東南大學
SOUTHEAST UNIVERSITY

Compiler Principles Course Lab Report

Student ID：09022107

Name：梁耀欣

School of Computer Science and Engineering,

Southeast University

2024.12

Contents

LR(1) Grammar Parser

1 Motivation/Aim

The aim of this project is to deepen the understanding of the syntax analysis process and implement a syntax analyzer for LR(1) grammars.

2 Content Description

2.1 Non-Programming Implementation

1. **Define Context-Free Grammar**
2. **Draw DFA**: For each state in the DFA, the closure of LR(1) items is constructed, and a tree is built to achieve this. (A photo of the DFA and a related table are provided.)
3. **Construct Action and Goto Tables**: Based on the DFA, the Action table and Goto table are drawn.

2.2 Programming Implementation

Programming is done according to the productions, Action table, and Goto table.

3 Ideas/Methods

The LR(1) grammar uses a bottom-up analysis method. The syntax analyzer scans the expression character by character from left to right, puts the symbols into the symbol stack, and performs shift or reduce operations according to the Action table and Goto table. At the same time, the symbol stack and state stack are modified, and the action description (shift or reduce, and the production used for reduction) is output. If there is no corresponding Action operation for the currently analyzed symbol and the top state of the state stack, an error is reported, and the input symbol string is considered illegal.

The main method is to iteratively analyze by reading the current state and the current symbol in the input string, and determine the action to be executed according to the action table.

- **Shift Action**: If the action table indicates a shift (action value > 0), the current symbol is pushed onto the symbol stack, the state corresponding to the action value is pushed onto the state stack, and then the next symbol in the input string is moved to.

- **Reduce Action**: If the action table indicates a reduce (action value < 0), the index of the grammar production to be applied is determined according to the action value. The same number of elements as the number of symbols on the right side of the production are popped from the symbol stack and the state stack. Then, according to the current state and the left side symbol of the production, the new state is obtained through the Goto table, the new state is pushed onto the state stack, and the left side symbol of the production is pushed onto the symbol stack.
- **Accept Action**: When the action value is 0, it indicates that the input string has been successfully analyzed and conforms to the grammar rules.
- **Error Handling**: If the action value is -1, it indicates that an unprocessable symbol has been encountered, and the input string does not conform to the grammar rules.

4 Assumptions

Using the tokens defined in project1, a context-free grammar is constructed as described as follows.

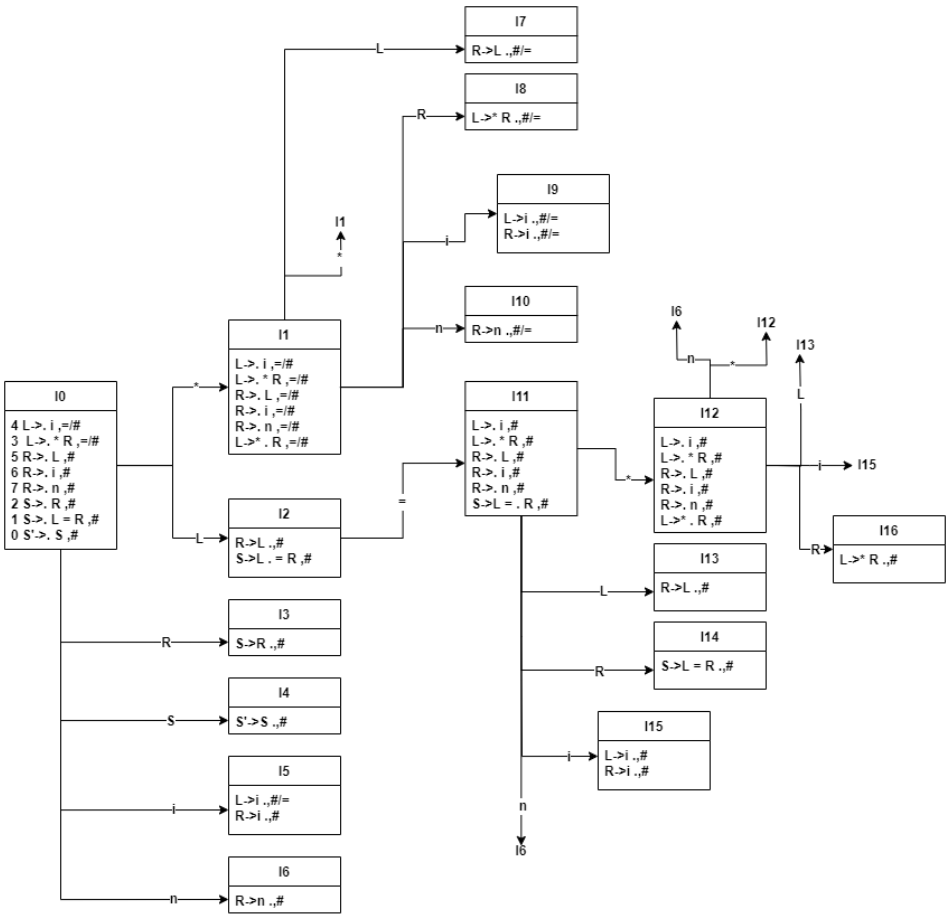
$$
\begin{aligned}
S' &\to S \\
S &\to L = R \\
S &\to R \\
L &\to *R \\
L &\to i \\
R &\to L \\
R &\to i \\
R &\to n
\end{aligned}
$$

For the sake of simplicity, the 'identifier' from project1 is defined as 'i', 'num' is defined as 'n', and the operators '=' and '*' are used to construct this grammar.

We number these rules as 0 to 7.

5 Related FA Descriptions

A state transition diagram DFA is drawn, and a related table is provided.



The table including the action table and goto table is provided:

状态	ACTION					GOTO		
	*	=	i	$	n	S	L	R
0	s1		s5		s6	4	2	3
1			s9		s10		7	8
2		s11		r5				
3				r2				
4				succ				
5				r4				
6				r7				
7		r5		r5				
8		r3		r3				
9		r4		r4				
10		r7		r7				
11	s12		s15		s6		13	14
12	s12		s15		s6		13	16
13				r5				
14				r1				
15				r6				
16				r3				

6 Description of Important Data Structures

6.1 Class `LRAnalyseTable`

This is the LR analysis table class.

```
class LRParseTable {
    private:
        // Productions
        string grammar[15] = { "T->S", "S->L=R", "S->R", "L->*R", "L->i
            ", "R->L", "R->i", "R->n" };
        // Terminals
        char terminalChars[10] = { '*', '=', 'i', '#', 'n' };
        // Non-terminals
        char nonTerminalChars[10] = { 'S', 'L', 'R' };
        // Number of terminals
        int numTerminals = 5;
        // Number of non-terminals
        int numNonTerminals = 3;

        // Action table
        int Action[20][5] = {
            {1, -1, 5, -1, 6}, {-1, -1, 9, -1, 10}, {-1, 11, -1, -15,
                -1}, {-1, -1, -1, -12, -1},
            {-1, -1, -1, 0, -1}, {-1, -1, -1, -14, -1}, {-1, -1, -1,
                -17, -1}, {-1, -15, -1, -15, -1},
            {-1, -13, -1, -13, -1}, {-1, -14, -1, -14, -1}, {-1, -17,
                -1, -17, -1}, {12, -1, 15, -1, 6},
            {12, -1, 15, -1, 6}, {-1, -1, -1, -15, -1}, {-1, -1, -1,
                -11, -1}, {-1, -1, -1, -16, -1}, {-1, -1, -1, -13, -1}
        };
        // Goto table
        int Goto[20][4] = {
            {4, 2, 3}, {-1, 7, 8}, {-1, -1, -1}, {-1, -1, -1}, {-1, -1,
                -1},
            {-1, -1, -1}, {-1, -1, -1}, {-1, -1, -1}, {-1, -1, -1},
            {-1, -1, -1}, {-1, -1, -1}, {-1, 13, 14}, {-1, 13, 16},
            {-1, -1, -1}, {-1, -1, -1}, {-1, -1, -1}, {-1, -1, -1}
        };

    public:
        LRParseTable() {}
        ~LRParseTable() {}
        int getTerminalIndex(char ch);
        int getNonTerminalIndex(char ch);
        int getAction(int state, char ch);
        int getGoto(int state, char ch);
        string getGrammar(int idx);
    };
```

- **Member Variables**: - `grammar`: An array of strings storing all productions. - `terminalChars`: An array of characters storing terminals. - `nonTerminalChars`: An array of characters storing non-terminals. - `numTerminals`: The number of terminals. - `numNonTerminals`: The number of non-terminals. - `Action`: A 2D array representing the Action table. - `Goto`: A 2D array representing the Goto table. The Action table and Goto table are initialized as follows. -1 indicates no

corresponding operation, i.e., the corresponding cell in the table is empty. Numbers less than -10 represent reductions. The production number used for reduction can be obtained by taking the opposite of the number plus 10 (the production number ranges from 0 to 7).

- **Member Functions**: - `getTerminalIndex(char ch)`: Gets the index of the terminal character in the terminal array. - `getNonTerminalIndex(char ch)`: Gets the index of the non-terminal character in the non-terminal array. - `getAction(int state, char ch)`: Queries the Action table based on the top state of the stack and the current symbol to get the corresponding action. - `getGoto(int state, char ch)`: Queries the Goto table based on the top state of the stack and the current symbol to get the corresponding action (used after reduction). - `getGrammar(int idx)`: Looks up the production array to get the corresponding production according to the production number used for reduction.

6.2 Class `Analyser`

This is the syntax analyzer class.

```
class Parser {
    private:
        string inputString; // String to be parsed
        int step;           // Step count
        int currentIndex;   // Current index of the input string
        vector<char> symbolStack; // Symbol stack
        vector<int> stateStack;   // State stack
    public:
        Parser();
        ~Parser();
        bool startParsing(string input);
        string stackToString(int option);
    };
    Parser::Parser() {
        this->step = 1;
        this->currentIndex = 0;
        symbolStack.push_back('#');
        stateStack.push_back(0);
    }
```

- **Member Variables**: - `inputString`: The string to be parsed. - `step`: The count of the current step. - `currentIndex`: The index of the current symbol in the input string. - `symbolStack`: A vector storing the symbol stack. - `stateStack`: A vector storing the state stack.

- **Member Functions**: - `Parser()`: Initializes the syntax analyzer. The step is initialized to 1, and the index of the current analyzed symbol is initialized to 0. - `stackToString(int option)`: Returns all the states or characters in the state stack or symbol stack for output display. This function selects whether to return the content in the state stack or the symbol stack according to the input parameter. - `startParsing(string input)`: The core analysis function. It determines whether the current character is legal. If it is legal, it performs a shift or reduce operation and modifies the symbol stack and state stack. After a reduction operation, the

Goto table is queried to modify the state stack. The specific algorithm is described in the core algorithm section.

7 Description of Core Algorithms

The LR core algorithm is implemented in the `startParsing` function.

```
bool Parser::startParsing(string input) {
    this->inputString = input;
    LRParseTable table;
    int currentState = 0;

    cout << setw(10) << "Step" << setw(15) << "State Stack" << setw(10)
            << "Symbol Stack" << setw(15) << "Current Symbol" << setw
                (15)
            << "Remaining String" << setw(20) << "Action" << endl;

    while (true) {
        int action = table.getAction(currentState, inputString[
            currentIndex]);
        if (action == 0) {
            cout << setw(10) << step << setw(15) << stackToString(0) <<
                setw(10)
                    << stackToString(1) << setw(15) << inputString[
                        currentIndex]
                    << setw(15) << inputString.substr(currentIndex) <<
                        setw(20)
                    << "Parsing complete" << endl;
            return true;
        }
        if (action == -1) {
            cerr << "Error: Unexpected symbol '" << inputString[
                currentIndex]
                    << "' at index " << currentIndex << endl;
            return false;
        }
        if (action > 0) {
            cout << setw(10) << step << setw(15) << stackToString(0) <<
                setw(10)
                    << stackToString(1) << setw(15) << inputString[
                        currentIndex]
                    << setw(15) << inputString.substr(currentIndex) <<
                        setw(20)
                    << "Action[S" << currentState << "][" <<
                        inputString[currentIndex]
                    << "]=S" << action << " (Shift)" << endl;
            stateStack.push_back(action);
            symbolStack.push_back(inputString[currentIndex]);
            currentIndex++;
        }
        else {
            int ruleIdx = -(action + 10);
            string rule = table.getGrammar(ruleIdx);
            if (rule.empty()) return false;

            cout << setw(10) << step << setw(15) << stackToString(0) <<
                setw(10)
```

```
                    << stackToString(1) << setw(15) << inputString[
                        currentIndex]
                    << setw(15) << inputString.substr(currentIndex) <<
                        setw(20)
                    << "R" << ruleIdx + 1 << ": " << rule << " (Reduce)
                        " << endl;

            for (size_t i = 0; i < rule.size() - 3; i++) {
                if (stateStack.empty() || symbolStack.empty()) {
                    cerr << "Error: Stack underflow during reduction."
                        << endl;
                    return false;
                }
                stateStack.pop_back();
                symbolStack.pop_back();
            }
            currentState = stateStack.back();
            int gotoResult = table.getGoto(currentState, rule[0]);
            if (gotoResult == -1) return false;
            cout << setw(10) << ++step << setw(15) << stackToString(0)
                << setw(10)
                    << stackToString(1) << setw(15) << rule[0] << setw
                        (15)
                    << inputString.substr(currentIndex) << setw(20)
                    << "Goto[S" << currentState << "][" << rule[0] << "
                        ]=" << gotoResult
                    << " (State Transition)" << endl;
            stateStack.push_back(gotoResult);
            symbolStack.push_back(rule[0]);
        }
        currentState = stateStack.back();
        step++;
    }
}
```

First, initialization is performed. The input string is stored in `inputString`, the step number `step` is set to 1, the current index `currentIndex` is set to 0, '#' is pushed onto the symbol stack `symbolStack`, and the initial state 0 is pushed onto the state stack `stateStack`. Then, it enters the loop analysis stage. The action `action` corresponding to the current state `currentState` and the current position `inputString[currentIndex]` of the input string is obtained, and different types of action judgments are made according to the value of `action`: If `action == 0`, it indicates that the analysis is successful, the analysis completion information is output, and `true` is returned; if `action == -1`, it means that an unexpected symbol is encountered, an error message is output, and `false` is returned; if `action > 0`, a shift operation is performed, that is, the state `action` is pushed onto the state stack, the current symbol `inputString[currentIndex]` is pushed onto the symbol stack, and then `currentIndex` is incremented by 1; if `action < 0`, a reduce operation is performed. First, the index `ruleIdx` of the grammar production to be applied is calculated according to `action`. Then, the same number of elements as the number of symbols on the right side of the production are popped from the symbol stack and the state stack. After that, the current state `currentState` is obtained, and the transition state `gotoResult` after applying the production is calculated. Finally, `gotoResult` is pushed onto the state stack, and the left side symbol of the production is pushed onto the symbol stack. After completing an action judgment, the current state is

updated to the top element of the state stack, and the step number `step` is incremented. This loop continues until the analysis is completed.

The `main` function is as follows:

```
int main() {
    FILE* in;
    FILE* out;
    if (freopen_s(&in, "test.txt", "r", stdin)!= 0) {
        cerr << "Error: Unable to open test.txt for reading." << endl;
        return 1;
    }
    if (freopen_s(&out, "result.txt", "w", stdout)!= 0) {
        cerr << "Error: Unable to open result.txt for writing." << endl
            ;
        return 1;
    }

    string str;
    int caseCount = 1;
    while (getline(cin, str)) {
        cout << "Case " << caseCount++ << ": " << str << endl;
        Parser parser;
        if (parser.startParsing(str)) {
            cout << "The input string '" << str << "' is valid." <<
                endl;
        }
        else {
            cout << "The input string '" << str << "' is invalid." <<
                endl;
        }
        cout << endl;
    }
    return 0;
}
```

In the `main` function, the program reads the input string from the `test.txt` file, performs LR analysis on each input string, and outputs the result to the `result.txt` file.

8 Use Cases on Running

The test cases are in the `test.txt` file:

```
i=i#
*i=i#
*i#
i#
=i#
ii#
```

The terminal output results are as follows:

```
Error: Unexpected symbol '=' at index 1
Error: Unexpected symbol '=' at index 0
Error: Unexpected symbol 'i' at index 1
```

The output results in the `result.txt` file:

```
Case 1: i=i#
    Step    State StackSymbol Stack Current SymbolRemaining String          Action
       1        0        #          i        i=i#          Action[S0][i]=S5 (Shift)
The input string 'i=i#' is invalid.

Case 2: *i=i#
    Step    State StackSymbol Stack Current SymbolRemaining String          Action
       1        0        #          *        *i=i#          Action[S0][*]=S1 (Shift)
       2      0_1       #*          i         i=i#          Action[S1][i]=S9 (Shift)
       3    0_1_9       #*i         =          =i#             R5: L->i (Reduce)
       4      0_1       #*          L          =i#          Goto[S1][L]=7 (State Transition)
       5    0_1_7       #*L         =          =i#             R6: R->L (Reduce)
       6      0_1       #*          R          =i#          Goto[S1][R]=8 (State Transition)
       7    0_1_8       #*R         =          =i#             R4: L->*R (Reduce)
       8        0        #          L          =i#          Goto[S0][L]=2 (State Transition)
       9      0_2       #L          =          =i#          Action[S2][=]=S11 (Shift)
      10   0_2_11      #L=          i          i#          Action[S11][i]=S15 (Shift)
      11 0_2_11_15    #L=i          #          #             R7: R->i (Reduce)
      12   0_2_11      #L=          R          #          Goto[S11][R]=14 (State Transition)
      13 0_2_11_14    #L=R          #          #             R2: S->L=R (Reduce)
      14        0        #          S          #          Goto[S0][S]=4 (State Transition)
      15      0_4       #S          #          #    Parsing complete
The input string '*i=i#' is valid.

Case 3: *i#
    Step    State StackSymbol Stack Current SymbolRemaining String          Action
       1        0        #          *        *i#          Action[S0][*]=S1 (Shift)
       2      0_1       #*          i         i#          Action[S1][i]=S9 (Shift)
       3    0_1_9       #*i         #          #             R5: L->i (Reduce)
       4      0_1       #*          L          #          Goto[S1][L]=7 (State Transition)
       5    0_1_7       #*L         #          #             R6: R->L (Reduce)
       6      0_1       #*          R          #          Goto[S1][R]=8 (State Transition)
       7    0_1_8       #*R         #          #             R4: L->*R (Reduce)
       8        0        #          L          #          Goto[S0][L]=2 (State Transition)
       9      0_2       #L          #          #             R6: R->L (Reduce)
      10        0        #          R          #          Goto[S0][R]=3 (State Transition)
      11      0_3       #R          #          #             R3: S->R (Reduce)
      12        0        #          S          #          Goto[S0][S]=4 (State Transition)
      13      0_4       #S          #          #    Parsing complete
The input string '*i#' is valid.

Case 4: i#
    Step    State StackSymbol Stack Current SymbolRemaining String          Action
       1        0        #          i         i#          Action[S0][i]=S5 (Shift)
       2      0_5       #i          #          #             R5: L->i (Reduce)
       3        0        #          L          #          Goto[S0][L]=2 (State Transition)
       4      0_2       #L          #          #             R6: R->L (Reduce)
       5        0        #          R          #          Goto[S0][R]=3 (State Transition)
       6      0_3       #R          #          #             R3: S->R (Reduce)
       7        0        #          S          #          Goto[S0][S]=4 (State Transition)
       8      0_4       #S          #          #    Parsing complete
The input string 'i#' is valid.

Case 5: =i#
    Step    State StackSymbol Stack Current SymbolRemaining String          Action
The input string '=i#' is invalid.

Case 6: ii#
    Step    State StackSymbol Stack Current SymbolRemaining String          Action
       1        0        #          i         ii#          Action[S0][i]=S5 (Shift)
The input string 'ii#' is invalid.
```

9 Problems Occurred and Related Solutions

Initially, there were errors in the construction of the action table and Goto table, resulting in incorrect program running results and the parser entering an infinite loop. Later, this problem was manually solved, and the program was able to run accurately.

In the ACTION table, there are both shift and reduce operations. To distinguish these two operations, reduce operations are represented by negative numbers less than -10. The specific conversion method is: take the opposite of the production number used for reduction plus 10.

The number of elements popped from the state stack and the symbol stack was incorrect, resulting in an attempt to access an empty stack. I modified the code responsible for the reduce operation so that it can correctly calculate the number of symbols to be popped according to the length of the production rule.

10 Feelings and Comments

1. Through this experiment, I have a deeper understanding of LR(1) analysis. Since it is a bottom-up analysis method, the characters to be analyzed can be scanned and analyzed at the same time. Before constructing an LR(1) syntax analyzer, some preparatory operations are required. First, a DFA needs to be drawn. For LR(1) analysis, one more symbol needs to be looked ahead. Therefore, LR(1) items not only include productions but also need to carry lookahead symbols. Even if the productions are the same, different lookahead symbols represent different states. Then, the ACTION and GOTO tables need to be drawn based on the DFA and used to write the program.

2. After reduction, the GOTO table needs to be queried to update the top of the state stack according to the updated symbol stack and the symbol and state at the top of the state stack. This is a detail that I did not notice in previous studies. At the same time, it also reflects the role of the GOTO table. Before the experiment, I was not very clear why there are two tables, ACTION and GOTO. After this experiment, I understand it more deeply.