



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»

## Отчёт по лабораторной работе №3 по курсу «Конструирование компиляторов»

Вариант №2

Тема Синтаксический разбор с использованием метода рекурсивного спуска

Студент Аскарян К.А.

Группа ИУ7-21М

Преподаватель Ступников А. А.

Москва — 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 Алгол-Паскаль).
2. Для модифицированной грамматики написать программу нисходящего синтаксического анализа с использованием метода рекурсивного спуска.

Блок в стиле Алгол-Паскаль:

```
<программа> -> <блок>
<блок> -> begin <список операторов> end
<список операторов> ->
    <оператор> | <список операторов> ; <оператор>
<оператор> -> <идентификатор> = <выражение>
```

Грамматика по варианту:

```
<выражение> ->
    <арифм выражение> <операция отношения> <арифм выражение> |
    <арифм выражение>

<арифм выражение> ->
```

```

<арифм выражение> <операция типа сложения> <терм> |
<терм>

<терм> ->
    <терм> <операция типа умножения> <фактор> |
    <фактор>

<фактор> ->
    <идентификатор> |
    <константа> |
    ( <арифм выражение> )

<операция отношения> -> < | <= | = | <> | > | >=
<операция типа сложения> -> + | -
<операция типа умножения> -> * | /

```

Грамматика по варианту после удаления левой рекурсии и добавления блока:

```

<программа> -> <блок>
<блок> -> begin <список операторов> end
<список операторов> -> <оператор> <оператор_хвост>
<оператор_хвост> -> ; <оператор> <оператор_хвост> | epsilon
<оператор> -> <идентификатор> = <выражение>

<выражение> ->
    <арифм выражение> <операция отношения> <арифм выражение> |
    <арифм выражение>

<арифм выражение> ->
    <терм> |
    <арифм выражение> <операция типа сложения> <терм>

```

<терм> ->  
    <фактор> |  
    <терм> <операция типа умножения> <фактор>

<фактор> ->  
    <идентификатор> |  
    <константа> |  
    ( <арифм выражение> )

<операция отношения> -> < | <= | = | <> | > | >=

<операция типа сложения> -> + | -

<операция типа умножения> -> \* | /

## 2 Практическая часть

### 2.1 Результат выполнения работы

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

Листинг 2.1 – Входная программа

```
1 begin
2   hello = 1 + 2;
3   world = hello*(1 + 2 * 3);
4   equals = hello <> world
5 end
```

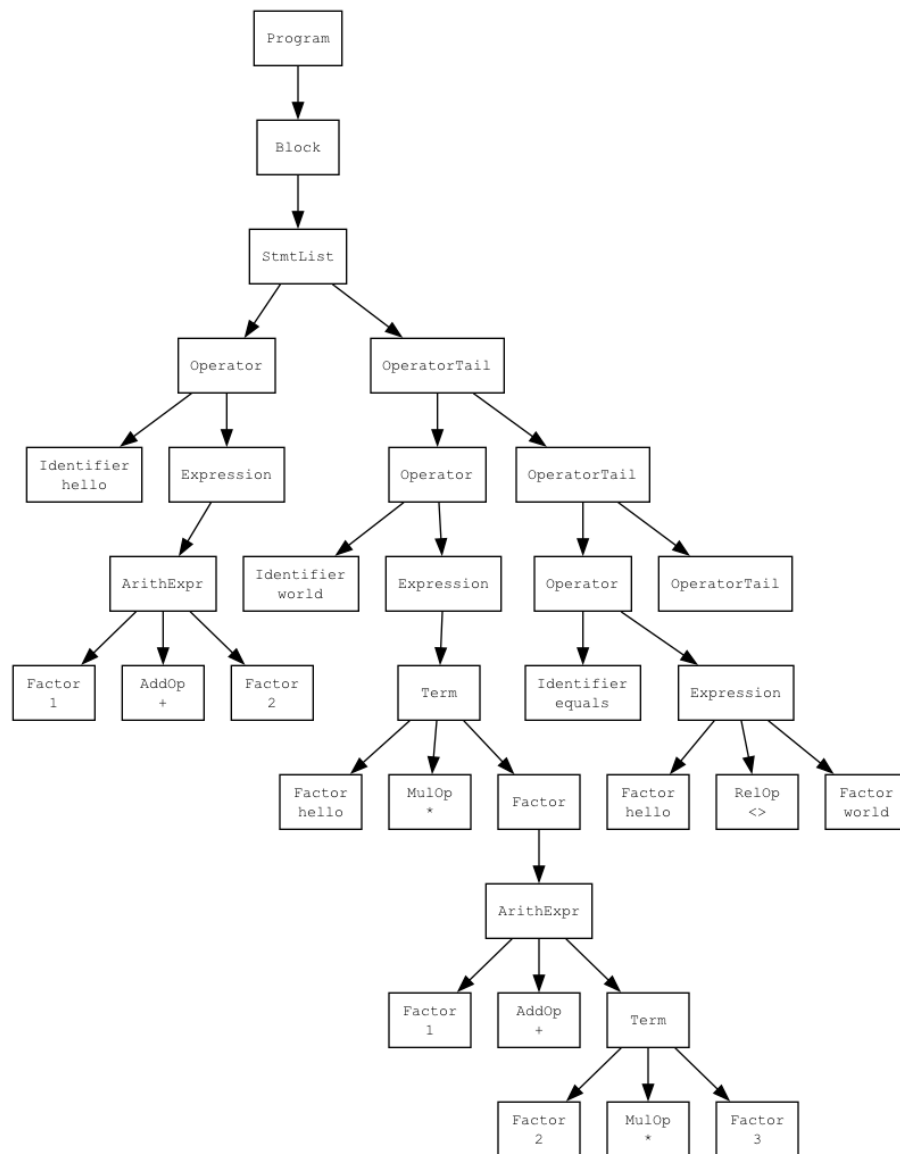


Рисунок 2.1

## 2.2 Код программы

В листингах 2.2 — 2.6 приведен код программы на языке Go.

Листинг 2.2 – Код модуля *lexer*

```
1 package lexer
2
3 import "unicode"
4
5 type Lexer struct {
6     input    string
7     pos      int
8     line     int
9     column   int
10    start     int
11    startLn   int
12    startCl   int
13 }
14
15 func NewLexer(input string) *Lexer {
16     return &Lexer{
17         input:  input,
18         line:   1,
19         column: 1,
20     }
21 }
22
23 func (l *Lexer) NextToken() Token {
24     l.skipWhitespace()
25
26     l.start = l.pos
27     l.startLn = l.line
28     l.startCl = l.column
29
30     if l.pos >= len(l.input) {
31         return Token{Type: TokenEOF}
32     }
33
34     if typ, ok := l.tryOperator(); ok {
35         return typ
36     }
37
38     ch := l.input[l.pos]
39
40     if unicode.IsLetter(rune(ch)) {
41         return l.readIdentifier()
42     }
43 }
```

```

44     if unicode.IsDigit(rune(ch)) {
45         return l.readNumber()
46     }
47
48     l.pos++
49     l.column++
50     return Token{Type: TokenERROR, Literal: string(ch)}
51 }
52
53 func (l *Lexer) tryOperator() (Token, bool) {
54     if l.pos+1 < len(l.input) {
55         sub := l.input[l.pos : l.pos+2]
56         if typ, ok := operators[sub]; ok {
57             tok := Token{
58                 Type:      typ,
59                 Literal: sub,
60             }
61             l.pos += 2
62             l.column += 2
63             return tok, true
64         }
65     }
66
67     sub := l.input[l.pos : l.pos+1]
68     if typ, ok := operators[sub]; ok {
69         tok := Token{
70             Type:      typ,
71             Literal: sub,
72         }
73         l.pos++
74         l.column++
75         return tok, true
76     }
77
78     return Token{}, false
79 }
80
81 func (l *Lexer) skipWhitespace() {
82     for l.pos < len(l.input) {
83         ch := l.input[l.pos]
84         if ch == '\n' {
85             l.line++
86             l.column = 1
87             l.pos++
88             continue
89         }
90         if unicode.IsSpace(rune(ch)) {
91             l.pos++

```

```

92         l.column++
93         continue
94     }
95     break
96 }
97 }
98
99 func (l *Lexer) readIdentifier() Token {
100     start := l.pos
101     for l.pos < len(l.input) {
102         ch := rune(l.input[l.pos])
103         if !unicode.IsLetter(ch) && !unicode.IsDigit(ch) && ch != '_' {
104             break
105         }
106         l.pos++
107         l.column++
108     }
109
110     literal := l.input[start:l.pos]
111
112     if typ, ok := keywords[literal]; ok {
113         return Token{Type: typ, Literal: literal}
114     }
115     return Token{Type: TokenIDENT, Literal: literal}
116 }
117
118 func (l *Lexer) readNumber() Token {
119     start := l.pos
120     for l.pos < len(l.input) {
121         ch := rune(l.input[l.pos])
122         if !unicode.IsDigit(ch) {
123             break
124         }
125         l.pos++
126         l.column++
127     }
128
129     literal := l.input[start:l.pos]
130     return Token{Type: TokenNUMBER, Literal: literal}
131 }

```

### Листинг 2.3 – Код модуля *lexer*

```

1 package lexer
2
3 const (
4     TokenEOF = iota
5     TokenERROR
6     TokenIDENT

```



```

7     TokenNUMBER
8     TokenBEGIN
9     TokenEND
10    TokenSEMICOLON
11    TokenASSIGN
12    TokenEQ
13    TokenPLUS
14    TokenMINUS
15    TokenMULT
16    TokenDIV
17    TokenLT
18    TokenLE
19    TokenGT
20    TokenGE
21    TokenNE
22    TokenLPAREN
23    TokenRPAREN
24 )
25
26 var (
27     keywords = map[string]int{
28         "begin": TokenBEGIN,
29         "end":   TokenEND,
30     }
31
32     operators = map[string]int{
33         ";": TokenSEMICOLON,
34         "=": TokenASSIGN,
35         "==": TokenEQ,
36         "+": TokenPLUS,
37         "-": TokenMINUS,
38         "*": TokenMULT,
39         "/": TokenDIV,
40         "(": TokenLPAREN,
41         ")": TokenRPAREN,
42         "<": TokenLT,
43         "<=": TokenLE,
44         ">": TokenGT,
45         ">=": TokenGE,
46         "<>": TokenNE,
47     }
48 )
49
50 type Token struct {
51     Type      int
52     Literal   string
53 }
54

```

```

55 func (l *Lexer) Tokenize() []Token {
56     var tokens []Token
57     for {
58         tok := l.NextToken()
59         tokens = append(tokens, tok)
60         if tok.Type == TokenEOF || tok.Type == TokenERROR {
61             break
62         }
63     }
64     return tokens
65 }

```

Листинг 2.4 – Код модуля *grammar* — устранение бесполезных символов

```

1 package parser
2
3 import "github.com/AskaryanKarine/BMSTU-CC/lab_03/internal/lexer"
4
5 type ASTNode struct {
6     Type      string
7     Value     string
8     Children  []*ASTNode
9     Token     lexer.Token
10 }
11
12 type Parser struct {
13     tokens []lexer.Token
14     pos    int
15 }
16
17 func NewParser(tokens []lexer.Token) *Parser {
18     return &Parser{
19         tokens: tokens,
20     }
21 }
22
23 func (p *Parser) CurrentToken() lexer.Token {
24     if p.pos >= len(p.tokens) {
25         return lexer.Token{Type: lexer.TokenEOF}
26     }
27     return p.tokens[p.pos]
28 }
29
30 func (p *Parser) advance() {
31     p.pos++
32 }
33
34 func (p *Parser) match(tokenType int) bool {
35     if p.CurrentToken().Type == tokenType {

```

```

36         p.advance()
37         return true
38     }
39     return false
40 }
41
42 func (p *Parser) expect(tokenType int) (lexer.Token, bool) {
43     tok := p.CurrentToken()
44     if tok.Type == tokenType {
45         p.advance()
46         return tok, true
47     }
48     return tok, false
49 }
50
51 // Правила грамматики
52 func (p *Parser) parseFactor() (*ASTNode, bool) {
53     tok := p.CurrentToken()
54     switch tok.Type {
55     case lexer.TokenIDENT:
56         p.advance()
57         return &ASTNode{
58             Type:  "Factor",
59             Value: tok.Literal,
60             Token: tok,
61         }, true
62     case lexer.TokenNUMBER:
63         p.advance()
64         return &ASTNode{
65             Type:  "Factor",
66             Value: tok.Literal,
67             Token: tok,
68         }, true
69     case lexer.TokenLPAREN:
70         p.advance()
71         expr, ok := p.parseArithExpr()
72         if !ok {
73             return nil, false
74         }
75         if _, ok := p.expect(lexer.TokenRPAREN); !ok {
76             return nil, false
77         }
78         return &ASTNode{
79             Type:      "Factor",
80             Children:  []*ASTNode{expr},
81             Token:     tok,
82         }, true
83     default:

```

```

84         return nil, false
85     }
86 }
87
88 func (p *Parser) parseTerm() (*ASTNode, bool) {
89     left, ok := p.parseFactor()
90     if !ok {
91         return nil, false
92     }
93
94     for {
95         tok := p.CurrentToken()
96         if tok.Type == lexer.TokenMULT || tok.Type == lexer.TokenDIV {
97             p.advance()
98             right, ok := p.parseFactor()
99             if !ok {
100                 return nil, false
101             }
102             left = &ASTNode{
103                 Type: "Term",
104                 Children: []*ASTNode{
105                     left,
106                     {Type: "MulOp", Value: tok.Literal, Token: tok},
107                     right,
108                 },
109                 Token: tok,
110             }
111         } else {
112             break
113         }
114     }
115     return left, true
116 }
117
118 func (p *Parser) parseArithExpr() (*ASTNode, bool) {
119     left, ok := p.parseTerm()
120     if !ok {
121         return nil, false
122     }
123
124     for {
125         tok := p.CurrentToken()
126         if tok.Type == lexer.TokenPLUS || tok.Type == lexer.TokenMINUS {
127             p.advance()
128             right, ok := p.parseTerm()
129             if !ok {
130                 return nil, false
131             }

```

```

132         left = &ASTNode{
133             Type: "ArithExpr",
134             Children: []*ASTNode{
135                 left,
136                 {Type: "AddOp", Value: tok.Literal, Token: tok},
137                 right,
138             },
139             Token: tok,
140         }
141     } else {
142         break
143     }
144 }
145 return left, true
146 }
147
148 func (p *Parser) parseExpression() (*ASTNode, bool) {
149     left, ok := p.parseArithExpr()
150     if !ok {
151         return nil, false
152     }
153
154     tok := p.CurrentToken()
155     if tok.Type == lexer.TokenLT || tok.Type == lexer.TokenLE ||
156        tok.Type == lexer.TokenGT || tok.Type == lexer.TokenGE ||
157        tok.Type == lexer.TokenEQ || tok.Type == lexer.TokenNE {
158         p.advance()
159         right, ok := p.parseArithExpr()
160         if !ok {
161             return nil, false
162         }
163         return &ASTNode{
164             Type: "Expression",
165             Children: []*ASTNode{
166                 left,
167                 {Type: "RelOp", Value: tok.Literal, Token: tok},
168                 right,
169             },
170             Token: tok,
171         }, true
172     }
173     return &ASTNode{
174         Type: "Expression",
175         Children: []*ASTNode{left},
176         Token: tok,
177     }, true
178 }
179

```

```

180 func (p *Parser) parseOperator() (*ASTNode, bool) {
181     idTok, ok := p.expect(lexer.TokenIDENT)
182     if !ok {
183         return nil, false
184     }
185
186     if _, ok := p.expect(lexer.TokenASSIGN); !ok {
187         return nil, false
188     }
189
190     expr, ok := p.parseExpression()
191     if !ok {
192         return nil, false
193     }
194
195     return &ASTNode{
196         Type: "Operator",
197         Children: []*ASTNode{
198             {Type: "Identifier", Value: idTok.Literal, Token: idTok},
199             expr,
200         },
201         Token: idTok,
202     }, true
203 }
204
205 func (p *Parser) parseOperatorTail() (*ASTNode, bool) {
206     if !p.match(lexer.TokenSEMICOLON) {
207         return &ASTNode{
208             Type: "OperatorTail",
209         }, true
210     }
211
212     op, ok := p.parseOperator()
213     if !ok {
214         return nil, false
215     }
216
217     tail, ok := p.parseOperatorTail()
218     if !ok {
219         return nil, false
220     }
221
222     return &ASTNode{
223         Type: "OperatorTail",
224         Children: []*ASTNode{op, tail},
225     }, true
226 }
227

```

```

228 func (p *Parser) parseStmtList() (*ASTNode, bool) {
229     first, ok := p.parseOperator()
230     if !ok {
231         return nil, false
232     }
233
234     tail, ok := p.parseOperatorTail()
235     if !ok {
236         return nil, false
237     }
238
239     return &ASTNode{
240         Type:      "StmtList",
241         Children: []*ASTNode{first, tail},
242     }, true
243 }
244
245 func (p *Parser) parseBlock() (*ASTNode, bool) {
246     if !p.match(lexer.TokenBEGIN) {
247         return nil, false
248     }
249
250     stmts, ok := p.parseStmtList()
251     if !ok {
252         return nil, false
253     }
254
255     if !p.match(lexer.TokenEND) {
256         return nil, false
257     }
258
259     return &ASTNode{
260         Type:      "Block",
261         Children: []*ASTNode{stmts},
262     }, true
263 }
264
265 func (p *Parser) parseProgram() (*ASTNode, bool) {
266     block, ok := p.parseBlock()
267     if !ok {
268         return nil, false
269     }
270
271     return &ASTNode{
272         Type:      "Program",
273         Children: []*ASTNode{block},
274     }, true
275 }

```





```

        currentID, childID))
40     }
41 }
42
43 return currentID
44 }

```

Листинг 2.6 – Код модуля *fs*

```

1 package fs
2
3 import (
4     "bufio"
5     "fmt"
6     "os"
7     "os/exec"
8     "strings"
9 )
10
11 func ReadProgramFile(filename string) (string, error) {
12     file, err := os.Open(filename)
13     if err != nil {
14         return "", fmt.Errorf("ошибка_открытия_файла:_%w", err)
15     }
16     defer file.Close()
17
18     scanner := bufio.NewScanner(file)
19     var lines []string
20     for scanner.Scan() {
21         lines = append(lines, scanner.Text())
22     }
23     input := strings.Join(lines, "\n")
24     return input, nil
25 }
26
27 func SaveDOTToFile(dot, filenameDOT, filenamePNG string) error {
28     file, err := os.Create(filenameDOT)
29     if err != nil {
30         return err
31     }
32     defer file.Close()
33
34     _, err = file.WriteString(dot)
35     if err != nil {
36         return err
37     }
38     cmd := exec.Command("dot", "-Tpng", "-o", filenamePNG, filenameDOT)
39     if _, err := cmd.CombinedOutput(); err != nil {
40         return err

```

```
41     }  
42  
43     return nil  
44 }
```