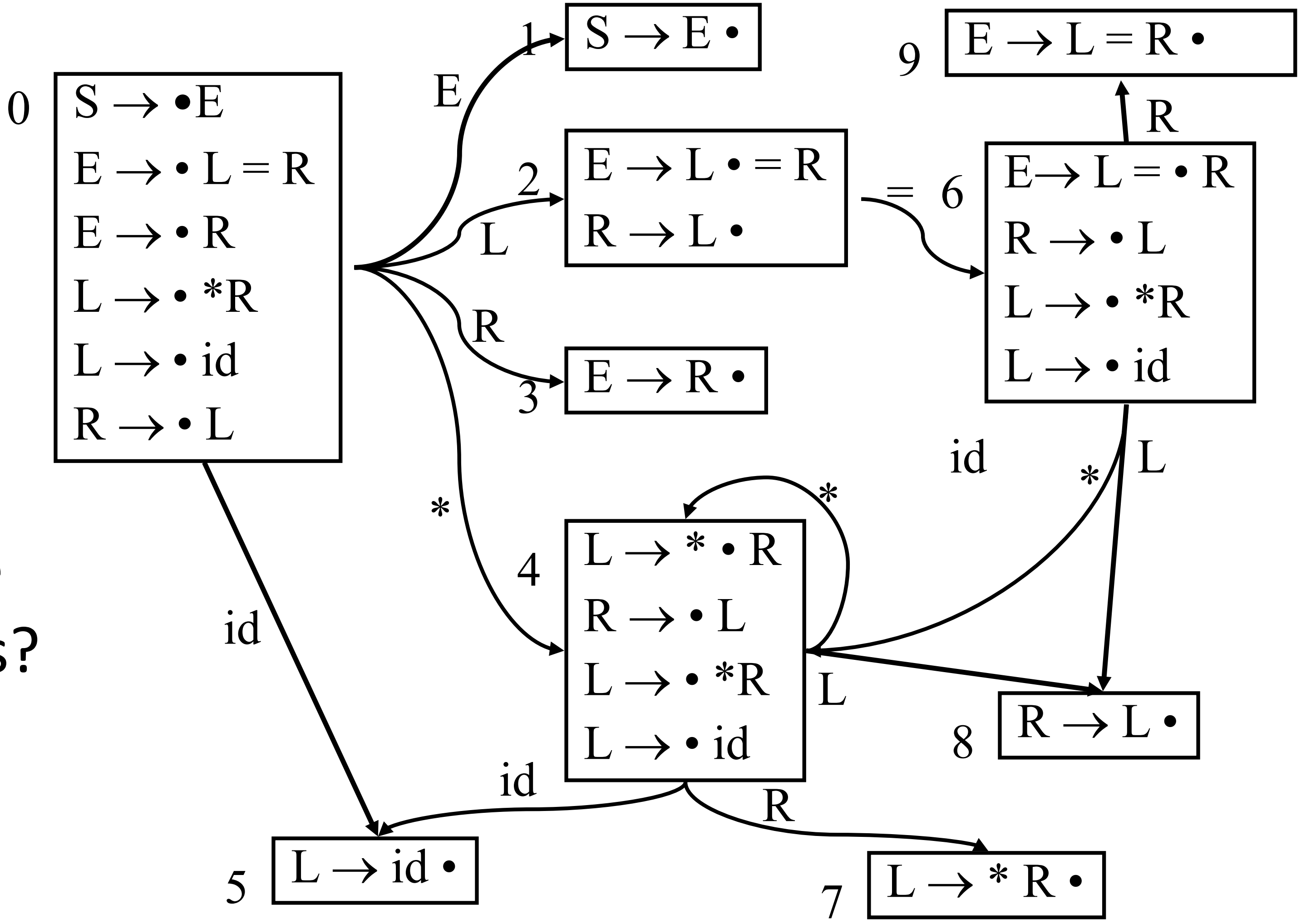
1 E → L = R

2 E → R

3 L → *id*

4 L → *R

5 R → L



0
S → •E
E → • L = R
E → • R
L → • *R
L → • id
R → • L

1
S → E •

2
E → L • = R
R → L •

3
E → R •

4
L → * • R
R → • L
L → • *R
L → • id

5
L → id •

6
E→ L = • R
R → • L
L → • *R
L → • id

7
L → * R •

8
R → L •

9
E → L = R •

What are the reduce states?

Example

0 S → E$

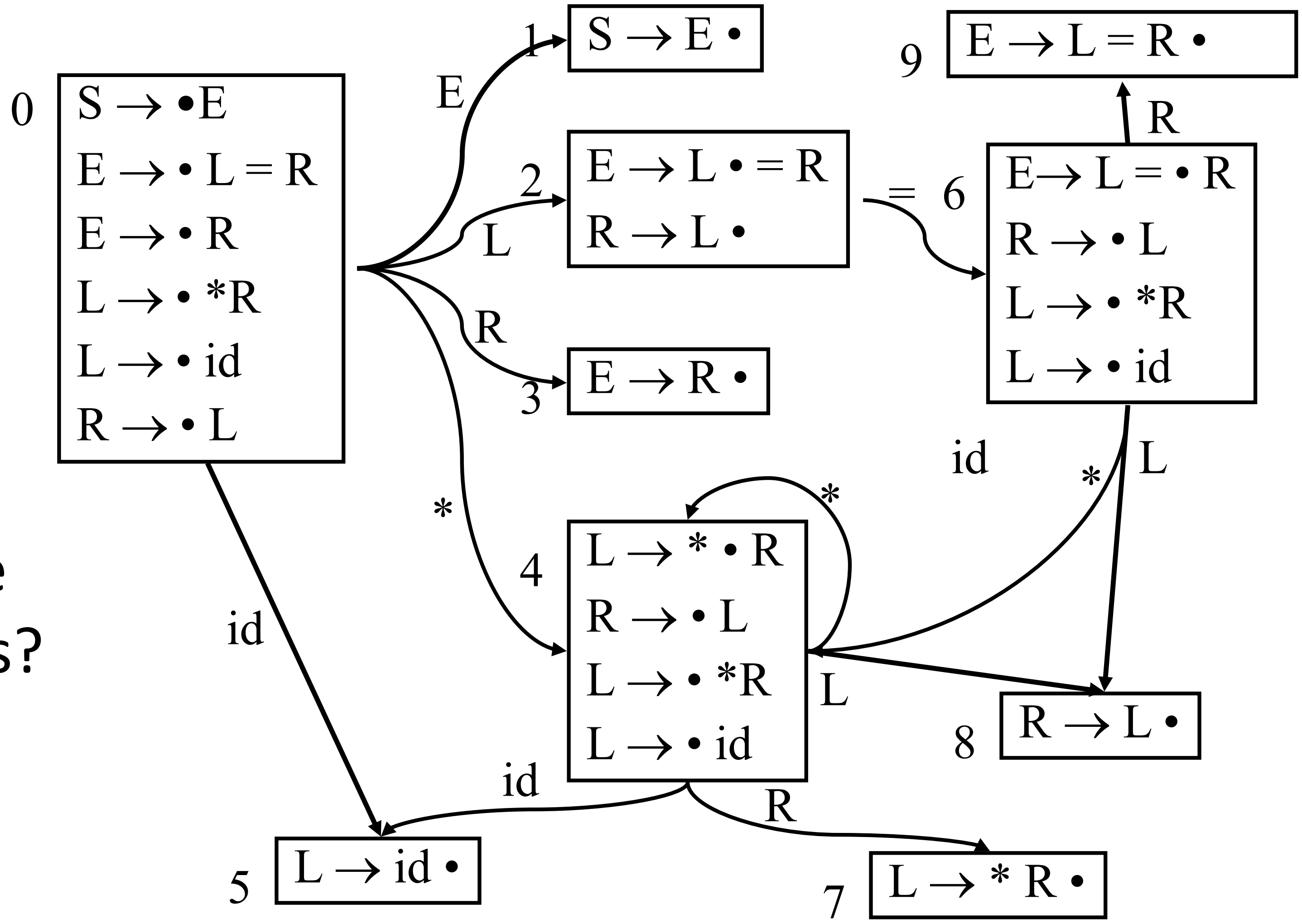
1 E → L = R

2 E → R

3 L → *id*

4 L → *R

5 R → L



0: S → •E, E → • L = R, E → • R, L → • *R, L → • id, R → • L

1: S → E •

2: E → L • = R, R → L •

3: E → R •

4: L → * • R, R → • L, L → • *R, L → • id

5: L → id •

6: E→ L = • R, R → • L, L → • *R, L → • id

7: L → * R •

8: R → L •

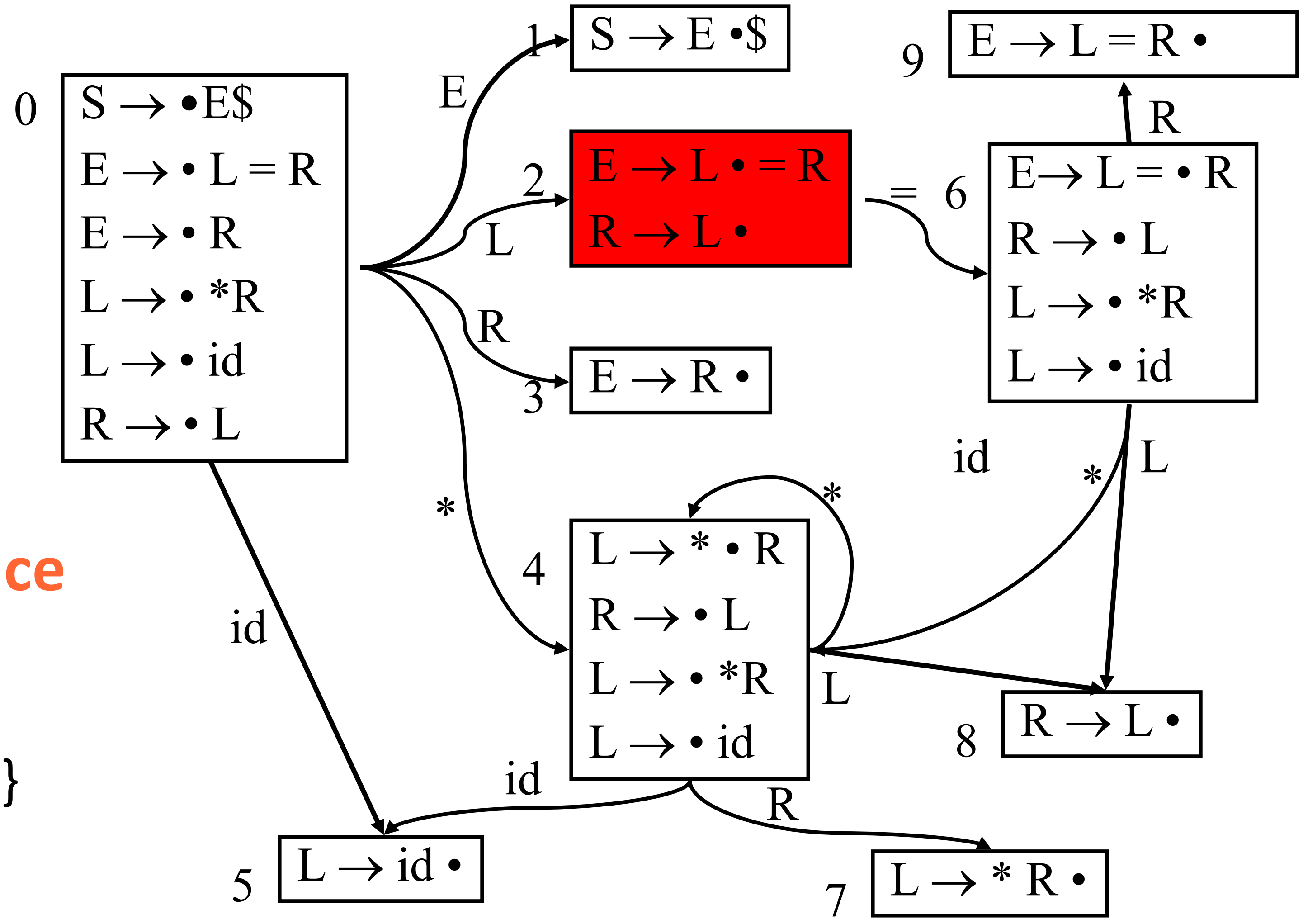
9: E → L = R •

What are the reduce states?

1,2,3,5,7,8,9

Example

0 S → E$
1 E → L = R
2 E → R
3 L → *id*
4 L → *R
5 R → L



0
S → •E$
E → • L = R
E → • R
L → • *R
L → • id
R → • L

1
S → E •$

2
E → L • = R
R → L •

3
E → R •

4
L → * • R
R → • L
L → • *R
L → • id

5
L → id •

6
E→ L = • R
R → • L
L → • *R
L → • id

7
L → * R •

8
R → L •

9
E → L = R •

shift/reduce conflict

follow(R) = {=,$}

Problem with SLR

- Reduce on ALL terminals in FOLLOW set

$$\begin{array}{lcl} S & \rightarrow & L = R \\ & | & R \\ L & \rightarrow & * \, R \\ & | & \mathbf{id} \\ R & \rightarrow & L \end{array}$$

2

$$\boxed{\begin{array}{l} S \rightarrow L \bullet = R \\ R \rightarrow L \bullet \end{array}}$$

- FOLLOW(R) = FOLLOW(L)
- But, we should never reduce R → L on ‘=‘
  I.e., R=… is not a viable prefix for a right sentential form
- Thus, there should be no reduction in state 2
- How can we solve this?

LR(1) Items

- An LR(1) item is an LR(0) item combined with a single terminal (the *lookahead*)
- $[X \rightarrow \alpha \bullet \beta, a]$ Means
  - α is at top of stack
  - Input string is derivable from βa
- In other words, when we reduce $X \rightarrow \alpha\beta$, a had better be the look ahead symbol.
- Or, Only put ‘reduce by $X \rightarrow \alpha\beta$’ in **action[**s,a**]**
- Can construct states as before, but have to modify closure

What LR(1) Items Mean

- $[X \rightarrow \bullet\ \alpha\ \beta\ \gamma, a]$
  input is consistent with $X \rightarrow \alpha\ \beta\ \gamma$
- $[X \rightarrow \alpha\ \bullet\ \beta\ \gamma, a]$
  input is consistent with $X \rightarrow \alpha\ \beta\ \gamma$ and we have already recognized α
- $[X \rightarrow \alpha\ \beta\ \bullet\ \gamma, a]$
  input is consistent with $X \rightarrow \alpha\ \beta\ \gamma$ and we have already recognized $\alpha\ \beta$
- $[X \rightarrow \alpha\ \beta\ \gamma\ \bullet, a]$
  input is consistent with $X \rightarrow \alpha\ \beta\ \gamma$ and if lookahead symbol is a, then we can reduce to X

LR(1) Closure

```
closure(state)
    repeat
        foreach item A → a•Xb, t in state
            foreach production X → w
              and each terminal t′ in FIRST(bt)
                state.add(X → •w, t′)
    until state does not change
    return state
```

Closure

closure({$S \rightarrow \bullet E\$$, ?}) =

$S \rightarrow \bullet E\$$, ?

[0] $S \rightarrow E\$$

[1] $E \rightarrow L = R$

[2] $E \rightarrow R$

[3] $L \rightarrow id$

[4] $L \rightarrow *R$

[5] $R \rightarrow L$

Closure

closure({S → •E$, ?}) =

S → •E$,	?
E → •L = R,	$
E → •R,	$

0 S → E$
1 E → L = R
2 E → R
3 L → *id*
4 L → *R
5 R → L

Closure

closure({$S \rightarrow \bullet E\$$, ?}) =

$S \rightarrow \bullet E\$$,	?
$E \rightarrow \bullet L = R$,	\$
$E \rightarrow \bullet R$,	\$
$L \rightarrow \bullet id$,	=
$L \rightarrow \bullet {*}R$,	=

0 $S \rightarrow E\$$

1 $E \rightarrow L = R$

2 $E \rightarrow R$

3 $L \rightarrow id$

4 $L \rightarrow {*}R$

5 $R \rightarrow L$

Closure

closure({$S \rightarrow \bullet E\$$, ?}) =

$S \rightarrow \bullet E\$,$	?
$E \rightarrow \bullet L = R,$	\$
$E \rightarrow \bullet R,$	\$
$L \rightarrow \bullet \mathit{id},$	=
$L \rightarrow \bullet {*}R,$	=
$R \rightarrow \bullet L,$	\$

0 $S \rightarrow E\$$
1 $E \rightarrow L = R$
2 $E \rightarrow R$
3 $L \rightarrow \mathit{id}$
4 $L \rightarrow {*}R$
5 $R \rightarrow L$

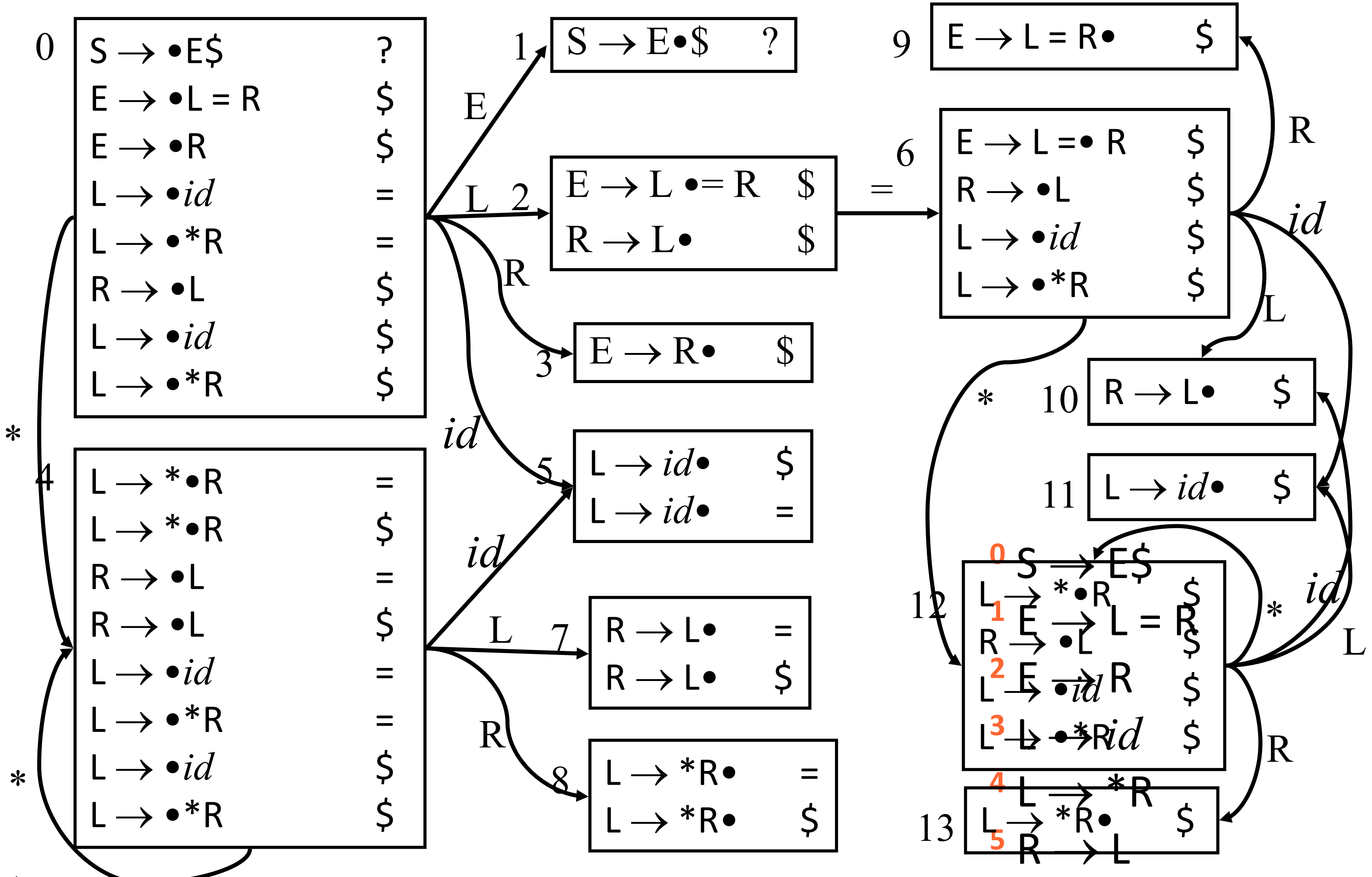
Closure

closure({$S \rightarrow \bullet E\$$, ?}) =

$S \rightarrow \bullet E\$$,	?
$E \rightarrow \bullet L = R$,	\$
$E \rightarrow \bullet R$,	\$
$L \rightarrow \bullet id$,	=
$L \rightarrow \bullet {*}R$,	=
$R \rightarrow \bullet L$,	\$
$L \rightarrow \bullet id$,	\$
$L \rightarrow \bullet {*}R$,	\$

0 $S \rightarrow E\$$
1 $E \rightarrow L = R$
2 $E \rightarrow R$
3 $L \rightarrow id$
4 $L \rightarrow {*}R$
5 $R \rightarrow L$

LR(1) Example



0
S → •E$?
E → •L = R $
E → •R $
L → •*id* =
L → •*R =
R → •L $
L → •*id* $
L → •*R $

1
S → E•$?

2
E → L •= R $
R → L• $

3
E → R• $

4
L → *•R =
L → *•R $
R → •L =
R → •L $
L → •*id* =
L → •*R =
L → •*id* $
L → •*R $

5
L → *id*• $
L → *id*• =

6
E → L =• R $
R → •L $
L → •*id* $
L → •*R $

7
R → L• =
R → L• $

8
L → *R• =
L → *R• $

9
E → L = R• $

10
R → L• $

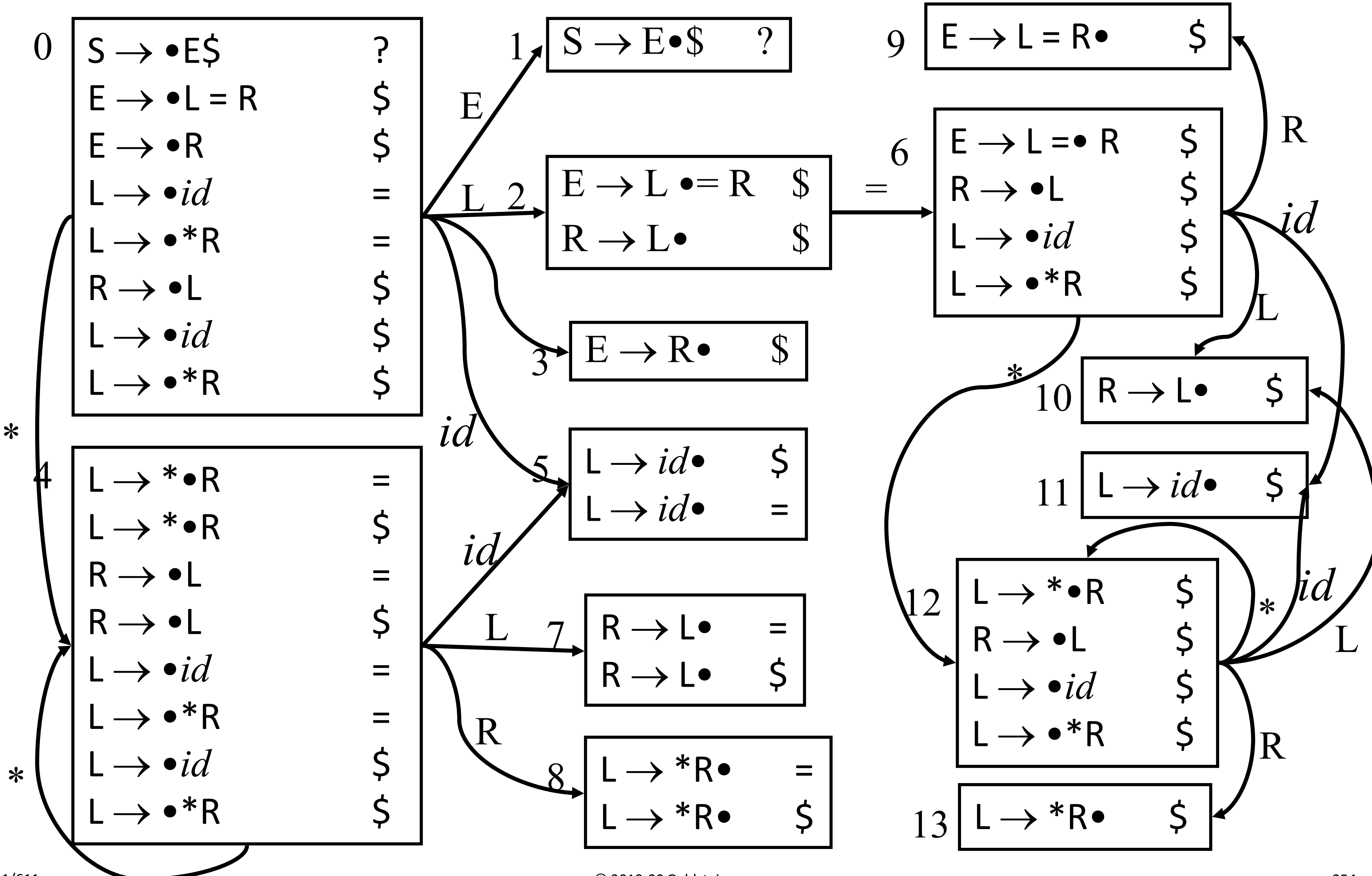
11
L → *id*• $

12
L → *•R $
R → •L $
L → •*id* $
L → •*R $

13
L → *R• $

0 S → E$
1 E → L = R
2 E → R
3 L → id
4 L → *R
5 R → L

LR(1) Example



0
S → •E$?
E → •L = R $
E → •R $
L → •*id* =
L → •*R =
R → •L $
L → •*id* $
L → •*R $

1
S → E•$?

2
E → L •= R $
R → L• $

3
E → R• $

4
L → *•R =
L → *•R $
R → •L =
R → •L $
L → •*id* =
L → •*R =
L → •*id* $
L → •*R $

5
L → *id*• $
L → *id*• =

6
E → L =• R $
R → •L $
L → •*id* $
L → •*R $

7
R → L• =
R → L• $

8
L → *R• =
L → *R• $

9
E → L = R• $

10
R → L• $

11
L → *id*• $

12
L → *•R $
R → •L $
L → •*id* $
L → •*R $

13
L → *R• $

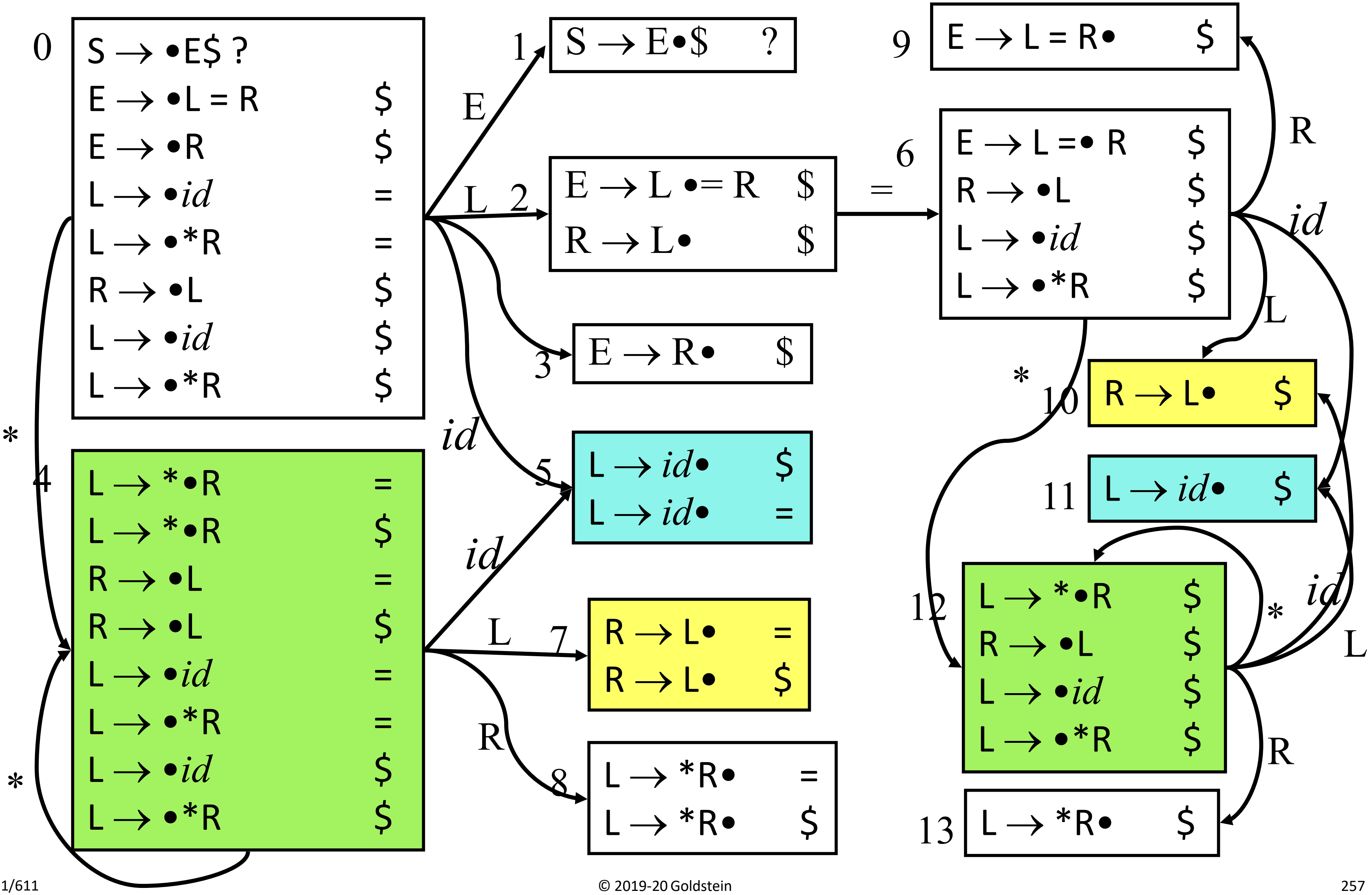
Parsing Table

- 14 states versus 10 LR(0) states
- In general, the number of states (and therefore size of the parsing table) is much larger with LR(1) items

LALR: Lookahead LR

- More powerful than SLR
- Given LR(1) states, merge states that are identical except for lookaheads
- End up with same size table as SLR
- Can this introduce conflicts?

Merge-able states



0
S → •E$?
E → •L = R $
E → •R $
L → •*id* =
L → •*R =
R → •L $
L → •*id* $
L → •*R $

1
S → E•$?

2
E → L •= R $
R → L• $

3
E → R• $

4
L → *•R =
L → *•R $
R → •L =
R → •L $
L → •*id* =
L → •*R =
L → •*id* $
L → •*R $

5
L → *id*• $
L → *id*• =

6
E → L =• R $
R → •L $
L → •*id* $
L → •*R $

7
R → L• =
R → L• $

8
L → *R• =
L → *R• $

9
E → L = R• $

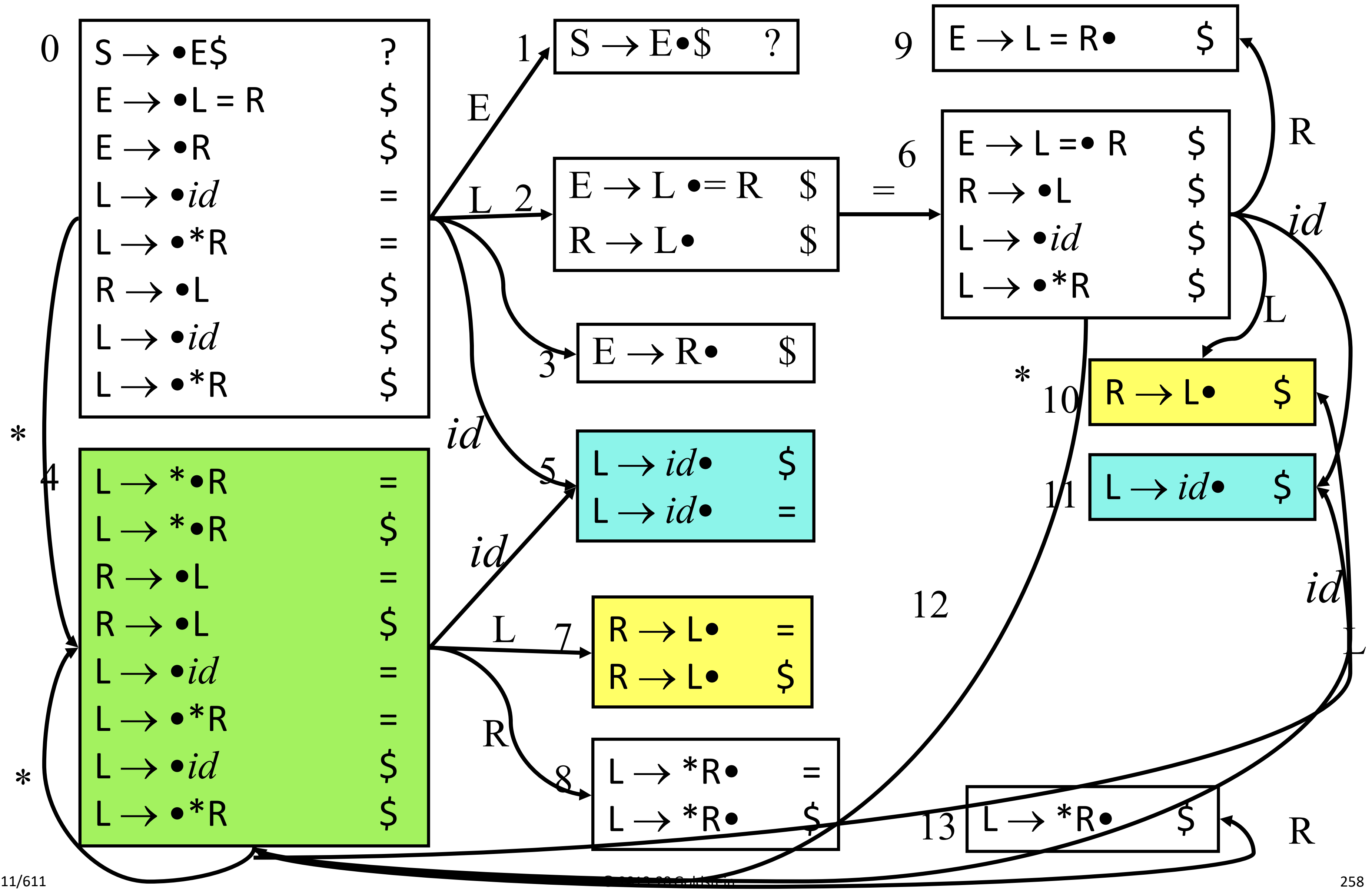
10
R → L• $

11
L → *id*• $

12
L → *•R $
R → •L $
L → •*id* $
L → •*R $

13
L → *R• $

Merge-able states



0
S → •E$?
E → •L = R $
E → •R $
L → •*id* =
L → •*R =
R → •L $
L → •*id* $
L → •*R $

1
S → E•$?

2
E → L •= R $
R → L• $

3
E → R• $

4
L → *•R =
L → *•R $
R → •L =
R → •L $
L → •*id* =
L → •*R =
L → •*id* $
L → •*R $

5
L → *id*• $
L → *id*• =

6
E → L =• R $
R → •L $
L → •*id* $
L → •*R $

7
R → L• =
R → L• $

8
L → *R• =
L → *R• $

9
E → L = R• $

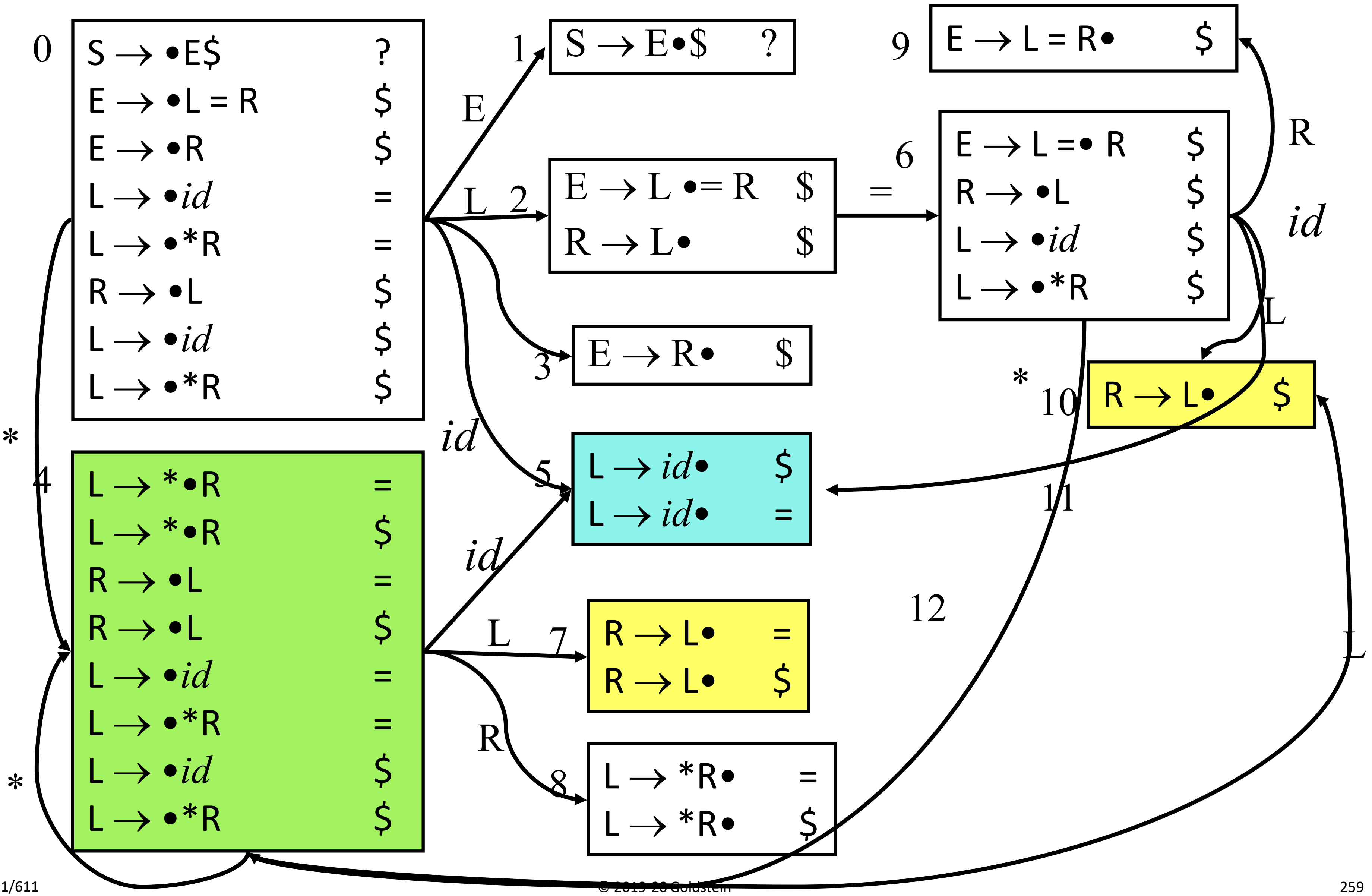
10
R → L• $

11
L → *id*• $

12

13
L → *R• $

Merge-able states



0
S → •E$?
E → •L = R $
E → •R $
L → •*id* =
L → •*R =
R → •L $
L → •*id* $
L → •*R $

1
S → E•$?

2
E → L •= R $
R → L• $

3
E → R• $

4
L → *•R =
L → *•R $
R → •L =
R → •L $
L → •*id* =
L → •*R =
L → •*id* $
L → •*R $

5
L → *id*• $
L → *id*• =

6
E → L =• R $
R → •L $
L → •*id* $
L → •*R $

7
R → L• =
R → L• $

8
L → *R• =
L → *R• $

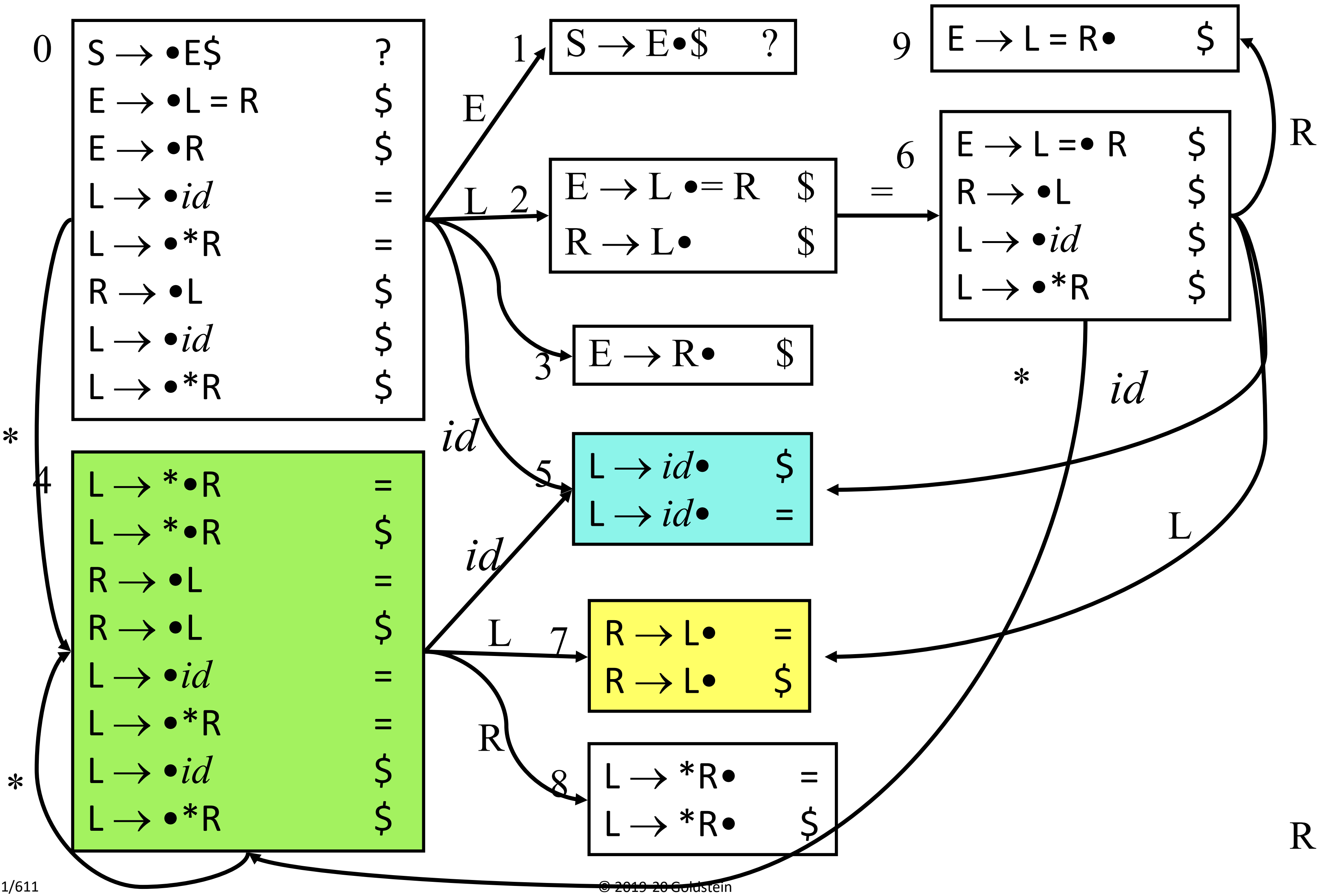
9
E → L = R• $

10
R → L• $

11

12

Merge-able states



0

S → •E$	?
E → •L = R	$
E → •R	$
L → •*id*	=
L → •*R	=
R → •L	$
L → •*id*	$
L → •*R	$

1

S → E•$	?

2

E → L •= R	$
R → L•	$

3

E → R•	$

4

L → *•R	=
L → *•R	$
R → •L	=
R → •L	$
L → •*id*	=
L → •*R	=
L → •*id*	$
L → •*R	$

5

L → *id*•	$
L → *id*•	=

6

E → L =• R	$
R → •L	$
L → •*id*	$
L → •*R	$

7

R → L•	=
R → L•	$

8

L → *R•	=
L → *R•	$

9

E → L = R•	$

LALR

- Can generate parse table without constructing LR(1) item sets
  - construct LR(0) item sets
  - compute *lookahead* sets
    - more precise than follow sets
- LALR is used by most parser generators (e.g., bison)

Recap

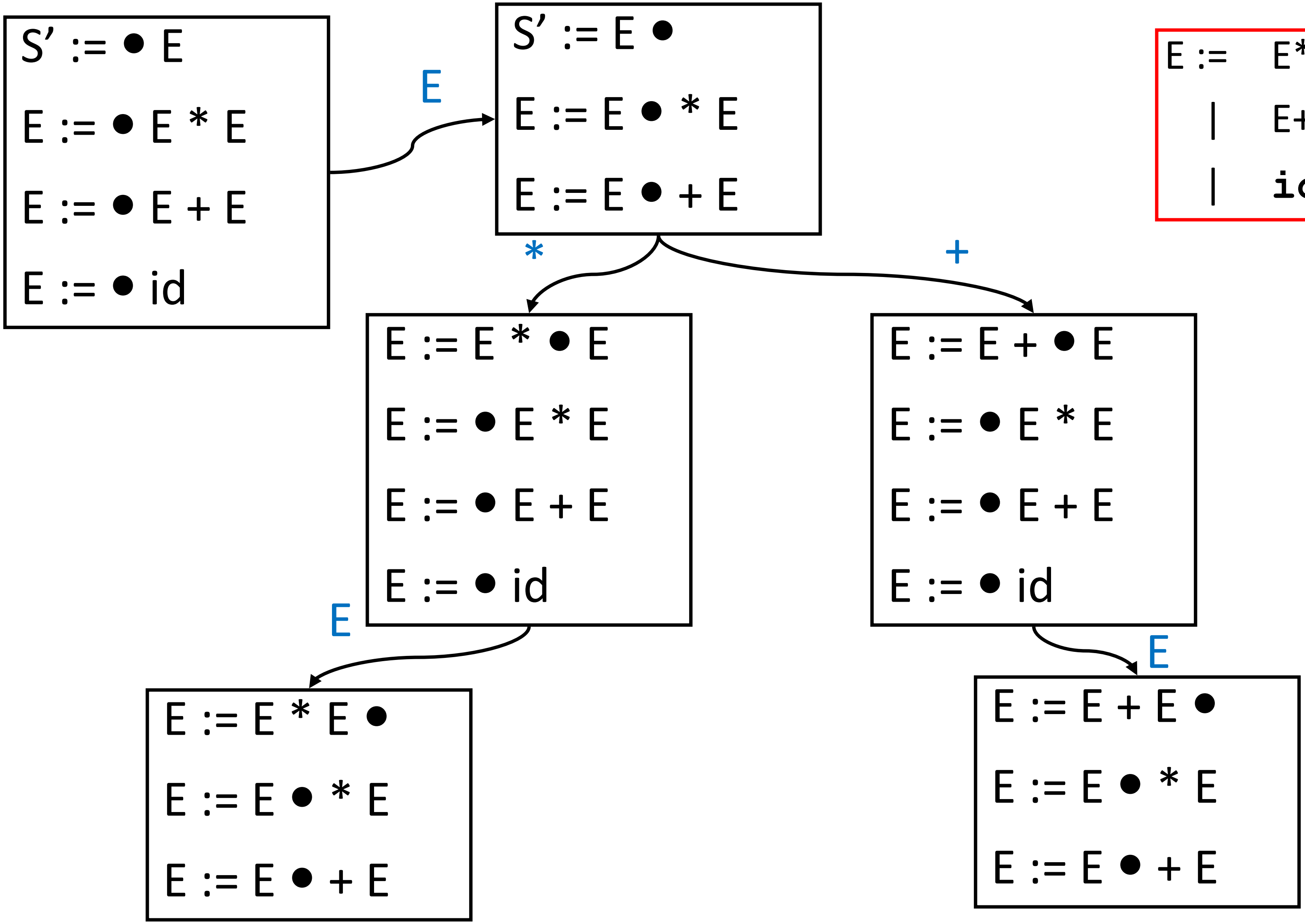
- LR(0) not very useful
- SLR uses follow sets to reduce
- LALR uses lookahead sets
- LR(1) uses full lookahead context

Power of shift-reduce parsers

- There are unambiguous grammars which which cannot be parsed with shift-reduce parsers.
- Such grammars can have
  - shift/reduce conflicts
  - reduce/reduce conflicts
- There grammars are not LR(k)
- But, we can often choose shift or reduce to recognize what want.

Expression Grammars & Precedence

```
E :=   E*E
   |   E+E
   |   id
```



S’ := ● E
E := ● E * E
E := ● E + E
E := ● id

E

S’ := E ●
E := E ● * E
E := E ● + E

*

E := E * ● E
E := ● E * E
E := ● E + E
E := ● id

E

E := E * E ●
E := E ● * E
E := E ● + E

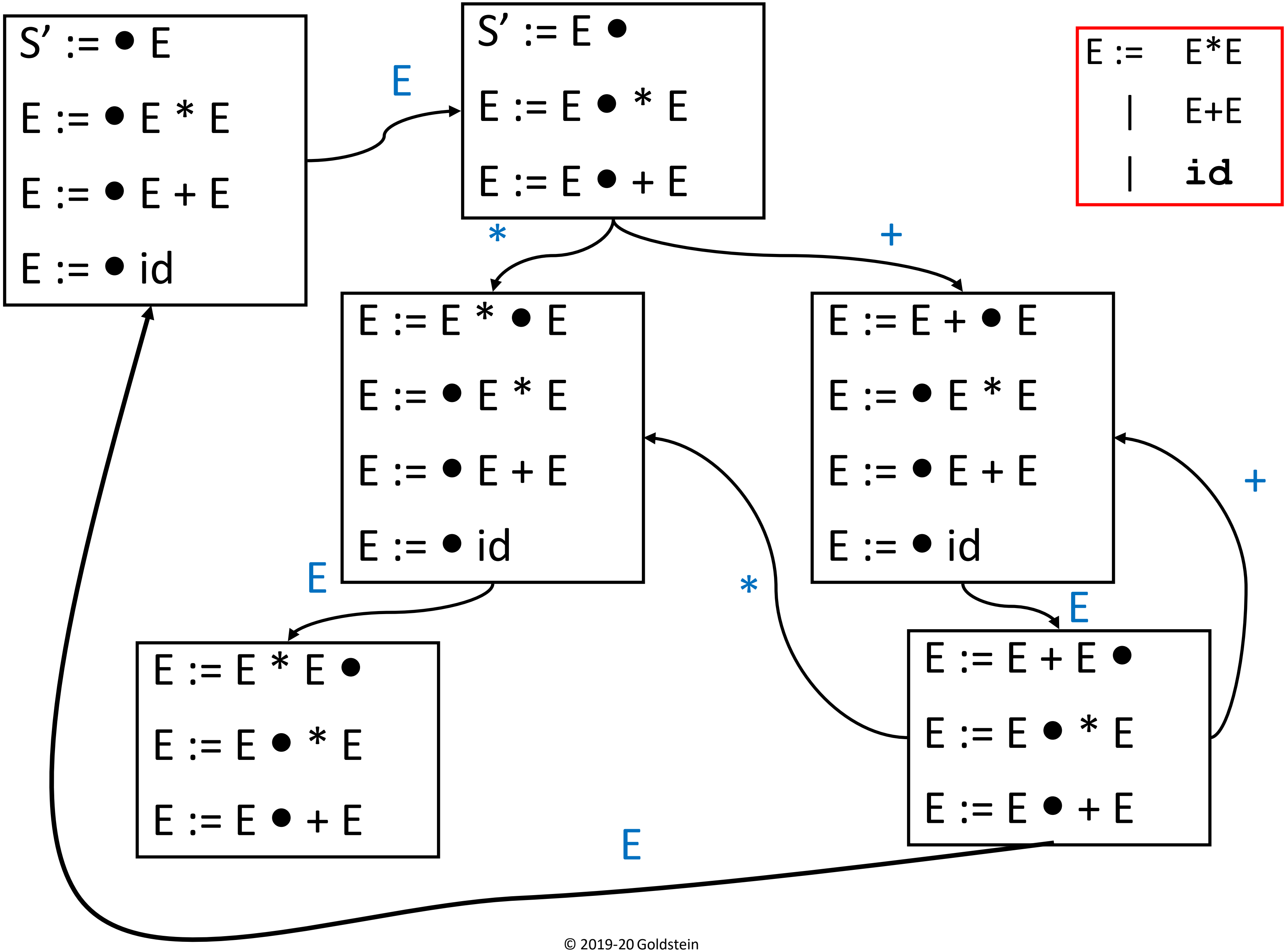
+

E := E + ● E
E := ● E * E
E := ● E + E
E := ● id

E

E := E + E ●
E := E ● * E
E := E ● + E

Expression Grammars & Precedence



S’ := ● E
E := ● E * E
E := ● E + E
E := ● id

S’ := E ●
E := E ● * E
E := E ● + E

E := E * ● E
E := ● E * E
E := ● E + E
E := ● id

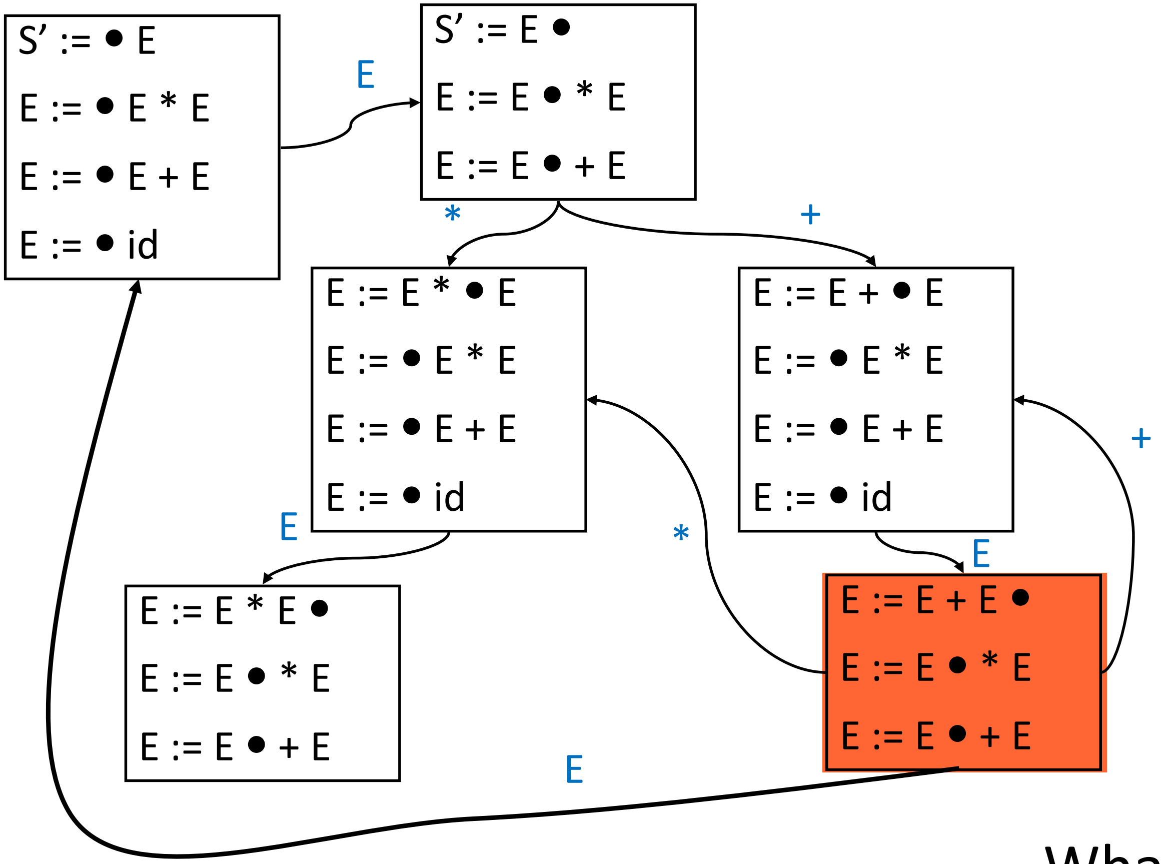
E := E + ● E
E := ● E * E
E := ● E + E
E := ● id

E := E * E ●
E := E ● * E
E := E ● + E

E := E + E ●
E := E ● * E
E := E ● + E

E := E*E
| E+E
| **id**

Handling Ambiguity



E := E*E
| E+E
| **id**

What to do on + or *?

- shift
- reduce by E → E+E?

Bison

- Precedence and Associativity declarations
- Precedence derived from order of directivies: from lowest to highest
- Associativity from %left, %right, %nonassoc
- Can be attached to rules as well (This can solve the dangling if-else problem

Dangling Else

S := **if** E **then** S

  | **if** E **then** S **else**

  | **other**

We will see a clean way to deal with this in a shift-reduce parser.

- We can be in the following state:

  … **if** E **then** S   **else** … **$**

- What do we do?
  - shift the **else** (hoping to reduce by second rule)
  - reduce by first rule

Next Time

- From words to sentences.
- From regular languages to context free languages.
- Parsing