SRM INSTITUTE OF SCIENCE AND TECHNOLOGY
Kattankulathur, Chengalpattu District - 603203

18CSC304J/ COMPLIER DESIGN

MINI PROJECT REPORT

Compiler Design Parsers

Guided by:
Dr. Nithyashri J

Submitted By:

Tanmay Shukla (RA2011003010119)
Khushi Suri (RA2011003010129)

COLLEGE OF ENGINEERING & TECHNOLOGY
SRM INSTITUTE OF SCIENCE & TECHNOLOGY
S.R.M. NAGAR, KATTANKULATHUR – 603 203

BONAFIDE CERTIFICATE

Certified that this project report **“Compiler Design Parsers”** is the Bonafide work of **“Tanmay Shukla (RA2011003010119) and Khushi Suri (RA2011003010129)”** of III Year/VI Sem B. Tech (CSE) who carried out the mini project work under my supervision for the course **18CSC304J- Compiler Design** in SRM Institute of Science and Technology during the academic year 2022-2023(Even sem).

(Signature) **(Department Seal)**

Dr. Nithyashi J
Assistant Professor
Department of Computing Technologies

ABSTRACT

Compilers are essential software tools that translate source code into executable code. Parser, a crucial component of a compiler, analyzes the structure of the source code and creates a parse tree that is used by other compiler components to generate the final executable code. There are different types of parsers, such as top-down parsers, bottom-up parsers, and recursive descent parsers. Each parser has its strengths and weaknesses, and the selection of the parser depends on the programming language being compiled. Compiler designers need to consider the parsing speed, memory usage, and ease of implementation while selecting a parser. This abstract presents a concise overview of compiler design parsers, their role in the compilation process, and the factors that influence their selection.

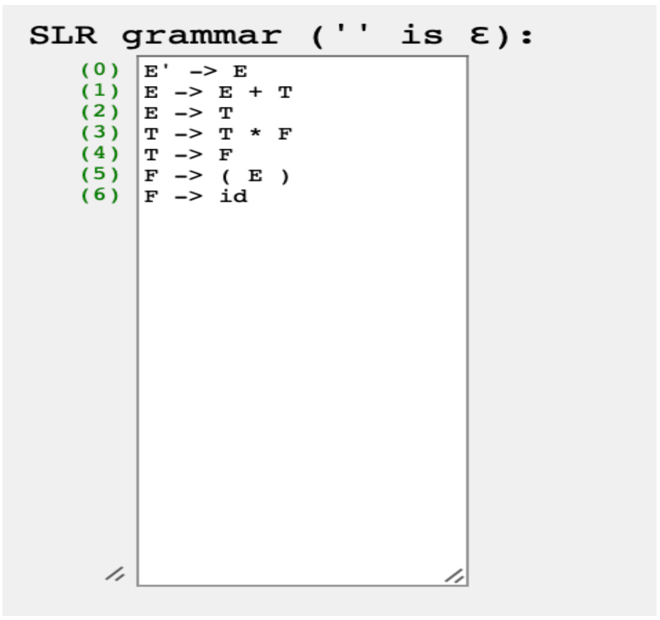
TABLE OF CONTENTS

INTRODUCTION

A compiler is a crucial software tool that translates source code written in a high-level programming language into executable code that can be executed on a target machine. The process of compilation involves several stages, including lexical analysis, parsing, semantic analysis, code optimization, and code generation. Among these stages, parsing is a critical component responsible for analyzing the structure of the source code and generating a parse tree that is used by other compiler components to generate the final executable code.
Parsers are programs that take a sequence of tokens as input and construct a parse tree, which represents the syntactic structure of the input program. There are different types of parsers, such as top-down parsers, bottom-up parsers, and recursive descent parsers, each with its strengths and weaknesses.
In this discussion, we will delve into the details of compiler design parsers, their role in the compilation process, and the factors that influence their selection. We will also explore the different types of parsers and their advantages and disadvantages, as well as the challenges and trade-offs involved in selecting an appropriate parser for a given programming language.

The screenshots of the sample input and target code output are as follows:
Sample Input:

<u>SLR PARSER</u>



SLR grammar ('' is ε):

```
(0) E' -> E
(1) E -> E + T
(2) E -> T
(3) T -> T * F
(4) T -> F
(5) F -> ( E )
(6) F -> id
```

Fig. 1.1

1. First and Follow:

In compiler design, the concepts of first and follow sets are important for constructing efficient and accurate parsers. Both these concepts are related to determining the set of valid symbols that can appear in a given production rule of a grammar.

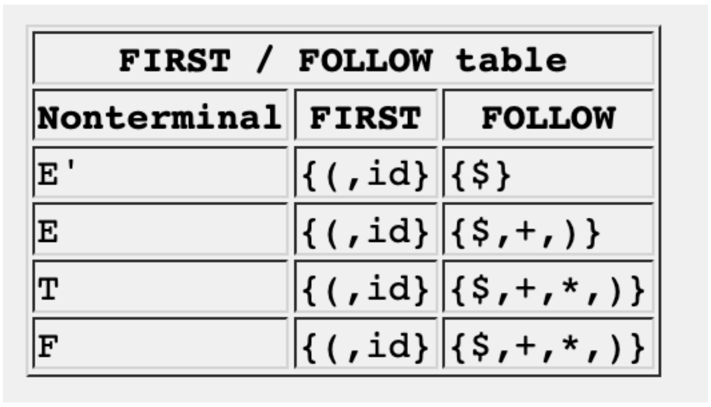
The first set of a production rule is defined as the set of terminal symbols that can appear as the first symbol in any string that can be derived from that rule. For example, if we have a production rule of the form $A \rightarrow BC$, where B and C are non-terminals, then the first set of A will be the set of all terminal symbols that can appear as the first symbol of a string derived from B or C.

The follow set of a non-terminal symbol is defined as the set of terminal symbols that can appear immediately after that symbol in any valid derivation of the grammar. For example, if we have a production rule of the form $A \rightarrow BC$, then the follow set of B will be the set of all terminal symbols that can appear immediately after B in any valid derivation of the grammar.

Both first and follow sets are essential for constructing predictive parsers such as LL(1) and LR(1) parsers. Predictive parsers use the first and follow sets to decide which production rule to use when parsing a given input string. The first set helps determine which production rule to apply based on the first symbol of the input string, while the follow set helps determine when to reduce the input string using a specific production rule.

In summary, first and follow sets are important concepts in compiler design for determining the set of valid symbols that can appear in a given production rule or derivation. These concepts play a critical role in constructing predictive parsers and are essential for efficient and accurate parsing of programming languages.

Output -



FIRST / FOLLOW table		
Nonterminal	FIRST	FOLLOW
E'	{(,id}	{$}
E	{(,id}	{$,+,)}
T	{(,id}	{$,+,*,)}
F	{(,id}	{$,+,*,)}

Fig. 1.2

2. **SLR Parser Table:**

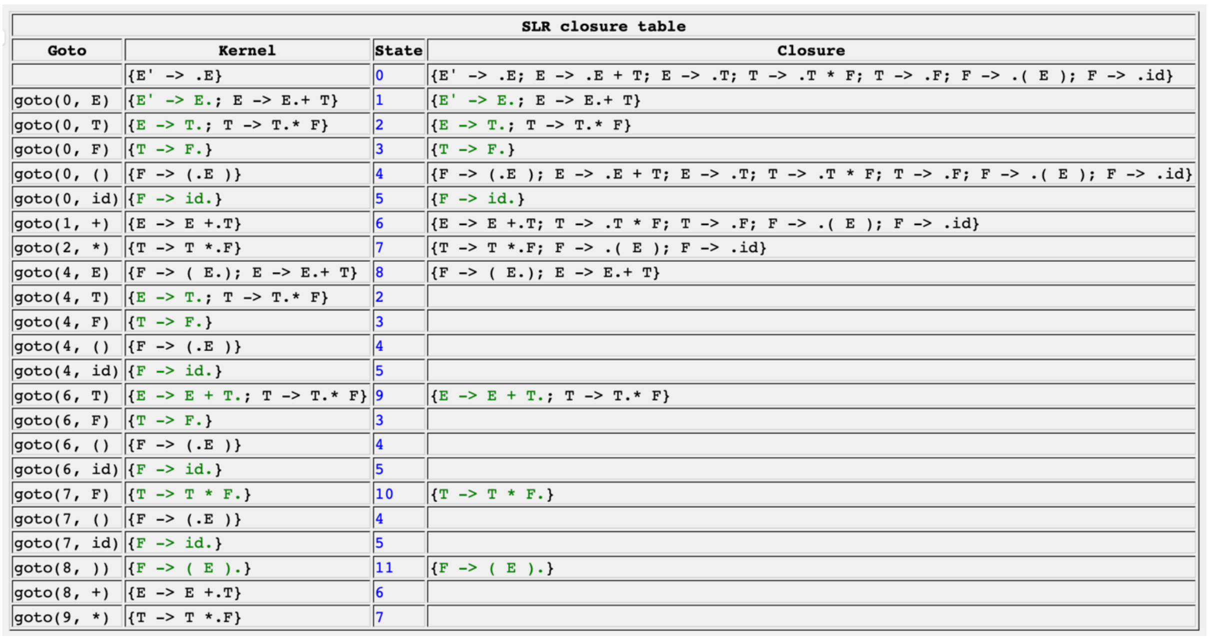
   An SLR (Simple LR) closure table is a table that is used in the process of constructing an SLR parsing table for a grammar. The closure table is used to keep track of the items that are generated by applying the closure operation to a set of LR(0) items.

   The closure operation involves expanding the set of items by adding any additional items that can be reached by applying a production rule to the items in the set. In other words, the closure operation takes the current set of items and adds all of the items that can be reached by applying a production rule to any of the items in the set. This process is repeated until no new items can be added to the set.

   The resulting closure table is a table that contains a row for each set of items generated during the closure process. Each row in the table corresponds to a set of LR(0) items and contains information about the lookahead symbols that can follow each item in the set. The lookahead symbols are used to determine the appropriate action to take when encountering a particular input symbol during parsing.

   In summary, an SLR closure table is an essential data structure used in the construction of an SLR parsing table. The table keeps track of the sets of LR(0) items generated during the closure process and provides information about the lookahead symbols that can follow each item. This information is used to determine the appropriate action to take when parsing a given input symbol.

Output:



SLR closure table			
Goto	Kernel	State	Closure
	{E' -> .E}	0	{E' -> .E; E -> .E + T; E -> .T; T -> .T * F; T -> .F; F -> .(E); F -> .id}
goto(0, E)	{E' -> E.; E -> E.+ T}	1	{E' -> E.; E -> E.+ T}
goto(0, T)	{E -> T.; T -> T.* F}	2	{E -> T.; T -> T.* F}
goto(0, F)	{T -> F.}	3	{T -> F.}
goto(0, ()	{F -> (.E)}	4	{F -> (.E); E -> .E + T; E -> .T; T -> .T * F; T -> .F; F -> .(E); F -> .id}
goto(0, id)	{F -> id.}	5	{F -> id.}
goto(1, +)	{E -> E +.T}	6	{E -> E +.T; T -> .T * F; T -> .F; F -> .(E); F -> .id}
goto(2, *)	{T -> T *.F}	7	{T -> T *.F; F -> .(E); F -> .id}
goto(4, E)	{F -> (E.); E -> E.+ T}	8	{F -> (E.); E -> E.+ T}
goto(4, T)	{E -> T.; T -> T.* F}	2	
goto(4, F)	{T -> F.}	3	
goto(4, ()	{F -> (.E)}	4	
goto(4, id)	{F -> id.}	5	
goto(6, T)	{E -> E + T.; T -> T.* F}	9	{E -> E + T.; T -> T.* F}
goto(6, F)	{T -> F.}	3	
goto(6, ()	{F -> (.E)}	4	
goto(6, id)	{F -> id.}	5	
goto(7, F)	{T -> T * F.}	10	{T -> T * F.}
goto(7, ()	{F -> (.E)}	4	
goto(7, id)	{F -> id.}	5	
goto(8,))	{F -> (E).}	11	{F -> (E).}
goto(8, +)	{E -> E +.T}	6	
goto(9, *)	{T -> T *.F}	7	

Fig. 1.3

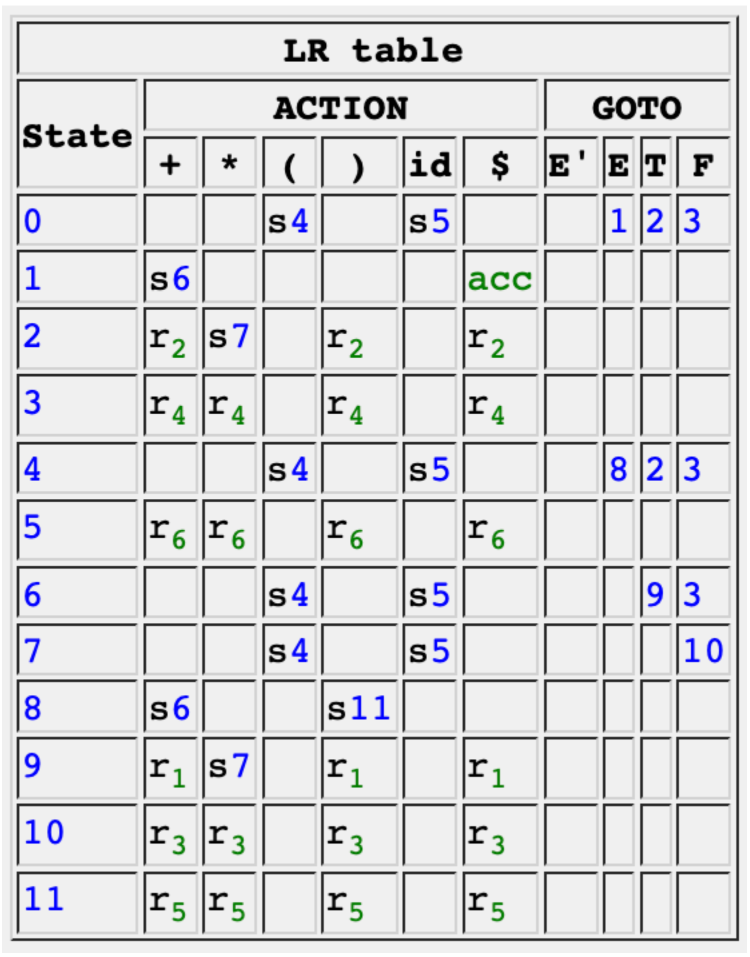
3. LR Table:

LR tables are a type of parsing table used by bottom-up parsers, which are a class of parsers that build parse trees starting from the leaves and working up to the root. LR stands for "left-to-right, rightmost derivation", which is the direction in which the parser builds the parse tree. An LR table is a table that describes the actions that an LR parser should take for each combination of input symbol and parser state. The LR table is constructed from a set of LR(0) or LR(1) items, which are the core building blocks of an LR parser. The LR table contains a set of entries for each combination of parser state and input symbol, and each entry specifies the action that the parser should take in that state when the input symbol is encountered.

The entries in the LR table can take one of several forms, depending on the type of LR parser being used. For example, an SLR parser may have entries that indicate whether to shift the next input symbol onto the parser stack, reduce a set of symbols on the parser stack to a single non-terminal symbol, or accept the input as being syntactically valid. An LALR parser, on the other hand, may have more complex entries that include lookaheads to handle conflicts between different possible parsing actions.

Overall, LR tables are a critical component of LR parsers and are used to guide the parsing process by specifying the appropriate actions to take for each input symbol and parser state.

Output:



LR table										
State	ACTION						GOTO			
	+	*	(	)	id	$	E'	E	T	F
0			s4		s5			1	2	3
1	s6					acc				
2	r_2	s7		r_2		r_2				
3	r_4	r_4		r_4		r_4				
4			s4		s5			8	2	3
5	r_6	r_6		r_6		r_6				
6			s4		s5				9	3
7			s4		s5					10
8	s6			s11						
9	r_1	s7		r_1		r_1				
10	r_3	r_3		r_3		r_3				
11	r_5	r_5		r_5		r_5				

Fig. 1.4

4. **Input Check:**

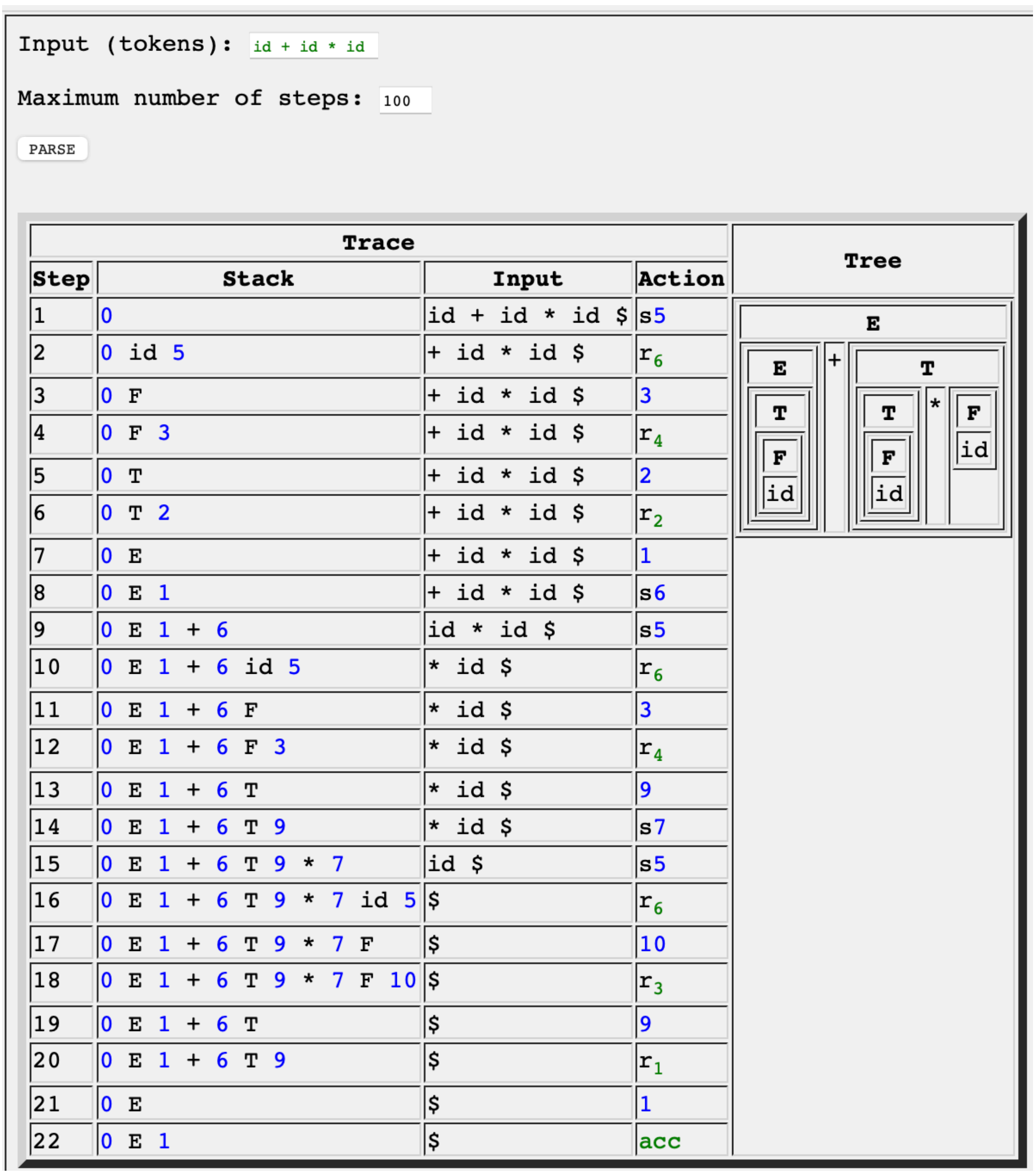
Parsing is a process of analyzing a string of symbols or tokens to determine its grammatical structure. Parsing can be performed using various techniques, such as top-down parsing, bottom-up parsing, and LR parsing. One common approach to parsing is to use a stack-based table-driven parsing algorithm.

In a stack-based table-driven parsing algorithm, the parsing process is carried out by maintaining a stack of symbols and a table that specifies the next action to be taken based on the current state and the input symbol. The stack contains both non-terminal and terminal symbols, and the table specifies the production rule to be used to reduce a set of symbols to their non-terminal equivalent or the state to shift to when encountering a terminal symbol.

To parse a string using a stack-based table-driven parsing algorithm, the input string is first split into a sequence of tokens. The initial state and the input symbol are pushed onto the stack, and the parsing process proceeds by repeatedly examining the top of the stack and the next input token to determine the appropriate action to take. If the top of the stack is a terminal symbol that matches the current input token, the terminal is popped from the stack, and the next input token is read. If the top of the stack is a non-terminal symbol, the table is consulted to determine the appropriate production rule to use to reduce the non-terminal to its equivalent.

The parsing process continues until the stack is empty, indicating successful parsing, or an error is encountered, indicating that the input string is not valid according to the grammar rules. Stack-based table-driven parsing algorithms, such as SLR, LR, and LALR parsing, are widely used in the development of compilers and programming language interpreters, as they can efficiently handle large grammars and complex input strings.

Output:



Input (tokens): id + id * id

Maximum number of steps: 100

PARSE

Trace			
Step	Stack	Input	Action
1	0	id + id * id $	s5
2	0 id 5	+ id * id $	r_6
3	0 F	+ id * id $	3
4	0 F 3	+ id * id $	r_4
5	0 T	+ id * id $	2
6	0 T 2	+ id * id $	r_2
7	0 E	+ id * id $	1
8	0 E 1	+ id * id $	s6
9	0 E 1 + 6	id * id $	s5
10	0 E 1 + 6 id 5	* id $	r_6
11	0 E 1 + 6 F	* id $	3
12	0 E 1 + 6 F 3	* id $	r_4
13	0 E 1 + 6 T	* id $	9
14	0 E 1 + 6 T 9	* id $	s7
15	0 E 1 + 6 T 9 * 7	id $	s5
16	0 E 1 + 6 T 9 * 7 id 5	$	r_6
17	0 E 1 + 6 T 9 * 7 F	$	10
18	0 E 1 + 6 T 9 * 7 F 10	$	r_3
19	0 E 1 + 6 T	$	9
20	0 E 1 + 6 T 9	$	r_1
21	0 E	$	1
22	0 E 1	$	acc

Fig 1.5

ARCHITECTURE OF LANGUAGE

The architecture of a language parser mini project using JavaScript, HTML, and CSS can be broken down into three main components: the user interface, the parsing engine, and the language grammar.

The user interface is responsible for presenting the input and output of the parser to the user in a clear and intuitive manner. The input can be provided using a text input field, while the output can be displayed using a text area or a console window. The user interface can be designed using HTML and CSS to provide a visually appealing and user-friendly experience. It can also include buttons or other interactive elements to allow the user to trigger the parsing process and interact with the output.

The parsing engine is the heart of the language parser mini project. It is responsible for parsing the input provided by the user and generating a parse tree or other output based on the grammar rules of the language. The parsing engine can be implemented using JavaScript and can incorporate various parsing algorithms, such as LL, LR, or recursive descent parsing. The choice of parsing algorithm will depend on the complexity of the grammar and the requirements of the project.

The language grammar defines the rules that govern the syntax and structure of the language being parsed. The grammar can be specified using a formal grammar notation such as BNF (Backus-Naur Form) or EBNF (Extended Backus-Naur Form) and can be implemented in JavaScript using object-oriented programming principles. The grammar should define the various language constructs, such as statements, expressions, and operators, and specify the rules for their use and combination.

The overall architecture of the language parser mini project can be summarized as follows:

The user interface provides a way for the user to input code and view the parsed output.

The parsing engine parses the input code using the language grammar and generates a parse tree or other output.

The language grammar specifies the rules that govern the syntax and structure of the language being parsed.

To implement the language parser mini project, the following steps can be taken:

Define the grammar for the language to be parsed using a formal grammar notation such as BNF or EBNF.

Implement the language grammar in JavaScript using object-oriented programming principles.

Develop the user interface using HTML and CSS, including a text input field for the user to input code and a text area or console window for the output.

Write the parsing engine using JavaScript and the chosen parsing algorithm, incorporating the language grammar and the user input.

Integrate the parsing engine with the user interface to provide a complete language parser mini project.

In conclusion, developing a language parser mini project using JavaScript, HTML, and CSS involves designing a user-friendly interface, implementing a parsing engine using JavaScript and a parsing algorithm, and specifying the language grammar using a formal notation. This project can be a valuable learning experience for those interested in programming language design and compiler construction.

TYPES OF PARSERS

1. SLR PARSER-

SLR (Simple LR) parser is a bottom-up parsing technique that can be used to parse a wide range of context-free grammars. It is a table-driven parsing technique that uses a deterministic finite automaton (DFA) to parse the input string. An SLR parser uses a parsing table, which is constructed by analyzing the grammar and the follow sets of the non-terminals in the grammar.

The SLR parser works by maintaining a stack and an input buffer. Initially, the input buffer contains the input string and the stack contains only the start symbol of the grammar. The parser repeatedly reads the next input symbol and checks the parsing table to determine the appropriate action. The parsing table entries can either instruct the parser to shift the input symbol onto the stack or to reduce the symbols on the top of the stack to their corresponding non-terminal symbol.

If the parsing table indicates that a shift operation should be performed, the input symbol is pushed onto the stack and the parser moves to the next input symbol. If the parsing table indicates that a reduce operation should be performed, the parser pops the necessary number of symbols from the stack and replaces them with their corresponding non-terminal symbol. This reduction operation may continue until the stack contains only the start symbol of the grammar, at which point the parsing is complete.

SLR parsing is efficient, but it has some limitations. For example, it cannot handle all context-free grammars, and it may produce shift-reduce conflicts and reduce-reduce conflicts in certain cases. Nevertheless, it is still widely used in the development of parsers for many programming languages, due to its simplicity and efficiency.

2. LL(1) PARSER –

An LL(1) parser is a type of top-down parser that uses a one-symbol lookahead to parse a grammar. The term "LL" stands for Left-to-right Leftmost derivation, meaning that it processes the input from left to right and generates the leftmost derivation of the input. The "1" in LL(1) refers to the one-symbol lookahead, which means that the parser only needs to look at the next token in the input to determine the next step in the parsing process.

LL(1) parsers are popular due to their simplicity and efficiency, and they are commonly used in compilers and other language processing tools. They can handle a wide range of context-free grammars, including those that are not LL(1) themselves.

The LL(1) parser works by building a predictive parse table based on the grammar rules and the lookahead symbols. This table is used to determine which production rule to use for each non-terminal symbol in the input. If a conflict arises in the parse table, it means that the grammar is not LL(1) and the parser cannot continue without additional lookahead symbols or a different parsing strategy.

One limitation of the LL(1) parser is that it cannot handle left-recursive grammar rules, where a non-terminal symbol can directly produce itself. This can be resolved by transforming the grammar rules to eliminate left recursion, or by using a different parsing strategy such as an LR parser.

Overall, the LL(1) parser is a useful and widely used parsing algorithm that can handle a wide range of context-free grammars efficiently.

3. LR(1) PARSER –

An LR(1) parser is a bottom-up parser that uses a one-symbol lookahead to parse a grammar. The "LR" stands for Left-to-right Rightmost derivation, meaning that it processes the input from left to right and generates the rightmost derivation of the input. The "1" in LR(1) refers to the one-symbol lookahead, which means that the parser only needs to look at the next token in the input to determine the next step in the parsing process.

LR(1) parsers are more powerful than LL(1) parsers because they can handle left-recursive grammar rules and more complex grammars. They are commonly used in compiler design and other language processing applications.

The LR(1) parser works by building a state machine and a parse table based on the grammar rules and the lookahead symbols. The state machine is used to track the progress of the parsing process and the parse table is used to determine which production rule to use for each non-terminal symbol in the input.

One limitation of the LR(1) parser is that it can be more complex to implement and may require more memory than an LL(1) parser. Additionally, LR(1) parsers may not be able to handle all context-free grammars, such as those with ambiguous parse trees.

Overall, the LR(1) parser is a powerful and commonly used parsing algorithm that can handle a wide range of context-free grammars. Its increased power and flexibility compared to LL(1) parsers make it a popular choice in compiler design and other language processing applications.

4. **LALR(1) PARSER –**

The LALR(1) parser, which stands for Look-Ahead LR(1) parser, is a parsing algorithm that is based on the LR(1) parser. Like the LR(1) parser, the LALR(1) parser is a bottom-up parser that uses a one-symbol lookahead to parse a grammar. The "LALR" part of the name refers to the fact that the algorithm performs a "look-ahead" operation to determine the next production rule to use in the parsing process.

The LALR(1) parser is more efficient than the LR(1) parser because it reduces the number of states and transitions in the parse table. This is achieved by merging LR(1) states that have identical core items (i.e. items with the same non-terminal symbol and dot position). This process can result in the loss of some information, which can make the LALR(1) parser more susceptible to conflicts in the parse table. However, these conflicts can often be resolved through careful grammar design or by using more advanced parsing techniques.

Overall, the LALR(1) parser is a commonly used parsing algorithm that is more efficient than the LR(1) parser. It can handle a wide range of context-free grammars and is widely used in compiler design and other language processing applications. Its reduced number of states and transitions make it more practical to implement and easier to maintain than the LR(1) parser.

DESIGN STRATEGY

1. DEFINING THE REQUIREMENTS

Before starting the project, it is essential to define the requirements. These include the type of parser to be used, the grammar rules to be parsed, and the format of the input and output. Additionally, the project requirements should be specific and measurable.

2. DEVELOPING THE GRAMMAR RULES

Once the requirements are defined, the next step is to develop the grammar rules for the parser. These rules define the syntax of the input and output and guide the parser in generating a parse tree. The grammar rules should be complete and unambiguous to avoid parsing errors.

3. SELECTING THE PARSER TYPE

The next step is to select the type of parser to be used. Depending on the complexity of the grammar rules, an LL(1), LR(1), or LALR(1) parser can be used. The choice of parser type depends on the size and complexity of the input and the performance requirements of the project.

4. BUILDING THE PARSER

The user interface is an essential component of the mini project. It should be designed using HTML and CSS to provide a user-friendly experience. The interface should allow the user to enter the input, select the parser type, and display the output. Additionally, the interface should display any parsing errors in a clear and concise manner.

5. TESTING AND DEBUGGING

Once the parser and user interface are developed, the next step is to test and debug the project. Testing involves verifying that the parser generates the correct parse tree based on the input and grammar rules. Debugging involves identifying and fixing any errors or issues that arise during testing.

6. REFINING AND ENHANCING

Finally, the mini project can be refined and enhanced to improve its performance and functionality. This may involve optimizing the parser code, adding features such as error correction or visualization of the parse tree, or improving the user interface.

IMPLEMENTATION DIAGRAMS

SLR PARSER

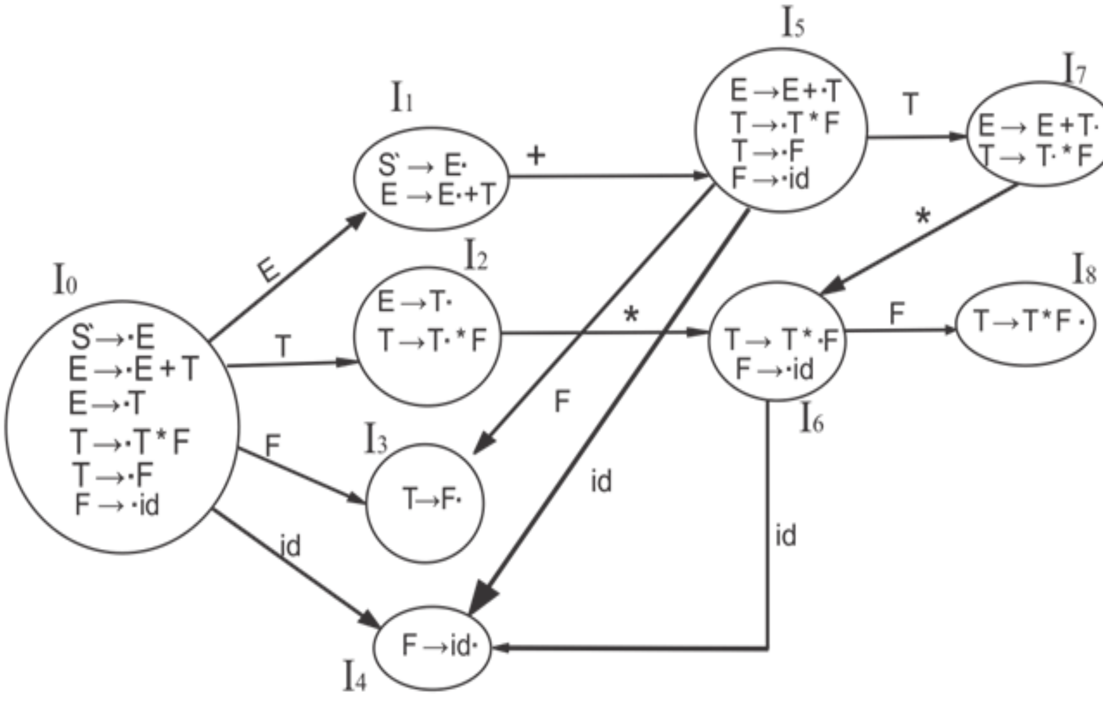


Fig. 5.1

LR PARSER

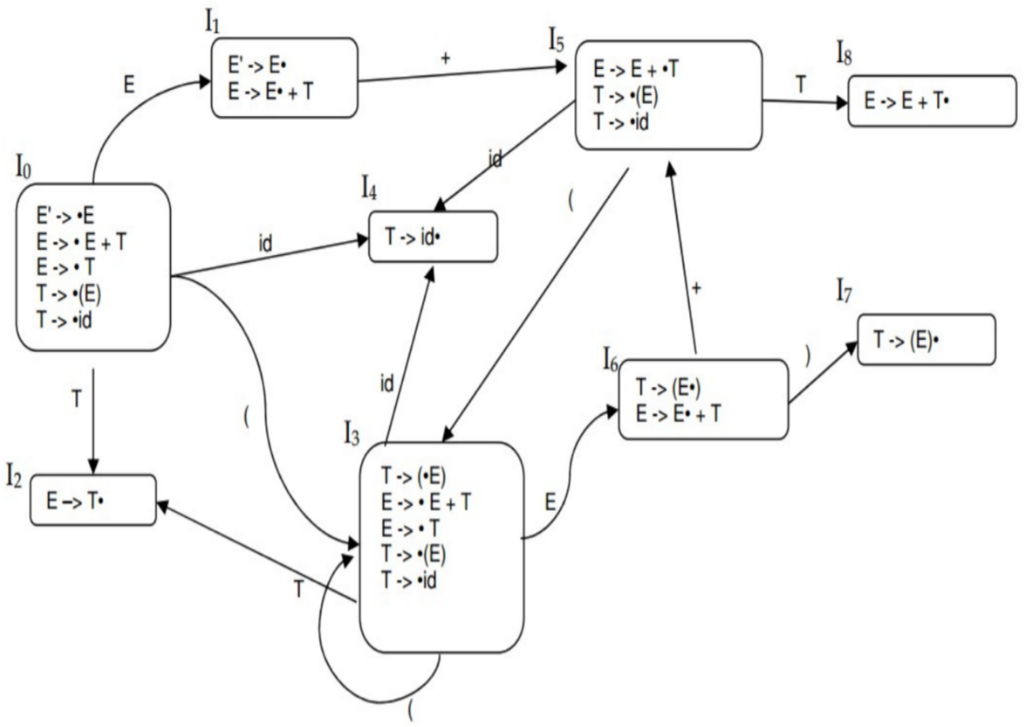
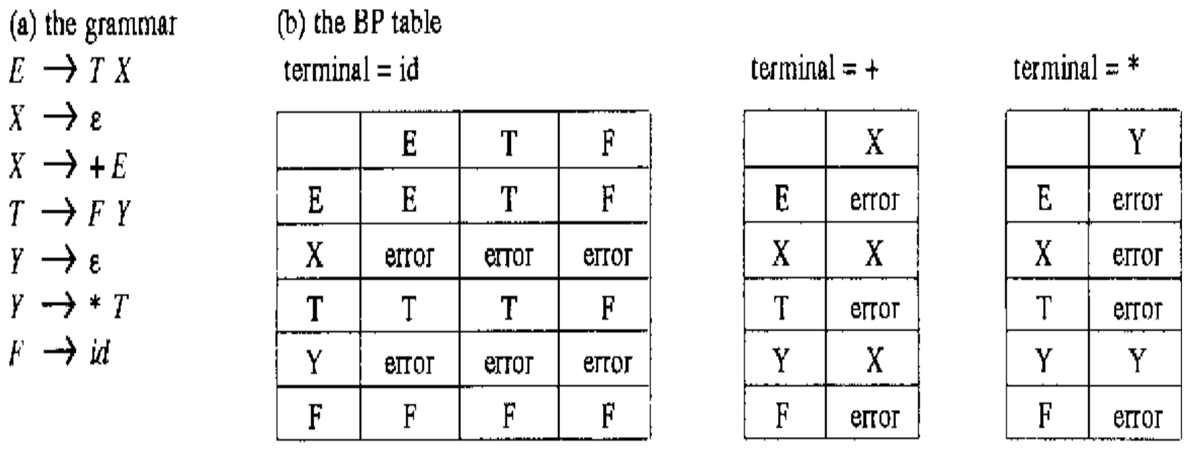


Fig. 5.2

LL(1) PARSER



(a) the grammar

$E \rightarrow T\ X$

$X \rightarrow \varepsilon$

$X \rightarrow +E$

$T \rightarrow F\ Y$

$Y \rightarrow \varepsilon$

$Y \rightarrow *\ T$

$F \rightarrow id$

(b) the BP table

terminal = id

	E	T	F
E	E	T	F
X	error	error	error
T	T	T	F
Y	error	error	error
F	F	F	F

terminal = +

	X
E	error
X	X
T	error
Y	X
F	error

terminal = *

	Y
E	error
X	error
T	error
Y	Y
F	error

Fig. 5.3

LALR(1) PARSER

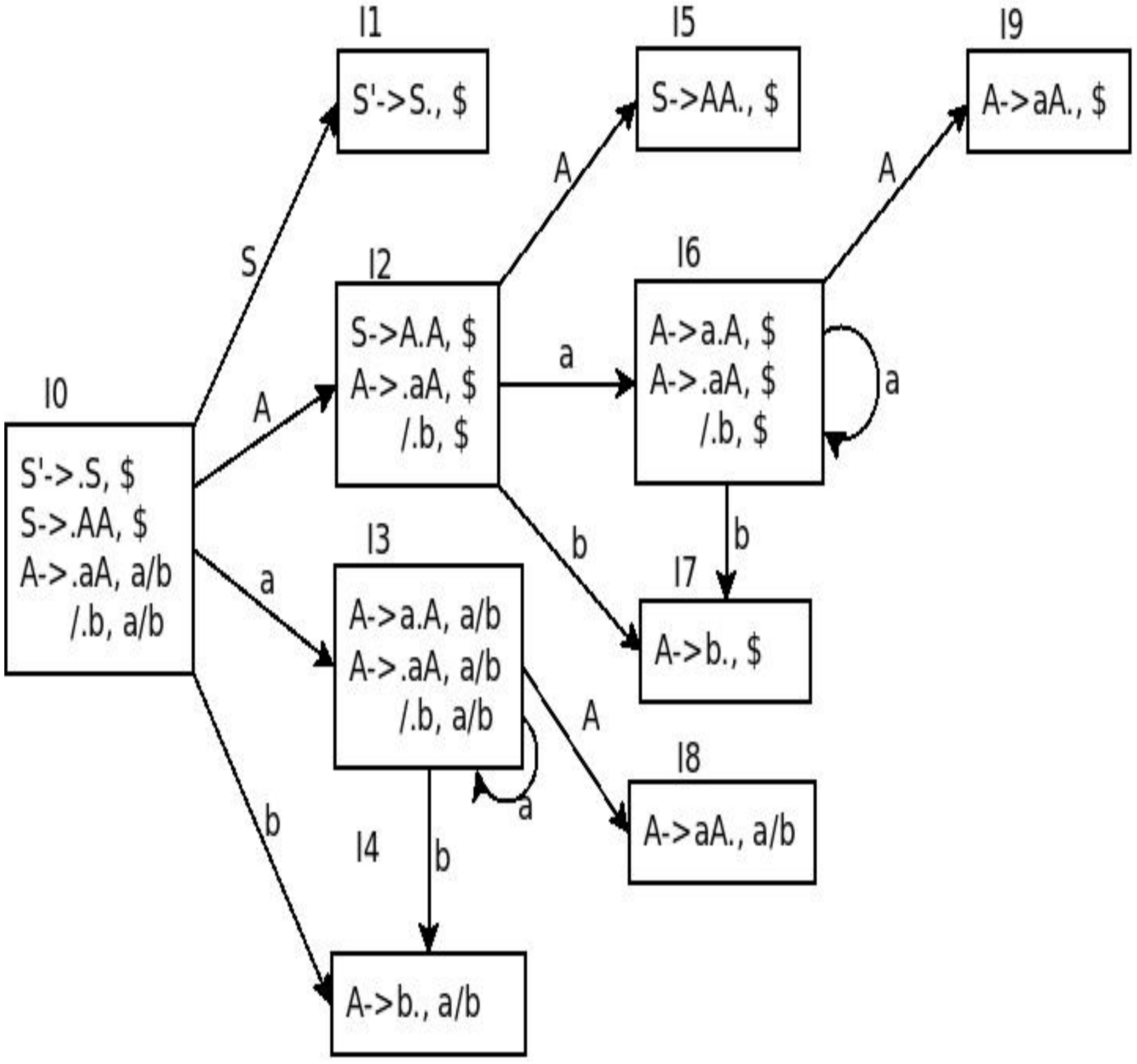


Fig. 5.4

TABLE GENERATION

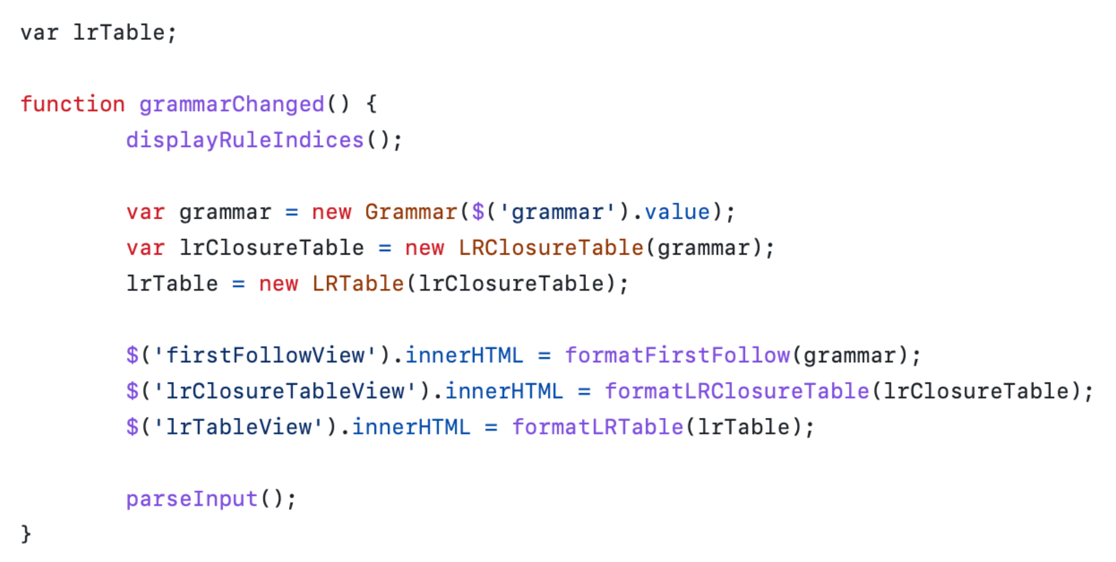
The LR table generation is a crucial step in constructing a shift-reduce parser, specifically an SLR parser. The LR table is a representation of the automaton that describes the parser's behaviour.

The table is generated using the closure and goto functions. The closure function is used to compute the set of items that can be derived from the initial state, while the goto function is used to compute the next state based on a given item and a symbol.

The LR table contains the actions and transitions for the parser. The actions include shift, reduce, and accept, while the transitions are the movements between states.

The LR table is constructed by iterating over each state and symbol and computing the corresponding action or transition. This process continues until no new states or actions are generated.

The generated LR table is then used by the parser to read the input and generate the parse tree. If a conflict arises during the parsing process, the LR table can be examined to identify the cause of the conflict and resolve it.



```
var lrTable;

function grammarChanged() {
        displayRuleIndices();

        var grammar = new Grammar($('grammar').value);
        var lrClosureTable = new LRClosureTable(grammar);
        lrTable = new LRTable(lrClosureTable);

        $('firstFollowView').innerHTML = formatFirstFollow(grammar);
        $('lrClosureTableView').innerHTML = formatLRClosureTable(lrClosureTable);
        $('lrTableView').innerHTML = formatLRTable(lrTable);

        parseInput();
}
```

Fig. 5.5

PARSER ACCESS:

The following screenshot displays how to use the parsers:

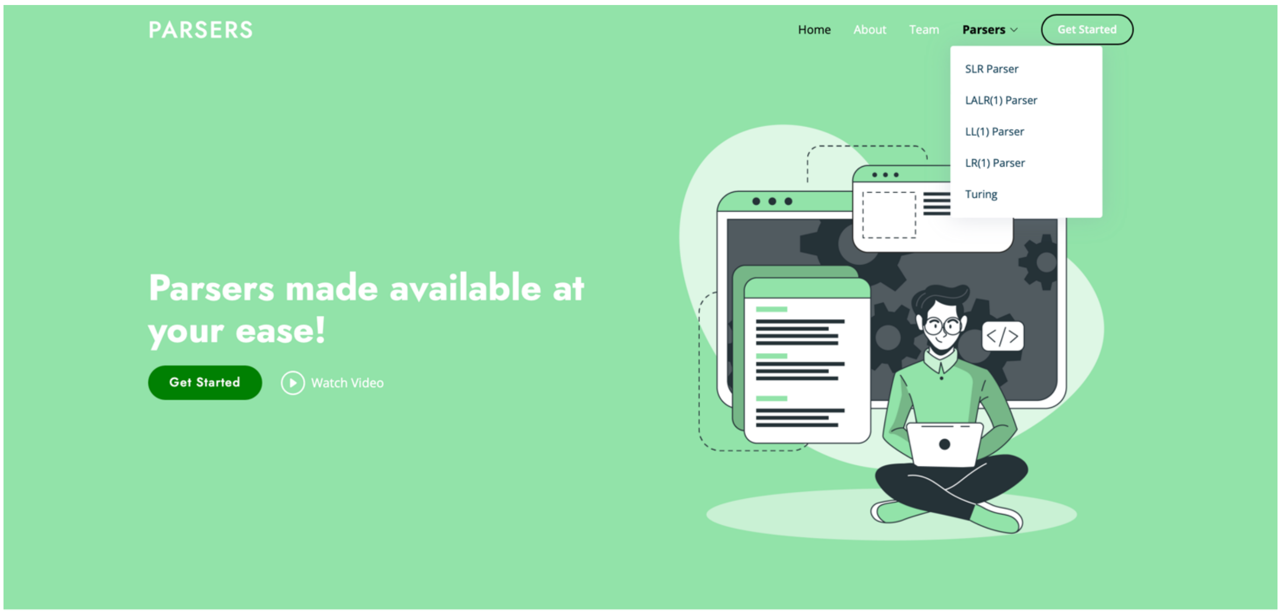


Fig. 5.6

RESULTS AND SHORTCOMINGS

The types of parsers used in a mini project can have a significant impact on its performance and accuracy. Here, we discuss the results and shortcomings of different types of parsers used in a mini project.

LL(1) Parser: An LL(1) parser is simple to implement and is efficient for a large class of grammars. However, it has some limitations. It cannot handle left-recursive grammar rules, and it may have to perform backtracking in case of conflicts in the parse table. This can lead to a decrease in performance.

LR(1) Parser: An LR(1) parser is more powerful than an LL(1) parser and can handle left-recursive grammar rules. It is also more efficient in cases where the grammar is more complex. However, it can be more difficult to implement and may require more memory than an LL(1) parser.

LALR(1) Parser: An LALR(1) parser is a compromise between an LL(1) parser and an LR(1) parser. It has the same power as an LR(1) parser but can be more efficient in terms of memory and speed. However, it may not handle all context-free grammars and may require more parsing conflicts than an LR(1) parser.

Here are the shortcomings of different types of parsers used in mini project:

LL(1) Parser:

Cannot handle left-recursive grammar rules.

May have to perform backtracking in case of conflicts in the parse table, leading to decreased performance.

LR(1) Parser:

Can be more difficult to implement than an LL(1) parser.

May require more memory than an LL(1) parser.

May not handle all context-free grammars.

May require more parsing conflicts than an LALR(1) parser.

LALR(1) Parser:

May not handle all context-free grammars.

May require more parsing conflicts than an LR(1) parser.

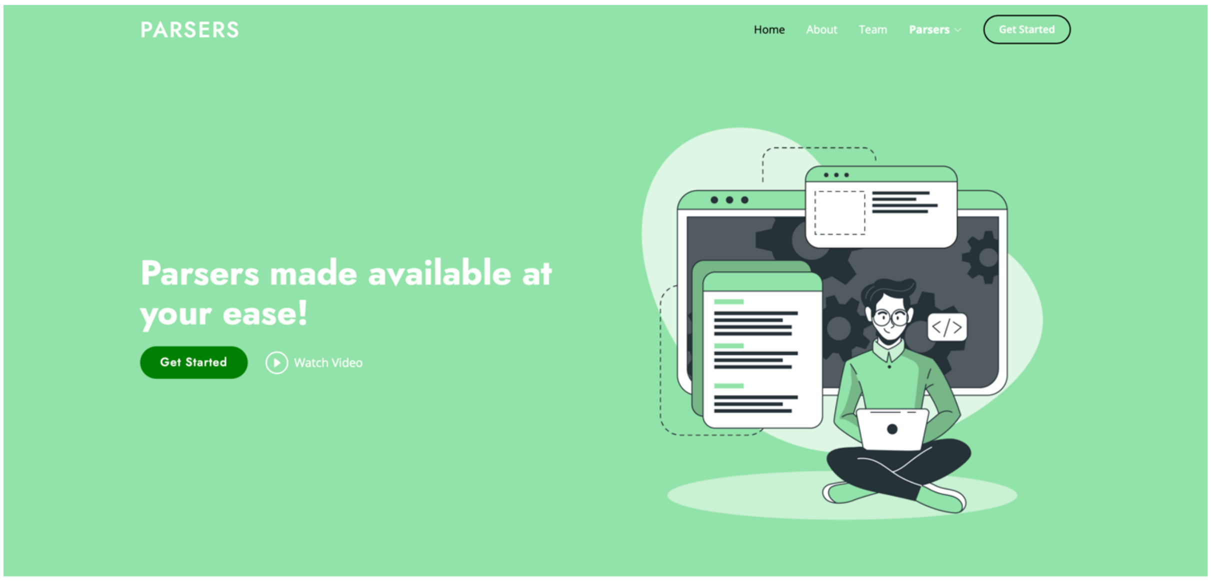
General shortcomings:

All parsers require a complete and unambiguous grammar. Any ambiguity or incompleteness in the grammar can lead to parsing errors.

The performance of a parser depends on the size and complexity of the input. As the input size grows, the time and memory requirements of the parser can become a bottleneck.

SNAPSHOTS

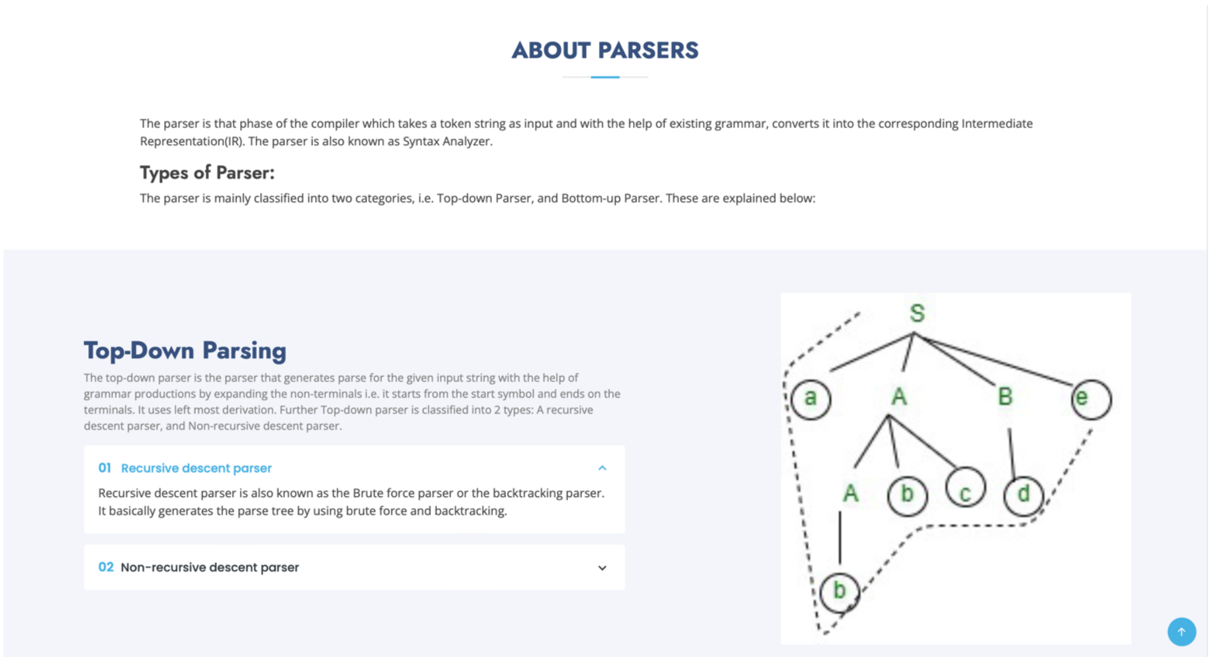
USER LANDING PAGE-



Parsers made available at your ease!

Fig. 7.1

ABOUT PARSERS:



ABOUT PARSERS

The parser is that phase of the compiler which takes a token string as input and with the help of existing grammar, converts it into the corresponding Intermediate Representation(IR). The parser is also known as Syntax Analyzer.

Types of Parser:

The parser is mainly classified into two categories, i.e. Top-down Parser, and Bottom-up Parser. These are explained below:

Top-Down Parsing

The top-down parser is the parser that generates parse for the given input string with the help of grammar productions by expanding the non-terminals i.e. it starts from the start symbol and ends on the terminals. It uses left most derivation. Further Top-down parser is classified into 2 types: A recursive descent parser, and Non-recursive descent parser.

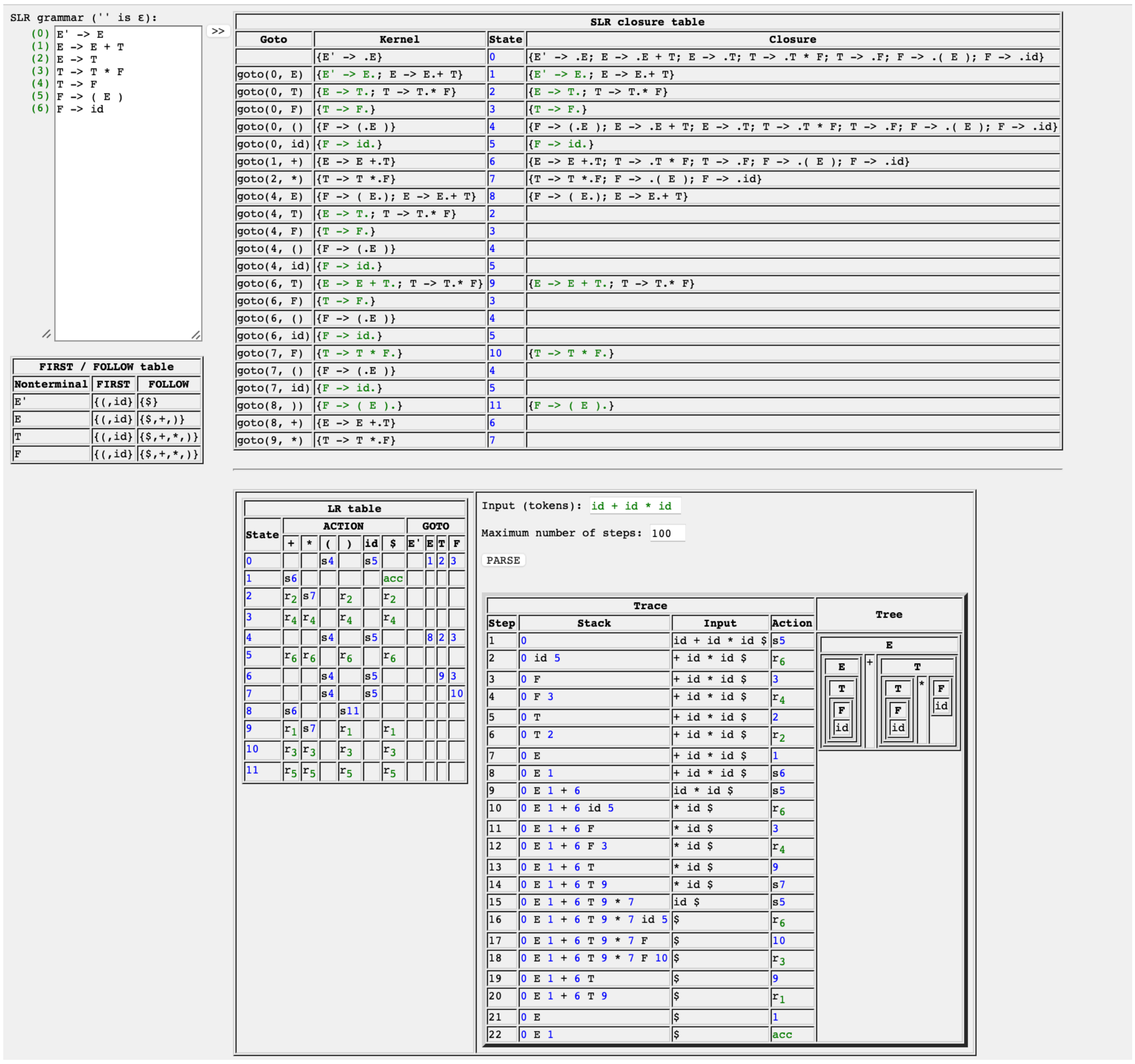
01 Recursive descent parser

Recursive descent parser is also known as the Brute force parser or the backtracking parser. It basically generates the parse tree by using brute force and backtracking.

02 Non-recursive descent parser

Fig. 7.2

SLR PARSER –



SLR grammar ('' is ε):

```
(0) E' -> E
(1) E -> E + T
(2) E -> T
(3) T -> T * F
(4) T -> F
(5) F -> ( E )
(6) F -> id
```

FIRST / FOLLOW table		
Nonterminal	**FIRST**	**FOLLOW**
E'	{(,id}	{$}
E	{(,id}	{$,+,)}
T	{(,id}	{$,+,*,)}
F	{(,id}	{$,+,*,)}

SLR closure table

Goto	Kernel	State	Closure
	{E' -> .E}	0	{E' -> .E; E -> .E + T; E -> .T; T -> .T * F; T -> .F; F -> .(E); F -> .id}
goto(0, E)	{E' -> E.; E -> E.+ T}	1	{E' -> E.; E -> E.+ T}
goto(0, T)	{E -> T.; T -> T.* F}	2	{E -> T.; T -> T.* F}
goto(0, F)	{T -> F.}	3	{T -> F.}
goto(0, ()	{F -> (.E)}	4	{F -> (.E); E -> .E + T; E -> .T; T -> .T * F; T -> .F; F -> .(E); F -> .id}
goto(0, id)	{F -> id.}	5	{F -> id.}
goto(1, +)	{E -> E +.T}	6	{E -> E +.T; T -> .T * F; T -> .F; F -> .(E); F -> .id}
goto(2, *)	{T -> T *.F}	7	{T -> T *.F; F -> .(E); F -> .id}
goto(4, E)	{F -> (E.); E -> E.+ T}	8	{F -> (E.); E -> E.+ T}
goto(4, T)	{E -> T.; T -> T.* F}	2	
goto(4, F)	{T -> F.}	3	
goto(4, ()	{F -> (.E)}	4	
goto(4, id)	{F -> id.}	5	
goto(6, T)	{E -> E + T.; T -> T.* F}	9	{E -> E + T.; T -> T.* F}
goto(6, F)	{T -> F.}	3	
goto(6, ()	{F -> (.E)}	4	
goto(6, id)	{F -> id.}	5	
goto(7, F)	{T -> T * F.}	10	{T -> T * F.}
goto(7, ()	{F -> (.E)}	4	
goto(7, id)	{F -> id.}	5	
goto(8,))	{F -> (E).}	11	{F -> (E).}
goto(8, +)	{E -> E +.T}	6	
goto(9, *)	{T -> T *.F}	7	

LR table

State	ACTION						GOTO			
	+	*	(	)	id	$	E'	E	T	F
0			s4		s5			1	2	3
1	s6					acc				
2	r_2	s7		r_2		r_2				
3	r_4	r_4		r_4		r_4				
4			s4		s5			8	2	3
5	r_6	r_6		r_6		r_6				
6			s4		s5				9	3
7			s4		s5					10
8	s6			s11						
9	r_1	s7		r_1		r_1				
10	r_3	r_3		r_3		r_3				
11	r_5	r_5		r_5		r_5				

Input (tokens): id + id * id

Maximum number of steps: 100

PARSE

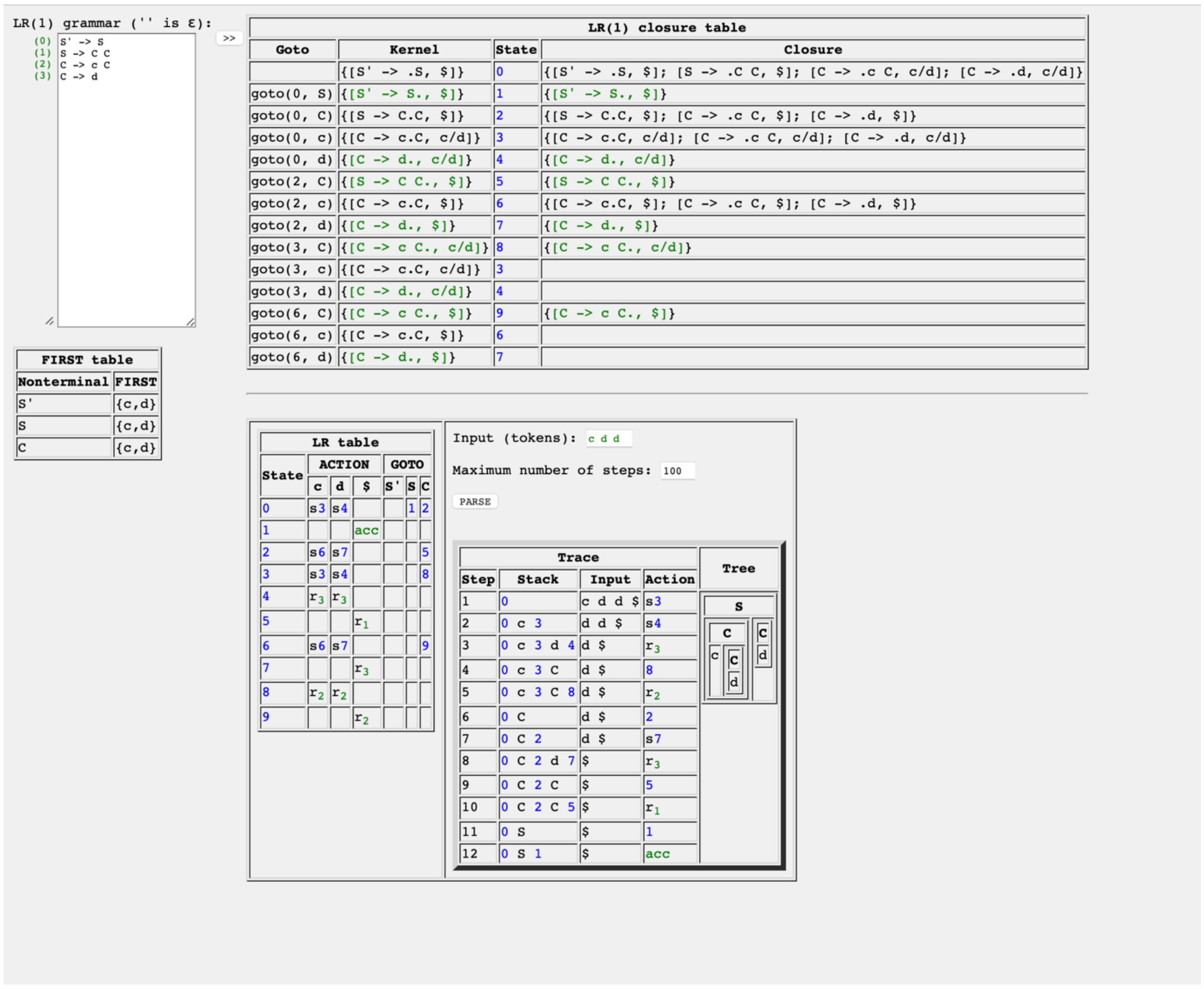
Trace

Step	Stack	Input	Action
1	0	id + id * id $	s5
2	0 id 5	+ id * id $	r_6
3	0 F	+ id * id $	3
4	0 F 3	+ id * id $	r_4
5	0 T	+ id * id $	2
6	0 T 2	+ id * id $	r_2
7	0 E	+ id * id $	1
8	0 E 1	+ id * id $	s6
9	0 E 1 + 6	id * id $	s5
10	0 E 1 + 6 id 5	* id $	r_6
11	0 E 1 + 6 F	* id $	3
12	0 E 1 + 6 F 3	* id $	r_4
13	0 E 1 + 6 T	* id $	9
14	0 E 1 + 6 T 9	* id $	s7
15	0 E 1 + 6 T 9 * 7	id $	s5
16	0 E 1 + 6 T 9 * 7 id 5	$	r_6
17	0 E 1 + 6 T 9 * 7 F	$	10
18	0 E 1 + 6 T 9 * 7 F 10	$	r_3
19	0 E 1 + 6 T	$	9
20	0 E 1 + 6 T 9	$	r_1
21	0 E	$	1
22	0 E 1	$	acc

Tree

Fig. 7.3

LR PARSER:



LR(1) grammar ('' is ε):

```
(0) S' -> S
(1) S -> C C
(2) C -> c C
(3) C -> d
```

LR(1) closure table			
Goto	**Kernel**	**State**	**Closure**
	{[S' -> .S, $]}	0	{[S' -> .S, $]; [S -> .C C, $]; [C -> .c C, c/d]; [C -> .d, c/d]}
goto(0, S)	{[S' -> S., $]}	1	{[S' -> S., $]}
goto(0, C)	{[S -> C.C, $]}	2	{[S -> C.C, $]; [C -> .c C, $]; [C -> .d, $]}
goto(0, c)	{[C -> c.C, c/d]}	3	{[C -> c.C, c/d]; [C -> .c C, c/d]; [C -> .d, c/d]}
goto(0, d)	{[C -> d., c/d]}	4	{[C -> d., c/d]}
goto(2, C)	{[S -> C C., $]}	5	{[S -> C C., $]}
goto(2, c)	{[C -> c.C, $]}	6	{[C -> c.C, $]; [C -> .c C, $]; [C -> .d, $]}
goto(2, d)	{[C -> d., $]}	7	{[C -> d., $]}
goto(3, C)	{[C -> c C., c/d]}	8	{[C -> c C., c/d]}
goto(3, c)	{[C -> c.C, c/d]}	3	
goto(3, d)	{[C -> d., c/d]}	4	
goto(6, C)	{[C -> c C., $]}	9	{[C -> c C., $]}
goto(6, c)	{[C -> c.C, $]}	6	
goto(6, d)	{[C -> d., $]}	7	

FIRST table	
Nonterminal	**FIRST**
S'	{c,d}
S	{c,d}
C	{c,d}

LR table						
State	**ACTION**			**GOTO**		
	c	**d**	**$**	**S'**	**S**	**C**
0	s3	s4			1	2
1			acc			
2	s6	s7				5
3	s3	s4				8
4	r_3	r_3				
5			r_1			
6	s6	s7				9
7			r_3			
8	r_2	r_2				
9			r_2			

Input (tokens): c d d

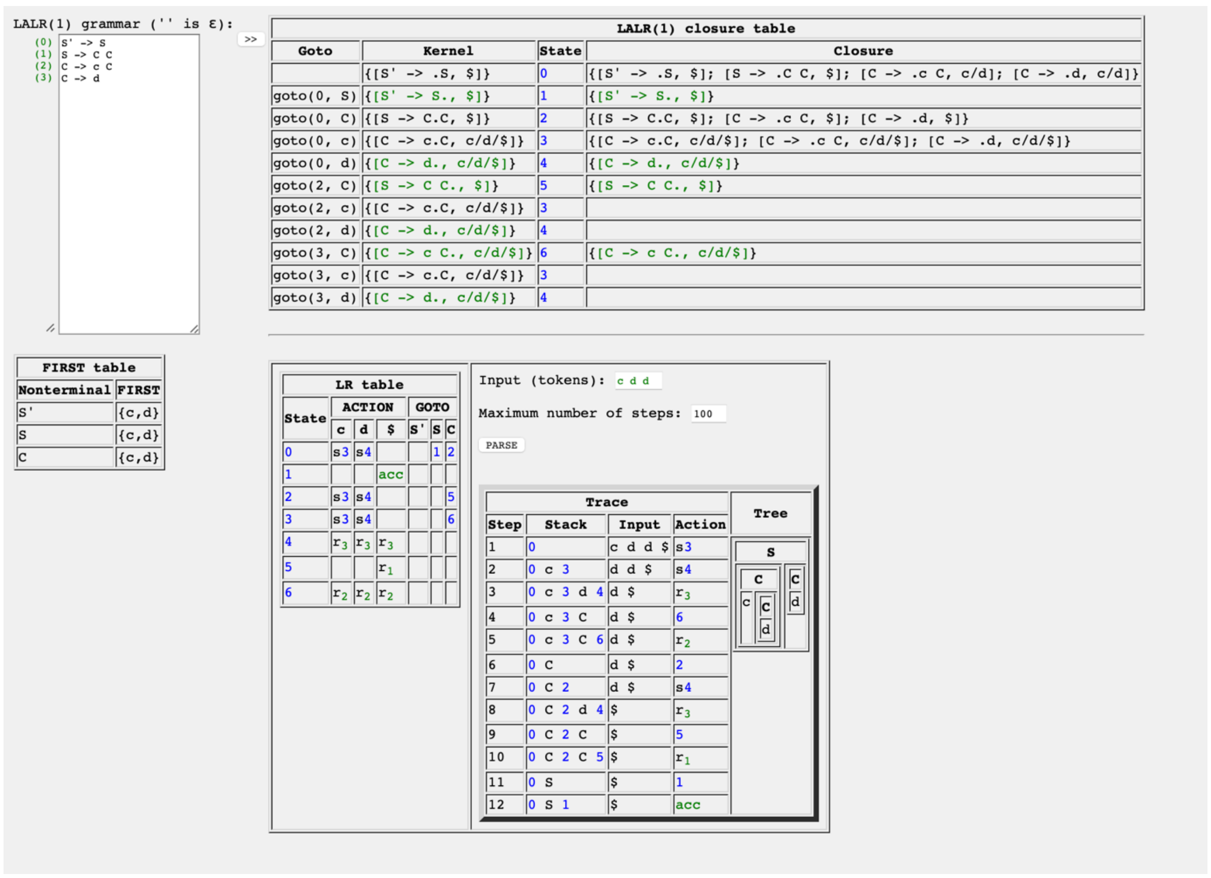
Maximum number of steps: 100

Trace			
Step	**Stack**	**Input**	**Action**
1	0	c d d $	s3
2	0 c 3	d d $	s4
3	0 c 3 d 4	d $	r_3
4	0 c 3 C	d $	8
5	0 c 3 C 8	d $	r_2
6	0 C	d $	2
7	0 C 2	d $	s7
8	0 C 2 d 7	$	r_3
9	0 C 2 C	$	5
10	0 C 2 C 5	$	r_1
11	0 S	$	1
12	0 S 1	$	acc

Tree

Fig. 7.4

LALR PARSER:



LALR(1) grammar ('' is ε):

```
(0) S' -> S
(1) S -> C C
(2) C -> c C
(3) C -> d
```

LALR(1) closure table			
Goto	Kernel	State	Closure
	{[S' -> .S, $]}	0	{[S' -> .S, $]; [S -> .C C, $]; [C -> .c C, c/d]; [C -> .d, c/d]}
goto(0, S)	{[S' -> S., $]}	1	{[S' -> S., $]}
goto(0, C)	{[S -> C.C, $]}	2	{[S -> C.C, $]; [C -> .c C, $]; [C -> .d, $]}
goto(0, c)	{[C -> c.C, c/d/$]}	3	{[C -> c.C, c/d/$]; [C -> .c C, c/d/$]; [C -> .d, c/d/$]}
goto(0, d)	{[C -> d., c/d/$]}	4	{[C -> d., c/d/$]}
goto(2, C)	{[S -> C C., $]}	5	{[S -> C C., $]}
goto(2, c)	{[C -> c.C, c/d/$]}	3	
goto(2, d)	{[C -> d., c/d/$]}	4	
goto(3, C)	{[C -> c C., c/d/$]}	6	{[C -> c C., c/d/$]}
goto(3, c)	{[C -> c.C, c/d/$]}	3	
goto(3, d)	{[C -> d., c/d/$]}	4	

FIRST table	
Nonterminal	FIRST
S'	{c,d}
S	{c,d}
C	{c,d}

LR table						
State	ACTION			GOTO		
	c	d	$	S'	S	C
0	s3	s4			1	2
1			acc			
2	s3	s4				5
3	s3	s4				6
4	r_3	r_3	r_3			
5			r_1			
6	r_2	r_2	r_2			

Input (tokens): c d d

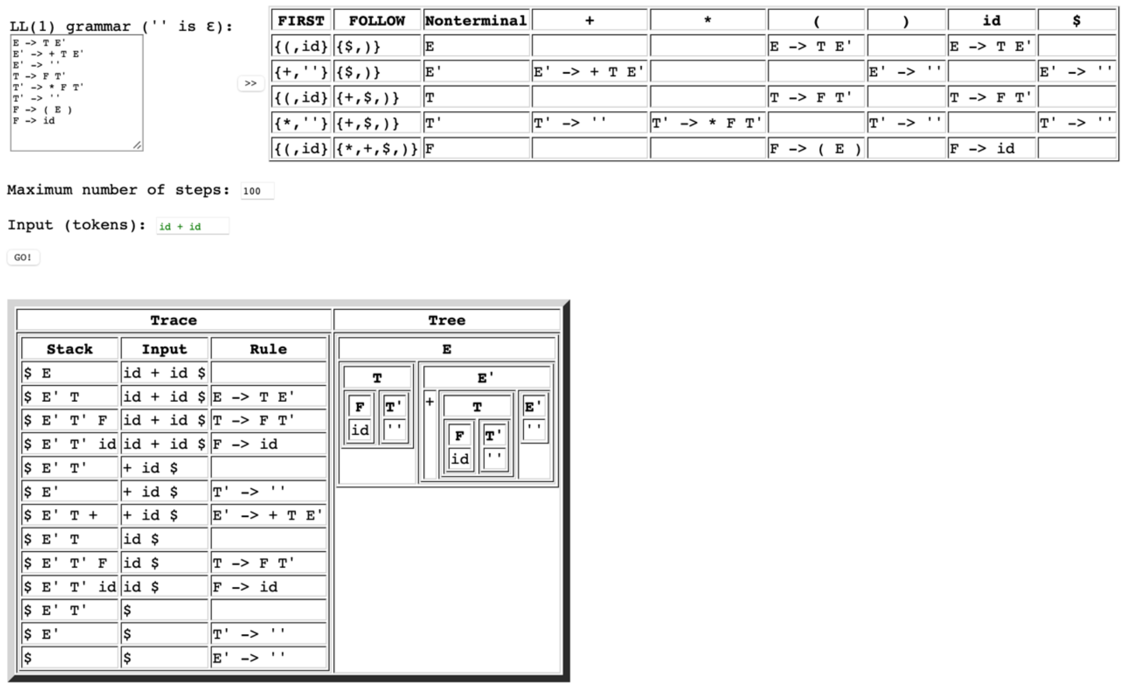
Maximum number of steps: 100

PARSE

Trace			
Step	Stack	Input	Action
1	0	c d d $	s3
2	0 c 3	d d $	s4
3	0 c 3 d 4	d $	r_3
4	0 c 3 C	d $	6
5	0 c 3 C 6	d $	r_2
6	0 C	d $	2
7	0 C 2	d $	s4
8	0 C 2 d 4	$	r_3
9	0 C 2 C	$	5
10	0 C 2 C 5	$	r_1
11	0 S	$	1
12	0 S 1	$	acc

Fig. 7.5

LL PARSER:



LL(1) grammar ('' is ε):

```
E -> T E'
E' -> + T E'
E' -> ''
T -> F T'
T' -> * F T'
T' -> ''
F -> ( E )
F -> id
```

FIRST	FOLLOW	Nonterminal	+	*	(	)	id	$
{(,id}	{$,)}	E			E -> T E'		E -> T E'	
{+,''}	{$,)}	E'	E' -> + T E'			E' -> ''		E' -> ''
{(,id}	{+,$,)}	T			T -> F T'		T -> F T'	
{*,''}	{+,$,)}	T'	T' -> ''	T' -> * F T'		T' -> ''		T' -> ''
{(,id}	{*,+,$,)}	F			F -> (E)		F -> id	

Maximum number of steps: 100

Input (tokens): id + id

Trace

Stack	Input	Rule
$ E	id + id $	
$ E' T	id + id $	E -> T E'
$ E' T' F	id + id $	T -> F T'
$ E' T' id	id + id $	F -> id
$ E' T'	+ id $	
$ E'	+ id $	T' -> ''
$ E' T +	+ id $	E' -> + T E'
$ E' T	id $	
$ E' T' F	id $	T -> F T'
$ E' T' id	id $	F -> id
$ E' T'	$	
$ E'	$	T' -> ''
$	$	E' -> ''

Fig. 7.6

CONCLUSIONS

In conclusion, the types of parsers mini project using HTML, CSS, and JavaScript provides a valuable opportunity to learn and practice the concepts and techniques involved in parsing. This project can be useful for students, researchers, and developers interested in programming languages, compilers, and related fields.

Here are some key takeaways from this mini project:

Understanding of Parser Types: The project provides a thorough understanding of the different types of parsers, including LL(1), LR(1), and LALR(1) parsers. This knowledge is essential for developing efficient and effective parsing algorithms.

Practical Experience with Parsing: The mini project offers practical experience with parsing techniques, including the design of grammar rules, the implementation of parsing algorithms, and the handling of parsing errors.

Proficiency with JavaScript: Building a mini project using JavaScript improves proficiency with this widely used programming language, including concepts such as functions, arrays, and object-oriented programming.

User Interface Design: The mini project also provides an opportunity to develop skills in user interface design using HTML and CSS. The project's interface can be designed to be user-friendly, intuitive, and visually appealing.

Debugging and Troubleshooting: Developing a mini project involves troubleshooting and debugging, which can help improve problem-solving skills and provide valuable experience for future programming projects.

Project Management: Finally, developing a mini project requires planning, organization, and project management skills. These skills are essential for working on larger projects and can be transferred to other areas of programming and software development.

FUTURE ENHANCEMENTS

There are several potential future enhancements that could be implemented in the types of parsers mini project to further improve its functionality and usefulness. Here are some possibilities:

- Error Recovery: One potential enhancement could be the addition of error recovery mechanisms to improve the parser's error-handling capabilities. This could involve implementing techniques such as panic-mode recovery or error productions to improve the parser's ability to recover from syntax errors.
- Advanced Parsing Techniques: Another potential enhancement could be the implementation of more advanced parsing techniques, such as recursive descent or GLR parsing. These techniques can provide improved parsing efficiency and handle more complex grammars.
- Grammar Visualization: Adding visualization capabilities to the parser could be an excellent enhancement. It could help users understand the parsing process and the underlying grammar more effectively. Graphical representation of the parsing process could be achieved using tools such as graphviz.
- Code Generation: Adding code generation capabilities to the parser could be an interesting enhancement. This would enable the parser to generate code in a target language such as Python, Java, or C++, based on the input code that is being parsed.

.

REFERENCES

[1]. **Parsing Expression Grammar (PEG)** -

https://en.wikipedia.org/wiki/Parsing_expression_grammar

[2]. **LL(1) Parsing** - https://en.wikipedia.org/wiki/LL_parser

[3]. **LR(1) Parsing** - https://en.wikipedia.org/wiki/LR_parser

[4]. **LALR(1) Parsing** - https://en.wikipedia.org/wiki/LALR_parser

[5]. **HTML, CSS, and JavaScript Tutorials** - https://www.w3schools.com/

[6]. **Bootstrap** - https://getbootstrap.com/