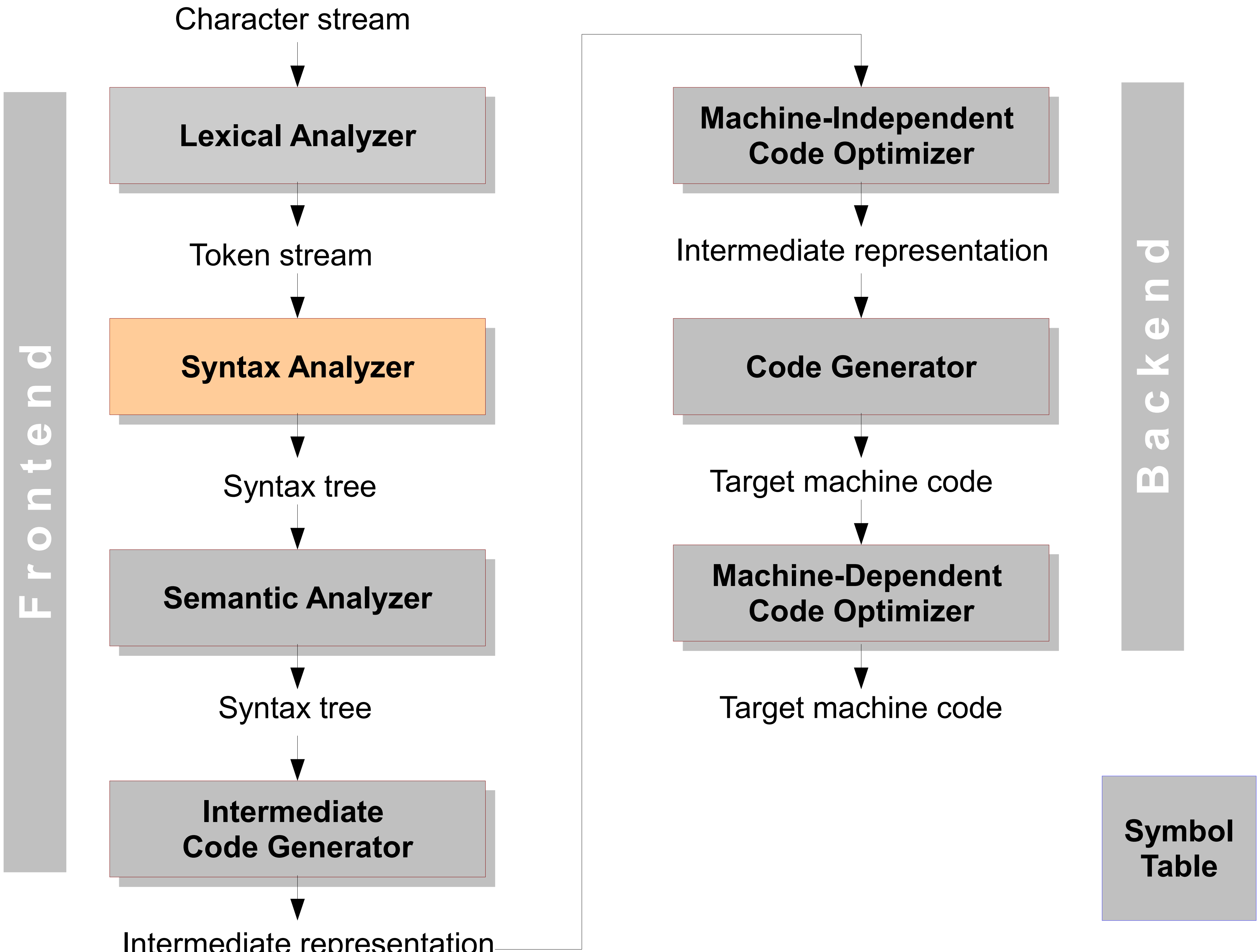
Parsing

Rupesh Nasre.

CS3300 Compiler Design
IIT Madras
Aug 2015



Character stream
Lexical Analyzer
Token stream
Syntax Analyzer
Syntax tree
Semantic Analyzer
Syntax tree
Intermediate
Code Generator
Intermediate representation
Machine-Independent
Code Optimizer
Intermediate representation
Code Generator
Target machine code
Machine-Dependent
Code Optimizer
Target machine code
Symbol
Table
Frontend
Backend

Jobs of a Parser

- Read specification given by the language implementor.
- Get help from lexer to collect tokens.
- Check if the sequence of tokens matches the specification.
- Declare successful program structure or report errors in a useful manner.
- Later: Also identify some semantic errors.

Parsing Specification

- In general, one can write a string manipulation program to recognize program structures (e.g., Lab 2).
- However, the string manipulation / recognition can be generated from a higher level description.
- We use Context-Free Grammars to specify.
  - Precise, easy to understand + modify, correct translation + error detection, incremental language development.

CFG

1. A set of terminals called *tokens*.
   - Terminals are elementary symbols of the parsing language.
2. A set of non-terminals called variables.
   - A non-terminal represents a set of strings of terminals.
3. A set of **productions**.
   - They define the syntactic rules.
4. A **start** symbol designated by a non-terminal.

list → list + digit
list → list – digit
list → digit
digit → *0* | *1* | *...* | *8* | *9*

Productions, Derivations and Languages

list → list + digit
list → list – digit
list → digit
digit → *0* | *1* | *...* | *8* | *9*

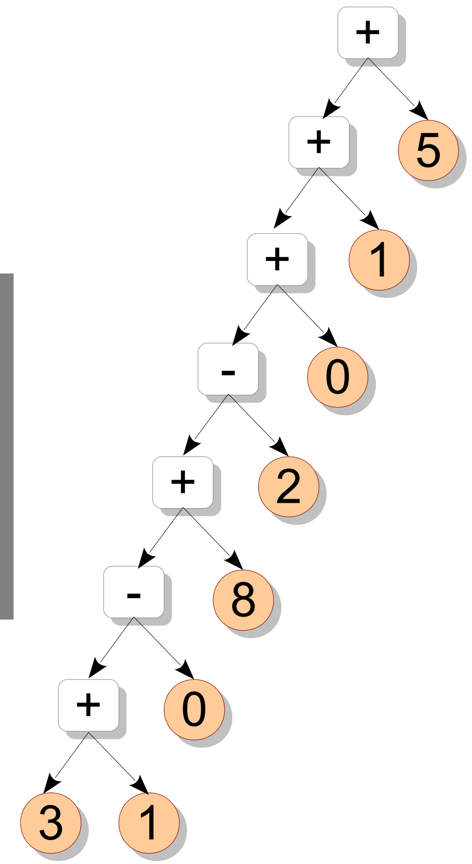
left or head

right or body

- We say a production is *for* a non-terminal if the non-terminal is the head of the production (first production is for list).
- A grammar *derives* strings by beginning with the start symbol and repeatedly replacing a non-terminal by the body of a production for that non-terminal (the grammar derives 3+1-0+8-2+0+1+5).
- The terminal strings that can be derived from the start symbol form the *language* defined by the grammar (0, 1, ..., 9, 0+0, 0-0, ... or infix expressions on digits involving plus and minus).

Parse Tree

list → list + digit
list → list – digit
list → digit
digit → *0* | *1* | *...* | *8* | *9*



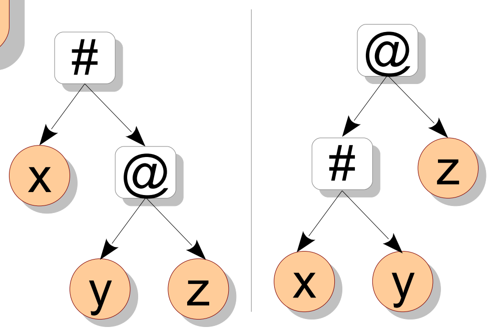
3+1-0+8-2+0+1+5

- A parse tree is a pictorial representation of operator evaluation.

Precedence

What if both the operators are the same?

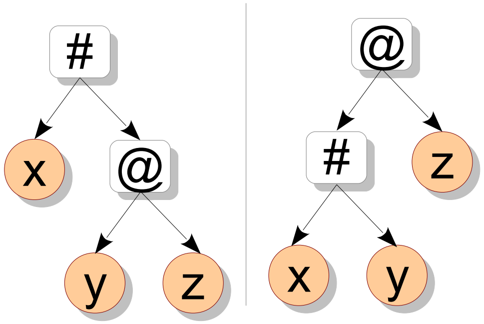
- x # y @ z
  - How does a compiler know whether to execute # first or @ first?
  - Think about x+y*z vs. x/y-z
  - A similar situation arises in if-if-else.



- Humans and compilers may “see” different parse trees.

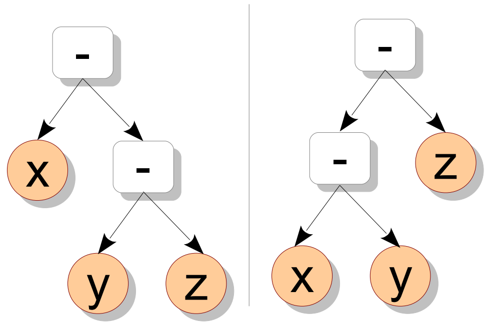
```
#define MULT(x)  x*x
int main() {
      printf(“%d”, MULT(3 + 1));
}
```

Same Precedence



x + y + z

Order of evaluation doesn't matter.

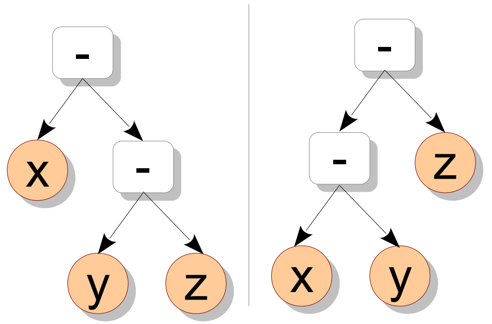


x - y - z

Order of evaluation matters.

Associativity

- Associativity decides the order in which multiple instances of same-priority operations are executed.
  - Binary minus is left associative, hence x-y-z is equal to (x-y)-z.



Homework: Write a C program to find out that assignment operator = is right associative.

Grammar for Expressions

Why is the grammar of expressions written this way?

$E \longrightarrow E + T \mid E - T \mid T$
$T \longrightarrow T * F \mid T / F \mid F$
$F \longrightarrow (E) \mid \text{number} \mid \text{name}$

Ambiguous / Unambiguous Grammars

Grammar for simple arithmetic expressions

$E \longrightarrow E + E \mid E * E \mid E - E \mid E / E \mid (E) \mid \text{number} \mid \text{name}$

Precedence not encoded

a + b * c

$E \longrightarrow E + E \mid E - E \mid T$
$T \longrightarrow T * T \mid T / T \mid F$
$F \longrightarrow (E) \mid \text{number} \mid \text{name}$

Associativity not encoded

a – b - c

$E \longrightarrow E + T \mid E - T \mid T$
$T \longrightarrow T * F \mid T / F \mid F$
$F \longrightarrow (E) \mid \text{number} \mid \text{name}$

Unambiguous grammar

Homework: Find out the issue with the final grammar.

Ambiguous / Unambiguous Grammars

Grammar for simple arithmetic expressions

$E \longrightarrow E + E \mid E * E \mid E - E \mid E / E \mid (E) \mid number \mid name$

Precedence not encoded
a + b * c

$E \longrightarrow E + E \mid E - E \mid T$
$T \longrightarrow T * T \mid T / T \mid F$
$F \longrightarrow (E) \mid number \mid name$

Associativity not encoded
a – b - c

$E \longrightarrow E + T \mid E - T \mid T$
$T \longrightarrow T * F \mid T / F \mid F$
$F \longrightarrow (E) \mid number \mid name$

Unambiguous grammar
Left recursive, not suitable for top-down parsing

$E \longrightarrow T\ E'$
$E' \longrightarrow + T\ E' \mid - T\ E' \mid \epsilon$
$T \longrightarrow F\ T'$
$T' \longrightarrow * F\ T' \mid / F\ T' \mid \epsilon$
$F \longrightarrow (E) \mid number \mid name$

Non-left-recursive grammar
Can be used for top-down parsing

Sentential Forms

- Example grammar — E → E + E | E * E | – E | (E) | id
- Sentence / string — - (id + id)
- Derivation — E => - E => - (E) => - (E + E) => - (id + E) => - (id + id)
- Sentential forms — E, -E, -(E), ..., - (id + id)
  - At each derivation step we make two choices
  - One, which non-terminal to replace
  - Two, which production to pick with that non-terminal as the head

E => -E => - (E) => - (E + E) => - (**E + id**) => - (id + id)

- Would it be nice if a parser doesn't have this confusion?

Leftmost, Rightmost

- Two special ways to choose the non-terminal
  - Leftmost: the leftmost non-terminal is replaced.

    E => -E => - (E) => - (E + E) => - (id + E) => - (id + id)

  - Rightmost: ...

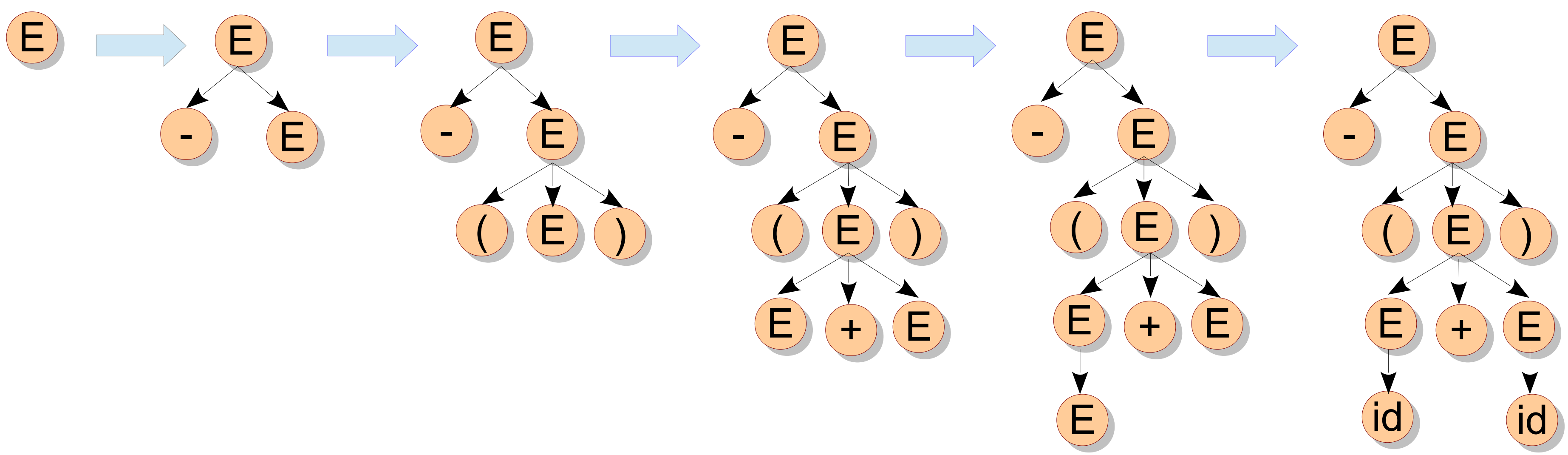
    E => -E => - (E) => - (E + E) => - (**E + id**) => - (id + id)

- Thus, we can talk about left-sentential forms and right-sentential forms.
- Rightmost derivations are sometimes called *canonical* derivations.

Parse Trees

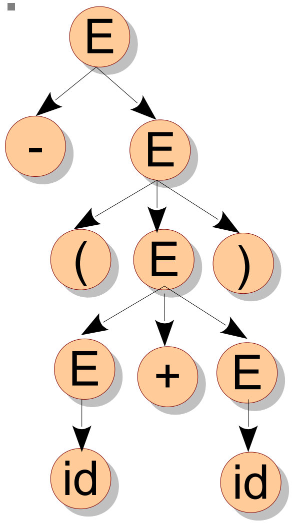
- Two special ways to choose the non-terminal
  - Leftmost: the leftmost non-terminal is replaced.

E => -E => - (E) => - (E + E) => - (id + E) => - (id + id)



Parse Trees

- Given a parse tree, it is unclear which order was used to derive it.
  - Thus, a parse is a pictorial representation of future operator order.
  - It is oblivious to a specific derivation order.
- Every parse tree has a unique leftmost derivation and a unique rightmost derivation
  - We will use them in uniquely identifying a parse tree.



Context-Free vs Regular

- We can write grammars for regular expressions.
  - Consider our regular expression *(a|b)*abb*.
  - We can write a grammar for it.

$$
\begin{aligned}
A &\longrightarrow aA \mid bA \mid aB \\
B &\longrightarrow bC \\
C &\longrightarrow bD \\
D &\longrightarrow \epsilon
\end{aligned}
$$

  - This grammar can be mechanically generated from an NFA.

Classwork

- Write a CFG for postfix expressions *{a,+,-,*,/}*.
  - Give the leftmost derivation for *aa-aa*/a+*.
  - Is your grammar ambiguous or unambiguous?
- What is this language: $S \rightarrow aSbS \mid bSaS \mid \epsilon$?
  - Draw a parse tree for *aabbab*.
  - Give the rightmost derivation for *aabbab*.
- Palindromes, unequal number of *a*s and *b*s, no substring 011.
- **Homework:** Section 4.2.8.

Error Recovery, viable prefix

- Panic-mode recovery
  - Discard input symbols until synchronizing tokens e.g. } or ;.
  - Does not result in infinite loop.
- Phrase-level recovery
  - Local correction on the remaining input
  - e.g., replace comma by semicolon, delete a char
- Error productions
  - Augment grammar with error productions by anticipating common errors [I differ in opinion]
- Global correction
  - Minimal changes for least-cost input correction
  - Mainly of theoretical interest
  - Useful to gauge efficacy of an error-recovery technique

Parsing and Context

- Most languages have keywords reserved.
- PL/I doesn't have reserved keywords.

```
if if = else then
   then = else
else
   then = if + else
```

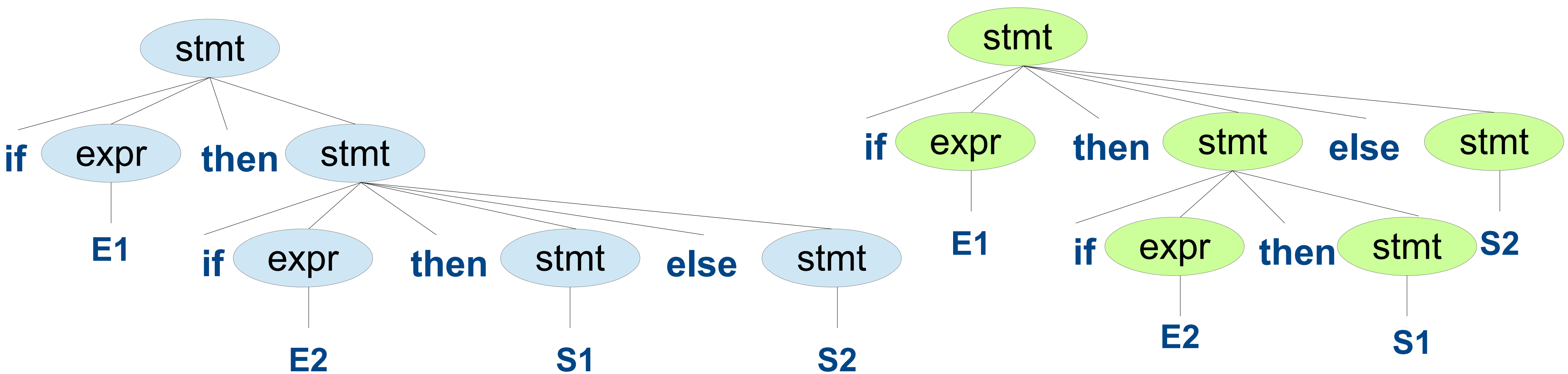
- Meaning is derived from the context in which a word is used.
- Needs support from lexer – it would return token IDENT for all words or IDENTKEYWORD.
- It is believed that PL/I syntax is notoriously difficult to parse.

if-else Ambiguity

stmt -> if expr then stmt
| if expr then stmt else stmt
| otherstmt

There are two parse trees for the following string

if E1 then if E2 then S1 else S2

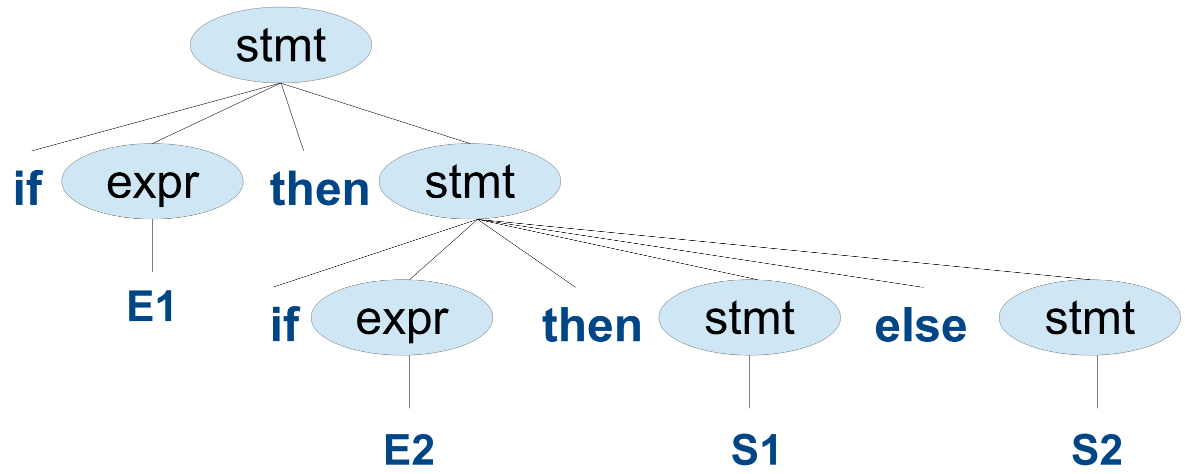


if-else Ambiguity

1. One way to resolve the ambiguity is to make yacc decide the precedence: *shift over reduce*.
   - Recall lex prioritizing longer match over shorter.
2. Second way is to change the grammar itself to not have any ambiguity.

```
stmt -> matched_stmt | open_stmt
matched_stmt -> if expr then matched_stmt else matched_stmt
              | otherstmt
open_stmt -> if expr then stmt
           | if expr then matched_stmt else open_stmt
```

if-else Ambiguity



if E1 then if E2 then S1 else S2

unambiguous

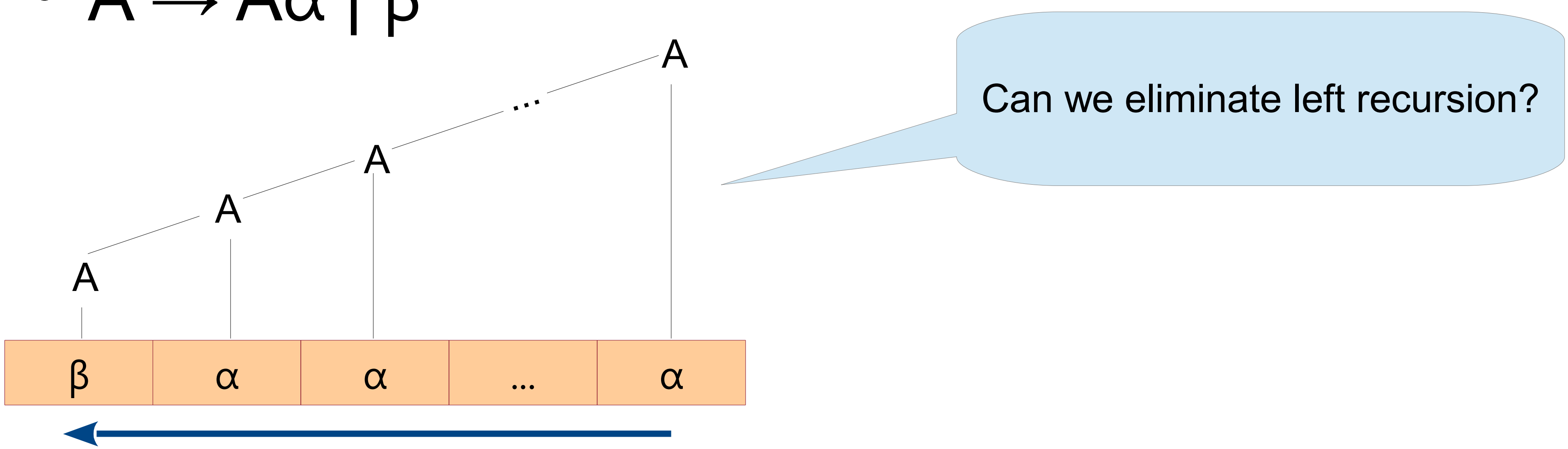
stmt -> matched_stmt | open_stmt
matched_stmt -> if expr then **matched_stmt** else matched_stmt
                              | otherstmt
open_stmt -> if expr then stmt
                       | if expr then **matched_stmt** else open_stmt

Classwork: Write an unambiguous grammar for associating *else* with the first *if*.

Left Recursion

A grammar is left-recursive if it has a non-terminal A such that there is a derivation $A \Rightarrow^+ A\alpha$ for some string α.

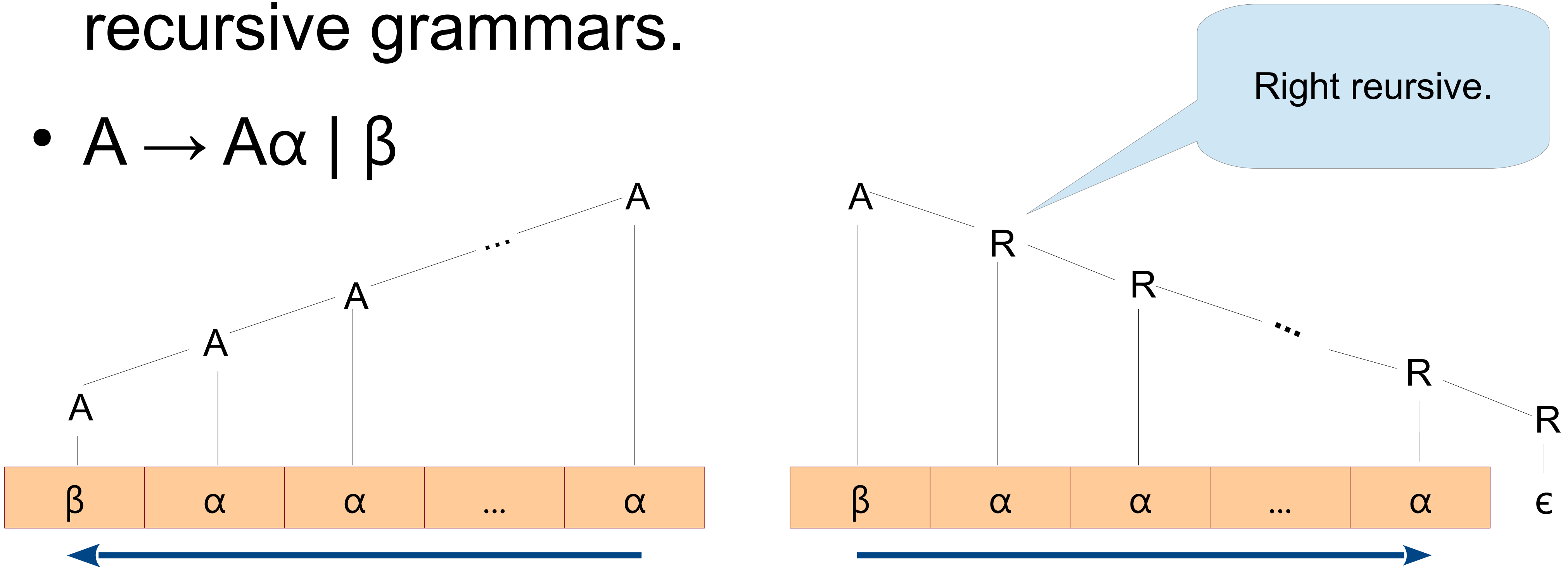
- Top-down parsing methods cannot handle left-recursive grammars.
- $A \to A\alpha \mid \beta$



Left Recursion

A grammar is left-recursive if it has a non-terminal A such that there is a derivation $A \Rightarrow^{+} A\alpha$ for some string α.

- Top-down parsing methods cannot handle left-recursive grammars.
- $A \rightarrow A\alpha \mid \beta$

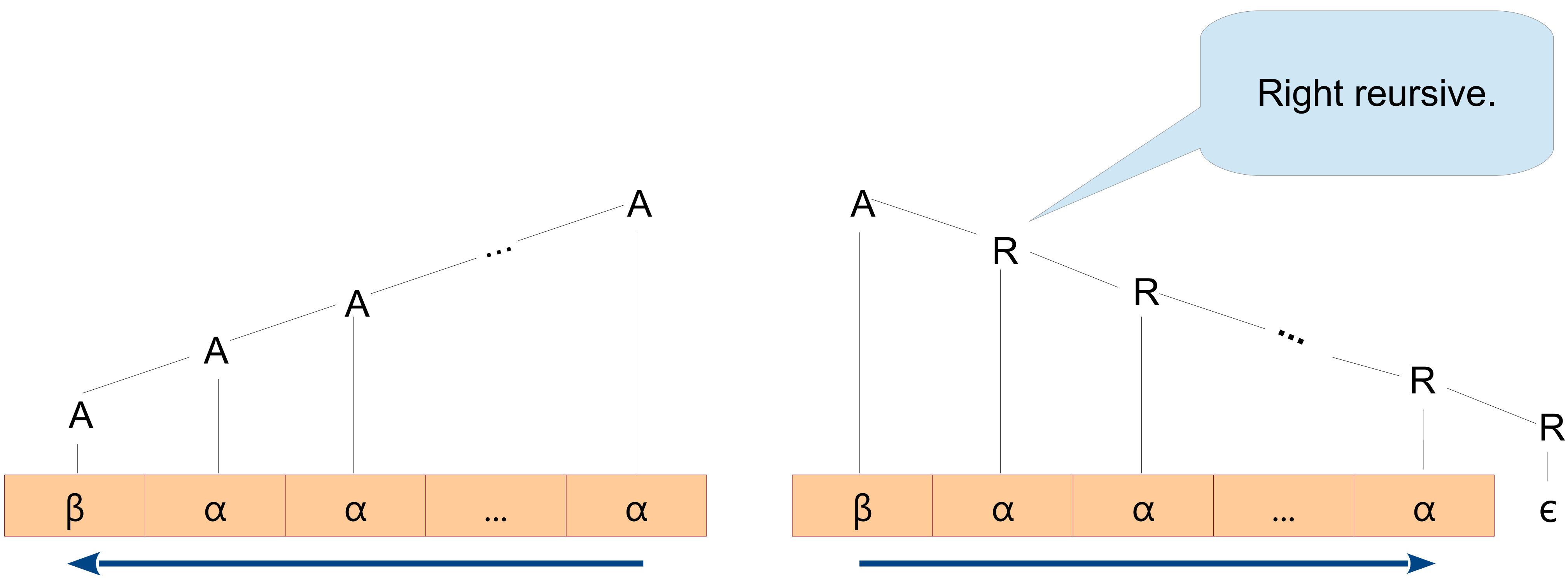


Left Recursion

$$A \longrightarrow A\alpha \mid \beta$$

$$A \longrightarrow \beta B$$

$$B \longrightarrow \alpha B \mid \epsilon$$



Left Recursion

$$A \longrightarrow A\alpha \mid \beta$$

$$A \longrightarrow \beta B$$
$$B \longrightarrow \alpha B \mid \epsilon$$

In general

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

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

Algorithm for Eliminating Left Recursion

arrange non-terminals in some order A1, ..., An.

for i = 1 to n {

for j = 1 to i -1 {

replace $A_i \longrightarrow A_j\alpha$ by $A_i \longrightarrow \beta_1\alpha \mid \ldots \mid \beta_k\alpha$

where $A_j \longrightarrow \alpha_1 \mid \ldots \mid \alpha_k$ are current A_j productions

}

eliminate immediate left recursion among Ai productions.

}

Classwork

- Remove left recursion from the following grammar.

$$E \longrightarrow E + T \mid T$$

$$T \longrightarrow T * F \mid F$$

$$F \longrightarrow (E) \mid \text{name} \mid \text{number}$$

$$E \longrightarrow T\,E'$$

$$E' \longrightarrow +\ T\,E' \mid \epsilon$$

$$T \longrightarrow F\,T'$$

$$T' \longrightarrow *\ F\,T' \mid \epsilon$$

$$F \longrightarrow (E) \mid \text{name} \mid \text{number}$$

Ambiguous / Unambiguous Grammars

Grammar for simple arithmetic expressions

$E \longrightarrow E + E \mid E * E \mid E - E \mid E / E \mid (E) \mid \text{number} \mid \text{name}$

Precedence not encoded
a + b * c

$E \longrightarrow E + E \mid E - E \mid T$
$T \longrightarrow T * T \mid T / T \mid F$
$F \longrightarrow (E) \mid \text{number} \mid \text{name}$

Associativity not encoded
a – b - c

$E \longrightarrow E + T \mid E - T \mid T$
$T \longrightarrow T * F \mid T / F \mid F$
$F \longrightarrow (E) \mid \text{number} \mid \text{name}$

Unambiguous grammar
Left recursive, not suitable for top-down parsing

$E \longrightarrow T\ E'$
$E' \longrightarrow +\ T\ E' \mid -\ T\ E' \mid \epsilon$
$T \longrightarrow F\ T'$
$T' \longrightarrow *\ F\ T' \mid /\ F\ T' \mid \epsilon$
$F \longrightarrow (E) \mid \text{number} \mid \text{name}$

Non-left-recursive grammar
Can be used for top-down parsing

Classwork

- Remove left recursion from the following grammar.

$$S \longrightarrow A\,a \mid b$$
$$A \longrightarrow A\,c \mid S\,d \mid \epsilon$$

$$S \longrightarrow A\,a \mid b$$
$$A \longrightarrow A\,c \mid A\,a\,d \mid b\,d \mid \epsilon$$

$$S \longrightarrow A\,a \mid b$$
$$A \longrightarrow b\,d\,A' \mid A'$$
$$A' \longrightarrow c\,A' \mid a\,d\,A' \mid \epsilon$$

Left Factoring

- When the choice between two alternative productions is unclear, rewrite the grammar to defer the decision until enough input is seen.
  - Useful for predictive or top-down parsing.
- $A \rightarrow \alpha\ \beta_1 \mid \alpha\ \beta_2$
  - Here, common prefix α can be left factored.
  - $A \rightarrow \alpha\ A'$
  - $A' \rightarrow \beta_1 \mid \beta_2$
- Left factoring doesn't change ambiguity. e.g. in dangling if-else.

Non-Context-Free Language Constructs

- *wcw* is an example of a language that is not CF.
- In the context of C, what does this language indicate?
- It indicates that declarations of variables (*w*) followed by arbitrary program text (*c*), and then use of the declared variable (*w*) cannot be specified in general by a CFG.
- Additional rules or passes (semantic phase) are required to identify declare-before-use cases.

What does the language $a^n b^m c^n d^m$ indicate in C?

Q1 Paper Discussion

- And attendance.
- And assignment marks.

Top-Down Parsing

- Constructs parse-tree for the input string, starting from root and creating nodes.
- Follows preorder (depth-first).
- Finds leftmost derivation.
- General method: recursive descent.
  - Backtracks
- Special case: Predictive (also called LL(k))
  - Does not backtrack
  - Fixed lookahead

Recursive Descent Parsing

```
void A() {
  saved = current input position;
  for each A-production A -> X1 X2 X3 ... Xk {
    for (i = 1 to k) {
      if (Xi is a nonterminal) call Xi();
      else if (Xi == next symbol) advance-input();
      else { yyless(); break; }
    }
    if (A matched) break;
    else current input position = saved;
  }
}
```

Nonterminal A

A-> BC | Aa | b

Terms in body

Term match

Term mismatch

Prod. match

Prod. mismatch

- Backtracking is rarely needed to parse PL constructs.
- Sometimes necessary in NLP, but is very inefficient. Tabular methods are used to avoid repeated input processing.

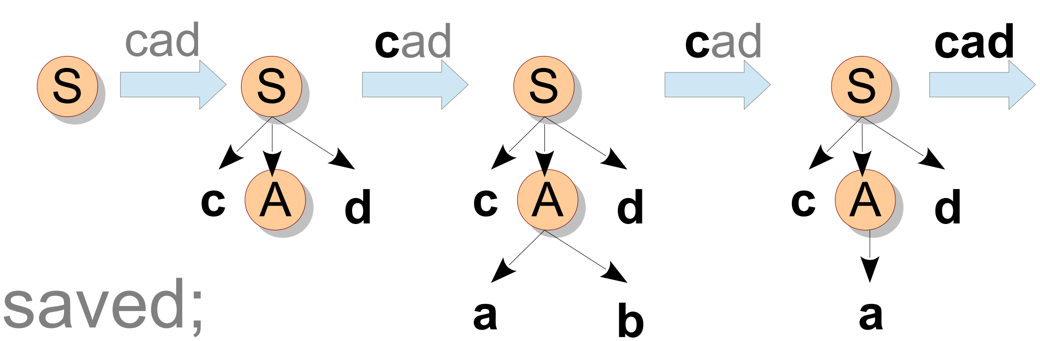
Recursive Descent Parsing

void A() {
  saved = current input position;
  for each A-production A -> X_1 X_2 X_3 ... X_k {
    for (i = 1 to k) {
      if (X_i is a nonterminal) call X_i();
      else if (X_i == next symbol) advance-input();
      else { yyless(); break; }
    }
    if (A matched) break;
    else current input position = saved;
  }
}

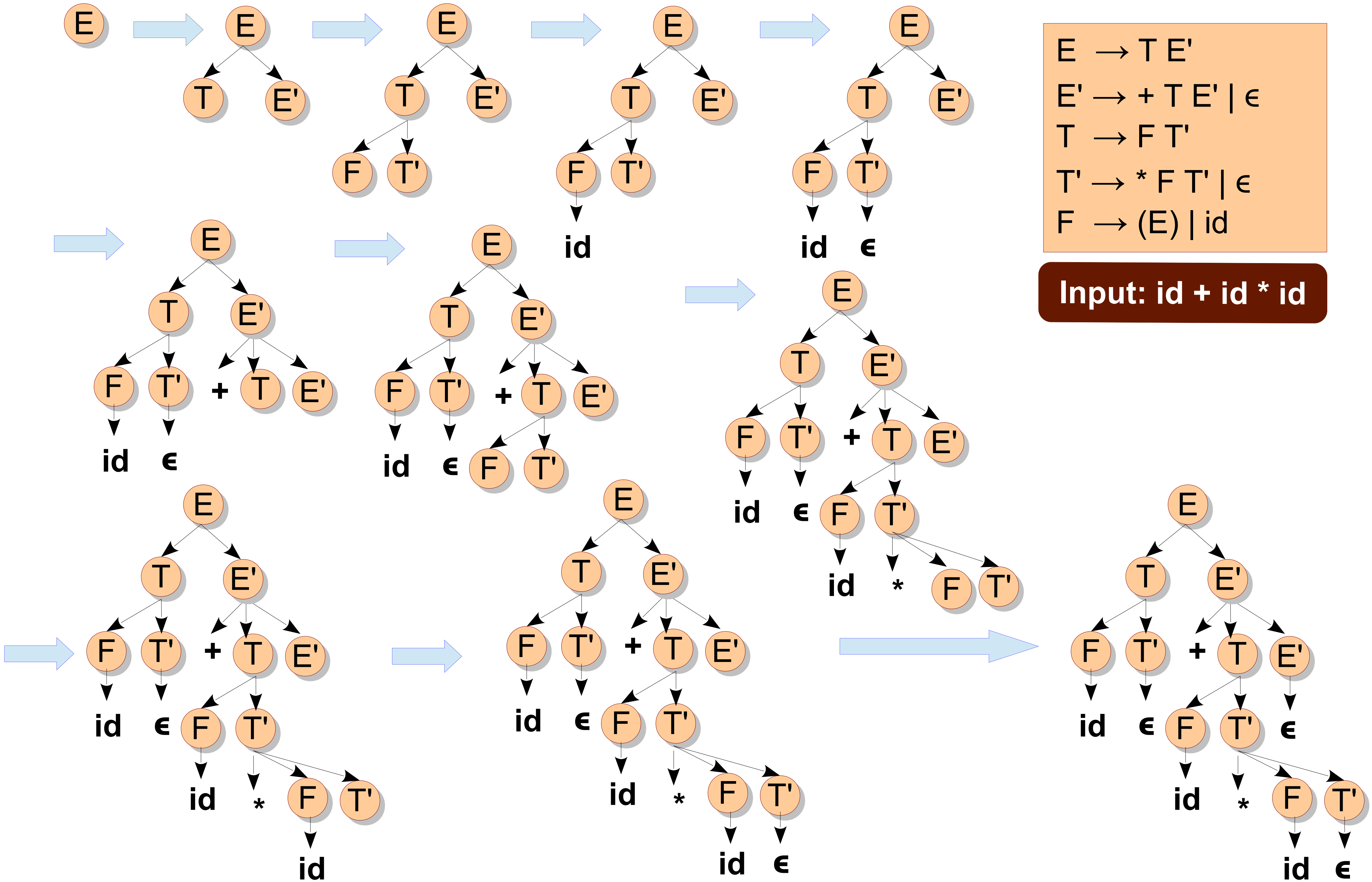
$S \longrightarrow c\ A\ d$

$A \longrightarrow a\ b \mid a$

Input string: cad



Classwork: Generate Parse Tree



E $\longrightarrow$ T E'

E' $\longrightarrow$ + T E' | ϵ

T $\longrightarrow$ F T'

T' $\longrightarrow$ * F T' | ϵ

F $\longrightarrow$ (E) | id

Input: id + id * id

FIRST and FOLLOW

- Top-down (as well as bottom-up) parsing is aided by FIRST and FOLLOW sets.
  - Recall firstpos, followpos from lexing.
- First and Follow allow a parser to choose which production to apply, based on lookahead.
- Follow can be used in error recovery.
  - While matching a production for $A \longrightarrow \alpha$, if the input doesn't match FIRST(α), use FOLLOW(A) as the synchronizing token.

FIRST and FOLLOW

- FIRST(**α**) is the set of terminals that begin strings derived from α, where α is any string of symbols
  - If α =>* ϵ, ϵ is also in FIRST(α)
  - If $A \longrightarrow \alpha \mid \beta$ and FIRST(α) and FIRST(β) are disjoint sets, then the lookahead decides the production to be applied.
- FOLLOW(**A**) is the set of terminals that can appear immediately to the right of A in some sentential form, where A is a nonterminal.
  - If S =>* $\alpha A a \beta$, then FOLLOW(A) contains a.
  - If S =>* $\alpha A B a \beta$ and B =>* ϵ then FOLLOW(A) contains a.
  - If A can be the rightmost symbol, we add $ to FOLLOW(A). This means FOLLOW(S) always contains $.

FIRST and FOLLOW

$E \rightarrow T\,E'$
$E' \rightarrow +\,T\,E' \mid \epsilon$
$T \rightarrow F\,T'$
$T' \rightarrow *\,F\,T' \mid \epsilon$
$F \rightarrow (E) \mid id$

- First(E) = {(, id}
- First(T) = {(, id}
- First(F) = {(, id}
- First(E') = {+, ϵ}
- First(T') = {*, ϵ}

- Follow(E) = {), $}
- Follow(T) = {+,), $}
- Follow(F) = {+, *,), $}
- Follow(E') = {), $}
- Follow(T') = {+,), $}

First and Follow

Non-terminal	FIRST	FOLLOW
E	(, id	), $
E'	+, ∈	), $
T	(, id	+,), $
T'	*, ∈	+,), $
F	(, id	+, *,), $

$E \longrightarrow T\ E'$
$E' \longrightarrow +\ T\ E' \mid \epsilon$
$T \longrightarrow F\ T'$
$T' \longrightarrow *\ F\ T' \mid \epsilon$
$F \longrightarrow (E) \mid id$

Predictive Parsing Table

Non-terminal	id	+	*	(	)	$
E						
E'						
T						
T'						
F						

Non-terminal	FIRST	FOLLOW
E	(, id	), $
E'	+, ∈	), $
T	(, id	+,), $
T'	*, ∈	+,), $
F	(, id	+, *,), $

$E \rightarrow T\ E'$
$E' \rightarrow +\ T\ E' \mid \epsilon$
$T \rightarrow F\ T'$
$T' \rightarrow *\ F\ T' \mid \epsilon$
$F \rightarrow (E) \mid id$

Predictive Parsing Table

Non-terminal	id	+	*	(	)	$
E	$E \longrightarrow T\ E'$			$E \longrightarrow T\ E'$		
E'		$E' \longrightarrow +TE'$			$E' \longrightarrow \epsilon$	$E' \longrightarrow \epsilon$
T	$T \longrightarrow F\ T'$			$T \longrightarrow F\ T'$		
T'		$T' \longrightarrow \epsilon$	$T' \longrightarrow *FT'$		$T' \longrightarrow \epsilon$	$T' \longrightarrow \epsilon$
F	$F \longrightarrow id$			$F \longrightarrow (E)$		

Non-terminal	FIRST	FOLLOW
E	(, id	), $
E'	+, ϵ	), $
T	(, id	+,), $
T'	*, ϵ	+,), $
F	(, id	+, *,), $

$E \rightarrow T\ E'$
$E' \rightarrow +\ T\ E' \mid \epsilon$
$T \rightarrow F\ T'$
$T' \rightarrow *\ F\ T' \mid \epsilon$
$F \rightarrow (E) \mid id$

Predictive Parsing Table

for each production $A \longrightarrow \alpha$

for each terminal a in FIRST(α)

Table[A][a].add($A \longrightarrow \alpha$)

Process terminals using FIRST

if ϵ is in FIRST(α) then

for each terminal b in FOLLOW(A)

Table[A][b].add($A \longrightarrow \alpha$)

Process terminals on nullable using FOLLOW

if $ is in FOLLOW(A) then

Table[A][$].add($A \longrightarrow \alpha$)

Process $ on nullable using FOLLOW

LL(1) Grammars

- Predictive parsers needing no backtracking can be constructed for LL(1) grammars.
  - First L is left-to-right input scanning.
  - Second L is leftmost derivation.
  - 1 is the maximum lookahead.
  - In general, LL(k) grammars.
  - LL(1) covers most programming constructs.
  - No left-recursive grammar can be LL(1).
  - No ambiguous grammar can be LL(1).

Any example of RR grammar?

LL(1) Grammars

- A grammar is LL(1) iff whenever $A \rightarrow \alpha \mid \beta$ are two distinct productions, the following hold:
  - FIRST(α) and FIRST(β) are disjoint sets.
  - If ϵ is in FIRST(β) then FIRST(α) and FOLLOW(A) are disjoint sets, and likewise if ϵ is in FIRST(α) FIRST(β) and FOLLOW(A) are disjoint sets.

Predictive Parsing Table

Non-terminal	id	+	*	(	)	$
E	$E \longrightarrow T\ E'$			$E \longrightarrow T\ E'$		
E'		$E' \longrightarrow +TE'$			$E' \longrightarrow \epsilon$	$E' \longrightarrow \epsilon$
T	$T \longrightarrow F\ T'$			$T \longrightarrow F\ T'$		
T'		$T' \longrightarrow \epsilon$	$T' \longrightarrow *FT'$		$T' \longrightarrow \epsilon$	$T' \longrightarrow \epsilon$
F	$F \longrightarrow id$			$F \longrightarrow (E)$		

- Each entry contains a single production.
- Empty entries correspond to **error states**.
- For LL(1) grammar, each entry uniquely identifies an entry or signals an **error**.
- If there are multiple productions in an entry, then *that grammar* is not LL(1). However, it does not guarantee that the language produced is not LL(1). We may be able to transform the grammar into an LL(1) grammar (by eliminating left-recursion and by left-factoring).
- There exist languages for which no LL(1) grammar exists.

Classwork: Parsing Table

Non-terminal	i	t	a	e	b	$
S	$S \longrightarrow i\ E\ t\ S\ S'$		$S \longrightarrow a$			
S'				$S' \longrightarrow e\ S$ $S' \longrightarrow \epsilon$		$S' \longrightarrow \epsilon$
E					$E \longrightarrow b$	

Non-terminal	FIRST	FOLLOW
S	i, a	e, $
S'	e, ∈	e, $
E	b	t

$S \rightarrow i\ E\ t\ S\ S' \mid a$
$S' \rightarrow eS \mid \epsilon$
$E \rightarrow b$

What is this grammar?

Need for Beautification

- Due to a human programmer, sometimes beautification is essential in the language (well, the language itself is due to a human).
  - e.g., it suffices for correct parsing not to provide an opening parenthesis, but it doesn't “look” good.

No opening parenthesis

```
for i = 0; i < 10; ++i)
    a[i+1] = a[i];
```

Example

```
forexpr: FOR expr; expr; expr ')'
                Block
          ;
```

YACC grammar

Homework

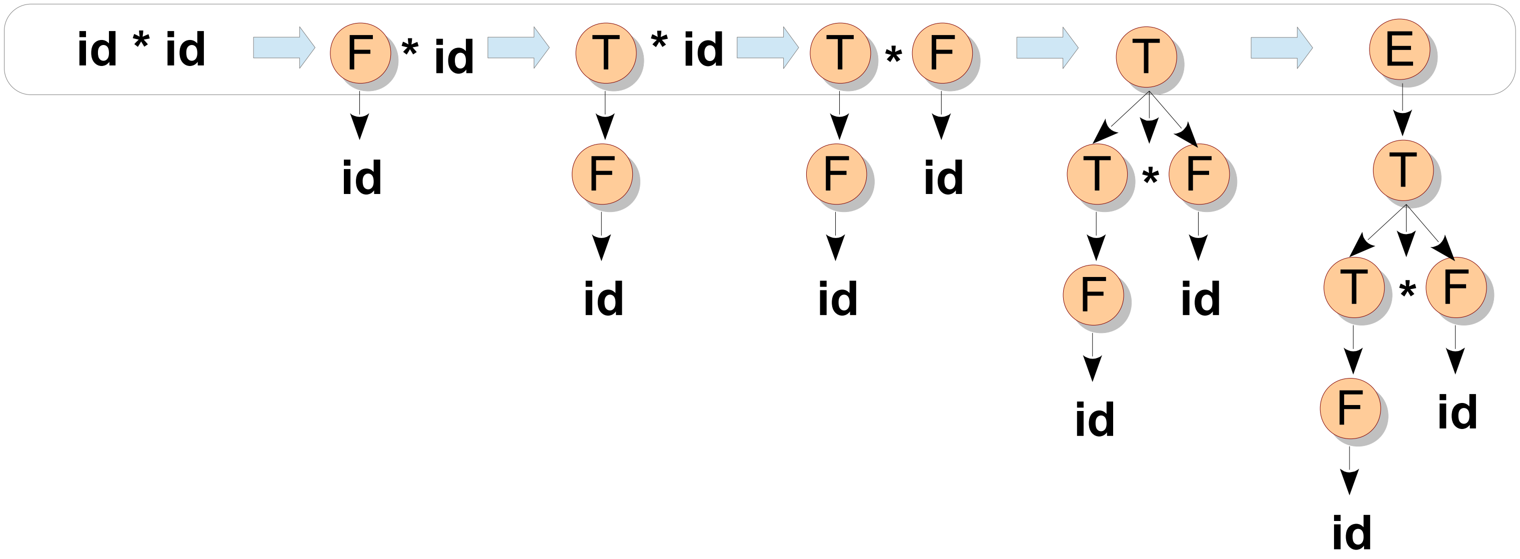
- Consider a finite domain (one..twenty), and four operators plus, minus, mult, div. Write a parser to parse the following.

```
for num1 in {one..twenty} {
    for num2 in {one..twenty} {
        for num3 in {one..twenty} {
            for num4 in {one..twenty} {
                for op1 in {plus, minus, mult, div} {
                    for op2 in {plus, minus, mult, div} {
                        if num1 op1 num2 == num3 op2 num4 {
                            print num1 op1 num2 "==" num3 op2 num4;
                        }
                    }
}   }   }   }   }
```

- Change the meaning of == from numeric equality to anagram / shuffle, and see the output.

Bottom-Up Parsing

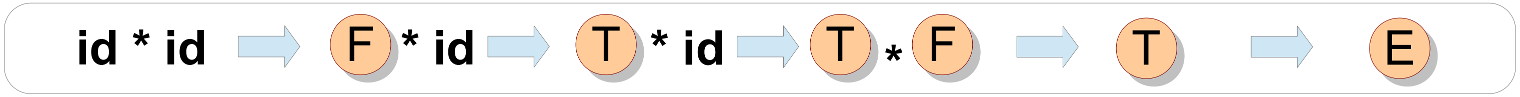
- Parse tree constructed bottom-up
  - In reality, an explicit tree may not be constructed.
  - It is also called a *reduction*.
  - At each reduction step, a specific substring matching the body of a production is replaced by the nonterminal at the head of the production.



Reduction sequence

Bottom-Up Parsing

- A reduction is the reverse of a derivation.
- Therefore, the goal of bottom-up parsing is to construct a derivation *in reverse*.



Bottom-Up Parsing

- A reduction is the reverse of a derivation.
- Therefore, the goal of bottom-up parsing is to construct a derivation *in reverse*.

id * id $\Leftarrow$ F *** id** $\Leftarrow$ T *** id** $\Leftarrow$ T ***** F $\Leftarrow$ T $\Leftarrow$ E

- This, in fact, is a rightmost derivation.
- Thus, scan the input from **L**eft, and construct a **R**ightmost derivation in reverse.

Handle Pruning

- A **handle** is a substring that matches the body of a production.
- Reduction of a handle represents one step in the reverse of a rightmost derivation.

id * id ⇐ F *** id** ⇐ T *** id** ⇐ T * F ⇐ T ⇐ E

Right Sentential Form	Handle	Reducing Production
$id_1 * id_2$	id_1	F -> id
$F * id_2$	F	T -> F
$T * id_2$	id_2	F -> id
T * F	T * F	T -> T * F
T	T	E -> T

We say ***a*** handle rather than ***the*** handle because ...

... the grammar could be ambiguous.

Shift-Reduce Parsing

- Type of bottom-up parsing
- Uses a stack (to hold grammar symbols)
- Handle appears at the stack top prior to pruning.

Stack	Input	Action
$	$id_1 * id_2$ $	shift
$ id_1	$* id_2$ $	reduce by F -> id
$ F	$* id_2$ $	rduce by T -> F
$ T	$* id_2$ $	shift
$ T *	id_2 $	shift
$ T * id_2	$	reduce by F -> id
$ T * F	$	reduce by T -> T * F
$ T	$	reduce by E -> T
$ E	$	accept

Shift-Reduce Parsing

- Type of bottom-up parsing
- Uses a stack (to hold grammar symbols)
- Handle appears at the stack top prior to pruning.

1. **Initially**, stack is empty ($...) and string w is on the input (w $).
2. During left-to-right input scan, the parser **shifts** zero or more input symbols on the stack.
3. The parser **reduces** a string to the head of a production (handle pruning)
4. This cycle is **repeated** until error or accept (stack contains start symbol and input is empty).

Conflicts

- There exist CFGs for which shift-reduce parsing cannot be used.
- Even with the knowledge of the whole stack (not only the stack top) and k lookahead
  - The parser doesn't know whether to shift (be lazy) or reduce (be eager) *(shift-reduce conflict)*.
  - The parser doesn't know which of the several reductions to make *(reduce-reduce conflict)*.

Shift-Reduce Conflict

- **Stack:** $ ... if expr then stmt
- **Input:** else ... $
  - Depending upon what the programmer intended, it may be correct to *reduce* if expr then stmt to stmt, or it may be correct to *shift* else.
  - One may direct the parser to prioritize shift over reduce (recall longest match rule of lex).
  - Shift-Reduce conflict is often not a show-stopper.

Reduce-Reduce Conflict

- **Stack:** $ ... id (id
- **Input:** , id) ... $
  - Consider a language where arrays are accessed as $\mathrm{arr}(\mathrm{i},\ \mathrm{j})$ and functions are invoked as $\mathrm{fun}(\mathrm{a},\ \mathrm{b})$.
  - Lexer may return id for both the array and the function.
  - Thus, by looking at the stack top and the input, a parser cannot deduce whether to reduce the handle as an array expression or a function call.
  - Parser needs to consult the symbol table to deduce the type of id (semantic analysis).
  - Alternatively, lexer may consult the symbol table and may return different tokens (array and function).

Ambiguity

The one above

Apni to har aah ek tufaan hai
Uparwala jaan kar anjaan hai...

LR Parsing

- **L**eft-to-right scanning, **R**ightmost derivation in reverse.
- Type of bottom-up parsers.
  - SLR (Simple LR)
  - CLR (Canonical LR)
  - LALR (LookAhead LR)
- LR(k) for k symbol lookahead.
  - k = 0 and k = 1 are of practical interest.
- Most prevalent in use today.

Why LR?

- LR > LL
- Recognizes almost all programming language constructs (structure, not semantics).
- Most general non-backtracking shift-reduce parsing method known.

Simple LR (SLR)

- We saw that a shift-reduce parser looks at the stack and the next input symbol to decide the action.
- But how does it know whether to shift or reduce?
  - In LL, we had a nice parsing table; and we knew what action to take based on it.
- For instance, if stack contains $\$ \mathrm{T}$ and the next input symbol is $*$, should it shift (anticipating $\mathrm{T} * \mathrm{F}$) or reduce ($\mathrm{E} \rightarrow \mathrm{T}$)?
- The goal, thus, is to build a parsing table similar to LL.

Items and Itemsets

- An LR parser makes shift-reduce decisions by maintaining **states** to keep track of *where we are in a parse*.
- For instance, $A \rightarrow XYZ$ may represent a state:

  1. $A \rightarrow \cdot XYZ$
  2. $A \rightarrow X \cdot YZ$
  3. $A \rightarrow XY \cdot Z$
  4. $A \rightarrow XYZ \cdot$

  LR(0) Item

  Itemset == state

- $A \rightarrow \epsilon$ generates a single item $A \rightarrow \cdot$
- An item indicates how much of a production the parser has seen so far.

LR(0) Automaton

1. Find sets of LR(0) items.
2. Build canonical LR(0) collection.
   - Grammar augmentation (start symbol)
   - CLOSURE (similar in concept to ϵ-closure in FA)
   - GOTO (similar to state transitions in FA)
3. Construct the FA

$E \longrightarrow E + T \mid T$
$T \longrightarrow T * F \mid F$
$F \longrightarrow (E) \mid id$

I_0 — Initial state

$E' \rightarrow .\ E$ — Kernel item

$E \rightarrow .\ E + T$

$E \rightarrow .\ T$

$T \rightarrow .\ T * F$

$T \rightarrow .\ F$

$F \rightarrow .\ (E)$

$F \rightarrow .\ id$

Non-kernel items

Item closure

Classwork:

Find closure set for $T \rightarrow T * .\ F$

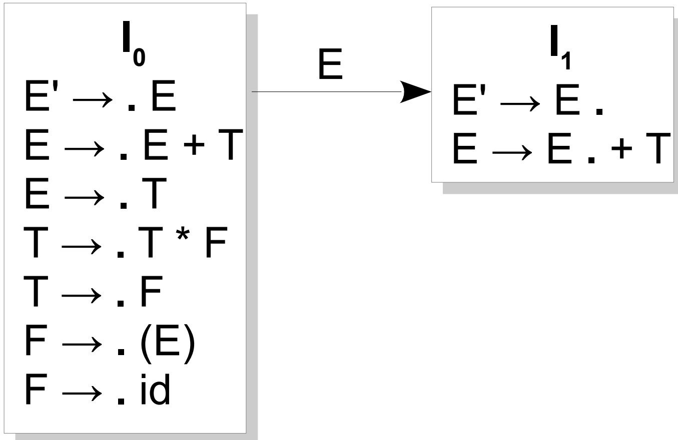
Find closure set for $F \rightarrow (E) .$

$E' \rightarrow E$
$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid id$

LR(0) Automaton

1. Find sets of LR(0) items.
2. Build canonical LR(0) collection.
   - ✓ Grammar augmentation (start symbol)
   - ✓ CLOSURE (similar in concept to ϵ-closure in FA)
   - GOTO (similar to state transitions in FA)
3. Construct the FA

$E' \longrightarrow E$
$E \longrightarrow E + T \mid T$
$T \longrightarrow T * F \mid F$
$F \longrightarrow (E) \mid id$



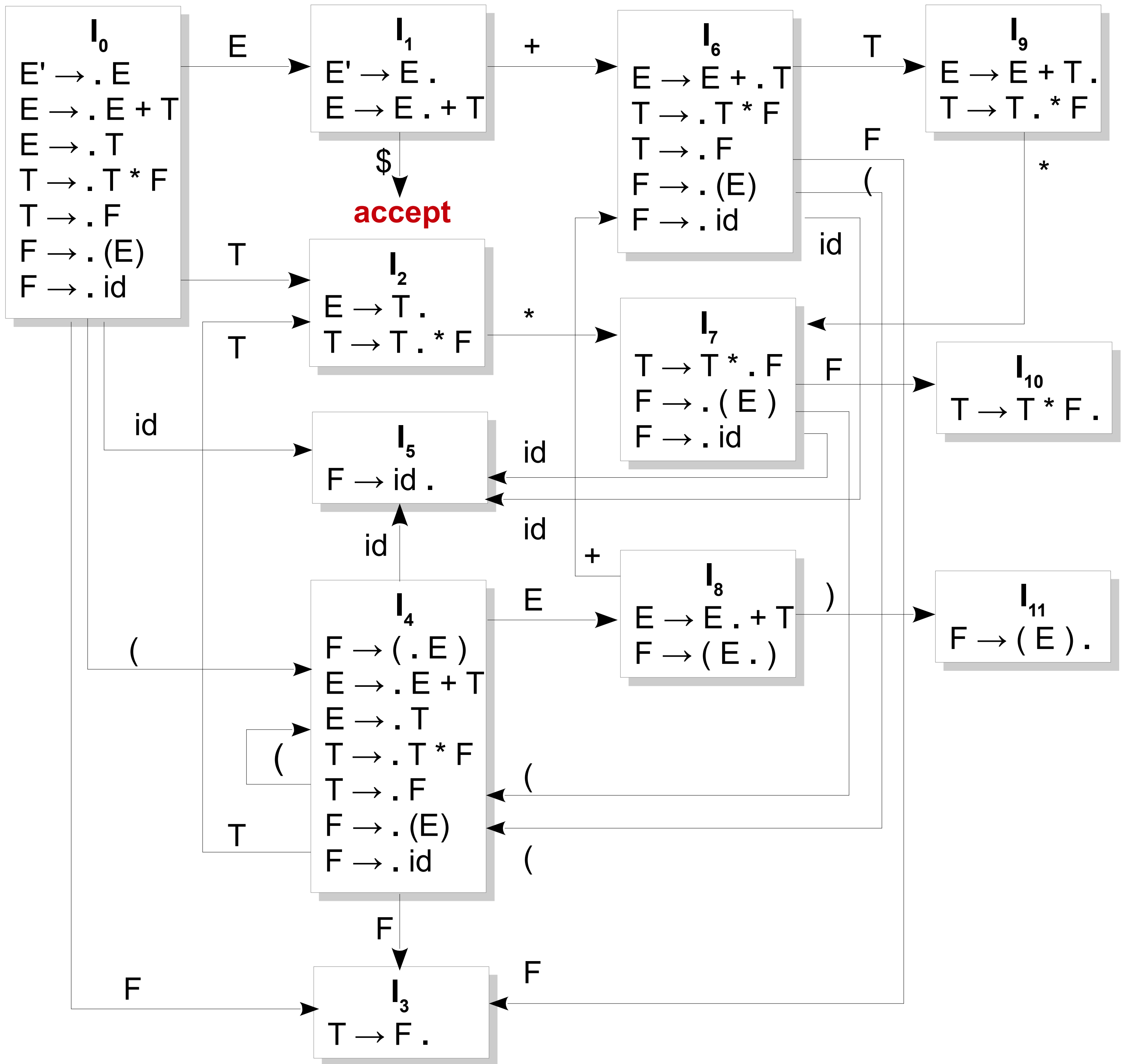
GOTO(I, X) is the closure of the set of items $[A \rightarrow \alpha\ X\ .\ \beta]$ such that $[A \rightarrow \alpha\ .\ X\ \beta]$ is in I.

- For instance, GOTO(I_0, E) is {E' → E ., E → E . + T}.

Classwork:

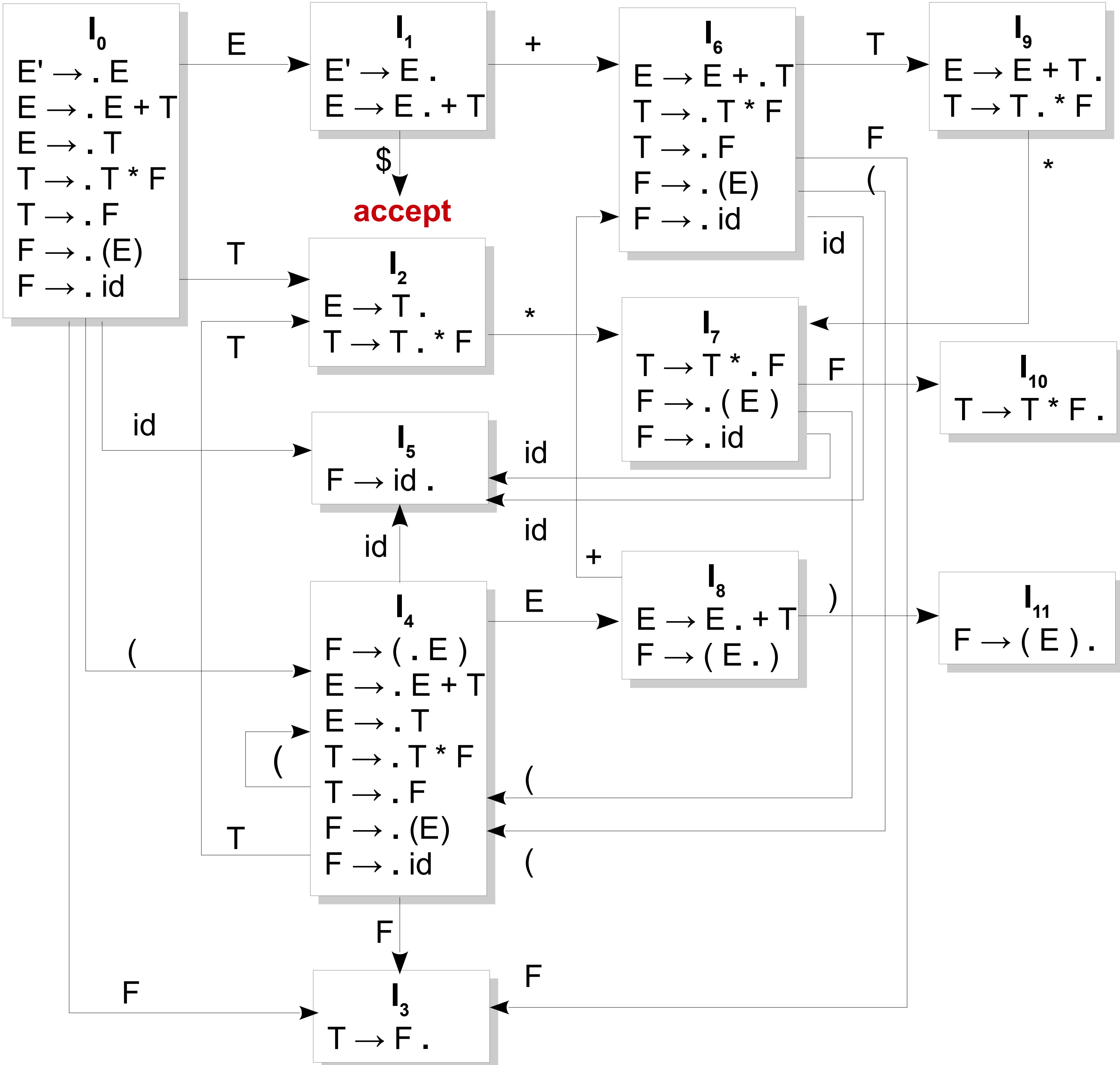
- Find GOTO(I, +) where I contains {E' → E ., E → E . + T}

E' → E
E → E + T | T
T → T * F | F
F → (E) | id



LR(0) Automaton

E' → E
E → E + T | T
T → T * F | F
F → (E) | id



- **Initially**, the state is 0 (for I_0).
- On seeing input symbol **id**, the state changes to 5 (for I_5).
- On seeing input *****, there is no action out of state 5.

E' → E
E → E + T | T
T → T * F | F
F → (E) | id

SLR Parsing using Automaton

Contains states like I_0, I_1, ...

Sr No	Stack	Symbols	Input	Action
1	0	$	id * id $	Shift to 5
2	0 5	$ id	* id $	Reduce by F -> id
3	0 3	$ F	* id $	Reduce by T -> F
4	0 2	$ T	* id $	Shift to 7
5	0 2 7	$ T *	id $	Shift to 5
6	0 2 7 5	$ T * id	$	Reduce by F -> id
7	0 2 7 10	$ T * F	$	Reduce by T -> T * F
8	0 2	$ T	$	Reduce by E -> T
9	0 1	$ E	$	**Accept**

Homework: Construct such a table for parsing **id * id + id**.

$E' \rightarrow E$
$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid id$

SLR(1) Parsing Table

State	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				accept			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

E' ⟶ E
E ⟶ E + T | T
T ⟶ T * F | F
F ⟶ (E) | id

LR Parsing

```
let a be the first symbol of w$
push 0 state on stack
while (true) {
    let s be the state on top of the stack
    if ACTION[s, a] == shift t {
        push t onto the stack
        let a be the next input symbol
    } else if ACTION[s, a] == reduce A → β {
        pop |β| symbols off the stack
        let state t now be on top of the stack
        push GOTO[t, a] onto the stack
        output the production A → β
    } else if ACTION[s, a] == accept { break }
    else yyerror()
}
```

Classwork

- Construct LR(0) automaton and SLR(1) parsing table for the following grammar.

$S \longrightarrow A\ S \mid b$
$A \longrightarrow S\ A \mid a$

- Run it on string *abab*.

SLR(1) Parsing Table

State	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				accept			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Why do we not have a transition out of state 5 on (?

$E' \rightarrow E$
$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid id$

Reduce Entries in the Parsing Table

- Columns for reduce entries are lookaheads.
- Therefore, they need to be in the FOLLOW of the head of the production.
- Thus, A -> α**.** is the production to be applied (that is, α is being reduced to A), then the lookahead (next input symbol) should be in FOLLOW(A).

I_5
$F \rightarrow id\ .$

Reduction F -> id should be applied only if the next input symbol is FOLLOW(F) which is {+, *,), $}.

State	id	+	*	(	)	$	E	T	F
5		r6	r6		r6	r6			

l-values and r-values

$S \longrightarrow L = R \mid R$
$L \longrightarrow {}^{*}R \mid id$
$R \longrightarrow L$

l-values and r-values

I_0
S' → . S
S → . L = R
S → . R
L → . * R
L → . id
R → . L

I_1
S' → S .

I_2
S → L . = R
R → L .

I_3
S → R .

I_4
L → * . R
R → . L
L → . * R
L → . id

I_5
L → id .

I_6
S → L = . R
R → . L
L → . * R
L → . id

I_7
L → * R .

I_8
R → L .

I_9
S → L = R .

Consider state **I_2**.

- Due to the first item (S → L **.** = R), ACTION[2, =] is *shift 6*.
- Due to the second item (R → L **.**), and because FOLLOW(R) contains =, ACTION[2, =] is *reduce R → L.*.

Thus, there is a shift-reduce conflict.
Does that mean the grammar is ambiguous?
Not necessarily; in this case no.
However, our SLR parser is not able to handle it.

S'→ S
S → L = R | R
L → *R | id
R → L

LR(0) Automaton and Shift-Reduce Parsing

- Why can LR(0) automaton be used to make shift-reduce decisions?
- LR(0) automaton characterizes the strings of grammar symbols that can appear on the stack of a shift-reduce parser.
- The stack contents must be a prefix of a right-sentential form *[but not all prefixes are valid]*.
- If stack hold β and the rest of the input is x, then a sequence of reductions will take βx to S. Thus, $S \Rightarrow^* \beta x$.

Viable Prefixes

- Example
  - *E =>* F * id => (E) * id*
  - At various times during the parse, the stack holds (, (E and (E).
  - However, it must not hold (E)*. Why?
  - Because (E) is a handle, which must be reduced.
  - Thus, (E) is reduced to F before shifting *.
- Thus, not all prefixes of right-sentential forms can appear on the stack.
- Only those that can appear are viable.

Viable Prefixes

- SLR parsing is based on the fact that LR(0) automata recognize viable prefixes.
- Item $A \rightarrow \beta_1 . \beta_2$ is valid for a viable prefix $\alpha\beta_1$ if there is a derivation $S \Rightarrow^* \alpha A w \Rightarrow \alpha\beta_1\beta_2 w$.
- Thus, when $\alpha\beta_1$ is on the parsing stack, it suggests we have not yet shifted the handle – so shift (not reduce).
  - Assuming $\beta_2 \rightarrow \epsilon$.

Homework

- Exercises in Section 4.6.6.

LR(1) Parsing

- Lookahead of 1 symbol.
- We will use similar construction (automaton), but with lookahead.
- This should increase the power of the parser.

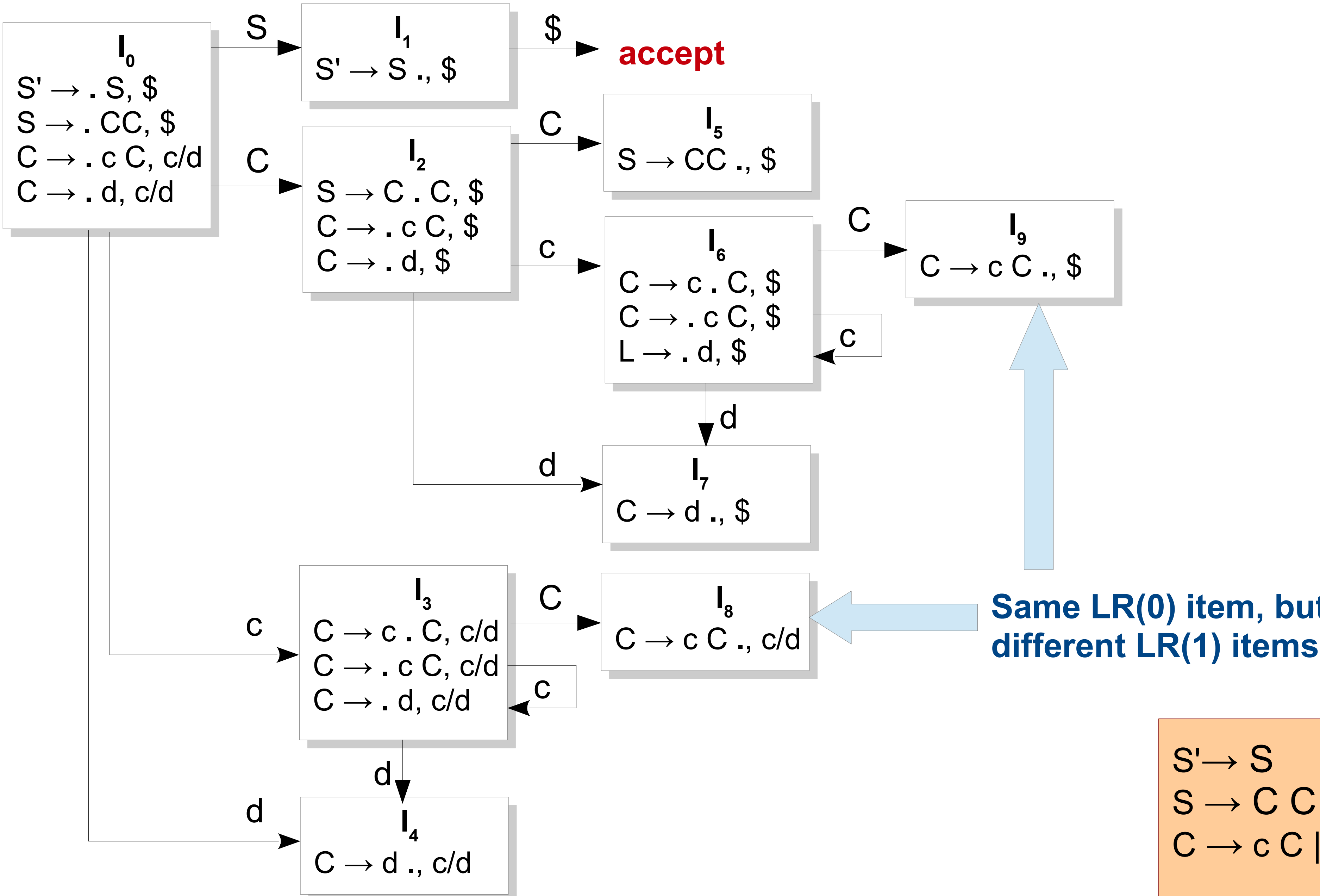
$S' \rightarrow S$
$S \rightarrow L = R \mid R$
$L \rightarrow *R \mid id$
$R \rightarrow L$

LR(1) Parsing

- Lookahead of 1 symbol.
- We will use similar construction (automaton), but with lookahead.
- This should increase the power of the parser.

$S' \longrightarrow S$
$S \longrightarrow C\ C$
$C \longrightarrow c\ C \mid d$

LR(1) Automaton



S'→ S
S → C C
C → c C | d

LR(1) Grammars

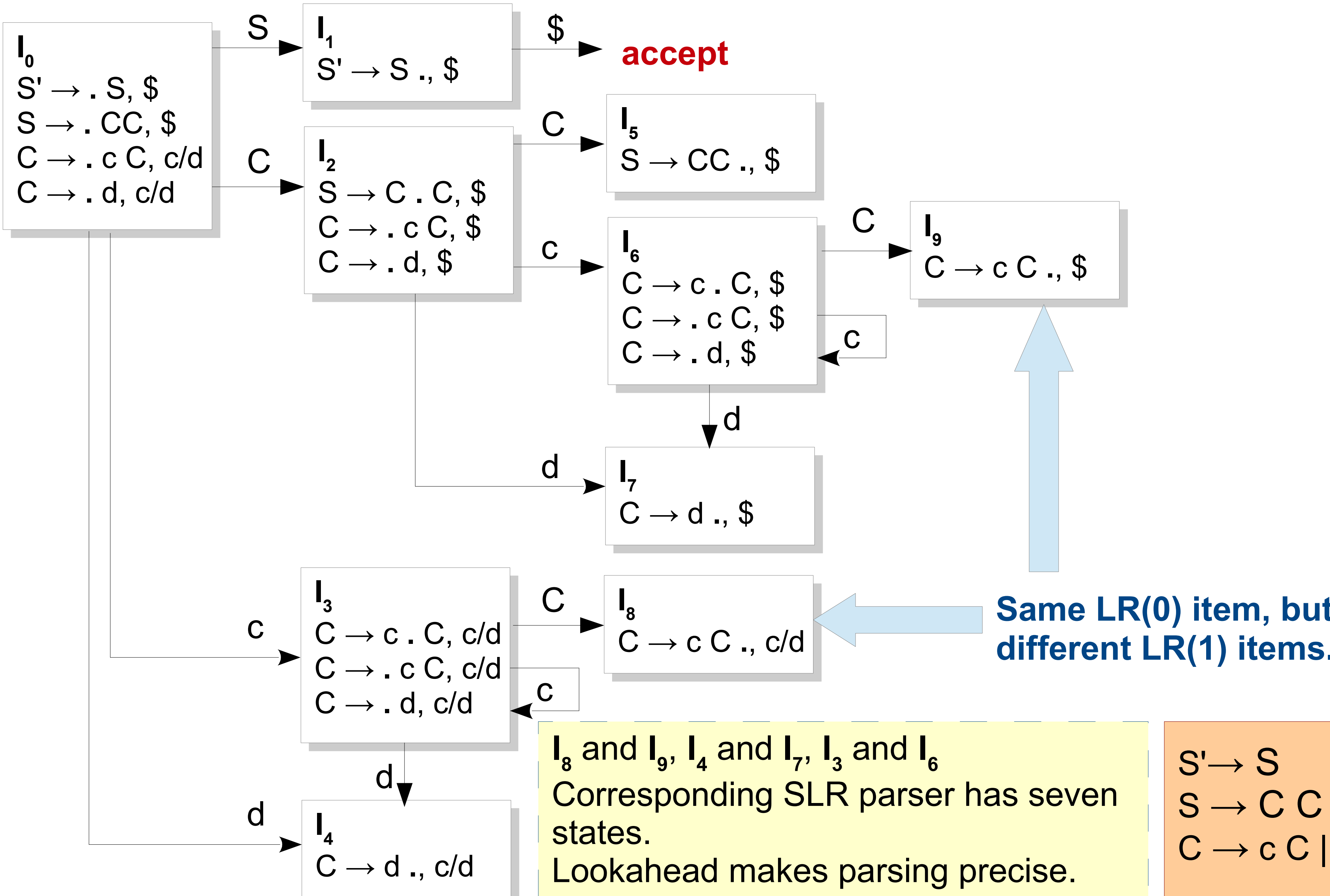
- Using LR(1) items and GOTO functions, we can build canonical LR(1) parsing table.
- An LR parser using the parsing table is canonical-LR(1) parser.
- If the parsing table does not have multiple actions in any entry, then the given grammar is LR(1) grammar.
- Every SLR(1) grammar is also LR(1).
  - SLR(1) < LR(1)
  - Corresponding CLR parser may have more states.

CLR(1) Parsing Table

State	c	d	$	S	C
0	s3	s4		1	2
1			accept		
2	s6	s7			5
3	s3	s4			8
4	**r3**	**r3**			
5			**r1**		
6	s6	s7			9
7			**r3**		
8	**r2**	**r2**			
9			**r2**		

S' → S
S → C C
C → c C | d

LR(1) Automaton



I_8 and I_9, I_4 and I_7, I_3 and I_6
Corresponding SLR parser has seven states.
Lookahead makes parsing precise.

S' → S
S → C C
C → c C | d

LALR Parsing

- Can we have memory efficiency of SLR and precision of LR(1)?
- For C, SLR would have a few hundred states.
- For C, LR(1) would have a few thousand states.
- How about merging states with same LR(0) items?
- Knuth invented LR in 1965, but it was considered impractical due to memory requirements.
- Frank DeRemer invented SLR and LALR in 1969 (LALR as part of his PhD thesis).
- YACC generates LALR parser.

State	c	d	$	S	C
0	s3	s4		1	2
1			accept		
2	s6	s7			5
3	s3	s4			8
4	**r3**	**r3**			
5			**r1**		
6	s6	s7			9
7			**r3**		
8	**r2**	**r2**			
9			**r2**		

CLR(1) Parsing Table

I_8 and I_9, I_4 and I_7, I_3 and I_6
Corresponding SLR parser has seven states.
Lookahead makes parsing precise.

- LALR parser mimics LR parser on correct inputs.
- On erroneous inputs, LALR may proceed with reductions while LR has declared an error.
- However, eventually, LALR is guaranteed to give the error.

LALR(1) Parsing Table

State	c	d	$	S	C
0	s36	s47		1	2
1			accept		
2	s36	s47			5
36	s36	s47			89
47	**r3**	**r3**	**r3**		
5			**r1**		
89	**r2**	**r2**	**r2**		

S' → S
S → C C
C → c C | d

State Merging in LALR

- State merging with common kernel items does not produce shift-reduce conflicts.
- A merge may produce a reduce-reduce conflict.

$S' \rightarrow S$
$S \rightarrow a\ A\ d\ |\ b\ B\ d\ |\ a\ B\ e\ |\ b\ A\ e$
$A \rightarrow c$
$B \rightarrow c$

$A \rightarrow c.,\ d/e$
$B \rightarrow c.,\ d/e$

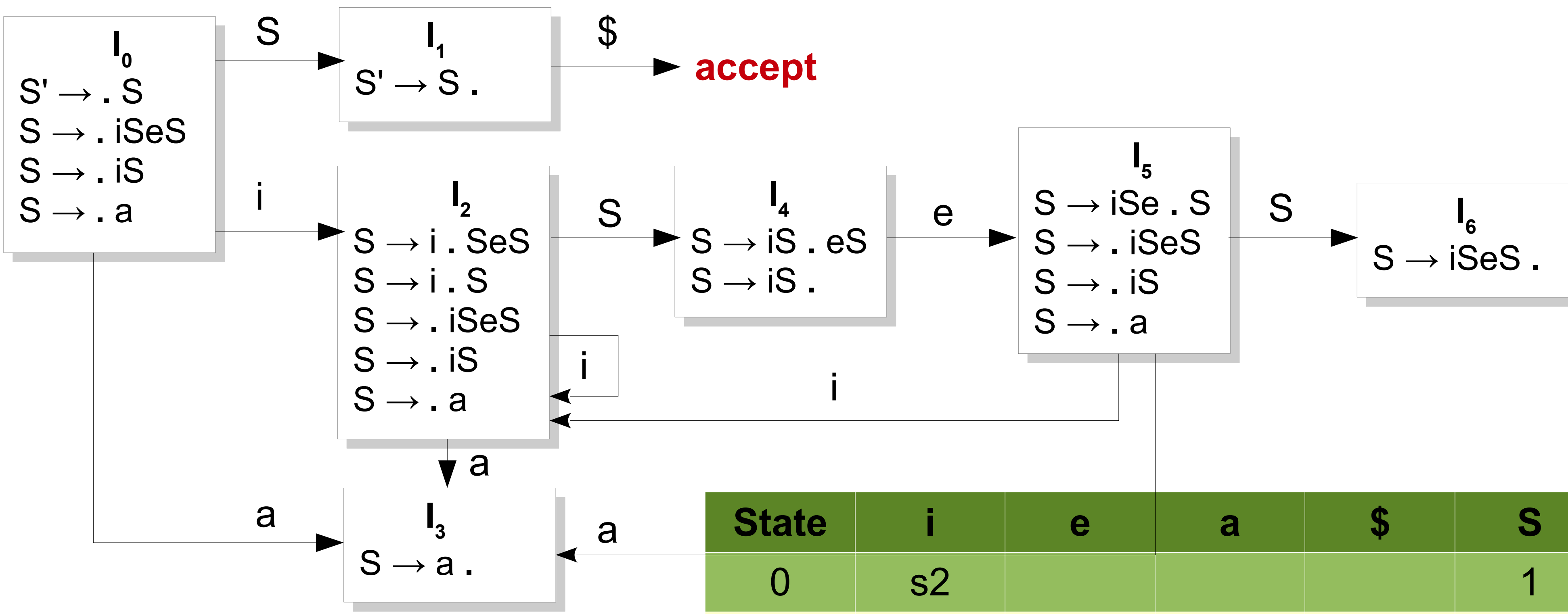
- This grammar is LR(1).
- Itemset {[A → c., d], [B → c., e]} is valid for viable prefix ac (due to *acd* and *ace*).
- Itemset {[A → c., e], [B → c., d]} is valid for viable prefix bc (due to *bcd* and *bce*).
- Neither of these states has a conflict. Their kernel items are the same.
- Their union / merge generates reduce-reduce conflict.

Using Ambiguous Grammars

- Ambiguous grammars should be sparingly used.
- They can sometimes be more natural to specify (e.g., expressions).
- Additional rules may be specified to resolve ambiguity.

$S' \rightarrow S$
$S \rightarrow i\ S\ e\ S \mid i\ S \mid a$

Using Ambiguous Grammars



State	i	e	a	$	S
0	s2				1
1			s3	accept	
2	s2		s3		4
3		r3		r3	
4		**s5/r2**		r2	
5	s2		s3		6
6		r1		r1	

S'→ S
S → i S e S | i S | a

Summary

- Precedence / Associativity
- Parse Trees
- Left Recursion
- Left factoring
- Top-Down Parsing
- LL(1) Grammars
- Bottom-Up Parsing
- Shift-Reduce Parsers
- LR(0), SLR
- LR(1), LALR