



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 №4  
по курсу «Конструирование компиляторов»  
на тему: «Синтаксически управляемый перевод»  
Вариант № 4

Студент ИУ7-22М  
(Группа)

\_\_\_\_\_  
(Подпись, дата)

И. А. Готов  
(И. О. Фамилия)

Преподаватель

\_\_\_\_\_  
(Подпись, дата)

А. А. Ступников  
(И. О. Фамилия)

2025 г.

# 1 Теоретическая часть

**Цель работы:** приобретение практических навыков реализации синтаксически управляемого перевода.

**Задачи работы:**

1. Разработать, тестировать и отладить программу синтаксического анализа в соответствии с предложенным вариантом грамматики.
2. Включить в программу синтаксического анализ семантические действия для реализации синтаксически управляемого перевода инфиксного выражения в обратную польскую нотацию.

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> ->
    ~ | & | !
```

## 2 Практическая часть

### 2.1 Результат выполнения работы

В листинге 2.1 представлены входные данные и выходные данные.

Листинг 2.1 – Входная и выходные данные

```
1 a & b ! ~c ! d & e ! f ! g ! h & k  
2 RPN: [a b & c ~ ! d e & ! f ! g ! h k & !]
```

## ЗАКЛЮЧЕНИЕ

В ходе выполнения лабораторной работы была реализована программа на языке Go, позволяющая преобразовывать инфиксное выражение в обратную польскую нотацию.

Таким образом, в результате выполнения лабораторной работы были приобретены практические навыки синтаксически управляемого перевода.

## ПРИЛОЖЕНИЕ А

Листинг А.1 – Код модуля *lexer*

```
1 package lexer
2
3 import (
4     "unicode"
5 )
6
7 const (
8     TokenEOF = iota
9     TokenERROR
10    TokenIDENT
11    TokenBOOL
12    TokenLBRACE
13    TokenRBRACE
14    TokenSEMICOLON
15    TokenASSIGN
16    TokenNOT
17    TokenAND
18    TokenOR
19    TokenEmpty
20 )
21
22 var (
23     keywords = map[string]int{
24         "true": TokenBOOL,
25         "false": TokenBOOL,
26     }
27
28     operators = map[string]int{
29         "{": TokenLBRACE,
30         "}": TokenRBRACE,
31         ";": TokenSEMICOLON,
32         "=": TokenASSIGN,
33         "~": TokenNOT,
34         "&": TokenAND,
35         "!": TokenOR,
36     }
37 )
```

```

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_04/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) parseProgram() (*ASTNode, bool) {
132     expr, ok := p.parseExpression()
133     if !ok {
134         return nil, false
135     }
136
137     if p.CurrentToken().Type != lexer.TokenEOF {
138         return nil, false
139     }
140
141     return expr, true
142 }
143
144 func (p *Parser) Parse() (*ASTNode, bool) {
145     return p.parseProgram()
146 }
147
148 func (p *Parser) incPos() {
149     p.pos++
150 }
151
152 func (p *Parser) isCurrentTokenMatchType(tokenType int) bool {
153     if p.CurrentToken().Type == tokenType {

```

```

154     p.incPos()
155     return true
156 }
157 return false
158 }
159
160 func (n *ASTNode) ToDot() string {
161     var builder strings.Builder
162     builder.WriteString("digraph AST {\n")
163     builder.WriteString("    node [shape=box, fontname=\"Courier\",
164         fontsize=10];\n")
165     builder.WriteString("    edge [fontname=\"Courier\", fontsize
166         =10];\n\n")
167
168     var nodeCounter int
169     generateDOTNode(&builder, n, &nodeCounter)
170
171     builder.WriteString("}\n")
172     return builder.String()
173 }
174
175 func generateDOTNode(builder *strings.Builder, node *ASTNode,
176     counter *int) int {
177     if node == nil {
178         return -1
179     }
180
181     currentID := *counter
182     *counter++
183
184     label := node.Type
185     if node.Value != "" {
186         label += fmt.Sprintf("\\n%s", node.Value)
187     }
188
189     builder.WriteString(fmt.Sprintf("    node%d [label=\"%s\"]; \n",
190         currentID, label))
191
192     for _, child := range node.Children {
193         childID := generateDOTNode(builder, child, counter)
194         if childID >= 0 {

```

```

191     builder.WriteString(fmt.Sprintf("  node%d -> node%d;\n",
192         currentID , childID))
193 }
194
195 return currentID
196 }
197
198 func (n *ASTNode) ToReversePolishNotation() []string {
199     var result []string
200     n.traversePostOrder(&result)
201     return result
202 }
203
204 func (n *ASTNode) traversePostOrder(result *[]string) {
205     if n == nil {
206         return
207     }
208
209     for _, child := range n.Children {
210         child.traversePostOrder(result)
211     }
212
213     if n.Value != "" {
214         *result = append(*result , n.Value)
215     }
216 }

```