CLR PARSER

A MINI PROJECT REPORT

Submitted by

ANTARA GUPTA[RA2011031010019]
STUTI JAIN [RA2011031010031]

Under the guidance of

Dr. M Anand

(Assistant Professor, Department of
Networking and Communications)

In partial satisfaction of the requirements for the degree of

BACHELOR OF TECHNOLOGY

in

COMPUTER SCIENCE & ENGINEERING
With specialization in Information Technology

SCHOOL OF COMPUTING

COLLEGE OF ENGINEERING AND TECHNOLOGY

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

KATTANKULATHUR - 603203

APRIL 2023

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

BONAFIDE CERTIFICATE

Certified that this project report **" CLR PARSER"** is the bonafide work of **"ANTARA GUPTA[RA2011031010019], STUTI JAIN [RA2011031010031]"** 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

Dr. M. Anand
Assistant Professor
Networking and Communications

SIGNATURE

Dr. Annapurani P K
Professor & Head of Department
Networking and Communications

ABSTRACT

CLR Parser, also known as LALR Parser, is a type of bottom-up parsing algorithm used to analyze the syntax of a programming language. It is a part of the compiler front-end that reads the source code and converts it into an Abstract Syntax Tree (AST) for further processing. The CLR Parser uses a table-driven approach that reducesthe time complexity of parsing and provides a more efficient way of handling errors. This abstract summarizes the basic concept of CLR Parser and its role in the compiler front-end.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

1.1 INTRODUCTION

The CLR parser stands for canonical LR parser.It is a more powerful LR parser.It makes use of lookahead symbols. This method uses a large set of items called LR(1) items.The main difference between LR(0) and LR(1) items is that, in LR(1) items, it is possible to carry more information in a state, which will rule out useless reduction states.This extra information is incorporated into the state by the lookahead symbol. The general syntax becomes [A->∝.B, a] where A->∝.B is the production and a is a terminal or right end marker $ LR(1) items=LR(0) items + look ahead.

1.2 PROBLEM STATEMENT

- Common Language Runtime (CLR) is not working properly.
- CLR Garbage Collection is causing performance problems.
- Difficulty in understanding the CLR.

1.3 OBJECTIVES

The objective of this project is to design and implement a CLR parser compiler for a given programming language. The parser should be able to correctly parse input programs written in the programming language and produce parse trees. The parse trees should then be usedto generate intermediate code for the input programs. The parser should be designed to be efficient and powerful, making it suitable for parsing complex programs. The project should also include thorough testing and documentation to ensure its reliability and usability.

1.4 SCOPE AND APPLICATIONS

The scope of this project includes designing and implementing a CLR parser compiler for a specific programming language. The parser should be capable of handling a wide range of syntax and semantics for the given programming language. The project should also include building a lexical analyser to generate a stream of tokens from the input program, which will be used by the parser for

parsing.

The parser should produce a parse tree that represents the input program's syntax, which will be used to generate intermediate code.The intermediate code should be optimized for execution, and the project should include a back-end to generate machine code for the target architecture.

The project should be implemented in a modular way, allowing for easy extension and modification of the grammar and language features. The parser should be designed to be efficient and scalable, making it suitable for parsing large programs. The project should include comprehensive testing to ensure the parser's correctness, efficiency, and scalability.

The project should also include thorough documentation, including auser manual and developer documentation, to ensure that the parser can be used and maintained effectively.

1.5 SOFTWARE REQUIREMENTS SPECIFICATION

Software Requirements:

- Python(Pandas,Numpy)

Hardware Requirements:

- CPU : Intel Core i5
- Memory : 8GB for RAM

CHAPTER 2

SYSTEM ARCHITECTURE AND DESIGN

2.1 ARCHITECTURE DIAGRAM

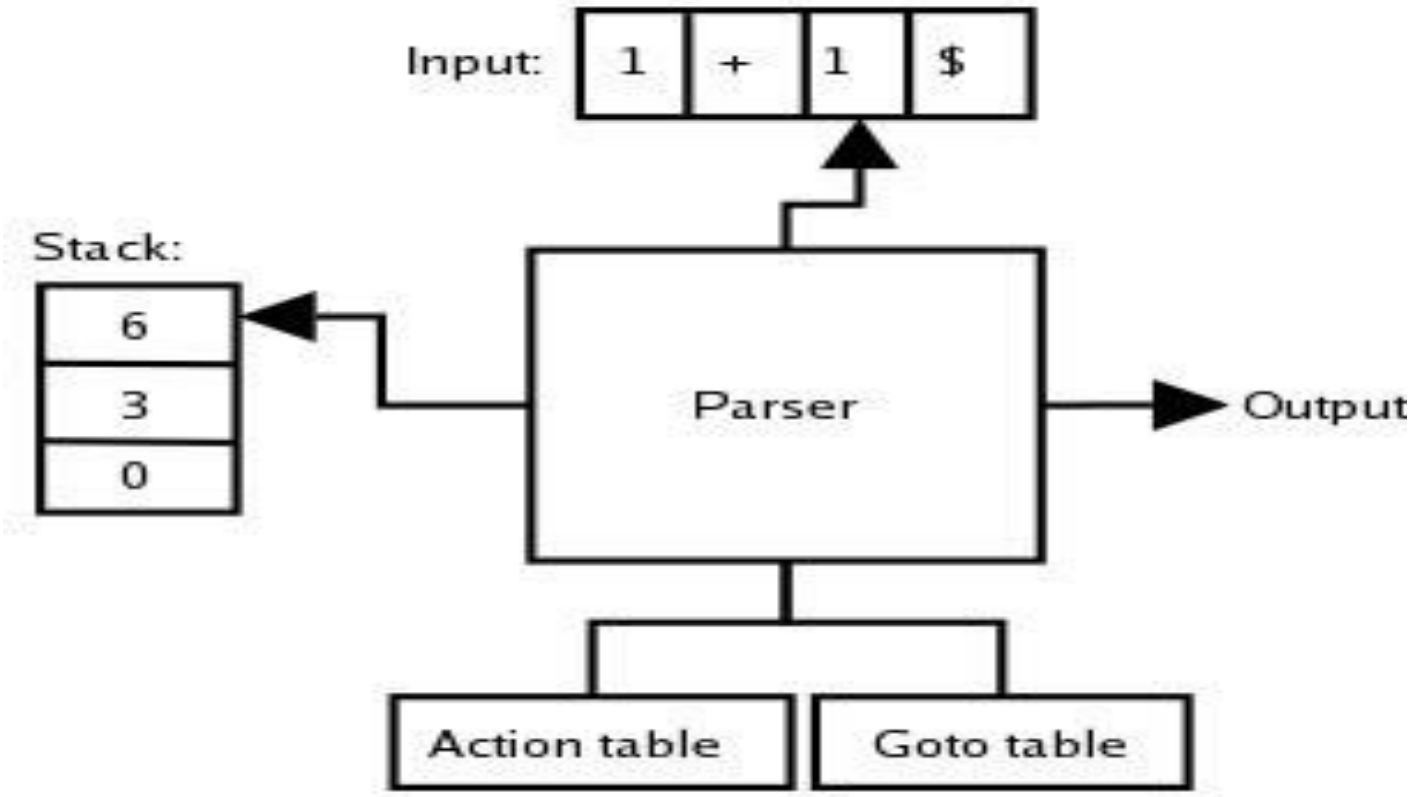

Figure 2.1 : Architecture Diagram of CLR Parser

The Common Language Runtime (CLR) is a core component of the .NET framework that manages the execution of code written in various programming languages. When a .NET application is compiled, it generates an Intermediate Language (IL) code, which is a platform-independent representation of the program. The CLR parser is responsible for reading the IL code and transforming it into machine-specific instructions that can be executed by the computer's processor. The parsing process involves several steps, including:

- Lexical Analysis: The parser first breaks down the IL code into a sequence of tokens, such as

keywords, identifiers, and operators. This process is called lexical analysis or tokenization.

- Syntax Analysis: The parser then analyzes the sequence of tokens to determine whether they form a valid program according to the language's syntax rules. This process is called syntax analysis or parsing.
- Semantic Analysis: Once the IL code has been parsed, the parser performs semantic analysis to check whether the program's meaning is consistent and correct. This involves checking for things like type mismatches, undeclared variables, and other semantic errors.
- Code Generation: After the IL code has been analyzed and validated, the parser generates machine-specific instructions that can be executed by the processor.

Overall, the CLR parser is an essential component of the .NET framework, responsible for transforming platform-independent IL code into machine-specific instructions that can be executed by the computer's processor.

CHAPTER 3

ALGORITHM AND SOURCE CODE

ALGORITHM:

CLR (Canonical LR) parser algorithm is a bottom-up parsing algorithm that uses a deterministic finite automaton (DFA) to parse a given input string. It works by constructing a parsing table and a parsetree, which represents the derivation of the input string.

Here are the steps of the CLR parsing algorithm:
Construct the LR(0) items for the given grammar. An LR(0) item is a production rule of the grammar with a dot at some position in the right-hand side of the rule, indicating the progress of the parser.

- Construct the DFA for the LR(0) items. Each state of the DFA corresponds to a set of LR(0) items, and each transition is based on the symbol following the dot in the LR(0) items.

- Compute the lookahead sets for each LR(0) item. The lookahead set for an LR(0) item is the set of terminals that could follow the non-terminal in the current state.

- Construct the parsing table. The parsing table has one entry for each pair (state, symbol), and the entry specifies the action to take when the parser is in the given state and sees the given symbol. The action could be either a shift, reduce, or accept.

- Parse the input string using the parsing table and the DFA. Starting with the initial state and the symbol on top of the stack, the parser reads symbols from the input string and performs the appropriate action according to the parsing table. If the parsing table specifies a shift, the symbol is pushed onto the stack and the parser transitions to the next state. If the parsing table specifies a reduce, the parser pops the right-hand side of

the production rule from the stack, and pushes the left-hand side onto the stack. If the parsing table specifies an accept, the parser successfully parses the input string.

- If the input string cannot be parsed using the given grammar, theparsing process will encounter a parsing error, such as a shift- reduce conflict or a reduce-reduce conflict.

CLR parsing algorithm is more powerful than other parsing algorithms such as SLR or LALR, but it is also more complex and requires more computation. However, it is often used in practice because it can handle a larger class of grammars and is efficient for parsing large input strings.

CODING:

```
import pandas as pd
import numpy as np

EPSILON = "e"


def get_productions(X):
    # This function will return all the productions X->A of the grammar
    productions = []
    for prod in grammar:
        lhs, rhs = prod.split('->')
        # Check if the production has X on LHS
        if lhs == X:
            # Introduce a dot
            rhs = '.'+rhs
            productions.append('->'.join([lhs, rhs]))
    return productions


def closure(I):
    # This function calculates the closure of the set of items I
    for production, a in I:
        # This means that the dot is at the end and can be ignored
        if production.endswith("."):
            continue
        lhs, rhs = production.split('->')
        alpha, B_beta = rhs.split('.')
        B = B_beta[0]
        beta = B_beta[1:]
```

```
        beta_a = beta + a
        first_beta_a = first(beta_a)
        for b in first_beta_a:
            B_productions = get_productions(B)
            for gamma in B_productions:
                new_item = (gamma, b)
                if (new_item not in I):
                    I.append(new_item)
    return I


def get_symbols(grammar):
    # Check the grammar and get the set of terminals and non_terminals
    terminals = set()
    non_terminals = set()
    for production in grammar:
        lhs, rhs = production.split('->')
        # Set of non terminals only
        non_terminals.add(lhs)
        for x in rhs:
            # Add add symbols to terminals
            terminals.add(x)
    # Remove the non terminals
    terminals = terminals.difference(non_terminals)
    terminals.add('$')
    return terminals, non_terminals


def first(symbols):
    # Find the first of the symbol 'X' w.r.t the grammar
    final_set = []
    for X in symbols:
        first_set = []  # Will contain the first(X)
        if isTerminal(X):
            final_set.extend(X)
            return final_set
        else:
            for production in grammar:
                # For each production in the grammar
                lhs, rhs = production.split('->')
                if lhs == X:
                    # Check if the LHS is 'X'
                    for i in range(len(rhs)):
                        # To find the first of the RHS
                        y = rhs[i]
                        # Check one symbol at a time
                        if y == X:
```

```
                        # Ignore if it's the same symbol as X
                        # This avoids infinite recursion
                        continue
                    first_y = first(y)
                    first_set.extend(first_y)
                    # Check next symbol only if first(current) contains EPSILON
                    if EPSILON in first_y:
                        first_y.remove(EPSILON)
                        continue
                    else:
                        # No EPSILON. Move to next production
                        break
                else:
                    # All symbols contain EPSILON. Add EPSILON to first(X)
                    # Check to see if some previous production has added epsilon already
                    if EPSILON not in first_set:
                        first_set.extend(EPSILON)

                    # Move onto next production
            final_set.extend(first_set)
            if EPSILON in first_set:
                continue
            else:
                break
    return final_set


def isTerminal(symbol):
    # This function will return if the symbol is a terminal or not
    return symbol in terminals


def shift_dot(production):
    # This function shifts the dot to the right
    lhs, rhs = production.split('->')
    x, y = rhs.split(".")
    if(len(y) == 0):
        print("Dot at the end!")
        return
    elif len(y) == 1:
        y = y[0]+"."
    else:
        y = y[0]+"."+y[1:]
    rhs = "".join([x, y])
    return "->".join([lhs, rhs])
```

```
def goto(I, X):
    # Function to calculate GOTO
    J = []
    for production, look_ahead in I:
        lhs, rhs = production.split('->')
        # Find the productions with .X
        if "."+X in rhs and not rhs[-1] == '.':
            # Check if the production ends with a dot, else shift dot
            new_prod = shift_dot(production)
            J.append((new_prod, look_ahead))
    return closure(J)


def set_of_items(display=False):
    # Function to construct the set of items
    num_states = 1
    states = ['I0']
    items = {'I0':  closure([('P->.S', '$')])}
    for I in states:
        for X in pending_shifts(items[I]):
            goto_I_X = goto(items[I], X)
            if len(goto_I_X) > 0 and goto_I_X not in items.values():
                new_state = "I"+str(num_states)
                states.append(new_state)
                items[new_state] = goto_I_X
                num_states += 1
    if display:
        for i in items:
            print("State", i, ":")
            for x in items[i]:
                print(x)
            print()

    return items


def pending_shifts(I):
    # This function will check which symbols are to be shifted in I
    symbols = []  # Will contain the symbols in order of evaluation
    for production, _ in I:
        lhs, rhs = production.split('->')
        if rhs.endswith('.'):
            # dot is at the end of production. Hence, ignore it
            continue
        # beta is the first symbol after the dot
        beta = rhs.split('.')[1][0]
        if beta not in symbols:
```

```
            symbols.append(beta)
    return symbols


def done_shifts(I):
    done = []
    for production, look_ahead in I:
        if production.endswith('.') and production != 'P->S.':
            done.append((production[:-1], look_ahead))
    return done


def get_state(C, I):
    # This function returns the State name, given a set of items.
    key_list = list(C.keys())
    val_list = list(C.values())
    i = val_list.index(I)
    return key_list[i]


def CLR_construction(num_states, terminals, non_terminals):
    # Function that returns the CLR Parsing Table function ACTION and GOTO
    C = set_of_items()  # Construct collection of sets of LR(1) items

    # Initialize two tables for ACTION and GOTO respectively
    ACTION = pd.DataFrame(columns=list(terminals), index=range(num_states))
    GOTO = pd.DataFrame(columns=list(non_terminals), index=range(num_states))

    # terminals = ['a', 'b', 'c']
    # num_states = 5

    # ACTION = pd.DataFrame(columns=[(col, '') for col in terminals],
index=range(num_states))
    # GOTO = pd.DataFrame(columns=[(col, '') for col in terminals], index=range(num_states))
    for Ii in C.values():
        # For each state in the collection
        i = int(get_state(C, Ii)[1:])
        pending = pending_shifts(Ii)
        for a in pending:
            # For each symbol 'a' after the dots
            Ij = goto(Ii, a)
            j = int(get_state(C, Ij)[1:])
            if isTerminal(a):
                # Construct the ACTION function
                ACTION.at[i, a] = "Shift "+str(j)
            else:
                # Construct the GOTO function
```

```
                GOTO.at[i, a] = j

        # For each production with dot at the end
        for production, look_ahead in done_shifts(Ii):
            # Set GOTO[I, a] to "Reduce"
            ACTION.at[i, look_ahead] = "Reduce " + str(grammar.index(production)+1)

        # If start production is in Ii
        if ('P->S.', '$') in Ii:
            ACTION.at[i, '$'] = "Accept"

    # Remove the default NaN values to make it clean
    ACTION.replace(np.nan, '', regex=True, inplace=True)
    GOTO.replace(np.nan, '', regex=True, inplace=True)

    return ACTION, GOTO


def get_grammar(filename):
    grammar = []
    F = open(filename, "r")
    for production in F:
        grammar.append(production[:-1])
    return grammar


if __name__ == "__main__":

    # Grammars
    #grammar = ['S->S+T', 'S->T', 'T->T*F', 'T->F', 'F->(S)', 'F->i']
    #grammar = ['S->CC', 'C->cC', 'C->d']
    # grammar = ['S->L=R', 'S->R', 'L->*R', 'L->i', 'R->L']
    grammar = get_grammar("grammar1.txt")
    #grammar = get_grammar("grammar2.txt")
    # grammar = ['S->AB', 'A->aB', 'S->A', 'A->B', 'B->C', 'C->d']
    terminals, non_terminals = get_symbols(grammar)
    symbols = terminals.union(non_terminals)

    # Demonstrating main functions
    start = [('P->.S', '$')]
    I0 = closure(start)
    goto(I0, '*')
    C = set_of_items(display=True)
    ACTION, GOTO = CLR_construction(num_states=len(C), terminals=terminals,
non_terminals=non_terminals)

    # Demonstrating helper functions:
```

```
    get_productions('L')
    shift_dot('L->.*R')
    pending_shifts(I0)
```

CHAPTER 4

RESULTS AND DISCUSSIONS

6.1 RESULT/OUTPUT:

State I0 :
('P->.S', '$')
('S->.S+T', '$')
('S->.T', '$')
('S->.S+T', '+')
('S->.T', '+')
('T->.T*F', '$')
('T->.F', '$')
('T->.T*F', '+')
('T->.F', '+')
('T->.T*F', '*')
('T->.F', '*')
('F->.(S)', '$')
('F->.i', '$')
('F->.(S)', '+')
('F->.i', '+')
('F->.(S)', '*')
('F->.i', '*')

State I1 :
('P->S.', '$')
('S->S.+T', '$')
('S->S.+T', '+')

State I2 :
('S->T.', '$')
('S->T.', '+')
('T->T.*F', '$')
('T->T.*F', '+')
('T->T.*F', '*')

State I3 :

('T->F.', '$')
('T->F.', '+')
('T->F.', '*')

State I4 :
('F->(.S)', '$')
('F->(.S)', '+')
('F->(.S)', '*')
('S->.S+T', ')')
('S->.T', ')')
('S->.S+T', '+')
('S->.T', '+')
('T->.T*F', ')')
('T->.F', ')')
('T->.T*F', '+')
('T->.F', '+')
('T->.T*F', '*')
('T->.F', '*')
('F->.(S)', ')')
('F->.i', ')')
('F->.(S)', '+')
('F->.i', '+')
('F->.(S)', '*')
('F->.i', '*')

State I5 :
('F->i.', '$')
('F->i.', '+')
('F->i.', '*')

State I6 :
('S->S+.T', '$')
('S->S+.T', '+')
('T->.T*F', '$')
('T->.F', '$')
('T->.T*F', '+')
('T->.F', '+')
('T->.T*F', '*')
('T->.F', '*')
('F->.(S)', '$')
('F->.i', '$')

('F->.(S)', '+')
('F->.i', '+')
('F->.(S)', '*')
('F->.i', '*')

State I7 :
('T->T*.F', '$')
('T->T*.F', '+')
('T->T*.F', '*')
('F->.(S)', '$')
('F->.i', '$')
('F->.(S)', '+')
('F->.i', '+')
('F->.(S)', '*')
('F->.i', '*')

State I8 :
('F->(S.)', '$')
('F->(S.)', '+')
('F->(S.)', '*')
('S->S.+T', ')')
('S->S.+T', '+')

State I9 :
('S->T.', ')')
('S->T.', '+')
('T->T.*F', ')')
('T->T.*F', '+')
('T->T.*F', '*')

State I10 :
('T->F.', ')')
('T->F.', '+')
('T->F.', '*')

State I11 :
('F->(.S)', ')')
('F->(.S)', '+')
('F->(.S)', '*')
('S->.S+T', ')')
('S->.T', ')')

('S->.S+T', '+')
('S->.T', '+')
('T->.T*F', ')')
('T->.F', ')')
('T->.T*F', '+')
('T->.F', '+')
('T->.T*F', '*')
('T->.F', '*')
('F->.(S)', ')')
('F->.i', ')')
('F->.(S)', '+')
('F->.i', '+')
('F->.(S)', '*')
('F->.i', '*')

State I12 :
('F->i.', ')')
('F->i.', '+')
('F->i.', '*')

State I13 :
('S->S+T.', '$')
('S->S+T.', '+')
('T->T.*F', '$')
('T->T.*F', '+')
('T->T.*F', '*')

State I14 :
('T->T*F.', '$')
('T->T*F.', '+')
('T->T*F.', '*')

State I15 :
('F->(S).', '$')
('F->(S).', '+')
('F->(S).', '*')

State I16 :
('S->S+.T', ')')
('S->S+.T', '+')
('T->.T*F', ')')

('T->.F', ')')
('T->.T*F', '+')
('T->.F', '+')
('T->.T*F', '*')
('T->.F', '*')
('F->.(S)', ')')
('F->.i', ')')
('F->.(S)', '+')
('F->.i', '+')
('F->.(S)', '*')
('F->.i', '*')

State I17 :
('T->T*.F', ')')
('T->T*.F', '+')
('T->T*.F', '*')
('F->.(S)', ')')
('F->.i', ')')
('F->.(S)', '+')
('F->.i', '+')
('F->.(S)', '*')
('F->.i', '*')

State I18 :
('F->(S.)', ')')
('F->(S.)', '+')
('F->(S.)', '*')
('S->S.+T', ')')
('S->S.+T', '+')

State I19 :
('S->S+T.', ')')
('S->S+T.', '+')
('T->T.*F', ')')
('T->T.*F', '+')
('T->T.*F', '*')

State I20 :
('T->T*F.', ')')
('T->T*F.', '+')
('T->T*F.', '*')

State I21 :
('F->(S).', ')')
('F->(S).', '+')
('F->(S).', '*')

CHAPTER 5

CONCLUSION

7.1 CONCLUSION

CLR (Canonical LR) parser algorithm is a bottom-up parsing algorithm that uses a deterministic finite automaton (DFA) to parse a

given input string. It works by constructing a parsing table and a parse tree, which represents the derivation of the input string. CLR parsing algorithm is more powerful than other parsing algorithms such as SLR or LALR, but it is also more complex and requires more computation. However, it is often used in practice because it can handle a larger class of grammars and is efficient for parsing large input strings. Overall, CLR parsing is an important technique for compilers and programming language theory.

REFERENCES

- — HTTP://www.javatpoint.com/clr-1-parsing
- —https://www.geeksforgeeks.org/clr-parser-with-examples/