Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное образовательное учреждение
высшего образования
«Московский государственный технический университет
имени Н. Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н. Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

ОТЧЕТ

по лабораторной работе №3

по курсу «Конструирование компиляторов»

на тему: «Синтаксический разбор с использованием метода рекурсивного спуска»

Вариант № 4

Студент ИУ7-22М (Группа) _____ (Подпись, дата) И. А. Глотов (И. О. Фамилия)

Преподаватель _____ (Подпись, дата) А. А. Ступников (И. О. Фамилия)

2025 г.

1 Теоретическая часть

Цель работы: приобретение практических навыков реализации алгоритма рекурсивного спуска для разбора грамматики и построения синтаксического дерева.

Задачи работы:

1. Познакомиться с методом рекурсивного спуска для синтаксического анализа.

2. Разработать, тестировать и отладить программу построения синтаксического дерева методом рекурсивного спуска в соответствии с предложенным вариантом грамматики.

1.1 Задание

1. Дополнить грамматику по варианту блоком, состоящим из последовательности операторов присваивания (выбран стиль Си).

2. Для модифицированной грамматики написать программу нисходящего синтаксического анализа с использованием метода рекурсивного спуска.

Блок в стиле Си:

```
<программа> ->
    <блок>
<блок> ->
     <список операторов>
<список операторов> ->
    <оператор> < хвост>
<хвост> ->
    ; <оператор> <хвост> | epsilon
```

Грамматика по варианту:

```
<выражение> ->
   <логическое выражение>
<логическое выражение> ->
    <логический одночлен> |
    <логическое выражение> ! <логический одночлен>
<логический одночлен> ->
    <вторичное логическое выражение> |
    <логический одночлен> & <вторичное логическое выражение>
<вторичное логическое выражение> ->
    <первичное логическое выражение> |
    ~ <первичное логическое выражение>
<первичное логическое выражение> ->
    <логическое значение> |
<идентификатор>
    <логическое значение> ->
    true | false
<знак логической операции> ->
    ~ | & | !
```

Грамматика по варианту после добавления блока:

```
<программа> ->
    <блок>
<блок> ->
     <список операторов>
<список операторов> ->
    <оператор> < хвост>
<хвост> ->
    ; <оператор> <хвост> | epsilon
```

```
<выражение> ->
    <логическое выражение>
<логическое выражение> ->
    <логический одночлен> |
    <логическое выражение> ! <логический одночлен>
<логический одночлен> ->
    <вторичное логическое выражение> |
    <логический одночлен> & <вторичное логическое выражение>
<вторичное логическое выражение> ->
    <первичное логическое выражение> |
    ~ <первичное логическое выражение>
<первичное логическое выражение> ->
    <логическое значение> |
    <идентификатор>
<логическое значение> ->
    true | false
<знак логической операции> ->
    ~ | & | !
```

2 Практическая часть

2.1 Результат выполнения работы

В листинге 2.1 представлены входные данные. На рисунке ?? — построенное AST-дерево.

Листинг 2.1 – Входная программа

```
{
  x = a & b !~c;
  y = d ! ~b & true;
  z = false
}
```

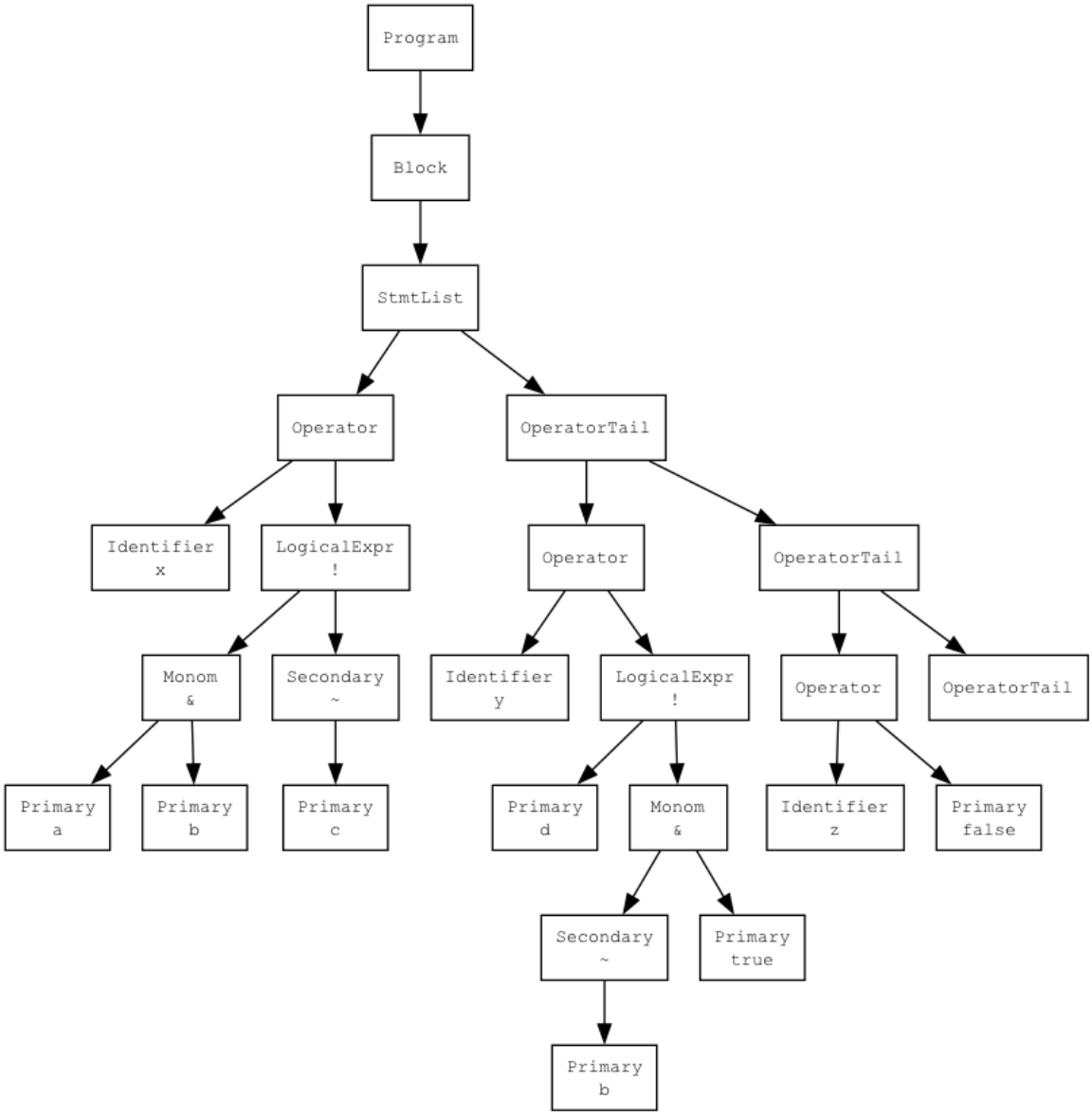

Рисунок 2.1 – AST

ЗАКЛЮЧЕНИЕ

В ходе выполнения лабораторной работы была реализована программа на языке Go, позволяющая строить абстрактное синтаксическое дерево по входной программе.

Таким образом, в результате выполнения лабораторной работы были приобретены практические навыки реализации алгоритма рекурсивного спуска для разбора грамматики и построения синтаксического дерева.

ПРИЛОЖЕНИЕ А

Листинг А.1 – Код модуля *lexer*

```
package lexer

import (
"unicode"
)

const (
TokenEOF = iota
TokenERROR
TokenIDENT
TokenBOOL
TokenLBRACE
TokenRBRACE
TokenSEMICOLON
TokenASSIGN
TokenNOT
TokenAND
TokenOR
TokenEmpty
)

var (
keywords = map[string]int{
  "true":  TokenBOOL,
  "false": TokenBOOL,
}

operators = map[string]int{
  "{": TokenLBRACE,
    "}": TokenRBRACE,
  ";": TokenSEMICOLON,
  "=": TokenASSIGN,
  "~": TokenNOT,
  "&": TokenAND,
  "!": TokenOR,
}
)
```

```
38
39  type Token struct {
40    Type    int
41    Literal string
42  }
43
44  type Lexer struct {
45    input string
46    pos   int
47  }
48
49  func NewLexer(input string) *Lexer {
50    return &Lexer{
51      input: input,
52    }
53  }
54
55  func (l *Lexer) Tokenize() []Token {
56    tokens := []Token{}
57
58    tok := Token{Type: TokenEmpty}
59    for tok.Type != TokenEOF && tok.Type != TokenERROR {
60      tok = l.NextToken()
61      tokens = append(tokens, tok)
62    }
63    return tokens
64  }
65
66  func (l *Lexer) NextToken() Token {
67    l.skipSpaces()
68
69    if l.pos >= len(l.input) {
70      return Token{Type: TokenEOF}
71    }
72
73    if typ, ok := l.readOperator(); ok {
74      return typ
75    }
76
77    ch := rune(l.input[l.pos])
78
```

```
79      if unicode.IsLetter(ch) {
80        return l.readIdentifierOrBool()
81      }
82
83      l.pos++
84      return Token{Type: TokenERROR, Literal: string(ch)}
85    }
86
87    func (l *Lexer) readOperator() (Token, bool) {
88      tok := l.input[l.pos : l.pos+1]
89      if typ, ok := operators[tok]; ok {
90        tok = Token{
91          Type:    typ,
92          Literal: tok,
93        }
94        l.pos++
95        return tok, true
96      }
97      return Token{}, false
98    }
99
100   func (l *Lexer) skipSpaces() {
101     if l.pos >= len(l.input) {
102       return
103     }
104
105     for ; l.pos < len(l.input); l.pos++ {
106       ch := rune(l.input[l.pos])
107       if !unicode.IsSpace(ch) {
108         return
109       }
110     }
111   }
112
113   func (l *Lexer) readIdentifierOrBool() Token {
114     start := l.pos
115     for l.pos < len(l.input) {
116       ch := rune(l.input[l.pos])
117       if !unicode.IsLetter(ch) && !unicode.IsDigit(ch) && ch != '_'
              {
118         break
```

```
119         }
120         l.pos++
121       }
122
123       literal := l.input[start:l.pos]
124
125       if typ, ok := keywords[literal]; ok {
126         return Token{Type: typ, Literal: literal}
127       }
128       return Token{Type: TokenIDENT, Literal: literal}
129     }
```

Листинг А.2 – Код модуля *parser*

```
1   package parser
2
3   import (
4   "fmt"
5   "strings"
6
7   "github.com/Erlendum/BMSTU_CC/lab_03/internal/lexer"
8   )
9
10  type ASTNode struct {
11    Type     string
12    Value    string
13    Children []*ASTNode
14    Token    lexer.Token
15  }
16
17  type Parser struct {
18    tokens []lexer.Token
19    pos    int
20  }
21
22  func NewParser(tokens []lexer.Token) *Parser {
23    return &Parser{
24      tokens: tokens,
25    }
26  }
27
28  func (p *Parser) CurrentToken() lexer.Token {
29    if p.pos >= len(p.tokens) {
```

```
30        return lexer.Token{Type: lexer.TokenEOF}
31      }
32      return p.tokens[p.pos]
33    }
34
35    func (p *Parser) expect(tokenType int) (lexer.Token, bool) {
36      tok := p.CurrentToken()
37      if tok.Type == tokenType {
38        p.incPos()
39        return tok, true
40      }
41      return tok, false
42    }
43
44    func (p *Parser) parsePrimary() (*ASTNode, bool) {
45      tok := p.CurrentToken()
46      switch tok.Type {
47        case lexer.TokenBOOL:
48        p.incPos()
49        return &ASTNode{
50          Type:  "Primary",
51          Value: tok.Literal,
52          Token: tok,
53        }, true
54        case lexer.TokenIDENT:
55        p.incPos()
56        return &ASTNode{
57          Type:  "Primary",
58          Value: tok.Literal,
59          Token: tok,
60        }, true
61        default:
62        return nil, false
63      }
64    }
65
66    func (p *Parser) parseSecondary() (*ASTNode, bool) {
67      if p.isCurrentTokenMatchType(lexer.TokenNOT) {
68        primary, ok := p.parsePrimary()
69        if !ok {
70          return nil, false
```

```
71        }
72        return &ASTNode{
73          Type:     "Secondary",
74          Children: []*ASTNode{primary},
75          Value:    "~",
76        }, true
77      }
78      return p.parsePrimary()
79    }
80
81    func (p *Parser) parseMonom() (*ASTNode, bool) {
82      left, ok := p.parseSecondary()
83      if !ok {
84        return nil, false
85      }
86
87      for p.CurrentToken().Type == lexer.TokenAND {
88        op := p.CurrentToken()
89        p.incPos()
90        right, ok := p.parseSecondary()
91        if !ok {
92          return nil, false
93        }
94        left = &ASTNode{
95          Type:     "Monom",
96          Children: []*ASTNode{left, right},
97          Value:    op.Literal,
98          Token:    op,
99        }
100     }
101     return left, true
102   }
103
104   func (p *Parser) parseLogicalExpr() (*ASTNode, bool) {
105     left, ok := p.parseMonom()
106     if !ok {
107       return nil, false
108     }
109
110     for p.CurrentToken().Type == lexer.TokenOR {
111       op := p.CurrentToken()
```

```
112         p.incPos()
113         right, ok := p.parseMonom()
114         if !ok {
115           return nil, false
116         }
117         left = &ASTNode{
118           Type:     "LogicalExpr",
119           Children: []*ASTNode{left, right},
120           Value:    op.Literal,
121           Token:    op,
122         }
123       }
124       return left, true
125     }
126
127     func (p *Parser) parseExpression() (*ASTNode, bool) {
128       return p.parseLogicalExpr()
129     }
130
131     func (p *Parser) parseOperator() (*ASTNode, bool) {
132       idTok, ok := p.expect(lexer.TokenIDENT)
133       if !ok {
134         return nil, false
135       }
136
137       if _, ok := p.expect(lexer.TokenASSIGN); !ok {
138         return nil, false
139       }
140
141       expr, ok := p.parseExpression()
142       if !ok {
143         return nil, false
144       }
145
146       return &ASTNode{
147         Type: "Operator",
148         Children: []*ASTNode{
149           {Type: "Identifier", Value: idTok.Literal, Token: idTok},
150           expr,
151         },
152         Token: idTok,
```

```
153     }, true
154   }
155
156   func (p *Parser) parseOperatorTail() (*ASTNode, bool) {
157     if !p.isCurrentTokenMatchType(lexer.TokenSEMICOLON) {
158       return &ASTNode{
159         Type: "OperatorTail",
160       }, true
161     }
162
163     op, ok := p.parseOperator()
164     if !ok {
165       return nil, false
166     }
167
168     tail, ok := p.parseOperatorTail()
169     if !ok {
170       return nil, false
171     }
172
173     return &ASTNode{
174       Type:     "OperatorTail",
175       Children: []*ASTNode{op, tail},
176     }, true
177   }
178
179   func (p *Parser) parseStmtList() (*ASTNode, bool) {
180     first, ok := p.parseOperator()
181     if !ok {
182       return nil, false
183     }
184
185     tail, ok := p.parseOperatorTail()
186     if !ok {
187       return nil, false
188     }
189
190     return &ASTNode{
191       Type:     "StmtList",
192       Children: []*ASTNode{first, tail},
193     }, true
```

```
194   }
195
196   func (p *Parser) parseBlock() (*ASTNode, bool) {
197     if !p.isCurrentTokenMatchType(lexer.TokenLBRACE) {
198       return nil, false
199     }
200
201     stmts, ok := p.parseStmtList()
202     if !ok {
203       return nil, false
204     }
205
206     if !p.isCurrentTokenMatchType(lexer.TokenRBRACE) {
207       return nil, false
208     }
209
210     return &ASTNode{
211       Type:     "Block",
212       Children: []*ASTNode{stmts},
213     }, true
214   }
215
216   func (p *Parser) parseProgram() (*ASTNode, bool) {
217     block, ok := p.parseBlock()
218     if !ok {
219       return nil, false
220     }
221
222     return &ASTNode{
223       Type:     "Program",
224       Children: []*ASTNode{block},
225     }, true
226   }
227
228   func (p *Parser) Parse() (*ASTNode, bool) {
229     return p.parseProgram()
230   }
231
232   func (p *Parser) incPos() {
233     p.pos++
234   }
```

```
235
236 func (p *Parser) isCurrentTokenMatchType(tokenType int) bool {
237   if p.CurrentToken().Type == tokenType {
238     p.incPos()
239     return true
240   }
241   return false
242 }
243
244 func (n *ASTNode) ToDot() string {
245   var builder strings.Builder
246   builder.WriteString("digraph AST {\n")
247     builder.WriteString("  node [shape=box, fontname=\"Courier\",
          fontsize=10];\n")
248     builder.WriteString("  edge [fontname=\"Courier\", fontsize
        =10];\n\n")
249
250     var nodeCounter int
251     generateDOTNode(&builder, n, &nodeCounter)
252
253     builder.WriteString("}\n")
254   return builder.String()
255 }
256
257 func generateDOTNode(builder *strings.Builder, node *ASTNode,
     counter *int) int {
258   if node == nil {
259     return -1
260   }
261
262   currentID := *counter
263   *counter++
264
265   label := node.Type
266   if node.Value != "" {
267     label += fmt.Sprintf("\\n%s", node.Value)
268   }
269
270   builder.WriteString(fmt.Sprintf("  node%d [label=\"%s\"];\n",
        currentID, label))
271
```

```
272     for _, child := range node.Children {
273       childID := generateDOTNode(builder, child, counter)
274       if childID >= 0 {
275         builder.WriteString(fmt.Sprintf("  node%d -> node%d;\n",
              currentID, childID))
276       }
277     }
278
279     return currentID
280   }
```