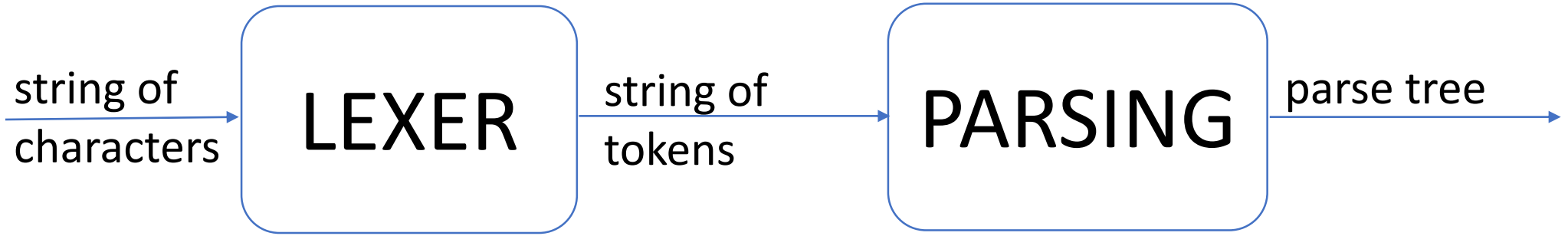
ĐẠI HỌC ĐÀ NẴNG
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG VIỆT - HÀN
Vietnam - Korea University of Information and Communication Technology

CHAPTER 3

SYNTAX ANALYSIS

Introduction to Parsing

- Relationship between lexical analysis and parsing.



- Regular languages
  - The weakest formal languages widely used
  - Many applications

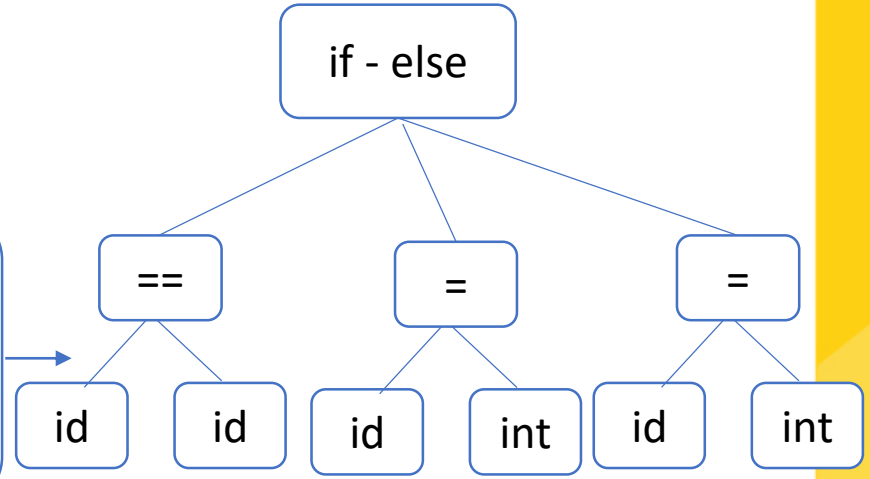
Introduction to Parsing

```
if (x==y)
   result = 1;
else
   result = 2;
```

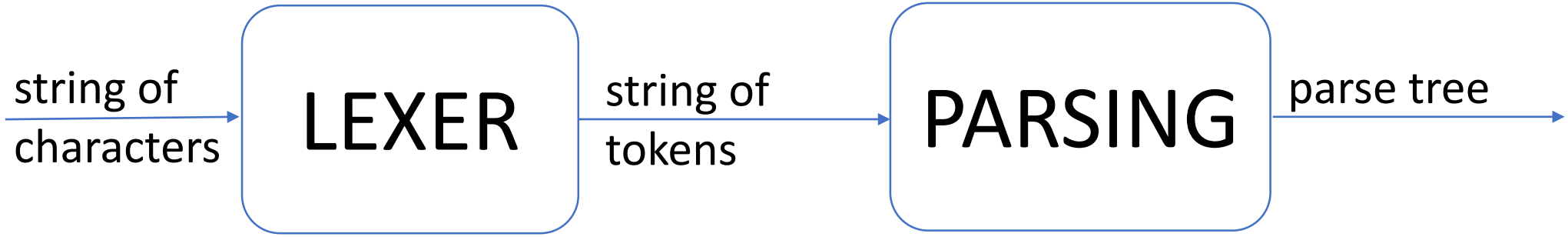
LEXER

<key, ‘if’>
<(, ‘(‘>
<id, ‘x’>
<==, ‘==‘>
<id, ‘y’>
<), ‘)’
<id, ‘result’>
<=, ‘=‘>
<int, ‘1’>
<pun, ‘;’>
<key, ‘else’>
<id, ‘result’>
<=, ‘=‘>
<int, ‘2’>
<pun, ‘;’>

PARSING



Introduction to Parsing



- Sometimes the parse tree is implicit. A compiler may never actually build the full parse tree.
- There are compilers that do combine lexer and parsing phases into one where everything is done by the parser.

Introduction to Parsing

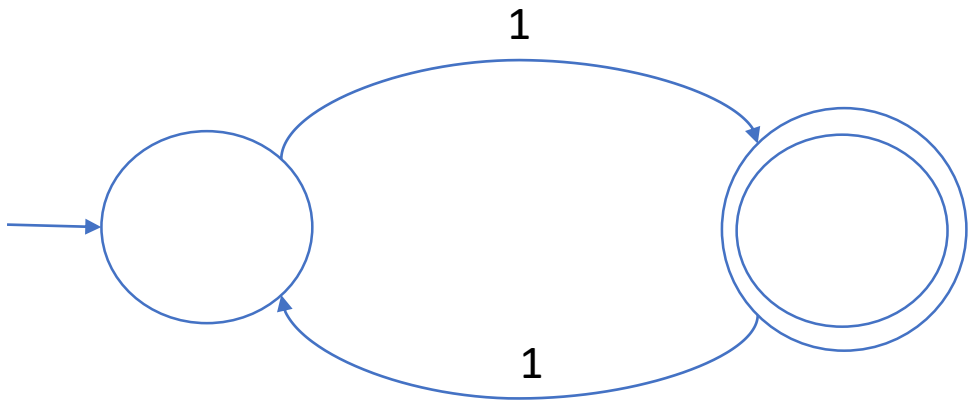
- The difficult with regular languages:
  - A lot of languages are simply not regular.
  - There's some pretty important languages that can't be expressed using regular expression.

- Consider the language:

$$\{(^n)^n \mid n >= 0\}$$

Introduction to Parsing

- What can regular languages can express?
- Why they aren’t sufficient for recognizing arbitrary nesting structure?
- Look a simple two state machine



Context-Free Grammars (CFGs)

- Not all strings of token are valid.
- The parser has to know which strings of tokens are valid and which ones are invalid and give error messages for the invalid ones.

⇒We need:

- some way of describing the valid strings of tokens
- some kind of algorithm for distinguishing the valid and invalid string of tokens from each other.

Context-Free Grammars (CFGs)

- Programming languages have recursive structure
- A statement (stmt) is

  if (expr) stmt else stmt

  while (expr) stmt

  for (expr; expr; expr) stmt

  ……

- Context-free grammars are a natural notation for this recursive structure

Context-Free Grammars (CFGs)

- A CFG consist of
  - A set of terminals **T**
  - A set of non-terminals **N**
  - A start symbol **S (S∈N)**
  - A set of productions (rules) **R**

$$A \rightarrow B_1 B_2 \ldots . B_n,$$
$$A \in N$$
$$B_i \in N \cup T \cup \{\varepsilon\}$$

means A can be replaced by $B_1 B_2 \ldots . B_n$

Context-Free Grammars (CFGs)

Key idea

1. Begin with a string consisting of the start symbol "S"
2. Replace any non-terminal A in the string by a the right-hand side of some production

$$A \rightarrow B_1 \ldots B_n$$

3. Repeat (2) until there are no non-terminals in the string

Context-Free Grammars (CFGs)

Notational conventions for grammars

1. These symbols are terminals:
   (a) Lowercase letters early in the alphabet, such as a, b, c.
   (b) Operator symbols such as +, *, and so on.
   (c) Punctuation symbols such as parentheses, comma, and so on.
   (d) The digits 0, 1, ….., 9.
   (e) Boldface strings such as **id** or **if**, each of which represents a single terminal symbol.

2. These symbols are nonterminals:
   (a) Uppercase letters early in the alphabet, such as A, B, C.
   (b) The letter S, which, when it appears, is usually the start symbol.
   (c) Lowercase, italic names such as *expr* or *stmt*.
   (d) When discussing programming constructs, uppercase letters may be used to represent nonterminals for the constructs. For example, nonterminals for expressions, terms, and factors are often represented by E, T, and F, respectively.

Context-Free Grammars (CFGs)

Notational conventions for grammars

3. Uppercase letters late in the alphabet, such as *X, Y, Z,* represent *grammar symbols*; that is, either nonterminals or terminals.

4. Lowercase letters late in the alphabet, chiefly *u, v, …, z,* represent (possibly empty) strings of terminals.

5. Lowercase Greek letters, α, β, γ for example, represent (possibly empty) strings of grammar symbols. Thus, a generic production can be written as $A \rightarrow \alpha$, where A is the head and α is the body.

6. A set of productions $A \rightarrow \alpha_1$, $A \rightarrow \alpha_2$, ……., $A \rightarrow \alpha_k$ with a common head A (call them A-productions), may be written $A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_k$ Call $\alpha_1, \alpha_2, \ldots, \alpha_k$ the *alternatives* for A.

7. Unless stated otherwise, the head of the first production is the start symbol.

Context-Free Grammars (CFGs)

- Example of a Context-free Grammar

$S \rightarrow (\ S \ )$

$S \rightarrow \varepsilon$

$\Rightarrow$ set of terminals $T = \{(,)\}$

$\Rightarrow$ set of non-terminals $N = \{S\}$

Context-Free Grammars (CFGs)

- Example of a Context-free Grammar

expression → *expression* + *term*

expression → *expression* - *term*

expression → *term*

term → *term* * *factor*

term → *term* / *factor*

term → *factor*

factor → (*expression*)

factor → ***id***

⇒ set of terminals T = ?

⇒ set of non-terminals N = ?

Derivations

- Consider the following grammar: $E \rightarrow E + E \mid E * E \mid - E \mid (E) \mid id$
- The production $E \rightarrow -E$ signies that if E denotes an expression, then E must also denote an expression.
- The replacement of a single E by -E will be described by writing $E \Rightarrow -E$ which is read, “E derives -E"
- The production $E \rightarrow (E)$ can be applied to replace any instance of E in any string of grammar symbols by (E), e.g., $E * E \Rightarrow (E) * E$ or $E * E \Rightarrow E * (E)$
- We can take a single E and repeatedly apply productions in any order to get a sequence of replacements.

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(id)$$

- We call such a sequence of replacements a derivation of **-(id)** from *E*. This derivation provides a proof that the string **-(id)** is one particular instance of an expression.

Derivations

- The string `-(id + id)` is a sentence of grammar because there is a derivation
  $E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E + E) \Rightarrow -(id + E) \Rightarrow -(id + id)$
- We write $E \Rightarrow^* -(id + id)$ to indicate that `-(id + id)` can be derived from E.

Derivations

- In *leftmost derivations*, the leftmost nonterminal in each sentential is always chosen.

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E + E) \Rightarrow -(id + E) \Rightarrow -(id + id)$$

- In *rightmost derivations*, the rightmost nonterminal is always chosen.

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E + E) \Rightarrow -(E + id) \Rightarrow -(id + id)$$

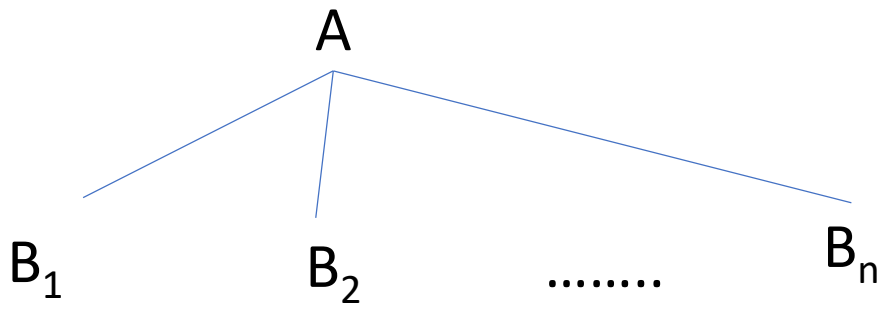
- Note that right-most and left-most derivations have the same parse tree.
- Grammar

  $E \rightarrow E + E \mid E * E \mid - E \mid (E) \mid id$
- String

  id * id + id
- Left-most derivation:
- Right-most derivation:

Parse Trees and Derivations

- A derivation can be drawn as a tree
  - Start symbol is the root of the tree
  - For a production $A \rightarrow B_1B_2\ldots.B_n$ add children $B_1B_2\ldots.B_n$ to node A



Parse Trees and Derivations

- Grammar

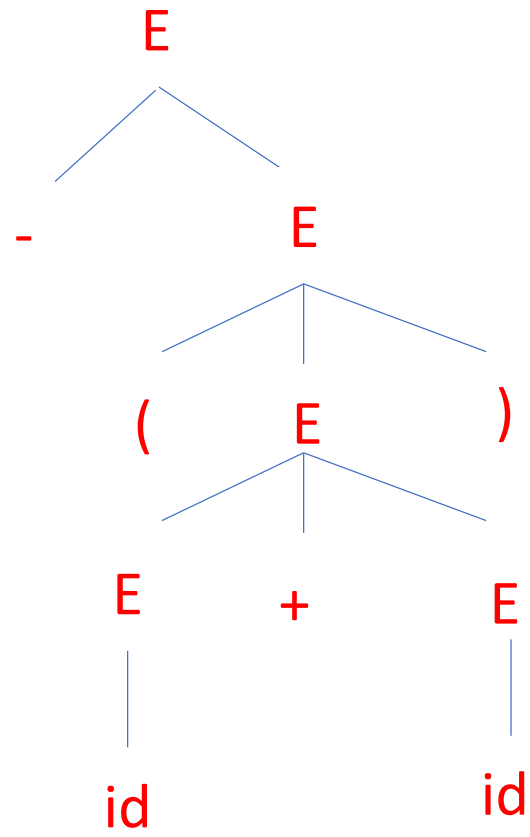
  $E \rightarrow E + E \mid E * E \mid - E \mid (E) \mid id$

- String

  -(id + id)

- Derivation

  E
  ⇒ -E
  ⇒ -(E)
  ⇒ -(E + E)
  ⇒ -(id + E)
  ⇒ -(id + id)



Parse Trees and Derivations

E

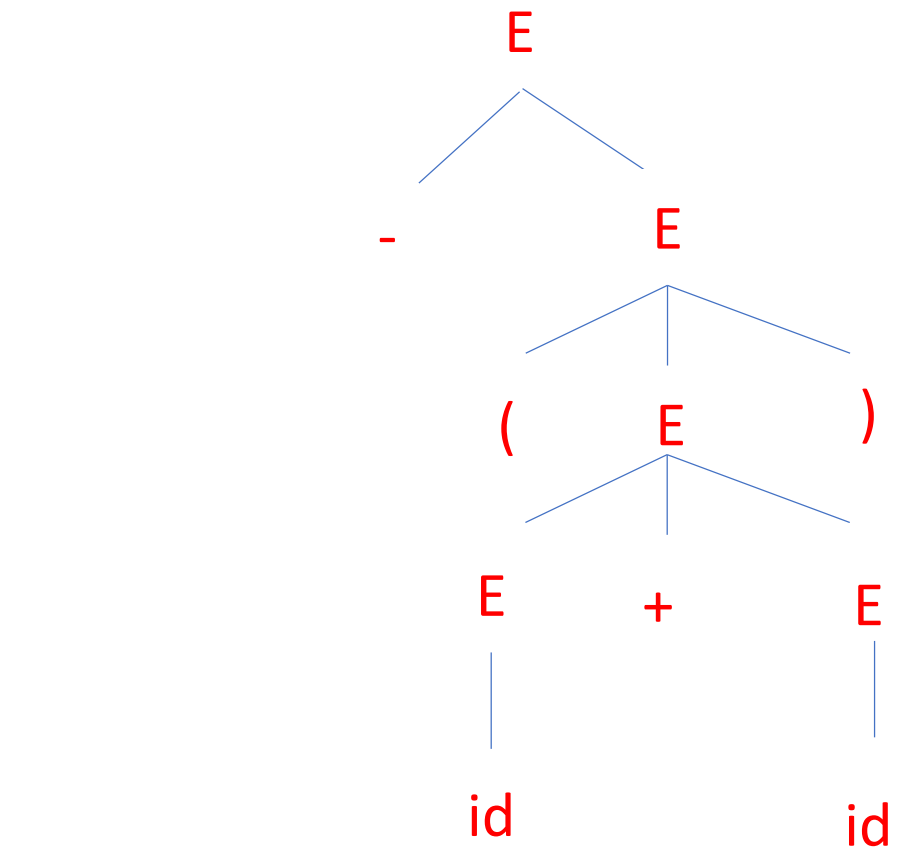
⇒ - E

⇒ - (E)

⇒ - (E + E)

⇒ - (id + E)

⇒ - (id + id)



Parse Trees and Derivations

- A parse tree is a graphical representation of a derivation that filters out the order in which productions are applied to replace nonterminals.
- A parse tree has
  - Terminals at the leaves
  - Non-terminals at the interior nodes
- Each interior node of a parse tree represents the application of a production
  - The interior node is labeled with the nonterminal (A) in the head of the production
  - The children of the node are labeled, from left to right, by the symbols in the body of the production by which this A was replaced during the derivation
- An in-order traversal of the leaves is the original input
- The parse tree shows the association of operations, the input string does not.

Parse Trees and Derivations

E

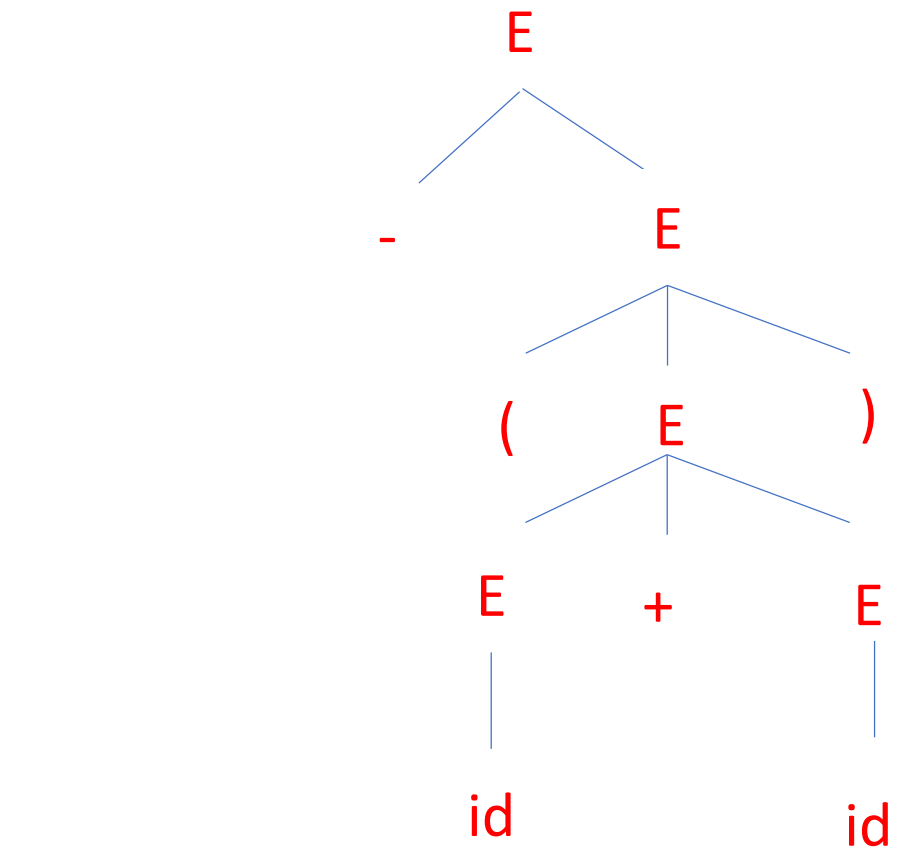
→ - E

→ - (E)

→ - (E + E)

→ - (id + E)

→ - (id + id)



Ambiguity

- A grammar that produces more than one parse tree for some string is said to be ambiguous.
  - Equivalently, there is more than one left-most or right-most derivation for some string.
- Grammar

  $E \rightarrow E + E \mid E * E \mid - E \mid (E) \mid id$

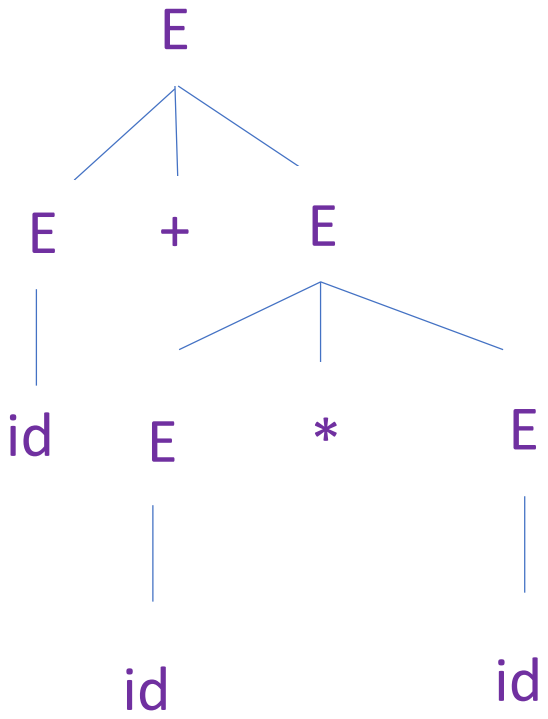
- String

  id + id * id

- This string has two parse trees

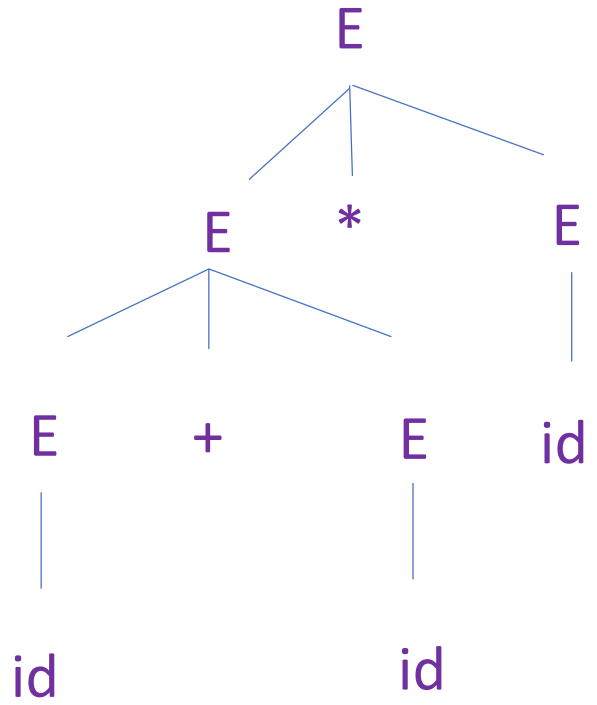
Ambiguity

- This string (id + id * id) has two parse trees



E => E + E
=> id + E
=> id + E * E
=> id + id * E
=> id + id * id

E => E * E
=> E + E * E
=> id + E * E
=> id + id * E
=> id + id * id



Ambiguity

- A grammar that produces more than one parse tree for some string is said to be ambiguous.
  - Equivalently, there is more than one left-most or right-most derivation for some string.
- Ambiguity is BAD.
- For most parsers, it is desirable that the grammar be made unambiguous, for if it is not, we cannot uniquely determine which parse tree to select for a sentence.

Exercises

Exercise 1: Consider the context-free grammar:

$$S \rightarrow S\ S\ +\ |\ S\ S\ *\ |\ a$$

and the string aa + a*

a) Give a leftmost derivation for the string.

b) Give a rightmost derivation for the string.

c) Give a parse tree for the string

d) Is the grammar ambiguous or unambiguous? Justify your answer

e) Describe the language generated by this grammar

Exercises

Exercise 2: Repeat Exercise 1 for each of the following grammars and strings:

a) S → 0 S 1 | 0 1 with string 000111

b) S → + S S | * S S | a with string + * aaa

c) S → S (S) S | ε with string (()())

d) S → S + S | S S | (S) | S * | a with string (a + a)*a

e) S → (L) | a

   L → L , S | S with string ((a, a), a, (a))

f) S → a S b S | b S a S | ε with string aabbab

Exercises

Exercise 3: Design grammars for the following languages:

a) The set of all strings of 0s and 1s that are palindromes; that is, the string reads the same backward as forward.

b) The set of all strings of 0s and 1s with an equal number of 0s and 1s

Exercises

Exercise 4:

- Let Σ = {void, int, float, double, name, (,), ,, ;}.
- Let's write a CFG for C-style function prototypes!
- Examples:

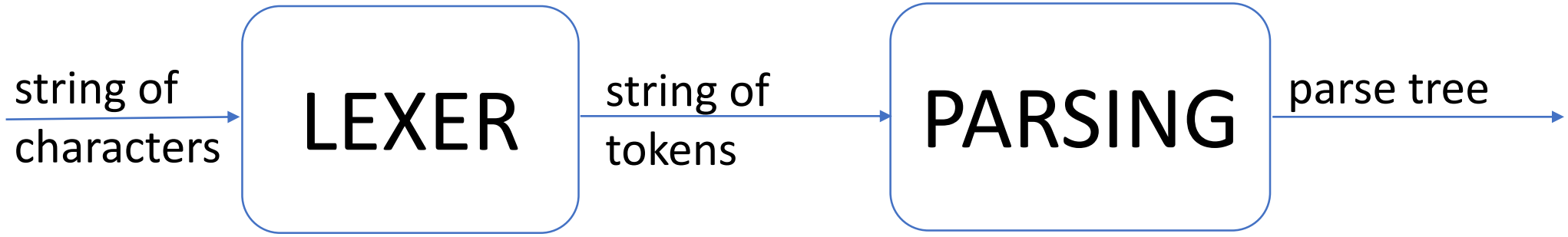
```
void name(int name, double name);
int name();
int name(double name);
int name(int, int name, float);
void name(void);
```

Function Prototypes

Here's one possible grammar:

```
S → Ret name (Args);
Ret → Type | void
Type → int | double | float
Args → ε | void | ArgList
ArgList → OneArg | ArgList, OneArg
OneArg → Type | Type name
```

Abstract Syntax Trees (AST)



- A parse tree is a graphical representation of a derivation that filters out the order in which productions are applied to replace nonterminals.
  - Parse tree (Concrete Syntax Tree – CST)

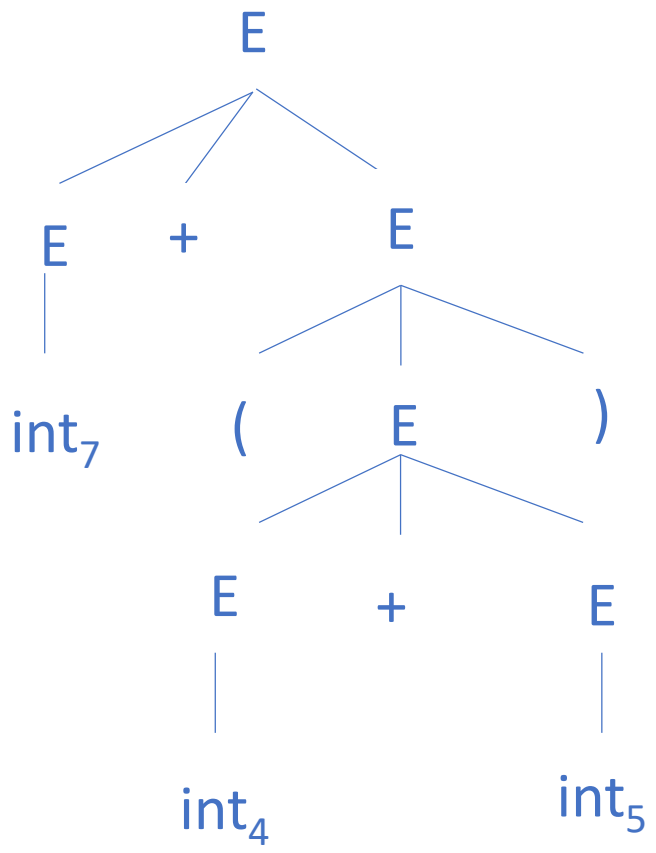
- Abstract syntax trees like parse trees but ignore some details
  - In the abstract syntax tree (syntax tree), interior nodes represent programming constructs
  - In the parse tree (concrete syntax trees), interior nodes represent nonteminals

- Abstract Syntax Trees or ASTs are tree representations of code.

Abstract Syntax Trees (AST)

- Consider the grammar: `E → int | (E) | E + E | E * E`
- And the string: `7 + (4 + 5)`
- After lexical analysis: $\texttt{int}_7$ ‘+’ ‘(‘ $\texttt{int}_4$ ‘+’ $\texttt{int}_5$ ‘)’
- During parsing we build a parse tree

Example of parse tree

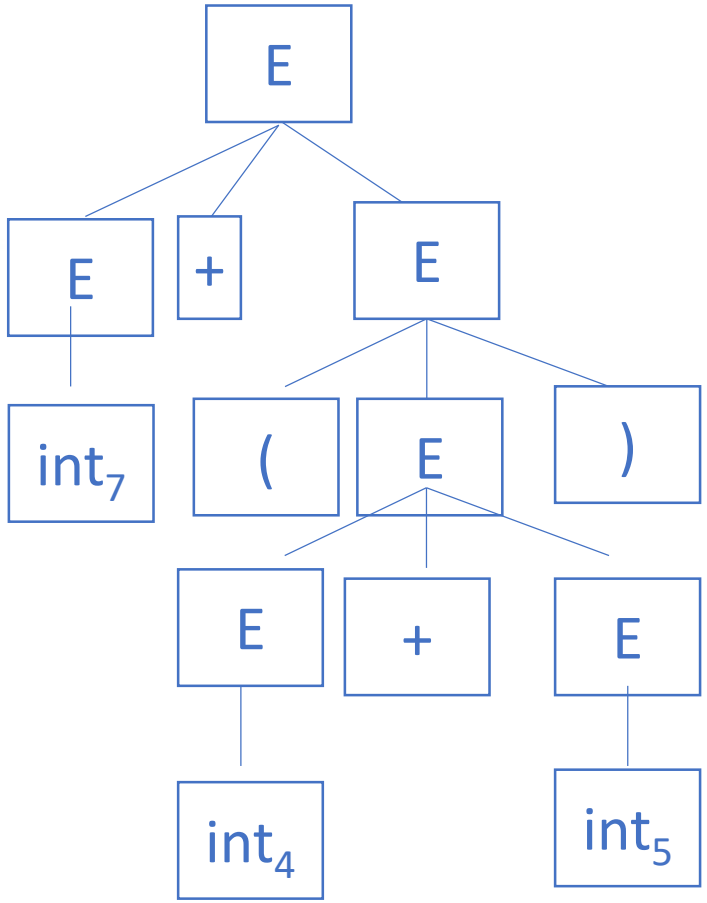


Too much information

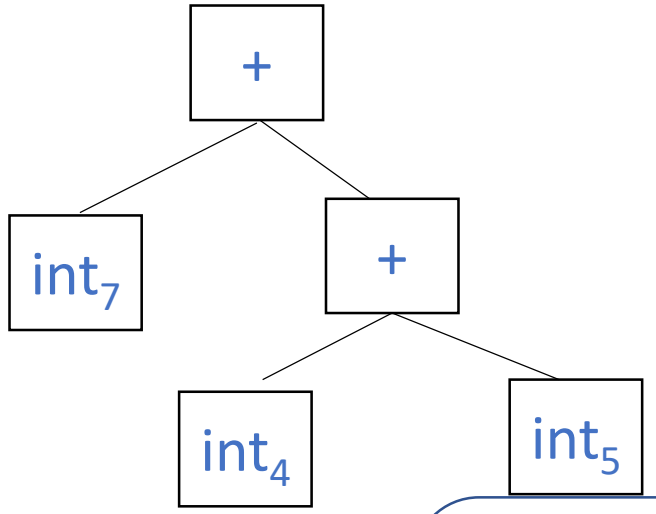
- Parentheses
- Single-successor nodes

Example of abstract syntax tree

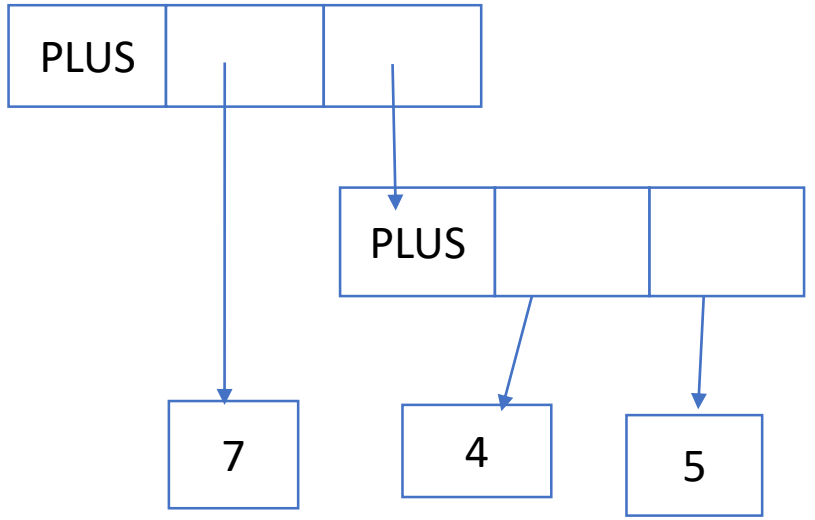
Parse Tree



Abstract Syntax Tree



AST is represented as



- Also captures the nesting structure
- But abstracts from the concrete syntax
  ⇒ More compact and easier to use
- An important data structure in a compiler

Parse Tree vs Syntax Tree

Parse Tree	Syntax Tree
A parse tree is a graphical representation of the replacement process in a derivation	A syntax tree is a condensed form of parse tree
In parse trees - Each interior node represents a grammar rule - Each leaf node represents a terminal	In syntax trees - Each interior node represents an operator - Each leaf node represents an operand
Parse tree represent every detail from the real syntax	Syntax trees does not represent every detail from the real syntax (that's why they are called abstract). For example: no rule nodes, no parentheses, …

Exercise 1

- Consider the following grammar:

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

$$F \rightarrow (E) \mid id$$

- For the string

$$id + id * id$$

- Genarate
  a) Parse tree
  b) Syntax tree

Error Handling

- Purpose of the compiler
  - To translate the valid programs
  - To detect non-valid programs
- Common programming errors

Error kind	Example (C)	Detected by
Lexical - Misspelling of identifiers, keywords, operators, … - Missing quotes around text intended as a string - ……	- …$…. - ……	Lexer
Syntactic When all the individual lexical units are correct but they're assembled insome way that doesn't make sense and we don't know how to compile it.	…..a*%…….	Parser
Semantic - Type mismatch	….. int a; b = a(8); …….	Type checker
Correctness The program is actually a valid program but it doesn't do what we intended.		Tester/User

Error Handling

- Error handler should
  - Report the presence of errors clearly and accurately
  - Recover from each error quickly enough to detect subsequent errors
  - Not slow down compilation of valid code

- How should an error handler report the presence of an error?
  - Report the place in the source program where an error is detected

Error recovery strategies

- Panic mode
- Error productions
- Global Correction

Error recovery strategies: Panic Mode

- Simplest, most polular method
- On discovering an error:
  - Discard tokens until one of a designated set of synchronizing tokens is found.
  - Continue from there
- Synchronizing token: next token that matches the rule being parsed
- The compiler designer must select the synchronizing tokens appropriate for the source language.

Error recovery strategies: Panic Mode

- Consider an expression: (1++2)+3

- Panic mode recovery:
  - Skip ahead to next integer and then continue

Error recovery strategies: Error Productions

- Idea:
  - Anticipate common errors that might be encountered
  - Augment the grammar for the language with productions that generate the erroneous constructs
- Exampe
  - Write 7a instead of 7*a
  - Add the production E → ……| EE
- Disadvantage
  - Complicate the grammar

Error recovery strategies: Global Correction

- Idea: find a correct "nearby" program

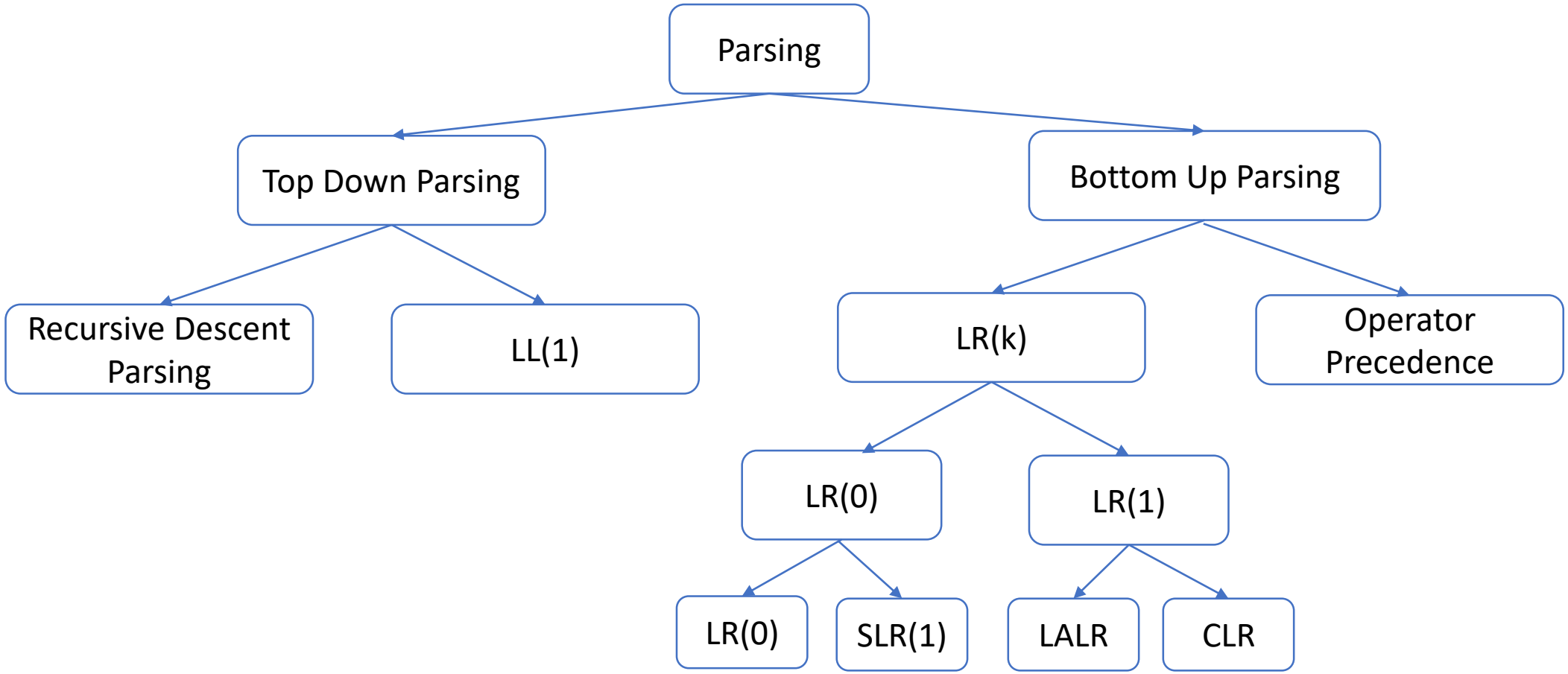
- Example:
  - Given an incorrect input string *x* and grammar *G*
  - Will find a parse tree for a related string *y*, such that the number of insertions, deletions, and changes of tokens required to transform *x* into *y* is as small as possible.

- Disadvantages:
  - Too costly to implement in terms of time and space
  - Slow down parsing of correct programs
  - "Nearby" may not be what the programmar had in mind

Error recovery strategies: Past and Present

- Past
  - Slow recompilation cycle (even once a day)
  - Find as many errors in one cycle as possible
- Present
  - Quick recompilation cycle
  - Users tend to correct one error/cycle
  - Panic-mode seems enough

Parsing


Parsing
Top Down Parsing
Bottom Up Parsing
Recursive Descent Parsing
LL(1)
LR(k)
Operator Precedence
LR(0)
LR(1)
LR(0)
SLR(1)
LALR
CLR

Top-Down Parsing

- Parsing is a process of deriving string from a given grammar.
- Top-down parsing is a process of constructing a parse tree for the input string,
  - From the top
  - From left to right
- Top-down parsing can be viewed as finding a leftmost derivation for an input string

- Consider the grammar:

$$
\begin{aligned}
E &\rightarrow TE' \\
E' &\rightarrow +TE' \mid \varepsilon \\
T &\rightarrow FT' \\
T' &\rightarrow *FT' \mid \varepsilon \\
F &\rightarrow (E) \mid \mathbf{id}
\end{aligned}
$$

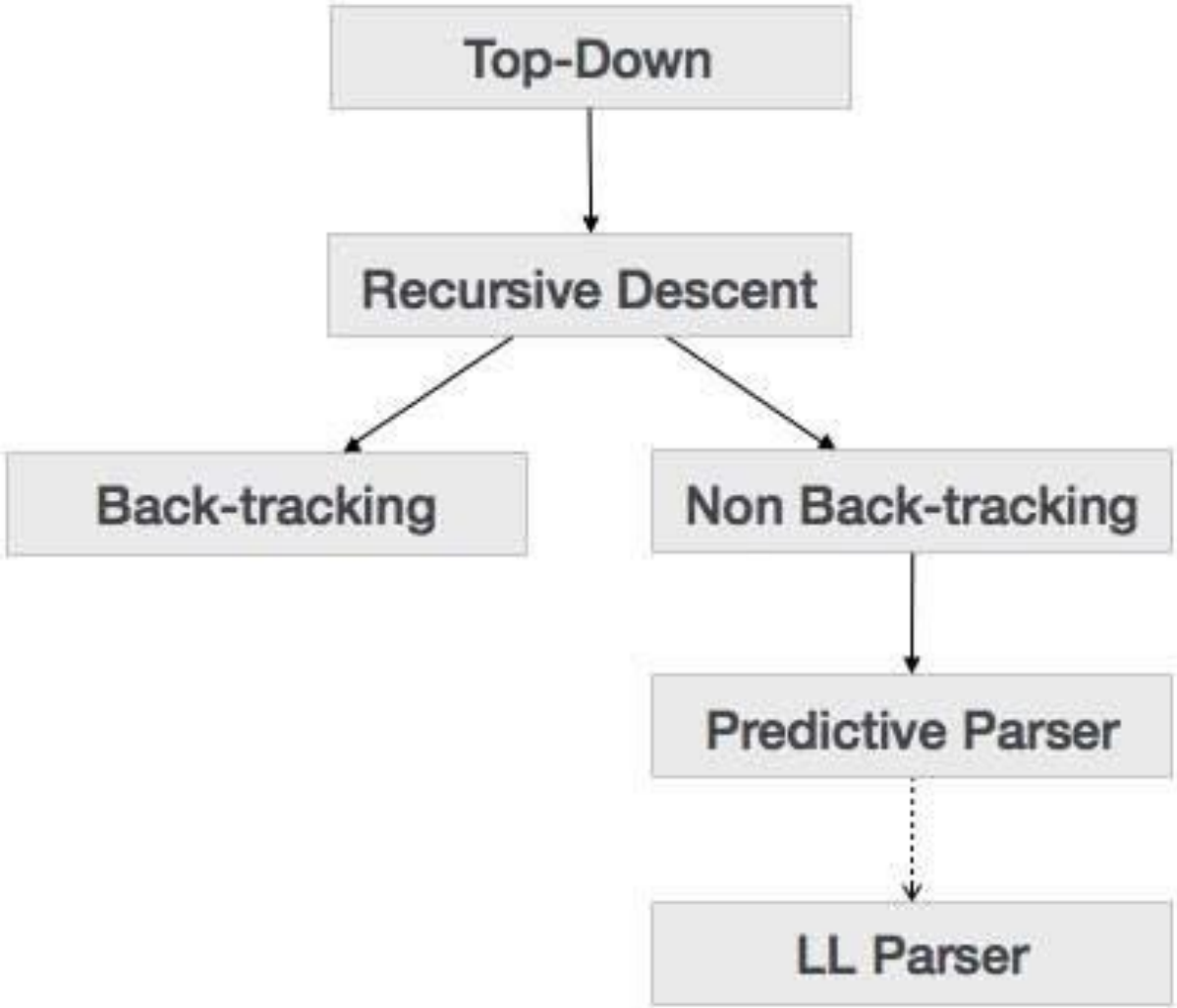
- And the string: `id+id*id`

- Give a leftmost derivation for the string
- Give a parse tree for the string

Recursive Descent Parsing

- is a top-down parsing technique that constructs the parse tree from the top and the input is read from left to right.
- uses procedures for every terminal and non-terminal entity.
- recursively parses the input to make a parse tree, which may or may not require back-tracking
- A form of recursive-descent parsing that does not require any back-tracking is known as **predictive parsing**.

Recursive Descent Parsing



https://www.tutorialspoint.com/

Recursive Descent Parsing

- Consider the grammar:

  E → T | T-E

  T → int| int*T | (E)

- And the string: (7)

- After lexical analysis: ‘(‘ int$_7$ ‘)’

- We build a parse tree

Recursive Descent Parsing

$E \rightarrow T \mid T\text{-}E$

$T \rightarrow int \mid int*T \mid (E)$

‘(‘ int_7 ‘)’

Recursive Descent Parsing

- A recursive-descent parsing program
  - Consists of a set of procedures,
  - One for each nonterminal

- Execution begins with the procedure for the start symbol

- General recusive-desent may require backtracking – it may require repeated scans over the input

Recursive Descent Parsing

- Define boolean functions that check the token string for a match of
  - A given token terminal

    ```
    bool term (TOKEN tok) {return *next++ == tok;}
    ```

  - The n^{th} production of S

    ```
    bool Sn() {……}
    ```

  - Try all productions of S

    ```
    bool S() {……..}
    ```

Recursive Descent Parsing

- For production `E → T`

```
bool E1() {return T();}
```

- For production `E → T – E`

```
bool E2() {return T() && term(MINUS) && E();}
```

- For all productions of E (with backtracking)

```
bool E(){
    TOKEN *save = next;
    return (next = save, E1())
        ||(next = save, E2());
}
```

Recursive Descent Parsing

- Functions for non-terminal T: $T \rightarrow int \mid int*T \mid (E)$

```
bool T1() {return term(INT);}
bool T2() {return term(INT) && term (TIMES) && T();}
bool T3() {return term(OPEN) && E() && term(CLOSE);}
bool T(){
    TOKEN *save = next;
    return (next = save, T1())
        ||(next = save, T2())
        ||(next = save, T3());
}
```

Recursive Descent Parsing

- To start the parser
  - Initialize next to point to first token
  - Invoke E()

- Easy to implement by hand
  - But not completely general
  - Cannot backtrack once a production is successful
  - Works for grammars where at most one production can succeed for a non-terminal

Recursive Descent Parsing

```
bool term (TOKEN tok) {return *next++ == tok;}
bool E1() {return T();}
bool E2() {return T() && term(MINUS) && E();}
bool E(){
        TOKEN *save = next;
        return (next = save, E1())
             ||(next = save, E2()); }
bool T1() {return term(INT);}
 bool T2() {return term(INT) && term (TIMES) && T();}
 bool T3() {return term(OPEN) && E() && term(CLOSE);}
 bool T(){
        TOKEN *save = next;
        return (next = save, T1())
             ||(next = save, T2())
             ||(next = save, T3()); }
```

‘(‘ int7 ‘)’

Recursive Descent Limitation

```
bool term (TOKEN tok) {return *next++ == tok;}
bool E1() {return T();}
bool E2() {return T() && term(MINUS) && E();}
bool E(){
        TOKEN *save = next;
        return (next = save, E1())
             ||(next = save, E2()); }
bool T1() {return term(INT);}
 bool T2() {return term(INT) && term (TIMES) && T();}
 bool T3() {return term(OPEN) && E() && term(CLOSE);}
 bool T(){
        TOKEN *save = next;
        return (next = save, T1())
             ||(next = save, T2())
             ||(next = save, T3()); }
```

int

int * int

When Recursive Descent Does Not Work

- Consider a production `E→ E + int`

- The Recursive Descent Algorithm for this production

```
bool term (TOKEN tok) {return *next++ == tok;}
bool E1() { return E() && term(PLUS) && term(INT); }
bool E()  { return E1(); }
```

-> E() goes into an infinite loop

Left recursion

- A grammar is left-recursive if and only if there exists a nonterminal symbol that can derive to a sentential form with itself as the leftmost symbol.

  $A \Rightarrow^{+} A\alpha,$

  where $\Rightarrow^{+}$ indicates the operation of making one or more substitutions, and α is any sequence of terminal and nonterminal symbols.

- Recursive descent parsing does not work in such cases.

Elimination of Left Recursion

- Consider the left-recursive grammar $A \rightarrow A\alpha \mid \beta$,
  where α and β are sequences of terminals and nonterminals that do not start with A.

- The nonterminal A and its production are said to be left recursive, because the production $A \rightarrow A\alpha$ has A itself as the leftmost symbol on the right hand side.

- A generates all strings starting with a β and followed by a number of α.

- For example, in `expr → expr + term | term`
  nonterminal A = `expr`, string α = `+ term`, and string β = `term`.

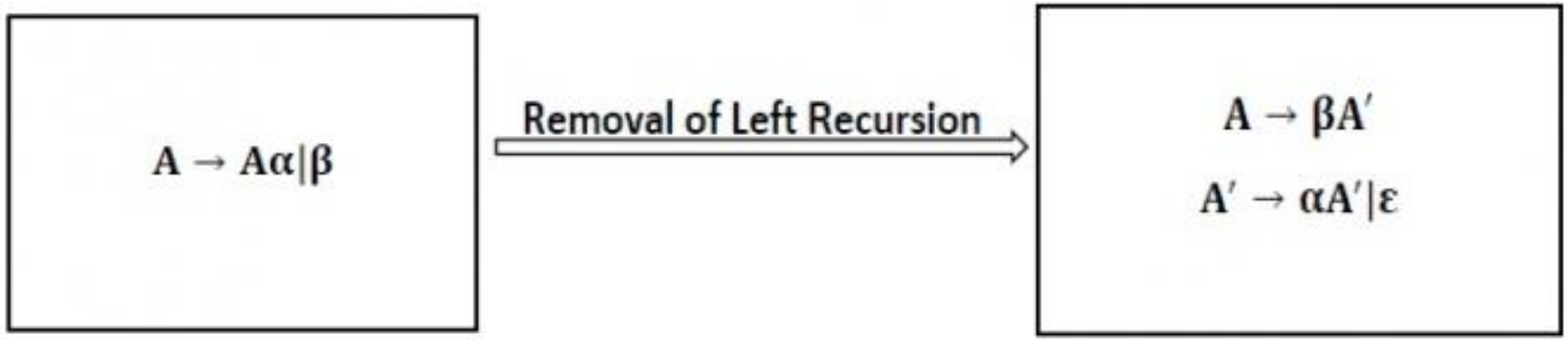
Removing left recursion

- Left recursion often poses problems for parsers
  - either because it leads them into infinite recursion (as in the case of most top-down parsers)
  - or because they expect rules in a normal form that forbids it (as in the case of many bottom-up parsers, including the CYK algorithm).

- Therefore, a grammar is often preprocessed to eliminate the left recursion.

- The left-recursive grammar $A \rightarrow A\alpha \mid \beta$,
  can rewrite using right-recursion
  $A \rightarrow \beta A'$
  $A' \rightarrow \alpha A' \mid \varepsilon$

Removing left recursion



Removal of Left Recursion

https://www.tutorialspoint.com/

Removing left recursion

- In general $A \rightarrow A\alpha_1 | \ldots | A\alpha_n | \beta_1 | \ldots | \beta_m$,
- A generates all strings starting with one of $\beta_1, \ldots, \beta_m$ and followed by several instances of $\alpha_1, \ldots, \alpha_n$
- Can rewrite as

$$A \rightarrow \beta_1 A' | \ldots | \beta_m A'$$
$$A' \rightarrow \alpha_1 A' | \ldots | \alpha_n A' | \varepsilon$$

General Left Recursion

- The grammar
  $A \rightarrow A'\alpha \mid \beta$
  $A' \rightarrow A\delta$
  -> Is it a left recursive grammar?

- It is also left-recursive because $A \Rightarrow^{+} A\delta\alpha$

A grammar is left-recursive if and only if there exists a nonterminal symbol that can derive to a sentential form with itself as the leftmost symbol.

$A \Rightarrow^{+} A\alpha,$

where $\Rightarrow^{+}$ indicates the operation of making one or more substitutions, and α is any sequence of terminal and nonterminal symbols.

General Left Recursion

- Consider the grammar

  $S \rightarrow Aa \mid b$

  $A \rightarrow Ac \mid Sd \mid \varepsilon$

-> The nonterminal S is left recursive???

A grammar is left-recursive if and only if there exists a nonterminal symbol that can derive to a sentential form with itself as the leftmost symbol.

$A \Rightarrow^{+} A\alpha,$

where $\Rightarrow^{+}$ indicates the operation of making one or more substitutions, and α is any sequence of terminal and nonterminal symbols.

- The nonterminal S is left recursive because $S \Rightarrow Aa \Rightarrow Sda$, but it is not immediately left recursive.

Left Recursion

- Consider the grammar

  S→ a|^(T)

  T→ ST’

  T’ →,ST’| ε

  -> Is it a left recursive grammar?

A grammar is left-recursive if and only if there exists a nonterminal symbol that can derive to a sentential form with itself as the leftmost symbol.

$A \Rightarrow^{+} A\alpha,$

where $\Rightarrow^{+}$ indicates the operation of making one or more substitutions, and α is any sequence of terminal and nonterminal symbols.

Algorithm For Eliminating Left Recursion

- INPUT: Grammar G with no cycles (derivations of the form $A \Rightarrow^+ A$) or ε-productions.
- OUTPUT: An equivalent grammar with no left recursion.
  - Note that the resulting non-left-recursive grammar may have ε-productions.
- METHOD:

1) arrange the nonterminals in some order $A_1, A_2, \ldots, A_n$.
2) for (each i from 1 to n) {
3) for (each j from 1 to i-1) {
4) replace each production of the form $A_i \rightarrow A_j\gamma$ by the productions $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$, where $A_i \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$ are all current A_j-productions
5) }
6) eliminate the immediate left recursion among the A_j-productions
7) }

Example

- Let apply that algorithm (Algorithm For Eliminating Left Recursion in the previous slide) to the grammar
  $S \rightarrow Aa \mid b$
  $A \rightarrow Ac \mid Sd \mid \varepsilon$
- Technically, the algorithm is not guaranteed to work, because of the ε-production, but in this case, the production $A \rightarrow \varepsilon$ turns out to be harmless.

- We order the nonterminals S, A.
- For i = 1: nothing happens
  (because there is no immediate left recursion among the S-productions)
- For i = 2:
  - substitute for S in $A \rightarrow Sd$ to obtain the following A-productions
    $A \rightarrow Ac \mid Aad \mid bd \mid \varepsilon$

  - eliminate the immediate left recursion among these A-productions yields the following grammar

$$S \rightarrow Aa \mid b$$
$$A \rightarrow bdA' \mid A'$$
$$A' \rightarrow cA' \mid adA' \mid \varepsilon$$

Example

- Let apply that algorithm (Algorithm For Eliminating Left Recursion in the previous slide) to the grammar

$$A \rightarrow A'\alpha \mid \beta$$
$$A' \rightarrow A\delta$$

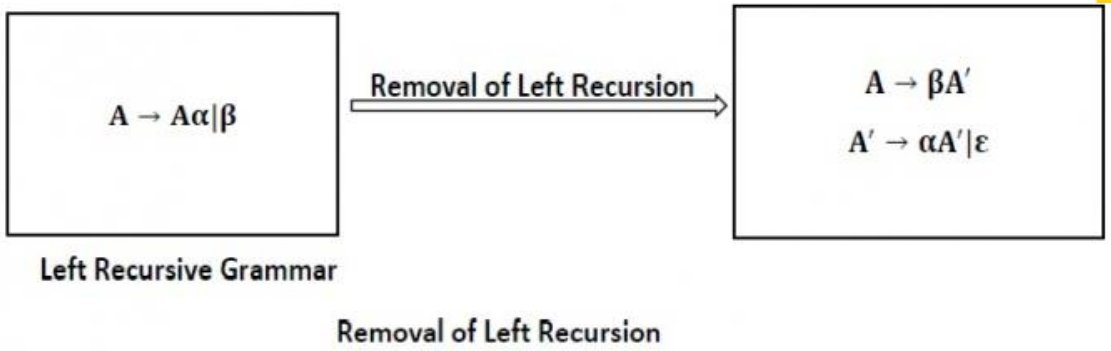
- We order the nonterminals ???
- For i = 1: ????
- For i = 2: ????

Exercise1

- Consider the Left Recursion Grammar

```
E → E + T|T
T → T * F|F
F → (E)|id
```

Eliminate immediate left recursion from the Grammar.



Exercise1

Solution

- Comparing $E \rightarrow E + T | T$ with $A \rightarrow A\ \alpha\ | \beta$

  A = E, α = +T, β = T

  1) A → A α |β is changed to A → βA'and A' → α A'|ε
  2) A → βA' means E → TE'
  3) A' → α A'|ε means E' → +TE'|ε

- Comparing $T \rightarrow T * F | F$ with $A \rightarrow A\alpha | \beta$

  4) A = T, α =* F, β = F
  5) A → β A' means T → FT'

  A → α A'|ε means T' →* FT'|ε

- Production F → (E)|id does not have any left recursion
- Combining productions 1, 2, 3, 4, 5, we get

  E → TE'
  E' → +TE'| ε
  T → FT'
  T →* FT'|ε
  F → (E)| id

Exercise2

- Consider the grammar

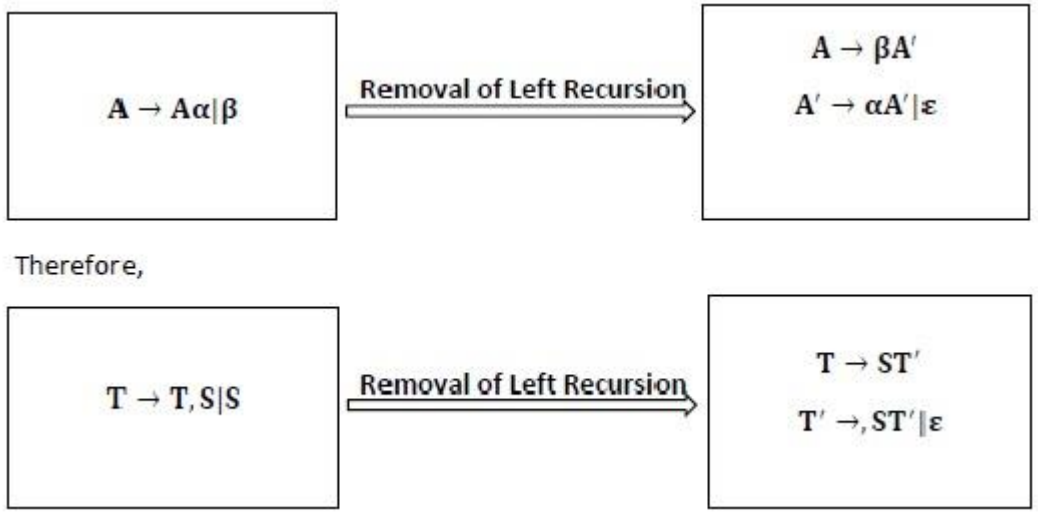
  S → a|^(T)

  T → T,S|S

  Eliminate the left recursion from the grammar.

Exercise2 - Solution

- We have immediate left recursion in T-productions.
- Comparing `T → T, S|S With A → A α | β where A = T, α =, S and β = S`



- Complete Grammar will be

```
S    → a|^(T)
T    → ST'
T'   →,ST'| ε
```

Exercise3

- Eliminate the left recursion from the grammar

```
E → E + T|T
T → T * F|F
F → (E)|id
```

Exercise4

- Eliminate the left recursion from the grammar

```
E → E(T)|T
T → T(F)|F
F → id
```

Predictive Parsing

- Recursive-descent parsing is a top-down method of syntax analysis
  - in which a set of recursive procedures is used to process the input.
  - one procedure is associated with each nonterminal of a grammar.

- Predictive parsing is a special case of recursive-descent parsing, but parser can “predict” which production to use
  - By looking at the next few tokens
  - No backtracking

- The sequence of procedure calls during the analysis of an input string implicitly defines a parse tree for the input.

Predictive Parsing

- Consider the grammar for some statements in C and Java

```
stmt → expr;
     | if (expr) stmt
     | for (optexpr; optexpr; optexpr) stmt
     | other

optexpr → ε
        | expr
```

Pseudocode for a

```
stmt → expr;
       | if (expr) stmt
       | for (expr; expr; expr) stmt
```

INPUT: for (expr; expr; expr) other

Parse tree for the input string?

```
void stmt () {
        switch (lookahead) {
        case expr:
                match(expr); match(';'); break;
        case if:
                match(if); match('('); match(expr); match(')'); stmt();
                break;
        case for:
                match(for); match('(');
                optexpr (); match(';'); optexpr (); match(';'); optexpr (); match(')');
                stmt (); break;
        case other:
                match(other); break;
        default:
                report ("syntax error");

        }
}
void optexpr () {
        if (lookahead == expr) match(expr);

}


void match(terminal t) {
        if (lookahead == t) lookahead = nextTerminal;
        else report ("syntax error");
}
```

- Predictive parsers accept LL(k)
  - **L** means “left-to-right” scan of input
  - **L** means “leftmost derivation”
  - **k** means “predict based on k tokens of lookahead”
  - In practice, **LL(1)** is used

LL(1) vs. Recursive Descent

- In recursive-descent,
  - At each step, many choices of production to use
  - Backtracking used to undo bad choices
- In LL(1),
  - At each step, only one choice of production
  - That is
    - When a non-terminal A is leftmost in a derivation
    - And the next input symbol is t
    - There is a unique production $A \rightarrow \alpha$ to use
      - Or no production to use (an error state)
- LL(1) is a recursive descent variant without backtracking

Predictive parsing

- Predictive parsing is a special case of recursive-descent parsing, but parser can “predict” which production to use
  - By looking at the next few tokens
  - No backtracking

- Predictive parsing relies on information about the first symbols that can be generated by a production body.

→It restricts the grammars.

→This is only works for a restricted form of grammars

Left factoring

- Consider the grammar

```
stmt → if expr then stmt else stmt
     | if expr then stmt
```

→ on seeing the input **if**, we cannot immediately tell which production to choose to expand *stmt*.

→ We need to left-factor the grammar

Left factoring

- Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive, or top-down, parsing.

- When the choice between two alternative A-productions is not clear, we may be able to rewrite the productions to defer the decision until enough of the input has been seen that we can make the right choice.
- Consider A-productions: $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$
  and the input begins with a nonempty string derived from α
  $\rightarrow$ we don't know whether to expand A to $\alpha\beta_1$ or $\alpha\beta_2$
  $\rightarrow$ we may defer the decision by
  - expanding A to $\alpha A'$
  - then exapanding A' to β_1 or β_2

  $\rightarrow$ the original productions become

  $A \rightarrow \alpha A'$

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

Left factoring

- Consider the grammar
  $E \rightarrow T + E \mid T$
  $T \rightarrow int \mid int * T \mid (E)$

=> Left factor this grammar.

Consider A-productions: $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$
and the input begins with a nonempty string derived from α
$\rightarrow$ we don't know whether to expand A to $\alpha\beta_1$ or $\alpha\beta_2$
$\rightarrow$ we may defer the decision by
- expanding A to $\alpha A'$
- then exapanding A' to β_1 or β_2

$\rightarrow$ the original productions become
$A \rightarrow \alpha A'$
$A' \rightarrow \beta_1 \mid \beta_2$

FIRST and FOLLOW

- During top-down parsing, FIRST and FOLLOW allow us to choose which production to apply, based on the next input token.
- FIRST
  - Definition: $FIRST(X) = \{b \mid X \rightarrow^* b\alpha\} \cup \{\varepsilon \mid X \rightarrow^* \varepsilon\}$
  - $FIRST(X)$ to be the set of terminals that begin strings derived from X
  - Compute $FIRST(X)$:
    - If X is a terminal: $FIRST(X) = \{X\}$
    - If X is a nonterminal
      - $\varepsilon \in FIRST(X)$
        - If $X \rightarrow \varepsilon$
        - If $X \rightarrow Y_1Y_2...Y_k$ and $\varepsilon \in FIRST(Y_i)$ for all $1 <= i <= k$
      - $b \in FIRST(X)$
        - If $X \rightarrow Y_1Y_2...Y_k$ and $b \in FIRST(Y_i)$ and $\varepsilon \in FIRST(Y_j)$ for all $1 <= j <= i-1$

FIRST example

- Consider the gr

  $E \rightarrow T\ F$

  $F \rightarrow +\ E \mid \varepsilon$

  $T \rightarrow \textbf{int}\ Y \mid (\ E\ )$

  $Y \rightarrow *\ T \mid \varepsilon$

- FIRST (E) =
  FIRST (F) =
  FIRST (T) =
  FIRST (Y) =

Compute FIRST(X):

If X is a terminal: FIRST(X) = {X}

If X is a nonterminal

$\varepsilon \in$ FIRST(X)

If $X \rightarrow \varepsilon$

If $X \rightarrow Y_1Y_2\ldots Y_k$ and $\varepsilon \in$ FIRST(Y_i) for all 1 <= i <= k

$b \in$ FIRST(X)

If $X \rightarrow Y_1Y_2\ldots Y_k$ and $b \in$ FIRST(Y_i) and $\varepsilon \in$ FIRST(Y_j)
for all 1 <= j <= i-1

FIRST example

- Consider the grammar
  $E \rightarrow T\ F$
  $F \rightarrow +\ E \mid \varepsilon$
  $T \rightarrow \textbf{int}\ Y \mid (\ E\ )$
  $Y \rightarrow *\ T \mid \varepsilon$

- FIRST (() =
  FIRST ()) =
  FIRST (int) =
  FIRST (+) =
  FIRST (*) =
- FIRST (E) =
  FIRST (F) =
  FIRST (T) =
  FIRST (Y) =

Compute FIRST(X):

If X is a terminal: FIRST(X) = {X}

If X is a nonterminal

$\varepsilon \in$ FIRST(X)

If $X \rightarrow \varepsilon$

If $X \rightarrow Y_1Y_2...Y_k$ and $\varepsilon \in$ FIRST(Y_i) for all 1 <= i <= k

$b \in$ FIRST(X)

If $X \rightarrow Y_1Y_2...Y_k$ and $b \in$ FIRST(Y_i) and $\varepsilon \in$ FIRST(Y_j) for all 1 <= j <= i-1

FIRST example

- Production Rules of Grammar

```
E  -> TE'
E' -> +T E'|ε
T  -> FT'
T' -> *F T' | ε
F  -> (E) | id
```

- FIRST (E) =
  FIRST (E') =
  FIRST (T) =
  FIRST (T') =
  FIRST (F) =

Compute FIRST(X):

If X is a terminal: FIRST(X) = {X}

If X is a nonterminal

$\varepsilon \in$ FIRST(X)

If $X \rightarrow \varepsilon$

If $X \rightarrow Y_1Y_2...Y_k$ and $\varepsilon \in$ FIRST(Y_i) for all 1 <= i <= k

$b \in$ FIRST(X)

If $X \rightarrow Y_1Y_2...Y_k$ and $b \in$ FIRST(Y_i) and $\varepsilon \in$ FIRST(Y_j) for all 1 <= j <= i-1

FOLLOW set

- Definition
  - FOLLOW (X) = {b | S $\rightarrow^*$ βXbδ}
  - FOLLOW (X) to be the set of terminals b that can appear immediately to the right of X in some sentential form.
- Compute FOLLOW(X)
  - $ $\in$ FOLLOW(S), where S is the start symbol
  - For each production A $\rightarrow$ αXβ
    - FIRST(β) – {ε} $\subseteq$ FOLLOW(X)
  - For each production A $\rightarrow$ αXβ where $\varepsilon\in$FIRST(β)
    - FOLLOW(A) $\subseteq$ FOLLOW(X)

FOLLOW set example

- Recall the grammar
  $E \rightarrow T\ F$
  $F \rightarrow +\ E \mid \varepsilon$
  $T \rightarrow \text{int}\ Y \mid (\ E\ )$
  $Y \rightarrow *\ T \mid \varepsilon$

- FOLLOW(E) =
  FOLLOW(F) =
  FOLLOW(T) =
  FOLLOW(Y) =

- FOLLOW(+) =
  FOLLOW(int) =
  FOLLOW(() =
  FOLLOW(*) =
  FOLLOW()) =

Compute FOLLOW(X)

- $ ∈ FOLLOW(S), where S is the start symbol
- For each production $A \rightarrow \alpha X \beta$
  $FIRST(\beta) - \{\varepsilon\} \subseteq FOLLOW(X)$
- For each production $A \rightarrow \alpha X \beta$ where $\varepsilon \in FIRST(\beta)$
  $FOLLOW(A) \subseteq FOLLOW(X)$

FOLLOW set example

- Consider the grammar

  stmt → *if-stmt* | **other**

  if-stmt → **if** (*exp*) *stmt else-part*

  else-part → **else** *stmt* | ε

  exp → 0 | 1

- FOLLOW(exp) =
  FOLLOW(else-part) =
  FOLLOW(if-stmt) =
  FOLLOW(stmt) =

Compute FOLLOW(X)

- $ ∈ FOLLOW(S), where S is the start symbol
- For each production $A \rightarrow \alpha X \beta$
  $FIRST(\beta) - \{\varepsilon\} \subseteq FOLLOW(X)$
- For each production $A \rightarrow \alpha X \beta$ where $\varepsilon \in FIRST(\beta)$
  $FOLLOW(A) \subseteq FOLLOW(X)$

LL(1) grammars

- Predictive parsers
  - is recursive-descent parsers needing no backtracking
  - can be constructed for a class of grammars called LL(1)
- LL(1)
  - First L stands for scanning the input from left to right
  - Second L stands for producing a leftmost derivation
  - 1 stands for using one input symbol of lookahead at each step to make parsing action decisions.
- LL(1) grammars are not ambiguous and not left recursive.

LL(1) grammars

- Predictive parsers can be constructed for LL(1) grammars
  - Since the proper production to apply for a nonterminal can be selected by looking only at the current input symbol.
  - Flow-of-control constructs, with their distinguishing keywords, generally satisfy the LL(1) constraints.

stmt → **if** (*expr*) *stmt* **else** *stmt*
| **while** (*expr*) *stmt*
| { *stmt_list* }

LL(1) Parsing Table

- Construct a parsing table M for context free grammar G.
  - M is a two-dimensional array.

- For each production $A \rightarrow \alpha$ in G do:
  - For each terminarl $t \in FIRST(\alpha)$ do
    - $M[A, t] = \alpha$
  - If $\varepsilon \in FIRST(\alpha)$, for each $t \in FOLLOW(A)$ do
    - $M[A, t] = \alpha$
  - If $\varepsilon \in FIRST(\alpha)$ and $\$ \in FOLLOW(A)$ do
    - $M[A, \$] = \alpha$

LL(1) Parsing Table

- Given left-factored grammar

```
E  -> TE'
E' -> +TE'|ε
T  -> FT'
T' -> *FT' | ε
F  -> (E) | id
```

- For each production $A \rightarrow \alpha$ in G do:
  - For each terminarl $t \in FIRST(\alpha)$ do
    - $M[A, t] = \alpha$
  - If $\varepsilon \in FIRST(\alpha)$, for each $t \in FOLLOW(A)$ do
    - $M[A, t] = \alpha$
  - If $\varepsilon \in FIRST(\alpha)$ and $\$ \in FOLLOW(A)$ do
    - $M[A, \$] = \alpha$

NON-TERMINAL	INPUT TOKEN					
	id	+	*	(	)	$
E	TE'			TE'		
E'		+TE'			ε	ε
T	FT'			FT'		
T'		ε	*FT'		ε	ε
F	id			(E)		

LL(1) Parsing Table

```
E  -> TE'
E' -> +TE'|ε
T  -> FT'
T' -> *FT' | ε
F  -> (E) | id
```

NON-TERMINAL	INPUT TOKEN					
	id	+	*	(	)	$
E	TE'			TE'		
E'		+TE'			ε	ε
T	FT'			FT'		
T'		ε	*FT'		ε	ε
F	id			(E)		

- Consider the `[E, id]` entry
  - When current non-terminal is `E` and next input is `id`, use production `E -> TE'`
  - This can generate an `id` in the first position

LL(1) Parsing Table

```
E  -> TE'
E' -> +TE'|ε
T  -> FT'
T' -> *FT' | ε
F  -> (E) | id
```

NON-TERMINAL	INPUT TOKEN					
	id	+	*	(	)	$
E	TE'			TE'		
E'		+TE'			ε	ε
T	FT'			FT'		
T'		ε	*FT'		ε	ε
F	id			(E)		

- Consider the [T', +] entry
  - "When current non-terminal is T' and current token is +, get rid of T' "
  - T' can be followed by + only of T' -> ε

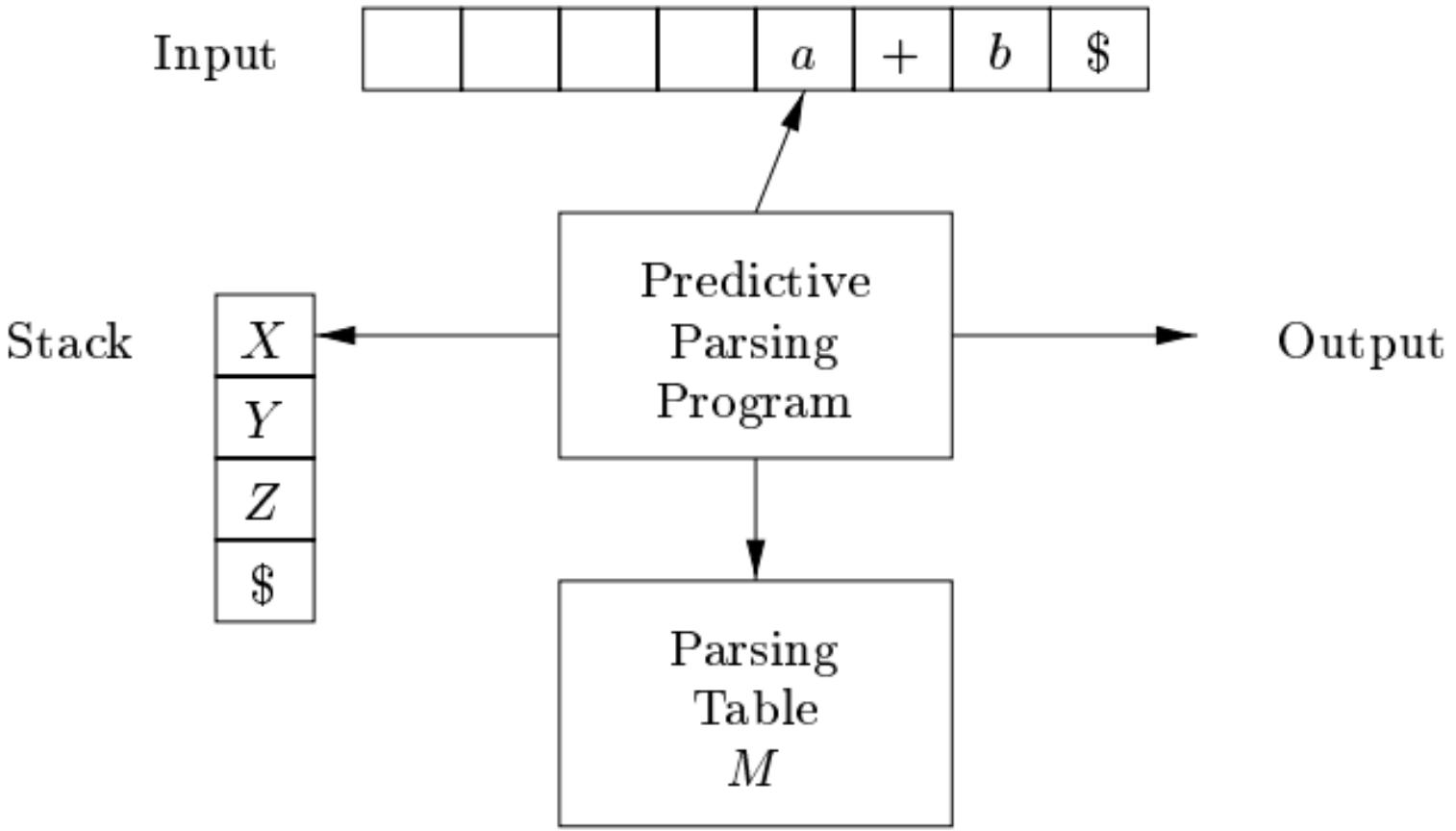
LL(1) Parsing Table

```
E  -> TE'
E' -> +TE'|ε
T  -> FT'
T' -> *FT' | ε
F  -> (E) | id
```

NON-TERMINAL	INPUT TOKEN					
	id	+	*	(	)	$
E	TE'			TE'		
E'		+TE'			ε	ε
T	FT'			FT'		
T'		ε	*FT'		ε	ε
F	id			(E)		

- Consider the [F, *] entry
  - "There is no way to derive a string starting with * from non-terminal F"
  - Blank entries indicate error situations

Using Parsing Tables



Model of a table-driven predictive parser

Table-driven predictive parsing algorithm

- INPUT: a string w and a prsing table M for grammar G
- OUTPUT:
  - If w is in L(G): a leftmost derivation of w,
  - Otherwise, an error indication

- METHOD
  - the parser is in a cofiguration with $w\$$ in the input buffer
  - and the start symbol S of G on top of the stack, above $
  - Follow program uses the predictive parsing table M to produce a predictive parse for the input

```
let a be the first symbol of w;
let X be the top stack symbol;
while ( X ≠$ ) f /* stack is not empty */
     if ( X = a ) pop the stack and let a be the next symbol of w;
     else if ( X is a terminal ) error();
     else if ( M [X, a] is an error entry ) error();
     else if ( M [X, a] = Y1Y2…Yk ) {
                  output the production X→Y1Y2…Yk;
                  pop the stack;
                  push Yk,Yk-1,…,Y1 onto the stack, with Y1 on top;
     }
     let X be the top stack symbol;
}
```

Example

- Given left-factored grammar

```
E  -> TE'
E' -> +TE'|ε
T  -> FT'
T' -> *FT' | ε
F  -> (E) | id
```

- Input: `id + id * id`

NON-TERMINAL	INPUT TOKEN					
	id	+	*	(	)	$
E	TE'			TE'		
E'		+TE'			ε	ε
T	FT'			FT'		
T'		ε	*FT'		ε	ε
F	id			(E)		

MATCHED	STACK	INPUT	ACTION
	E$	id+id*id$	
	TE'$	id+id*id$	Output E -> TE'
	FT'E'$	id+id*id$	Output T->FT'
	idT'E'$	id+id*id$	Output F->id
id	T'E'$	+id*id$	Match id
id	E'$	+id*id$	Output T'->ε
id	+TE'$	+id*id$	Output E'->+TE'
id +	TE'$	id*id$	Match +
id +	FT'E'$	id*id$	Output T->FT'
id +	idT'E'$	id*id$	Output F->id
id + id	T'E'$	*id$	Match id
id + id	*FT'E'$	*id$	Output T'->*FT'
id + id *	FT'E'$	id$	Match *
id + id *	idT'E'$	id$	Output F->id
id + id * id	T'E'$	$	Match id
id + id * id	E'$	$	Output T'->ε
id + id * id	$	$	Output E'->ε

LL(1) Parsing Table

- For each production $A \rightarrow \alpha$ in G do:
  - For each terminarl $t \in FIRST(\alpha)$ do
    - M[A, t] = α
  - If $\varepsilon \in FIRST(\alpha)$, for each $t \in FOLLOW(A)$ do
    - M[A, t] = α
  - If $\varepsilon \in FIRST(\alpha)$ and $\$ \in FOLLOW(A)$ do
    - M[A, $] = α

- Consider the grammar: $S \rightarrow Sa \mid b$
- The parsing table

	a	b	$
S		b Sa	

=> This grammar is not LL(1)

LL(1) Parsing Table

- If any entry is multiply defined then G is not LL(1)
  - Any grammar is not left factored, will not be LL(1)
  - Any grammar is left recursive, will not be LL(1)
  - Any grammar is ambiguous, will not be LL(1)
  - Grammar required more than one token of lookahead, will not be LL(1)
  - Other grammars are not LL(1) too

⇒Mechanical way to check that the grammar is LL(1)

⇒Build the LL(1) parsing table

⇒And see if all the entries in the table is unique

- Most programming language CFGs are not LL(1)
  - The LL(1) grammars are too weak to actually capture all of the interesting and important constructs in commonly using programming languages.

Exercise

- Consider left-factored grammar
  $E \rightarrow T\ F$
  $F \rightarrow +\ E \mid \varepsilon$
  $T \rightarrow int\ Y \mid (\ E\ )$
  $Y \rightarrow *\ T \mid \varepsilon$

- Show the parsing table
- Devise predictive parsers

- For each production $A \rightarrow \alpha$ in G do:
  - For each terminarl $t \in FIRST(\alpha)$ do
    - $M[A, t] = \alpha$
  - If $\varepsilon \in FIRST(\alpha)$, for each $t \in FOLLOW(A)$ do
    - $M[A, t] = \alpha$
  - If $\varepsilon \in FIRST(\alpha)$ and $\$ \in FOLLOW(A)$ do
    - $M[A, \$] = \alpha$

If any entry is multiply defined then G is not LL(1)
Any grammar is not left factored, will not be LL(1)
Any grammar is left recursive, will not be LL(1)
Any grammar is ambiguous, will not be LL(1)
Grammar required more than one token of lookahead, will not be LL(1)
Other grammars are not LL(1) too

Cho văn phạm G:

S -> S + A | A

A -> A – B | B

B -> (S) | a

1. Biến đổi văn phạm G về văn phạm LL(1)
2. Xây dựng bảng phân tích cú pháp (parsing table) cho văn phạm LL(1) ở câu 1.
3. Vẽ cây phân tích cú pháp cho chuỗi (a - a) dựa trên văn phạm LL(1) ở câu 1.