

Compiler term project

20202586학번 강노겸 | 20204093 이승현

1.CFG ambiguity removal

The final CFG used in our project is saved in the submission directory as “cfg.txt”. Opening it shows the following:

CODE' -> CODE

CODE -> VDECL CODE | FDECL CODE | ''

VDECL -> vtype id semi | vtype ASSIGN semi

ASSIGN -> id assign RHS

RHS -> EXPR | literal | character | boolstr

EXPR -> EXPR addsub TERM | TERM

TERM -> TERM multdiv FACTOR | FACTOR

FACTOR -> lparen EXPR rparen | id | num

FDECL -> vtype id lparen ARG rparen lbrace BLOCK RETURN rbrace

ARG -> vtype id MOREARGS | ''

MOREARGS -> comma vtype id MOREARGS | ''

BLOCK -> STMT BLOCK | ''

STMT -> VDECL | ASSIGN semi | if lparen COND rparen lbrace BLOCK rbrace ELSE | while lparen COND rparen lbrace BLOCK rbrace

ELSE -> else lbrace BLOCK rbrace | ''

COND -> boolstr COND'

COND' -> comp boolstr COND' | ''

RETURN -> return RHS semi

The notable sections in the above CFG that differ from the original CFG are highlighted above, and their descriptions are as follows:

a. Separation of ‘EXPR’ and ‘TERM’

From the original CFG that was responsible for production of expressions including the four basic operators,

Original CFG:

EXPR-> EXPR addsub EXPR | EXPR multdiv EXPR | lparen EXPR rparen | id | num

We made some modifications to separate the addition/subtraction (lower precedence) from multiplication/division (higher precedence) by introducing new non-terminals ‘TERM’ and ‘FACTOR’. The addition of this precedence rules ensures correct parsing of input sequences that deal with expressions including multiple operators, resolving potential ambiguities in expression evaluation order.

Modified CFG:

EXPR -> EXPR addsub TERM | TERM

TERM -> TERM multdiv FACTOR | FACTOR

FACTOR -> lparen EXPR rparen | id | num

b. Modifications in the grammar regarding Conditional expressions

Original CFG:

COND -> COND comp COND | boolstr

The part in the original CFG that handles conditional expressions allow for condensed conditional comparisons, when the flow of conditional checks should simply be from left to right. The changes made to the CFG introduces a new non-terminal ‘COND ’‘ to handle multiple comparisons more clearly, by enforcing a strict rule in the comparison sequence.

Modified CFG:

COND -> boolstr COND'

COND' -> comp boolstr COND' | ''

2.SLR parsing table

The parsing table used for the syntax analyzer implementation is included in the submission directory as parsing\_table.csv. We’ve also included a parsing\_table.xlsx file for convenience.

3.Syntax analyzer description

We’ve realized that it is possible to integrate an error detecting system as well as a parse tree construction algorithm into the parsing method, and in the end created a single function that takes an input token sequence to return a root node from the constructed parse tree.

An example execution of the syntax analyzer can be done through the command line as follows:

in linux:

python syntax\_analyzer.py test.txt

On execution, the program first reads the input token sequence from the filename specified by the input argument using the read\_input\_file(file\_path) method defined below.

def read\_input\_file(file\_path):

    try:

        with open(file\_path, 'r', encoding='utf-8') as file:

            tokens = file.read().split()

            return tokens

    except FileNotFoundError:

        print(f"{file\_path} 파일을 찾지 못했습니다.")

    except Exception as e:

        print(f"오류가 발생했습니다:{e}")

this function reads the contents of the input file and tokenizes the strings separated by whitespace. Then it returns the list of tokens.

The SLR table to be used by the parser within the script is generated by the function shown below. It takes the .csv file in the submission directory that contains our SLR table generated online.

def create\_parsing\_table(filename):

    # Read the parsing table from CSV

    # Initialize SLR table

    slr\_table = {

        'action': {},

        'goto': {}

    }

    labels = []

    action\_index = 0

    #goto\_index = 0

    with open(filename, 'rt', encoding='UTF8') as file:

        csvFile = csv.reader(file)

        for i, line in enumerate(csvFile):

            if i == 0:

                #goto\_index = len(line) - 1

                for j, entry in enumerate(line):

                    if j != 0:

                        if entry == "$":

                            action\_index = j

                        labels.append(entry)

            else:

                state = i - 1

                slr\_table['action'][state] = {}

                slr\_table['goto'][state] = {}

                for j, entry in enumerate(line):

                    if j != 0 and entry != "":

                        if j <= action\_index:

                            slr\_table['action'][state][labels[j-1]] = entry

                        else:

                            slr\_table['goto'][state][labels[j-1]] = int(entry)

        return slr\_table

The logical flow of the generation process is as follows:

1. It creates a dictionary named slr\_table that is to contain two separate dictionaries identified by the keys ‘action’ , and ‘goto’.
2. It opens the .csv file and reads each row. The first row contains the labels of the columns, which means the terminals and non-terminals. The first row is read and the distinct labels are contained in a list called ‘labels’. The index of the end marker within the first row is also saved in a variable called ‘action\_index’, to distinguish the later to-be-read table entries as either ‘action’ or ‘goto’.
3. Then it reads the rest of the rows and saves their content in the slr\_table dictionary. If the table entry is empty, it moves on to the next entry in the row. If the index of the entry is less or equal to ‘action\_index’, then it is saved within the ‘action’ dictionary, otherwise it is saved in the ‘goto’ dictionary, the key being the row number, or the state number.
4. The complete SLR table-dictionary is then returned.

The generated token list and SLR table-dictionary is passed into a parsing function defined below.

def parse\_with\_error\_reporting(tokens,slr\_table):

    stack = [(0, ParseTreeNode("CODE"))]  # Stack contains state and parse tree node

    index = 0  # Token index

    error\_message = None

    while stack:

        print()

        state, tree\_node = stack[-1]

        print("state: ", state)

        token = tokens[index] if index < len(tokens) else '$'

        print("token: ", token)

        print("slr\_table['action'][state] : ", slr\_table['action'][state])

        if token not in slr\_table['action'][state]:

            # Syntax error detected

            # if the table entry for the token and current state read does not exist,

            # then that means there exists a syntax error within the input token sequence.

            expected\_tokens = list(slr\_table['action'][state].keys())

            error\_message = f"Syntax error at token '{token}'. Expected one of: {', '.join(expected\_tokens)}"

            break

        action = slr\_table['action'][state][token]

        print("action: ", action)

        if action[0] == 's':  # Shift

            next\_state = int(action[1:])

            new\_tree\_node = ParseTreeNode(token)

            stack.append((next\_state, new\_tree\_node))

            index += 1

            print\_stack(stack)

        elif action[0] == 'r':  # Reduce

            production = new\_grammar[int(action[1:])]

            print("production: ", production)

            lhs, rhs = production

            new\_tree\_node = ParseTreeNode(lhs)

            for \_ in rhs:

                if \_ != "":

                    \_, child\_node = stack.pop()

                    new\_tree\_node.children.insert(0, child\_node)

            next\_state = slr\_table['goto'][stack[-1][0]][lhs]

            stack.append((next\_state, new\_tree\_node))

            print\_stack(stack)

        elif action == 'acc':  # Accept action

            print("Input accepted.")

            return tree\_node, None

        else:

            error\_message = f"Unexpected action '{action}'"

            break

    # Error occurred, construct error report

    error\_report = {

        'message': error\_message,

        'token\_position': index,

        'token': token,

        'expected\_tokens': expected\_tokens if expected\_tokens else [],

        'context': tokens[max(0, index - 5):index + 5]

    }

    return None, error\_report

4.Testing syntax analyzer

1. Execution through terminal
2. Example execution with test input files