Syntax Analysis

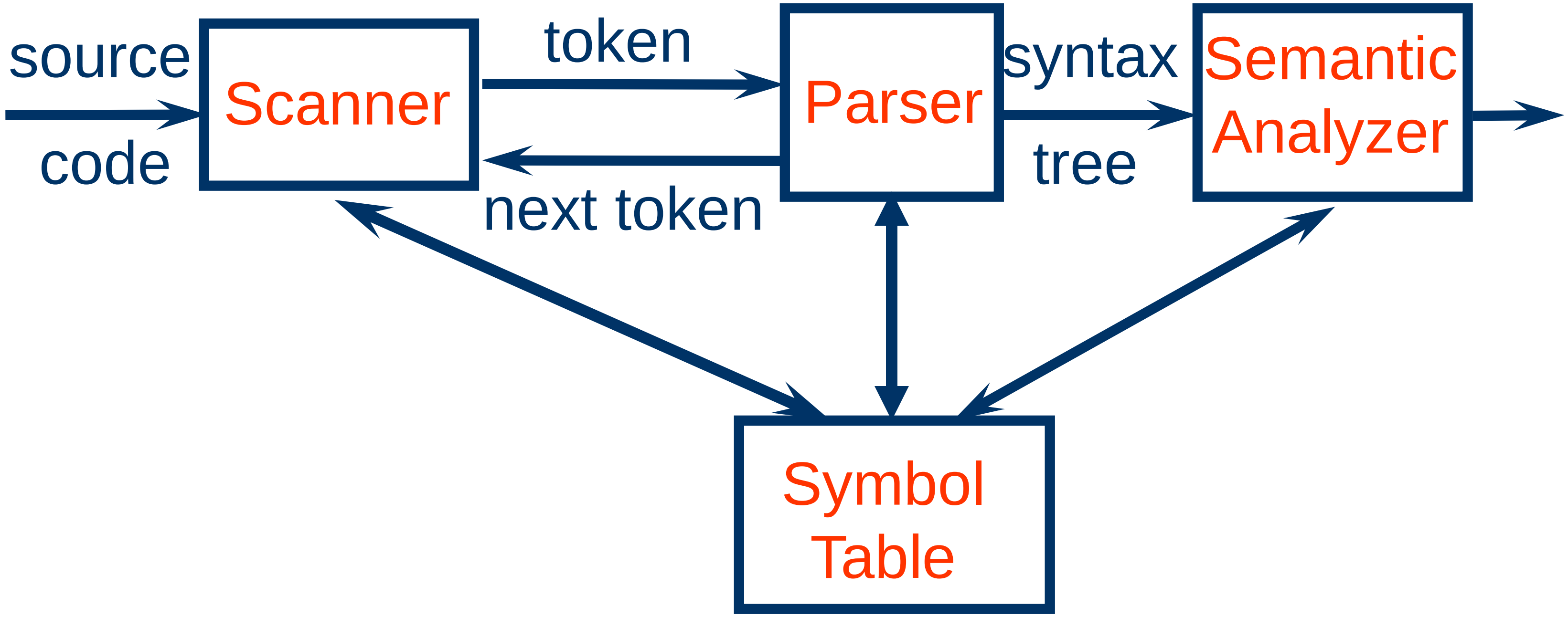
Syntax Analysis

- Syntax analysis recognizes the syntactic structure of the programming language and transforms a string of tokens into a tree of tokens
- Parser is the program that performs syntax analysis

Contents

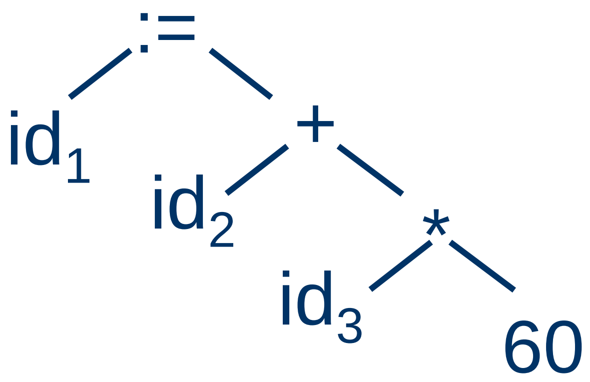
- Introduction to parser
- Syntax trees
- Context-free grammars
- Push-down automata
- Top-down parsing
- Bottom-up parsing
- Bison - a parser generator

Introduction to Parser



Syntax Trees

- A syntax tree represents the syntactic structure of tokens in a program defined by the grammar of the programming language



Context-Free Grammars (CFG)

- A set of terminals: basic symbols (token types) from which strings are formed
- A set of nonterminals: syntactic categories denoting sets of strings
- A set of productions: rules specifying how the terminals and nonterminals can be combined to form strings
- The start symbol: a distinguished nonterminal denoting the language

An Example

- Terminals: id, ‘+’, ‘-’, ‘*’, ‘/’, ‘(’, ‘)’
- Nonterminals: *expr*, *op*
- Productions:

  expr → *expr op expr*
  expr → ‘(’ *expr* ‘)’
  expr → ‘-’ *expr*
  expr → id
  op → ‘+’ | ‘-’ | ‘*’ | ‘/’

- Start symbol: *expr*

Derivations

- A derivation step is an application of a production as a rewriting rule, namely, replacing a nonterminal in the string by one of its right-hand sides

  $$\ldots N \ldots \Rightarrow \ldots \alpha \ldots$$

- Starting with the start symbol, a sequence of derivation steps is called a derivation

  $$S \Rightarrow \ldots \Rightarrow \alpha \quad \text{or} \quad S \Rightarrow^* \alpha$$

An Example

Grammar:

expr → *expr op expr*
expr → ‘(’ *expr* ‘)’
expr → ‘-’ *expr*
expr → id
op → ‘+’ | ‘-’ | ‘*’ | ‘/’

Derivation:

$\underline{\textit{expr}}$
$\Rightarrow$ - $\underline{\textit{expr}}$
$\Rightarrow$ - ($\underline{\textit{expr}}$)
$\Rightarrow$ - ($\underline{\textit{expr}}$ *op expr*)
$\Rightarrow$ - (id $\underline{\textit{op}}$ *expr*)
$\Rightarrow$ - (id + $\underline{\textit{expr}}$)
$\Rightarrow$ - (id + id)

Left- & Right-Most Derivations

- If there are more than one nonterminal in the string, many choices are possible
- A leftmost derivation always chooses the leftmost nonterminal to rewrite
- A rightmost derivation always chooses the rightmost nonterminal to rewrite

An Example

Leftmost derivation:	Rightmost derivation:
expr	*expr*
⇒ - *expr*	⇒ - *expr*
⇒ - (*expr*)	⇒ - (*expr*)
⇒ - (*expr op expr*)	⇒ - (*expr op expr*)
⇒ - (id *op expr*)	⇒ - (*expr op* id)
⇒ - (id + *expr*)	⇒ - (*expr* + id)
⇒ - (id + id)	⇒ - (id + id)

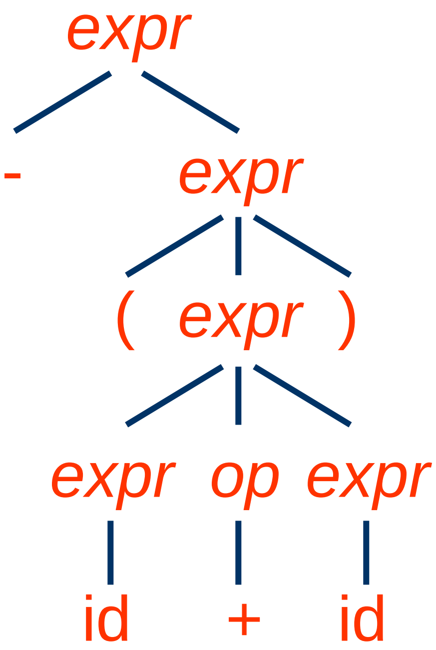
Parse Trees

- A parse tree is a graphical representation for a derivation that filters out the order of choosing nonterminals for rewriting
- Many derivations may correspond to the same parse tree, but every parse tree has associated with it a unique leftmost and a unique rightmost derivation

An Example

Leftmost derivation:

expr
⇒ - *expr*
⇒ - (*expr*)
⇒ - (*expr* *op* *expr*)
⇒ - (id *op* *expr*)
⇒ - (id + *expr*)
⇒ - (id + id)



Rightmost derivation:

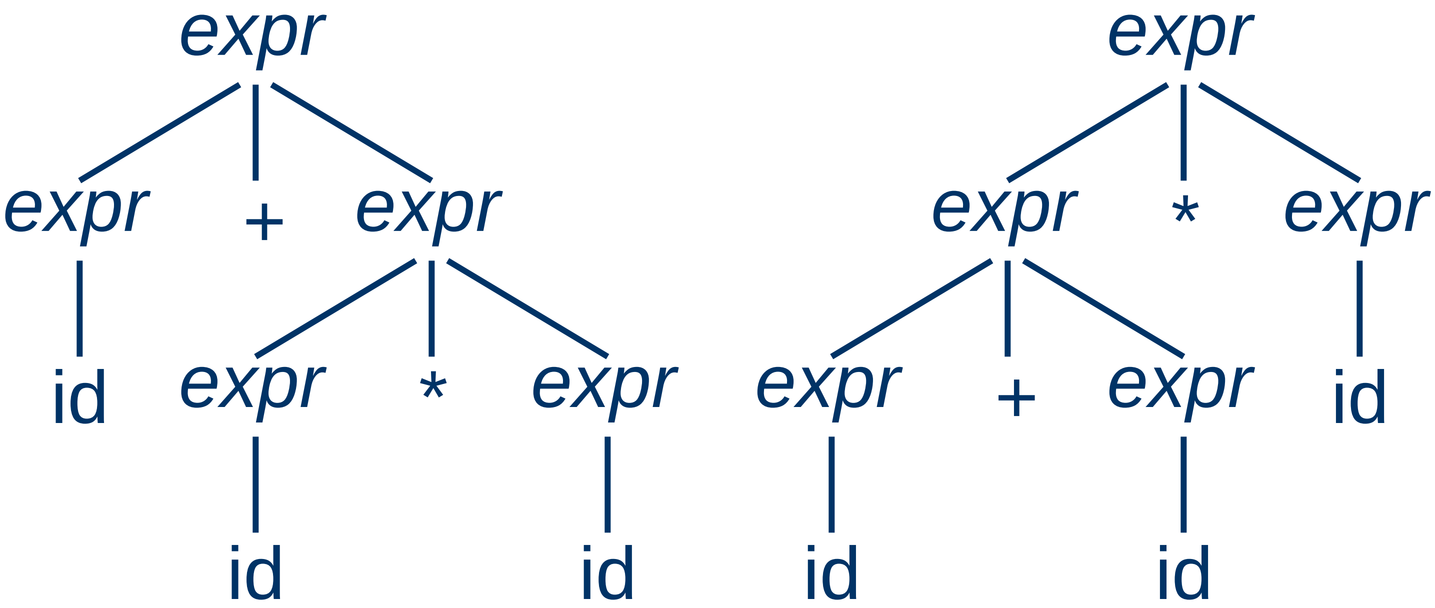
expr
⇒ - *expr*
⇒ - (*expr*)
⇒ - (*expr* *op* *expr*)
⇒ - (*expr* *op* id)
⇒ - (*expr* + id)
⇒ - (id + id)

Ambiguous Grammars

- A grammar is ambiguous if it can derive a string with two different parse trees
- If we use the syntactic structure of a parse tree to interpret the meaning of the string, the two parse trees have different meanings
- Since compilers do use parse trees to derive meaning, we would prefer to have unambiguous grammars

An Example

id + id * id



expr

expr + *expr*

id *expr* * *expr*

id id

expr

expr * *expr*

expr + *expr* id

id id

Transform Ambiguous Grammars

Ambiguous grammar:

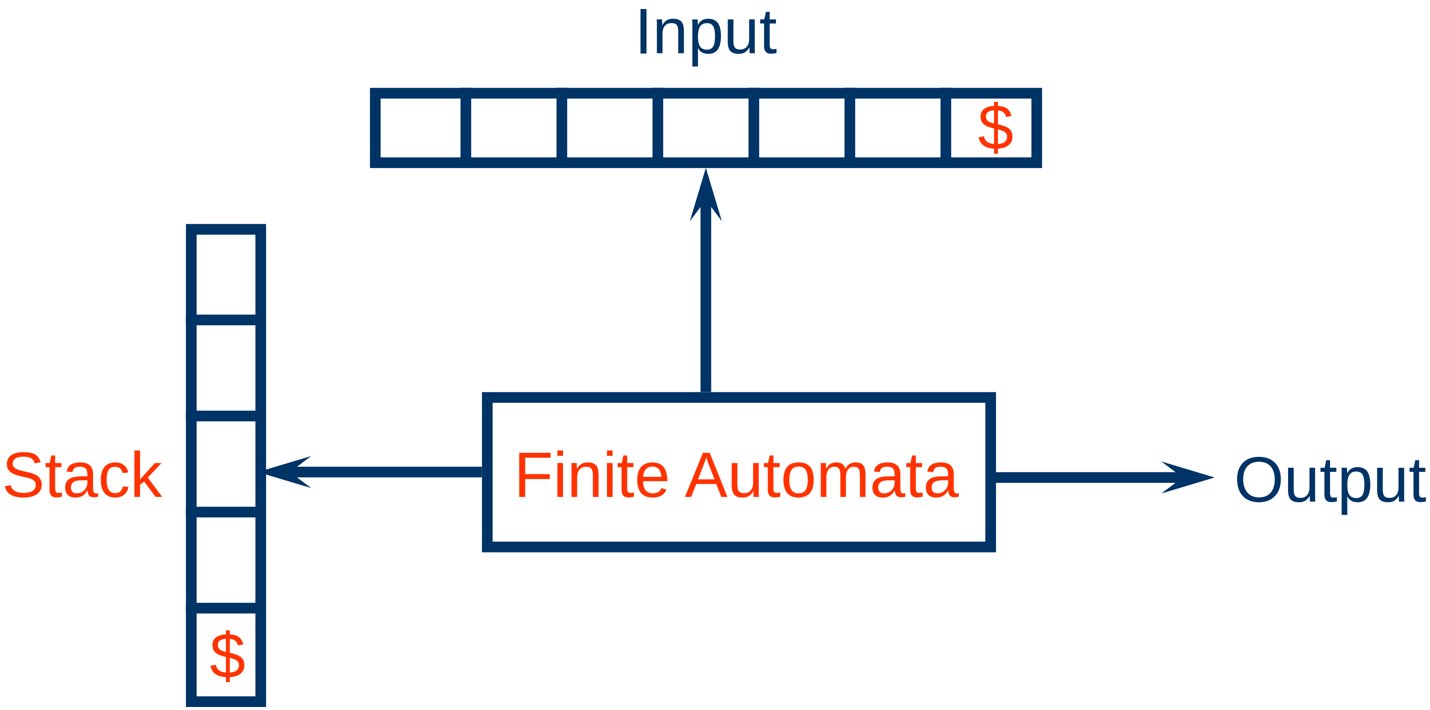
expr → *expr op expr*
expr → ‘(’ *expr* ‘)’
expr → ‘-’ *expr*
expr → id
op → ‘+’ | ‘-’ | ‘*’ | ‘/’

Not every ambiguous grammar can be transformed to an unambiguous one!

Unambiguous grammar:

expr → *expr* ‘+’ *term*
expr → *expr* ‘-’ *term*
expr → *term*
term → *term* ‘*’ *factor*
term → *term* ‘/’ *factor*
term → *factor*
factor → ‘(’ *expr* ‘)’
factor → ‘-’ *expr*
factor → id

Push-Down Automata



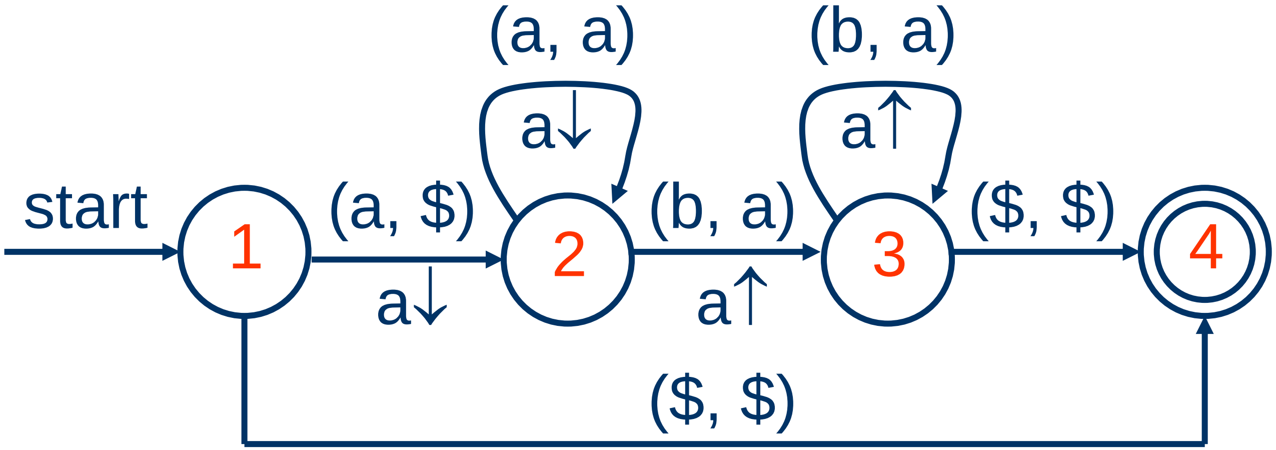
End-Of-File Marker

- Parsers must read not only terminal symbols but also the end-of-file marker
- We will use $ to represent end of file
- We will also use $ as the bottom-of-stack maker

An Example

S → a S b

S → ε



CFG versus RE

- Every language defined by a RE can also be defined by a CFG
- Why use REs for lexical syntax?
  - do not need a notation as powerful as CFGs
  - are more concise and easier to understand than CFGs
  - More efficient lexical analyzers can be constructed from REs than from CFGs
  - Provide a way for modularizing the front end into two manageable-sized components

Nonregular Languages

- REs can denote only a fixed number of repetitions or an unspecified number of repetitions of one given construct
- A nonregular language: $L = \{a^n b^n \mid n \geq 0\}$

$S \rightarrow a\ S\ b$

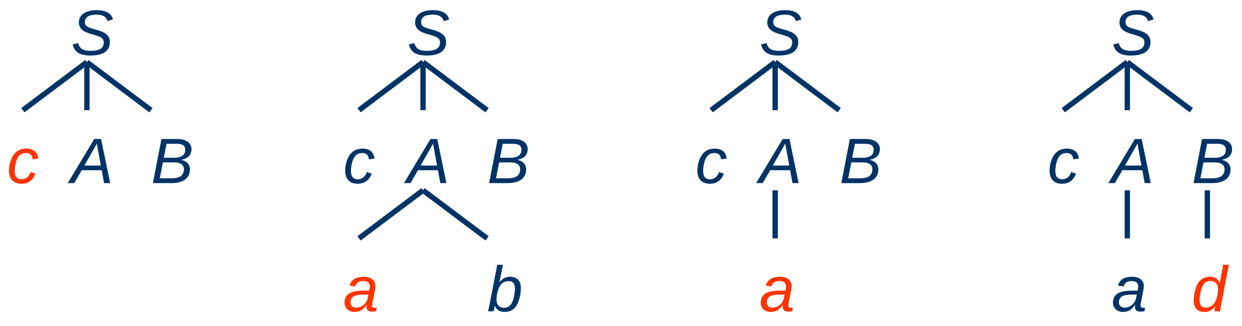
$S \rightarrow \varepsilon$

Top-Down Parsing

- Construct a parse tree from the root to the leaves using leftmost derivation

$S \rightarrow c\ A\ B$
$A \rightarrow a\ b$
$A \rightarrow a$
$B \rightarrow d$

input: *cad*


S
c A B

S
c A B
a b

S
c A B
a

S
c A B
a d

Predictive Parsing

- Predictive parsing is a top-down parsing without backtracking
- Namely, according to the next token, there is only one production to choose at each derivation step

stmt → **if** *expr* **then** *stmt* **else** *stmt*
  | **while** *expr* **do** *stmt*
  | **begin** *stmt_list* **end**

LL(k) Parsing

- Predictive parsing is also called LL(k) parsing
- The first L stands for scanning the input from left to right
- The second L stands for producing a leftmost derivation
- The k stands for using k lookahead input symbol to choose alternative productions at each derivation step

LL(1) Parsing

- We will only describe LL(1) parsing from now on, namely, parsing using only one lookahead input symbol
- Recursive-descent parsing – hand written or tool (e.g. PCCTS and CoCo/R) generated
- Table-driven predictive parsing – tool (e.g. LISA and LLGEN) generated

Recursive Descent Parsing

- A procedure is associated with each nonterminal of the grammar
- A clause in the procedure is associated with each production of that nonterminal
- A match of a token is associated with each terminal in the right hand side of the production
- A procedure call is associated with each nonterminal in the right hand side of the production

An Example

S → **if** *E* **then** *S* **else** *S*

| **begin** *L* **end**

| **print** *E*

L → *S* **;** *L*

| **ε**

E → **num = num**

An Example

```
final int IF = 1, THEN = 2, ELSE = 3, BEGIN = 4,
                END =5, PRINT = 6, SEMI = 7, NUM = 8,
                EQ = 9;
int tok = gettoken();

void advance() { tok = gettoken(); }
void match(int t)
{
    if (tok == t) advance(); else error();
}
```

An Example

```
void S() {
    switch (tok) {
        case IF: match(IF); E(); match(THEN); S();
                    match(ELSE); S(); break;
        case BEGIN: match(BEGIN); L();
                    match(END); break;
        case PRINT: match(PRINT); E(); break;
        default: error();
    }
}
```

An Example

```
void L() {
    switch (tok) {
        case END: break;
        case IF: case BEGIN: case PRINT:
                S(); match(SEMI); L(); break;
        default: error();
    }
}

void E() { match(NUM); match(EQ); match(NUM); }
```

First and Follow Sets

- The first set of a string α, FIRST(α), is the set of terminals that can begin the strings derived from α. If $\alpha \Rightarrow^* \varepsilon$, then ε is also in FIRST(α)
- The follow set of a nonterminal X, Follow(X), is the set of terminals that can immediately follow X

Computing First Sets

- If X is terminal, then FIRST(X) is {X}
- If X is nonterminal and $X \to \varepsilon$ is a production, then add ε to FIRST(X)
- If X is nonterminal and $X \to Y_1\ Y_2 \ldots Y_k$ is a production, then add a to FIRST(X) if for some i, a is in FIRST(Y_i) and ε is in all of FIRST(Y_1), ..., FIRST(Y_{i-1}). If ε is in FIRST(Y_j) for all j, then add ε to FIRST(X)

An Example

$S \rightarrow$ **if** E **then** S **else** S | **begin** L **end** | **print** E

$L \rightarrow S$ **;** $L \mid \varepsilon$

$E \rightarrow$ **num = num**

FIRST(S) = { **if**, **begin**, **print** }

FIRST(L) = { **if**, **begin**, **print** , ε }

FIRST(E) = { **num** }

Computing Follow Sets

- Place \$ in FOLLOW(S), where S is the start symbol and \$ is the end-of-file marker
- If there is a production $A \rightarrow \alpha B \beta$, then everything in FIRST(β) except for ε is placed in FOLLOW(B)
- If there is a production $A \rightarrow \alpha B$ or $A \rightarrow \alpha B \beta$ where FIRST(β) contains ε, then everything in FOLLOW(A) is in FOLLOW(B)

An Example

S → **if** *E* **then** *S* **else** *S* | **begin** *L* **end** | **print** *E*

L → *S* **;** *L* | ε

E → **num = num**

FOLLOW(*S*) = { **$**, **else**, ; }

FOLLOW(*L*) = { **end** }

FOLLOW(*E*) = { **then**, **$**, **else**, ; }

Table-Driven Predictive Parsing

Input. Grammar *G*.   Output. Parsing Table *M*.

Method.

1. For each production $A \rightarrow \alpha$ of the grammar,
   do steps 2 and 3.
2. For each terminal *a* in FIRST(α), add $A \rightarrow \alpha$ to $M[A, a]$.
3. If ε is in FIRST(α), add $A \rightarrow \alpha$ to $M[A, b]$ for each terminal *b* in FOLLOW(*A*). If ε is in FIRST(α) and \$ is in FOLLOW(*A*), add $A \rightarrow \alpha$ to $M[A, \$]$.
4. Make each undefined entry of *M* be *error*.

An Example

	S	*L*	*E*
if	*S* → **if** *E* **then** *S* **else** *S*	*L* → *S* **;** *L*	
then			
else			
begin	*S* → **begin** *L* **end**	*L* → *S* **;** *L*	
end		*L* → ε	
print	*S* → **print** *E*	*L* → *S* **;** *L*	
num			*E* → **num = num**
;			
$			

An Example

Stack	Input
$ S	begin print num = num ; end $
$ end L begin	begin print num = num ; end $
$ end L	print num = num ; end $
$ end L ; S	print num = num ; end $
$ end L ; E print	print num = num ; end $
$ end L ; E	num = num ; end $
$ end L ; num = num	num = num ; end $
$ end L ;	; end $
$ end L	end $
$ end	end $
$	$

LL(1) Grammars

- A grammar is LL(1) iff its predictive parsing table has no multiply-defined entries
- A grammar G is LL(1) iff whenever $A \rightarrow \alpha \mid \beta$ are two distinct productions of G, the following conditions hold:

  $\mathrm{FIRST}(\alpha) \cap \mathrm{FIRST}(\beta) = \varnothing$,

  If $\varepsilon \in \mathrm{FIRST}(\alpha)$, $\mathrm{FOLLOW}(A) \cap \mathrm{FIRST}(\beta) = \varnothing$,

  If $\varepsilon \in \mathrm{FIRST}(\beta)$, $\mathrm{FOLLOW}(A) \cap \mathrm{FIRST}(\alpha) = \varnothing$

A Counter Example

S → **i** E **t** S S' | **a**
S' → **e** S | ε
E → **b**

	a	**b**	**e**	**i**	**t**	**$**
S	S → **a**			S → **i** E **t** S S'		
S'			S' → ε S' → **e** S			S' → ε
E		E → **b**				

$\varepsilon \in \mathrm{FIRST}(\varepsilon) \wedge \mathrm{FOLLOW}(S') \cap \mathrm{FIRST}(\mathbf{e}\ S) = \{\mathbf{e}\} \neq \emptyset$

Left Recursive Grammars

- A grammar is left recursive if it has a nonterminal A such that $A \Rightarrow^* A\ \alpha$
- Left recursive grammars are not LL(1) because

  $A \rightarrow A\ \alpha$

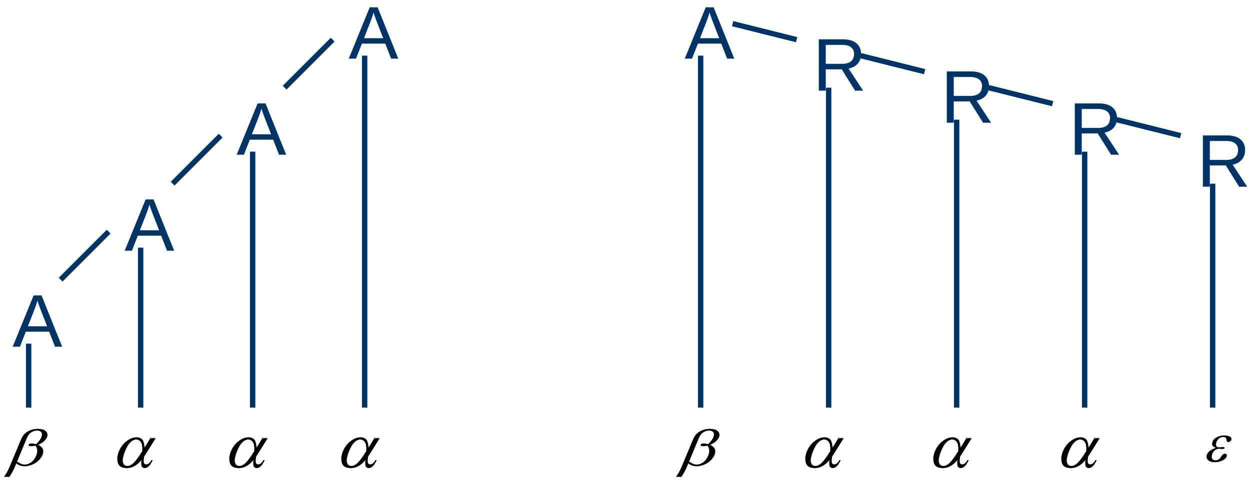
  $A \rightarrow \beta$

  will cause $\mathrm{FIRST}(A\ \alpha) \cap \mathrm{FIRST}(\beta) \neq \emptyset$
- We can transform them into LL(1) by eliminating left recursion

Eliminating Left Recursion

$A \rightarrow A\,\alpha \mid \beta \;\Rightarrow\;$ $A \rightarrow \beta\, R$

$R \rightarrow \alpha\, R \mid \varepsilon$



Direct Left Recursion

$$A \rightarrow A\,\alpha_1 \mid A\,\alpha_2 \mid \ldots \mid A\,\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$$

$$\Downarrow$$

$$A \rightarrow \beta_1\,A' \mid \beta_2\,A' \mid \ldots \mid \beta_n\,A'$$

$$A' \rightarrow \alpha_1\,A' \mid \alpha_2\,A' \mid \ldots \mid \alpha_m\,A' \mid \varepsilon$$

An Example

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid \textbf{id}$

⇩

$E \rightarrow T E'$

$E' \rightarrow + T E' \mid \varepsilon$

$T \rightarrow F T'$

$T' \rightarrow * F T' \mid \varepsilon$

$F \rightarrow (E) \mid \textbf{id}$

Indirect Left Recursion

$S \rightarrow A\,a \mid b$

$A \rightarrow A\,c \mid S\,d \mid \varepsilon$

$S \Rightarrow A\,a \Rightarrow S\,d\,a$

$A \rightarrow A\,c \mid A\,a\,d \mid b\,d \mid \varepsilon$

⇩

$S \rightarrow A\,a \mid b$

$A \rightarrow b\,d\,A' \mid A'$

$A' \rightarrow c\,A' \mid a\,d\,A' \mid \varepsilon$

Left factoring

- A grammar is not LL(1) if two productions of a nonterminal A have a nontrivial common prefix. For example, if $\alpha \neq \varepsilon$, and $A \rightarrow \alpha\ \beta_1 \mid \alpha\ \beta_2$, then FIRST($\alpha\ \beta_1$) $\cap$ FIRST($\alpha\ \beta_2$) $\neq \varnothing$
- We can transform them into LL(1) by performing left factoring

$$A \rightarrow \alpha\ A'$$

$$A' \rightarrow \beta_1 \mid \beta_2$$

An Example

$S \rightarrow \mathbf{i}\ E\ \mathbf{t}\ S \mid \mathbf{i}\ E\ \mathbf{t}\ S\ \mathbf{e}\ S \mid \mathbf{a}$

$E \rightarrow \mathbf{b}$

⇩

$S \rightarrow \mathbf{i}\ E\ \mathbf{t}\ S\ S' \mid \mathbf{a}$

$S' \rightarrow \mathbf{e}\ S \mid \varepsilon$

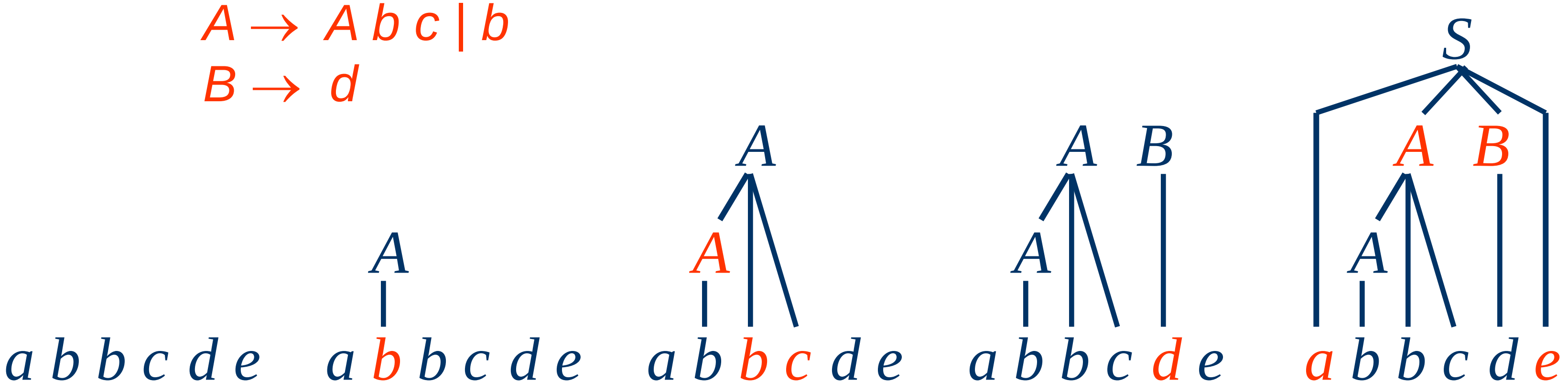
$E \rightarrow \mathbf{b}$

Bottom-Up Parsing

- Construct a parse tree from the leaves to the root using rightmost derivation in reverse

S → a A B e
A → A b c | b
B → d

input: *abbcde*



$abbcde \Leftarrow aAbcde \Leftarrow aAde \Leftarrow aABe \Leftarrow S$

LR(k) Parsing

- The L stands for scanning the input from left to right
- The R stands for producing a rightmost derivation
- The k stands for using k lookahead input symbol to choose alternative productions at each derivation step

An Example

1. *S'* → *S*
2. *S* → **if** *E* **then** *S* **else** *S*
3. *S* → **begin** *L* **end**
4. *S* → **print** *E*
5. *L* → *S* **;** *L*
6. *L* → **ε**
7. *E* → **num = num**

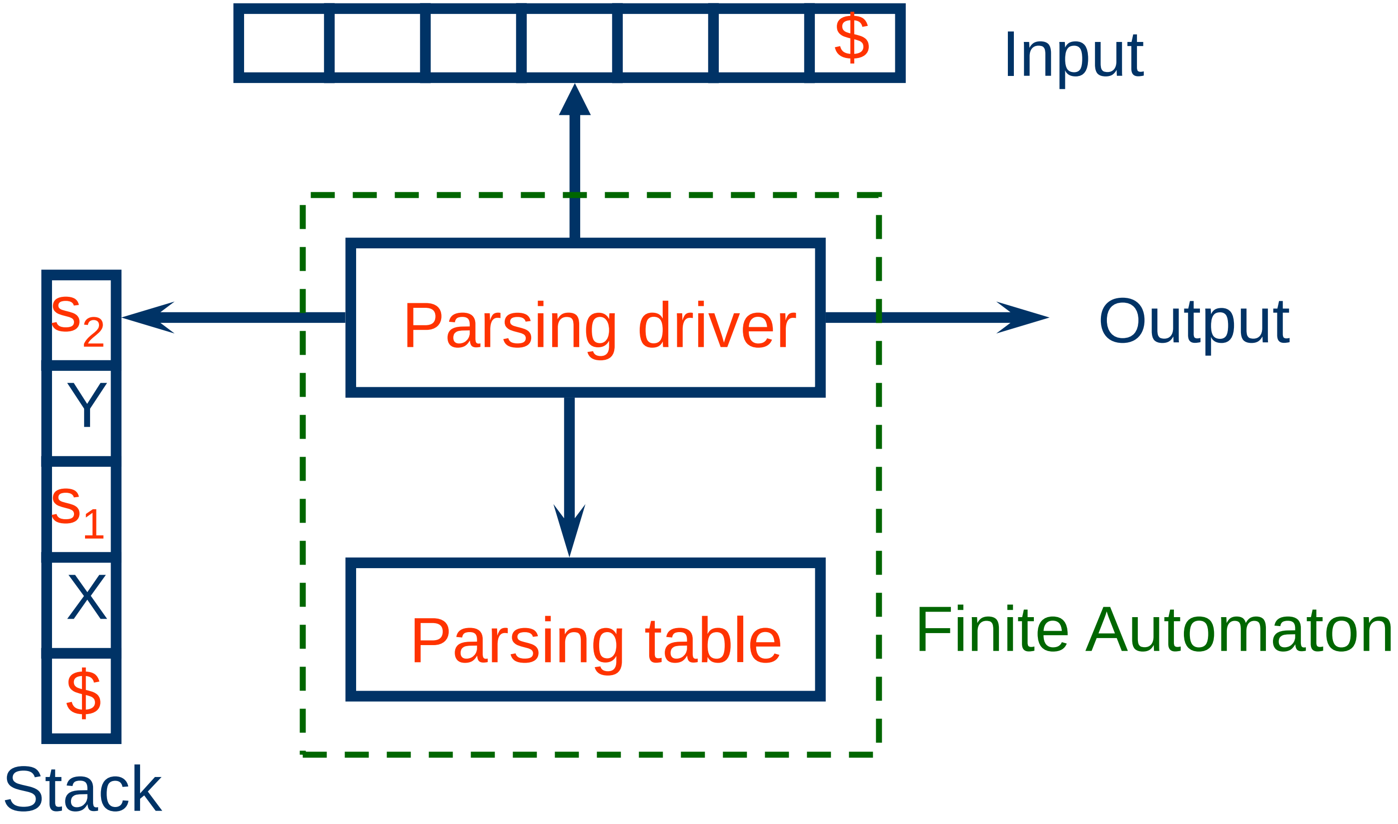
An Example

Stack	Input	Action
$	begin print num = num ; end $	shift
$ begin	print num = num ; end $	shift
$ begin print	num = num ; end $	shift
$ begin print num	= num ; end $	shift
$ begin print num =	num ; end $	shift
$ begin print num = num	; end $	reduce
$ begin print E	; end $	reduce
$ begin S	; end $	shift
$ begin S ;	end $	reduce
$ begin S ; L	end $	reduce
$ begin L	end $	shift
$ begin L end	$	reduce
$ S	$	accept

LL(k) versus LR(k)

- LL(k) parsing must predict which production to use after seeing only the first k tokens of the right-hand side
- LR(k) parsing is able to postpone the decision until it has seen tokens corresponding to the entire right-hand side and k more tokens beyond
- LR(k) parsing thus can handle more grammars than LL(k) parsing

LR Parsers


$
Input
Parsing driver
Output
Parsing table
Finite Automaton
s_2
Y
s_1
X
$
Stack

LR Parsing Tables

	if	then	else	begin	end	print	;	num	=	$	S	L	E
1	s3			s4		s5					g2		
2										a			
3								s7					g6
4	s3			s4	r5	s5					g9	g8	
5								s7					g10
6		s11											
7									s12				
8					s13								
9							s14						
10		r4	r4				r4			r4			

LR Parsing Tables

	if	then	else	begin	end	print	;	num	=	$	S	L	E
11	s3			s4		s5					g15		
12								s16					
13		r3					r3			r3			
14					r5						g9	g17	
15			s18										
16		r7	r7				r7			r7			
17					r6								
18	s3			s4		s5					g19		
19			r2				r2			r2			

action

goto

An Example

Stack	Input	Action
$\$_1$	begin print num = num ; end $	s4
$\$_1\text{begin}_4$	print num = num ; end $	s5
$\$_1\text{begin}_4\text{print}_5$	num = num ; end $	s7
$\$_1\text{begin}_4\text{print}_5\text{num}_7$	= num ; end $	s12
$\$_1\text{begin}_4\text{print}_5\text{num}_7{=}_{12}$	num ; end $	s16
$\$_1\text{begin}_4\text{print}_5\text{num}_7{=}_{12}\text{num}_{16}$	; end $	r7
$\$_1\text{begin}_4\text{print}_5 E_{10}$	; end $	r4
$\$_1\text{begin}_4 S_9$	; end $	s14
$\$_1\text{begin}_4 S_9 ;_{14}$	end $	r5
$\$_1\text{begin}_4 S_9 ;_{14} L_{17}$	end $	r6
$\$_1\text{begin}_4 L_8$	end $	s13
$\$_1\text{begin}_4 L_8\text{end}_{13}$	$	r3
$\$_1 S_2$	$	a

LR Parsing Driver

```
while (true) {
   s = top();  a = gettoken();
   if (action[s, a] == shift s') { push(a); push(s'); }
   else if (action[s, a] == reduce A → α) {
       pop 2 * | α | symbols off the stack;
       s' = goto[top(), A]; push(A); push(s'); }
   else if (action[s, a] == accept) { return; }
   else { error(); }
}
```

LR Parsing Table Generation

- An LR parsing table generation algorithm transforms a CFG to an LR parsing table
- SLR(1) parsing table generation
- LR(1) parsing table generation
- LALR(1) parsing table generation

From CFG to NPDA

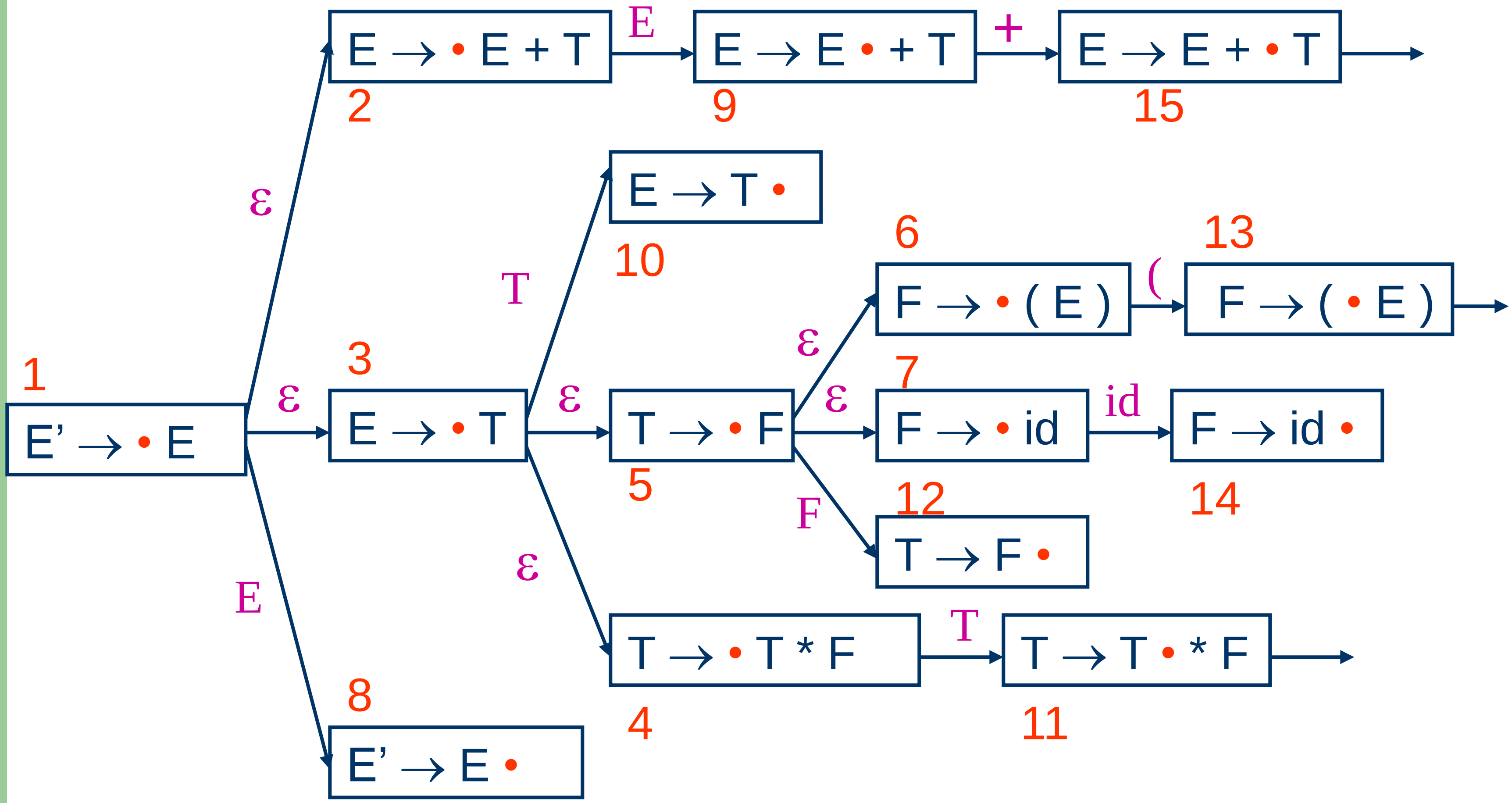
- An LR(0) item of a grammar in G is a production of G with a dot at some position of the right-hand side, $A \rightarrow \alpha \bullet \beta$
- The production $A \rightarrow X\ Y\ Z$ yields the following four LR(0) items

  $A \rightarrow \bullet\, X\ Y\ Z$, $A \rightarrow X \bullet Y\ Z$,
  $A \rightarrow X\ Y \bullet Z$, $A \rightarrow X\ Y\ Z\, \bullet$
- An LR(0) item represents a state in a NPDA indicating how much of a production we have seen at a given point in the parsing process

An Example

1. E’ → E
2. E → E + T
3. E → T
4. T → T * F
5. T → F
6. F → (E)
7. F → **id**

An Example



2 E → • E + T

9 E → E • + T

15 E → E + • T

10 E → T •

6 F → • (E)

13 F → (• E)

1 E’ → • E

3 E → • T

5 T → • F

7 F → • id

14 F → id •

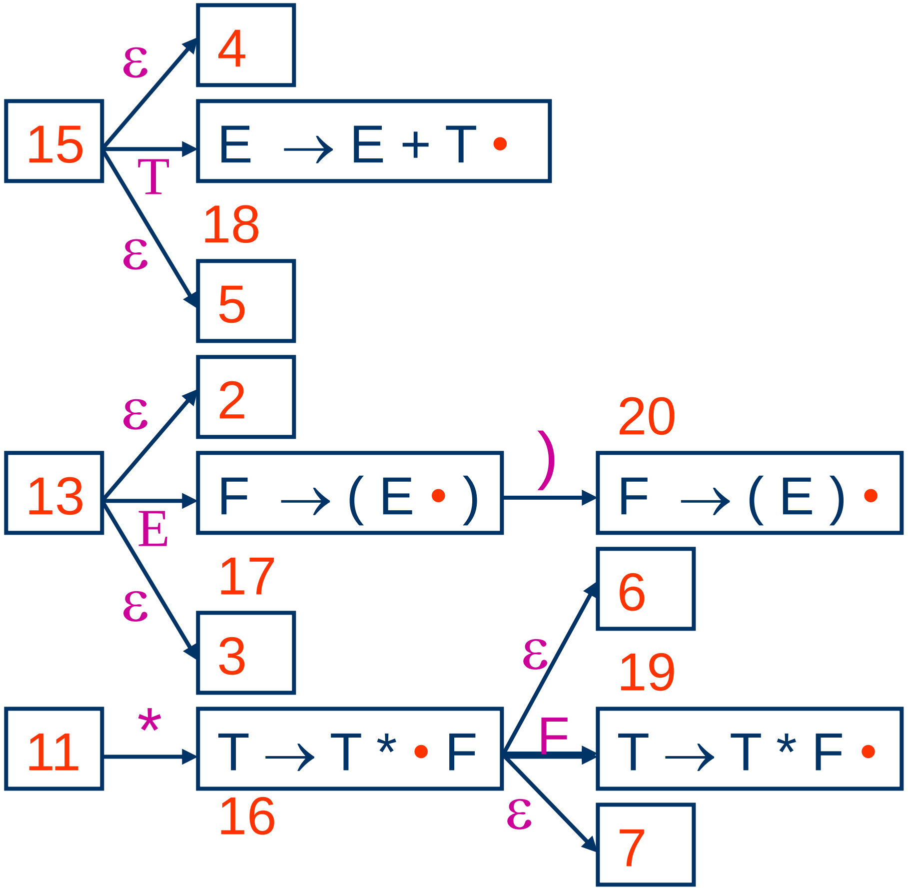
12 T → F •

4 T → • T * F

11 T → T • * F

8 E’ → E •

An Example



15

ε

4

T

E → E + T •

18

ε

5

13

ε

2

E

F → (E •)

)

20

F → (E) •

17

ε

3

11

*

T → T * • F

16

ε

6

F

19

T → T * F •

ε

7

From NPDA to DPDA

- There are two functions performed on sets of LR(0) items (states)
- The function closure(I) adds more items to I when there is a dot to the left of a nonterminal
- The function goto(I, X) moves the dot past the symbol X in all items in I that contain X

The Closure Function

```
closure(I) =
    repeat
        for any item $A \to \alpha \bullet X \beta$ in I
            for any production $X \to \gamma$
                $I = I \cup \{ X \to \bullet \gamma \}$
    until I does not change
    return I
```

An Example

1. $E' \rightarrow E$
2. $E \rightarrow E + T$
3. $E \rightarrow T$
4. $T \rightarrow T * F$
5. $T \rightarrow F$
6. $F \rightarrow (E)$
7. $F \rightarrow \textbf{id}$

$s_1 = E' \rightarrow \bullet E$,

$I_1 = closure(\{s_1\}) = \{$

$E' \rightarrow \bullet E$,

$E \rightarrow \bullet E + T$,

$E \rightarrow \bullet T$,

$T \rightarrow \bullet T * F$,

$T \rightarrow \bullet F$,

$F \rightarrow \bullet (E)$,

$F \rightarrow \bullet \textbf{id} \}$

The Goto Function

```
goto(I, X) =
    set J to the empty set
    for any item $A \rightarrow \alpha \bullet X \beta$ in I
        add $A \rightarrow \alpha X \bullet \beta$ to J
    return closure(J)
```

An Example

$I_1 = \{E' \rightarrow \bullet E,$
$E \rightarrow \bullet E + T, E \rightarrow \bullet T,$
$T \rightarrow \bullet T * F, T \rightarrow \bullet F,$
$F \rightarrow \bullet (E), F \rightarrow \bullet \mathbf{id} \}$

$\text{goto}(I_1 , E)$
$= \text{closure}(\{E' \rightarrow E \bullet, E \rightarrow E \bullet + T \})$
$= \{E' \rightarrow E \bullet, E \rightarrow E \bullet + T \}$

The Subset Construction Function

```
subset-construction(cfg) =
    initialize T to {closure({S’ → • S})}
    repeat
        for each state I in T and each symbol X
            let J be goto(I, X)
            if J is not empty and not in T then
                T = T ∪ { J }
    until T does not change
    return T
```

An Example

$I_1 : \{E' \rightarrow \bullet E, E \rightarrow \bullet E + T, E \rightarrow \bullet T, T \rightarrow \bullet T * F,$
$T \rightarrow \bullet F, F \rightarrow \bullet (E), F \rightarrow \bullet \mathbf{id}\}$

$goto(I_1, E) = I_2 : \{E' \rightarrow E \bullet, E \rightarrow E \bullet + T\}$
$goto(I_1, T) = I_3 : \{E \rightarrow T \bullet, T \rightarrow T \bullet * F\}$
$goto(I_1, F) = I_4 : \{T \rightarrow F \bullet\}$
$goto(I_1, \text{‘(’}) = I_5 : \{F \rightarrow (\bullet E), E \rightarrow \bullet E + T, E \rightarrow \bullet T$
$T \rightarrow \bullet T * F, T \rightarrow \bullet F, F \rightarrow \bullet (E), F \rightarrow \bullet \mathbf{id}\}$
$goto(I_1, \mathbf{id}) = I_6 : \{F \rightarrow \mathbf{id} \bullet\}$

$goto(I_2, \text{‘+’}) = I_7 : \{E \rightarrow E + \bullet T, T \rightarrow \bullet T * F, T \rightarrow \bullet F$
$F \rightarrow \bullet (E), F \rightarrow \bullet \mathbf{id}\}$

An Example

goto(I_3, ‘*’) = I_8 : {T → T * • F, F → • (E), F → • **id**}

goto(I_5, E) = I_9 : {F → (E •), E → E • + T}
goto(I_5, T) = I_3
goto(I_5, F) = I_4
goto(I_5, ‘(’) = I_5
goto(I_5, **id**) = I_6

goto(I_7, T) = I_{10} : E → E + T •, T → T • * F}
goto(I_7, F) = I_4
goto(I_7, ‘(’) = I_5
goto(I_7, **id**) = I_6

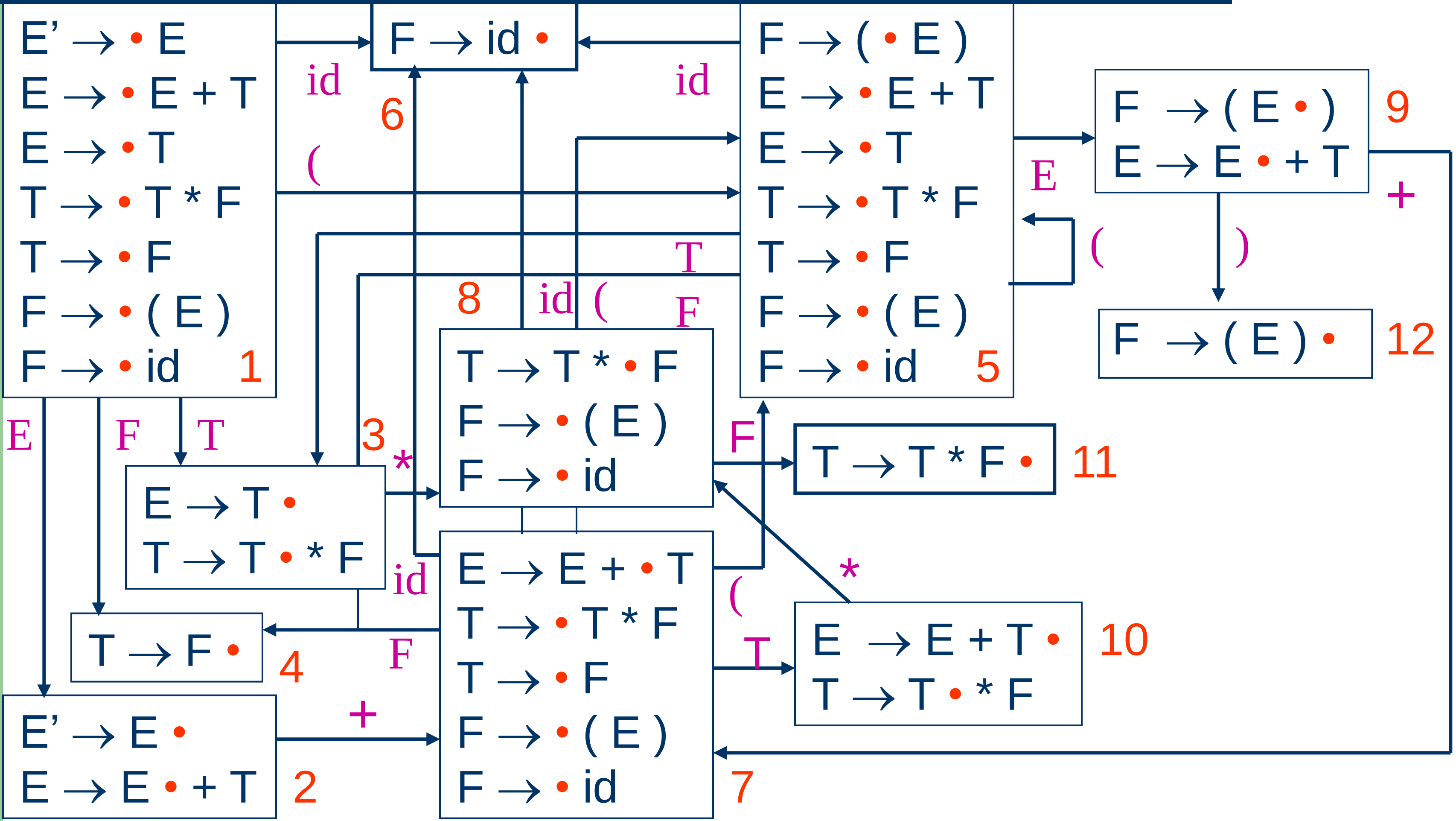
An Example

goto(I_8, F) = I_{11} : {T $\rightarrow$ T * F •}
goto(I_8, ‘(‘) = I_5
goto(I_8, **id**) = I_6

goto(I_9, ‘)’) = I_{12} : {F $\rightarrow$ (E) •}
goto(I_9, ‘+’) = I_7

goto(I_{10}, ‘*’) = I_8

An Example



E' → • E
E → • E + T
E → • T
T → • T * F
T → • F
F → • (E)
F → • id 1

F → id • 6

F → (• E)
E → • E + T
E → • T
T → • T * F
T → • F
F → • (E)
F → • id 5

F → (E •)
E → E • + T 9

F → (E) • 12

8
T → T * • F
F → • (E)
F → • id

T → T * F • 11

3
E → T •
T → T • * F

E → E + • T
T → • T * F
T → • F
F → • (E)
F → • id 7

E → E + T •
T → T • * F 10

T → F • 4

E' → E •
E → E • + T 2

SLR(1) Parsing Table Generation

```
SLR(cfg) =
  for each state I in subset-construction(cfg)
    if $A \rightarrow \alpha \bullet a \beta$ in I and goto(I, a) = J for a terminal a then
      action[I, a] = “shift J”
    if $A \rightarrow \alpha \bullet$ in I and $A \neq S'$ then
      action[I, a] = “reduce $A \rightarrow \alpha$” for all a in Follow(A)
    if $S' \rightarrow S \bullet$ in I then action[I, $] = “accept”
    if $A \rightarrow \alpha \bullet X \beta$ in I and goto(I, X) = J for a nonterminal X
      then goto[I, X] = J
    all other entries in action and goto are made error
```

An Example

	+	*	(	)	id	$	E	T	F
1			s5		s6		g2	g3	g4
2	s7					a			
3	r3	s8		r3		r3			
4	r5	r5		r5		r5			
5			s5		s6		g9	g3	g4
6	r7	r7		r7		r7			
7			s5		s6			g10	g4
8			s5		s6				g11
9	s7			s12					
10	r2	s8		r2		r2			

An Example

	+	*	(	)	id	$	E	T	F
11	r4	r4		r4		r4			
12	r6	r6		r6		r6			

LR(I) Items

- An LR(1) item of a grammar in G is a pair, ($A \rightarrow \alpha \bullet \beta$, a), of an LR(0) item $A \rightarrow \alpha \bullet \beta$ and a lookahead symbol a
- The lookahead has no effect in an LR(1) item of the form ($A \rightarrow \alpha \bullet \beta$, a), where β is not ε
- An LR(1) item of the form ($A \rightarrow \alpha \bullet$, a) calls for a reduction by $A \rightarrow \alpha$ only if the next input symbol is a

The Closure Function

```
closure(I) =
    repeat
        for any item $(A \rightarrow \alpha \bullet X \beta, a)$ in I
            for any production $X \rightarrow \gamma$
                for any $b \in First(\beta a)$
                    $I = I \cup \{ (X \rightarrow \bullet \gamma, b) \}$
    until I does not change
    return I
```

An Example

1. $S' \rightarrow S$
2. $S \rightarrow C\ C$
3. $C \rightarrow c\ C$
4. $C \rightarrow d$

I_1 = closure($\{(S' \rightarrow \bullet\ S, \$)\}$) =

$\{(S' \rightarrow \bullet\ S, \$)$,

First($) = {$}

$(S \rightarrow \bullet\ C\ C, \$)$,

$(C \rightarrow \bullet\ c\ C, c)$, $(C \rightarrow \bullet\ c\ C, d)$,

$(C \rightarrow \bullet\ d, c)$, $(C \rightarrow \bullet\ d, d)\}$

First(C$) = {c, d}

The Goto Function

```
goto(I, X) =
    set J to the empty set
    for any item $(A \rightarrow \alpha \bullet X \beta, a)$ in I
        add $(A \rightarrow \alpha X \bullet \beta, a)$ to J
    return closure(J)
```

An Example

$$
\begin{aligned}
& \text{goto}(I_1, C) \\
= \; & \text{closure}(\{S \rightarrow C \bullet C, \$)\}) \\
= \; & \{S \rightarrow C \bullet C, \$), (C \rightarrow \bullet\, c\, C, \$), (C \rightarrow \bullet\, d, \$)\}
\end{aligned}
$$

The Subset Construction Function

```
subset-construction(cfg) =
    initialize T to {closure({($S' \to \bullet S$ , \$)})}
    repeat
        for each state I in T and each symbol X
            let J be goto(I, X)
            if J is not empty and not in T then
                $T = T \cup \{ J \}$
    until T does not change
    return T
```

An Example

1. S’ → S
2. S → C C
3. C → c C
4. C → d

An Example

I_1: closure($\{(S' \rightarrow \bullet\ S, \$)\}$) =
$(S' \rightarrow \bullet\ S, \$)$
$(S \rightarrow \bullet\ C\ C, \$)$
$(C \rightarrow \bullet\ c\ C, c/d)$
$(C \rightarrow \bullet\ d, c/d)$

I_2: goto(I_1, S) = $(S' \rightarrow S\ \bullet, \$)$

I_3: goto(I_1, C) =
$(S \rightarrow C\ \bullet\ C, \$)$
$(C \rightarrow \bullet\ c\ C, \$)$
$(C \rightarrow \bullet\ d, \$)$

I_4: goto(I_1, c) =
$(C \rightarrow c\ \bullet\ C, c/d)$
$(C \rightarrow \bullet\ c\ C, c/d)$
$(C \rightarrow \bullet\ d, c/d)$

I_5: goto(I_1, d) =
$(C \rightarrow d\ \bullet, c/d)$

I_6: goto(I_3, C) =
$(S \rightarrow C\ C\ \bullet, \$)$

An Example

I_7: goto(I_3, c) =

$(C \rightarrow c \bullet C, \$)$

$(C \rightarrow \bullet c C, \$)$

$(C \rightarrow \bullet d, \$)$

I_8: goto(I_3, d) =

$(C \rightarrow d \bullet, \$)$

I_9: goto(I_4, C) =

$(C \rightarrow c C \bullet, c/d)$

: goto(I_4, c) = I_4

: goto(I_4, d) = I_5

I_{10}: goto(I_7, C) =

$(C \rightarrow c C \bullet, \$)$

: goto(I_7, c) = I_7

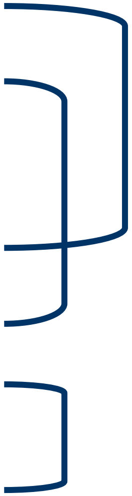
: goto(I_7, d) = I_8

LR(1) Parsing Table Generation

```
LR(cfg) =
  for each state I in subset-construction(cfg)
    if ($A \rightarrow \alpha \bullet a\ \beta$, b) in I and goto(I, a) = J for a terminal a
      then action[I, a] = “shift J”
    if ($A \rightarrow \alpha \bullet$, a) in I and $A \neq S'$
      then action[I, a] = “reduce $A \rightarrow \alpha$”
    if ($S' \rightarrow S \bullet$ , $) in I then action[I, $] = “accept”
    if ($A \rightarrow \alpha \bullet X\ \beta$, a) in I and goto(I,X) = J for a nonterminal X
      then goto[I, X] = J
    all other entries in action and goto are made error
```

An Example

	c	d	$	S	C
1	s4	s5		g2	g3
2			a		
3	s7	s8			g6
4	s4	s5			g9
5	r4	r4			
6			r2		
7	s7	s8			g10
8			r4		
9	r3	r3			
10			r3		



The Core of LR(1) Items

- The core of a set of LR(1) Items is the set of their first components (i.e., LR(0) items)
- The core of the set of LR(1) items

  { (C → c • C, c/d),
  (C → • c C, c/d),
  (C → • d, c/d) }

  is { C → c • C,
  C → • c C,
  C → • d }

Merging Cores

I_4: { ($C \to c \bullet C$, c/d), ($C \to \bullet c C$, c/d), ($C \to \bullet d$, c/d) }
$\cup$ I_7: { ($C \to c \bullet C$, \$), ($C \to \bullet c C$, \$), ($C \to \bullet d$, \$) }
$\Rightarrow$ I_{47}: { ($C \to c \bullet C$, c/d/\$), ($C \to \bullet c C$, c/d/\$),
($C \to \bullet d$, c/d/\$) }

I_5: { ($C \to d \bullet$, c/d) } $\cup$ I_8: { ($C \to d \bullet$, \$) }
$\Rightarrow$ I_{58}: { ($C \to d \bullet$, c/d/\$) }

I_9: { ($C \to c C \bullet$, c/d) } $\cup$ I_{10}: { ($C \to c C \bullet$, \$) }
$\Rightarrow$ I_{910}: { ($C \to c C \bullet$, c/d/\$) }

LALR(1) Parsing Table Generation

```
LALR(cfg) =
  for each state I in merge-core(subset-construction(cfg))
    if ($A \to \alpha \bullet a \beta$, b) in I and goto(I, a) = J for a terminal a
      then action[I, a] = “shift J”
    if ($A \to \alpha \bullet$, a) in I and $A \neq S'$
      then action[I, a] = “reduce $A \to \alpha$”
    if ($S' \to S \bullet$, $) in I then action[I, $] = “accept”
    if ($A \to \alpha \bullet X \beta$, a) in I and goto(I,X) = J for a nonterminal X
      then goto[I, X] = J
    all other entries in action and goto are made error
```

An Example

	c	d	$	S	C
1	s47	s58		g2	g3
2			a		
3	s47	s58			g6
47	s47	s58			g910
58	r4	r4	r4		
6			r2		
910	r3	r3	r3		

Shift/Reduce Conflicts

stmt → **if** *expr* **then** *stmt*
| **if** *expr* **then** *stmt* **else** *stmt*
| **other**

Stack	Input
$ - - - **if** *expr* **then** *stmt*	**else** - - - $

Shift ⇒ **if** *expr* **then** *stmt* **else** *stmt*
Reduce ⇒ **if** *expr* **then** *stmt*

Reduce/Reduce Conflicts

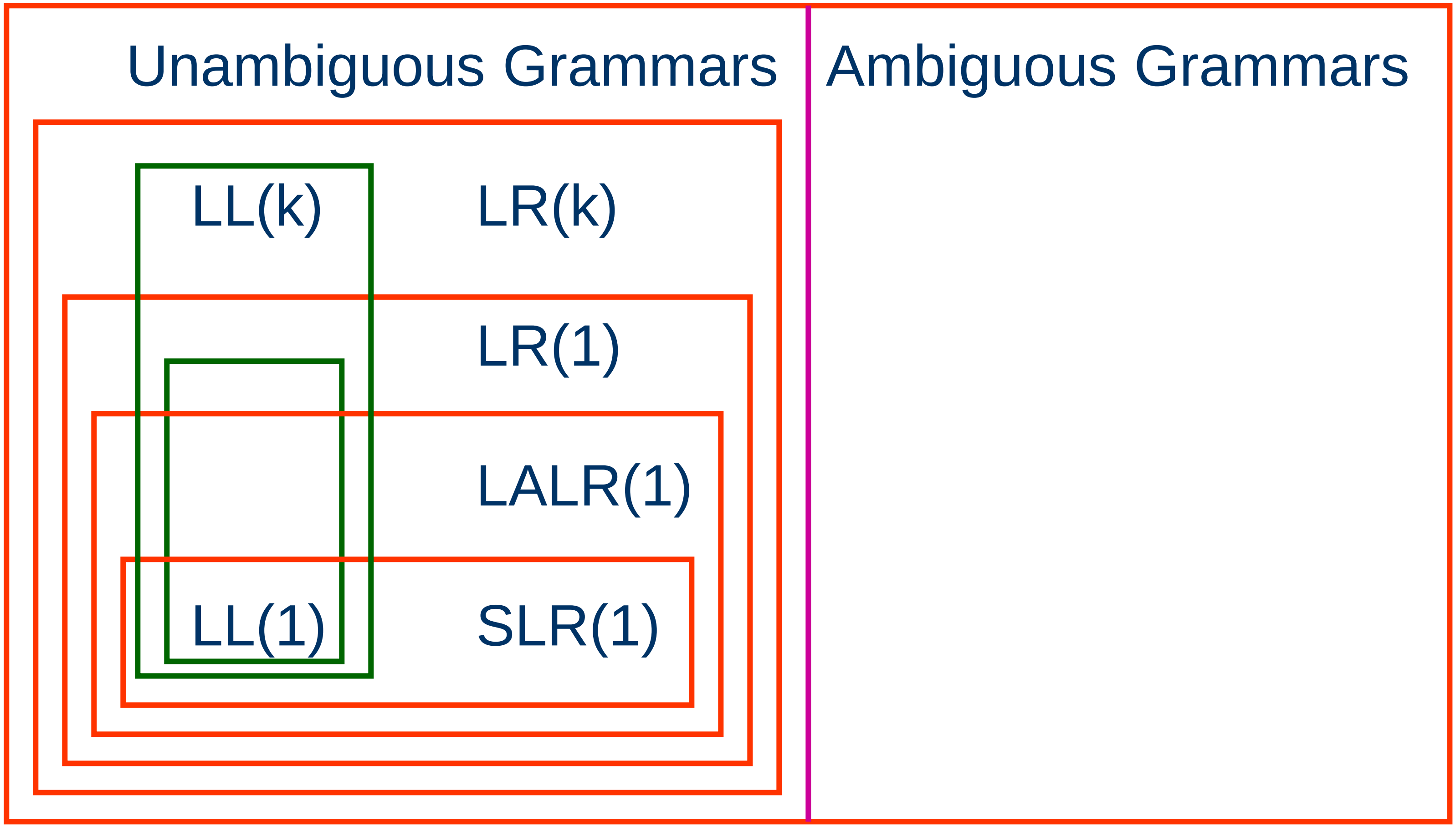
stmt → **id (** *para_list* **) |** *expr* **:=** *expr*
para_list → *para_list* **,** *para* | *para*
para → **id**
expr_list → *expr_list* **,** *expr* | *expr*
expr → **id (** *expr_list* **) | id**

Stack	Input
$ - - - **id (id**	**, id)** - - - $
$- - - **procid (id**	**, id)** - - - $

LR Grammars

- A grammar is SLR(1) iff its SLR(1) parsing table has no multiply-defined entries
- A grammar is LR(1) iff its LR(1) parsing table has no multiply-defined entries
- A grammar is LALR(1) iff its LALR(1) parsing table has no multiply-defined entries

Hierarchy of Grammar Classes



Unambiguous Grammars

Ambiguous Grammars

LL(k)

LR(k)

LR(1)

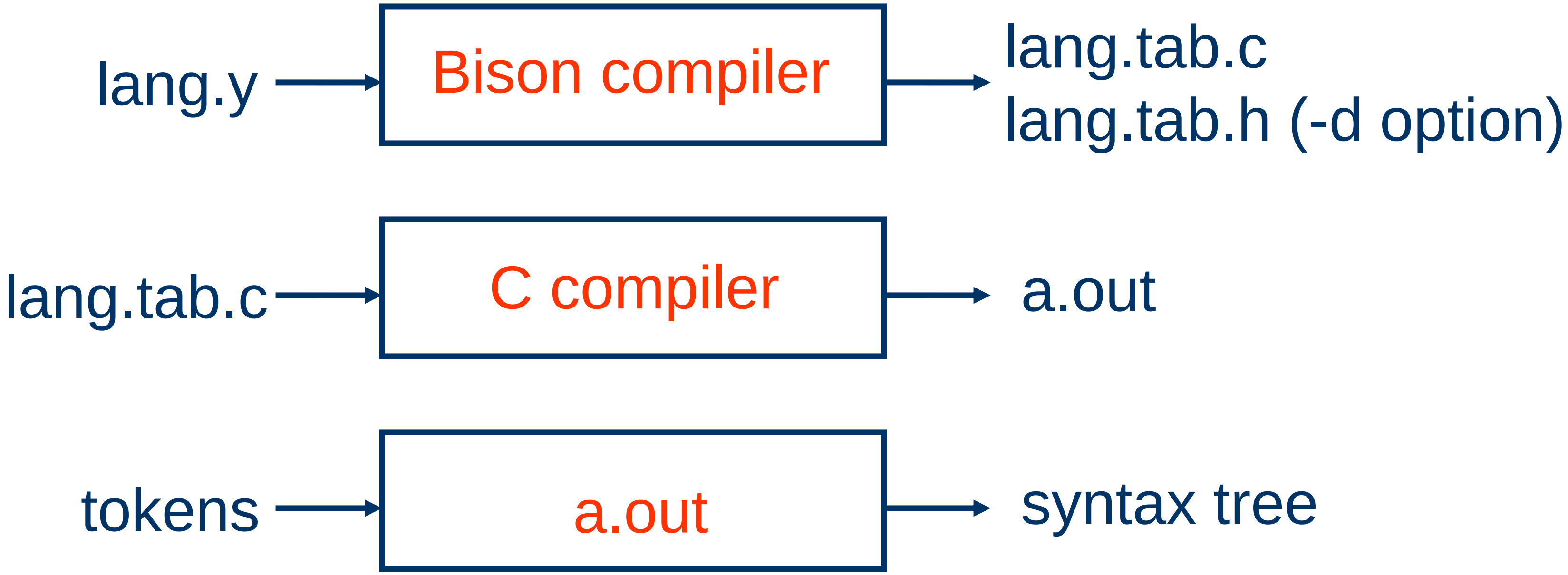
LALR(1)

LL(1)

SLR(1)

Bison – A Parser Generator

A langauge for specifying parsers and semantic analyzers



lang.y → Bison compiler → lang.tab.c
lang.tab.h (-d option)

lang.tab.c → C compiler → a.out

tokens → a.out → syntax tree

Bison Programs

```
%{
C declarations
%}
Bison declarations
%%
Grammar rules
%%
Additional C code
```

An Example

line → expr ‘\n’
expr → expr ‘+’ term | term
term → term ‘*’ factor | factor
factor → ‘(’ expr ‘)’ | DIGIT

An Example - expr.y

```
%token DIGIT
%start line
```

```
%%
line:  expr  ‘\n’
       ;
expr:  expr  ‘+’  term
       |  term
       ;
term:  term  ‘*’ factor
       |  factor
       ;
factor:  ‘(’  expr  ‘)’
       |  DIGIT
       ;
```

An Example - expr.y

```
%token NEWLINE
%token ADD
%token MUL
%token LP
%token RP
%token DIGIT
%start line
```

```
%%
line:  expr  NEWLINE
       ;
expr:  expr  ADD  term
       |  term
       ;
term:  term  MUL factor
       |  factor
       ;
factor:  LP  expr  RP
       |  DIGIT
       ;
```

An Example - expr.tab.h

```
#define NEWLINE  278
#define ADD  279
#define MUL  280
#define LP  281
#define RP 282
#define DIGIT 283
```

Semantic Actions

```
line:  expr  ‘\n’  {printf(“line: expr \\n\n”);}
      ;
expr:  expr  ‘+’  term {printf(“expr: expr + term\n”);}
      |  term  {printf(“expr: term\n”}
      ;
term:  term  ‘*’ factor  {printf(“term: term * factor\n”;}
      |  factor  {printf(“term: factor\n”);}
      ;
factor:  ‘(’  expr  ‘)’  {printf(“factor: ( expr )\n”);}
      |  DIGIT  {printf(“factor: DIGIT\n”);}
      ;
```

Semantic action

Functions

- yyparse(): the parser function
- yylex(): the lexical analyzer function. Bison recognizes any non-positive value as indicating the end of the input

Variables

- yylval: the attribute value of a token. Its default type is int, and can be declared to be multiple types in the first section using

```
%union {
        int  ival;
        double  dval;
}
```

- Tokens with attribute value can be declared as

```
%token  <ival>  intcon
%token  <dval> doublecon
```

Conflict Resolutions

- A reduce/reduce conflict is resolved by choosing the production listed first
- A shift/reduce conflict is resolved in favor of shift
- A mechanism for assigning precedences and assocoativities to terminals

Precedence and Associativity

- The precedence and associativity of operators are declared simultaneously

```
%nonassoc  '<'          /* lowest */
%left  '+'  '-'
%right  '^'             /* highest */
```

- The precedence of a rule is determined by the precedence of its rightmost terminal
- The precedence of a rule can be modified by adding %prec <terminal> to its right end

An Example

```
%{
#include  <stdio.h>
%}

%token NUMBER
%left  ‘+’  ‘-’
%left  ‘*’  ‘/’
%right  UMINUS

%%
```

An Example

```
line :  expr  ‘\n’
        ;
expr:  expr  ‘+’ expr
        |  expr  ‘-’  expr
        |  expr  ‘*’  expr
        |  expr  ‘/’  expr
        |  ‘-’  expr  %prec  UMINUS
        |  ‘(’  expr  ‘)’
        |  NUMBER
        ;
```

Error Report

- The parser can report a syntax error by calling the user provided function yyerror(char *)

```
yyerror(char *s)
{
    fprintf(stderr, “%s: line %d\n”, s, yylineno);
}
```