BUILDING A CLR PARSER IN PYTHON

A COURSE PROJECT REPORT

By

Roshana S V (RA2011027010074)

A.S. Sri Ram (RA2011027010093)

Sanjukta Goswami (RA2011027010110)

Under the guidance of

Dr. S. Sharanya

In partial fulfillment for the Course

of

18CSC304J – COMPILER DESIGN

in

Data Science and Business Systems

FACULTY OF ENGINEERING AND TECHNOLOGY

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

Kattankulathur, Chengalpattu District
APRIL 2023

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

(Under Section 3 of UGC Act, 1956)

BONAFIDE CERTIFICATE

This is to certify that this project report " **Building A Compiler For Pl/0 Language In C**" is the bonafide work of **Roshana S V (RA2011027010074), A.S. Sri Ram(RA2011027010093) and Sanjukta Goswami(RA2011027010110)** who carried out the project work under my supervision.

Dr. S. Sharanya,
Subject Staff
Assistant
Professor,
Data Science and Business Systems
SRM Institute of Science and
Technology Potheri, SRM Nagar,
Kattankulathur, Tamil Nadu 603203

TABLE OF CONTENTS

OBJECTIVE AND SCOPE

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 used to 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.

SCOPE

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 a user manual and developer documentation, to ensure that the parser can be used and maintained effectively

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 reduces the 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.

INTRODUCTION

CLR Parser, also known as LALR Parser, is a 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 reduces the time complexity of parsing and provides a more efficient way of handling errors. The CLR Parser is based on the LALR(1) parsing algorithm, which stands for Look-Ahead Left-to-Right, Rightmost Derivation, with one symbol of look-ahead.

It is an extension of the LR(1) parsing algorithm that can handle a larger class of context-free grammars.
The CLR Parser uses a set of tables that are generated automatically from the grammar of the programming language. These tables contain the states of the parser, the transitions between the states, and the actions to be taken when a particular symbol is encountered.

The parsing process involves shifting symbols onto a stack, reducing them to nonterminals, and ultimately building an AST that represents the structure of the program.

One advantage of the CLR Parser is its efficiency, as it can parse a large class of context-free grammars in linear time. Additionally, the CLR Parser can provide detailed error messages, making it easier for developers to identify and correct syntax errors in their code.

Overall, CLR Parser is an important component of the compiler front- end that plays a critical role in the compilation process of programming languages.

HARDWARE/SOFTWARE REQUIREMENTS

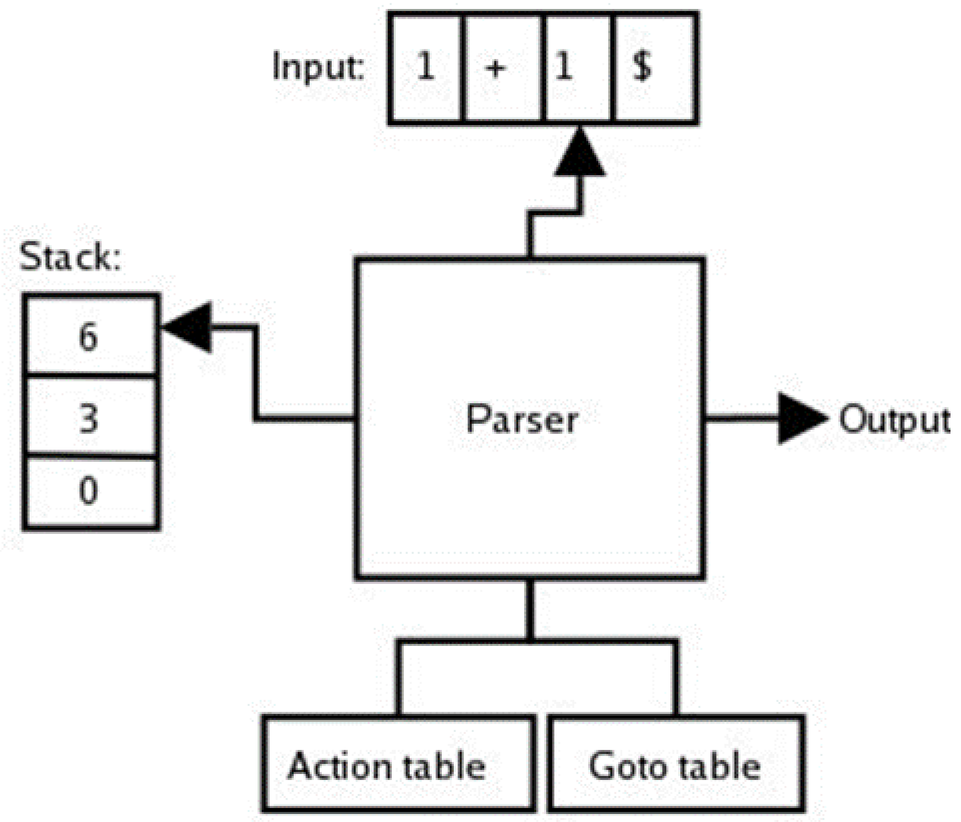
Software Requirements:

- Python(Pandas,Numpy)

Hardware Requirements:

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

ARCHITECTURE DIAGRAM



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 parse tree, 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, the parsing 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.

SOURCE CODE

```
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

epsilon already

else:
    # All symbols contain EPSILON. Add EPSILON to first(X) # Check to see if some previous
    production has added

        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)
```

RESULT/OUTPUT

```
grammar = ['S->S+T', 'S->T', 'T->T*F', 'T->F', 'F->(S)', 'F->i']
```

```
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', '*')
```

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

https://www.javatpoint.com/clr-1-parsing

https://www.geeksforgeeks.org/clr-parser-with- examples

https://www.cs.cmu.edu/~aplatzer/course/Compilers/waitegoos.pdf

https://www.webopedia.com/definitions/high-level-language/

https://en.wikibooks.org/wiki/Introduction_to_Programming_Languages/Grammars?fbclid=
I wAR0nLkq2rIAyA5DbDRHBXYpHWsNo21XYas-7GjeUe82G-DWtdAydk8oeBys

https://softwareengineering.stackexchange.com/questions/165543/how-to-write-a-very-basic-
compiler

https://visualstudiomagazine.com/articles/2014/05/01/how-to-write-your-own-compiler-part-
1.aspx