compiler

spring, 2024



**term project**

20202586 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 terminal as follows:

in linux (from the submission directory):

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

The logical flow of the parsing and error reporting process is as follows:

1. First, declare variables to store the stack containing states and parse tree nodes, as well as variables for the token index and error message.
2. While there are values remaining in the stack, store the topmost value of the stack into the variables state and tree\_node. The tree\_node variable will become the root node of the parse tree if parsing is successful. Store the input text token into the token variable, and if the input text has been fully read, save '$' as the token value.
3. If there is no action corresponding to the current token in the action table's state, detect a syntax error. Save the expected\_tokens along with the error token and an error message, then terminate parsing.
4. Save the action indicated by the current stack state and token from the action table.
   1. If the action begins with 's', consider it as a shift. Receive the number following 's' as the next\_state. Then, create a new\_tree\_node by putting the current token into the ParseTreeNode class and push next\_state and new\_tree\_node onto the stack.
   2. If the action begins with 'r', consider it as a reduce. Save the grammar indicated by the number following 'r' as the production. Divide the lhs and rhs of the production, create a new\_tree\_node with lhs. Pop tokens and states from the stack as many times as the number of tokens in the rhs. Additionally, insert the popped child\_node as a child of new\_tree\_node. Put the value found in the goto table of the slr\_table under the row of the top state number on the stack and the column of lhs as next\_state.
   3. If the current action is 'acc', consider it as 'accept' and return tree\_node.
   4. If the action does not fall into the above cases, put a value into error\_message and terminate parsing.
5. When an error occurs during parsing, terminate parsing and return an error\_report. The error\_report includes the error\_message indicating the location of the problem, the index and token where the error occurred, and the expected\_tokens that should have occurred if parsing had been successful, along with the context surrounding the error token.

Below is the main code that executes :

if \_\_name\_\_ == "\_\_main\_\_":

    if len(sys.argv) != 2:

        print("Usage: python read\_file.py <filename>")

        sys.exit(1)

    token\_filename = sys.argv[1]

    tokens = read\_input\_file(token\_filename)

    slr\_parsing\_table = create\_parsing\_table("parsing\_table.csv")

    parse\_tree\_root, error\_report = parse\_with\_error\_reporting(tokens,slr\_parsing\_table)

    if error\_report:

        print("\nError occurred -----------------------",end='\n\n')

        msg\_length = len(error\_report["message"])

        print("Parsing error " + "="\*msg\_length)

        print(error\_report['message'])

        print("=" \* (14+msg\_length))

        print("Token Read:", error\_report['token'])

        print("Expected tokens:", error\_report['expected\_tokens'])

        print("Error occured at:", error\_report['context'])

    else:

        print("")

        print("-------------------------- Parse Tree ---------------------------------")

        print("    CODE'")

        print\_parse\_tree(parse\_tree\_root,is\_last\_child=True)

The program is executed by receiving two arguments in the terminal in the form of "python syntax\_analyzer.py test.txt". If there are not exactly 2 arguments, it prints an error message and exits the program. It stores the second argument as the input\_text to be parsed in the token\_filename variable.

It passes this file to the read\_input\_file function to obtain tokens, and creates the slr\_parsing\_table using the parsing\_table.csv file. It then passes the tokens and the corresponding slr\_parsing\_table to the parse\_with\_error\_reporting function, receiving parse\_tree\_root and error\_report as return values.

If an error\_report exists, it utilizes the error\_report object to print detailed error information. If parsing is successful without an error\_report, it passes parse\_tree\_root to the print\_parse\_tree function to print the parse tree.

4. Testing syntax analyzer

The execution of the syntax analyzer through the terminal was shown before, in the terminal just type:

python syntax\_analyzer.py <input\_filename>

the input file format we used for this project was .txt file, and the python version on which the program was tested was python 3.12.1.

The syntax analyzer was tested on a variety of test input files – their input and output are as shown in the next page.

“test.txt” content:

vtype id semi

vtype id lparen rparen lbrace

if lparen boolstr comp boolstr rparen lbrace rbrace

return num semi

rbrace

Although the file is not visually structured like it is shown above (though it can be), the sematic structure is the same. In the actual file, the tokens are simply spaced with a single whitespace.

텍스트, 소프트웨어, 멀티미디어 소프트웨어, 스크린샷이(가) 표시된 사진

자동 생성된 설명[test.txt input]

\*\* While in the example above the program was run in windows, we have tested the program execution in a virtual linux environment, and the output was identical. \*\*s

텍스트, 도표, 스크린샷이(가) 표시된 사진

자동 생성된 설명텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명[Syntax\_analyzer execution output – parsing log and parse tree]

In the picture on the left above, the parsing log of the input token sequence is printed. The current state, the token that was read, the action taken, and the stack content is printed. If the action taken was reduction, then the production rule that was used is also printed.

after the parsing log is done printing, the constructed parse tree is printed as shown above on the right.

The “test.txt” input file examines the grammar regarding variable declaration, function declaration, and using if statements without else.

Here is another example that thoroughly examines all the grammar in the CFG.

“test2.txt” content

vtype id semi

vtype id assign id semi

vtype id assign num semi

vtype id lparen vtype id comma vtype id rparen lbrace

if lparen boolstr rparen lbrace

id assign num semi

rbrace else lbrace

id assign literal semi

rbrace

while lparen boolstr comp boolstr rparen lbrace

id assign id addsub num semi

rbrace

return id semi

rbrace

the test file above was made to check all the grammar; It includes variable declaration, variable assignments, a function declaration with multiple arguments, if statement with else, and a while statement.

The output parse tree is shown in the next page.

텍스트, 스크린샷, 도표이(가) 표시된 사진

자동 생성된 설명텍스트, 스크린샷, 도표이(가) 표시된 사진

자동 생성된 설명[Syntax\_analyzer execution output – parse tree]