Preface

This Compiler Science tutorials based on Compilers course of Stanford University. At first, appreciate the Stanford University share the course free of charge with the global, let other nations knowledge-thirsty student learn from it. This is why so great school all the universe. Secondly, this tutorial ONLY maintains "Lexical Analysis" and "Parsing" section, the other section hope another student accomplish for us. It's also a great guilty NOT whole going through the Compile whole procedure, ONLY interact the first 2 sections. Finally, enjoy this DOC free of charge.

By atie-M2G

This drawn presentation 1st released on GitHub.

GitHub Repo.: https://github.com/atie-m2g/Compiler.git;

DOC URI: https://github.com/atie-m2g/Compiler/blob/main/Compiler_Science_Tutorials.pdf;

Compilers — Stanford University

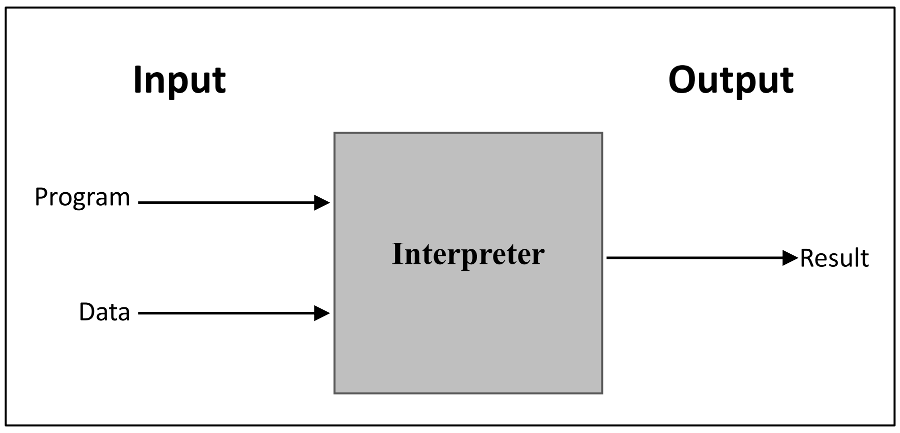
1 Introduction

We know that program written by programing language, and compile the language getting an executable which can be run on system. That means programs implemented by programing language. How does the programing language implemented? Who set the programing language wring style? How does the implementation of programing language?

There're two major approaches to implementing programing languages. And they're **Compilers** and **Interpreters**. This course more about compilers. But also talk a little bit about interpreters at the begging lecture parts.

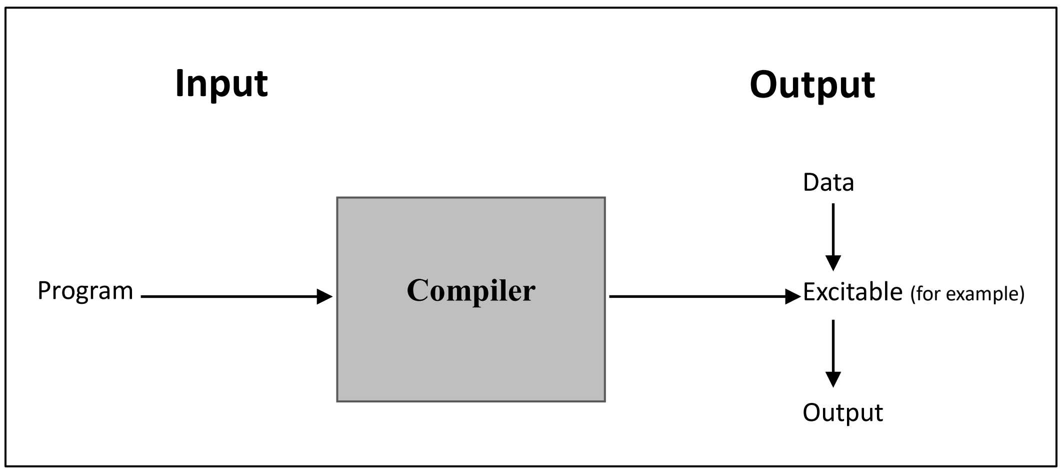
So, what does interpreter do? also, the compiler?

Let's talk about interpreter first. Interpreter also a program, ONLY interpretation for a specified single programing language. It takes program (source code) and the data as an input, and it produces output directly, which means that it doesn't do any processing of the program (source code) before it executes. As long as coding finishes, the interpreter invoke code with the data, and then the program immediately begins running. So, can say that the interpreter is "Online", which means the work that it does is all part of running.



[Figure 1-1] Interpreter Working Flow

But the compiler structured differently. Compiler takes program (source code) as an input, and produces an executable, is another program; another program maybe Assembly language OR bytecode OR machine-code (executable program), also could be any number of different implementation languages. And the outputted executable COULD be run, separated with the data, it WOULD produce output if run.

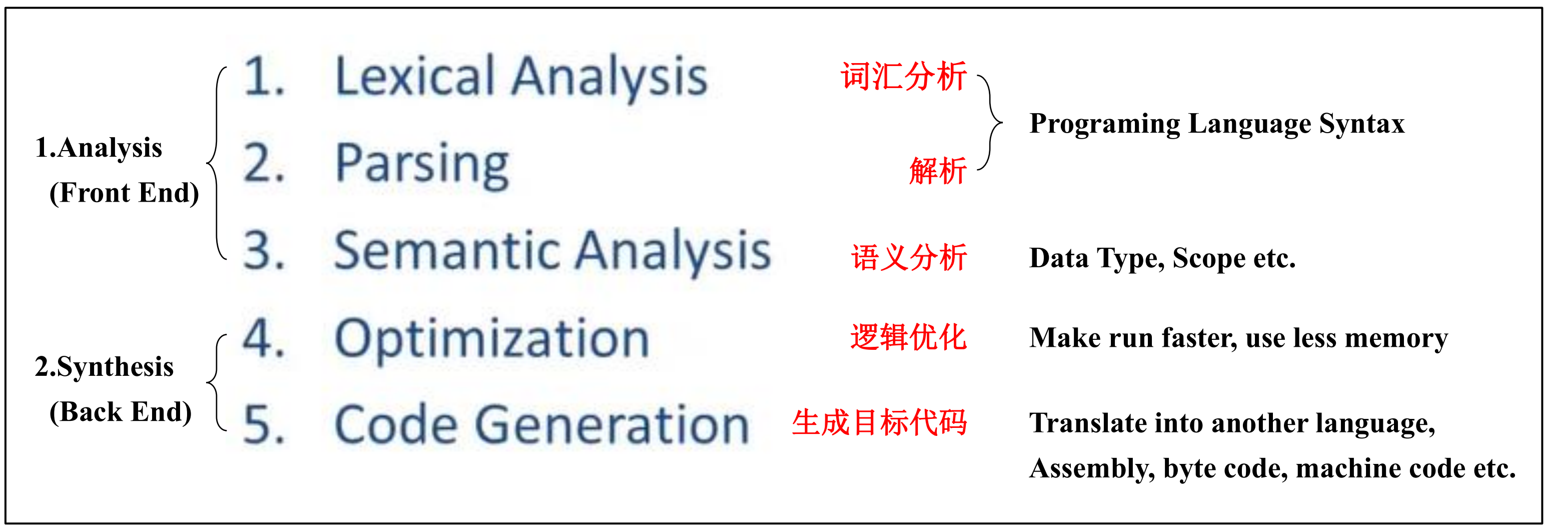


[Figure 1-2] Compiler Working Flow

1.1 Compiler Overview

1.1.1 The First Compiler Structure

The first high level programing language Fortran and the early Compiler Structure of Fortran still preserves nowadays compilers. So, what is the Fortran Compiler structure?



1.Analysis (Front End)

1. Lexical Analysis 词汇分析
2. Parsing 解析

Programing Language Syntax

3. Semantic Analysis 语义分析 Data Type, Scope etc.

2.Synthesis (Back End)

4. Optimization 逻辑优化 Make run faster, use less memory
5. Code Generation 生成目标代码 Translate into another language, Assembly, byte code, machine code etc.

[Figure 1-3] Classic Compiler Inner Structure

1.1.2 Compiler Profiling

Just like as above the First Compiler Structure consist of **2 big chapter 5 small phases**. And they're:

1) Lexical Analysis

   A text (sentences) separates into some words.

2) Parsing

   Diagramming sentence, make clear words rule in sentence, which word is object, which one is subject etc. and make a diagram statistic with the words rule.

   The two components (Lexical Analysis & Parsing) mission is **Programing Language Syntax**, to detect the written code by user is legal or not.

3) Semantic Analysis

   The Data Type, Size of The Data Memory, The Data Limits etc.

4) Optimization

   Make program slimmer, run faster, use less memory.

5) Code Generation

   Generate targeted program language. The target language NOT ONLY Assembly, byte code OR Machine Code, MAYBE another high-level language (C++ translate into C, for instance).

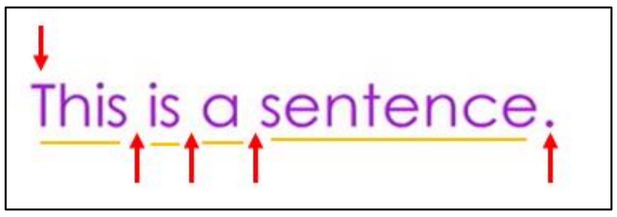
But how compiler understand the analogy written by. This is very similar how human understands a sentence. Think a sentence exists below, how to understand it?

This is a sentence.

[Figure 1-4] A Sentence for Mankind

1. First step: **Recognize words**.

Very easy to understand the sentence, but how we did it. We look up the blanks, punctuation like periods and capital, these carets make sperate the sentence into words.



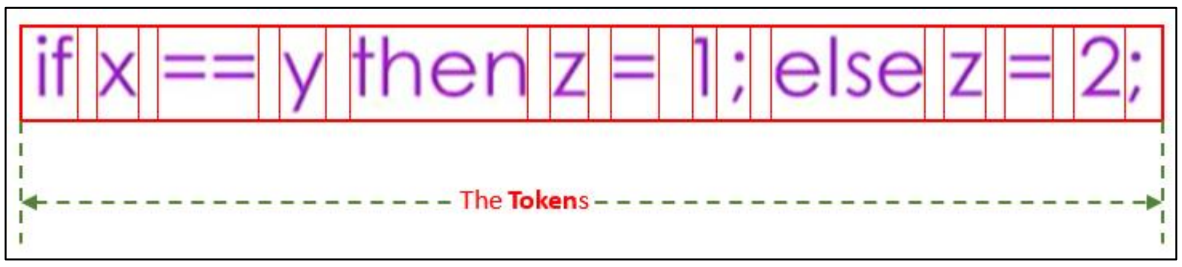
This is a sentence.

[Figure 1-5] Words Split from A Sentence by Mankind

The first component of Compiler Structure named **Lexical Analysis** divide the program text (sentences) into words, and the words called the **Token**s in Compiler domain.

Lexical analysis divides program text into “words” or “tokens”

[Figure 1-6] Term Tokens in Compiler



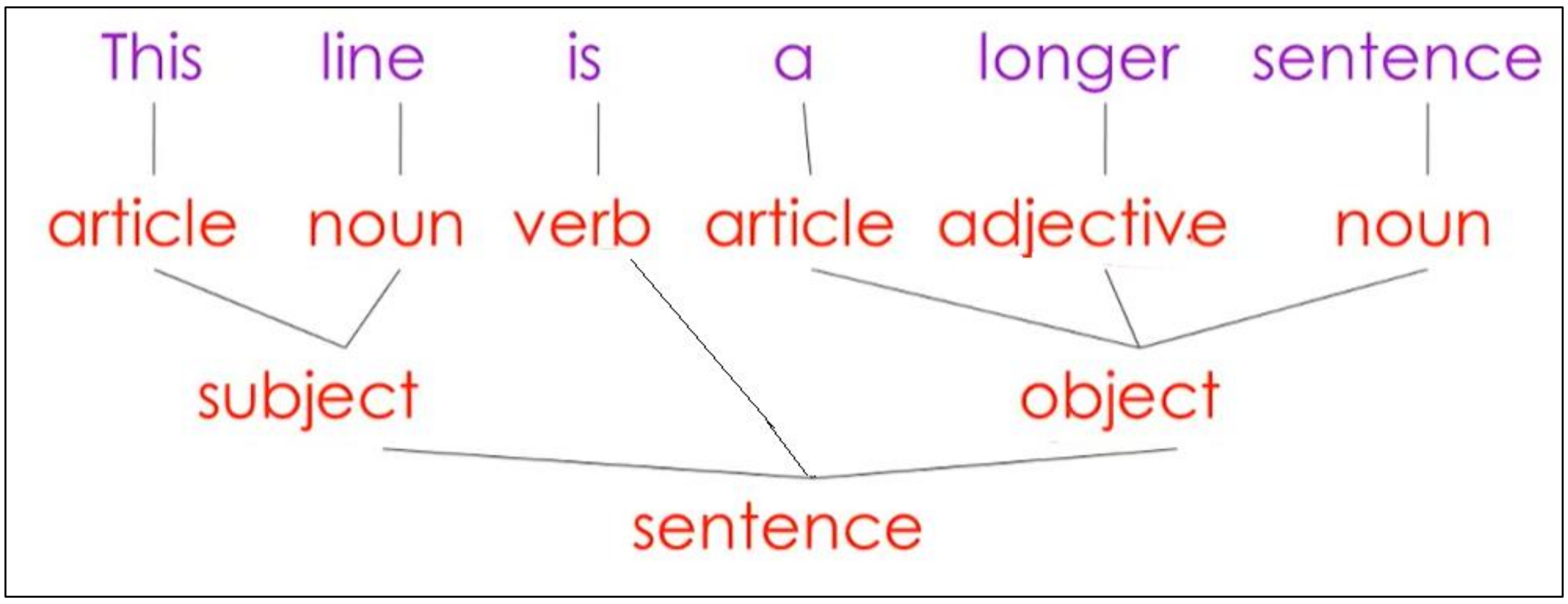
[Figure 1-7] Tokens in Compiler on if-then-else Statement

2. Second step: **Parsing**

Once words (Tokens) known, the next step is to understand sentence ($sencense \in text$) structure, and this called Parsing.

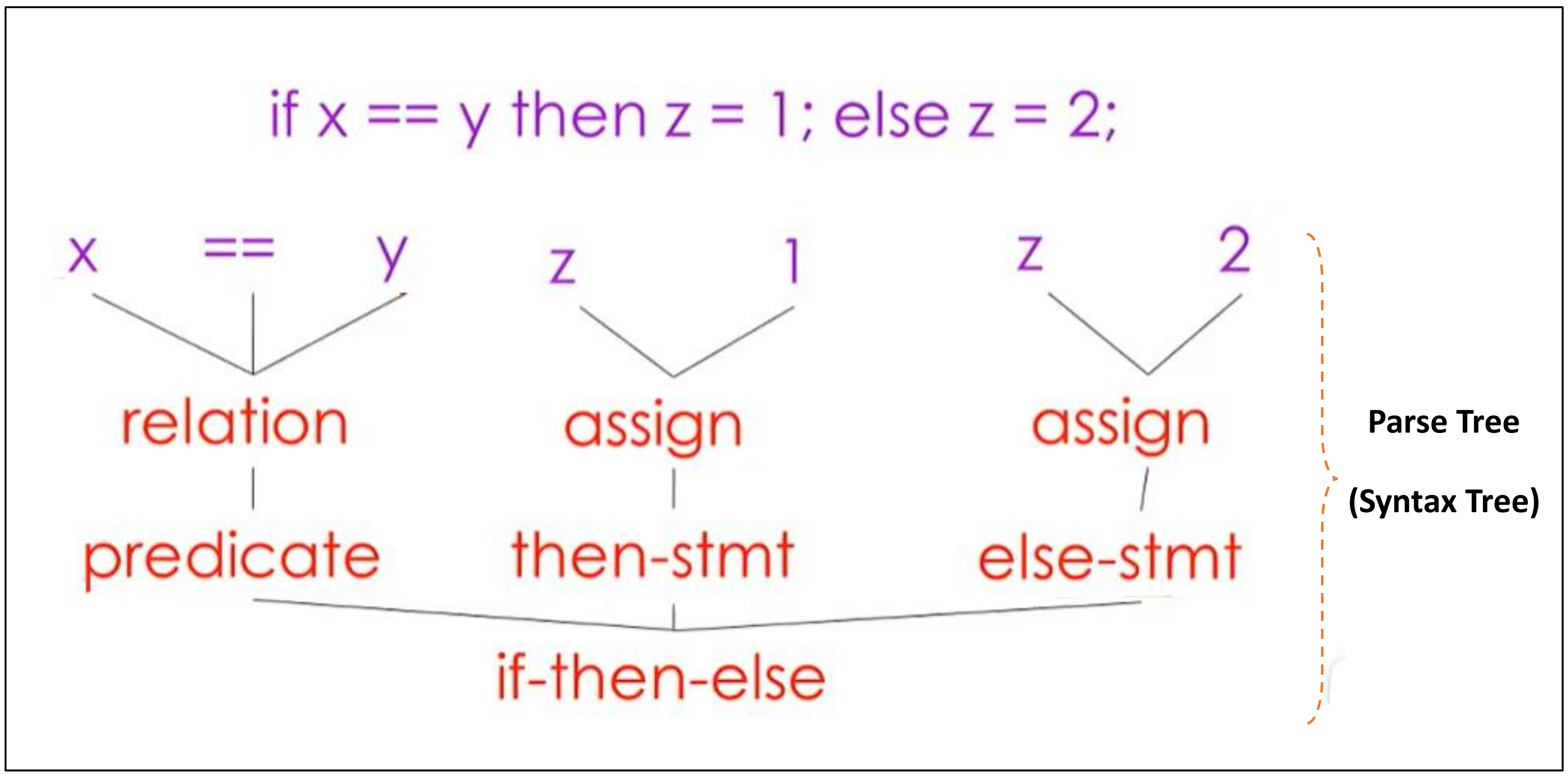
Parsing = Diagramming Sentences

Diagramming Sentences (Parsing), the diagram is a tree. The first step in Parsing is to identify the role of each word of a sentence. The figure below for human how understand a sentence structure after Lexical Analysis, more spiffily in Parsing period.



[Figure 1-8] Diagramming A Sentence

The figure below how a sentence of the programing language composed, and how to understand the structure of the sentence?



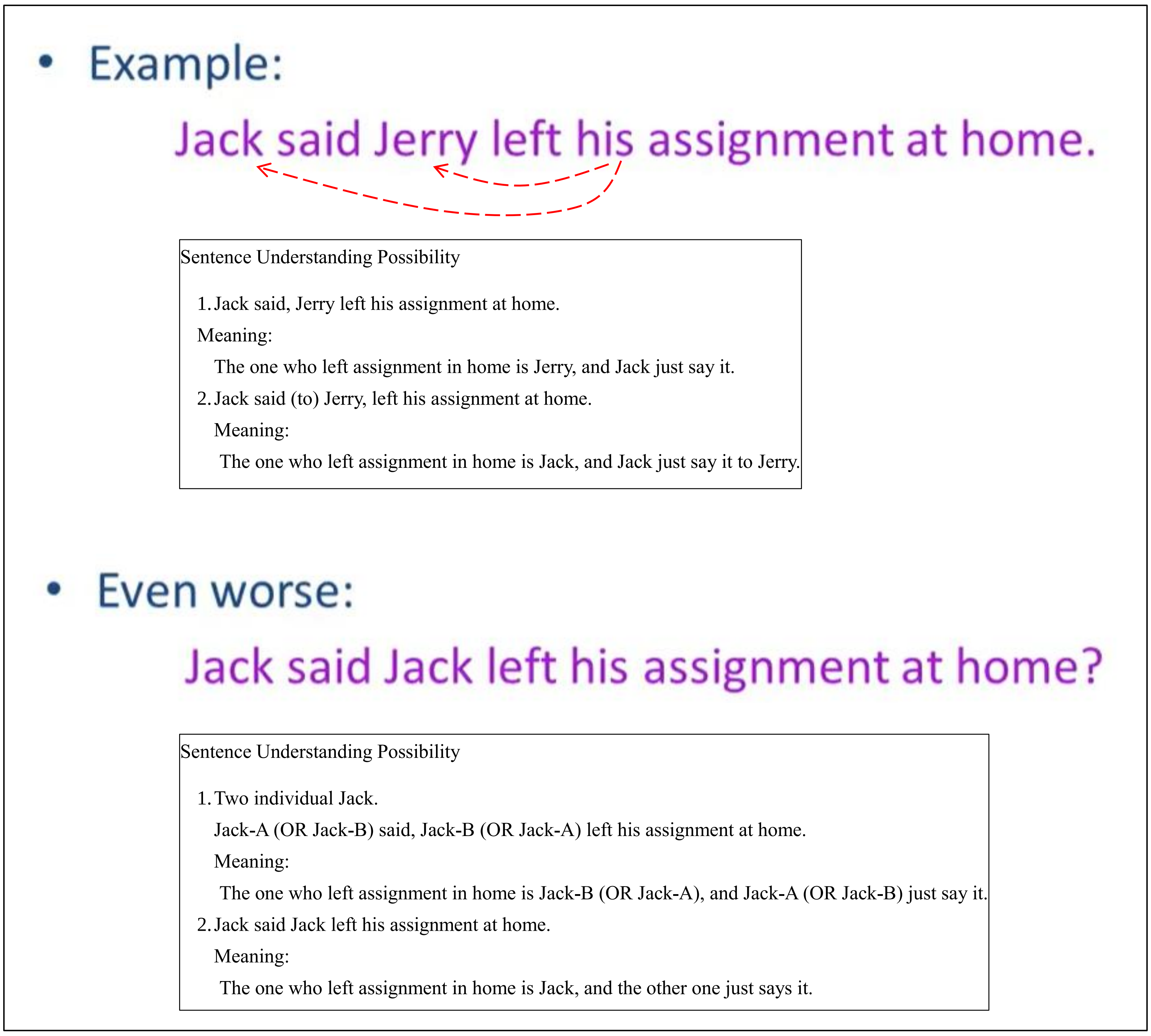
[Figure 1-9] A Parsing Tree in Compiler

3. Third step: Semantic Analysis

Once sentence structure is understood (in other words, after Parsing Tree shown), we can try to understand "meaning", what has been written; but this is hard. Compilers perform limited **Semantic Analysis** to catch inconsistencies.

After **Lexical Analysis** and **Parsing** for human, we still don't know how it works. But frankly, understanding the meaning something is simply too hard for Compilers. So, the first important thing understanding about **Semantic Analysis** is that Compilers can only do very limited kinds of Semantic Analysis. And, in particular, the kinds of things that Compilers generally do are try to catch inconsistencies. So, when program somehow self-inconsistent, Compilers can often notice that and report errors. But Compiler really don't know what the program is supposed to do.

Understand sentence always very hard, getting some ambiguous if not know the whole context, also for human. Just look down the two examples below.



- Example:

Jack said Jerry left his assignment at home.

Sentence Understanding Possibility

1. Jack said, Jerry left his assignment at home.

Meaning:

The one who left assignment in home is Jerry, and Jack just say it.

2. Jack said (to) Jerry, left his assignment at home.

Meaning:

The one who left assignment in home is Jack, and Jack just say it to Jerry.

- Even worse:

Jack said Jack left his assignment at home?

Sentence Understanding Possibility

1. Two individual Jack.

Jack-A (OR Jack-B) said, Jack-B (OR Jack-A) left his assignment at home.

Meaning:

The one who left assignment in home is Jack-B (OR Jack-A), and Jack-A (OR Jack-B) just say it.

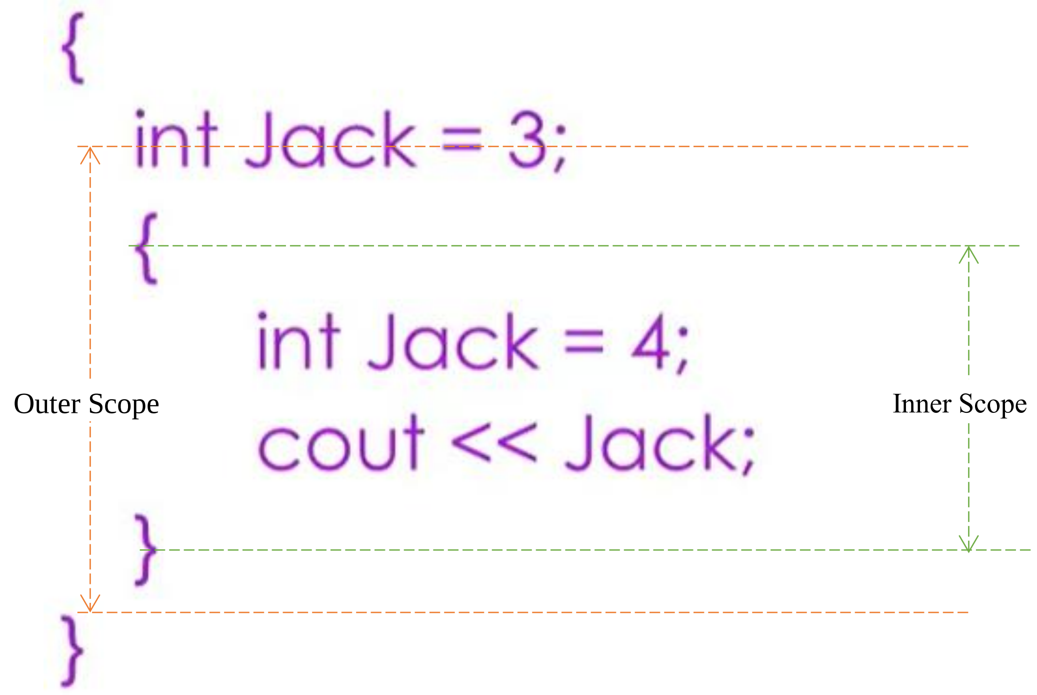
2. Jack said Jack left his assignment at home.

Meaning:

The one who left assignment in home is Jack, and the other one just says it.

[Figure 1-10] A Circes in Semantic Analysis: Ambiguousness

This ambiguous in programing language present as below style.



```
{
  int Jack = 3;
  {
      int Jack = 4;
      cout << Jack;
  }
}
```

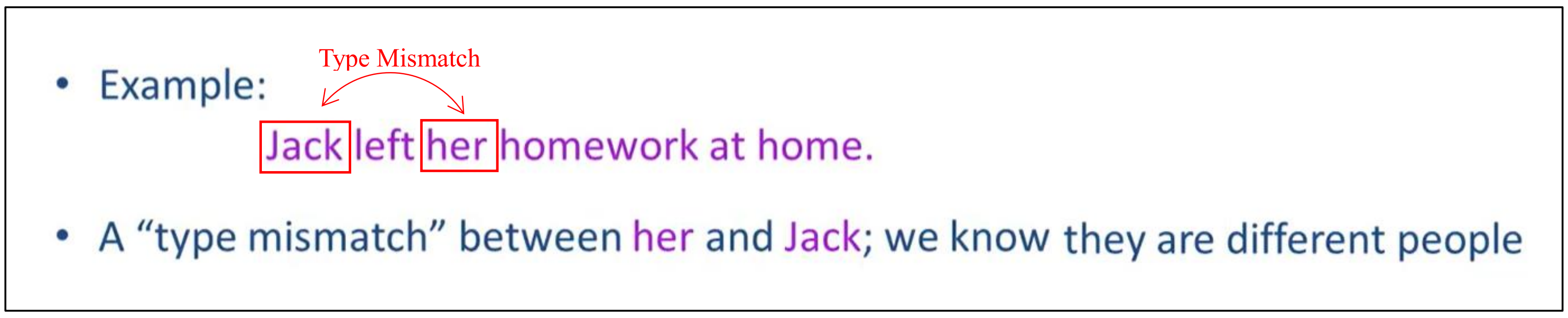
The ONLY way like above ambiguous on programming language can define this illegal expression OR making lexical scope, which valued Jack 3 as outer scope and hidden in inner scope where valued Jack 4.

[Figure 1-11] A Circes in Semantic Analysis Implementation

Note:

1. **Ambiguity is the really problem in Semantic Analysis.**
2. **The variable and its value (key-value) analogy in programming language called Variable Binding.**

Compilers perform many Semantic checks besides Variable Bindings, Data Type Check for an instance. The Data Type Check just like Subject and Object matches check in a sentence.



- Example:

  Jack left her homework at home.

- A "type mismatch" between her and Jack; we know they are different people

Type Mismatch

[Figure 1-12] A Circes in Semantic Analysis: Data Type Mismatch

4. Fourth step: Optimization

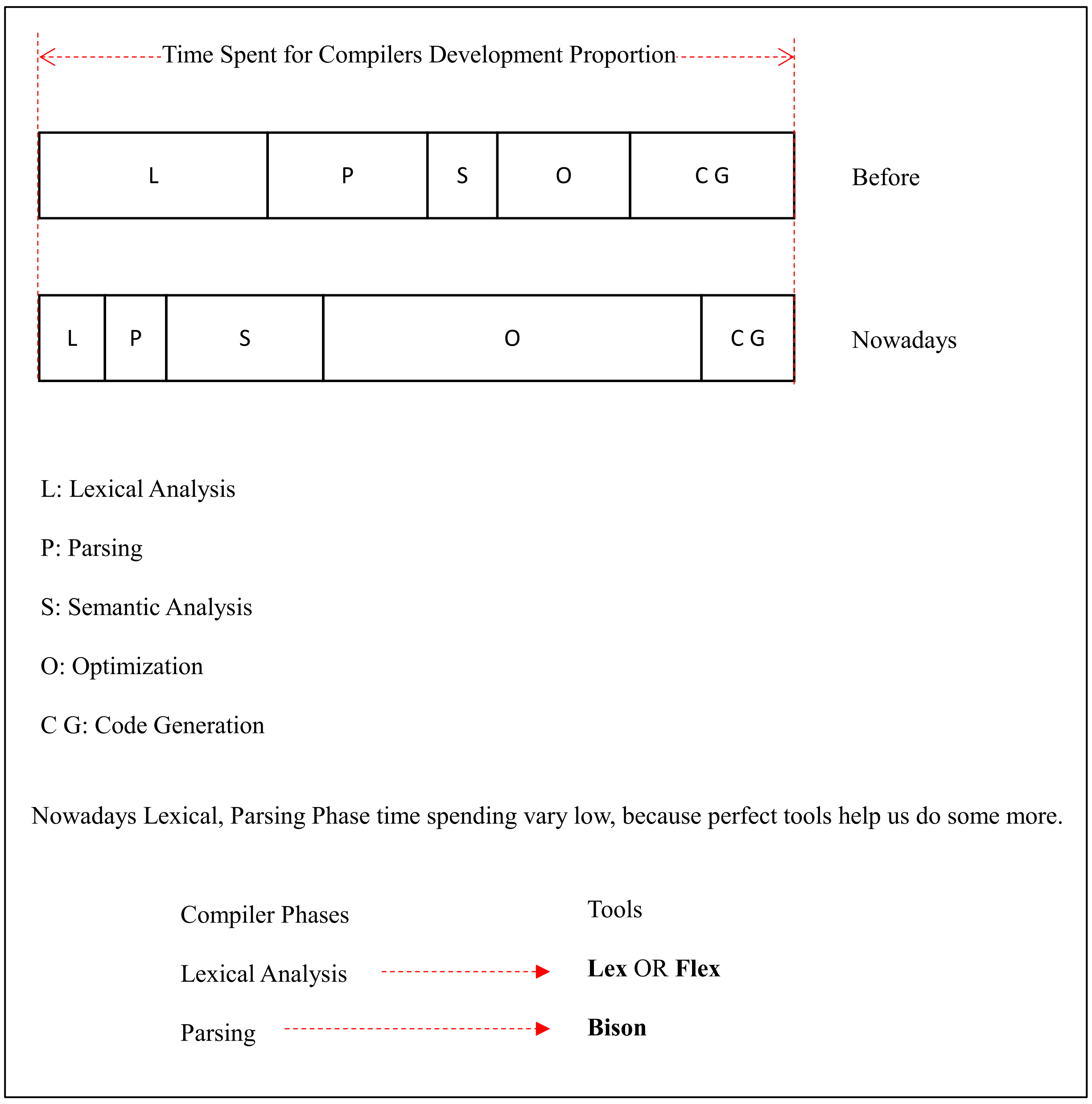
The Optimization a little bit like professional, export editing an article that getting the article un-bugged, the article words limitation, and words used high-level words not the simple ones. The Optimization in Compilers also like the same purpose. The purpose Optimization in Compilers as below, but the first two purpose is common ones, others be related to source program usage.

- Run faster
- Use less memory
- Reduce Power Consume
- Reduce Network Messages Commutations
- Reduce Data Base Accesses

5. Fifth step: Code Generation

The Code Generation, commonly, is to high-level language translate into Assembly Language OR another programming language.

If want to develop a Compiler, time spends every phase little different long before and nowadays.



L: Lexical Analysis

P: Parsing

S: Semantic Analysis

O: Optimization

C G: Code Generation

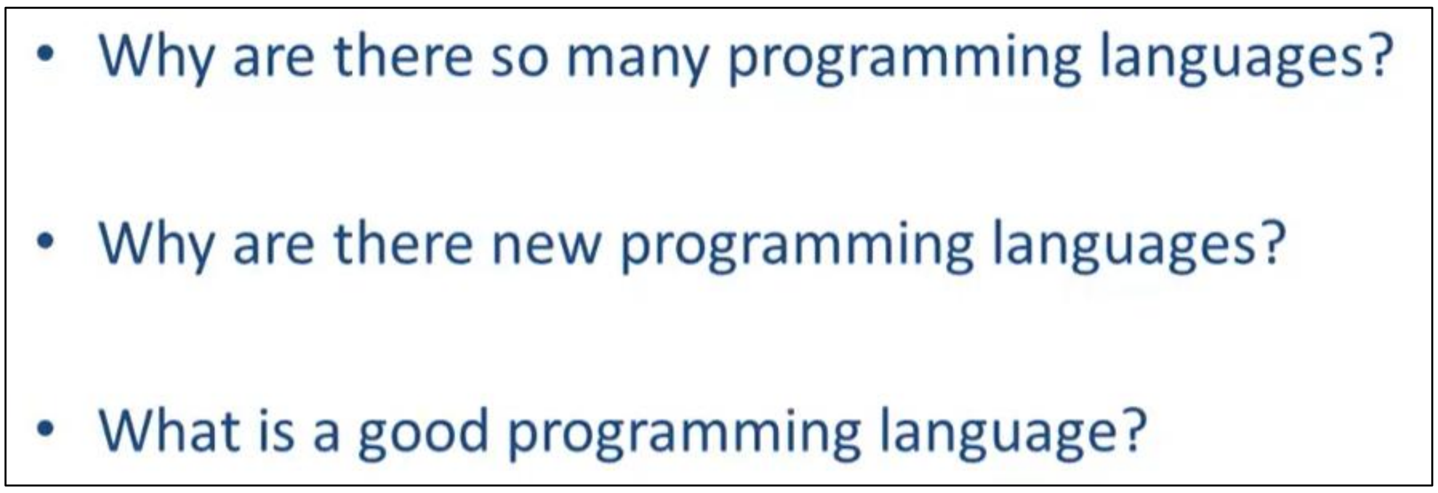
Nowadays Lexical, Parsing Phase time spending vary low, because perfect tools help us do some more.

Compiler Phases	Tools
Lexical Analysis	**Lex** OR **Flex**
Parsing	**Bison**

[Figure 1-13] Compiler Implementation Time Rate in Phases

1.1.3 The Economy of Programing Language

Lots of us MAY have some question about programing languages. Questions WOULD be like bellow, I believe.

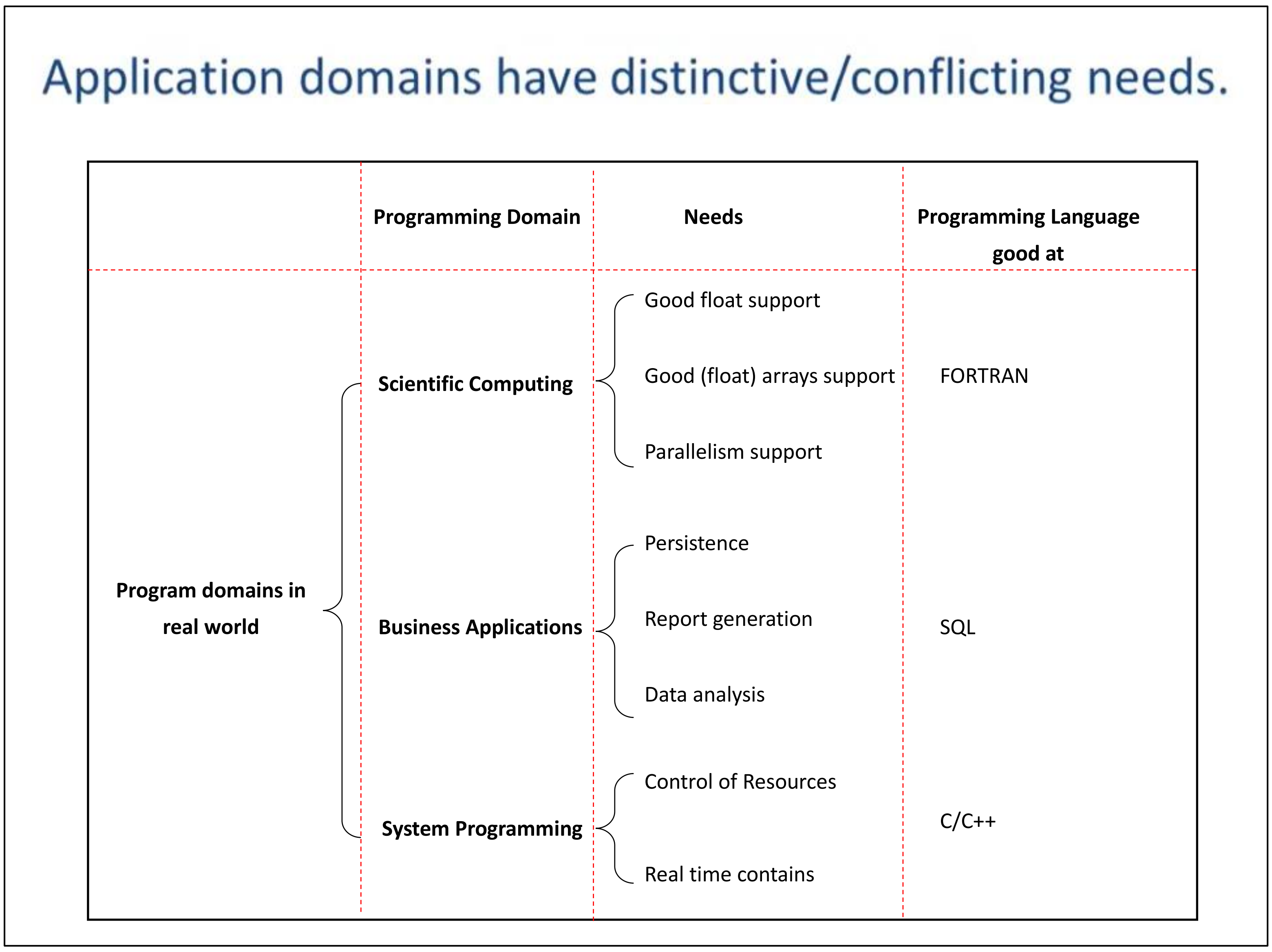


- Why are there so many programming languages?
- Why are there new programming languages?
- What is a good programming language?

[Figure 1-14] Typical Question of Programming Language

Let’s come to the first question: **Why are there so many programming languages**?

A programing language cann’t be export every aspect of real world. Real world got lots of needs and diffirent OR very unique, even worse confilicting needs. So, a programming language REALLY hard to statisfied that all aspect. Thus got some programming languages for special area, OR do something a little bit every aspect but not expert that areas. And this is why real world has a mount of programing languages and some of them ONLY for special area.



Application domains have distinctive/conflicting needs.

	Programming Domain	Needs	Programming Language good at
Program domains in real world	Scientific Computing	Good float support Good (float) arrays support Parallelism support	FORTRAN
	Business Applications	Persistence Report generation Data analysis	SQL
	System Programming	Control of Resources Real time contains	C/C++

[Figure 1-15] Programming Language Living Domain

So, different domain has very different needs that REALLY NOT to integrate. So here is the solution of question 1.

Let’s come to the second question: **Why are there new programming languages?**

Claim: Programmer training is the dominant cost for a programming language.

Programmer training is huge cost, so real world doesn’t wanna create new one EXCEPT programmer training fee less than new language productivity skills. So, if wanna new programming language MUST be training fee more less than new language productivity. And another view new language tends to old language, because it’s not easy to create a whole new thing, and the similarity to old language very helpful to reduce programmer training fee.

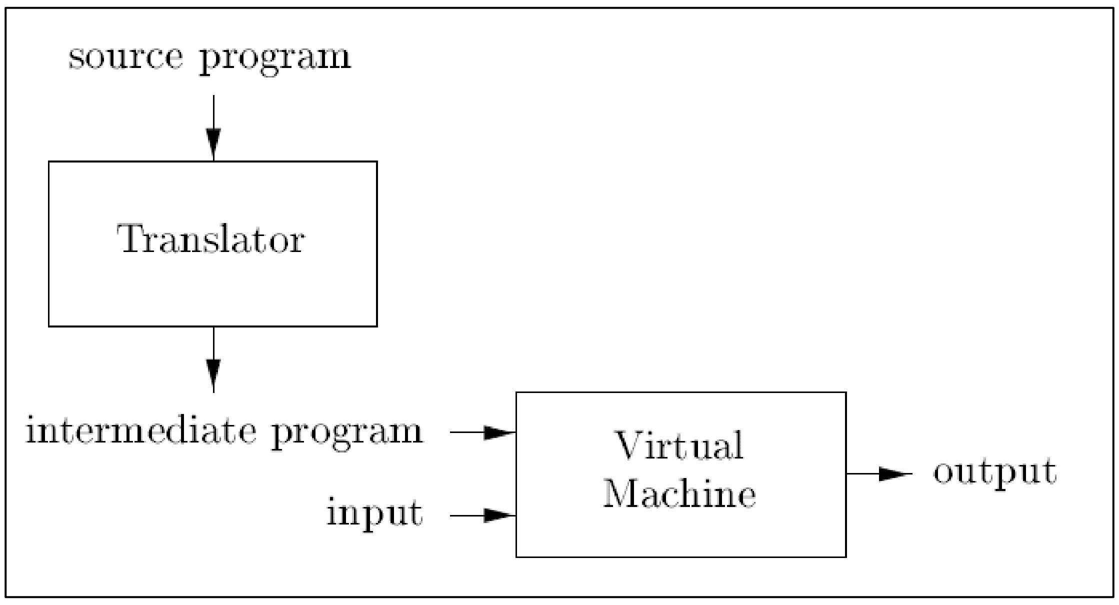
Let’s come to the third question: **What is a good programming language?**

There is NO universally accepted consensus on what makes a good language. If a programing language mostly used in real world, it doesn’t mean the language is a good one.

1.1.4 Compiler VS. Interpreter

#	Expert	
	Compiler	Interpreter
Run faster at input data map to output	✓	
Better error diagnostics		✓

1.1.5 Advanced Compile & Interpreter



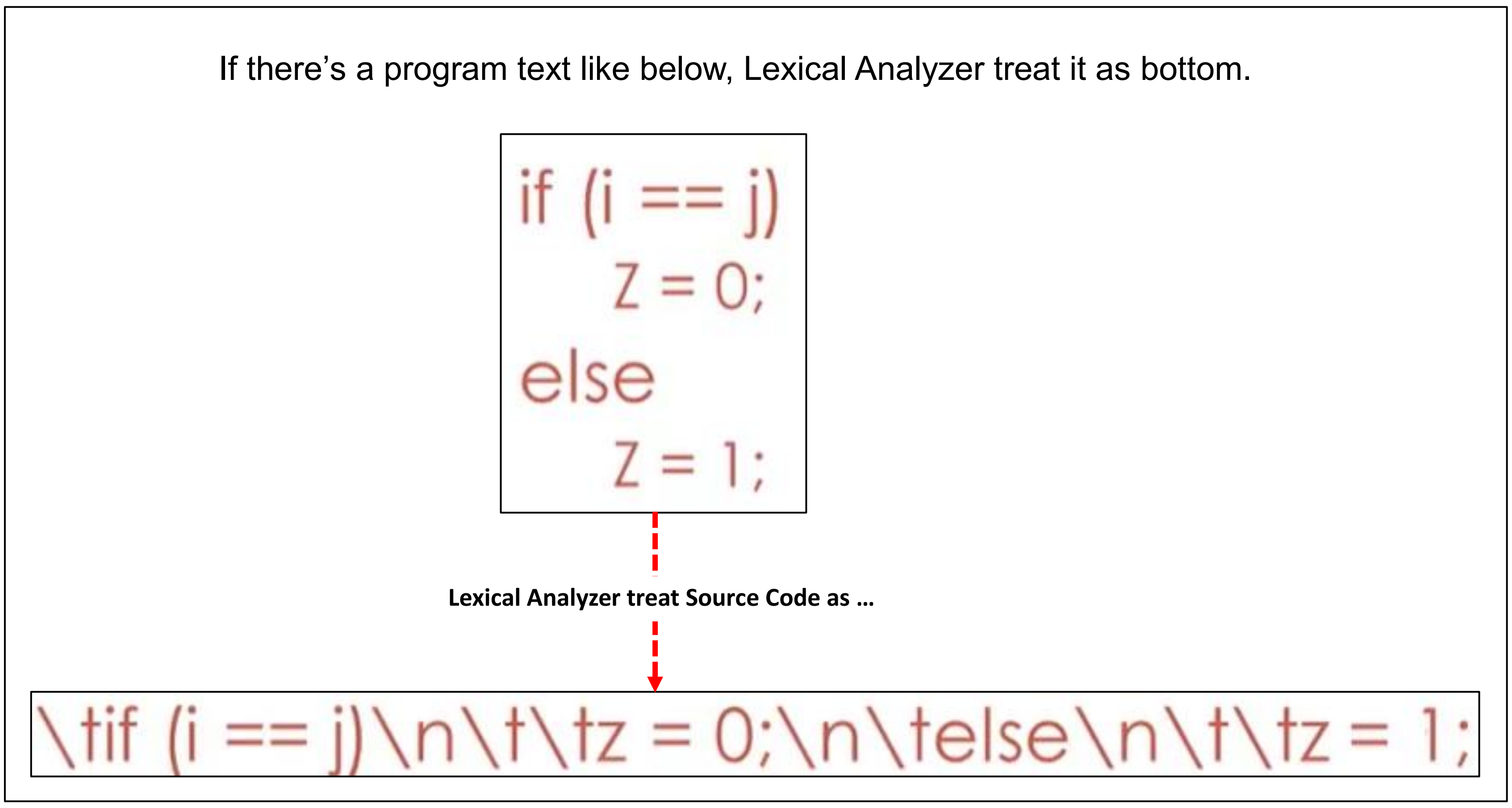
[Figure 1-16] Advanced Compiler Structure

2 Lexical Analysis

2.1 Introduce

Mission:

Program text (sentences) split into words OR Tokens in Complier speaking.



If there’s a program text like below, Lexical Analyzer treat it as bottom.

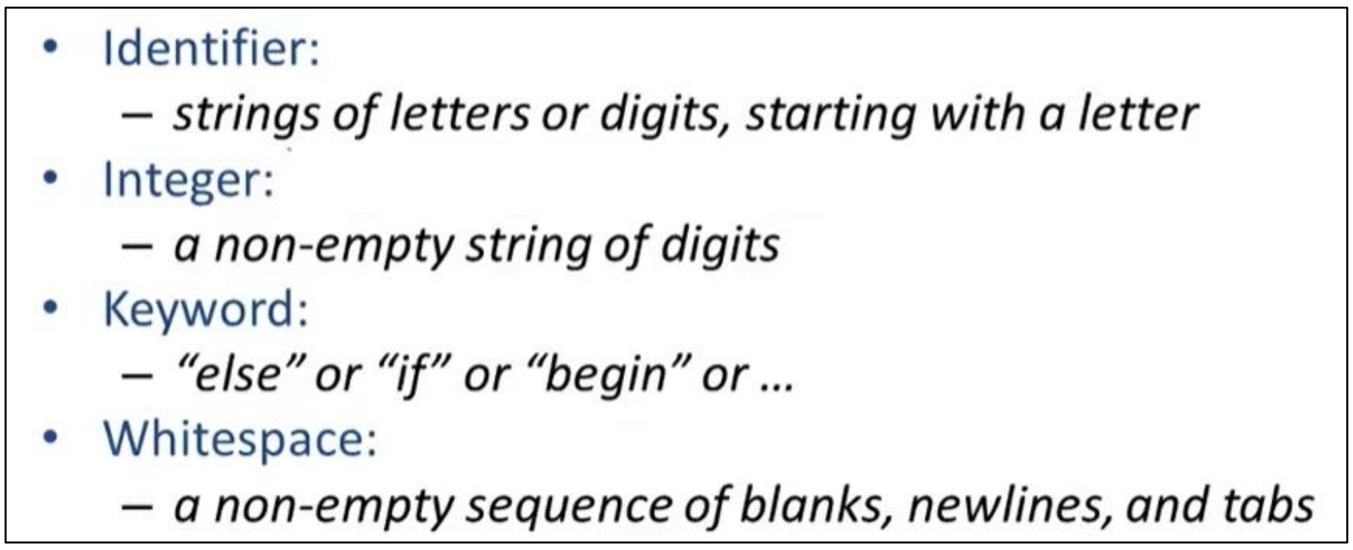
```
if (i == j)
    Z = 0;
else
    Z = 1;
```

Lexical Analyzer treat Source Code as …

```
\tif (i == j)\n\t\tz = 0;\n\telse\n\t\tz = 1;
```

[Figure 2-1] Point of View of Lexical Analysis Lookup Program Source Code

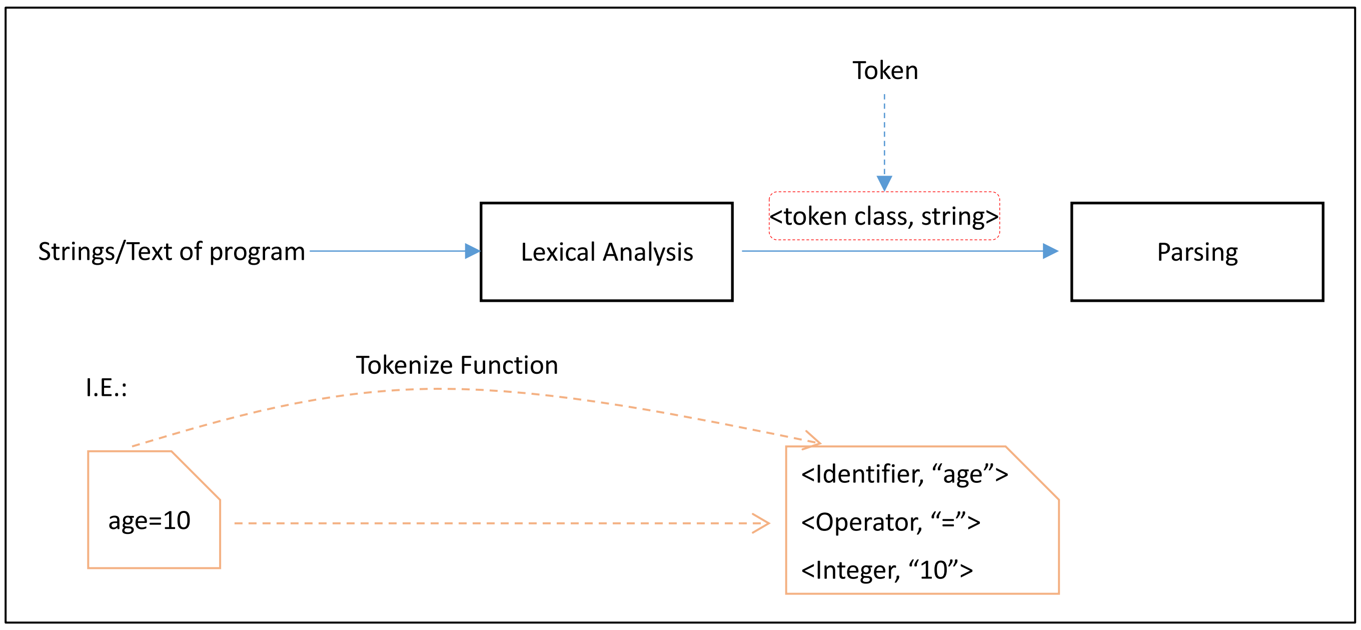
The Lexical Analysis can’t recognize the strings, it has to split it out lots of very basic units called Token Class (OR Class of Token), simply **Token**. In English, the Tokens are Non, Verb, Adj and more. But in programming, the Tokens are Identifier, Keywords, Brackets, Numbers and so on. You can see that **the Tokens OR Token Classes is sets of strings**.



- Identifier:
  – *strings of letters or digits, starting with a letter*
- Integer:
  – *a non-empty string of digits*
- Keyword:
  – *“else” or “if” or “begin” or …*
- Whitespace:
  – *a non-empty sequence of blanks, newlines, and tabs*

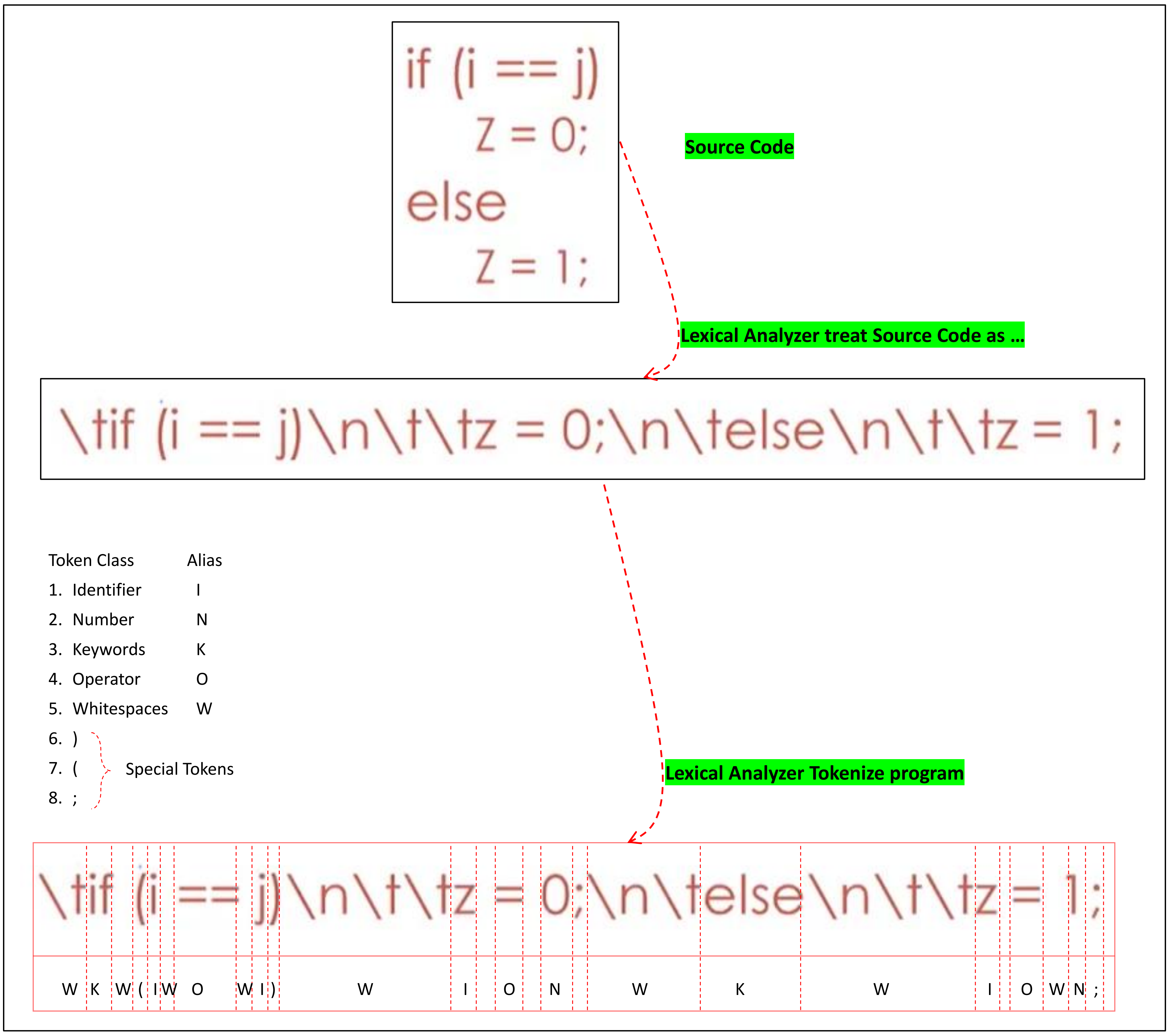
[Figure 2-2] Instances of Tokens

Lexical Analysis is to classify substring of the program (source code) according to their role. And the Tokens send to second phase Parsing in Compiler.



[Figure 2-3] Lexical Analysis IO

There's a whole perfect example how Tokenize the program text which is the mission of Lexical Analyzer.


if (i == j)
Z = 0;
else
Z = 1;
Source Code
Lexical Analyzer treat Source Code as …
\tif (i == j)\n\t\tz = 0;\n\telse\n\t\tz = 1;
Token Class
Alias
1. Identifier I
2. Number N
3. Keywords K
4. Operator O
5. Whitespaces W
6.)
7. (
8. ;
Special Tokens
Lexical Analyzer Tokenize program
\tif (i == j)\n\t\tz = 0;\n\telse\n\t\tz = 1;
W K W (I W O W I) W I O N W K W I O W N ;

[Figure 2-4] Lexical Analysis Mission: Tokenization

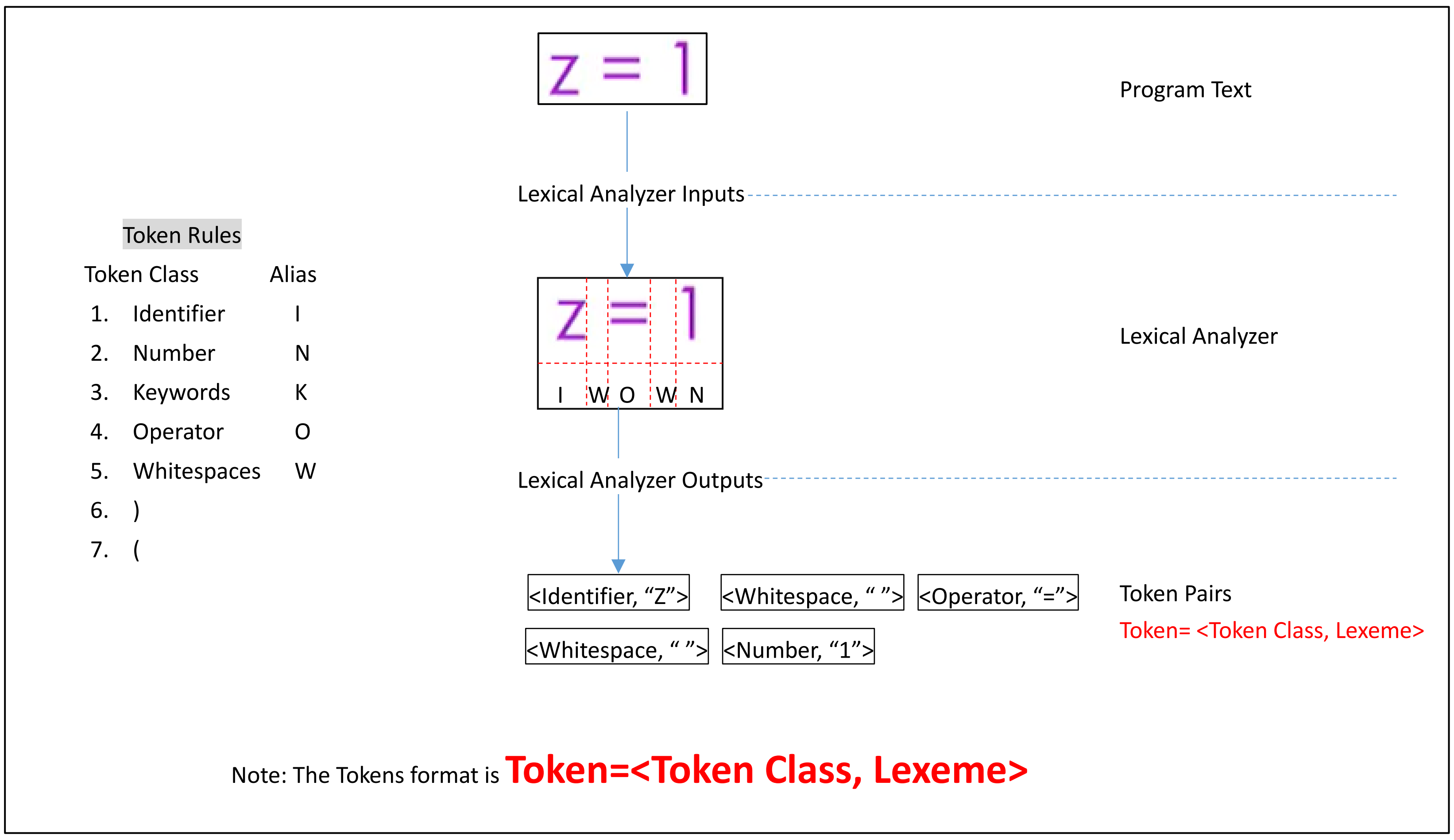
To summarize, the Lexical Analysis implementation MUST do two things.

1. Recognize corresponding the substrings into tokens.
    - The lexemes

      Lexeme is a Compiler lingo, which these substrings of Lexical Analyzer split off the programming text (sentences) to words, the words are called the Lexemes. And the Lexeme would become value of a Token later.

2. Identify the token class of each lexeme.

So, in the end the really output of Lexical Analyzer is Token. And the Token consist of Token Class and Lexeme.

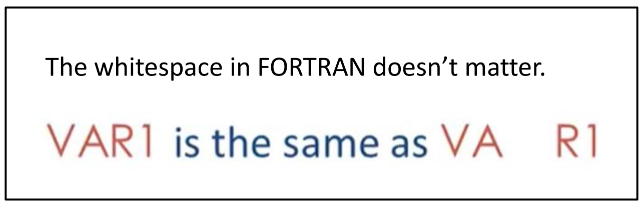


[Figure 2-5] Lexical Analysis Terms: Token, Token Class, Lexeme

2.2 Lexical Analyzer Example

1. FORTRAN Demo

FORTRAN Rule: Whitespace is insignificant

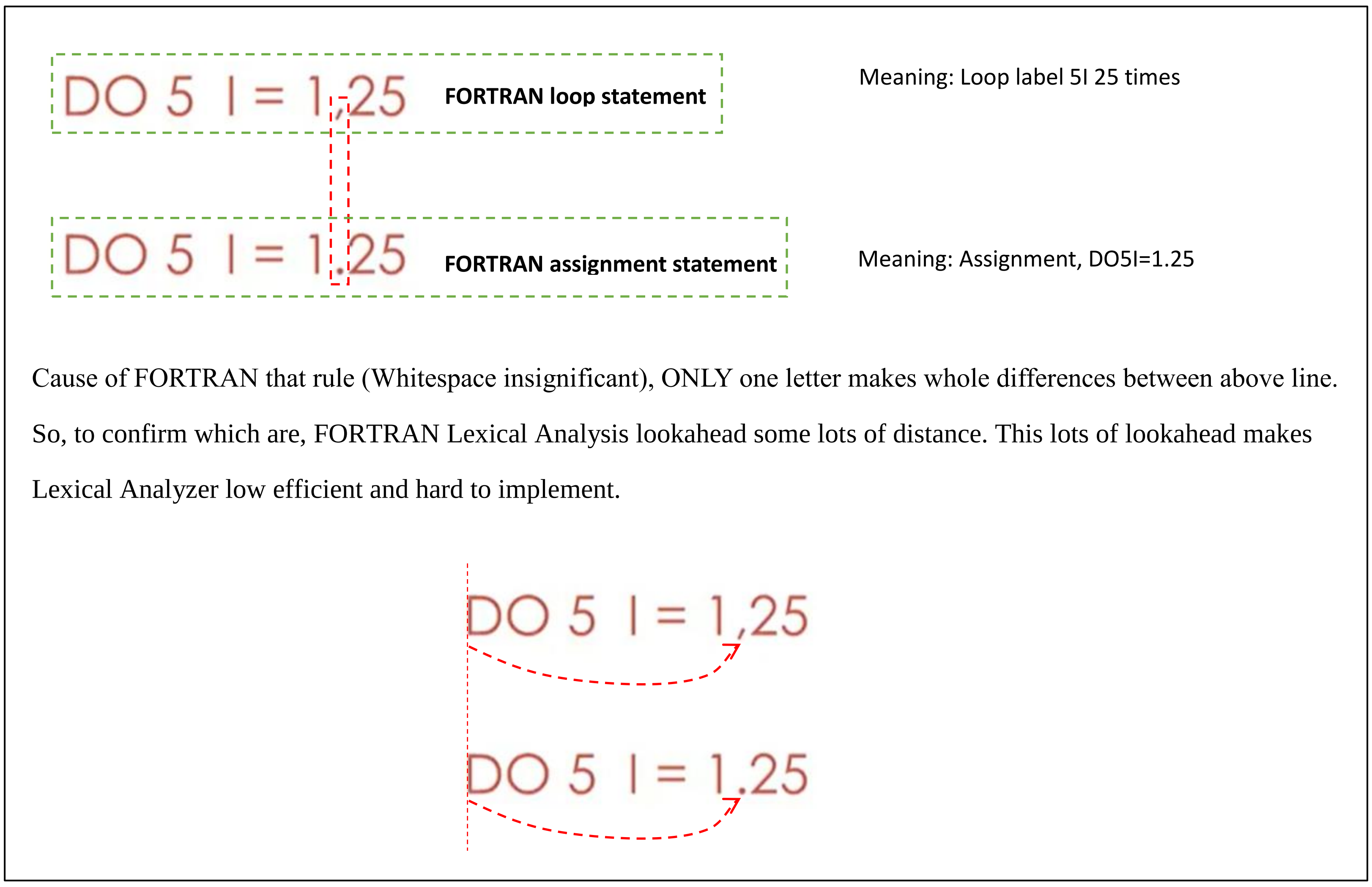


The whitespace in FORTRAN doesn't matter.

VAR1 is the same as VA R1

[Figure 2-6] FORTRAN Rule: Whitespace Legal

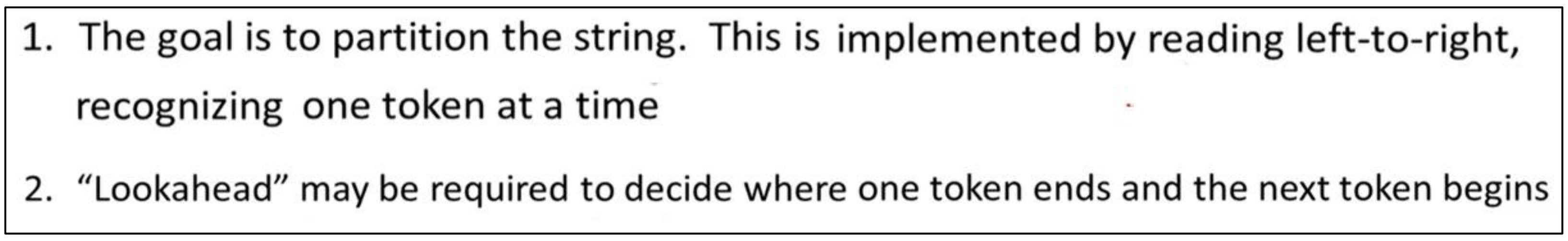
So, this feature makes Lexical Analyzer so hard to implement.


DO 5 I = 1,25
FORTRAN loop statement
Meaning: Loop label 5I 25 times
DO 5 I = 1.25
FORTRAN assignment statement
Meaning: Assignment, DO5I=1.25
DO 5 I = 1,25
DO 5 I = 1.25

Cause of FORTRAN that rule (Whitespace insignificant), ONLY one letter makes whole differences between above line. So, to confirm which are, FORTRAN Lexical Analysis lookahead some lots of distance. This lots of lookahead makes Lexical Analyzer low efficient and hard to implement.

[Figure 2-7] FORTRAN Rule: Whitespace Legal Lexical Analyzer Difficulty

So, the example helps us what would trying to do in Lexical Analyzer implementation.

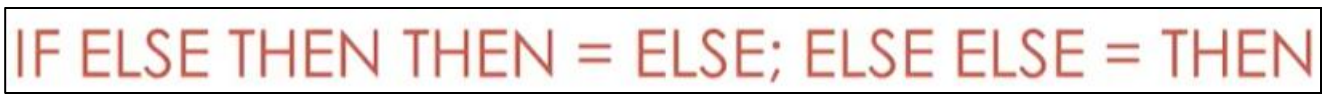


1. The goal is to partition the string. This is implemented by reading left-to-right, recognizing one token at a time
2. “Lookahead” may be required to decide where one token ends and the next token begins

[Figure 2-8] Methodology of FORTRAN Rule Whitespace Legal Implementation

2. PL/1 NOT reserved keyword

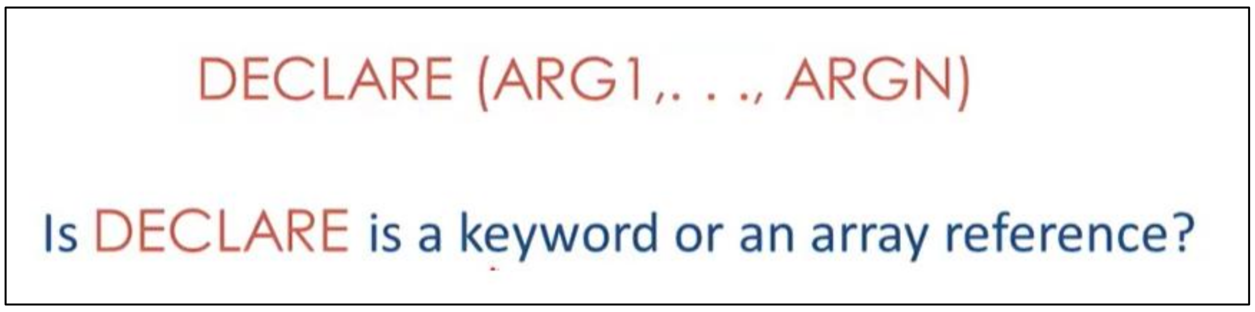
PL/1 is a programming language, which developed by IBM. And one of feature of this language is keywords NOT reserved, that means keywords can be use by user as variable names OR something else somewhere in program text/sentences. PL/1 code block as below is legal.



IF ELSE THEN THEN = ELSE; ELSE ELSE = THEN

[Figure 2-9] PL/1 Rule: No Reserved Keywords

Another example for PL/1.



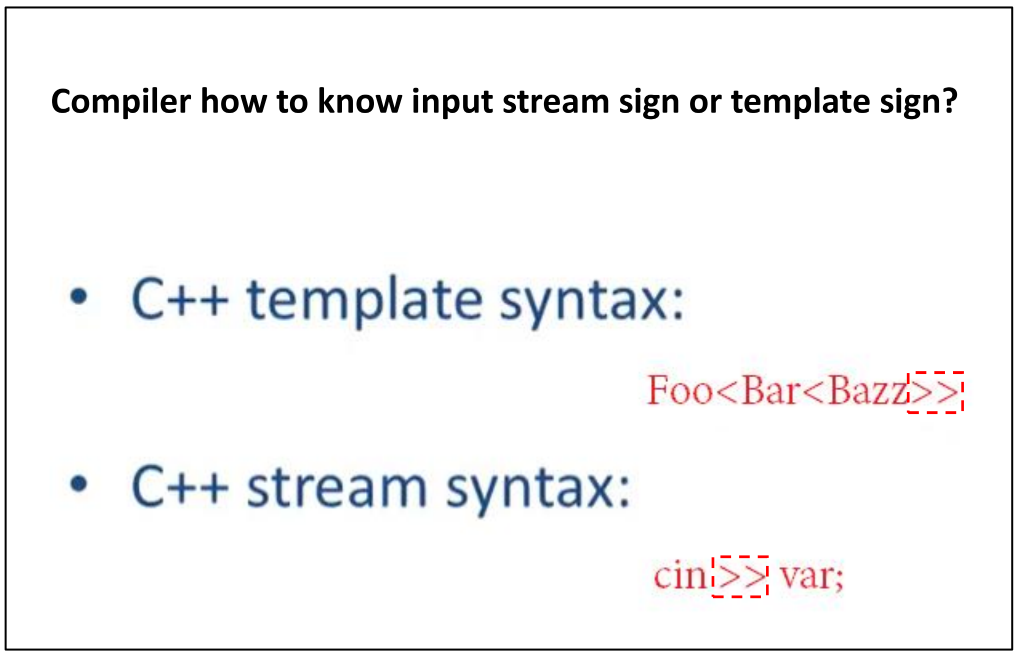
DECLARE (ARG1,. . ., ARGN)

Is DECLARE is a keyword or an array reference?

[Figure 2-10] PL/1 Rule: No Reserved Keywords Lexical Analyzer Difficulty

This is hard confirmed keyword OR an array reference, need to lookahead.

3. C++ Example



Compiler how to know input stream sign or template sign?

- C++ template syntax:

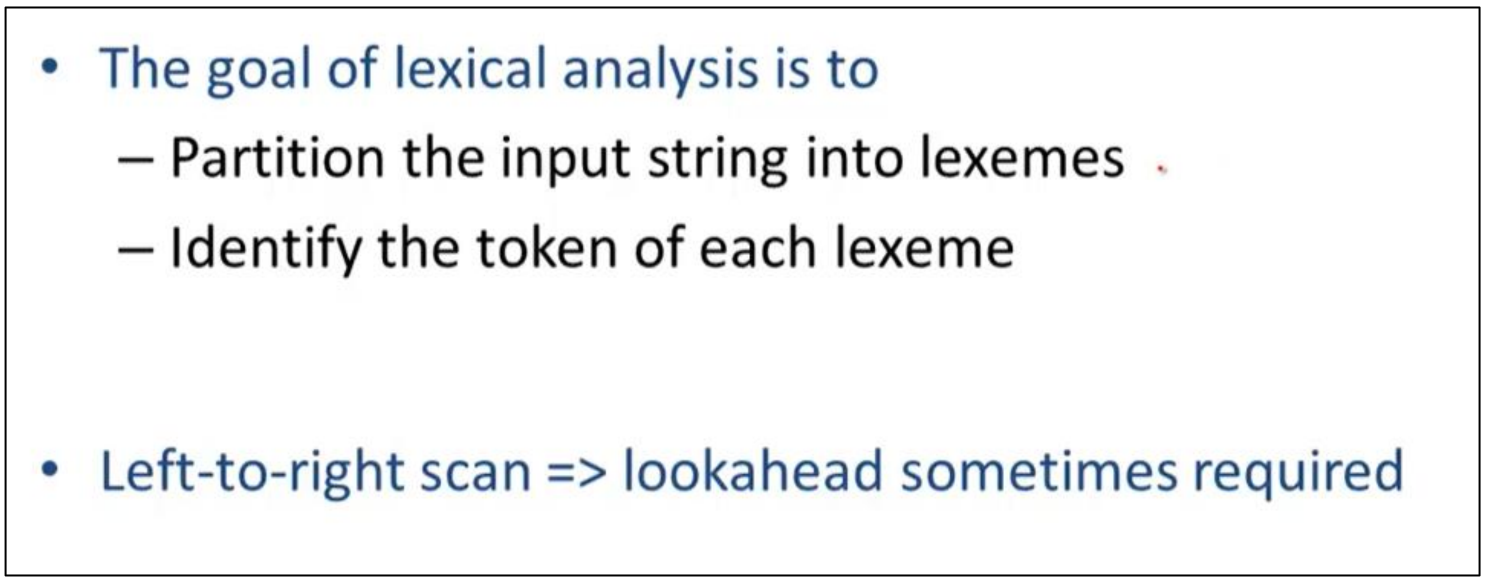
  Foo<Bar<Bazz>>

- C++ stream syntax:

  cin>> var;

[Figure 2-11] C++ Token >> Overwrite

For summarize, the Lexical Analyzer great mission is listed below.

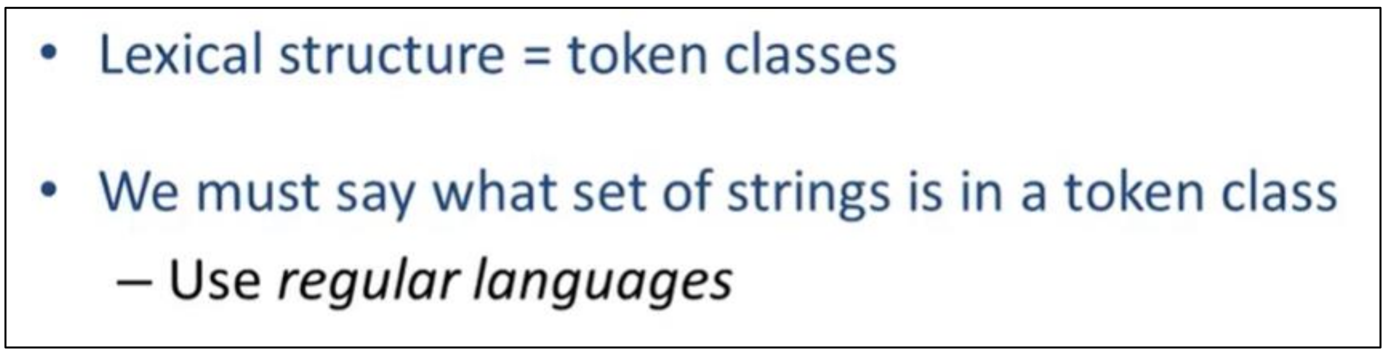


- The goal of lexical analysis is to
  - Partition the input string into lexemes
  - Identify the token of each lexeme
- Left-to-right scan => lookahead sometimes required

[Figure 2-12] Lexical Analysis Mission

2.3 Regular Language

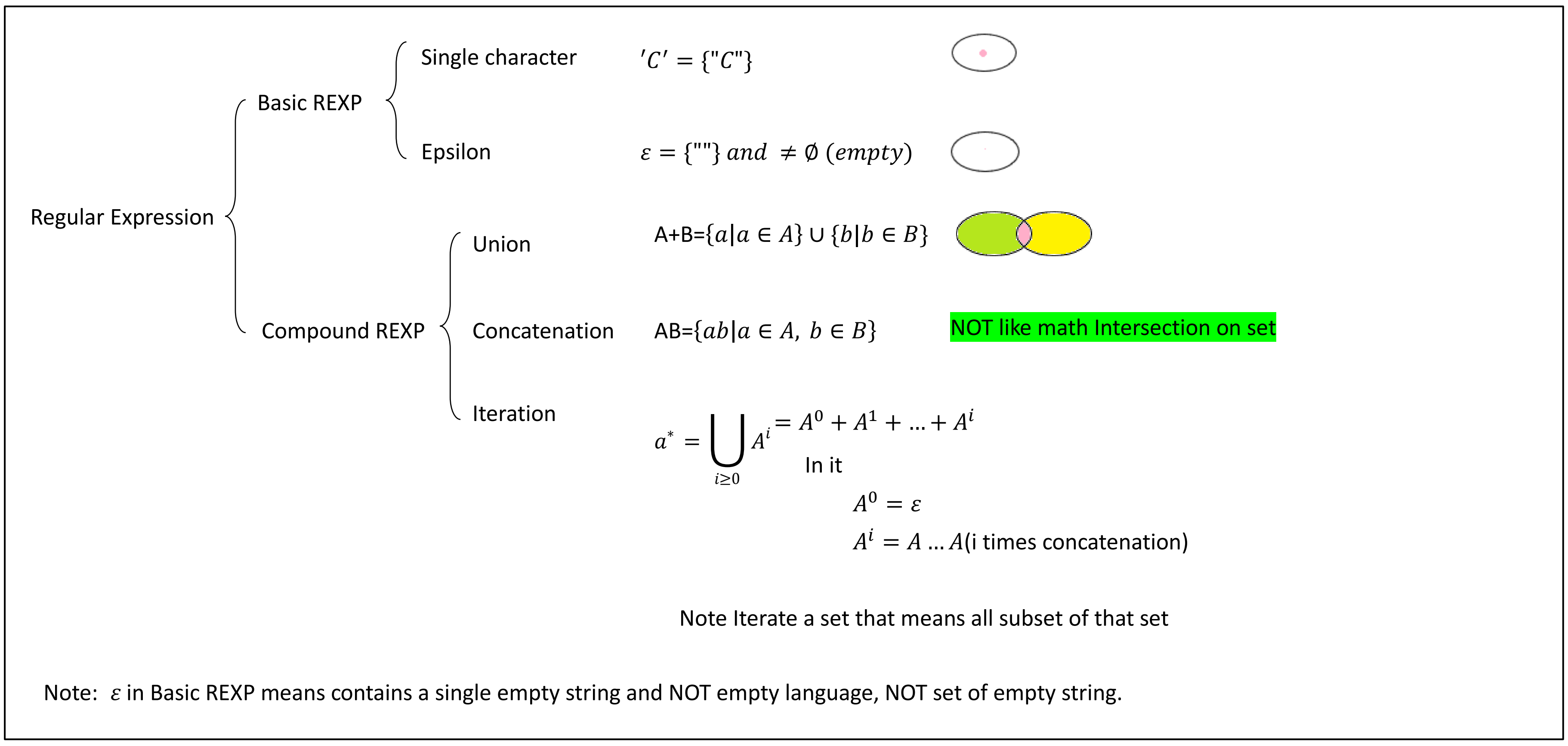
This lecture will talk about Regular Language, which are used to specify Lexical Structure of Programming Language. **Generally speaking, the Lexical Structure of a Programming Language is a set of Token Classes**.



- Lexical structure = token classes
- We must say what set of strings is in a token class
  - Use *regular languages*

[Figure 2-13] Why Regular Language?

To define Regular Language, generally use something called Regular Expression and each Regular Expression contains multiple elements, in other word a Regular Expression specify a set.



Regular Expression

Basic REXP

Single character $'C' = \{"C"\}$

Epsilon $\varepsilon = \{""\}\ and\ \neq \emptyset\ (empty)$

Compound REXP

Union A+B=$\{a|a \in A\} \cup \{b|b \in B\}$

Concatenation AB=$\{ab|a \in A,\ b \in B\}$ NOT like math Intersection on set

Iteration $a^* = \bigcup_{i\geq 0} A^i = A^0 + A^1 + \ldots + A^i$

In it

$A^0 = \varepsilon$

$A^i = A \ldots A$(i times concatenation)

Note Iterate a set that means all subset of that set

Note: ε in Basic REXP means contains a single empty string and NOT empty language, NOT set of empty string.

[Figure 2-14] Regular Expression Profiling

To summarize the Regular Expression over an alphabet Σ (prenominated sigma) are the smallest set of expressions including below.

An alphabet expressed by Regular Expression R= ε

| 'c' $c \in \Sigma$

| R + R

| RR

| R^*

Grammar

[Figure 2-15] Alphabet Expression by Regular Language (Regular Expression)

You MAYBE can't understand the logic behind the figure above, lookback it after learned Parsing. Also, below example MAY be helpful.

$\Sigma = \{0, 1\}$

This is an ε

Repeat i times/Concatenate i times

$1^* = \cup_{i\geq 0} 1^i =$ "" + 1 + 11 + 111 + 1111 + 1....1 All strings of 1's over alphabet Σ

(1+0)1 = $\{ab|a \in 1+0, b \in 1\}$ = {11, 01}

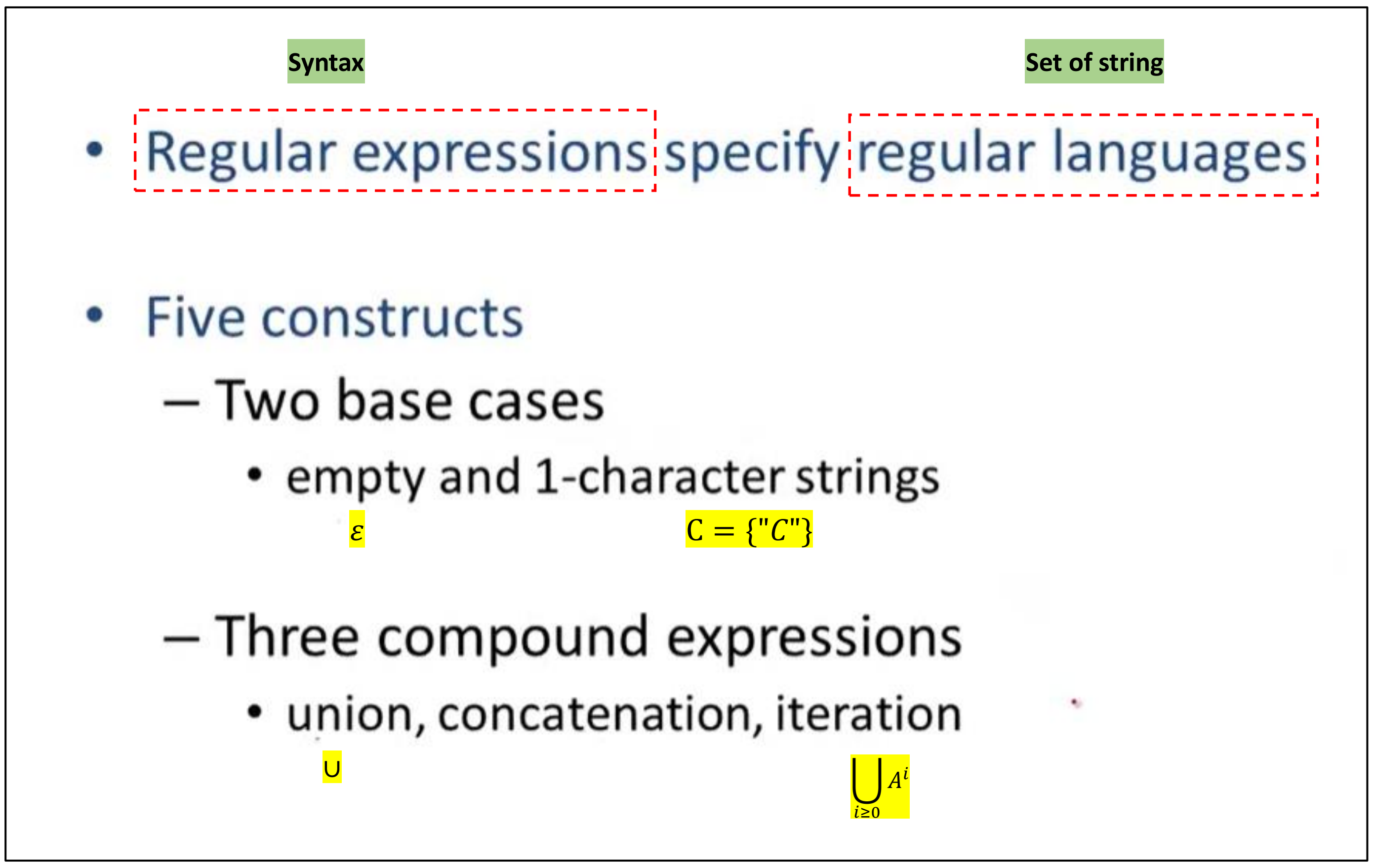
$0^* + 1^* = \{0^*|i \geq 0\} \cup \{1^*|i \geq 0\}$

Repeat i times/Concatenate i times

$(0+1)^* = \varepsilon$ + (0+1) + (0+1) (0+1) + (0+1) (0+1) … (0+1) All strings of 0's and 1's over alphabet Σ

= Σ^* (All string over an alphabet OR all string form out of an alphabet)

[Figure 2-16] Regular Language (Regular Expression) with an Alphabet



Syntax

Set of string

- Regular expressions specify regular languages
- Five constructs
  - Two base cases
    - empty and 1-character strings

ε $\quad$ $\mathrm{C} = \{"C"\}$

  - Three compound expressions
    - union, concatenation, iteration

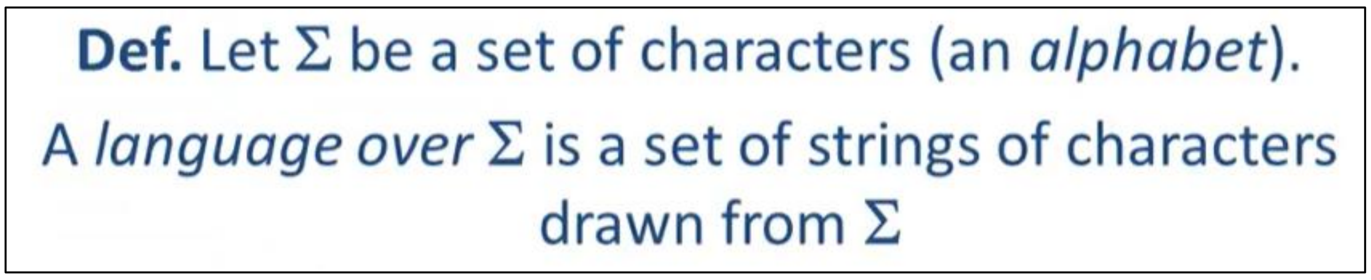
$\cup$ $\quad$ $\bigcup_{i\geq 0} A^i$

[Figure 2-17] Regular Expression Inner Structure

2.4 Formal Language

This lecture WOULD take a little digression and talk about Formal Language. Formal Language has played a big role in theoretical computer science but they're also very important in compilers. Because inside of compiler, we typically have several different formal languages that we're manipulating. Regular Expression is one example of Formal Language.

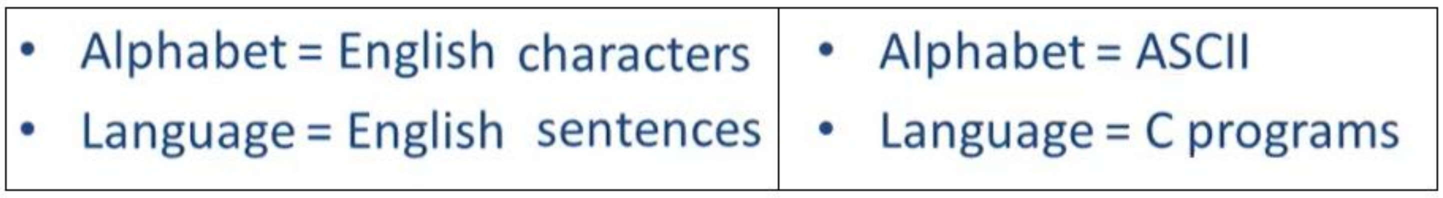
Formal Language is just any set of strings over alphabet.



Def. Let Σ be a set of characters (an *alphabet*).
A *language over* Σ is a set of strings of characters drawn from Σ

[Figure 2-18] Formal Language Definition

Examples for Formal Language

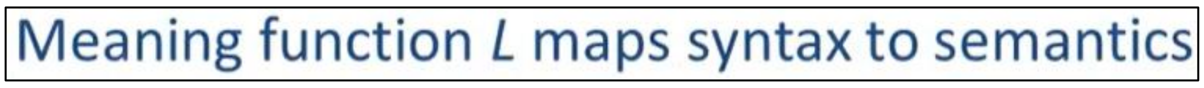


• Alphabet = English characters • Language = English sentences	• Alphabet = ASCII • Language = C programs

[Figure 2-19] Formal Language Instances

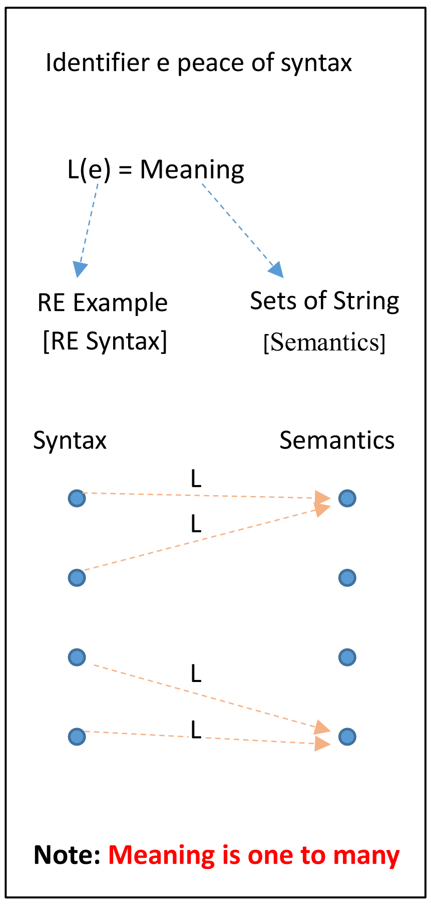
- Meaning Function

  Meaning Function is another concept of Formal Language. Meaning Function mapping the Syntax to Semantics (meaning).

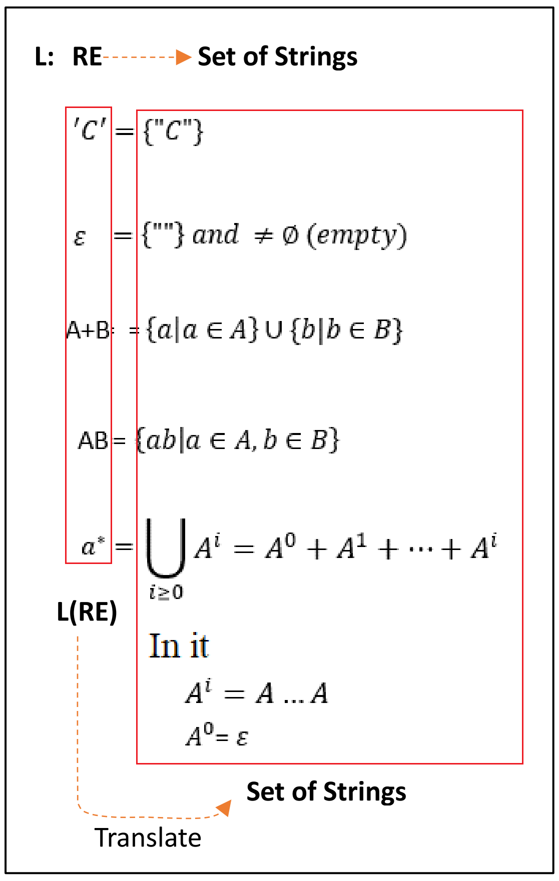


Meaning function *L* maps syntax to semantics

[Figure 2-20] Meaning Function Definition



[Figure 2-21] Syntax to Semantics



L: RE → Set of Strings

$'C' = \{"C"\}$

$\varepsilon = \{""\}\ and\ \neq \emptyset\ (empty)$

A+B $= \{a | a \in A\} \cup \{b | b \in B\}$

AB $= \{ab | a \in A, b \in B\}$

$a^* = \bigcup_{i \geq 0} A^i = A^0 + A^1 + \cdots + A^i$

L(RE)

In it

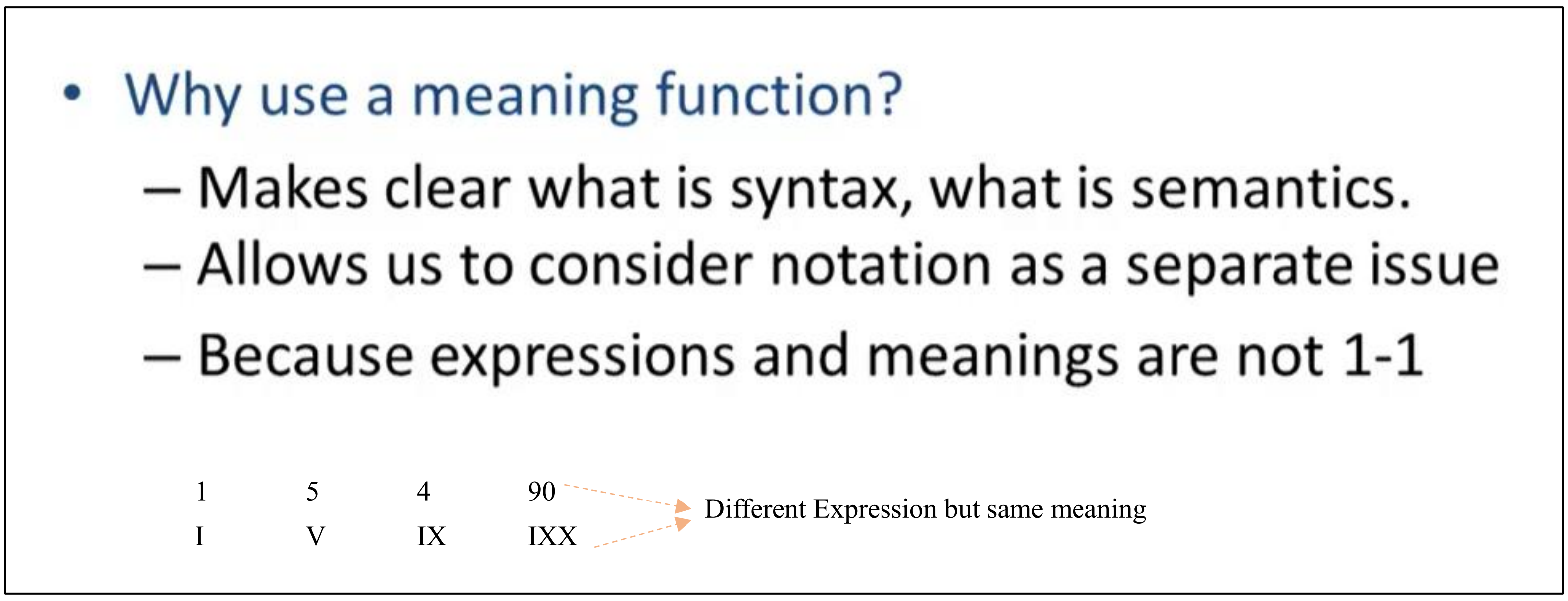
$A^i = A \ldots A$

$A^0 = \varepsilon$

Set of Strings

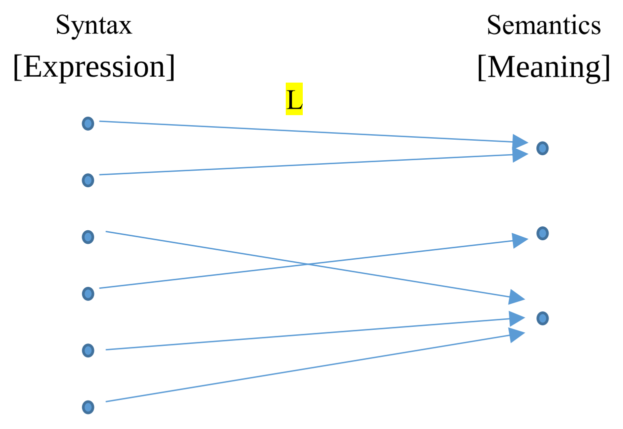
Translate

[Figure 2-22] Meaning Function Mission Example



- Why use a meaning function?
  - Makes clear what is syntax, what is semantics.
  - Allows us to consider notation as a separate issue
  - Because expressions and meanings are not 1-1

[Figure 2-23] Why Another New Concept: Meaning Function?

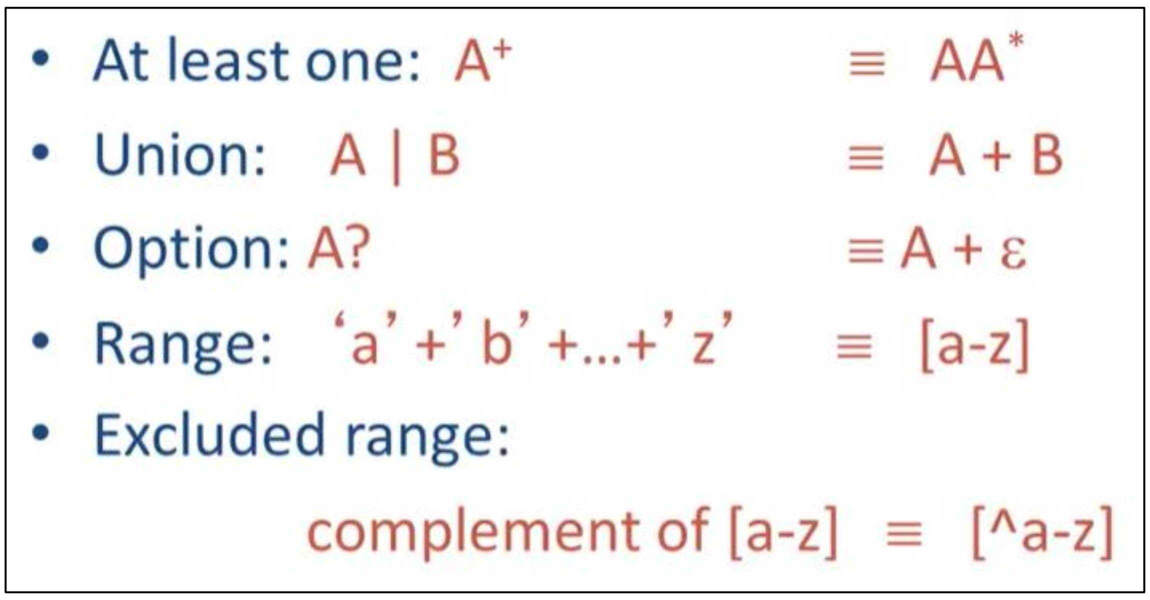


Different Syntax got the same meaning. In computing, another syntax/expression MAY easy computes for machine, this situation can replace the expression make it easy to compute. **THIS IS BASIC THOUGHT FOR OPTIMIZAITON PHASE IN COMPILER.**

[Figure 2-24] Syntax to Semantics Many to One

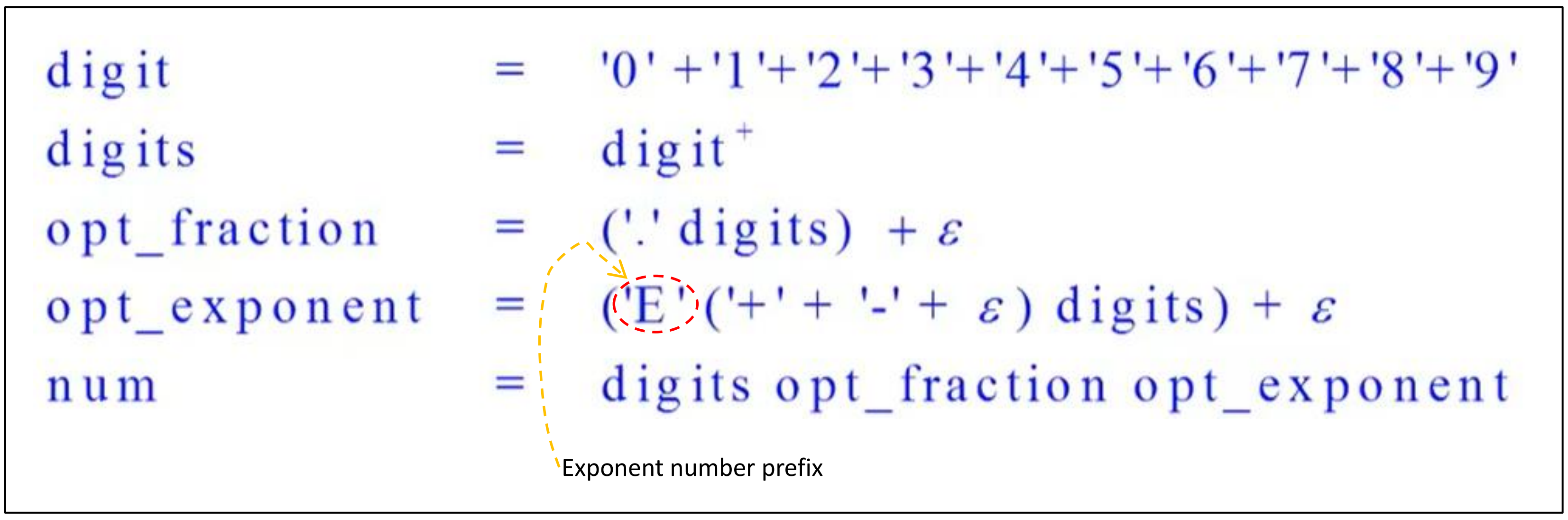
2.5 Lexical Specification

This section would learn How Regular Expression construct a full Lexical Specification on programming language. The primitive REXP not strong enough, so nowadays made some extensions on REXP.



- At least one: A^+ $\equiv$ AA^*
- Union: A | B $\equiv$ A + B
- Option: A? $\equiv$ A + ε
- Range: 'a' +' b' +...+' z' $\equiv$ [a-z]
- Excluded range:
  complement of [a-z] $\equiv$ [^a-z]

[Figure 2-25] Review: The Programming Language Specification with Regular Expression



digit = '0' + '1' + '2' + '3' + '4' + '5' + '6' + '7' + '8' + '9'

digits = digit$^+$

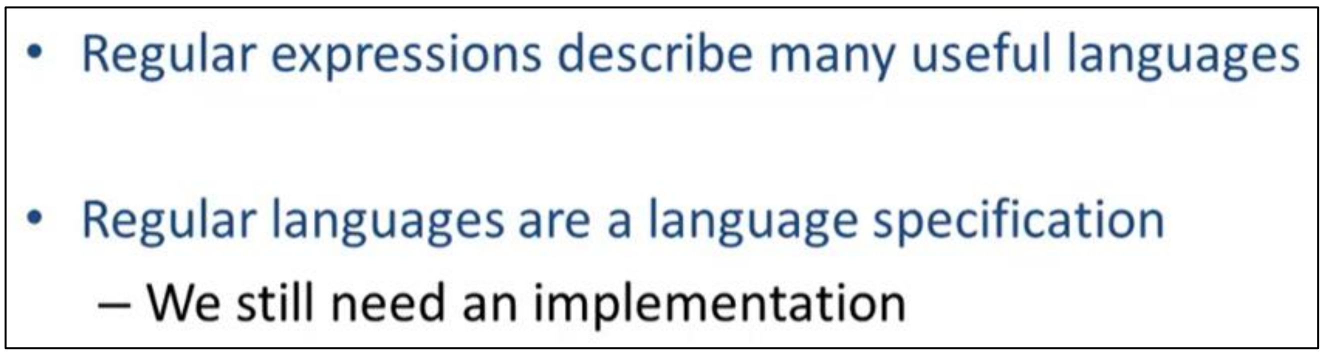
opt_fraction = ('.' digits) + ε

opt_exponent = ('E' ('+' + '-' + ε) digits) + ε

num = digits opt_fraction opt_exponent

Exponent number prefix

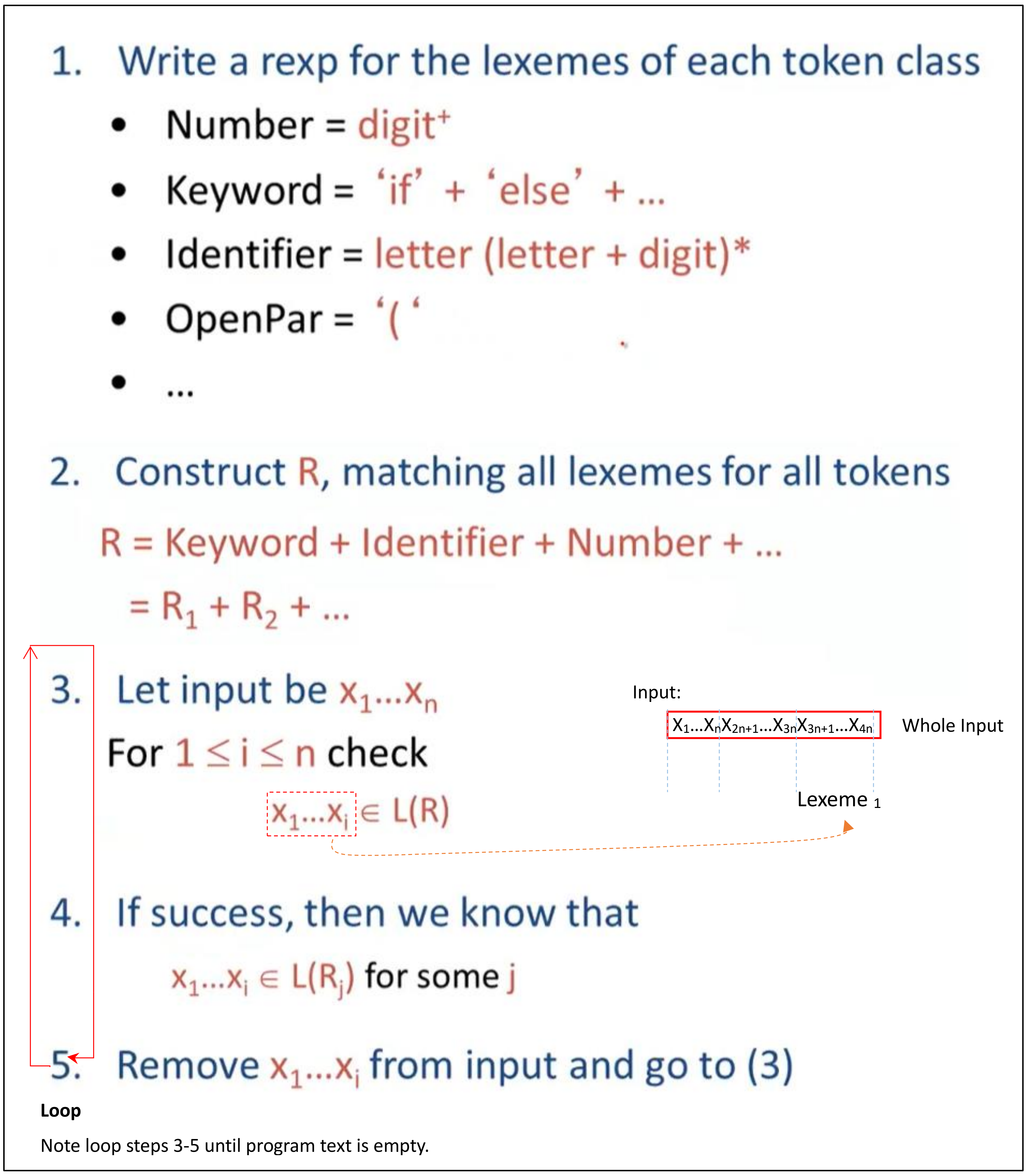
[Figure 2-26] The Lexical Specification of a Language



- Regular expressions describe many useful languages
- Regular languages are a language specification
  – We still need an implementation

[Figure 2-27] Regular Expression Summarize

Given a string s and a REXP R, is $s \in L(R)$? How to decide it? How to handle the string s? Below is the analogy of it.



1. Write a rexp for the lexemes of each token class
   - Number = digit$^+$
   - Keyword = 'if' + 'else' + ...
   - Identifier = letter (letter + digit)*
   - OpenPar = '('
   - ...
2. Construct R, matching all lexemes for all tokens

   R = Keyword + Identifier + Number + ...

   = $R_1 + R_2 + \ldots$
3. Let input be $x_1 \ldots x_n$

   For $1 \leq i \leq n$ check

   $x_1 \ldots x_i \in L(R)$
4. If success, then we know that

   $x_1 \ldots x_i \in L(R_j)$ for some j
5. Remove $x_1 \ldots x_i$ from input and go to (3)

Note loop steps 3-5 until program text is empty.

[Figure 2-28] Regular Language Implementation Logic

MAYBE we got three question after definition of Lexical Specification.

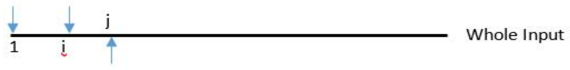
• How much input is used?

In the program text, if occurs some strings are more like more than one token, how to match it?

Construct R, matching all lexemes for all tokens

R = Keyword + Identifier + Number + …

= $R_1 + R_2$ + …



$X_1 \ldots X_i \in L(R)$

$i \neq j$

$X_1 \ldots X_j \in L(R)$

Example, if occur string “=” and next letter is also “=”, that means this token is equal mark “=” OR logical mark “==”?

If occur this, generally, choose Max Match.

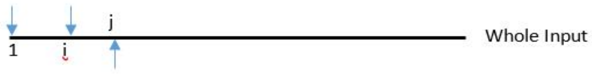
• Which token is used?

In some programming language, keyword and identifier can be the same names. HOW to confirm that token whether keywords OR identifier?

Construct R, matching all lexemes for all tokens

R = Keyword + Identifier + Number + …

= $R_1 + R_2$ + …



$X_1 \ldots X_i \in L(R_1)$

$i \neq j$

$X_1 \ldots X_j \in L(R_2)$

keywords =” if” +” else” +…

identifiers = letter (letter + digits)*

if lexeme $\in L(Keywords)$

if lexeme $\in L(Identifiers)$

If occur this, how to do?

Example:

"If" $\in L(Keywords)$

"If" $\in L(Identifiers)$

Generally, we use Priority Order.

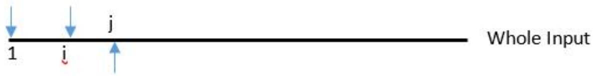
• What if no rule matches?

If no matches, what WOULD do?

Construct R, matching all lexemes for all tokens

R = Keyword + Identifier + Number + …

= $R_1 + R_2$ + …

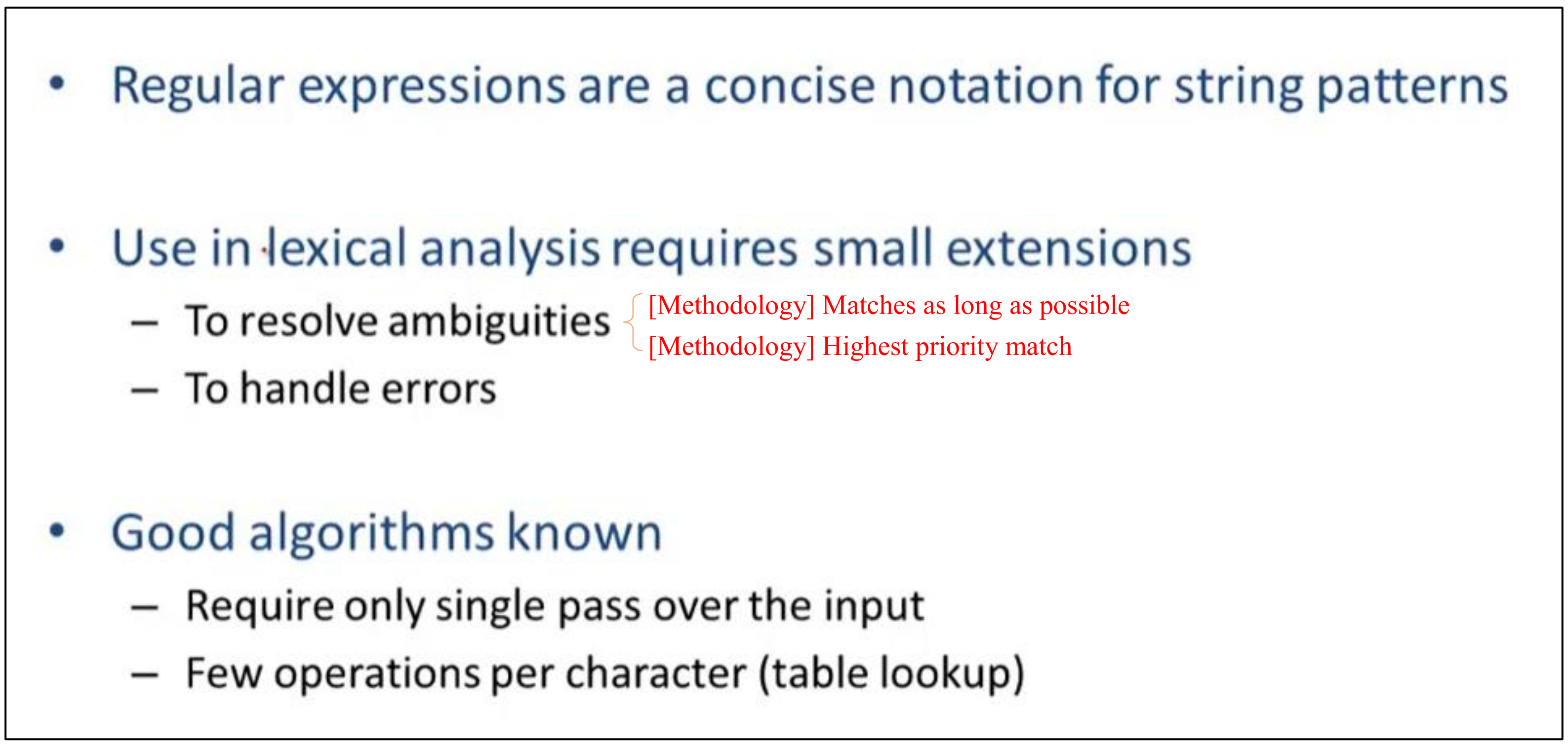


$X_1 \ldots X_i \notin L(R)$

If no matches, don’t crash the Compiler give error tips. Like do some error message and put it the last on Priority Order.

If really happens, give some error tips instead Compiler crashes.

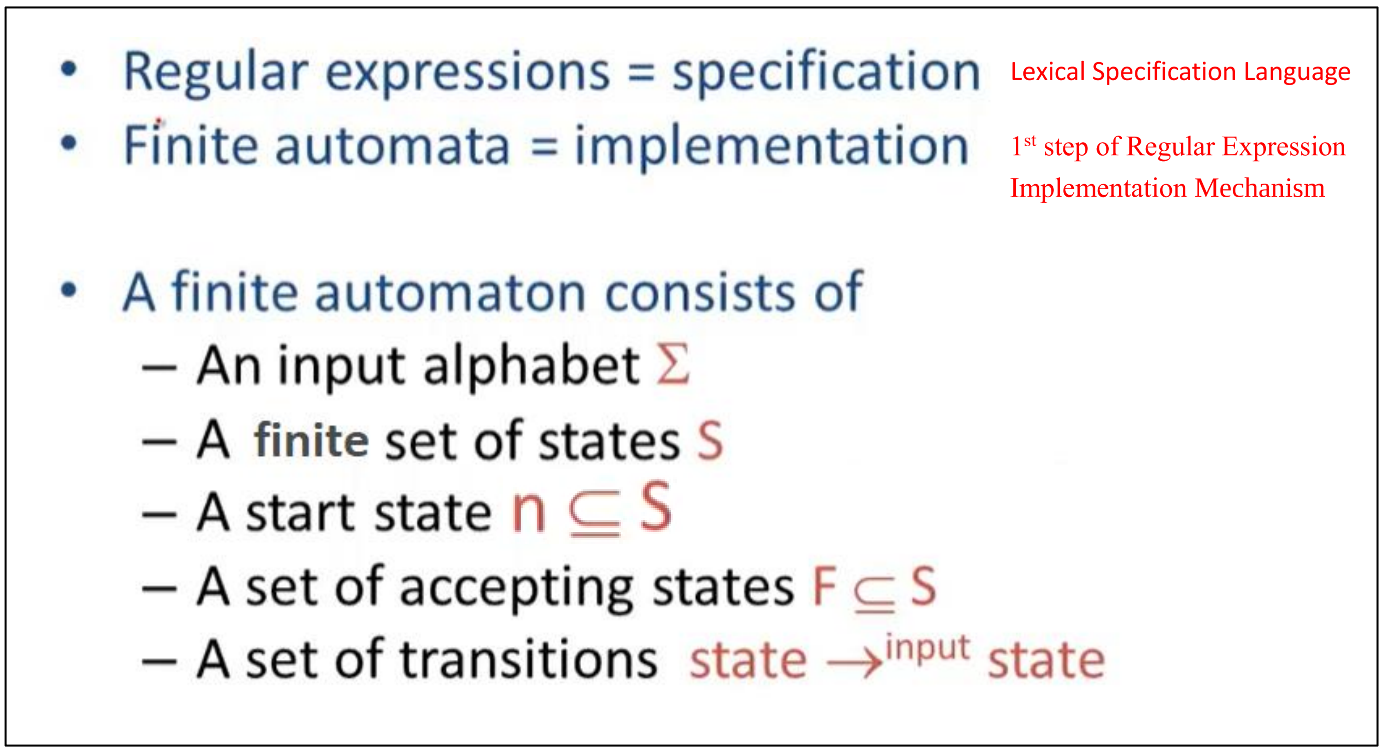
[Figure 2-29] Typical Question of Lexical Specification



- Regular expressions are a concise notation for string patterns
- Use in lexical analysis requires small extensions
  - To resolve ambiguities
    - [Methodology] Matches as long as possible
    - [Methodology] Highest priority match
  - To handle errors
- Good algorithms known
  - Require only single pass over the input
  - Few operations per character (table lookup)

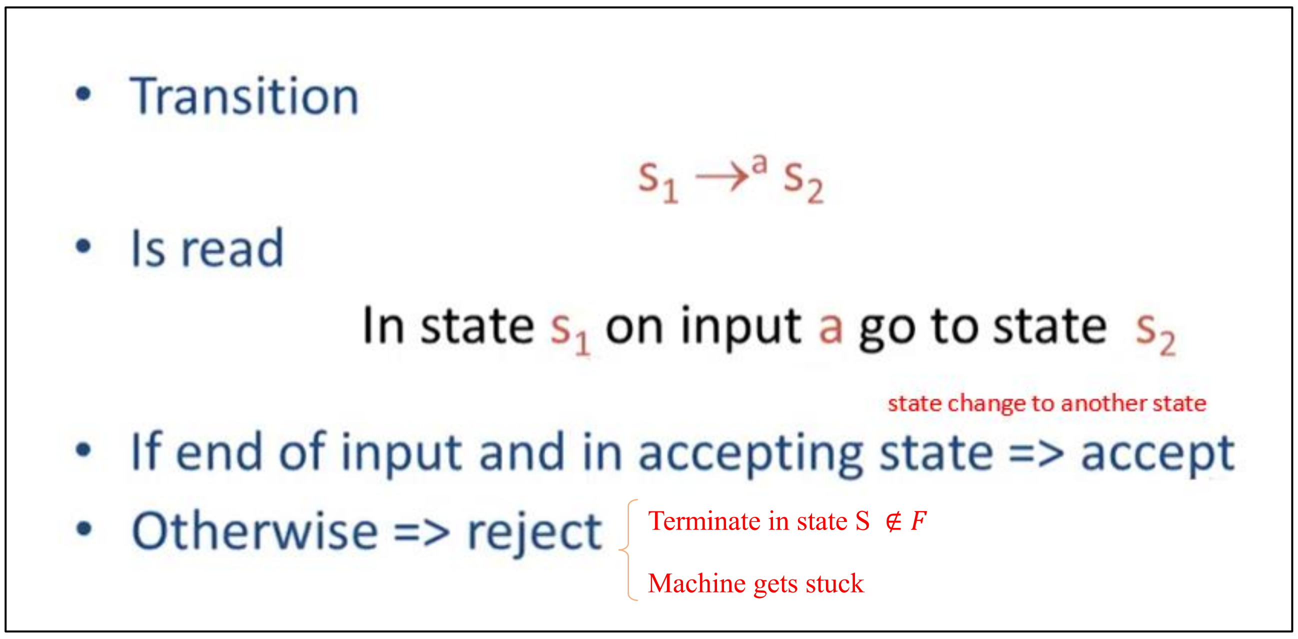
[Figure 2-30] Mixed Summary

2.6 Finite Automata



- Regular expressions = specification Lexical Specification Language
- Finite automata = implementation 1st step of Regular Expression Implementation Mechanism
- A finite automaton consists of
  - An input alphabet Σ
  - A finite set of states S
  - A start state $n \subseteq S$
  - A set of accepting states $F \subseteq S$
  - A set of transitions state $\rightarrow^{input}$ state

[Figure 2-31] Finite Automata Introduce



- Transition

$$s_1 \rightarrow^{a} s_2$$

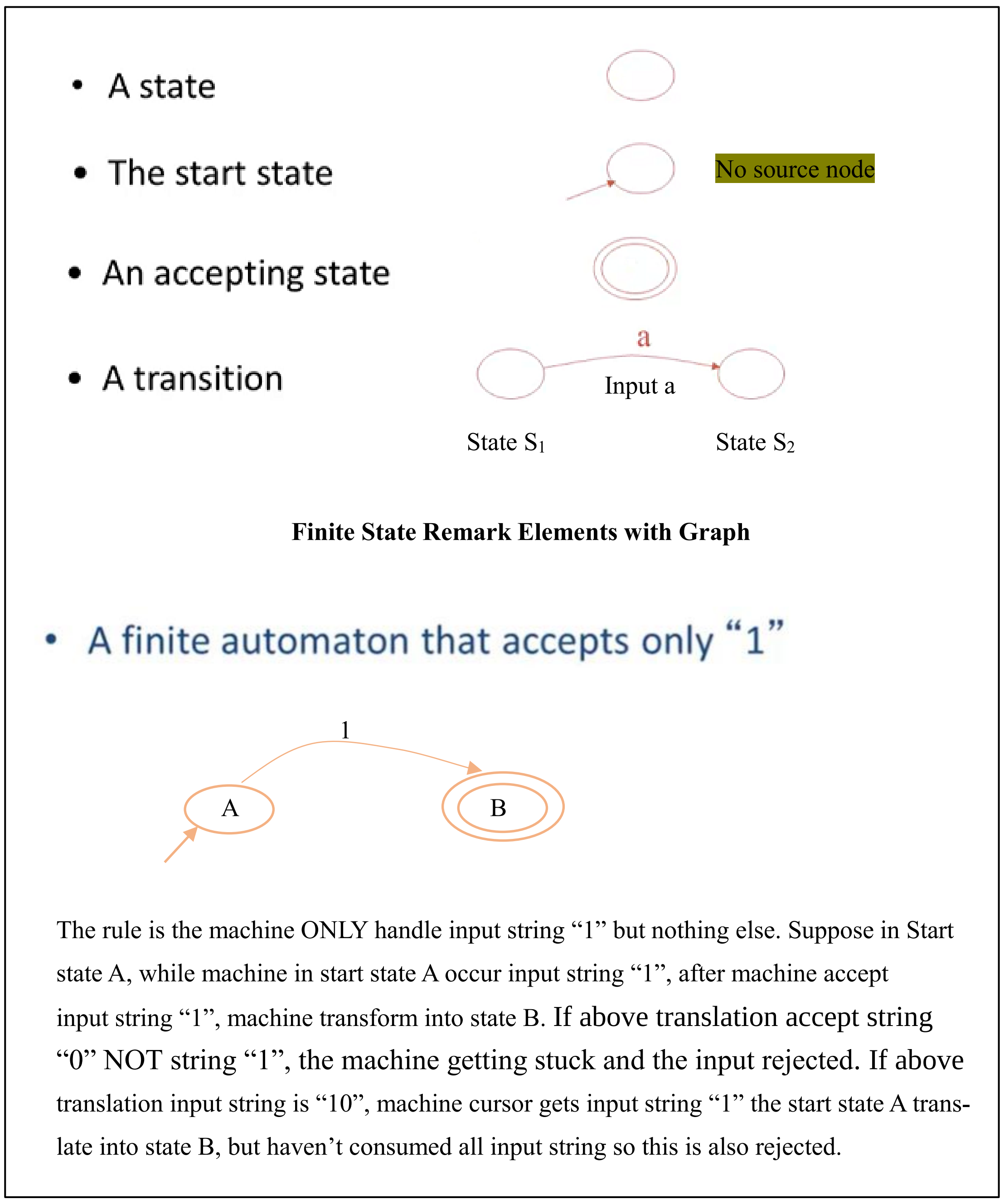
- Is read

In state s_1 on input a go to state s_2

state change to another state

- If end of input and in accepting state => accept
- Otherwise => reject
  - Terminate in state $S \notin F$
  - Machine gets stuck

[Figure 2-32] Demo: Finite Automata Translation



- A state
- The start state
- An accepting state
- A transition

No source node

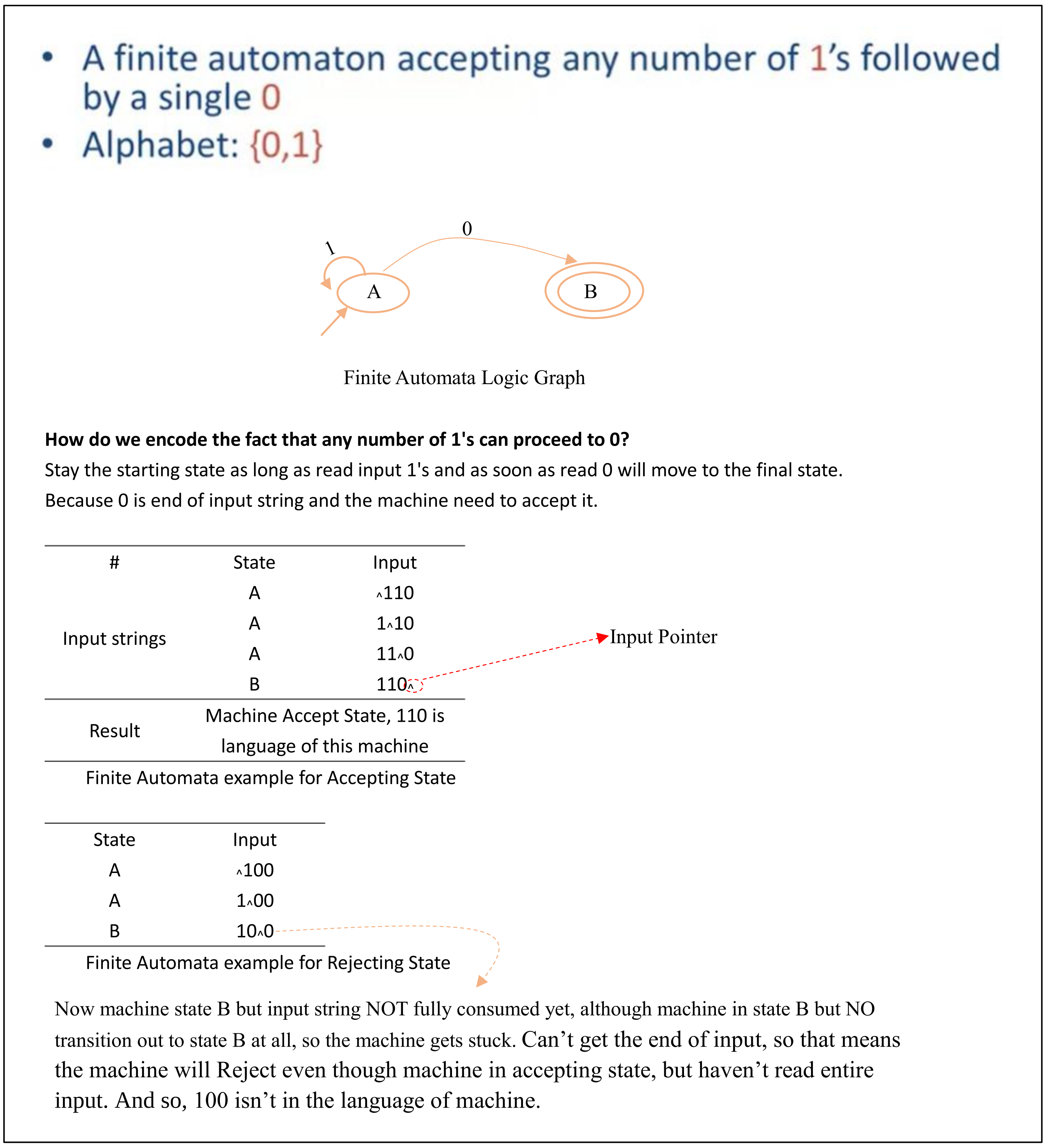
Finite State Remark Elements with Graph

- A finite automaton that accepts only “1”

The rule is the machine ONLY handle input string “1” but nothing else. Suppose in Start state A, while machine in start state A occur input string “1”, after machine accept input string “1”, machine transform into state B. If above translation accept string “0” NOT string “1”, the machine getting stuck and the input rejected. If above translation input string is “10”, machine cursor gets input string “1” the start state A translate into state B, but haven’t consumed all input string so this is also rejected.

[Figure 2-33] Finite Automata Notation with Graph and Example

Note that Language of a Finite Automaton is set of accepting strings.



- A finite automaton accepting any number of 1’s followed by a single 0
- Alphabet: {0,1}

Finite Automata Logic Graph

How do we encode the fact that any number of 1's can proceed to 0?

Stay the starting state as long as read input 1's and as soon as read 0 will move to the final state. Because 0 is end of input string and the machine need to accept it.

#	State	Input
Input strings	A	‸110
	A	1‸10
	A	11‸0
	B	110‸
Result	Machine Accept State, 110 is language of this machine	

Input Pointer

Finite Automata example for Accepting State

State	Input
A	‸100
A	1‸00
B	10‸0

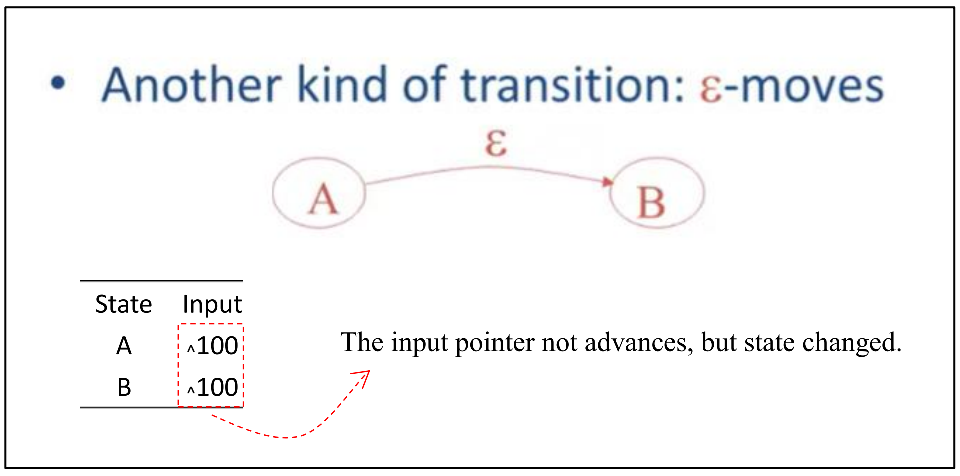
Finite Automata example for Rejecting State

Now machine state B but input string NOT fully consumed yet, although machine in state B but NO transition out to state B at all, so the machine gets stuck. Can’t get the end of input, so that means the machine will Reject even though machine in accepting state, but haven’t read entire input. And so, 100 isn’t in the language of machine.

[Figure 2-34] A Simple Example: Finite Automata

Up to this point Finite Automata consumes a character of input every time it makes a move. So, the move happens only if there is inputs, the input pointer advances and character accept. Is there any move (transition) without no inputs and no consuming inputs to make a move?

The answer is it’s existed, and called **Epsilon** move. The behind of the Epsilon move is that machine can make a state transition without consuming input.

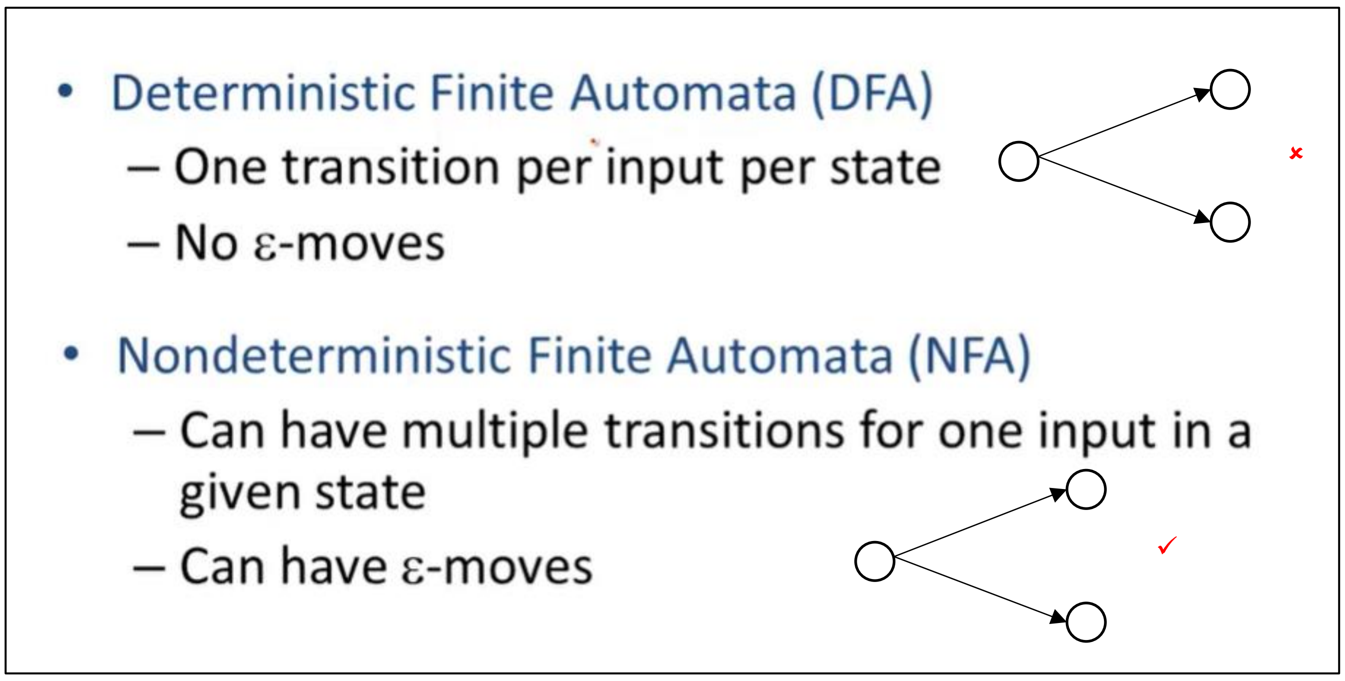


- Another kind of transition: ε-moves

State	Input
A	‸100
B	‸100

The input pointer not advances, but state changed.

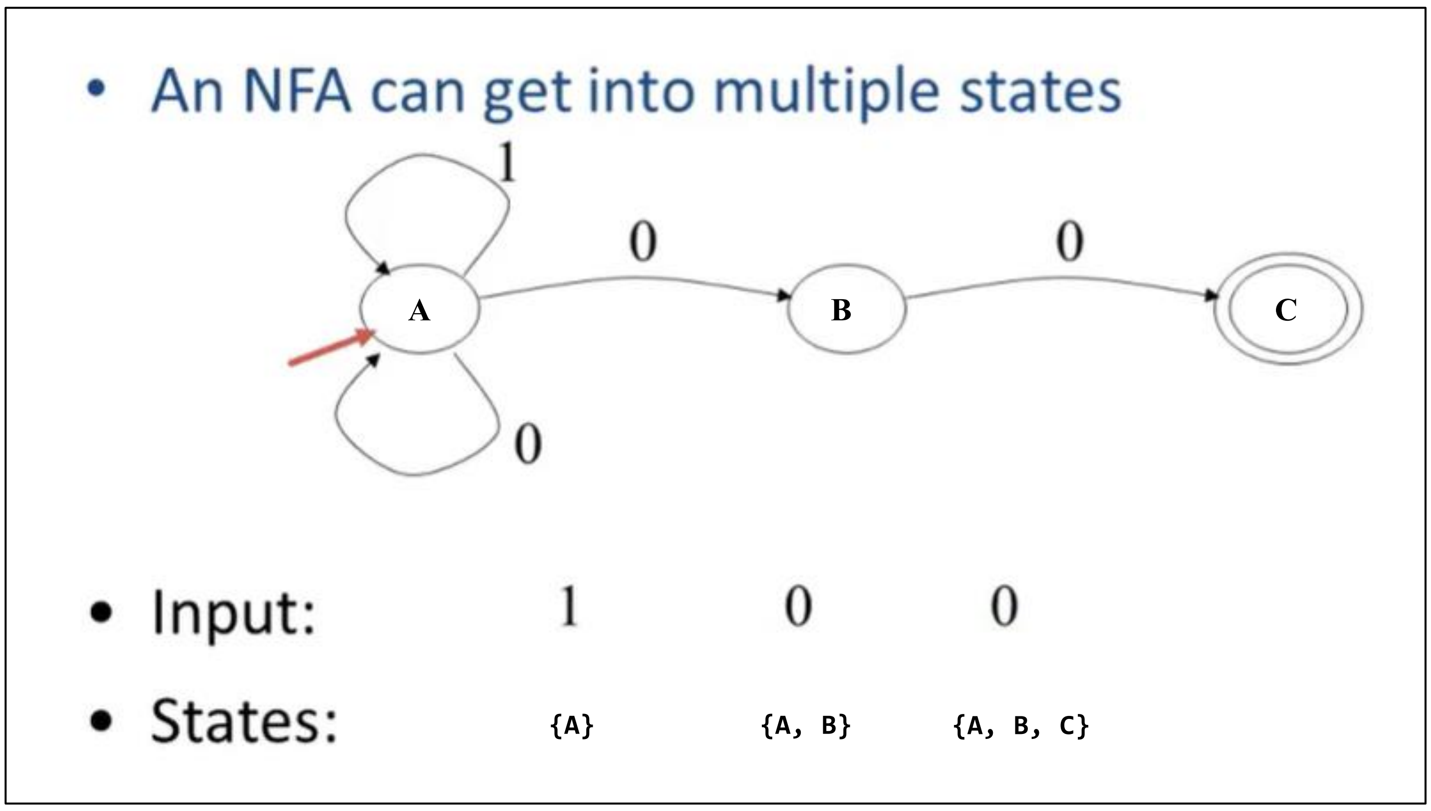
[Figure 2-35] Epsilon move



- Deterministic Finite Automata (DFA)
  - One transition per input per state
  - No ε-moves
- Nondeterministic Finite Automata (NFA)
  - Can have multiple transitions for one input in a given state
  - Can have ε-moves

[Figure 2-36] DFA VS. NFA

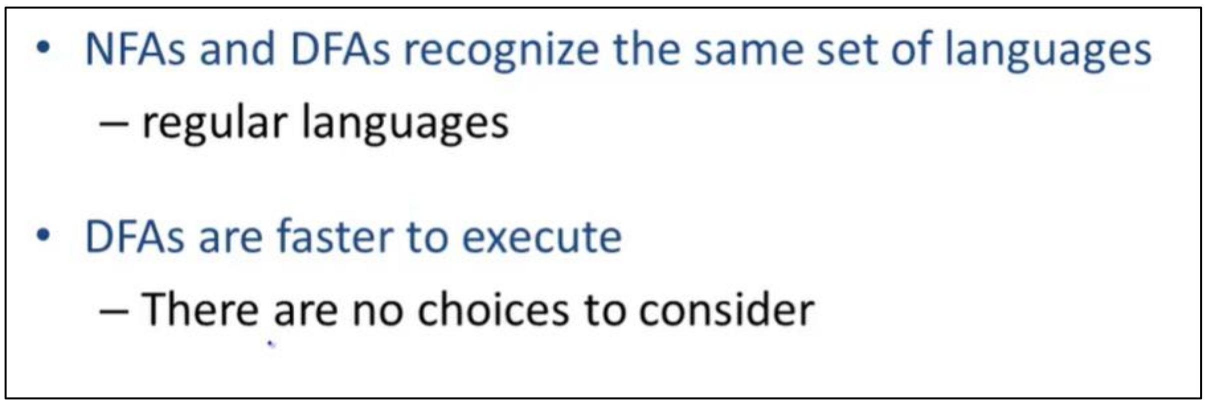
Note that the fundamental difference between Deterministic Finite Automata and Nondeterministic Finite Automata is whether have Epsilon Automata or Not.



- An NFA can get into multiple states

- Input: 1 0 0
- States: {A} {A, B} {A, B, C}

[Figure 2-37] NFA Example

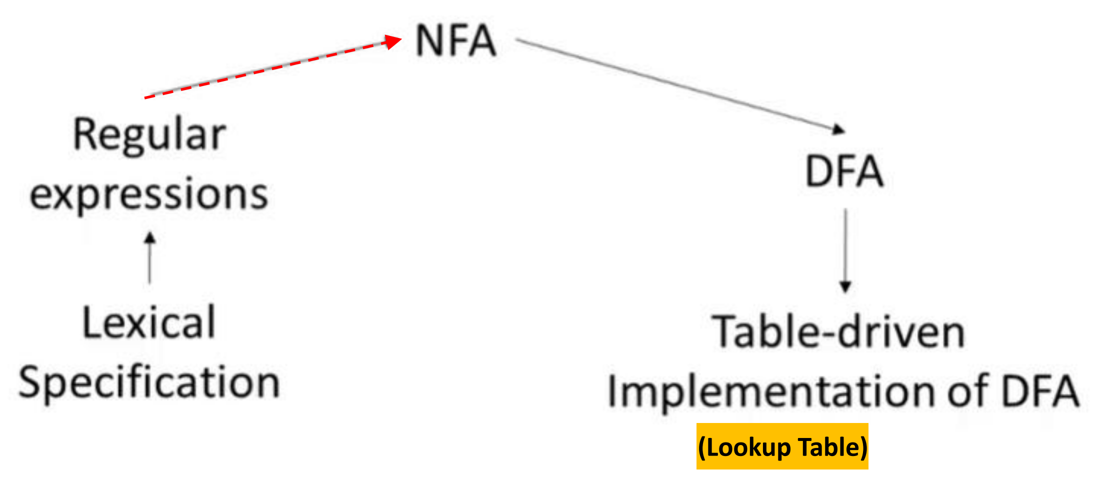


- NFAs and DFAs recognize the same set of languages
  - regular languages
- DFAs are faster to execute
  - There are no choices to consider

[Figure 2-38] DFA, NFA summary

2.7 RE to NFAs

This section would cover about Regular Expression (RE) converting into Non-deterministic Finite Automata (NFA). Before get started give an overview of next sections as below.



[Target]

Lexical Specification Implementation

[Solution]

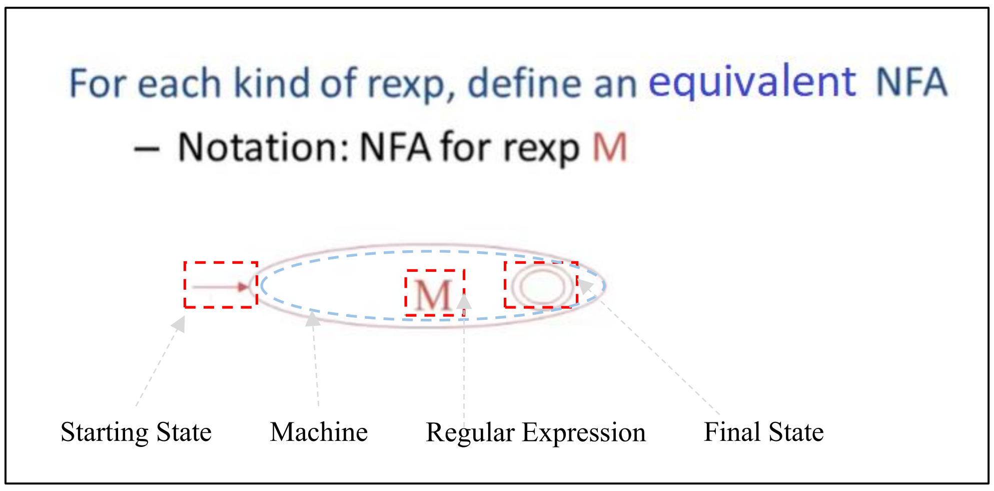
We have Lexical Specification to implement. So, the first step is to write down as a set of Regular Expressions. The second step, the Regular Expressions convert into Non-deterministic Finite Automata (NFA). The third step, NFA converts into Deterministic Finite Automata (DFA). The forth step, DFA converts into Lookup Table.

[Methodology]

Lexical Specification writes down equivalent Regular Expressions. The Regular Expression translates into NFA. Next step, NFA translates into DFA. The last DFA implemented as Lookup Table.

This section concentrates on the REXP translates into NFA red arrow shown above processes.

[Figure 2-39] Course Overview

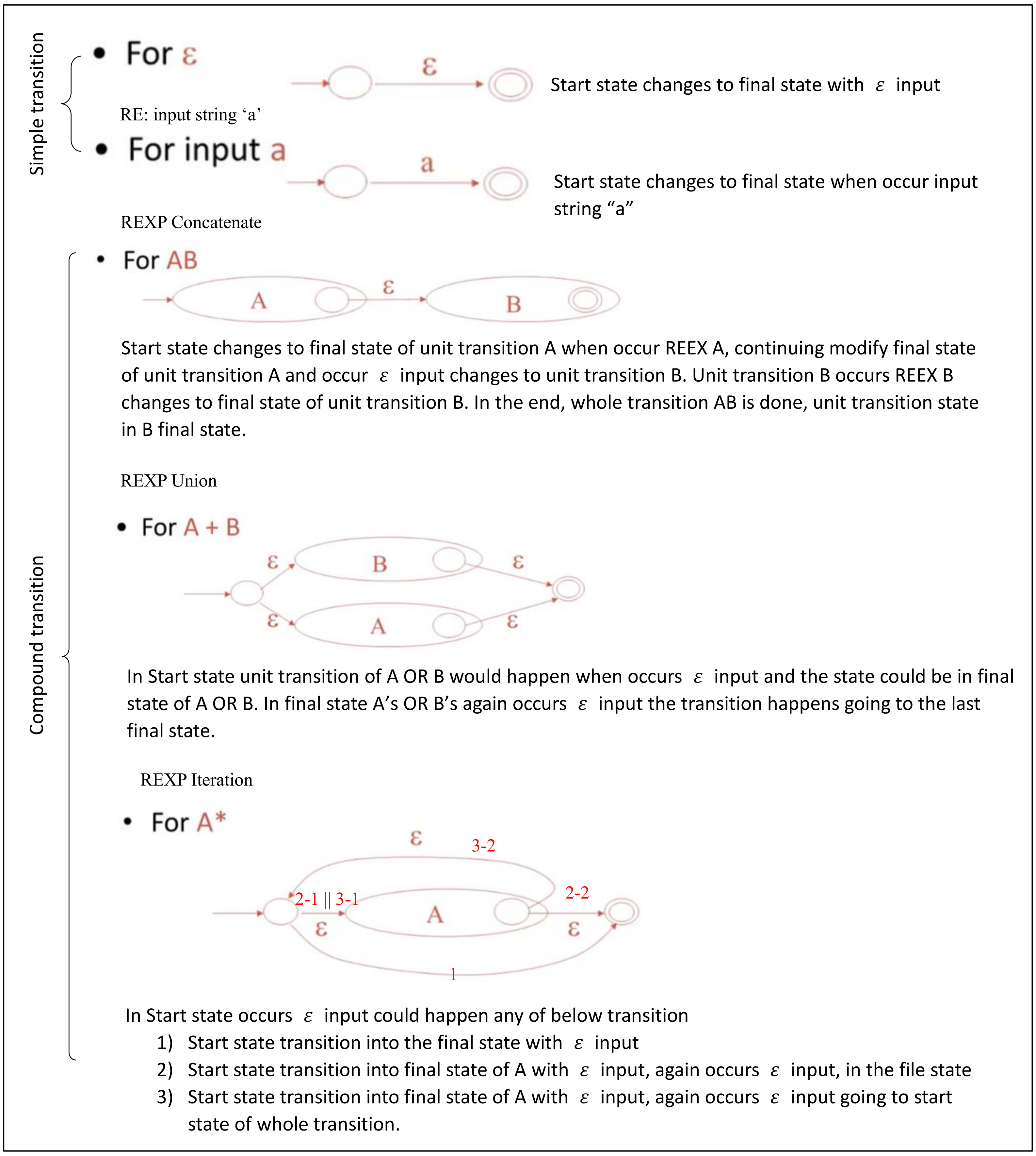


For each kind of rexp, define an equivalent NFA

– Notation: NFA for rexp M

[Figure 2-40] Notation of RE to NFA Transmission

Note that in the RE to NFA transmission notation, the machine detail or structure is unnecessary.



- For ε

Start state changes to final state with ε input

RE: input string ‘a’

- For input a

Start state changes to final state when occur input string “a”

REXP Concatenate

- For AB

Start state changes to final state of unit transition A when occur REEX A, continuing modify final state of unit transition A and occur ε input changes to unit transition B. Unit transition B occurs REEX B changes to final state of unit transition B. In the end, whole transition AB is done, unit transition state in B final state.

REXP Union

- For A + B

In Start state unit transition of A OR B would happen when occurs ε input and the state could be in final state of A OR B. In final state A’s OR B’s again occurs ε input the transition happens going to the last final state.

REXP Iteration

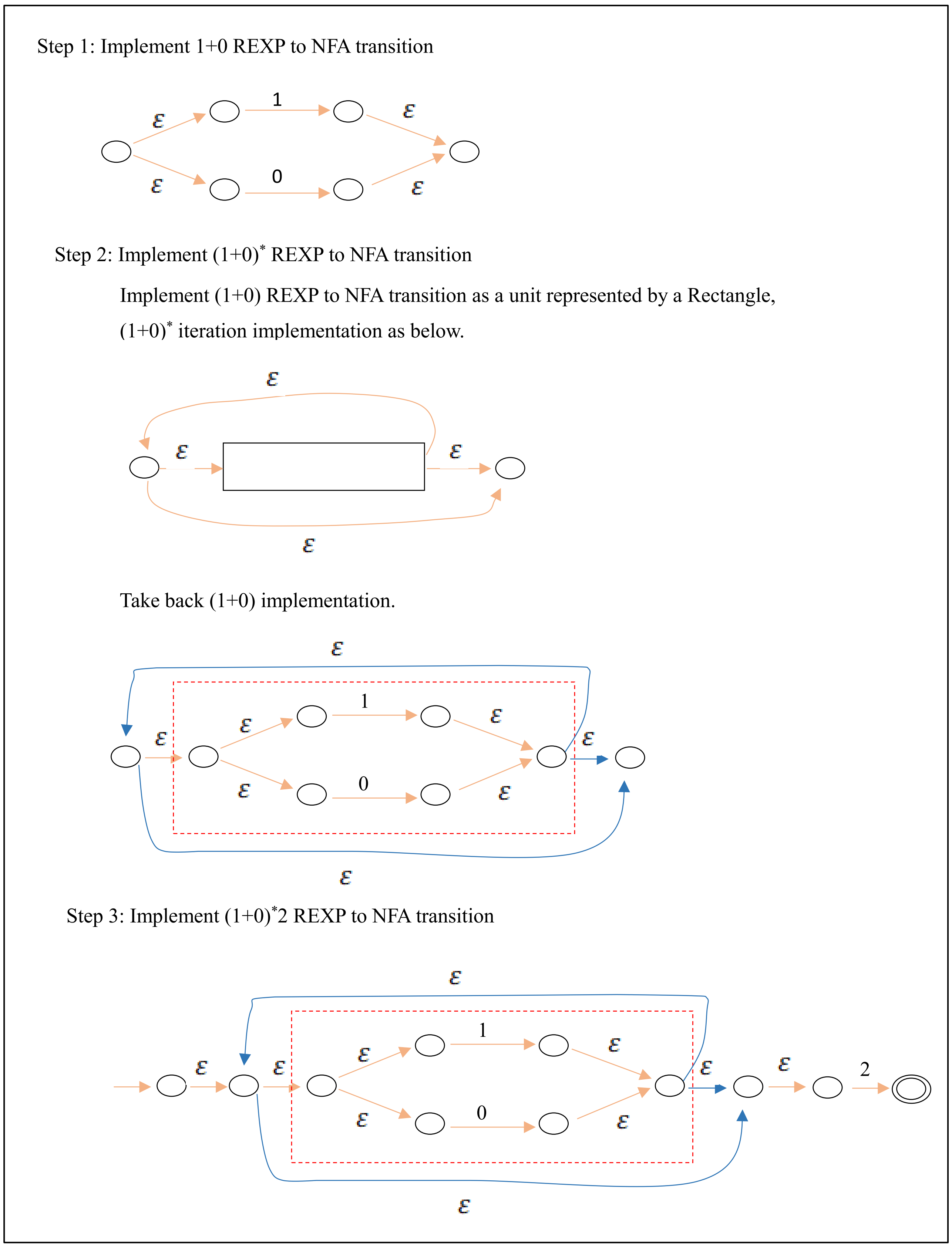
- For A*

In Start state occurs ε input could happen any of below transition

1) Start state transition into the final state with ε input
2) Start state transition into final state of A with ε input, again occurs ε input, in the file state
3) Start state transition into final state of A with ε input, again occurs ε input going to start state of whole transition.

[Figure 2-41] Notation of RE to NFA Transmission

There is a demonstration of REXP to NFA transmission, HOW to implement NFA transition with REXP string $(1+0)^{*}2$?


Step 1: Implement 1+0 REXP to NFA transition

Step 2: Implement $(1+0)^*$ REXP to NFA transition

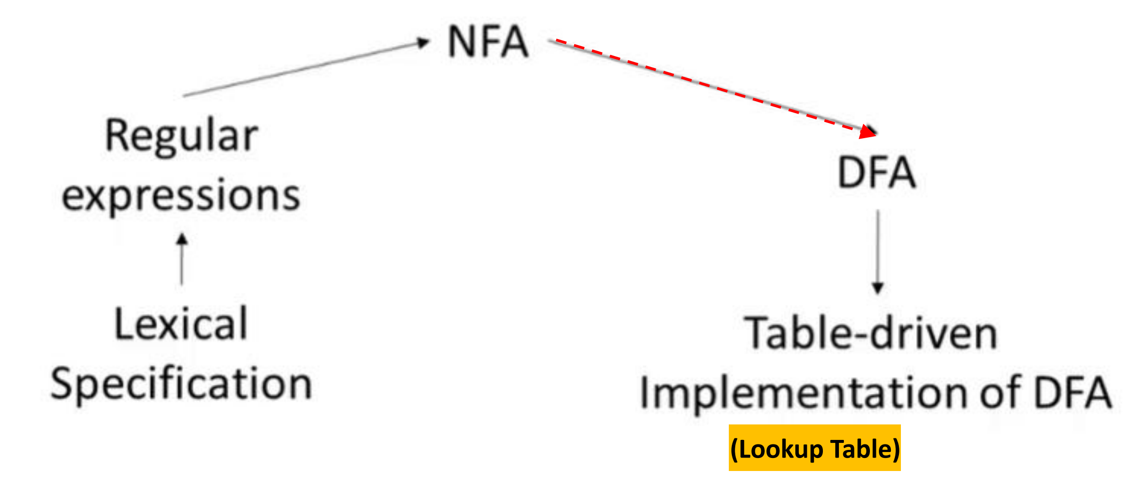
Implement (1+0) REXP to NFA transition as a unit represented by a Rectangle, $(1+0)^*$ iteration implementation as below.

Take back (1+0) implementation.

Step 3: Implement $(1+0)^*2$ REXP to NFA transition

[Figure 2-42] RE to NFA Transmission Example

2.8 NFA to DFA

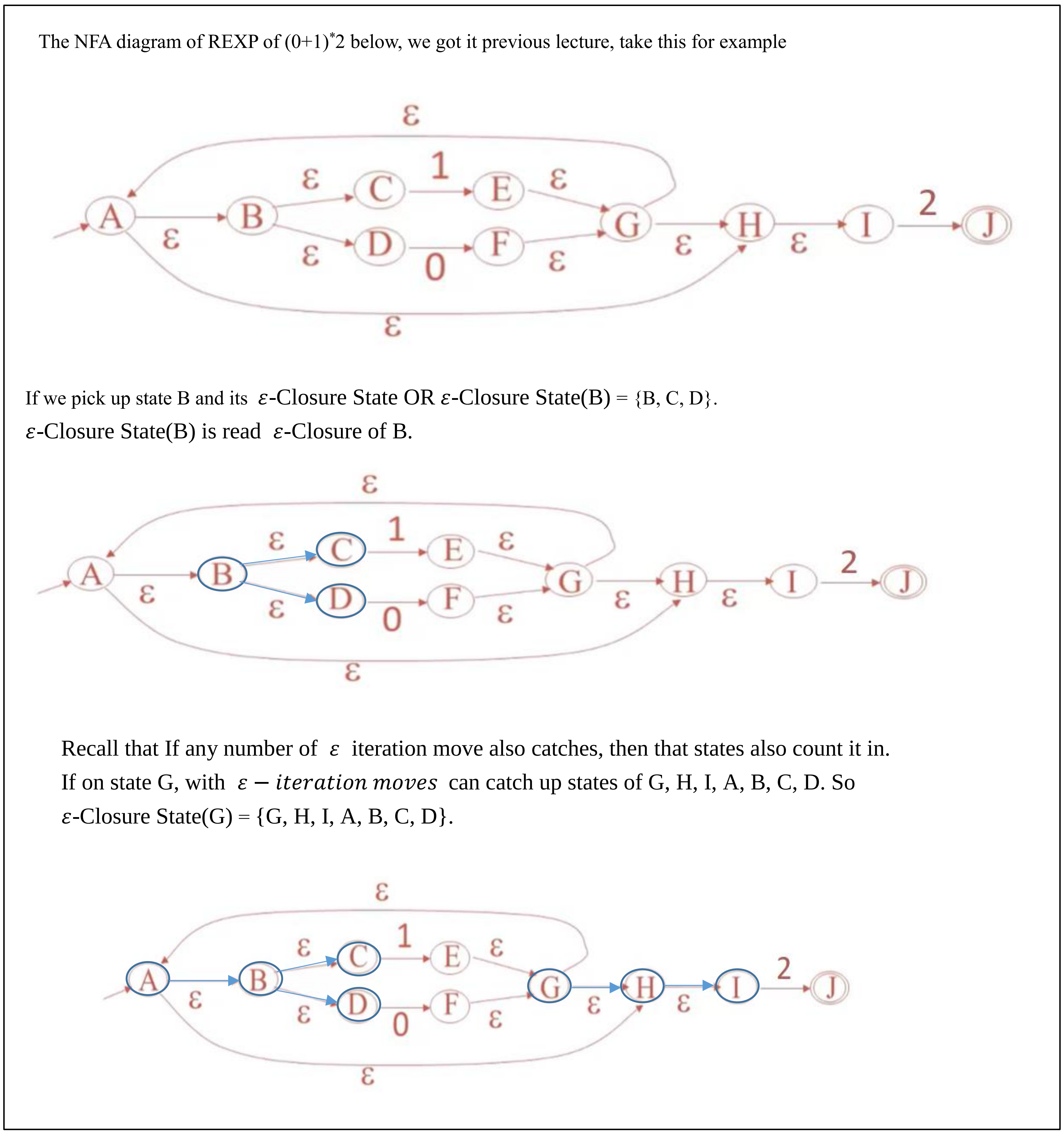


Above is Lexical Specifications implement structure. Last section concentrates on the REXP translates into NFA, and this section concentrates on NFA translation into DFA.

[Figure 2-43] Course Overview

There we got a new conception called ε-Closure State of a state. What's an ε-Closure State of a state?

Pick up states, the states maybe a set of states but just do it a single state, and look at all state that can reach by following only ε moves. If any number of ε iteration move also catches, then that states also count it in.



The NFA diagram of REXP of (0+1)*2 below, we got it previous lecture, take this for example

If we pick up state B and its ε-Closure State OR ε-Closure State(B) = {B, C, D}.
ε-Closure State(B) is read ε-Closure of B.

Recall that If any number of ε iteration move also catches, then that states also count it in.
If on state G, with $\varepsilon - iteration\ moves$ can catch up states of G, H, I, A, B, C, D. So ε-Closure State(G) = {G, H, I, A, B, C, D}.

[Figure 2-44] ε-Closure State

We had known that an NFA may be in many states at any time from previous lecture. So, there's a question: how many different states of a state in the same time then if a move happens?

Imagine there is an NFA consists of N states which is a set element-ed N, a subset of NFA has got $|B| \leq N$ element. And specific element size is 2^N-1 (except $\emptyset$) which also a DAF, that's say 2^N-1 itself Deterministic Automata Finite which consists of 2^N-1 finite set of possible configurations. This is going to give us the seed of the idea for converting an NFA into a DFA. Because all we've to be able to do to convert an NDA into DFA is come up with a way for the DFA to simulate for the behavior of NFA. And the fact of that the NFA can only get into a finite set of configurations even that configurations is very large number ($2^N - 1$,that's say except element $\emptyset$, the large number is finite set of possible configurations).

So, how to implement an arbitrary NFA to an equivalent DFA?

NFA

An NFA consists of states as a set S, starting state as s, final state as F.

Starting state s: $s \epsilon S$

Final state F: $F \epsilon S$

Transition function

a: A Character(s) in the input language

X: Possible transition/move happens set if occurs input character 'a'

Transition function: $a(X) = \{y | x \in X_n, x \overset{a}{\rightarrow} y\}$

function meaning: Input character 'a' apply over states X equals to those states y such that there is some single state x (single state (element) of X_n) such that there's transition from x to y on input character 'a'.

function mean: For a given set of state X, show all the single state that can reach on input character 'a'.

[Figure 2-45] NFA Definition

Now we can define out DFA. And what will the DFA be? where the states are? What is the start state be? What's the final state be? What's the transition function be?

DFA

States: subset (except $\emptyset$) of the states S (S from NFA set of states, S big number but still finite)

The states of the DFA will be all possible states of the NFA. So, there will be one state of DFA for each subset of possible, each possible subset of states of the NFA. The number is very big, but finite. So, this finite state uses as based of DFA.

Start state: ε-Clousure (s)

ε-Clousure of NFA start state s

Final State:

F = $\{X \mid X' \cap F' = \emptyset\}$

F: final state of DFA

F': final state of NFA

X': set of states of NFA

X: set of states of DAF

Transition function:

$$X \overset{a}{\rightarrow} Y$$

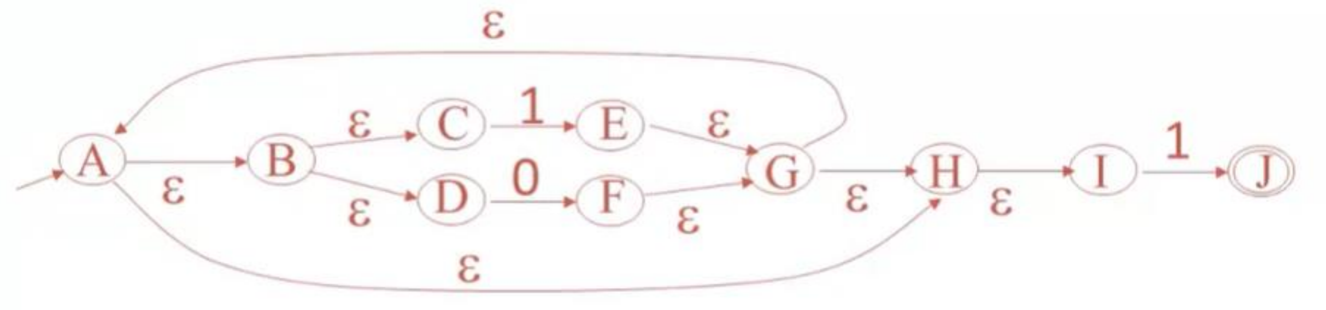
A given state X, there's a state Y when occur input character 'a' transition happens. The transition WOULD be like this:

In state X, set of state we can reach on input 'a' (MAY happen ε-move), so have to take ε-Clousure. The transition WOULD happen when Y equals the ε-Clousure.

Transition Function is $X \overset{a}{\rightarrow} Y, if\ Y = \varepsilon - Clousure(a(X))$.

[Figure 2-46] Transition of NFA to DFA Definition

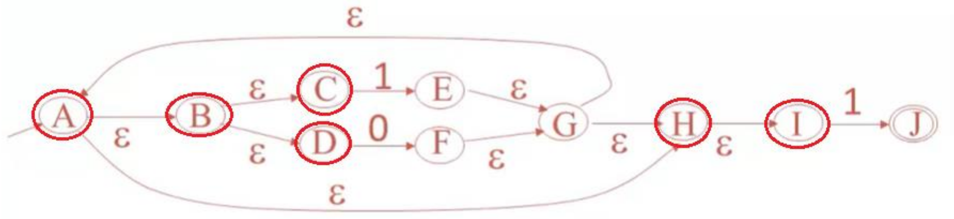
There's an NFA, how to take the transition NFA to DAF?



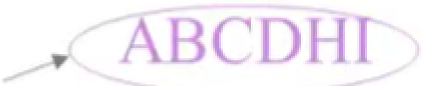
Apparently start it from starting state.

Start state of NFA is state A, then start state of DAF is ε-Clousure of state A in NFA.

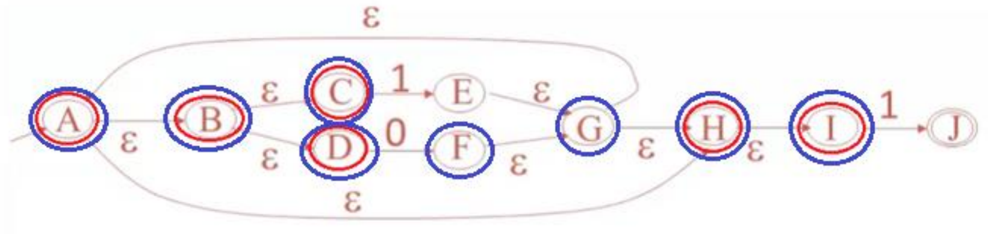
So, ε-Closure (A) = {A,B,C,D,H,I}, DFA start state is subset of ε-Clousure(A).



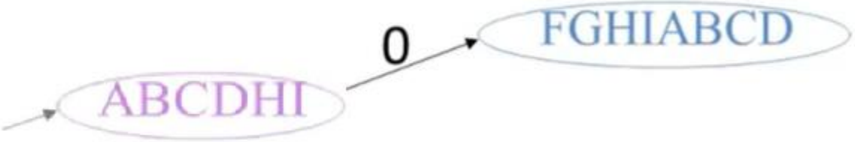
So, we got DFA start state like below.



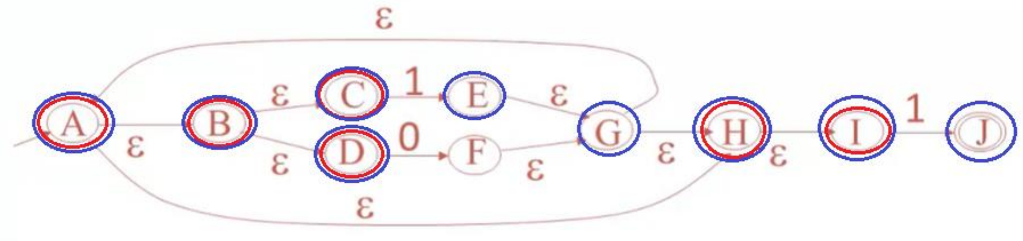
In the start state, what transition would happen if a character occurs on alphabet Σ, of course the alphabet only consists of two member 0, 1, and could happen two transition on start state. Take character '0' for example. Easy to say the start state subset ONLY state D occurs input character '0', make transition get state F, but once get state F WOULD happen lots of ε-move. So, in fact second state of DFA larger which is the ε-Closure of F.



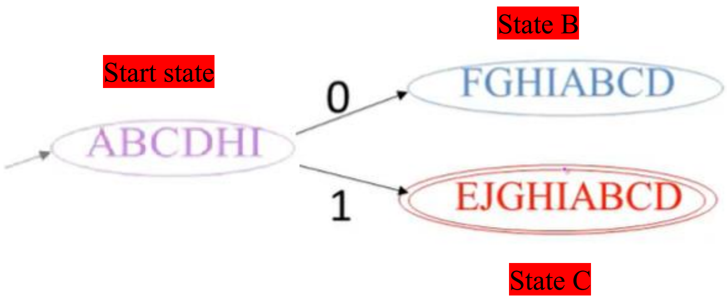
So, second state of DFA like below.



In the start state, occurs character '1' the transition can take shown below.



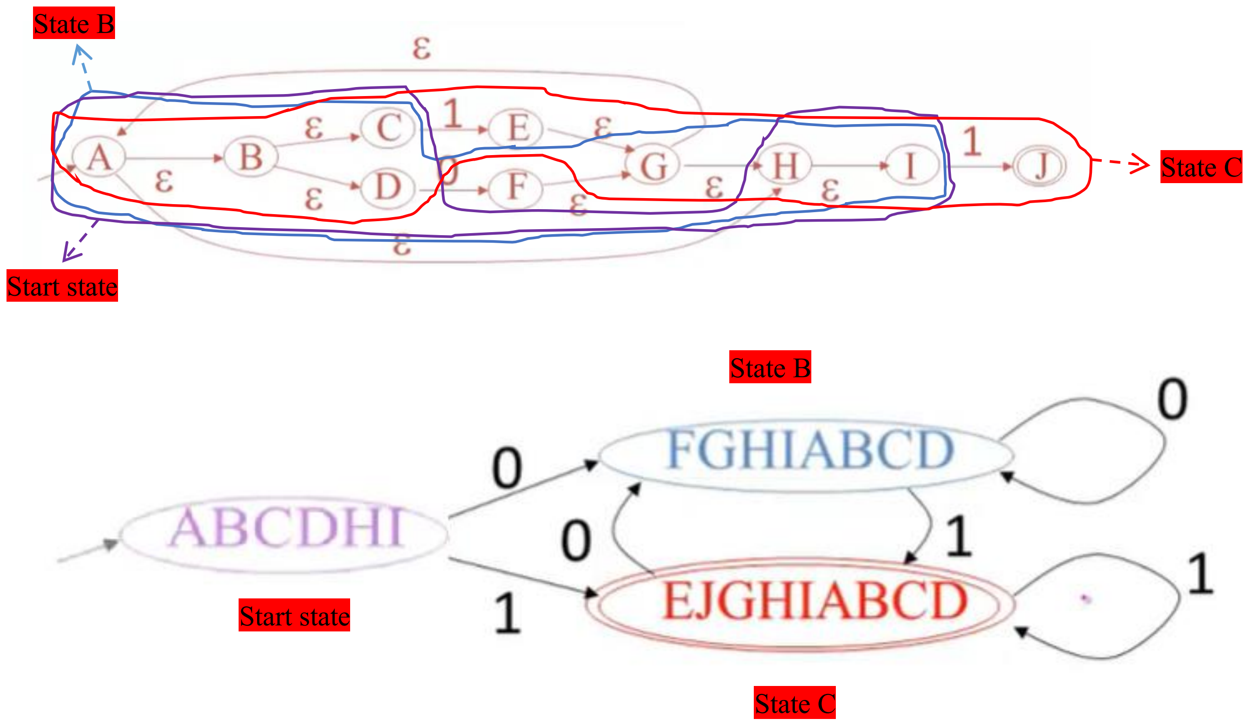
And DFA layout also shown below. Because final state J also can reach, so the DFA state draw double circles.



When in state B OR state C what would be the next state?

[Figure 2-47] Transition of NFA to DFA Example-1

In state B, State C occur inputs again, state transmission is below.

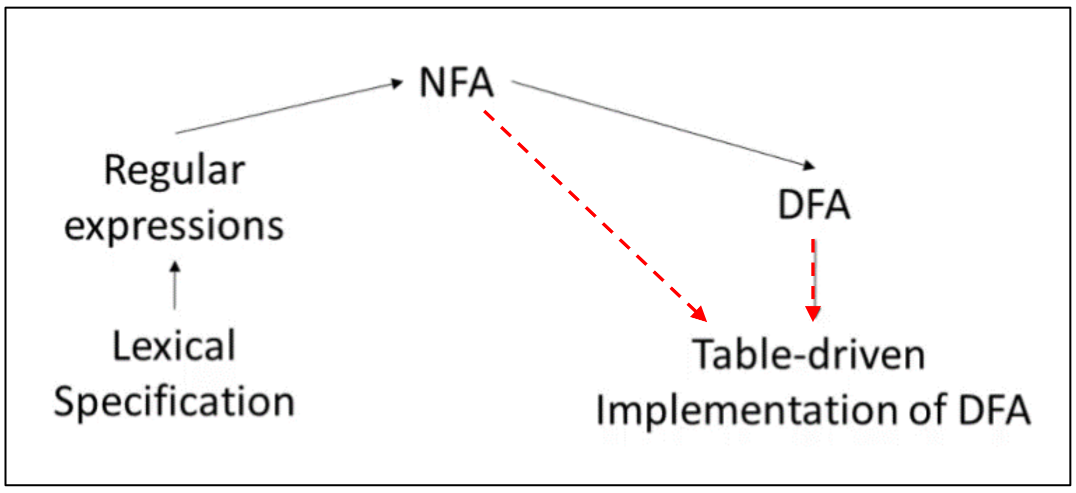


Note that The DFA simulate the NFA, so every move in DFA machine it records set of possible states that the NFA could be in and it WOULD accept a string infinitely if the NFA could accept the string.

[Figure 2-48] Transition of NFA to DFA Example-2

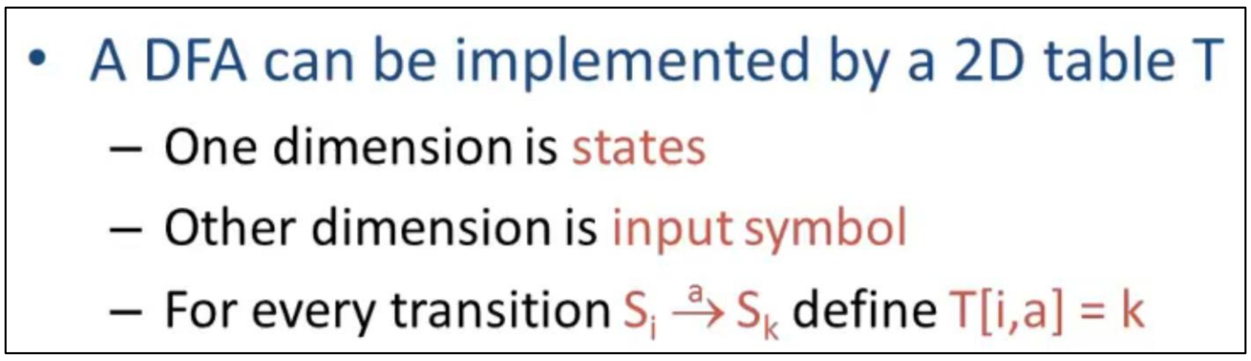
2.9 Implement FA

This section we'll focus on the implementation of Finite Automata. Actually, we use variety of different techniques to implement. Below is the flow chart we seen last videos, it says how the Lexical Analyzer constructed. This section discusses on DFA's implementation, actually should say the chart is not quite completely accurate because sometimes we don't go all the way to DFA, sometimes we stop NFA and implement directly. This part we WOULD discuss them all.



[Figure 2-49] Two Diffident Implementation Logics

How to implement DFA's?



- A DFA can be implemented by a 2D table T
  - One dimension is states
  - Other dimension is input symbol
  - For every transition $S_i \xrightarrow{a} S_k$ define T[i,a] = k

[Figure 2-50] DFA's Implementation Logics

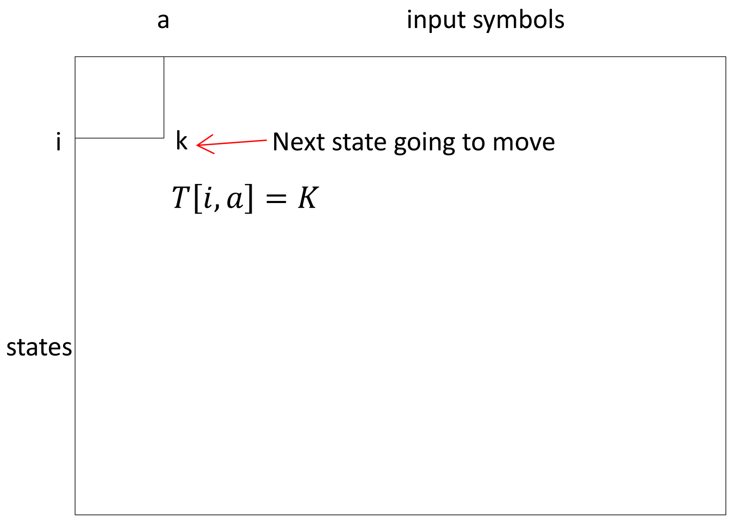
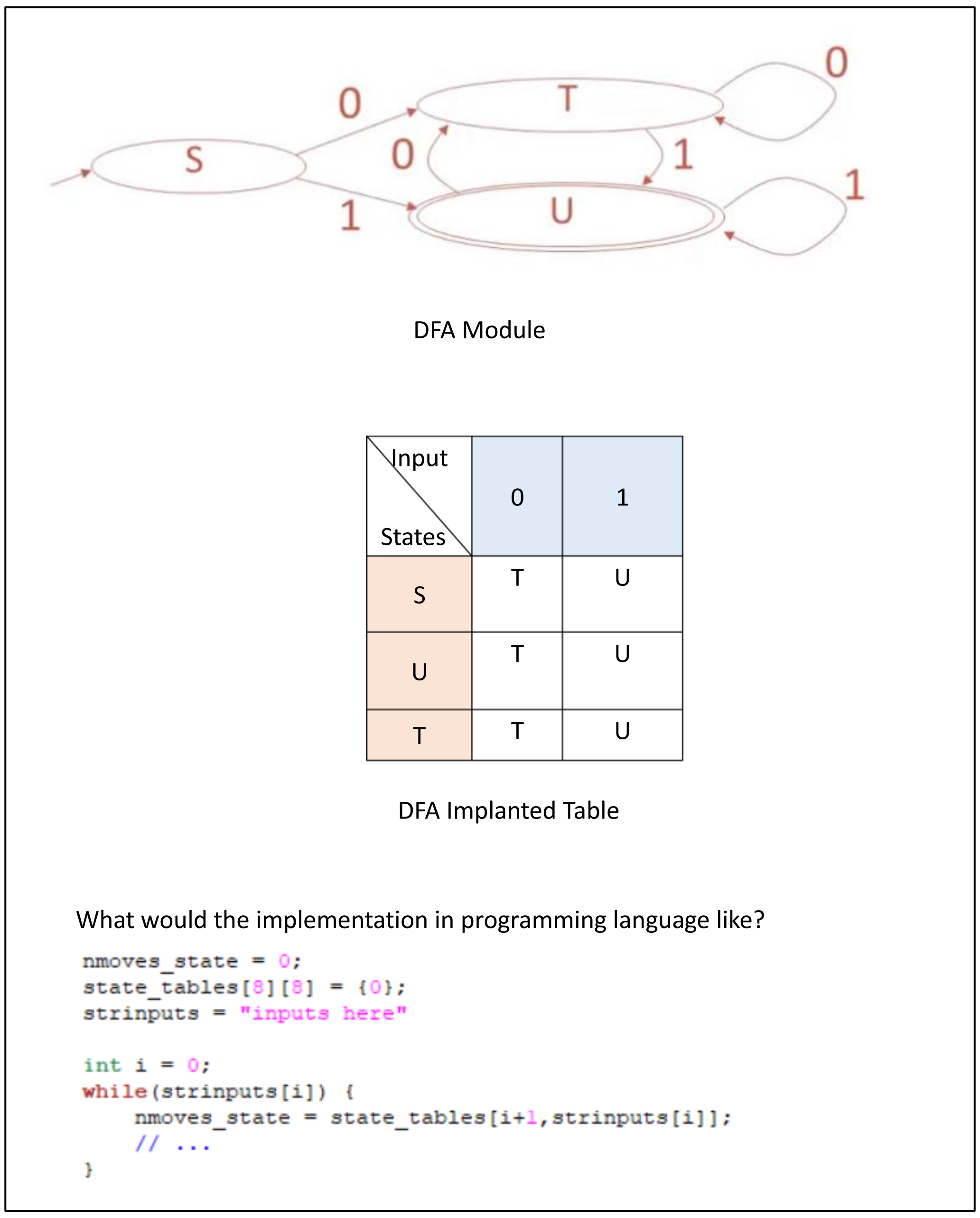


Figure [2-51] DFA’s Implemented Table



DFA Module

Input / States	0	1
S	T	U
U	T	U
T	T	U

DFA Implanted Table

What would the implementation in programming language like?

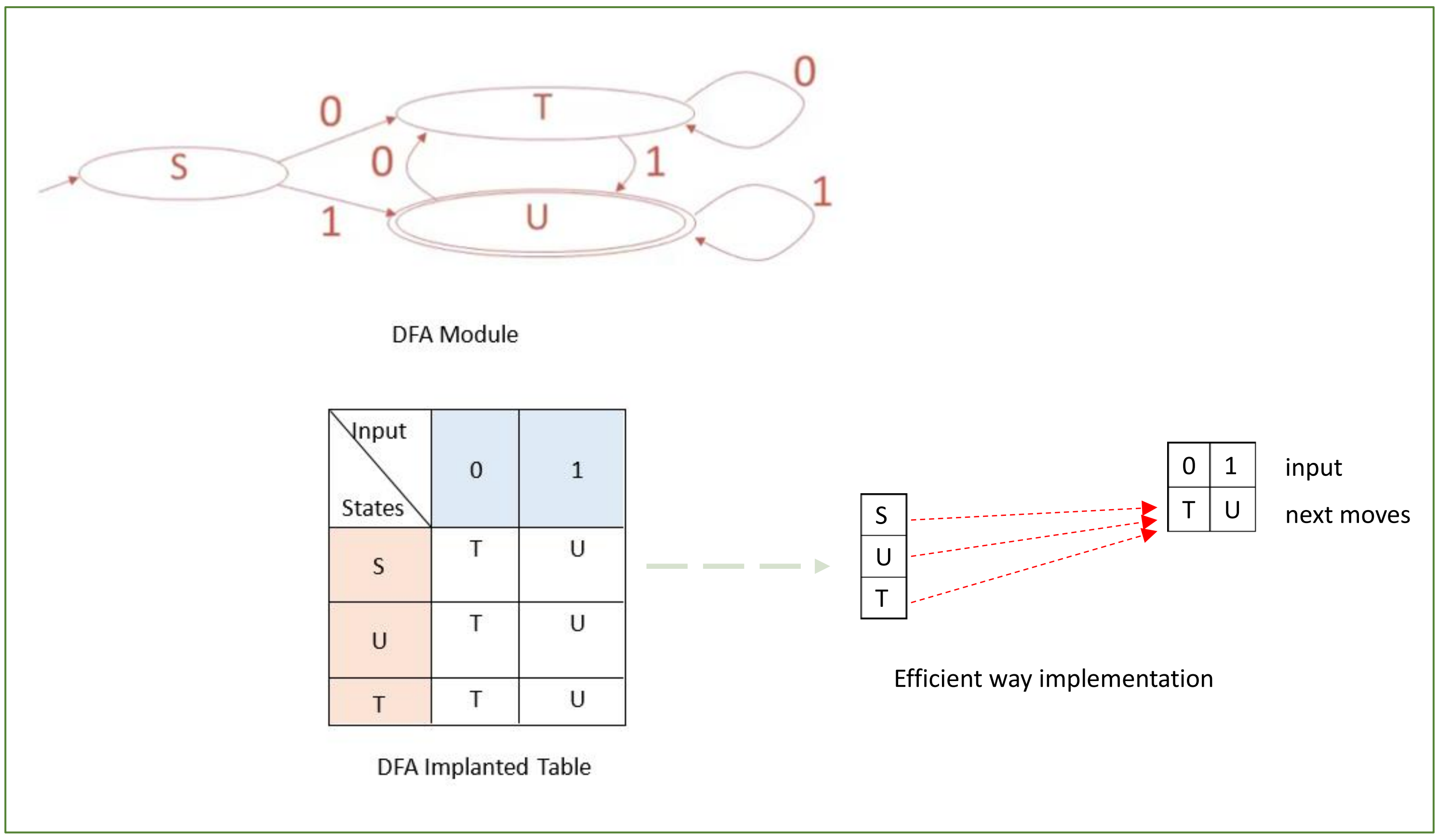
```
nmoves_state = 0;
state_tables[8][8] = {0};
strinputs = "inputs here"

int i = 0;
while(strinputs[i]) {
    nmoves_state = state_tables[i+1,strinputs[i]];
    // ...
}
```

Figure [2-52] DFA’s Implementation Example

If we make Table-driven implementation of DFA like above table that’d a catastrophe, because if an NFA has N state and simulated by DFA, the DFA got 2^N-1 state, the number very huge needs huge memory. Is there another efficient way to make table scientifically?

Actually, we have, the table above has duplicated some states. Also, in programing got so many duplicated lines, so we can share that same’s.



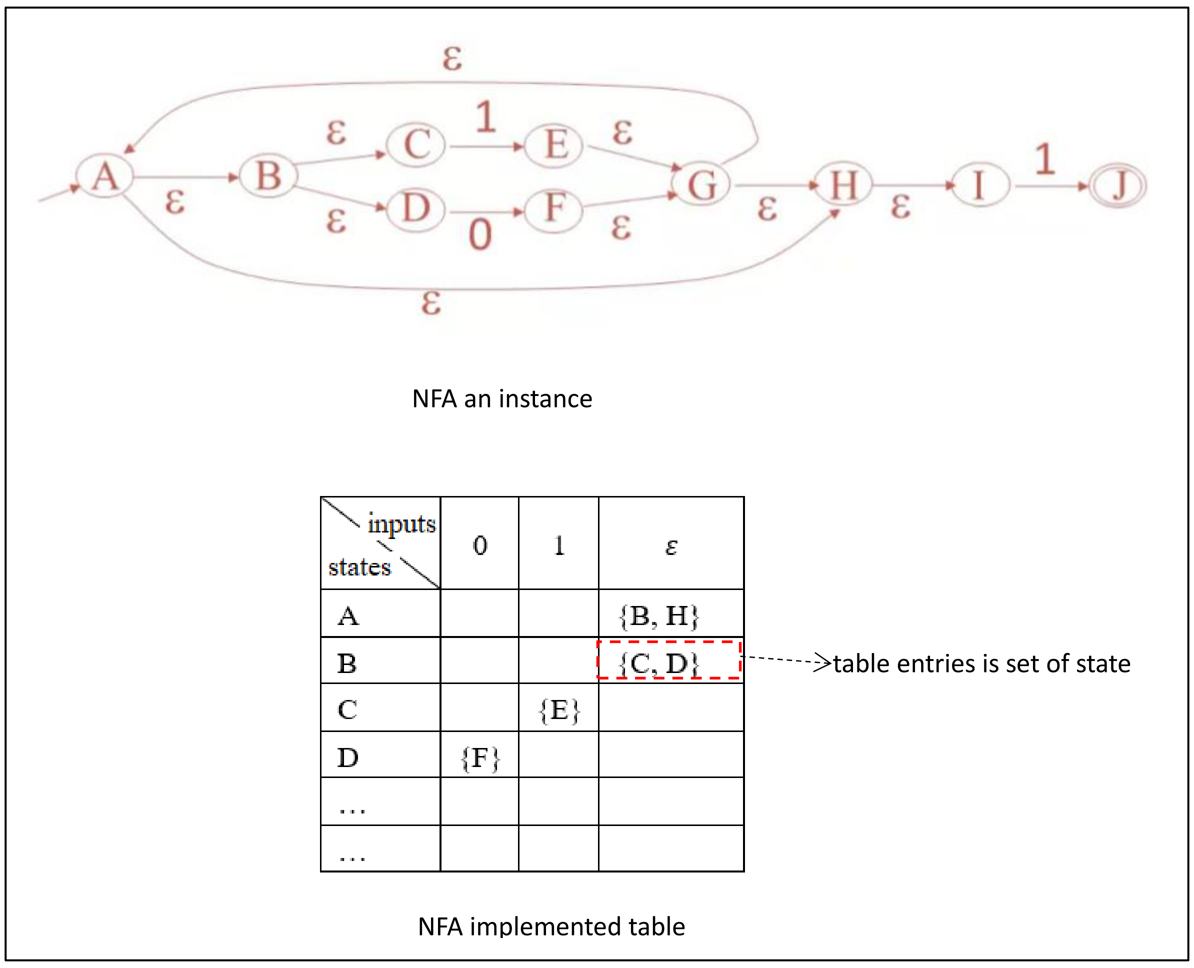
States \ Input	0	1
S	T	U
U	T	U
T	T	U

S
U
T

0	1
T	U

Figure [2-53] DFA's Efficient Implementation Example

If always simulate NFA generate equivalent DFA, the DFA very huge 2N-1 states and the implemented table also got huge entries. So sometimes, we choose generate table directly from NFA like said above. Here is a discussion how to generate table directly from NFA.



states \ inputs	0	1	ε
A			{B, H}
B			{C, D}
C		{E}	
D	{F}		
…			
…			

Figure [2-54] DFA's Efficient Implementation Example

The table directly implemented from NFA and generated is a little bit huge, if optimize is better.

To summarize:

- NFA -> DFA conversion is key

  Use REXP to convert NFA to DFA.

- Tools trade between speed and space

  DFA's:

  Faster, less compact

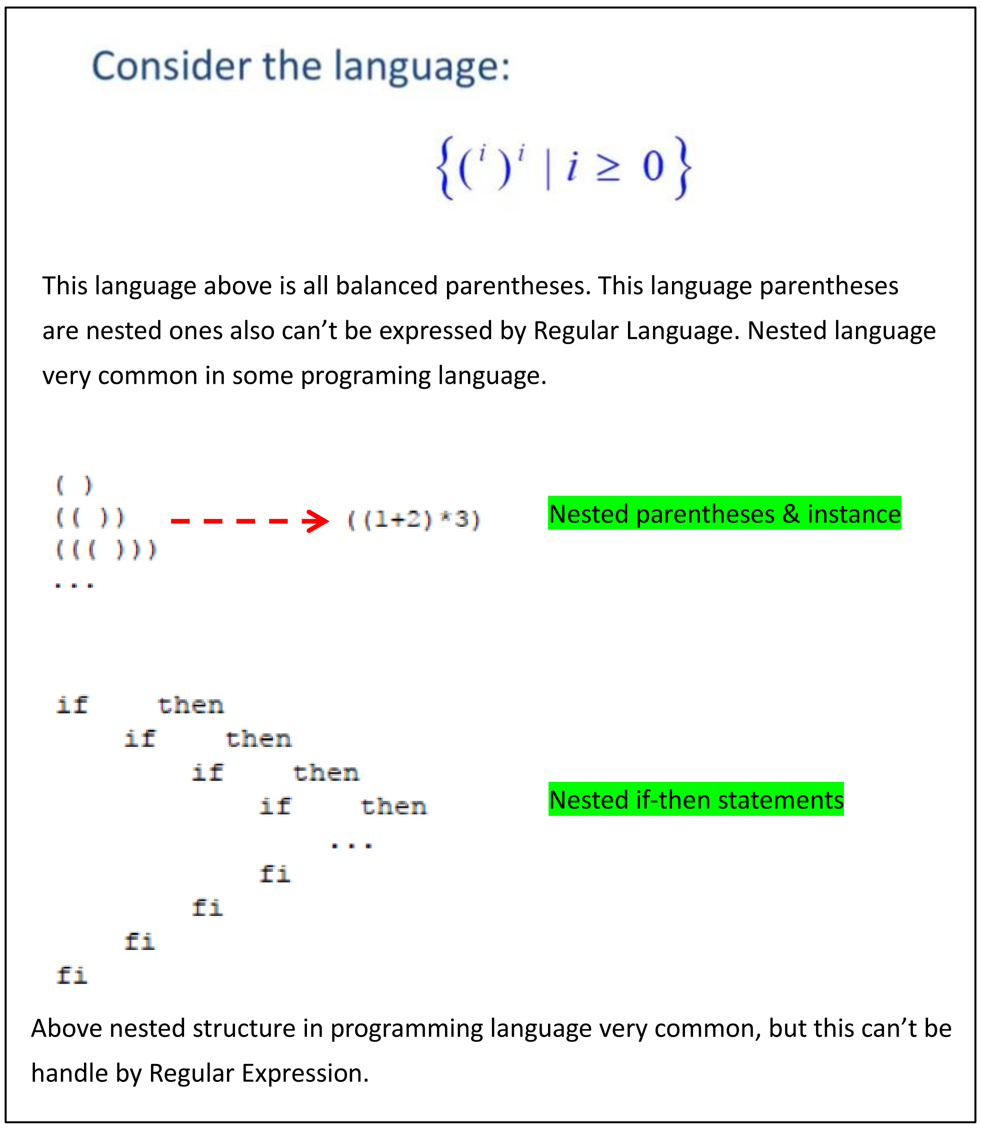
  NFA's:

Slower, concise

3 Parsing

3.1 Introduction

Actually, the weakest Formal Language is widely used. The difficulty of Regular Languages is that a lot of language are simply not regular. There're some pretty important languages that can't be expressed using Regular Expression OR Finite Automata.



Consider the language:

$$\{(^i)^i \mid i \geq 0\}$$

This language above is all balanced parentheses. This language parentheses are nested ones also can't be expressed by Regular Language. Nested language very common in some programing language.

```
( )
(( ))   - - - - >  ((1+2)*3)
((( )))
...
```

Nested parentheses & instance

```
if    then
      if    then
            if    then
                  if    then
                        ...
                  fi
            fi
      fi
fi
```

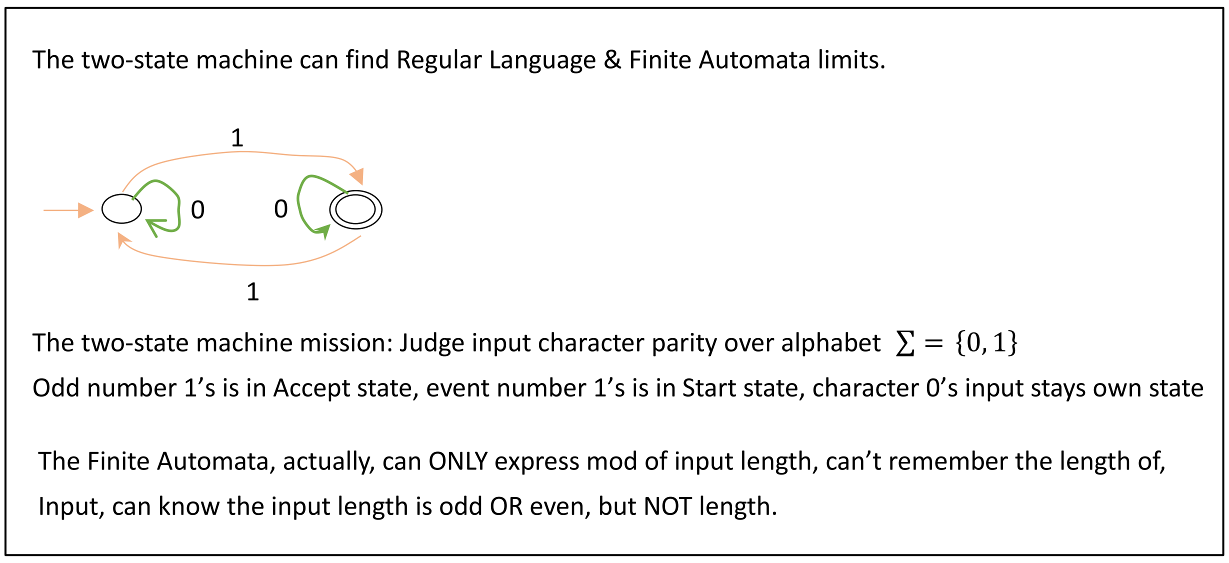
Nested if-then statements

Above nested structure in programming language very common, but this can't be handle by Regular Expression.

[Figure 3-1] Regular Expression Shortcoming

Though Regular Language isn't universal then, what the Regular Language can expression? And why the Regular Language aren't sufficient for recognizing arbitrary nesting structure?

Actually, it's very easy to illustrate the limitations of Regular Language and Finite Automata.



The two-state machine can find Regular Language & Finite Automata limits.

The two-state machine mission: Judge input character parity over alphabet $\sum = \{0, 1\}$

Odd number 1's is in Accept state, event number 1's is in Start state, character 0's input stays own state

The Finite Automata, actually, can ONLY express mod of input length, can't remember the length of, Input, can know the input length is odd OR even, but NOT length.

[Figure 3-2] Finite Automata Ability

So, what does Parsing actually do? What are the inputs and outputs of Parsing?

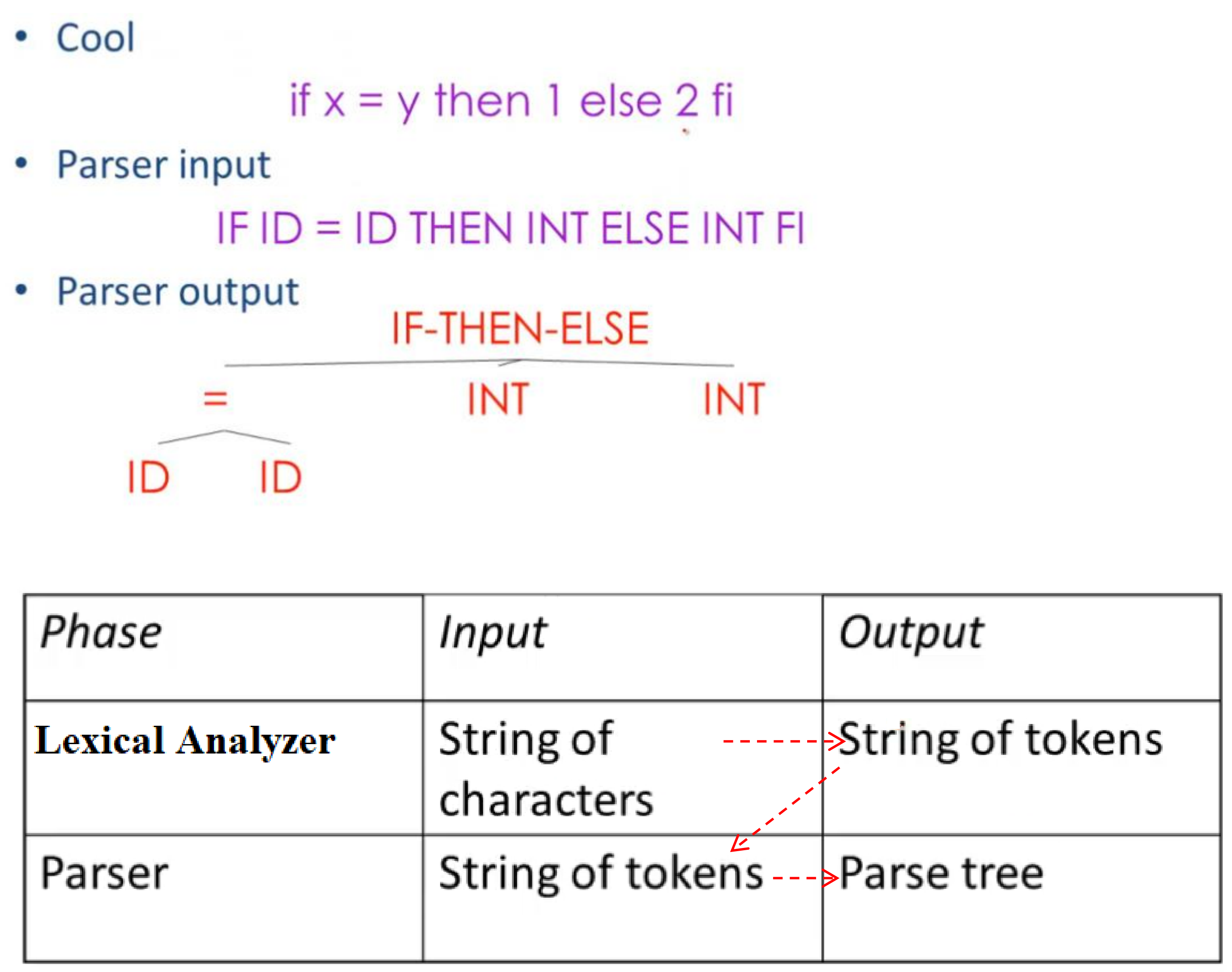
Take the tokens from Lexical Analyzer, handle the tokens, structure the programing language with Parse Tree (also call Syntax Tree).

- Input

  Sequences of tokens from Lexical Analyzer.

- Output

Parse tree of program text.



- Cool

if x = y then 1 else 2 fi

- Parser input

IF ID = ID THEN INT ELSE INT FI

- Parser output

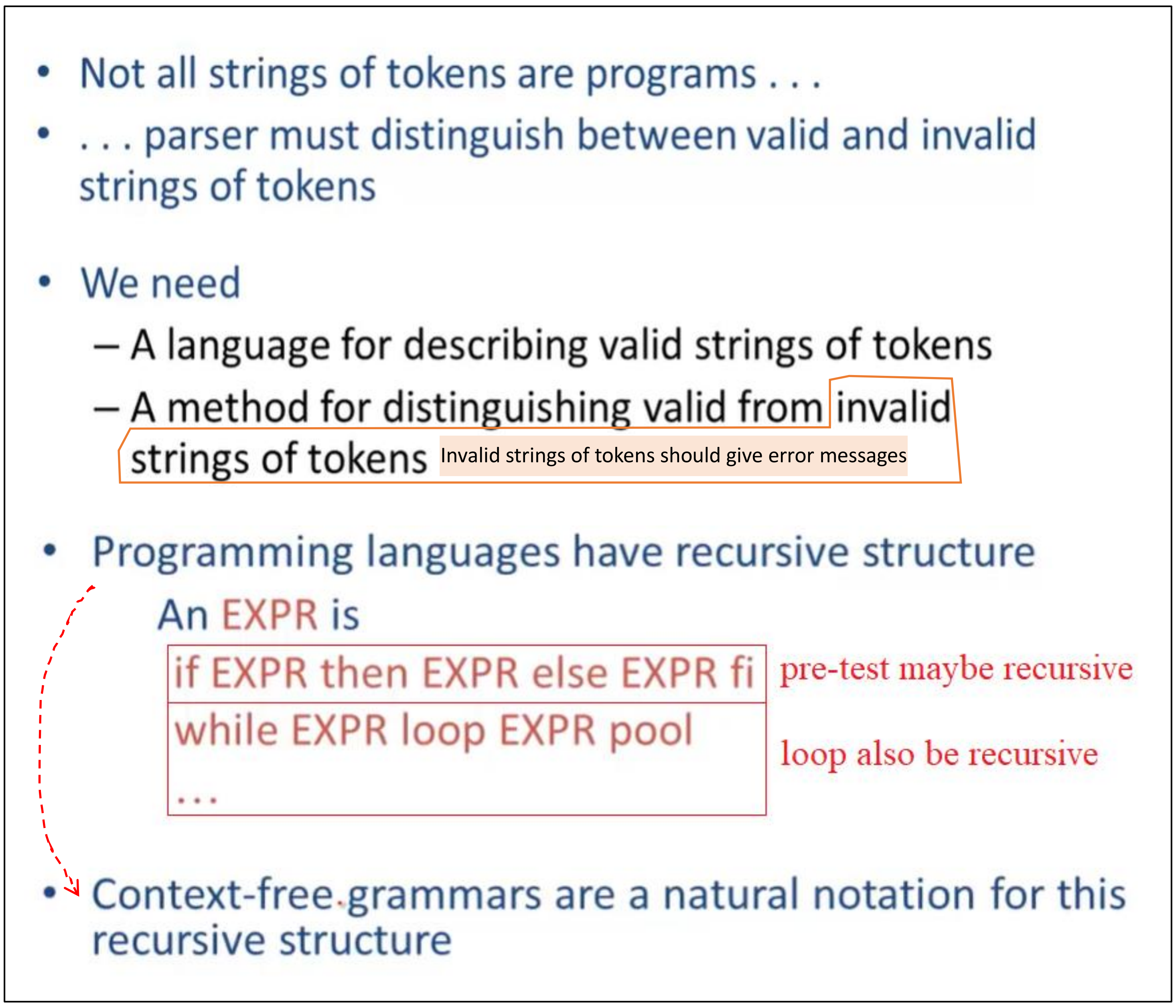
Phase	*Input*	*Output*
Lexical Analyzer	String of characters	String of tokens
Parser	String of tokens	Parse tree

Couple of things need to mention.

1) The Parse Tree MAYBE implicant. Some Compiler maybe never build a full Parse Tree, some of Compiler does.
2) Some Compiler MAYBE combine Lexical Analyzer & Parser phase, everything done in phase parser, while most Compiler still separates.

[Figure 3-3] Phase Lexical Analyzer & Parser Relationships

3.2 Context-Free Grammar



- Not all strings of tokens are programs . . .
- . . . parser must distinguish between valid and invalid strings of tokens
- We need
  - A language for describing valid strings of tokens
  - A method for distinguishing valid from invalid strings of tokens Invalid strings of tokens should give error messages
- Programming languages have recursive structure

  An EXPR is

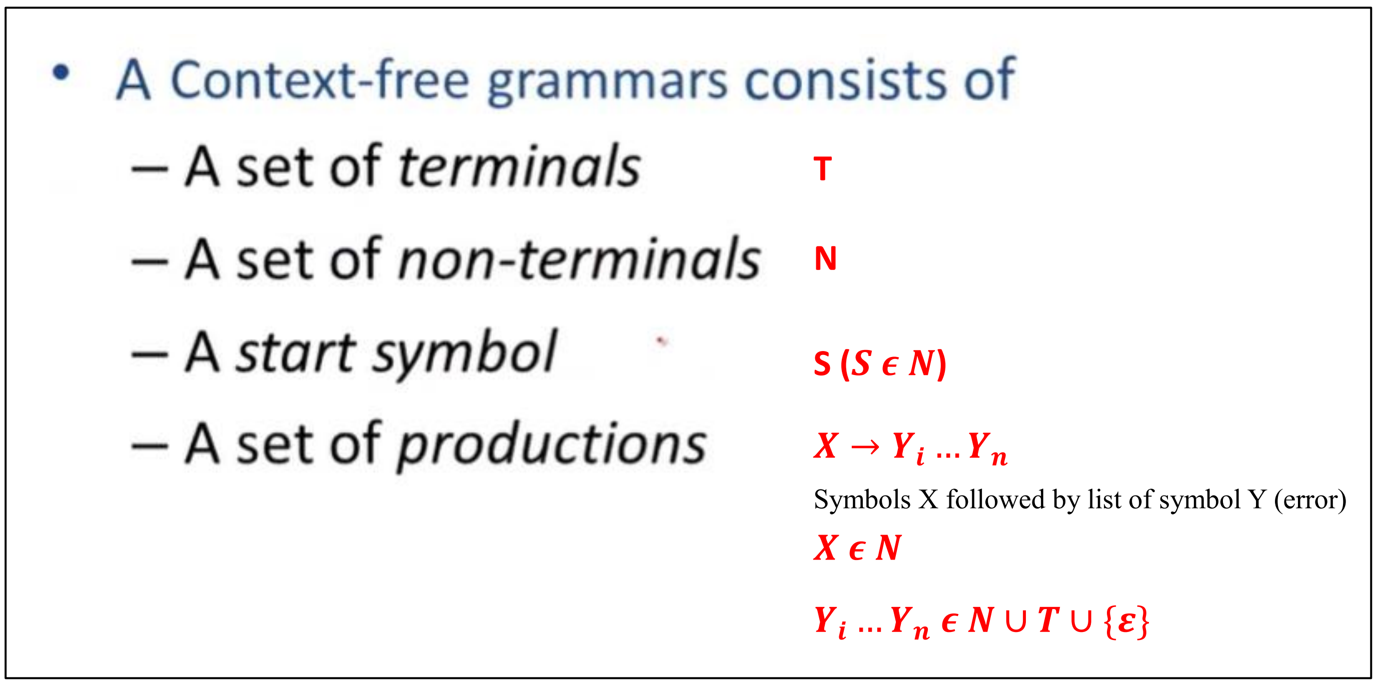
  if EXPR then EXPR else EXPR fi pre-test maybe recursive

  while EXPR loop EXPR pool loop also be recursive

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

[Figure 3-4] What's a Context-Free Grammar

From above we got that Context-free grammars mission is to handle recursive structure.



- A Context-free grammars consists of
  - A set of *terminals* T
  - A set of *non-terminals* N
  - A *start symbol* S ($S \epsilon N$)
  - A set of *productions* $X \to Y_i \dots Y_n$

Symbols X followed by list of symbol Y (error)

$X \epsilon N$

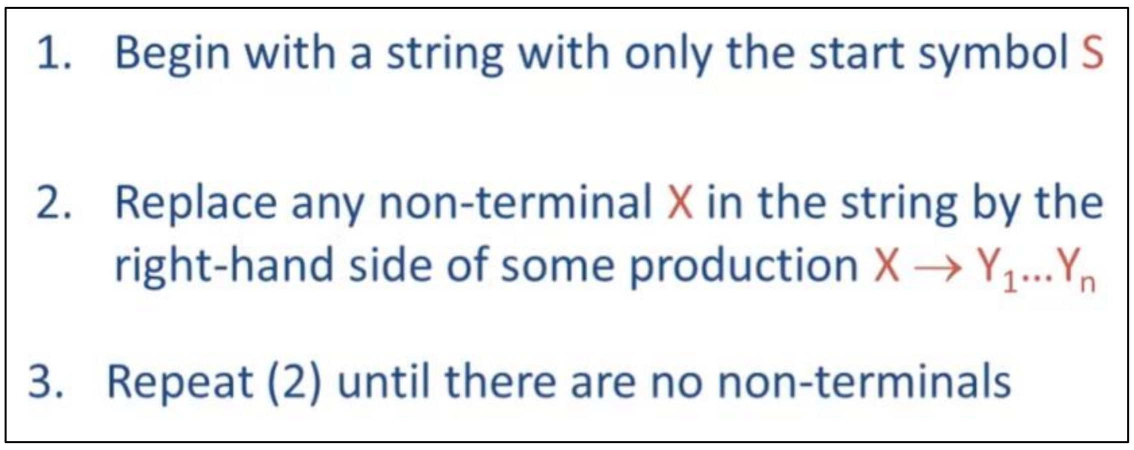
$Y_i \dots Y_n \epsilon N \cup T \cup \{\varepsilon\}$

[Figure 3-5] Context-Free Grammar Terminology

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

[Figure 3-6] Context-Free Grammar Example

What does a production mean? This means whatever we see, we can replace it by string of symbols on right hand side. So, productions can be read as replacement rule. The process of replace productions call **Context-Free Derivation**. Suppose here is a production $S \rightarrow (S)$, this means wherever see ‘S’, it can be substitute by ‘(S)’ on the left hand.



1. Begin with a string with only the start symbol S

2. Replace any non-terminal X in the string by the right-hand side of some production $X \rightarrow Y_1 \dots Y_n$

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

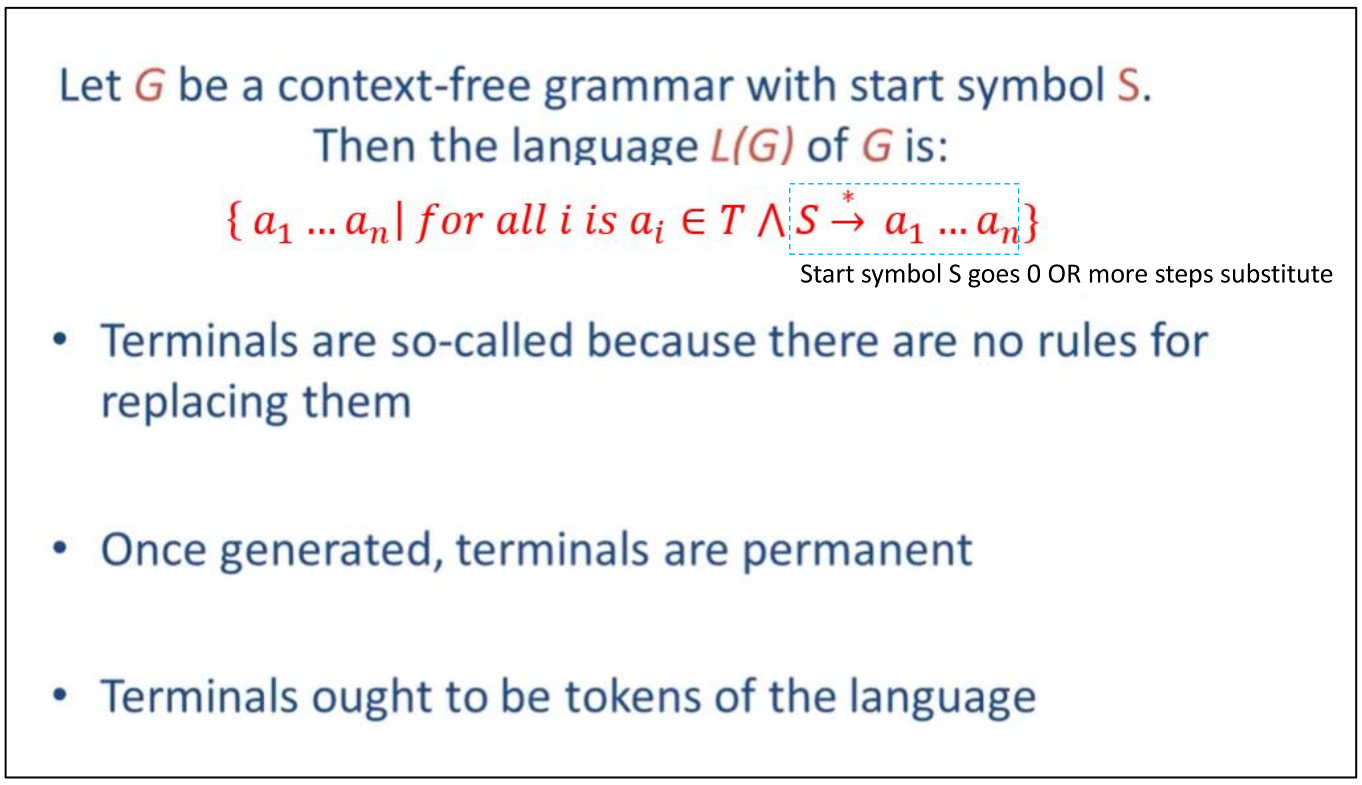
[Figure 3-7] Formal Definition of Production Rule

$$S = \alpha_0 \rightarrow \alpha_1 \rightarrow \alpha_2 \rightarrow \cdots \rightarrow \alpha_n$$

$$\alpha_0 \overset{*}{\rightarrow} \alpha_n$$

Production recursively substitute until all symbols is non-terminals, the whole process MAYBE 0 OR N steps to make.

[Figure 3-8] Production Rule Example



Let *G* be a context-free grammar with start symbol S.
Then the language *L(G)* of *G* is:

$$\{ a_1 \dots a_n |\, for\ all\ i\ is\ a_i \in T \wedge \boxed{S \overset{*}{\rightarrow} a_1 \dots a_n} \}$$

Start symbol S goes 0 OR more steps substitute

- Terminals are so-called because there are no rules for replacing them

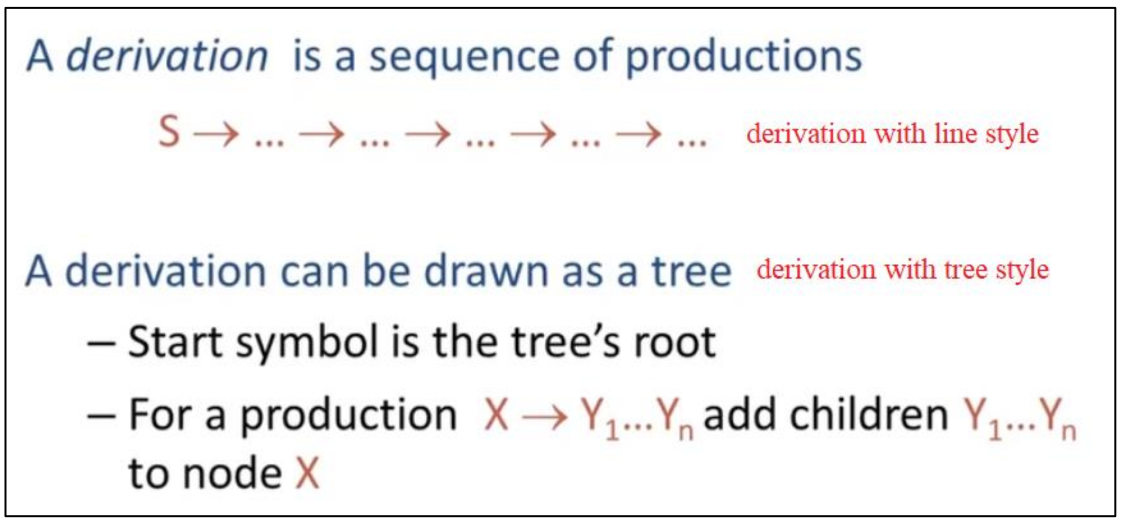
- Once generated, terminals are permanent

- Terminals ought to be tokens of the language

[Figure 3-9] Definition of Context-Free Grammar Language

3.3 Derivation

Derivation is a sequence of productions, one production applied each step all alone the way, cause of also a production the derivation also starts with starting symbol. The derivation directly drawn as a line, actually, also can be drawn a tree instead of a line.



A *derivation* is a sequence of productions

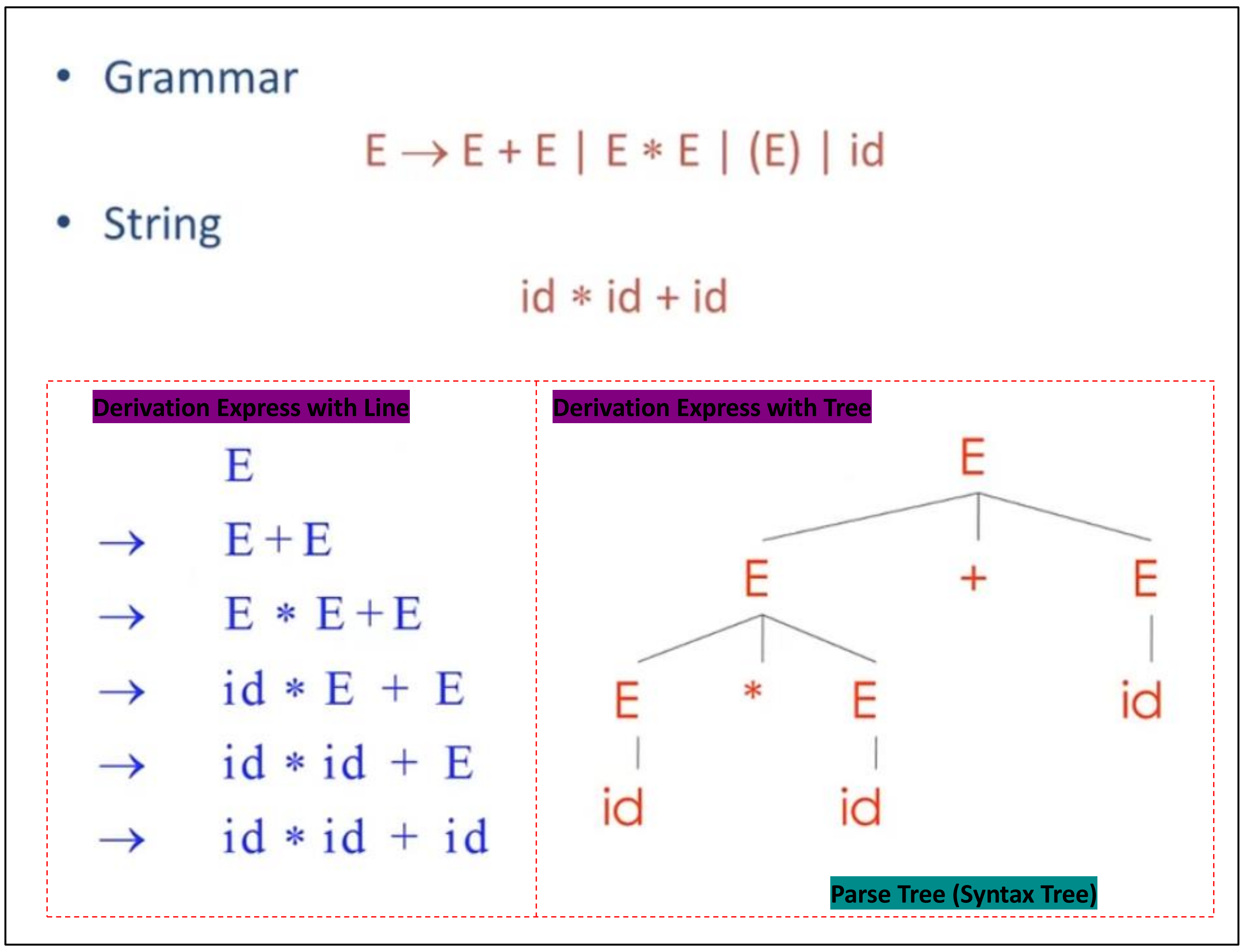
$S \rightarrow \dots \rightarrow \dots \rightarrow \dots \rightarrow \dots \rightarrow \dots$ derivation with line style

A derivation can be drawn as a tree derivation with tree style

– Start symbol is the tree’s root

– For a production $X \rightarrow Y_1 \dots Y_n$ add children $Y_1 \dots Y_n$ to node X

[Figure 3-10] Two Different Style of Express Derivation



- Grammar

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

- String

$$id * id + id$$

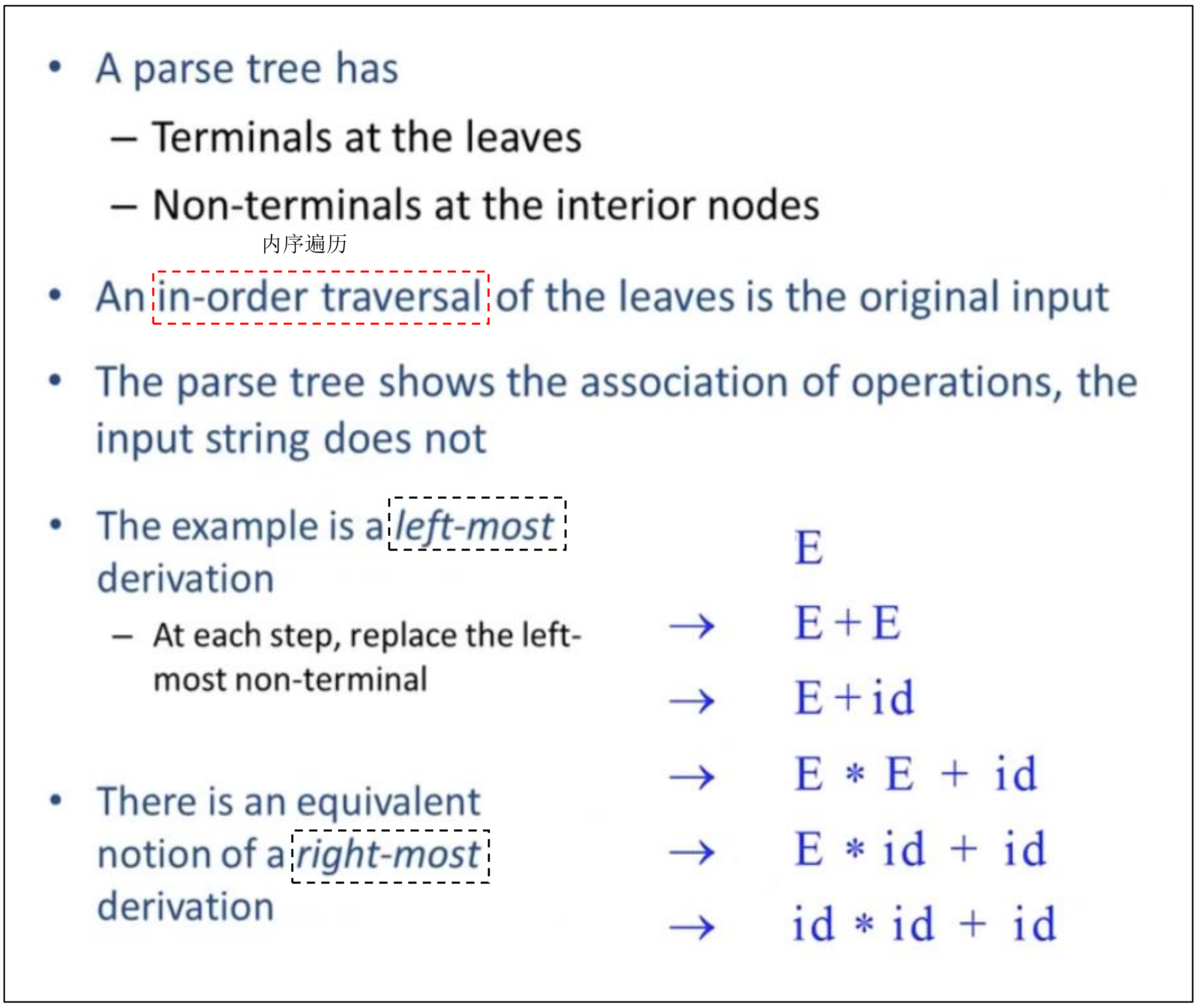
Derivation Express with Line

E
$\rightarrow E + E$
$\rightarrow E * E + E$
$\rightarrow id * E + E$
$\rightarrow id * id + E$
$\rightarrow id * id + id$

Derivation Express with Tree

Parse Tree (Syntax Tree)

[Figure 3-11] Two Different Style of Express Derivation



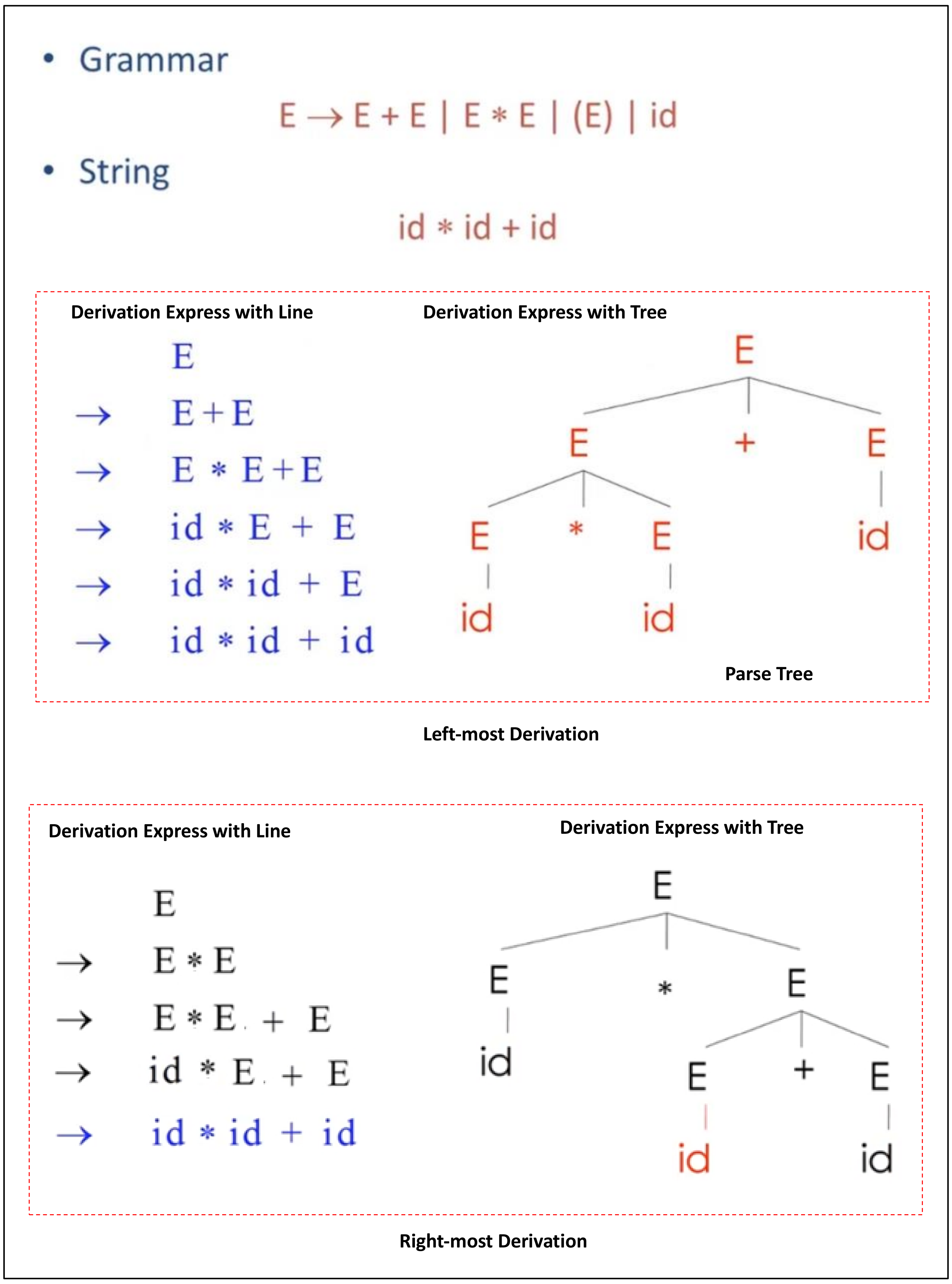
- A parse tree has
  - Terminals at the leaves
  - Non-terminals at the interior nodes
- An in-order traversal of the leaves is the original input

内序遍历

- The parse tree shows the association of operations, the input string does not
- The example is a *left-most* derivation
  - At each step, replace the left-most non-terminal
- There is an equivalent notion of a *right-most* derivation

E
$\rightarrow E + E$
$\rightarrow E + id$
$\rightarrow E * E + id$
$\rightarrow E * id + id$
$\rightarrow id * id + id$

[Figure 3-12] Parse Tree Features



- Grammar

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

- String

$id * id + id$

Derivation Express with Line

$$
\begin{aligned}
& E \\
\rightarrow\ & E + E \\
\rightarrow\ & E * E + E \\
\rightarrow\ & id * E + E \\
\rightarrow\ & id * id + E \\
\rightarrow\ & id * id + id
\end{aligned}
$$

Derivation Express with Tree

Parse Tree

Left-most Derivation

Derivation Express with Line

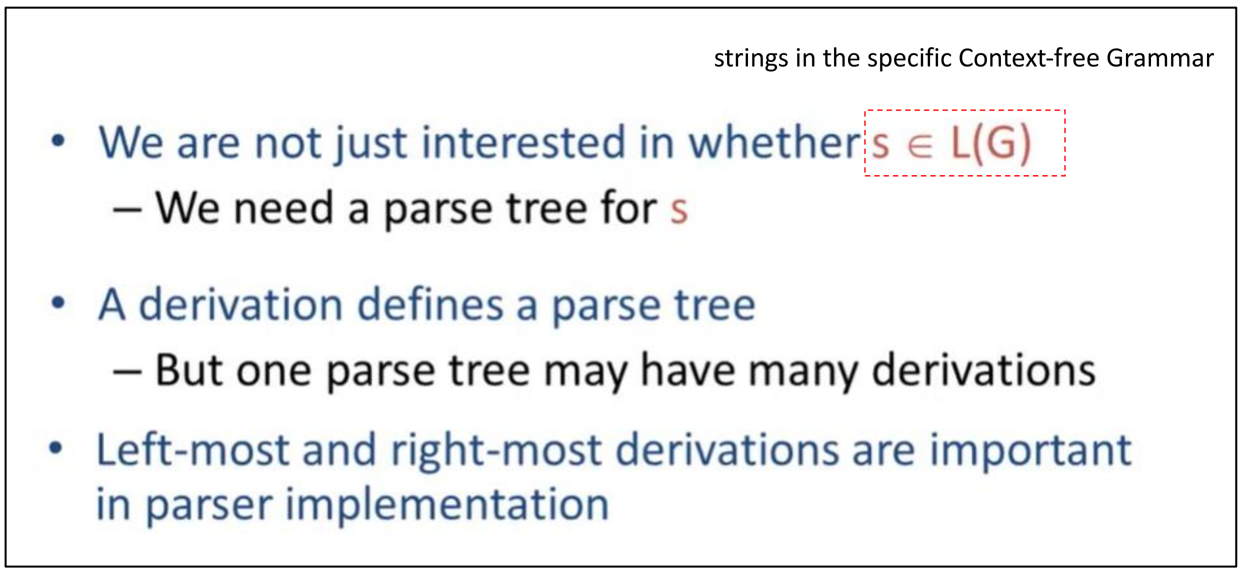
$$
\begin{aligned}
& E \\
\rightarrow\ & E * E \\
\rightarrow\ & E * E + E \\
\rightarrow\ & id * E + E \\
\rightarrow\ & id * id + id
\end{aligned}
$$

Derivation Express with Tree

Right-most Derivation

[Figure 3-13] Left-most Derivation VS. Right-most Derivation

Note that left-most derivation and right-most derivation have the same Parse Tree.

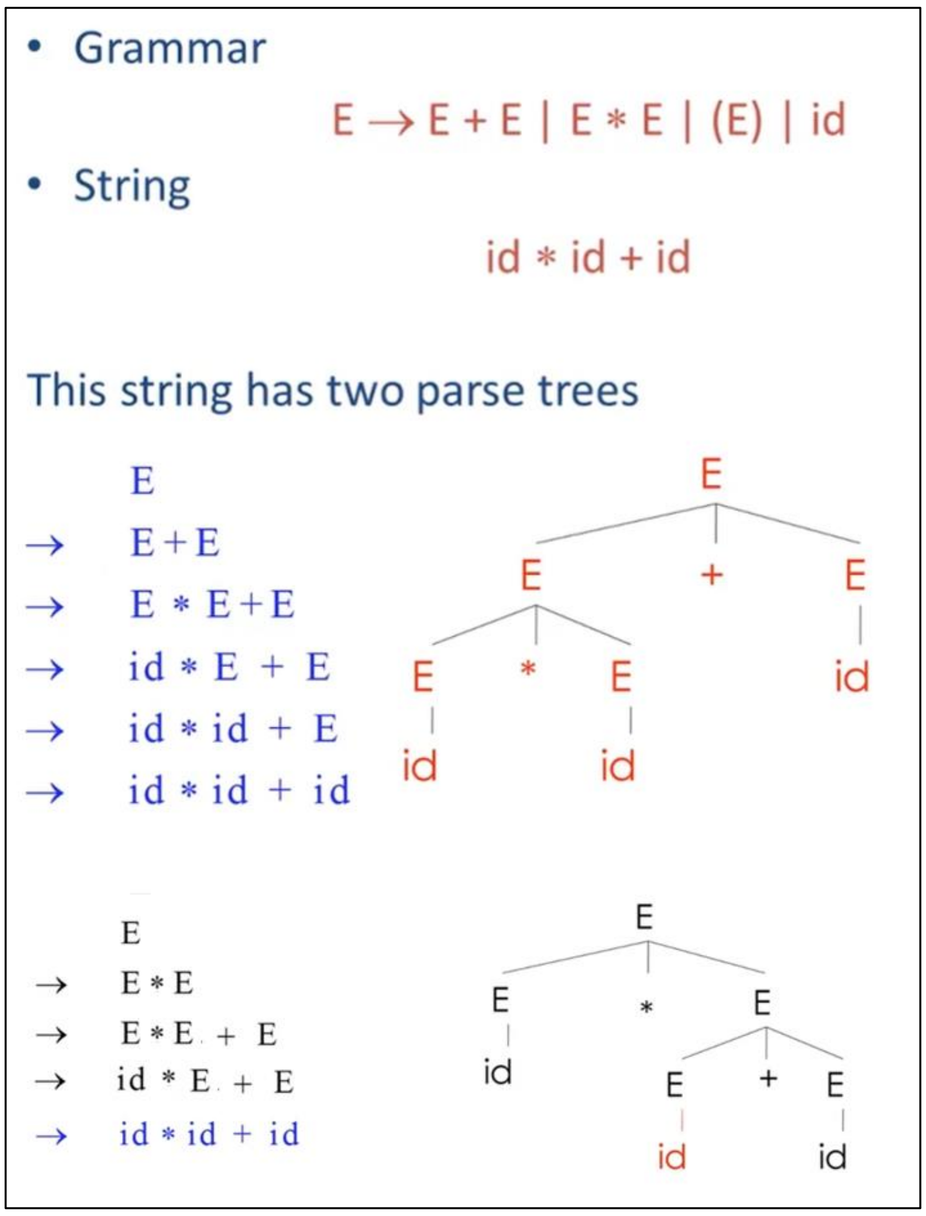


strings in the specific Context-free Grammar

- We are not just interested in whether $s \in L(G)$
  - We need a parse tree for s
- A derivation defines a parse tree
  - But one parse tree may have many derivations
- Left-most and right-most derivations are important in parser implementation

[Figure 3-14] Derivation Summarize

3.4 Ambiguity



- Grammar

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

- String

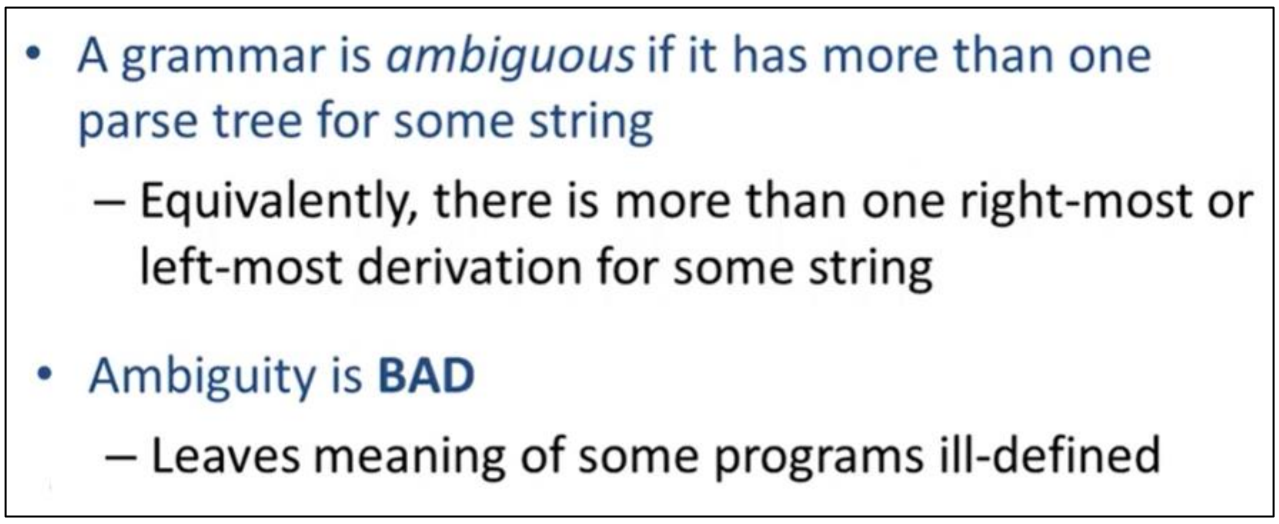
$$id * id + id$$

This string has two parse trees

$$\begin{aligned} & E \\ \rightarrow\ & E + E \\ \rightarrow\ & E * E + E \\ \rightarrow\ & id * E + E \\ \rightarrow\ & id * id + E \\ \rightarrow\ & id * id + id \end{aligned}$$

$$\begin{aligned} & E \\ \rightarrow\ & E * E \\ \rightarrow\ & E * E + E \\ \rightarrow\ & id * E + E \\ \rightarrow\ & id * id + id \end{aligned}$$

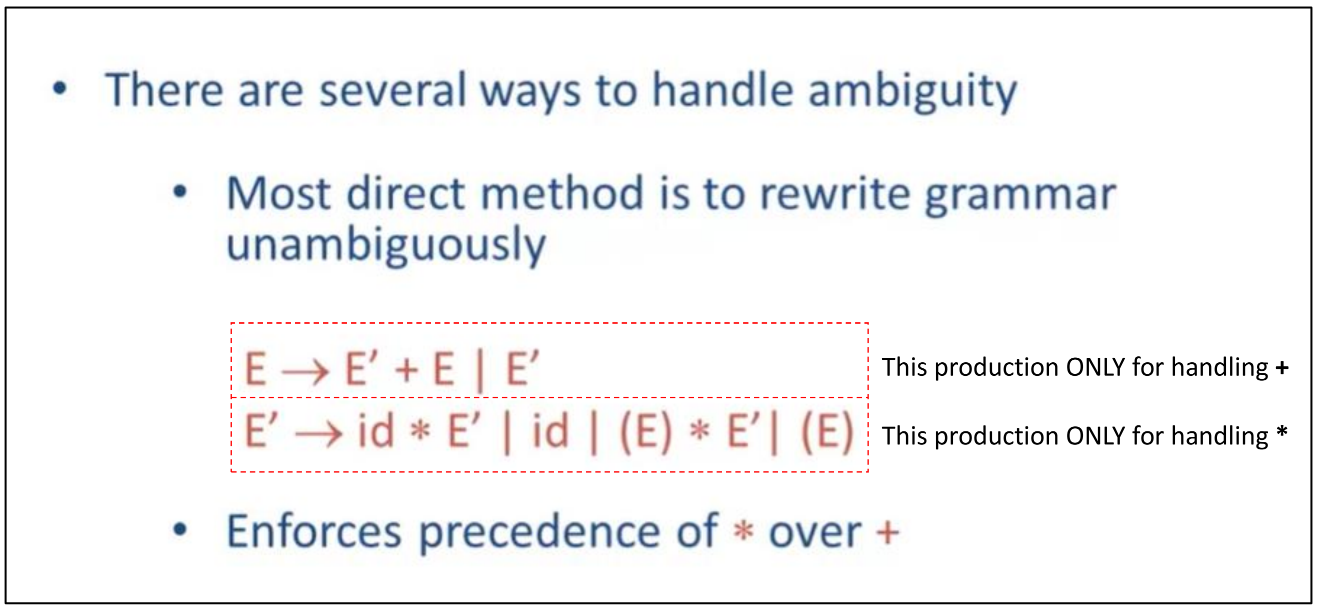
[Figure 3-15] Two Different Parse Tree on The Same Input String



- A grammar is *ambiguous* if it has more than one parse tree for some string
  - Equivalently, there is more than one right-most or left-most derivation for some string
- Ambiguity is **BAD**
  - Leaves meaning of some programs ill-defined

[Figure 3-16] Ambiguity Summarize

Think about it, what if got Ambiguity in program, HOW to handle & fix it?



- There are several ways to handle ambiguity
  - Most direct method is to rewrite grammar unambiguously

$$E \rightarrow E' + E \mid E'$$

This production ONLY for handling +

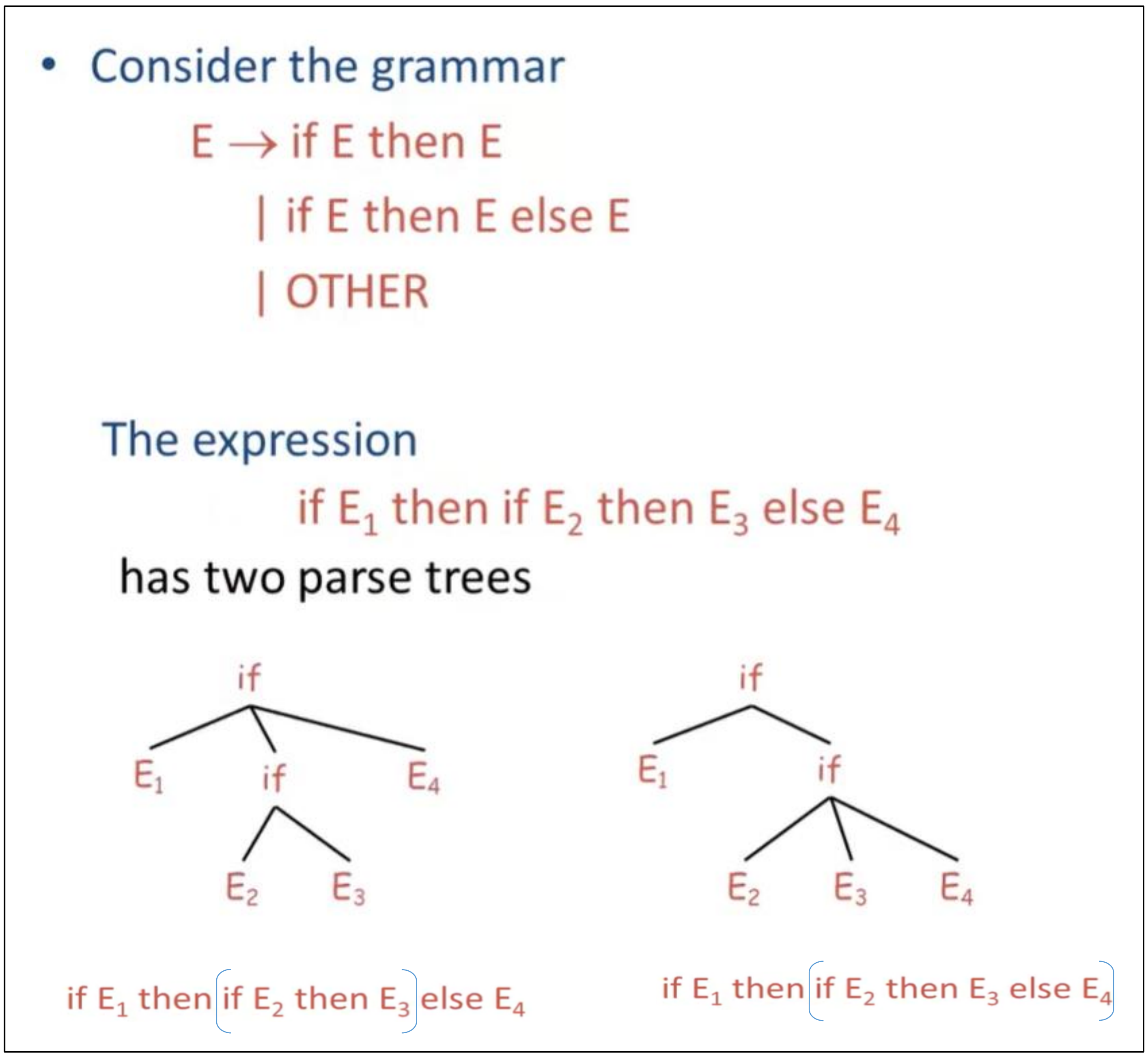
$$E' \rightarrow id * E' \mid id \mid (E) * E' \mid (E)$$

This production ONLY for handling *

  - Enforces precedence of * over +

[Figure 3-17] Fix Ambiguity

There's another example for ambiguousness. This is an if-else derivation, you know the else statement is optional.



- Consider the grammar

  $E \rightarrow$ if E then E

  | if E then E else E

  | OTHER

The expression

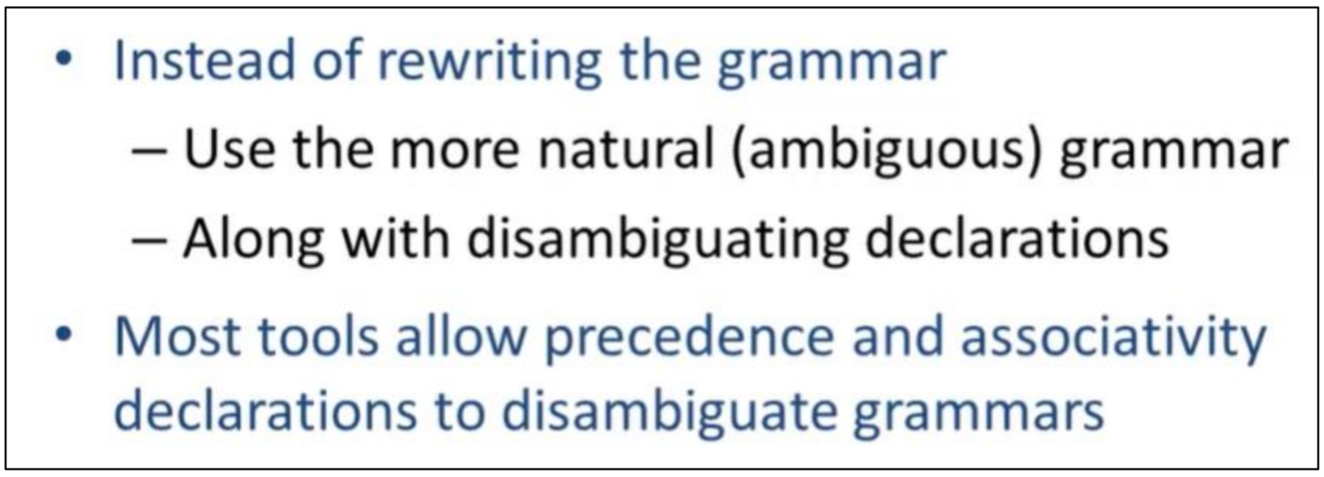
if E_1 then if E_2 then E_3 else E_4

has two parse trees

if E_1 then (if E_2 then E_3) else E_4

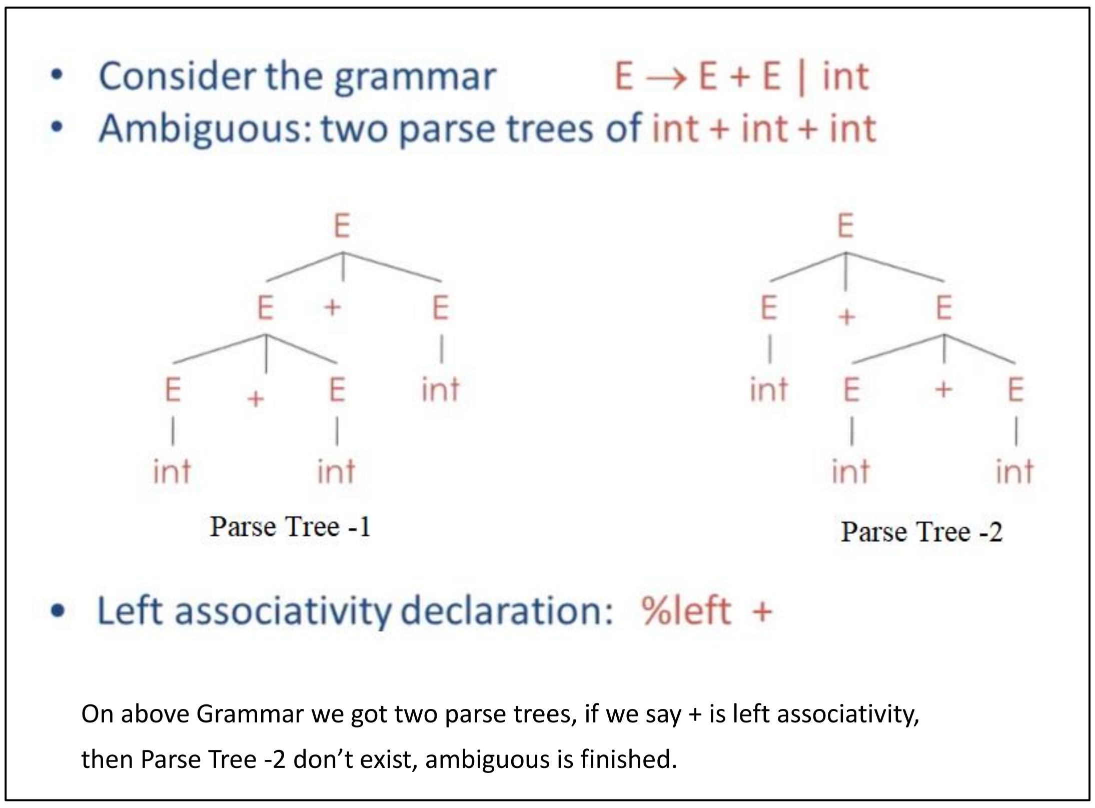
if E_1 then (if E_2 then E_3 else E_4)

[Figure 3-18] Ambiguity Example: if-then-else statement



- Instead of rewriting the grammar
  - Use the more natural (ambiguous) grammar
  - Along with disambiguating declarations
- Most tools allow precedence and associativity declarations to disambiguate grammars

[Figure 3-19] Ambiguity Example: if-then-else statement



- Consider the grammar E → E + E | int
- Ambiguous: two parse trees of int + int + int

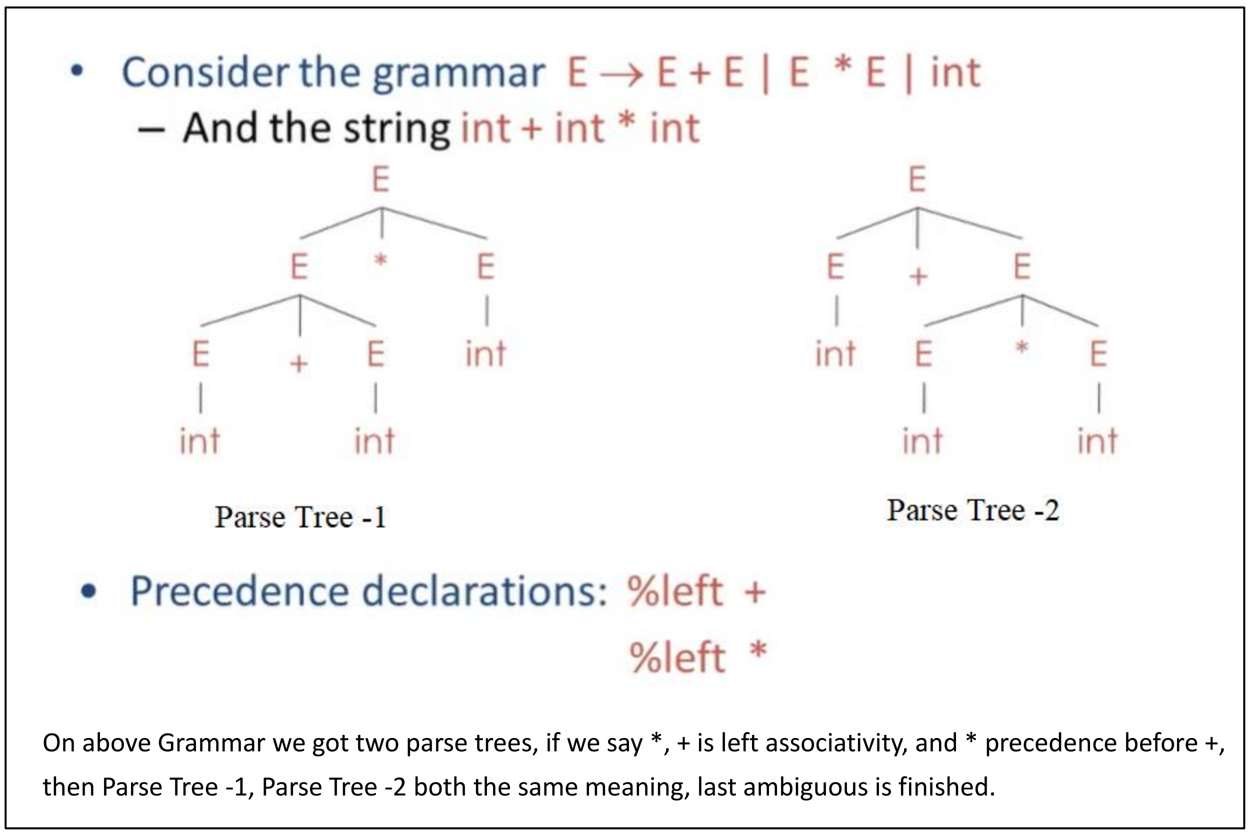
Parse Tree -1

Parse Tree -2

- Left associativity declaration: %left +

On above Grammar we got two parse trees, if we say + is left associativity, then Parse Tree -2 don't exist, ambiguous is finished.

[Figure 3-20] Ambiguity Solves Example-1



- Consider the grammar E → E + E | E * E | int
  - And the string int + int * int

Parse Tree -1

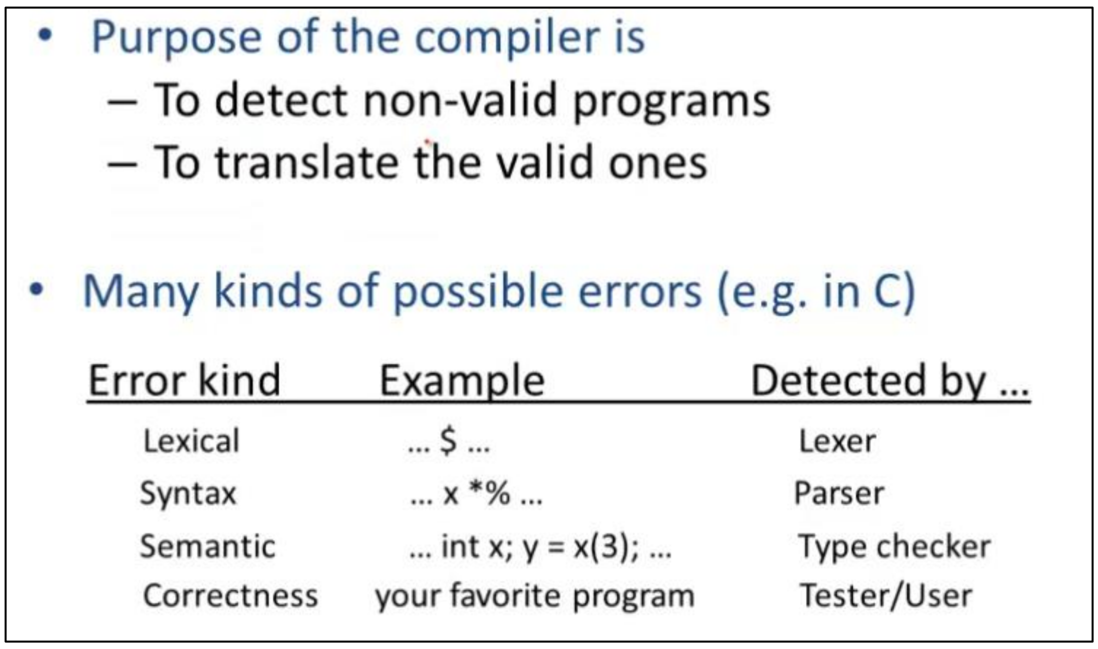
Parse Tree -2

- Precedence declarations: %left +
  %left *

On above Grammar we got two parse trees, if we say *, + is left associativity, and * precedence before +, then Parse Tree -1, Parse Tree -2 both the same meaning, last ambiguous is finished.

[Figure 3-21] Ambiguity Solves Example-2

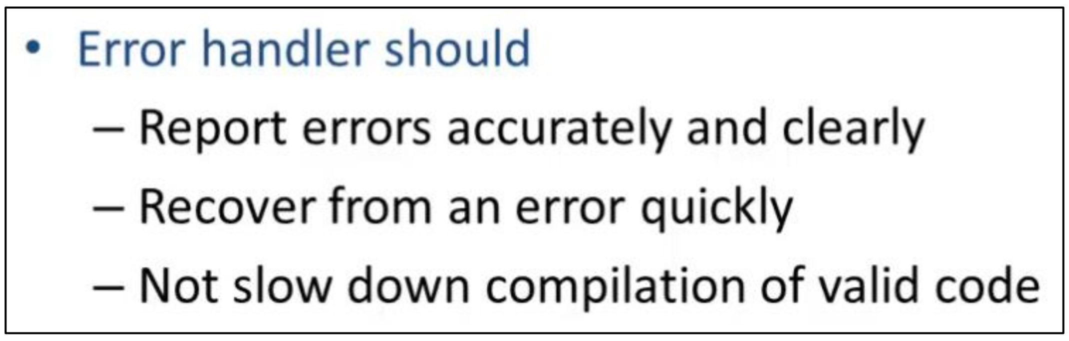
3.5 Error Handling



- Purpose of the compiler is
  - To detect non-valid programs
  - To translate the valid ones
- Many kinds of possible errors (e.g. in C)

Error kind	Example	Detected by ...
Lexical	... $ ...	Lexer
Syntax	... x *% ...	Parser
Semantic	... int x; y = x(3); ...	Type checker
Correctness	your favorite program	Tester/User

[Figure 3-22] Errors in Source Program



- Error handler should
  - Report errors accurately and clearly
  - Recover from an error quickly
  - Not slow down compilation of valid code

[Figure 3-23] Errors Handling Rule

Error handling methodology

1) Panic Mode

   Very simple, most popular, and widely used today.

   When an error detected, the decision is to discard tokens until one with a clear role find and continue from there. The target token called **Synchronizing Token**, which is typically the statement OR expression terminators. Which means that if an error detected just skip that unit, unit maybe a token, a sentence OR even a function, and going to next correct place.

   For example, below is an error expression, when see the second + sign error happens; in Panic Mode try to discard the second + sign skip to next integer and continue the expression handling.

```
(1 + + 2) + 3
     ^
```

2) Error Productions

   Simple, most popular, and widely used today.

   Specify known common mistakes, errors in the grammar. And result show the error to user if error on common error list.

3) Error Correction

   History code coded by Assembly, even punch cards very hard to find errors by programmer. So, compiler help us to find error with spending lots of time, maybe days, even weeks which today unacceptable, NOT used.

   Basic idea is if occur detected try to fix the errors as soon as possible.

   - Try token insertions & deletions
   - Exhaustive search

   But disadvantage is as below.

   - Hard to implement.
   - Slow down parsing of correct programs

Panic mode and Error productions is used today, the last methodology used history and nowadays NOT recommended.

3.6 Abstract Syntax Tree

Core data structure

Flex OR Lex is Lexical analyzer generator. Flex usually used GNU/Linux while Lex on UNIX.

Bison OR **YACC** is a general-purpose parser generator, Bison by one of GNU Projects generally used in GNU/Linux while YACC uses in UNIX.

Lex & YACC on UNIX

Flex & Bison on Linux

4 Semantic Analysis

5 Optimization

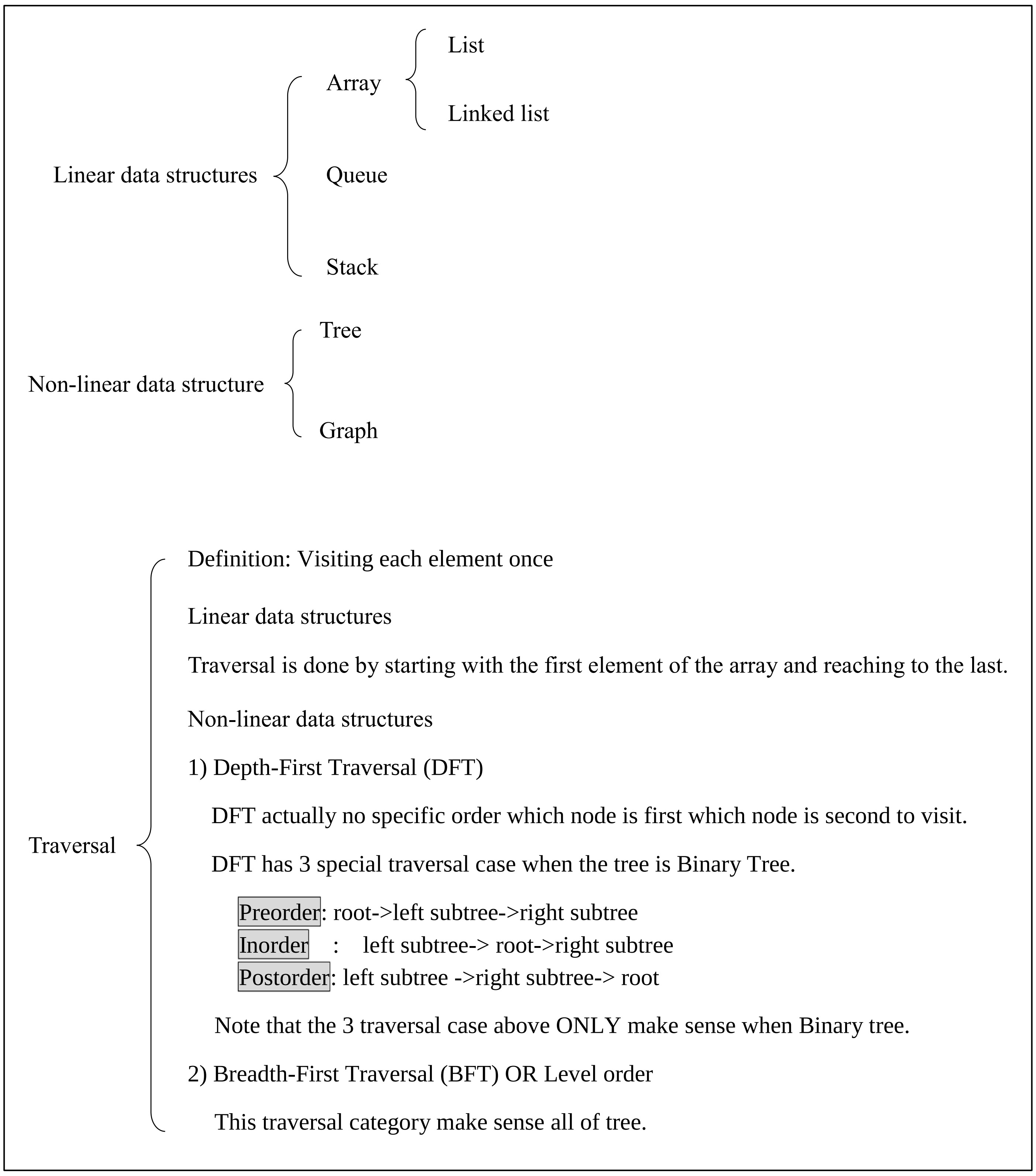
6 Code Generation

7 References

[1] [Web]; Compiler Principle – Stanford University; https://www.bilibili.com/video/BV1NE411376V

[2] [Book]; Dragon Book: Compilers Principles, Techniques, & Tools (Second Edition)

8 Data Structure



Linear data structures
- Array
  - List
  - Linked list
- Queue
- Stack

Non-linear data structure
- Tree
- Graph

Traversal

Definition: Visiting each element once

Linear data structures

Traversal is done by starting with the first element of the array and reaching to the last.

Non-linear data structures

1) Depth-First Traversal (DFT)

DFT actually no specific order which node is first which node is second to visit.

DFT has 3 special traversal case when the tree is Binary Tree.

Preorder: root->left subtree->right subtree
Inorder : left subtree-> root->right subtree
Postorder: left subtree ->right subtree-> root

Note that the 3 traversal case above ONLY make sense when Binary tree.

2) Breadth-First Traversal (BFT) OR Level order

This traversal category make sense all of tree.