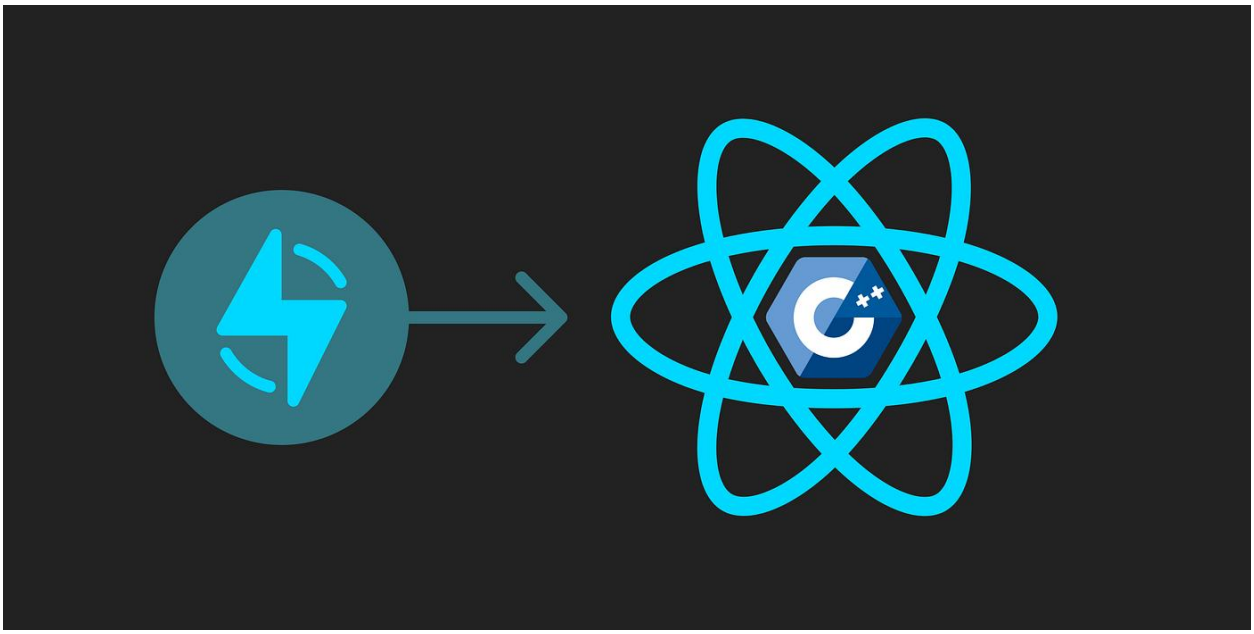
Compiler Construction

Language Name: React++

Language Theme: Modern Web Framework terminologies with C++ Structure



Project Phase # 01 Documentation	
Submitted By	Rameez Ali
Registration Number	L1F22BSCS0374
Section	G1
Submitted To	Sir Asif Farooq
Project Topic	Lexical Analyzer (Tokenization, Error handling and Line Number track of each token)

1. Regex Definitions:

Token Name	Regex	Description / Example
IDENTIFIER (variables)	@[_a-zA-Z][a-zA-Z0-9]*	Variables must start with @, e.g., @count
NUMBER (int/float/exp)	[0-9]+(\.[0-9]+)?([eE][+-]?[0-9]+)?	123, 3.14, 1e-5
INTEGER	[0-9]+	42
FLOAT	[0-9]*\.[0-9]+	.5, 3.14 (require at least one digit after . recommended)
STRING_LITERAL	\"([^\"\n]*)\"	"Hello World"
CHAR_LITERAL	\'[a-zA-Z0-9!@#$%^&*(_)`\,./;’]\'	'A', '$'
MULTI_LINE_COMMENT	\[:[^\]]*:\]	from [: to :] (may span lines)
SINGLE_LINE_COMMENT	:[^=<>!~].*$	from : to end-of-line
OUTPUT_OPERATOR	<<	render << "Hi"$
INPUT_OPERATOR	>>	fetch >> @name$
EQ_OPERATOR	:=	equality test operator
NEQ_OPERATOR	~=	not-equal operator
ARITHMETIC OPS	\+ - * / %	+, -, *, /, %
RELATIONAL OPS	< > <= >= ==	comparators
BLOCK_START	</	custom block start
BLOCK_END	\\>	custom block end
END_OF_STATEMENT	\$	explicit statement terminator $
PUNCTUATION	\(\) , ;	parentheses, comma, semicolon
WHITESPACE	[\t\r\n]+	skip in lexer
RE’s for invalid cases are defined in scanner.l		

2. Punctuations:

Symbol	Meaning
</	Block start
\>	Block end
$	Statement terminator
:	Single-line comment start
[: :]	Multi-line comment delimiters
( ) , ;	Parentheses, comma, semicolon

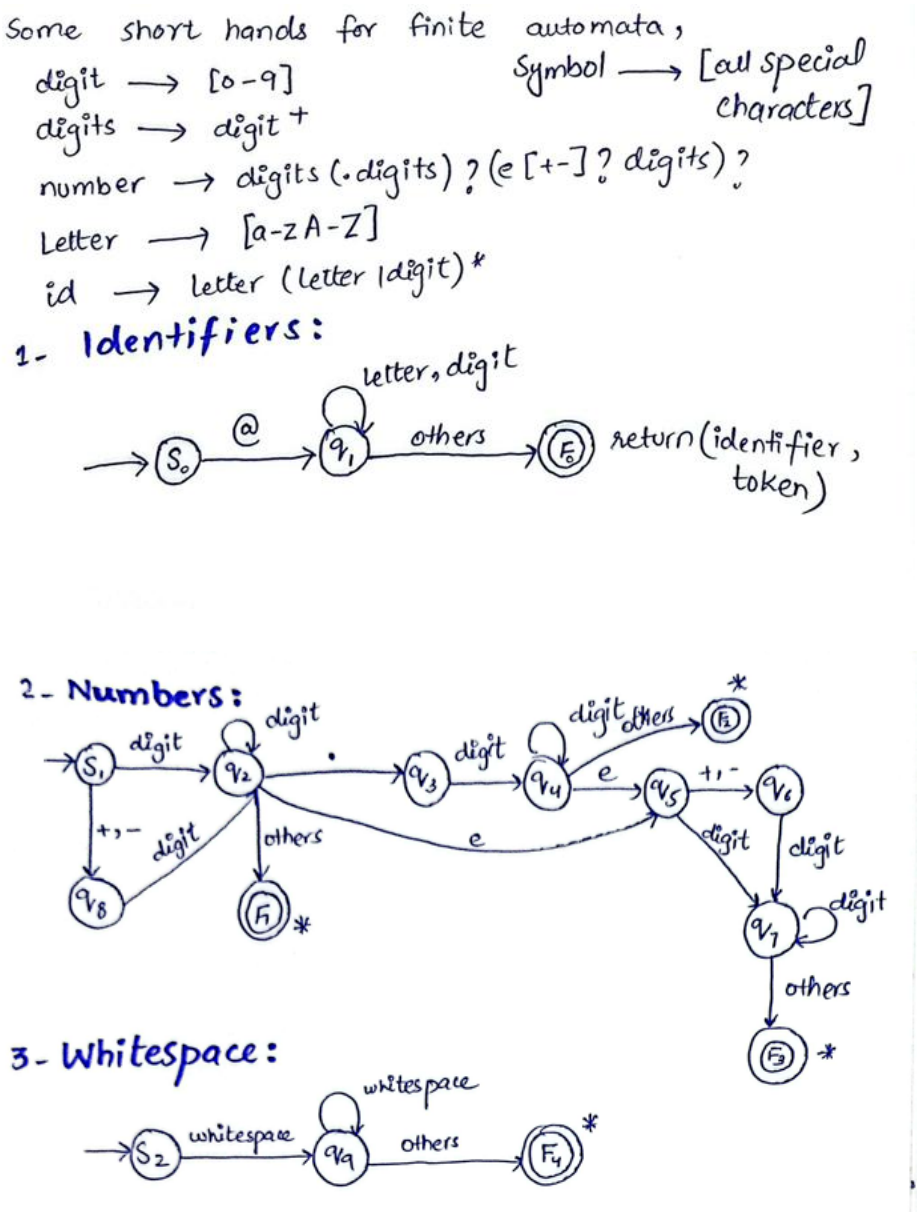
3. Keywords:

Keyword	Equivalent (C++)	Example usage
state	int	state @count = 0$
scale	float	scale @speed = 1.5$
tag	char	tag @grade = 'A'$
content	string	content @name = "Rameez"$
lock	const	lock @PI = 3.14$
detect	auto	detect @value = 20$
toggle	bool	toggle @isTrue = false$
render	cout	render << "Hi"$
fetch	cin	fetch >> @name$
promise	if	promise (@x > 0) ... deceit ...
deceit	else	deceit</….\>
navigate	for	navigate (@i=0; @i<5; @i=@i+1) </ ... \>
observe	while	observe (@x < 10) </ ... \>
dispatch	do	dispatch </ ... \> observe (@cond)$
route	switch	route (@opt) </ path(...) ... \>
path	case	branch inside route
rerender	continue	loop continue
unmount	break	loop break
redirect	goto	redirect @label$
send	return	send @value$
root	main	state root() </ ... \>

4. Operators:

Operator	Purpose
<<	Output operator (render)
>>	Input operator (fetch)
:=	Equality operator (custom choice)
~=	Not equal
+ - * / %	Arithmetic
< > <= >= ==	Relational
and, or, ~	Logical (word-form)

5. Finite Automata (Numbers & Identifiers):



Explanation of Keyword Design (React++)

1. Why These Keywords Were Chosen

- The keywords were selected to **bridge C++ fundamentals with modern web development terminology**.
- Each keyword closely **resembles a C++ construct** (for example, **state** refers to **int**, **scale** refers to **float**, **promise** refers to **if**) but uses **names inspired by React/Web frameworks**.
- This design helps students **relate procedural logic to real-world frameworks** where similar concepts (state, render, dispatch, etc.) are widely used.
- The naming convention maintains **clarity, memorability, and contextual meaning**, so even a beginner can intuitively understand the keyword's role.

2. Reason for Punctuation and Operator Design

- The **punctuation symbols** such as </, \>, and $ were chosen to **differentiate React++ from traditional C-like syntax** while maintaining readability.
    1. </ and \> replace { and } to resemble **HTML/React component structure**, making the code visually familiar to web developers.
    2. $ is used as an **explicit statement terminator**, ensuring clear line endings and simplifying lexical analysis.
    3. [: :] are used for comments to avoid conflicts with other operators and to give a **distinct, custom syntax style**.
- The **operators** are designed to balance **C++ familiarity and clarity**:
    1. Operators like :=, ~= were introduced to make **assignment and comparison visually distinct**, reducing ambiguity during tokenization.
    2. Logical words such as and, or, and ~ (for NOT) are **word-based** for readability and beginner friendliness.
    3. Arithmetic (+, -, *,**, /, %) and relational (<, >, <=, >=) operators follow C++ conventions for smooth learning transition.
    4. The I/O operators << and >> are retained from C++ to keep **semantic consistency** with output/input operations like render and fetch.

3. Why React++ Is Unique

- React++ integrates **programming syntax with web-inspired semantics**, giving a feeling of "code meets UI".
- It uses </ **and** \> as block delimiters, resembling HTML/JSX, helping students mentally map logic blocks to UI components.
- The **explicit $ statement terminator** provides readability and ensures simpler lexical boundaries.
- By adopting **promise, render, dispatch, route** etc., learners can transition more smoothly from traditional programming to **component-based web frameworks**.
- The language serves as a **conceptual bridge** understanding C++ flow while becoming familiar with React-like concepts.

Sample Program:

```
[: --- Multi-line comment example --- :]
state root()
</
    : Single-line comment starts here
    state @x = 5$
    scale @y = 2.5$
    scale @exp = 3.14e+2$
    tag @grade = 'A'$
    content @msg = "Hello"$

    @1invalid = 99$       : invalid identifier
```

```
    'AB'                  : invalid char
    "Unclosed string      : will trigger unterminated string error

    promise (@x >= 5 and @y < 100)
        render << @msg$
    deceit
        render << "Bye"$
    navigate (@x = 0; @x < 3; @x = @x + 1)
    </ render << "Loop"$ \>
    send 0$
\>
```

Sample Tokens:

Comments are ignored by analyzer just printed on console not in file.

```
ubuntu@ubuntu:~$ flex scanner.l
ubuntu@ubuntu:~$ gcc lex.yy.c -lfl -o scanner
ubuntu@ubuntu:~$ ./scanner < sample.txt
MultiLine: [: --- Multi-line comment example --- :]
Comment: : Single-line comment starts here
Comment: : invalid identifier
Comment: : invalid char
All tokens written to output.txt successfully!
```

Tokens generated by Lex:

Line No.	Token Type	Lexeme / Value
2	KEYWORD	state
2	KEYWORD	root
2	PUNCTUATION	(
2	PUNCTUATION	)
3	PUNCTUATION	</
5	KEYWORD	state
5	IDENTIFIER	@x
5	OPERATOR	=
5	INTEGER	5
5	END OF LINE	$
6	KEYWORD	scale
6	IDENTIFIER	@y

6	OPERATOR	=
6	FLOAT	2.5
6	END OF LINE	$
7	KEYWORD	scale
7	IDENTIFIER	@exp
7	OPERATOR	=
7	EXPONENTIAL NUMBER	3.14e+2
7	END OF LINE	$
8	KEYWORD	tag
8	IDENTIFIER	@grade
8	OPERATOR	=
8	CHAR	'A'
8	END OF LINE	$
9	KEYWORD	content
9	IDENTIFIER	@msg
9	OPERATOR	=
9	STRING	"Hello"
9	END OF LINE	$
11	INVALID IDENTIFIER	@1invalid
11	OPERATOR	=
11	INTEGER	99
11	END OF LINE	$
12	INVALID CHAR	'AB'
13	UNTERMINATED STRING	"Unclosed string
14	KEYWORD	promise
14	PUNCTUATION	(
14	IDENTIFIER	@x
14	OPERATOR	>=
14	INTEGER	5
14	INVALID IDENTIFIER	and
14	IDENTIFIER	@y
14	OPERATOR	<
14	INTEGER	100
14	PUNCTUATION	)
15	KEYWORD	render
15	OPERATOR	<<
15	IDENTIFIER	@msg
15	END OF LINE	$
16	KEYWORD	deceit
17	KEYWORD	render
17	OPERATOR	<<
17	STRING	"Bye"
17	END OF LINE	$
18	KEYWORD	navigate
18	PUNCTUATION	(

18	IDENTIFIER	@x
18	OPERATOR	=
18	INTEGER	0
18	ERROR	;
18	IDENTIFIER	@x
18	OPERATOR	<
18	INTEGER	3
18	ERROR	;
18	IDENTIFIER	@x
18	OPERATOR	=
18	IDENTIFIER	@x
18	OPERATOR	+
18	INTEGER	1
18	PUNCTUATION	)
19	PUNCTUATION	</
19	KEYWORD	render
19	OPERATOR	<<
19	STRING	"Loop"
19	END OF LINE	$
19	PUNCTUATION	>
20	KEYWORD	send
20	INTEGER	0
20	END OF LINE	$
21	PUNCTUATION	>

Note: The **Regular Expressions (REs)** used for **error detection** (such as invalid identifiers, invalid characters, and unterminated strings) are explicitly defined within the **Lexical Analyzer**. Any lexical pattern **not matched by the defined Regular Expressions (REs)** is automatically classified as an **error token**.
Such unmatched inputs are handled by the **catch-all rule** `(.)` in the Lexical Analyzer, ensuring that every undefined or invalid symbol is properly detected and reported.