Compiler Principle
Course Projects Report

December 24, 2024

Table of Contents

1 Project 1 XL-Scanner

1.1 Motivation/Aim

The aim of this project is to design a scanner for a simple programming language. The scanner should be able to recognize and classify tokens in the input string based on the defined rules.

1.2 Content description

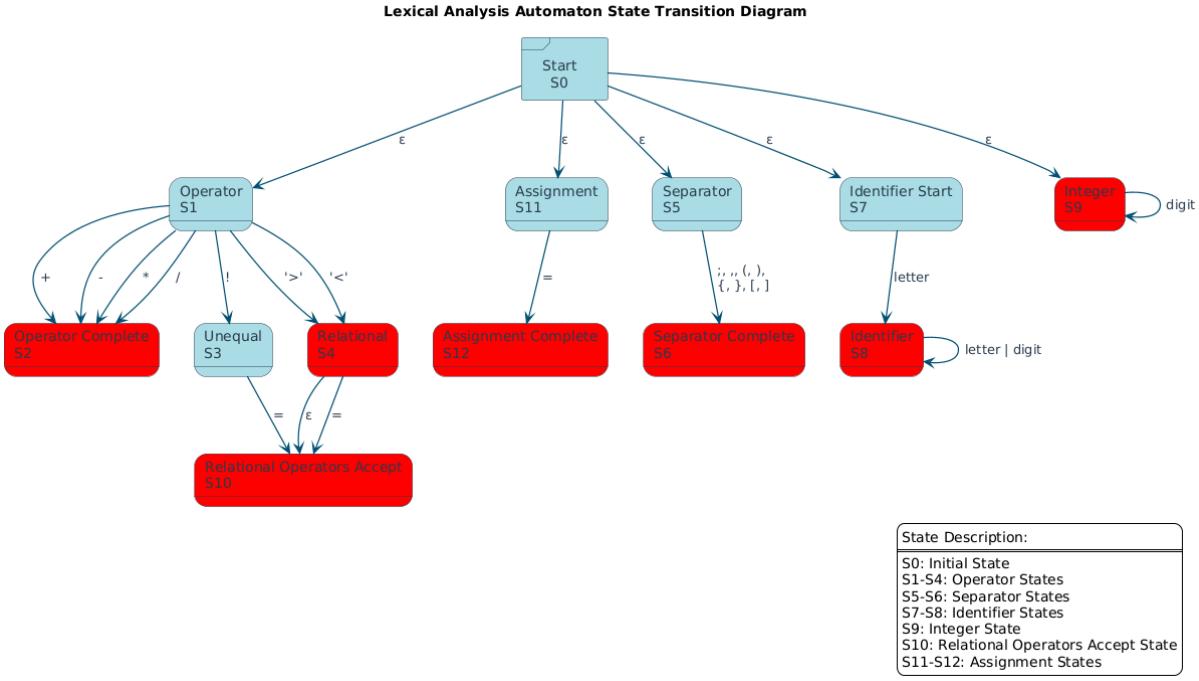
1. Define REs by yourself for the common tokens used in our programming languages.
2. Convert REs into NFAs.
3. Merge these NFAs into a single NFA.
4. Convert the NFA into a DFA with minimum states.
5. Develop a scanner based on the DFA.

1.3 Ideas/Methods

1.3.1 Define REs by yourself for the common tokens used in our programming languages

- Algebra Operators: `+`, `-`, `*`, `/`, `=`
- Relational Operators: `!=`, `>`, `<`, `>=`, `<=`
- Assignment Operators: `=`
- Separators: `;`, `,`, `(`, `)`, `{`, `}`, `[`, `]`
- Identifiers: `letter (letter | digit)*`
- Integers: `digit+`

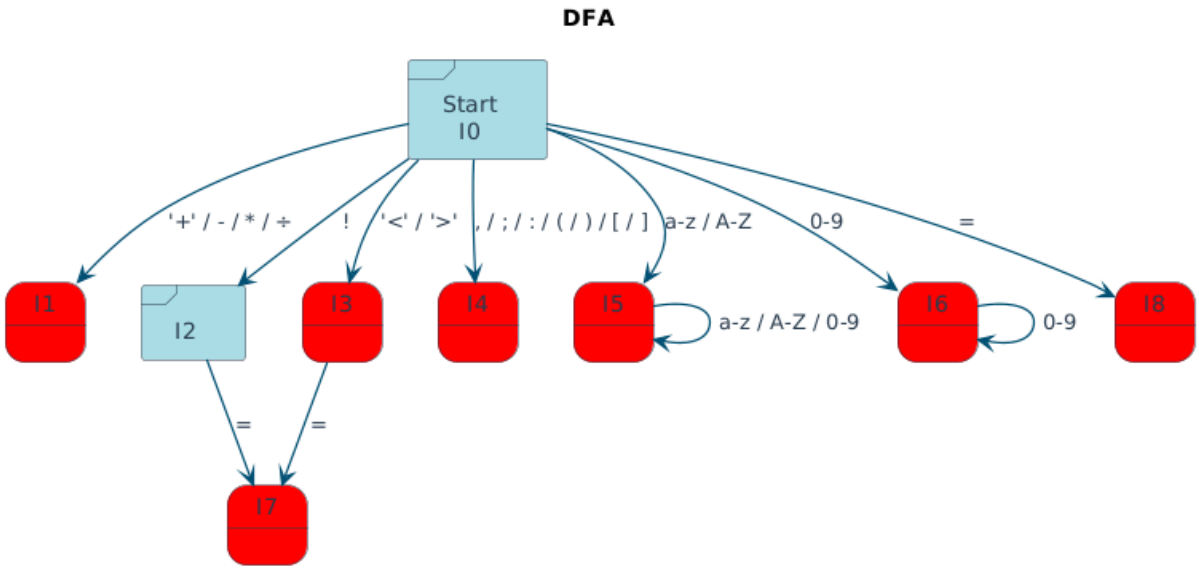
1.3.2 Convert REs into NFAS and Merge these NFAs into a single NFA



Where red represents the accept state. From the figure, the initial state transitions to the initial state of each sub-NFA through ε, and the accept states of each sub-NFA transition to the final accept state through ε.

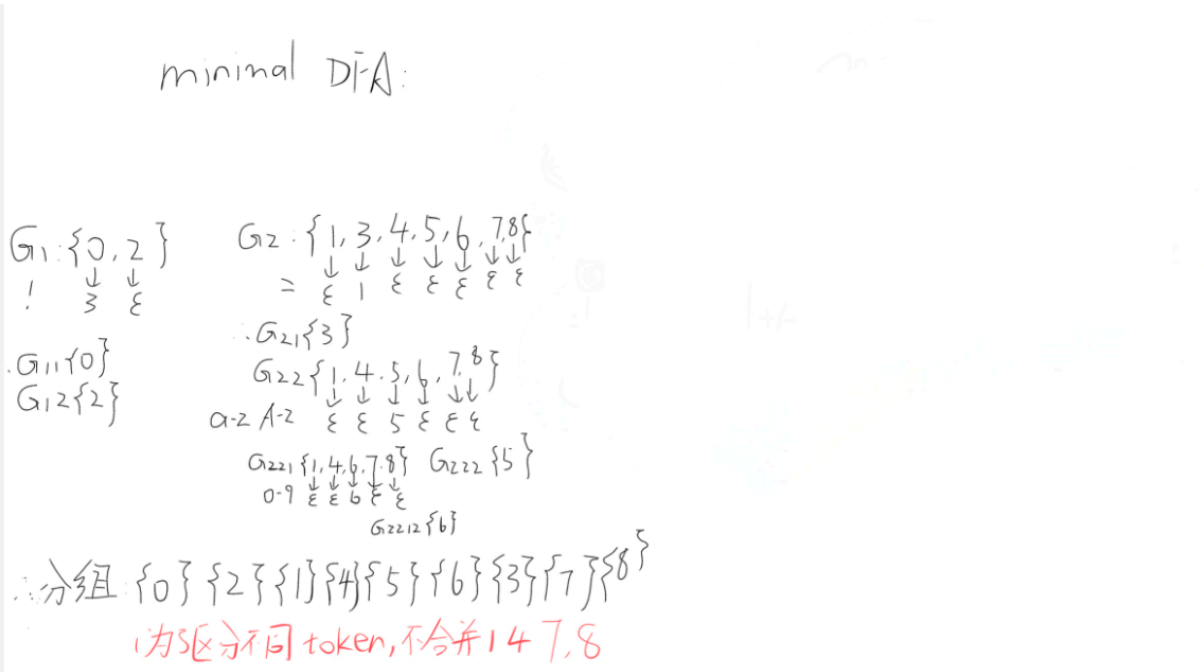
1.3.3 Convert the NFA into a DFA

transitions on ε closure	+ / - / * / ÷	!	< / >	=	; / , / (/)	a-z/A-Z	0-9
I0 {0,1,5,7,9,11}	{2} I1	{3} I2	{4,10} I3	{12} I8	{6} I4	{8} I5	{9} I6
I2				{10} I7			
I3				I7			
I5						I5	I5
I6							I6

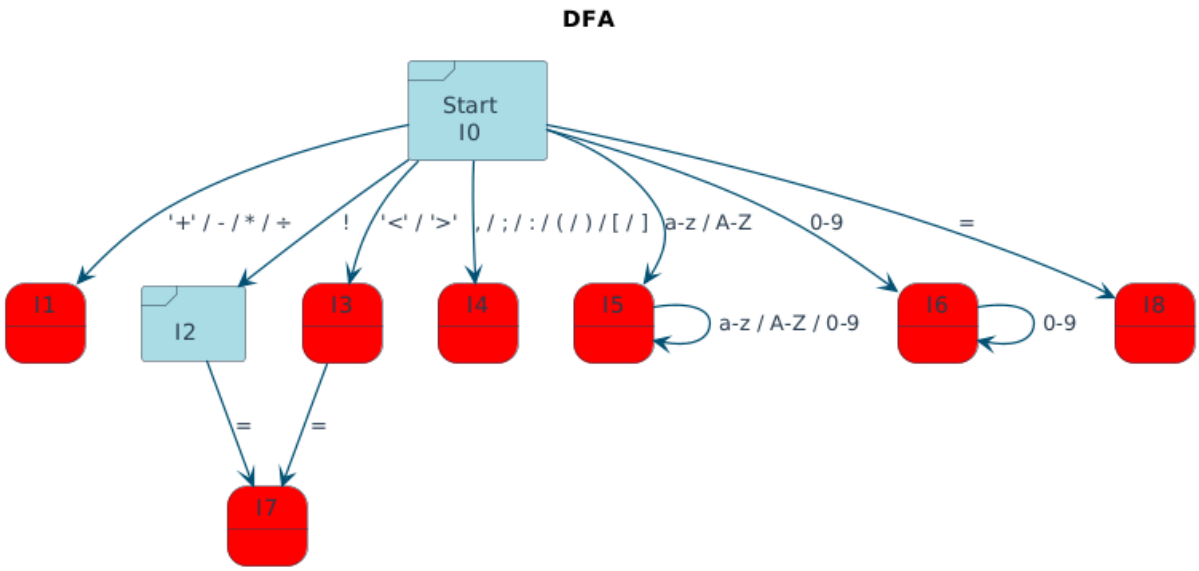


Accoring to the epsilon closure and the transition table, the state transition relationship of the DFA is obtained.

1.3.4 Minimize the DFA



So the minimal DFA is obtained as follows



The state transition relationship of the minimal DFA is shown as manually deduced in the above figure.

1.4 Develop a scanner based on the DFA.

1.4.1 Scanner Initialization

```
Algorithm 1: Scanner Initialization
Input: A minimal DFA
Output: Initialized DFA scanner

1: procedure Initialize()
2:     transitions ← empty dictionary
3:     accept_states ← {1,3,4,5,6,7,8}
4:
5:     // Initialize digit transitions (0-9)
6:     for each d in Digits do
7:         transitions[0][d] ← 1
8:         transitions[1][d] ← 1
9:     end for

10:    // Initialize identifier transitions (a-z,A-Z,_)
11:    for each c in Letters  do
12:        transitions[0][c] ← 4
13:        transitions[4][c] ← 4
14:    end for

15:    // Initialize operator transitions
16:    for each op in {<, >, !} do
17:        transitions[0][op] ← 2
18:    end for

19:    // Initialize special operators
20:    for each sp in {=,+,-,*,/} do
21:        transitions[0][sp] ← 3
22:    end for

23:    return transitions, accept_states
```

For the initialization of the scanner, according to the state transition table of the DFA, the transition relationships of digits, identifiers, operators, and special operators are initialized, and a two-dimensional dictionary transitions and a set of accept states accept_states are initialized.

The reason for using a two-dimensional dictionary instead of if-else statements is that the query efficiency of a two-dimensional dictionary is higher and easier to maintain. It is also easier for the next step of state transition.

1.4.2 Scan Algorithm

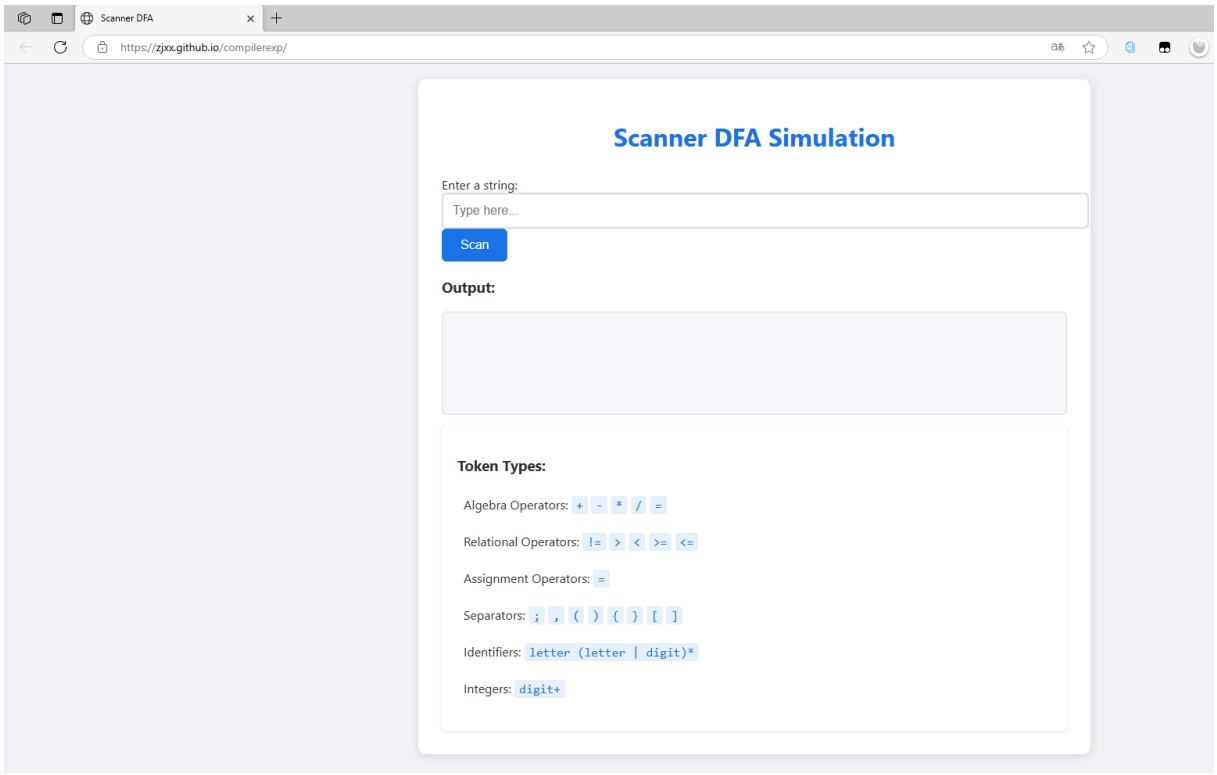
```
Algorithm 2: Token Scanning
Input: input_str
Output: List of tokens with their types

1: procedure Scan(input_str)
2:     tokens ← empty list
3:     state ← 0
4:     token ← empty string

5:     for each c in input_str do
6:         if c exists in transitions[state] then
7:             state ← transitions[state][c]
8:             token ← token + c
9:         else
10:            if state in accept_states then
11:                tokens.append((token, GetTokenType(state)))
12:                token ← empty string
13:                state ← 0
14:                // Try processing current char from initial state
15:                if c exists in transitions[0] then
16:                    state ← transitions[0][c]
17:                    token ← token + c
18:                end if
19:            else
20:                return "Invalid token"
21:            end if
22:        end if
23:    end for

24:    return tokens
```

Scan method takes a string as input and returns a token list. The longest match principle is adopted, that is, after reading a character each time, if the current state cannot be converted to the next state, the current token is added to the token list, the current token is cleared, and the current state is reset to 0. Each element in the token list is a tuple containing the value of the token and the type of the token. If the input string contains illegal characters, “Invalid token” is returned.



And this code has been ported and embedded into a simple web page, which can be accessed through (https://zjxx.github.io/compilerexp/) for token scanning.

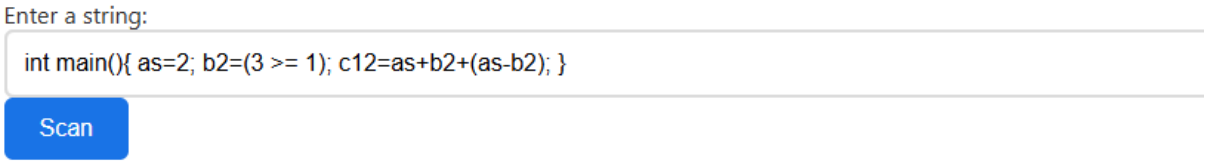
1.4.3 Description of important Data Structures

- `transitions`: DFA transition table, used to store state transition relationships.
- `accept_states`: Set of accept states, used to determine whether the current state is an accept state.
- `tokens`: Used to store recognized tokens and their types.
- `state`: Current state, used to record the state of the DFA.

1.4.4 Description of core Algorithms

- `scan`: Scan the input string, perform state transitions based on the DFA's transition table, recognize tokens, and return a token list.After reading a character each time, if the current state cannot be converted to the next state, the current token is added to the token list, the current token is cleared, and the current state is reset to 0, which is the longest match principle.
- `get_token_type`: Returns the type of the token based on the state of the DFA, a simple mapping function.

1.4.5 Test the scanner with the following input strings



Output:

```
Token: int   Type: Identifiers
Token: main   Type: Identifiers
Token: (   Type: Separators
Token: )   Type: Separators
Token: {   Type: Separators
Token: as   Type: Identifiers
Token: =   Type: Assignment Operator
Token: 2   Type: Integers
Token: ;   Type: Separators
Token: b2   Type: Identifiers
Token: =   Type: Assignment Operator
Token: (   Type: Separators
Token: 3   Type: Integers
Token: >=   Type: Relational Operators
Token: 1   Type: Integers
Token: )   Type: Separators
Token: ;   Type: Separators
Token: c12   Type: Identifiers
Token: =   Type: Assignment Operator
Token: as   Type: Identifiers
Token: +   Type: Algebra Operators
Token: b2   Type: Identifiers
Token: +   Type: Algebra Operators
Token: (   Type: Separators
Token: as   Type: Identifiers
Token: -   Type: Algebra Operators
Token: b2   Type: Identifiers
Token: )   Type: Separators
Token: ;   Type: Separators
Token: }   Type: Separators
```

It can be seen that the scanner can correctly identify the tokens in the input string and return the correct token type.

1.5 Problems occurred and related solutions

1. When converting states to minDFA, states need to be merged, and for accept states, there may be multiple accept states that need to be merged, making it impossible to distinguish which token it is.

Solution: Cancel the merging of accept states and maintain the uniqueness of accept states. (Solution obtained from consulting the teacher after class)

1.6 Your feelings and comments

This experiment has given me a deeper understanding of compiler theory, especially the conversion of DFA and NFA, and the minimization of DFA. By implementing a simple scanner, I have gained a deeper understanding of the lexical analysis part of the compiler. At the same time, by porting the code to a web page, I have also learned how to apply the code to practical projects, which is of great help to my future learning and work.

2 Project 2 LR(1) based Parser

2.1 Motivation/Aim

The aim of this project is to design an LR(1) parser for a simple programming language. The parser should be able to analyze the input token sequence based on the defined grammar and determine whether it conforms to the grammar rules.

2.2 Content description

1. Design a CFG that should describe the main sentence patterns in programming languages by using the tokens defined in the project 1.
2. Construct LR(1) parsing table for the CFG.
3. Implement the parser based on the LR(1) paring table.

2.3 Ideas/Methods

2.3.1 Design a CFG using the tokens defined in the project 1.

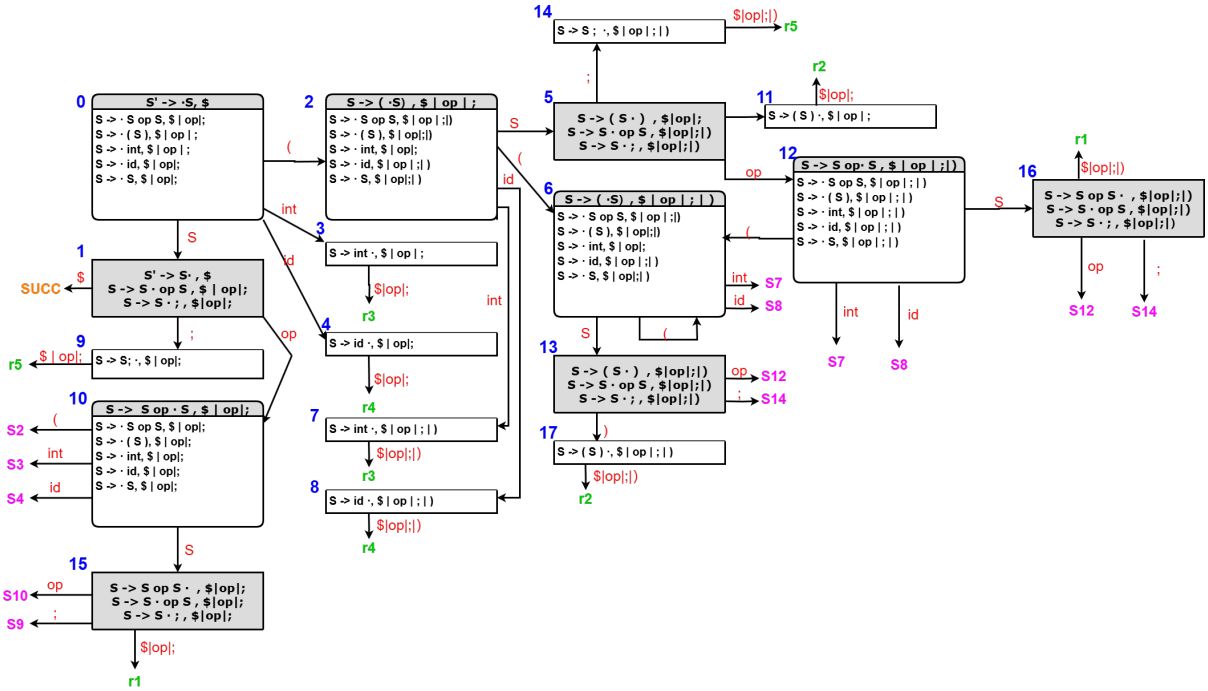
1. S -> S op S
2. S-> (S)
3. S -> int
4. S -> id
5. S -> S;

Where op represents arithmetic operators, id represents identifiers, and int represents integers.

This CFG describes some simple sentence patterns, including arithmetic expressions, bracket expressions, integers, identifiers, and assignment statements.

2.3.2 Develop a LR(1) based parser for the CFG

transition diagram:



Parsing Table:

	ACTION							GOTO
	op	(	)	int	id	$	;	S
0		s2		s3	s4			s1
1	s10					SUCC	s9	
2		s6		s7	s8			s5
3	r3					r3	r3	
4	r4					r4	r4	
5	s12		s11				s14	
6		s6		s7	s8			s13
7	r3		r3			r3	r3	
8	r4		r4			r4	r4	
9	r5					r5	r5	
10		s2		s3	s4			s15
11	r2					r2	r2	
12				s7	s8			s16
13	s12		s17				s14	
14	r5		r5			r5	r5	
15	r1					r1	s9	
16	r1		r1			r1	s14	
17	r2		r2			r2	r2	

2.3.3 Parser Initialization

```
class Parser:
    def __init__(self):
        self.action_table = {
            0: {'(': 's2', 'int': 's3', 'id': 's4'},
            1: {'$': 'SUCC', ';': 's9', 'op': 's10'},
            2: {'(': 's6', 'int': 's7', 'id': 's8'},
            3: {'op': 'r3', '$': 'r3', ';': 'r3'},
            4: {'op': 'r4', '$': 'r4', ';': 'r4'},
            5: {'op': 's12', ')': 's11', ';': 's14'},
            6: {'(': 's6', 'int': 's7', 'id': 's8'},
            7: {'op': 'r3', ')': 'r3', '$': 'r3', ';': 'r3'},
            8: {'op': 'r4', ')': 'r4', '$': 'r4', ';': 'r4'},
            9: {'op': 'r5', '$': 'r5', ';': 'r5'},
            10: {'(': 's2', 'int': 's3', 'id': 's4'},
            11: {'op': 'r2', '$': 'r2', ';': 'r2'},
            12: {'int': 's7', 'id': 's8'},
            13: {'op': 's12', ')': 's17', ';': 's14'},
            14: {'op': 'r5', ')': 'r5', '$': 'r5', ';': 'r5'},
            15: {'op': 'r1', '$': 'r1', ';': 's9'},
            16: {'op': 'r1', ')': 'r1', '$': 'r1', ';': 's14'},
            17: {'op': 'r2', ')': 'r2', '$': 'r2', ';': 'r2'}
        }
        self.goto_table = {
            0: {'S': 1},
            2: {'S': 5},
            6: {'S': 13},
            10: {'S': 15},
            12: {'S': 16},
            13: {'S': 14},
        }
        self.grammar = [
            ('S', 'S op S'),# 空格是为了区分两个字符)
```

```
            ('S', '( S )'),
            ('S', 'int'),
            ('S', 'id'),
            ('S', 'S ;')
        ]
```

The action table, goto table, and grammar rules are hard-coded directly to facilitate subsequent analysis.

2.3.4 Parse Algorithm

```
Algorithm: LR_PARSER
Input: tokens - token stream
Output: parsing result or error

procedure Parse(tokens):
    Initialize:
        stack ← [0]  // State stack
        index ← 0    // Input position

    while true:
        state ← top of stack
        token ← tokens[index] or '$' if index >= len(tokens)
        action ← action_table[state][token]

        if action is None:
            THROW ERROR("Unexpected token")

        switch(action):
            case Shift sn:
                Push state n to stack
                index ← index + 1

            case Reduce rn:
                rule ← grammar[n-1]
                Pop |rule.RHS| elements from stack
                next_state ← goto_table[stack.top][rule.LHS]
                Push next_state to stack

            case Accept:
                RETURN "Parse successful"

            default:
                THROW ERROR("Invalid action")
```

The idea of this part is to refer to the implementation of the LR(1) parser in the book.

The parse method in the Parser class takes a token list as input and implements the functionality of the LR(1) parser.

If action starts with ‘s’, it means a shift operation, push the corresponding state onto the stack, increment index by 1, and read the next character; if action starts with ‘r’, it means a reduction operation, pop the elements from the stack according to the rule, and then push the state from the goto table onto the stack.

2.3.5 Description of important Data Structures

- `action_table`: Action table, used to store state transition relationships.

- `goto_table`: Goto table, used to store state transition relationships during reduction.
- `grammar`: Grammar rules, used for state transitions during reduction.
- `stack`: Used to store states during state transitions.

2.3.6 Description of core Algorithms

- `parse`: Analyze the input token sequence, perform state transitions based on the LR(1) parsing table, and determine whether the analysis is successful. If action starts with 's', it means a shift operation, push the corresponding state onto the stack, increment index by 1, and read the next character; if action starts with 'r', it means a reduction operation, pop the elements from the stack according to the rule, and then push the state from the goto table onto the stack.

2.3.7 Test the parser with the following token sequences

1. int op int $
2. id op id $
3. (int op int) $
4. int ; int $
5. id ; op int $
6. (int) op int ; $

These six cases correspond to simple expressions, identifier expressions, expressions with brackets, expressions with semicolons, complex expressions, and the situation where state 15 encounters op.

Result：

- The first case tests the addition of two integers.

```
(base) yan_tai@Yantai:/mnt/e/bianyi/LR1$ python parser.py
Parsing tokens: ['int', 'op', 'int', '$']
State: 0, Token: int, Action: s3
State: 3, Token: op, Action: r3
State: 1, Token: op, Action: s10
State: 10, Token: int, Action: s3
State: 3, Token: $, Action: r3
State: 15, Token: $, Action: r1
State: 1, Token: $, Action: SUCC
Parsing succeeded!
```

- The second case tests the addition of two identifiers.

```
Parsing tokens: ['id', 'op', 'id', '$']
State: 0, Token: id, Action: s4
State: 4, Token: op, Action: r4
State: 1, Token: op, Action: s10
State: 10, Token: id, Action: s4
State: 4, Token: $, Action: r4
State: 15, Token: $, Action: r1
State: 1, Token: $, Action: SUCC
Parsing succeeded!
```

- The third case tests the addition of two expressions with brackets.

```
Parsing tokens: ['(', 'int', 'op', 'int', ')', '$']
State: 0, Token: (, Action: s2
State: 2, Token: int, Action: s7
State: 7, Token: op, Action: r3
State: 5, Token: op, Action: s12
State: 12, Token: int, Action: s7
State: 7, Token: ), Action: r3
State: 16, Token: ), Action: r1
State: 5, Token: ), Action: s11
State: 11, Token: $, Action: r2
State: 1, Token: $, Action: SUCC
Parsing succeeded!
```

- The fourth case is a wrong expression.

```
Parsing tokens: ['int', ';', 'int', '$']
State: 0, Token: int, Action: s3
State: 3, Token: ;, Action: r3
State: 1, Token: ;, Action: s9
State: 9, Token: int, Action: None
Unexpected token: int
```

- The fifth case tests the assignment of an identifier and an expression with a semicolon.

```
Parsing tokens: ['id', ';', 'op', 'int', '$']
State: 0, Token: id, Action: s4
State: 4, Token: ;, Action: r4
State: 1, Token: ;, Action: s9
State: 9, Token: op, Action: r5
State: 1, Token: op, Action: s10
State: 10, Token: int, Action: s3
State: 3, Token: $, Action: r3
State: 15, Token: $, Action: r1
State: 1, Token: $, Action: SUCC
Parsing succeeded!
```

- The sixth case tests the assignment of an expression with brackets and an integer.

```
Parsing tokens: ['(', 'int', ')', 'op', 'int', ';', '$']
State: 0, Token: (, Action: s2
State: 2, Token: int, Action: s7
State: 7, Token: ), Action: r3
State: 5, Token: ), Action: s11
State: 11, Token: op, Action: r2
State: 1, Token: op, Action: s10
State: 10, Token: int, Action: s3
State: 3, Token: ;, Action: r3
State: 15, Token: ;, Action: s9
State: 9, Token: $, Action: r5
State: 15, Token: $, Action: r1
State: 1, Token: $, Action: SUCC
Parsing succeeded!
```

It can be seen that the LR(1) parser can correctly analyze the input token sequence, demonstrate the process and reduct it according to the CFG, and finally determine whether the analysis is successful.

2.3.8 Problems occurred and related solutions

1. When constructing the LR(1) parsing table, it is necessary to consider all possible state transitions, which may lead to a large table that is difficult to maintain.

Solution: Changed from hand-drawing to using Excel for management, which is convenient for viewing and modification. It can also be read in with one click through the script to prevent errors when converting from the table to the code.

2. For cases like state 15 encountering op, there may be reduction or shift situations, making it impossible to distinguish between reduction and shift.

Solution: Following the form of the assignment, priority is specified, that is, reduction takes precedence over shift for op, and shift takes precedence over reduction for ;.

2.4 Your feelings and comments

This experiment has given me a deeper understanding of compiler theory, especially the construction and analysis process of the LR(1) parser. By completing the specific porting of the LR(1) parsing table, I have gained a deeper understanding of how compilers perform syntax analysis and a deeper understanding of the role of LR(1) parsing tables. During the code implementation process, I encountered some problems, but through consulting materials and teachers, I finally solved them, which gave me more confidence in learning compiler theory.