Compiler Design

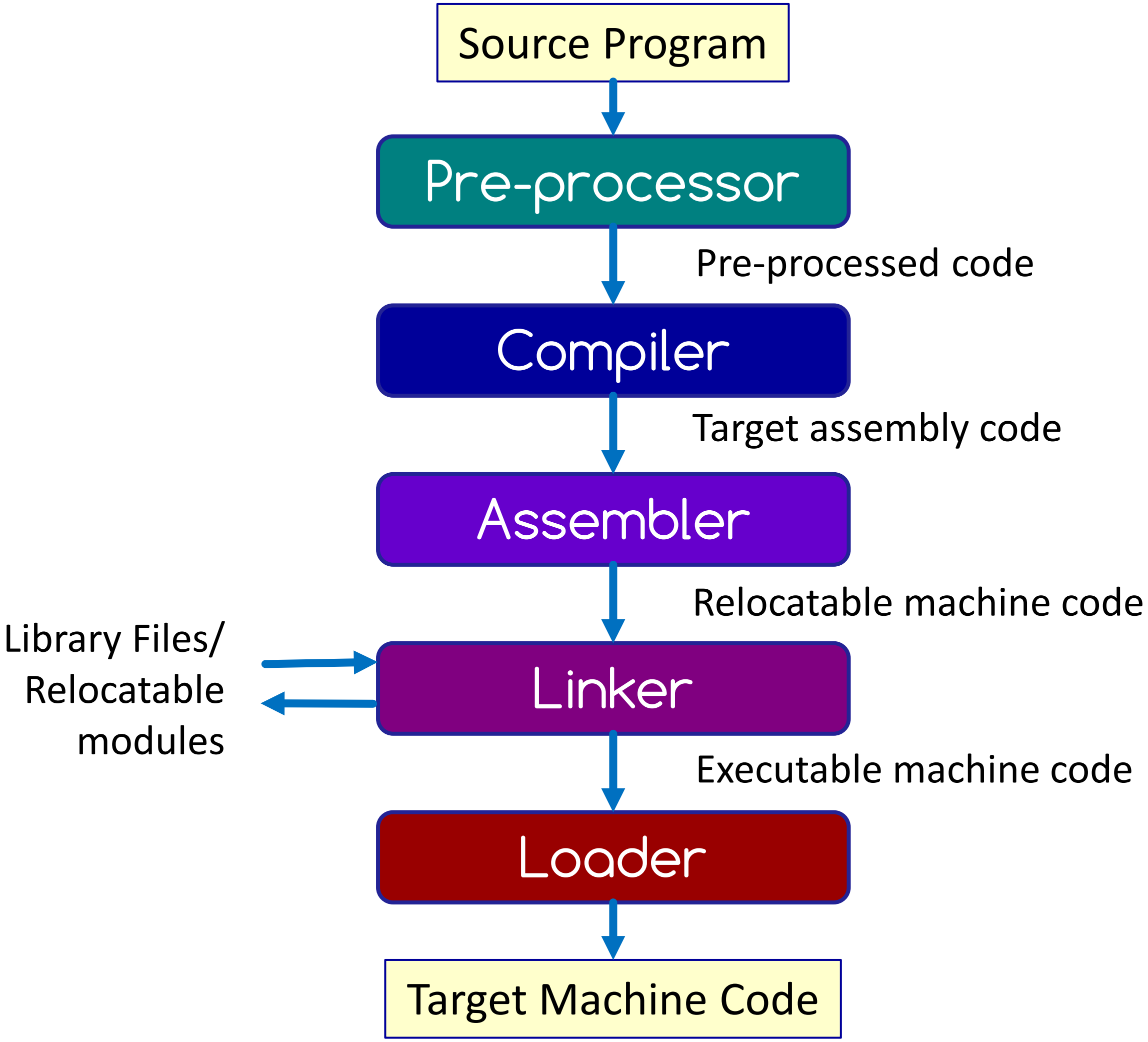
Lecture 2: Lexical Analysis

Sahar Selim

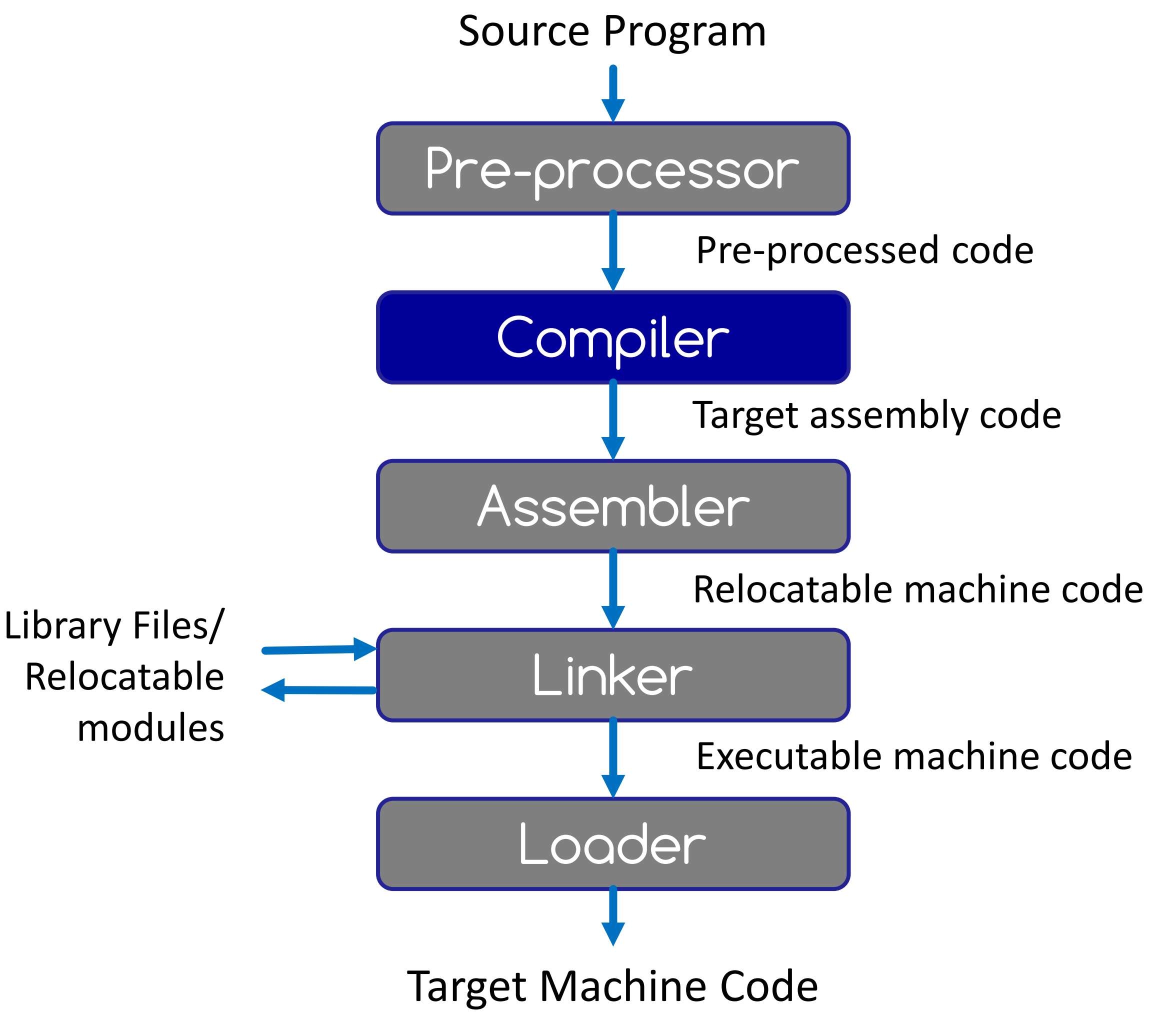
Agenda

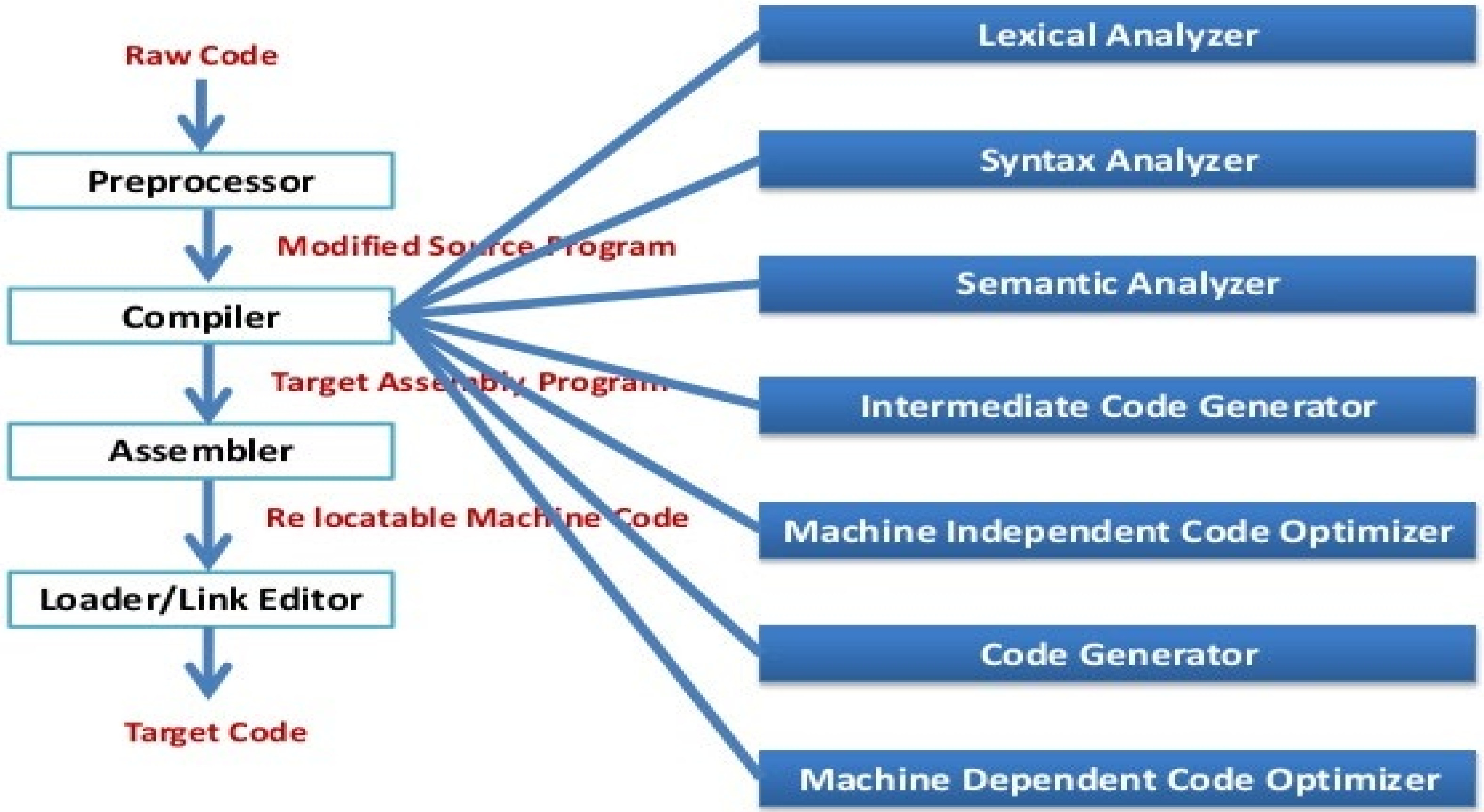
- Phases of a Compiler
- Lexical Analysis Phase
  - Scanning Process
  - Regular Expressions
  - Finite Automata

Language Processing System



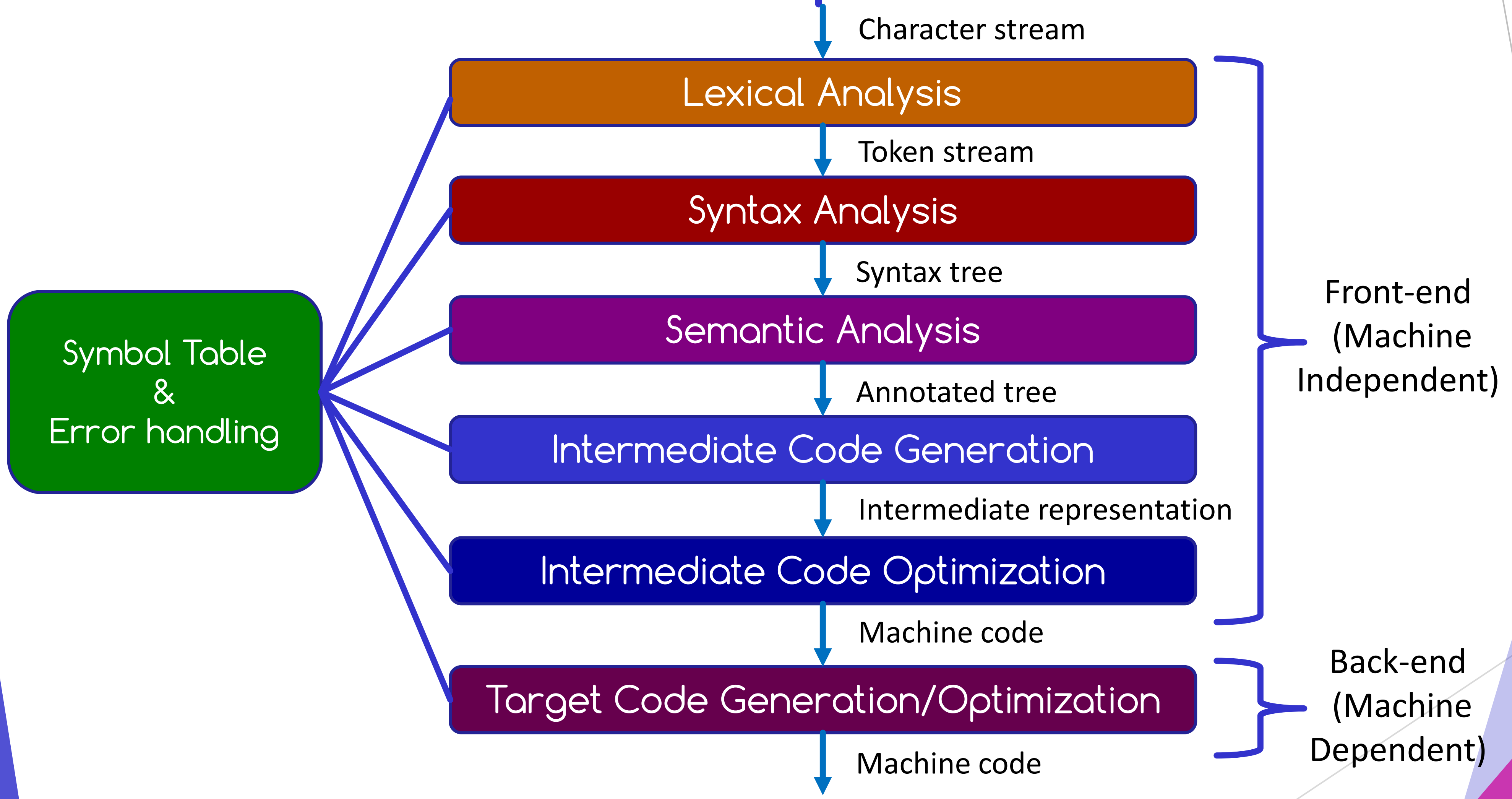
Language Processing System





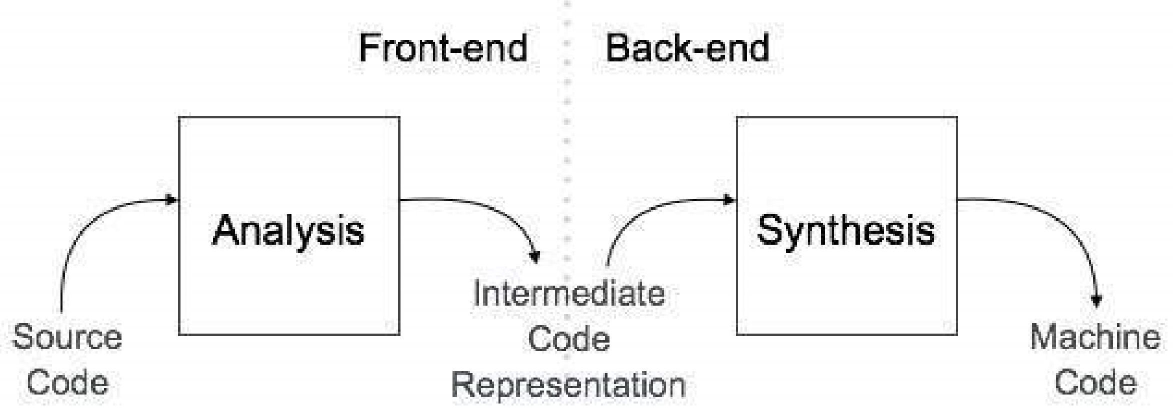
Phases of a Compiler

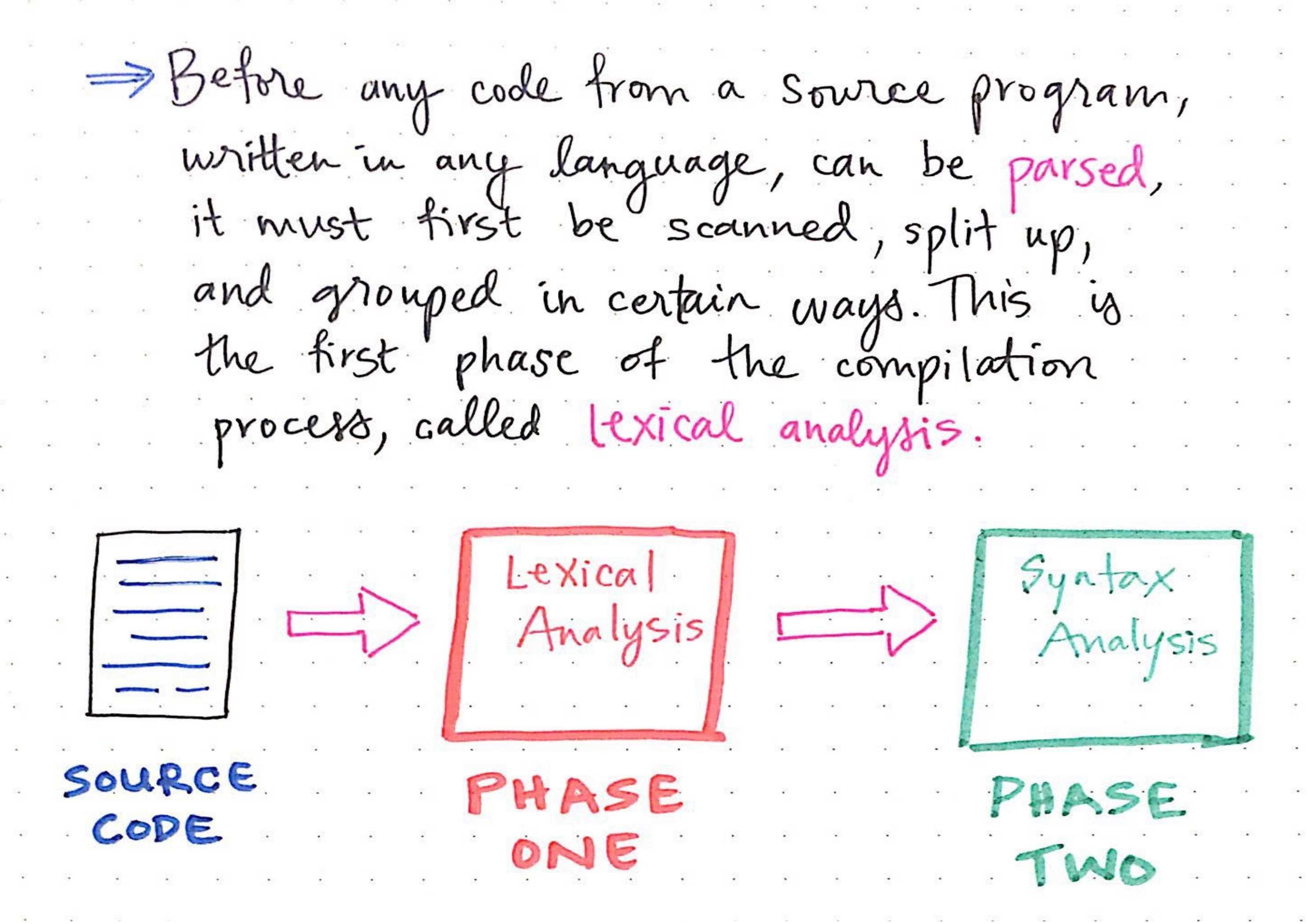
Phases of a Complier



Analysis and Synthesis

- The analysis part of the compiler analyzes the source program to compute its properties
  - Lexical analysis, syntax analysis and semantics analysis, as well as optimization
  - More mathematical and better understood
- The synthesis part of the compiler produces the translated codes
  - Code generation, as well as optimization
  - More specialized
- The two parts can be changed independently of the other

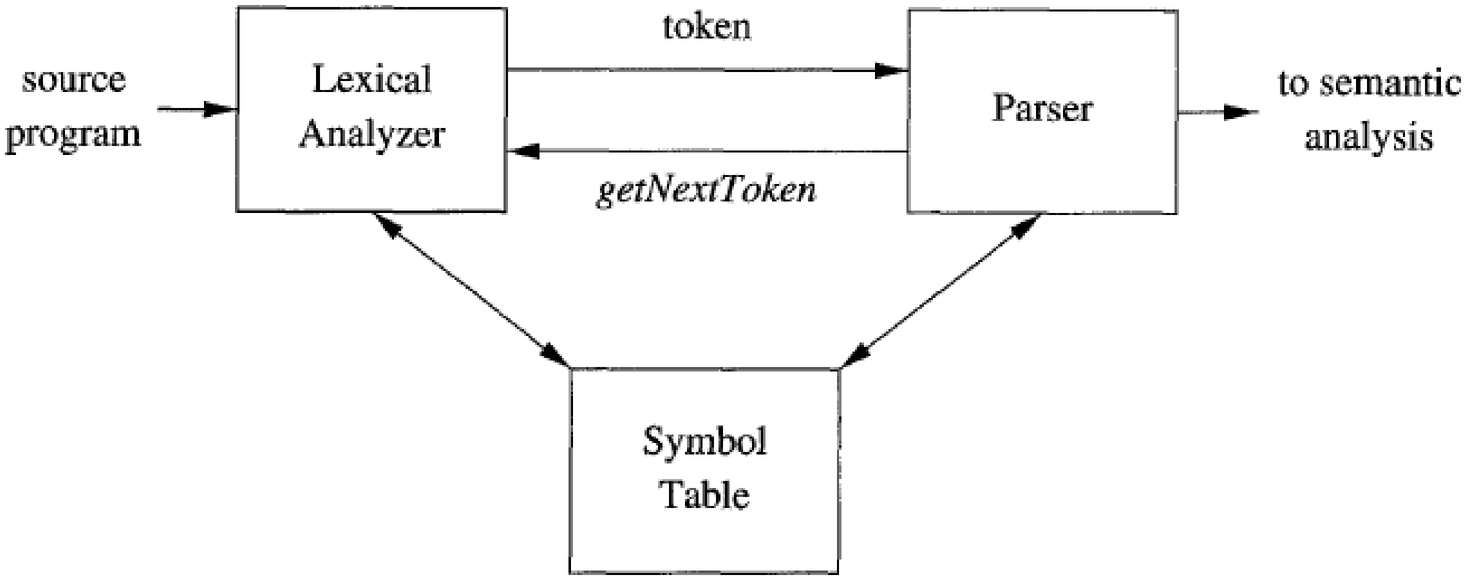


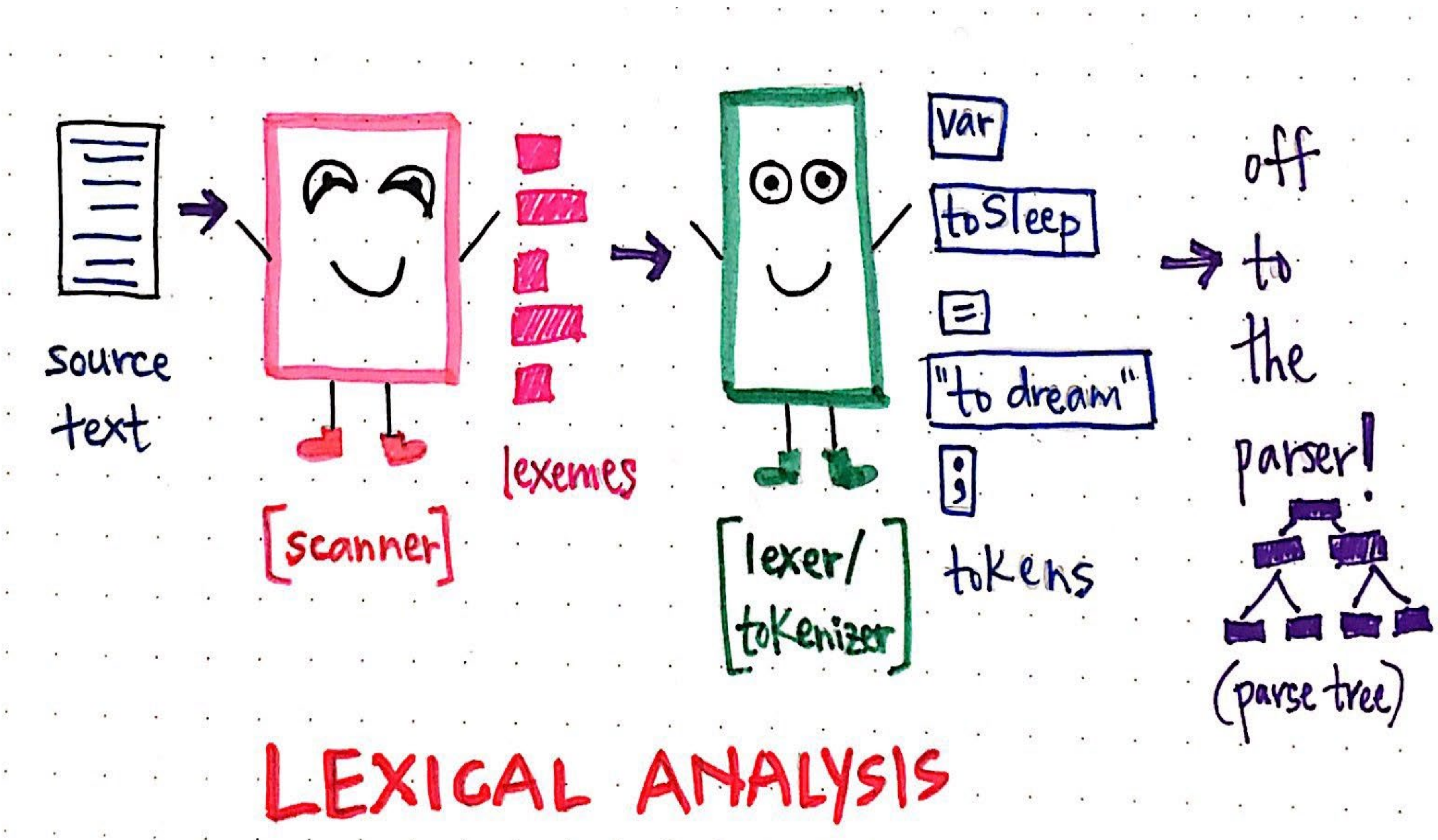


Lexical Analysis

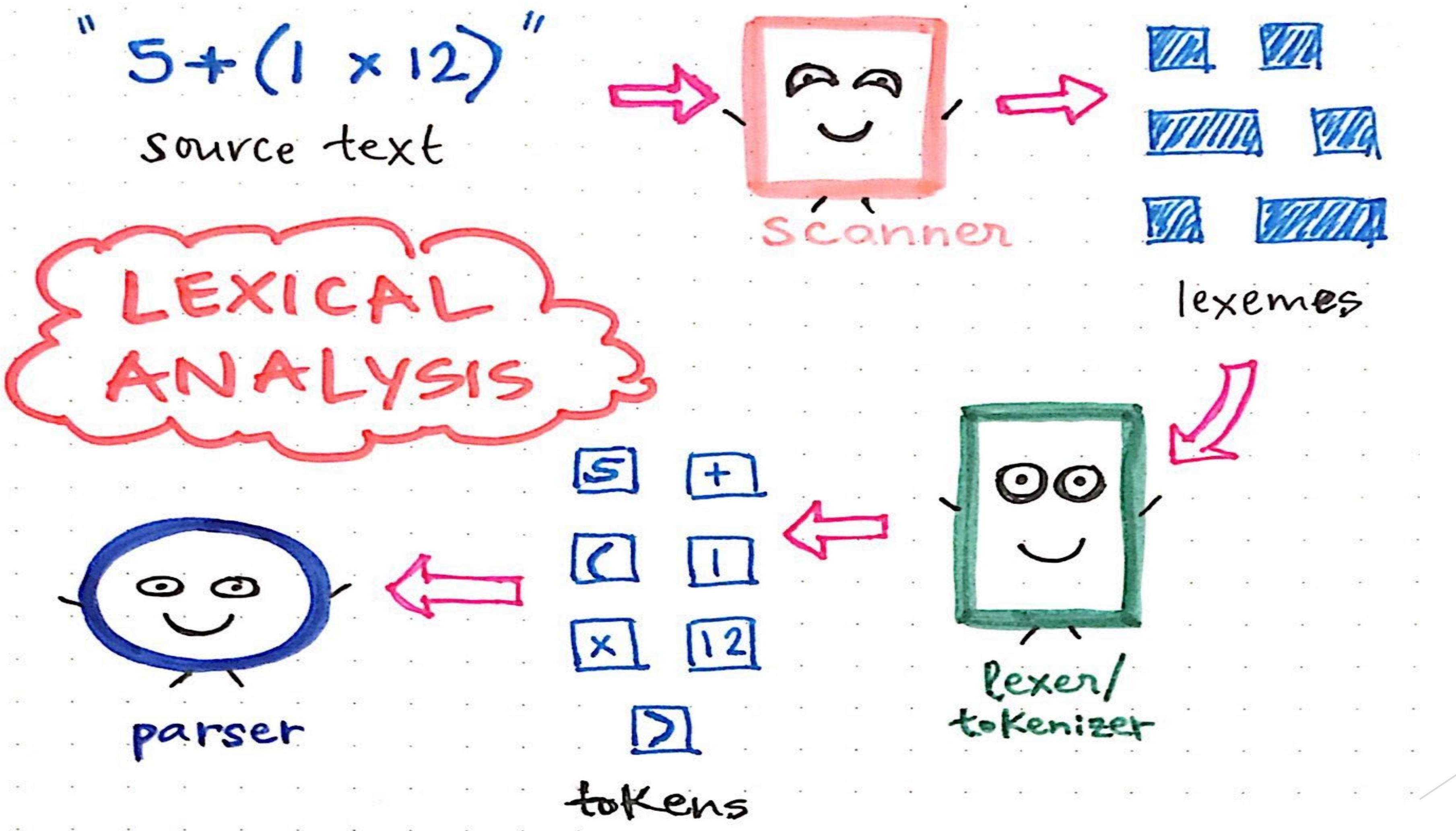
Lexical Analysis

- Provide the interface between the source program and the parser





source text
[scanner]
lexemes
[lexer/ tokenizer]
var
toSleep
=
"to dream"
;
tokens
off to the parser!
(parse tree)
LEXICAL ANALYSIS


"5+(1 x 12)"
source text
scanner
lexemes
LEXICAL ANALYSIS
lexer/
tokenizer
5
+
(
1
x
12
)
tokens
parser

Goals of the lexical analysis

- Divide the character stream into meaningful sequences called lexemes.
- Label each lexeme with a token (like constants, and reserved words) that is passed to the parser (syntax analysis)
- Remove non-significant blanks and comments
- Optional: update the symbol tables with all identifiers (and numbers)

The Lexical Analysis Terminology

- LEXEME
  - is a sequence of characters in the source program that matches the pattern for a token and is identified by the lexical analyzer as an instance of that token.
  - **Identifiers:** x, count, name, etc…
- TOKEN
  - A *token* is a ***category*** of lexemes
  - It consists of a token name and an *optional* attribute value
  - **Examples:** <Identifier>, <number>, <keyword> etc.
- PATTERN
  - The rules which characterize the set of strings for a token
  - Defined using regular expression

The Lexical Analysis Terminology

Token	Sample Lexemes	Informal description of pattern
if	if	if
While	While	while
Relation	<, <=, = , <>, > >=	< or <= or = or <> or > or >=
Id	count, sun, i, j, pi, D2	Letter followed by letters and digits
Num	0, 12, 3.1416, 6.02E23	Any numeric constant

Classifies pattern

Actual values
1. Stored in the symbol table
2. Returned to the parser

The Categories of Tokens

- RESERVED WORDS
  - Such as IF and THEN, which represent the strings of characters "if" and "then"
- SPECIAL SYMBOLS
  - Such as PLUS and MINUS, which represent the characters "+" and "-"
- OTHER TOKENS
  - Such as NUM and ID, which represent numbers and identifiers

Example

- Lexical analysis collects sequences of characters into meaningful units called ***tokens***
- An example: a[index]=4+2

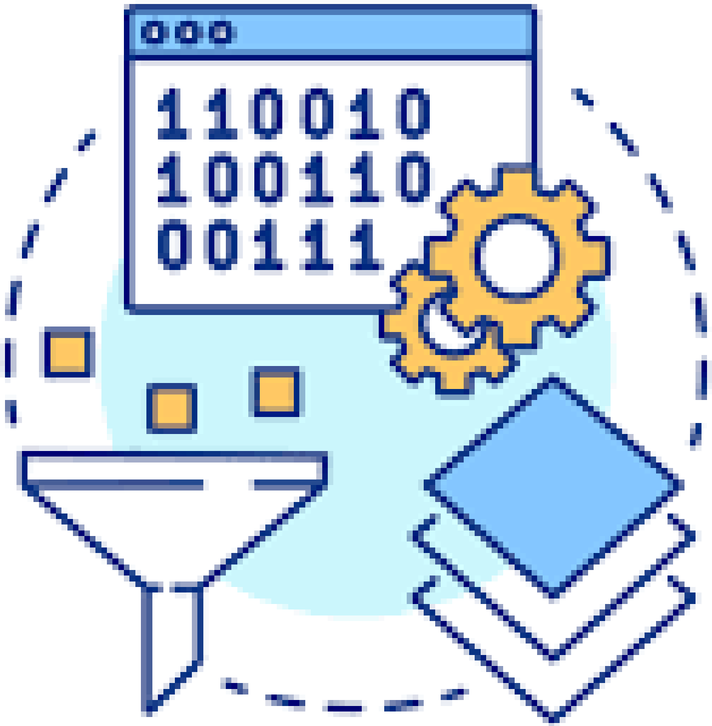
a	identifier
[	left bracket
index	identifier
]	right bracket
=	assignment
4	number
+	plus sign
2	number

Relationship between Tokens and its String

- The string is called **STRING VALUE** or LEXEME of token
- Some tokens have only one lexeme, such as reserved words
  - Example: (Token) **IF** → (Lexeme) **if**
- Some tokens may have infinitely many lexemes, such as the token ID
  - Example: (Token) **ID** → (Lexeme) **num1**

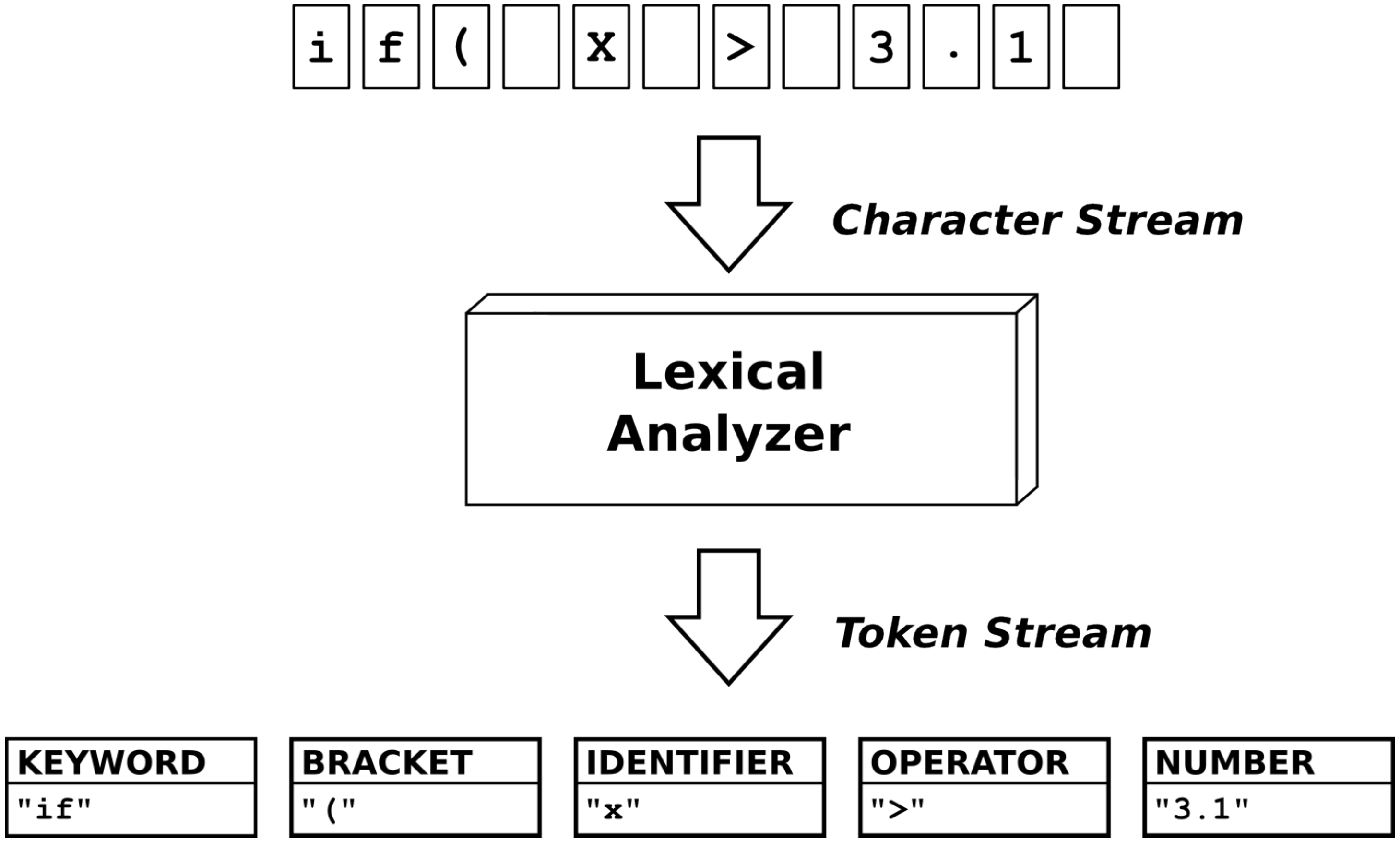
Relationship between Tokens and its String

- Any value associated to a token is called attribute of a token
  - String value is an example of an attribute.
  - A NUM token may have a *string value* such as “32767” and *actual value* 32767
  - A PLUS token has the string value “+” as well as arithmetic operation +
- The token can be viewed as the collection of all of its attributes
  - Only need to compute as many attributes as necessary to allow further processing
  - The numeric value of a NUM token need not compute immediately



Lexical Analysis Phase: Scanning & Tokenizing

Scanning and Tokenizing



Some Practical Issues of the Scanner

- One structured data type to collect all the attributes of a token, called a token record

```
struct TokenRecord
{
    TokenType tokenval; //the enum keywords
    char *stringval;
    int numval;
};
```

Example

$$x = y + 10$$

Lexeme	Token
x	identifier
=	Assignment operator
y	identifier
+	Addition operator
10	Number

Example

```
while (ip < z)
    ++ip;
```

(Keith Schwarz)

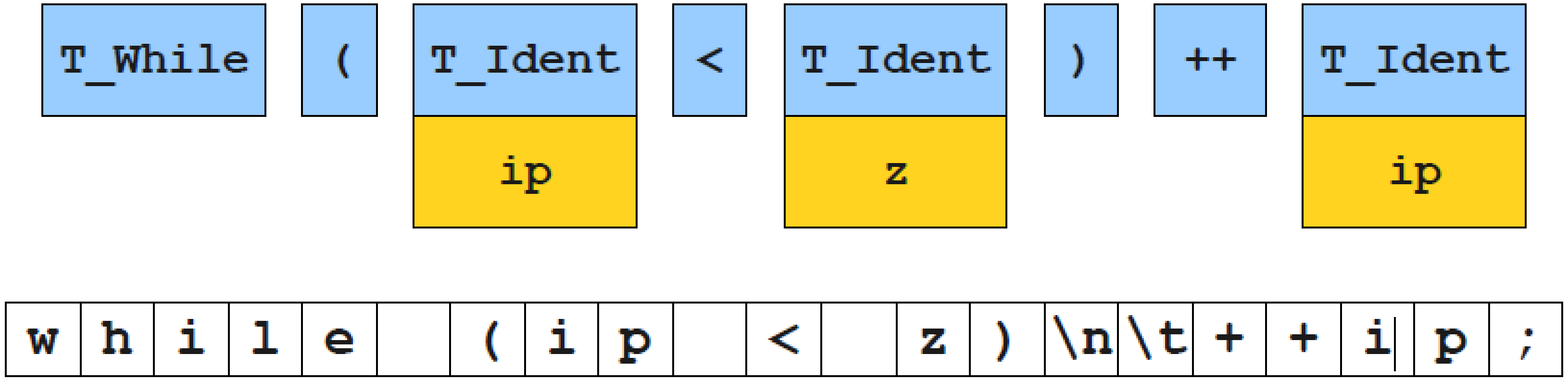
Example

w	h	i	l	e		(	i	p		<		z	)	\n	\t	+	+	i	p	;

```
while (ip < z)
    ++ip;
```

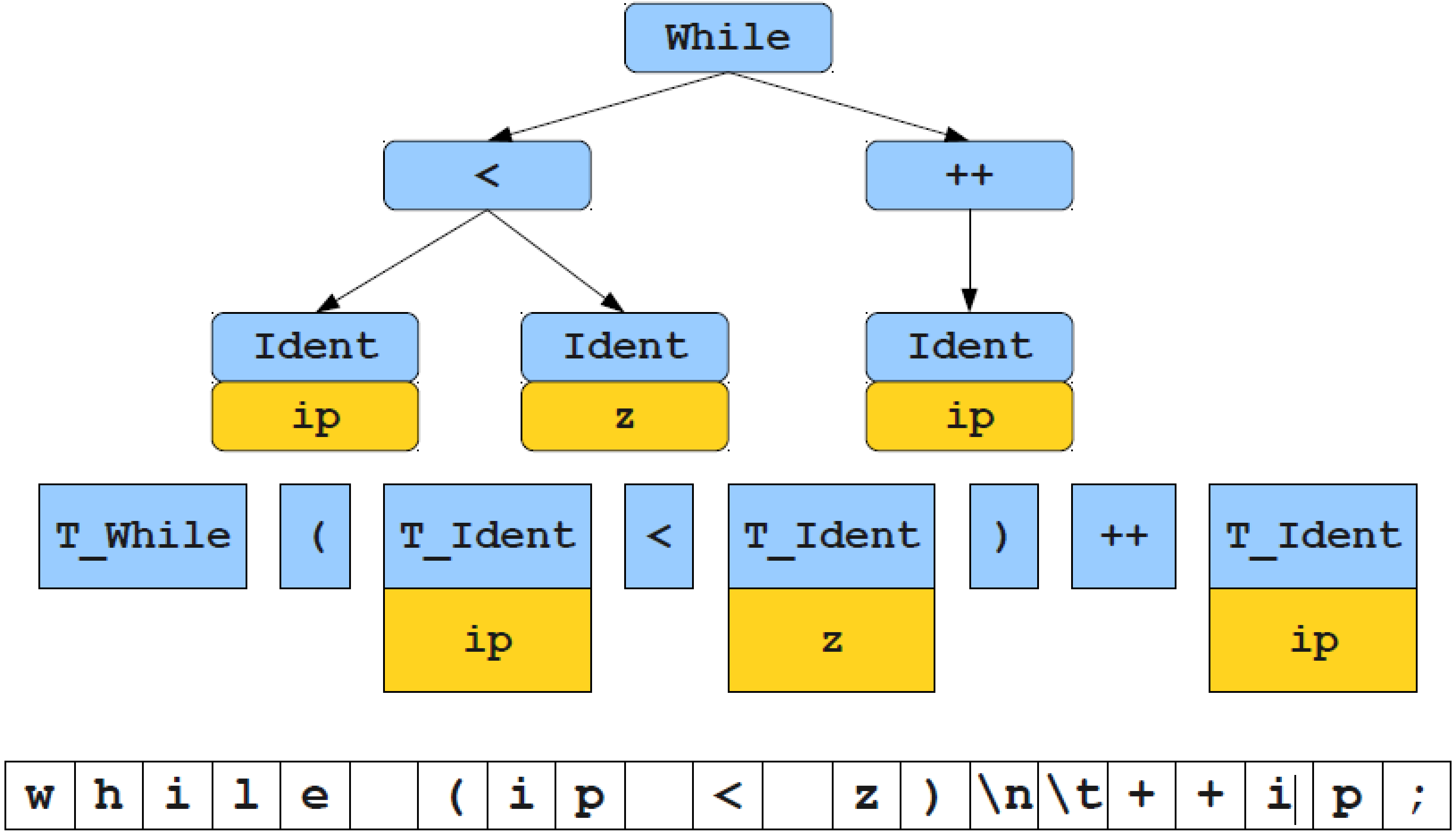
(Keith Schwarz)

Example




```
while (ip < z)
    ++ip;
```

(Keith Schwarz)




```
while (ip < z)
    ++ip;
```

(Keith Schwarz)

Scanning a Source File

w	h	i	l	e		(	1	3	7		<		i	)	\n	\t	+	+	i	;

w	h	i	l	e		(	1	3	7		<		i	)	\n	\t	+	+	i	;

w	h	i	l	e		(	1	3	7		<		i	)	\n	\t	+	+	i	;

w	h	i	l	e		(	1	3	7		<		i	)	\n	\t	+	+	i	;

w	h	i	l	e		(	1	3	7		<		i	)	\n	\t	+	+	i	;

w	h	i	l	e		(	1	3	7		<		i	)	\n	\t	+	+	i	;

Scanning a Source File

w	**h**	**i**	**l**	**e**		(	1	3	7		<		i	)	\n	\t	+	+	i	;

The piece of the original program from which we made the token is called a **lexeme**.

Scanning a Source File

w	**h**	**i**	**l**	**e**		(	1	3	7		<		i	)	\n	\t	+	+	i	;

The piece of the original program from which we made the token is called a **lexeme**.

`T_While`

This is called a **token**. You can think of it as an enumerated type representing what logical entity we read out of the source code.

Scanning a Source File

w	h	i	l	e		(	1	3	7		<		i	)	\n	\t	+	+	i	;

`T_While`

Scanning a Source File

w	h	i	l	e		(	1	3	7		<		i	)	\n	\t	+	+	i	;

T_While

Scanning a Source File

w	h	i	l	e		(	1	3	7		<		i	)	\n	\t	+	+	i	;

T_While

Sometimes we will discard a lexeme rather than storing it for later use. Here, we ignore whitespace, since it has no bearing on the meaning of the program.

Scanning a Source File

w	h	i	l	e		(	1	3	7		<		i	)	\n	\t	+	+	i	;

T_While

Scanning a Source File

w	h	i	l	e		(	1	3	7		<		i	)	\n	\t	+	+	i	;

T_While

Scanning a Source File

w	h	i	l	e		(	1	3	7		<		i	)	\n	\t	+	+	i	;

T_While

Scanning a Source File

w	h	i	l	e		(	1	3	7		<		i	)	\n	\t	+	+	i	;

T_While	(

Scanning a Source File

w	h	i	l	e		(	1	3	7		<		i	)	\n	\t	+	+	i	;

`T_While` `(`

Scanning a Source File

w	h	i	l	e		(	1	3	7		<		i	)	\n	\t	+	+	i	;

T_While	(

Scanning a Source File

w	h	i	l	e		(	1	3	7		<		i	)	\n	\t	+	+	i	;

T_While

(

Scanning a Source File

w	h	i	l	e		(	1	3	7		<		i	)	\n	\t	+	+	i	;

T_While (

Scanning a Source File

w	h	i	l	e		(	1	3	7		<		i	)	\n	\t	+	+	i	;

T_While	(

Scanning a Source File

w	h	i	l	e		(	1	3	7		<		i	)	\n	\t	+	+	i	;

T_While

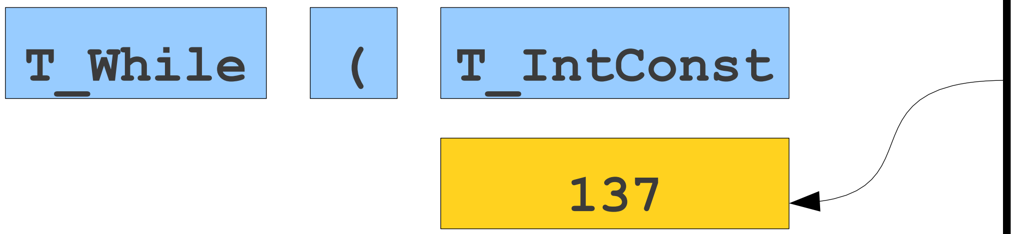
(

T_IntConst

137

Scanning a Source File

w	h	i	l	e		(	1	3	7		<		i	)	\n	\t	+	+	i	;



Some tokens can have **attributes** that store extra information about the token. Here we store which integer is represented.

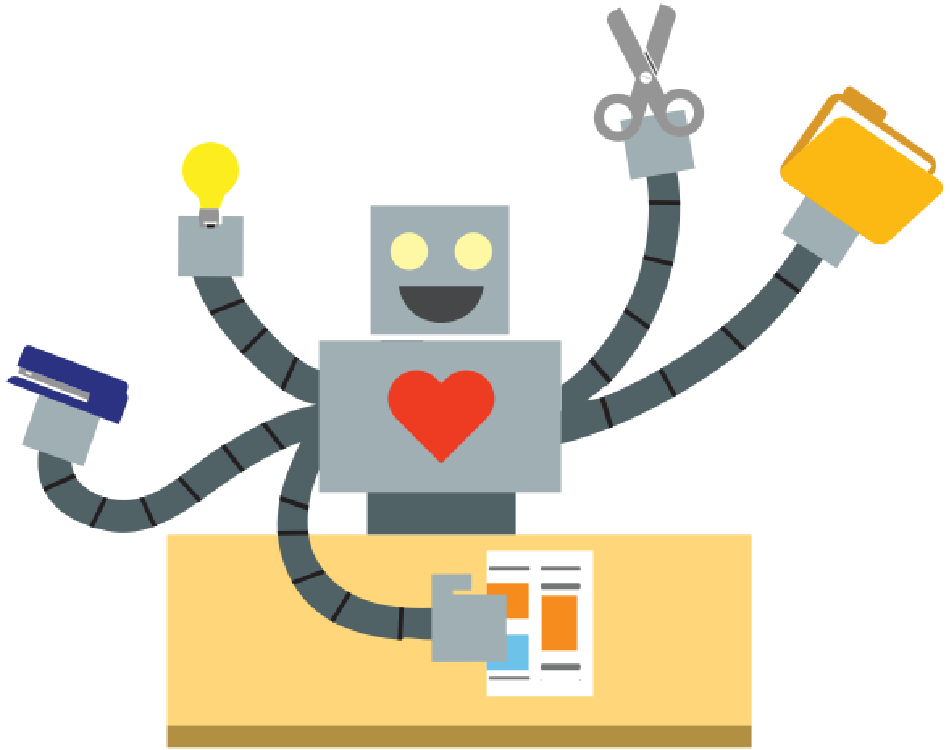

Q
A

5 MINUTES BREAK

How do we describe which set of lexemes is associated with each token type

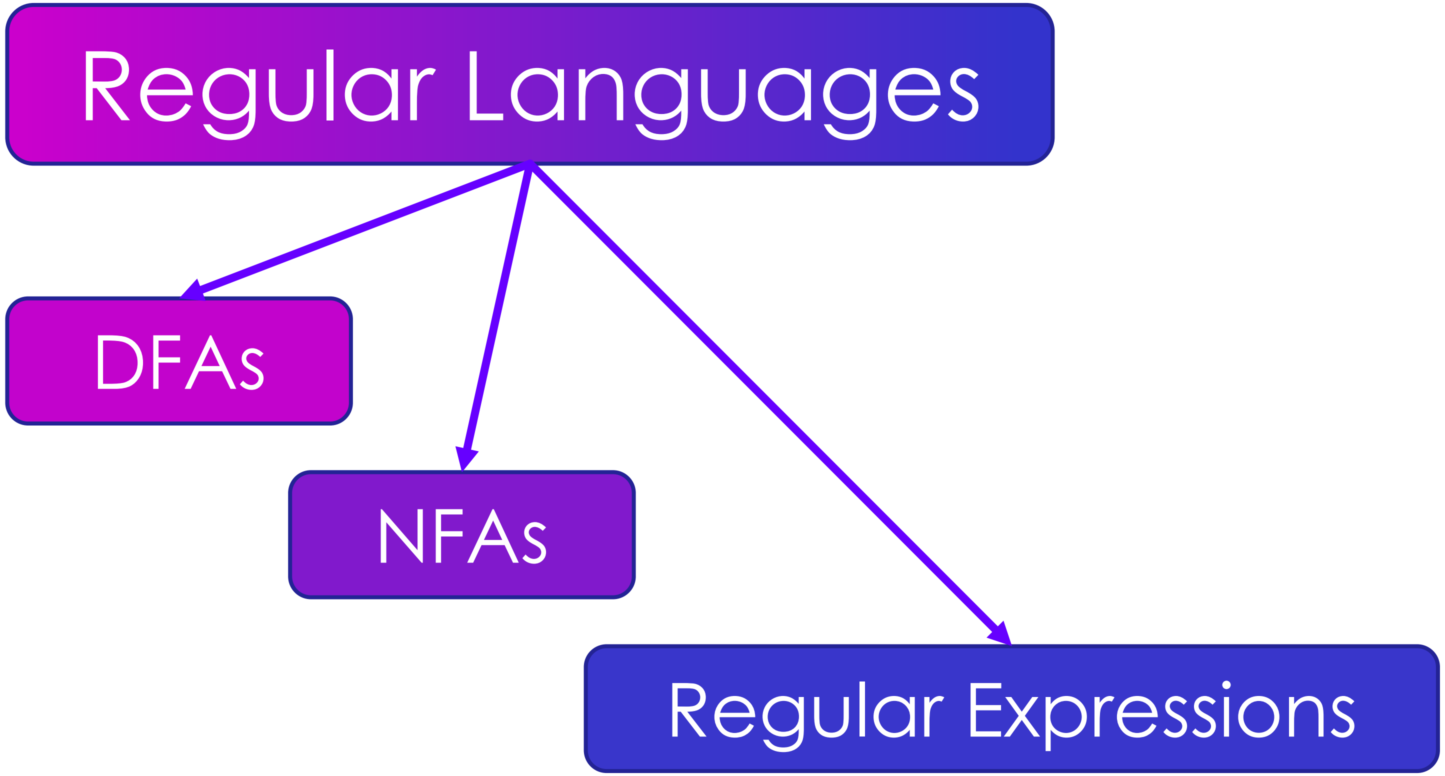
Formal Languages

- A **formal language** is a set of strings.
- Many infinite languages have **finite descriptions**:
  a) Define the language using a **regular expression**
  b) Define the language using an **automaton**
  c) Define the language using a **grammar**
- We can use these compact descriptions of the language to define sets of strings.



Regular Expressions

Standard Representations of Regular Languages



Regular expressions in practice

- **Compilers:** first phase of compiling transforms Strings to Tokens keywords, operators, identifiers, literals
- One regular expression for each token type

Regular Expression

- Regular Expressions are used for representing certain sets of strings in an algebraic fashion.
- Regular expressions describe regular languages
  - L(r): language of regular expression r

- Example: $(a+b\cdot c)^*$ describes the language

$$\{a,bc\}^* = \{\lambda, a, bc, aa, abc, bca, \ldots\}$$

Regular Operations

- Let A and B be languages. We define the regular operations union, concatenation, and star as follows:
- **Union:** $A \cup B = \{x \mid x \in A \text{ or } x \in B\}$.
- **Concatenation:** $A \circ B = \{xy \mid x \in A \text{ and } y \in B\}$
- **Star:** $A^* = \{x_1 x_2 \ldots x_k \mid k \geq 0 \text{ and each } x_i \in A\}$

Regular Expressions

1. $R = a$, where $a \in \Sigma$
2. $R = \varepsilon$
3. $R = \emptyset$
4. $R = (R_1 \cup R_2)$
5. $R = (R_1 \circ R_2)$
6. (R_1^*)

Conventions:

- Σ is shorthand for (0 U 1) if Σ = { 0,1 }
- Parentheses may be omitted
- Precedence:
  1. Star
  2. Concatenation
  3. Union
- R^+ is shorthand for RR^* (one or more)
- R^k is shorthand for R concatenated with itself k times
- Circle indicated concatenation may be omitted

Regular Expressions

- Any terminal symbol i.e. symbols $\in \sum$ including ε and Ø are regular expressions.

a, b, c, …. ε , Ø

- The union of two regular expressions is also a regular expression.

$$R_1, R_2 \rightarrow (R_1 \cup R_2)$$

Regular Expressions

- The concatenation of two regular expressions is also a regular expression.

$$R_1, R_2 \quad \rightarrow \quad (R_1 \circ R_2)$$

- The iteration (or Closure) of a regular expression is also a regular expression.

$$R \rightarrow R^* \quad \rightarrow \quad a^* = \varepsilon\ ,\ a,\ aa,\ aaa,\ ..$$

- The regular expression over $\sum$ are precisely those obtained recursively by the application of the above rules once or several times.

From RegEx to Languages

The language described by a regular expression L(R):

- $L(a) = \{ a \}$ (for all a in Σ)
- $L(\varepsilon) = \{ \varepsilon \}$
- $L(\emptyset) = \emptyset$
- $L(R_1 \cup R_2) = \{ w \mid w \text{ in } L(R_1) \text{ or } w \text{ in } L(R_2) \}$
- $L(R_1 \circ R_2) = \{ w_1 w_2 \mid w_1 \text{ in } L(R_1) \text{ and } w_2 \text{ in } L(R_2) \}$
- $L(R^*) = L(R)^*$

Languages of Regular Expressions

- $L(r)$:   language of regular expression r

- Example

$$L((a+b\cdot c)^*)=\{\lambda,a,bc,aa,abc,bca,\ldots\}$$

Continue . . .

- For regular expressions r_1 and r_2

$$L(r_1 + r_2) = L(r_1) \cup L(r_2)$$

$$L(r_1 \cdot r_2) = L(r_1)\, L(r_2)$$

$$L(r_1 *) = (L(r_1))*$$

$$L((r_1)) = L(r_1)$$

Example

- Regular expression: $(a+b)\cdot a^*$

$$
\begin{aligned}
L((a+b)\cdot a^*) &= L((a+b))\,L(a^*) \\
&= L(a+b)\,L(a^*) \\
&= (L(a)\cup L(b))\,(L(a))^* \\
&= (\{a\}\cup\{b\})\,(\{a\})^* \\
&= \{a,b\}\,\{\lambda,a,aa,aaa,\ldots\} \\
&= \{a,aa,aaa,\ldots,b,ba,baa,\ldots\}
\end{aligned}
$$

Example

- Define the language of the given regular expression

$$r = (aa)^*(bb)^*b$$

$$L(r) = \{a^{2n}b^{2m}b : \quad n, m \geq 0\}$$

Example

- Define the language of the given regular expression

$$r = (0+1)^*00(0+1)^*$$

$L(r)$ = { all strings containing substring 00 }

Review Questions

What Tokens are Useful Here?

```
for (int k = 0; k < myArray[5]; ++k) {
    cout << k << endl;
}
```

What Tokens are Useful Here?

```
for (int k = 0; k < myArray[5]; ++k) {
    cout << k << endl;
}
```

`for`	`{`
`int`	`}`
`<<`	`;`
`=`	`<`
`(`	`[`
`)`	`]`
`++`	

What Tokens are Useful Here?

```
for (int k = 0; k < myArray[5]; ++k) {
    cout << k << endl;
}
```

for	{
int	}
<<	;
=	<
(	[
)	]
++	

Identifier
IntegerConstant

Q2

- Suppose the only characters are 0 and 1.
- Define a regular expression for strings containing 00 as a substring.

Q2 Solution

- Suppose the only characters are 0 and 1.
- Define a regular expression for strings containing 00 as a substring.

$$(0|1)^*\,00(0|1)^*$$

11011100101
0000
111110111100111111

Q3

- Suppose the only characters are 0 and 1.
- Write a regular expression for strings of length exactly four.

Q3 Solution

- Suppose the only characters are 0 and 1.
- Write a regular expression for strings of length exactly four.

$$(0|1)(0|1)(0|1)(0|1)$$

Q3 Solution

- Suppose the only characters are 0 and 1.
- Write a regular expression for strings of length exactly four.

(0|1)(0|1)(0|1)(0|1)

0000
1010
1111
1000

Q3 Solution

- Suppose the only characters are 0 and 1.
- Write a regular expression for strings of length exactly four.

(0|1)(0|1)(0|1)(0|1)

0000
1010
1111
1000

Q3 (Another Solution)

- Suppose the only characters are 0 and 1.
- Write a regular expression for strings of length exactly four.

Another Solution

(0|1){4}

0000
1010
1111
1000


Q
A

Q4

- Suppose our alphabet is a, @, and .
- Write a regular expression for email address format.

Q4 Solution

- Suppose our alphabet is a, @, and .
- Write a regular expression for email address format.

$$aa^*(.aa^*)^*@aa^*.aa^*(.aa^*)^*$$

$$a^+(.a^+)^*@a^+.a^+(.a^+)^*$$

$$a^+(.a^+)^*@a^+(.a^+)^{+*}$$

compiler@nu.edu.eg

first.middle.last@mail.site.org

Q5

Suppose that our alphabet is all ASCII characters.

A regular expression for even numbers is

Q5 Solution

Suppose that our alphabet is all ASCII characters.

A regular expression for even numbers is

(+|-)?(0|1|2|3|4|5|6|7|8|9)*(0|2|4|6|8)

Q5 Solution

Suppose that our alphabet is all ASCII characters.

A regular expression for even numbers is

(+|-)?(0|1|2|3|4|5|6|7|8|9)*(0|2|4|6|8)

42
+1370
-3248
-9999912

Q5 Solution

Suppose that our alphabet is all ASCII characters.

A regular expression for even numbers is

(+|-)?(0|1|2|3|4|5|6|7|8|9)*(0|2|4|6|8)

42

+1370

-3248

-9999912

Q5 Solution

Suppose that our alphabet is all ASCII characters.

A regular expression for even numbers is

(+|-)?[0123456789]*[02468]

42
+1370
-3248
-9999912

Q5 Solution

Suppose that our alphabet is all ASCII characters.

A regular expression for even numbers is

(+|-)?[0-9]*[02468]

42

+1370

-3248

-9999912

See you next lecture