Министерство науки и высшего образования Российской Федерации
Федеральное государственное автомное 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 №4
по курсу «Конструирование компиляторов»

Вариант №2

Тема Синтаксический управляемый перевод

Студент Аскарян К.А.

Группа ИУ7-21М

Преподаватель Ступников А. А.

Москва — 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.

1.1 Задание

Реализовать синтаксически управляемый перевод инфиксного выражения в обратную польскую нотацию для грамматики выражений из лабораторной работы №3. Для построения дерева разбора использовать синтаксический анализатор для данной грамматики разработанный в лабораторной работы №3.

Грамматика по варианту для выражений:

```
<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> -> * | /
```

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

```
a < b + c
```

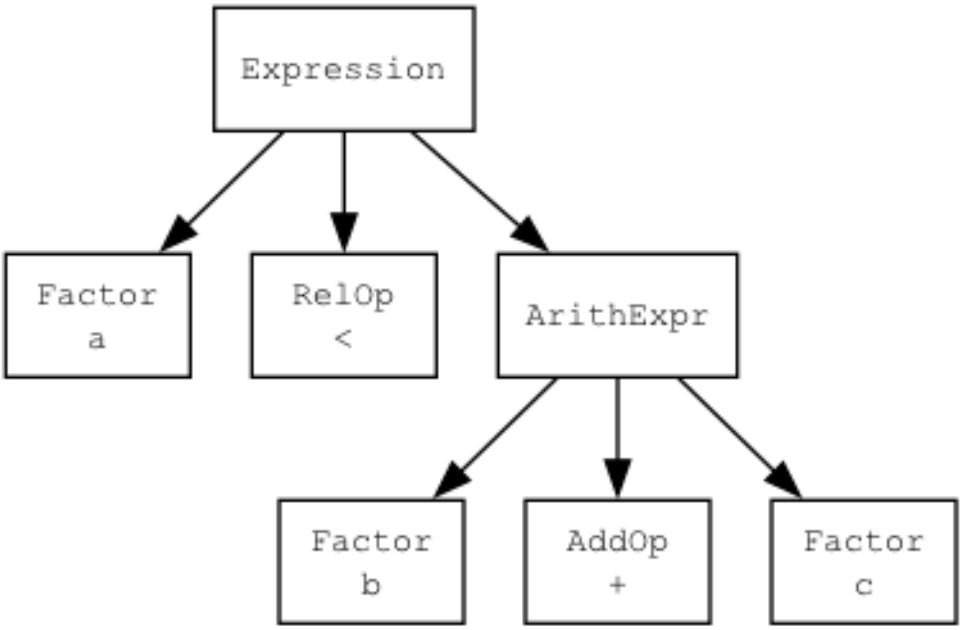

Рисунок 2.1

2.2 Код программы

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

Листинг 2.2 – Код модуля *lexer*

```
package lexer

import "unicode"

type Lexer struct {
    input   string
    pos     int
    line    int
    column  int
    start   int
    startLn int
    startCl int
```

```
}

func NewLexer(input string) *Lexer {
    return &Lexer{
        input:  input,
        line:   1,
        column: 1,
    }
}

func (l *Lexer) NextToken() Token {
    l.skipWhitespace()

    l.start = l.pos
    l.startLn = l.line
    l.startCl = l.column

    if l.pos >= len(l.input) {
        return Token{Type: TokenEOF}
    }

    if typ, ok := l.tryOperator(); ok {
        return typ
    }

    ch := l.input[l.pos]

    if unicode.IsLetter(rune(ch)) {
        return l.readIdentifier()
    }

    if unicode.IsDigit(rune(ch)) {
        return l.readNumber()
    }

    l.pos++
    l.column++
    return Token{Type: TokenERROR, Literal: string(ch)}
}

func (l *Lexer) tryOperator() (Token, bool) {
    if l.pos+1 < len(l.input) {
        sub := l.input[l.pos : l.pos+2]
        if typ, ok := operators[sub]; ok {
            tok := Token{
                Type:    typ,
                Literal: sub,
            }
```

```
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
```

```
)

var (
    keywords = map[string]int{
        "begin": TokenBEGIN,
        "end":   TokenEND,
    }

    operators = map[string]int{
        ";":  TokenSEMICOLON,
        "=":  TokenASSIGN,
        "==": TokenEQ,
        "+":  TokenPLUS,
        "-":  TokenMINUS,
        "*":  TokenMULT,
        "/":  TokenDIV,
        "(":  TokenLPAREN,
        ")":  TokenRPAREN,
        "<":  TokenLT,
        "<=": TokenLE,
        ">":  TokenGT,
        ">=": TokenGE,
        "<>": TokenNE,
    }
)

type Token struct {
    Type    int
    Literal string
}

func (l *Lexer) Tokenize() []Token {
    var tokens []Token
    for {
        tok := l.NextToken()
        tokens = append(tokens, tok)
        if tok.Type == TokenEOF || tok.Type == TokenERROR {
            break
        }
    }
    return tokens
}
```

Листинг 2.4 – Код модуля *parser*

```
package parser

import "github.com/AskaryanKarine/BMSTU-CC/lab_04/internal/lexer"

```

```
type ASTNode struct {
    Type     string
    Value    string
    Children []*ASTNode
    Token    lexer.Token
}

type Parser struct {
    tokens []lexer.Token
    pos    int
}

func NewParser(tokens []lexer.Token) *Parser {
    return &Parser{
        tokens: tokens,
    }
}

func (p *Parser) CurrentToken() lexer.Token {
    if p.pos >= len(p.tokens) {
        return lexer.Token{Type: lexer.TokenEOF}
    }
    return p.tokens[p.pos]
}

func (p *Parser) advance() {
    p.pos++
}

func (p *Parser) match(tokenType int) bool {
    if p.CurrentToken().Type == tokenType {
        p.advance()
        return true
    }
    return false
}

func (p *Parser) expect(tokenType int) (lexer.Token, bool) {
    tok := p.CurrentToken()
    if tok.Type == tokenType {
        p.advance()
        return tok, true
    }
    return tok, false
}

// Правила грамматики
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
```

```
            }
            left = &ASTNode{
                Type: "Term",
                Children: []*ASTNode{
                    left,
                    {Type: "MulOp", Value: tok.Literal, Token: tok},
                    right,
                },
                Token: tok,
            }
        } else {
            break
        }
    }
    return left, true
}

func (p *Parser) parseArithExpr() (*ASTNode, bool) {
    left, ok := p.parseTerm()
    if !ok {
        return nil, false
    }

    for {
        tok := p.CurrentToken()
        if tok.Type == lexer.TokenPLUS || tok.Type == lexer.TokenMINUS {
            p.advance()
            right, ok := p.parseTerm()
            if !ok {
                return nil, false
            }
            left = &ASTNode{
                Type: "ArithExpr",
                Children: []*ASTNode{
                    left,
                    {Type: "AddOp", Value: tok.Literal, Token: tok},
                    right,
                },
                Token: tok,
            }
        } else {
            break
        }
    }
    return left, true
}

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) parseProgram() (*ASTNode, bool) {
181     expr, ok := p.parseExpression()
182     if !ok {
183         return nil, false
184     }
185 
186     if p.CurrentToken().Type != lexer.TokenEOF {
187         return nil, false
188     }
189     return expr, true
190 }
191 
192 func (p *Parser) Parse() (*ASTNode, bool) {
193     return p.parseProgram()
194 }
195 
196 func ASTToRPN(node *ASTNode) []string {
```

```
197     if node == nil {
198         return nil
199     }
200 
201     var rpn []string
202 
203     if len(node.Children) == 3 &&
204         (node.Type == "Expression" || node.Type == "ArithExpr" ||
               node.Type == "Term") {
205         left := ASTToRPN(node.Children[0])
206         right := ASTToRPN(node.Children[2])
207         op := ASTToRPN(node.Children[1])
208 
209         rpn = append(rpn, left...)
210         rpn = append(rpn, right...)
211         rpn = append(rpn, op...)
212         return rpn
213     }
214 
215     if node.Type == "Factor" {
216         if len(node.Children) == 3 {
217             return ASTToRPN(node.Children[1])
218         } else if len(node.Children) == 1 {
219             return ASTToRPN(node.Children[0])
220         }
221     }
222 
223     if node.Value != "" {
224         return []string{node.Value}
225     }
226 
227     for _, child := range node.Children {
228         rpn = append(rpn, ASTToRPN(child)...)
229     }
230 
231     return rpn
232 }
233 
234 func ASTToRPN1(node *ASTNode) ([]string, *ASTNode) {
235     if node == nil {
236         return nil, nil
237     }
238 
239     switch node.Type {
240     case "Expression", "ArithExpr", "Term":
241         var rpn []string
242         var rpnNode *ASTNode
243         var childrenNodes []*ASTNode
```

```
244
245         for _, child := range node.Children {
246             childRPN, childNode := ASTToRPN1(child)
247             rpn = append(rpn, childRPN...)
248             if childNode != nil {
249                 childrenNodes = append(childrenNodes, childNode)
250             }
251         }
252
253         // Для RPN дерева создаем узел-оператор с операндами в качестве детей
254         if len(childrenNodes) > 0 && node.Value != "" {
255             rpnNode = &ASTNode{
256                 Type:  node.Type + "_RPN",
257                 Value: node.Value,
258             }
259             for _, child := range childrenNodes {
260                 rpnNode.Children = append(rpnNode.Children, child)
261             }
262         } else if len(childrenNodes) > 0 {
263             rpnNode = &ASTNode{
264                 Type:     node.Type + "_RPN",
265                 Children: childrenNodes,
266             }
267         }
268
269         return rpn, rpnNode
270
271     case "Factor", "Identifier", "Number":
272         return []string{node.Value}, &ASTNode{
273             Type:  node.Type + "_RPN",
274             Value: node.Value,
275             Token: node.Token,
276         }
277
278     case "AddOp", "MulOp", "RelOp":
279         return []string{node.Value}, &ASTNode{
280             Type:  node.Type + "_RPN",
281             Value: node.Value,
282             Token: node.Token,
283         }
284
285     default:
286         return nil, nil
287     }
288 }
```

Листинг 2.5 – Код модуля *parser*

```
1 package parser
```

```
2
3  import (
4      "fmt"
5      "strings"
6  )
7
8  func GenerateDOT(node *ASTNode) string {
9      var builder strings.Builder
10     builder.WriteString("digraph␣AST␣{\n")
11     builder.WriteString("␣␣node␣[shape=box,␣fontname=\"Courier\",␣
           fontsize=10];\n")
12     builder.WriteString("␣␣edge␣[fontname=\"Courier\",␣fontsize=10];\n\n")
13
14     var nodeCounter int
15     generateDOTNode(&builder, node, &nodeCounter)
16
17     builder.WriteString("}\n")
18     return builder.String()
19 }
20
21 func generateDOTNode(builder *strings.Builder, node *ASTNode, counter
       *int) int {
22     if node == nil {
23         return -1
24     }
25
26     currentID := *counter
27     *counter++
28
29     label := node.Type
30     if node.Value != "" {
31         label += fmt.Sprintf("\\n%s", node.Value)
32     }
33
34     builder.WriteString(fmt.Sprintf("␣␣node%d␣[label=\"%s\"];\n",
           currentID, label))
35
36     for _, child := range node.Children {
37         childID := generateDOTNode(builder, child, counter)
38         if childID >= 0 {
39             builder.WriteString(fmt.Sprintf("␣␣node%d␣->␣node%d;\n",
                   currentID, childID))
40         }
41     }
42
43     return currentID
44 }
```

Листинг 2.6 – Код модуля *fs*

```
package fs

import (
    "bufio"
    "fmt"
    "os"
    "os/exec"
    "strings"
)

func ReadProgramFile(filename string) (string, error) {
    file, err := os.Open(filename)
    if err != nil {
        return "", fmt.Errorf("ошибка открытия файла: %w", err)
    }
    defer file.Close()

    scanner := bufio.NewScanner(file)
    var lines []string
    for scanner.Scan() {
        lines = append(lines, scanner.Text())
    }
    input := strings.Join(lines, "\n")
    return input, nil
}

func SaveDOTToFile(dot, filenameDOT, filenamePNG string) error {
    file, err := os.Create(filenameDOT)
    if err != nil {
        return err
    }
    defer file.Close()

    _, err = file.WriteString(dot)
    if err != nil {
        return err
    }
    cmd := exec.Command("dot", "-Tpng", "-o", filenamePNG, filenameDOT)
    if _, err := cmd.CombinedOutput(); err != nil {
        return err
    }

    return nil
}
```

3 Контрольные вопросы

3.1 Что такое операторная грамматика?

Операторная грамматика — КС-грамматика без ϵ-правил, в которой правые части всех правил не содержат смежных нетерминальных символов

3.2 Что такое грамматика операторного предшествования?

Операторная грамматика G называется грамматикой операторного предшествования, если между любыми двумя терминальными символами выполняется не более одного отношения операторного предшествования.

3.3 Как определяются отношения операторного предшествования?

1. $a \doteq b$, если $A \to \alpha a \gamma b \beta \in P$ и $\gamma \in N \cup \{\epsilon\}$.
2. $a \lessdot b$, если $A \to \alpha a B \beta \in P$ и $B \implies \gamma b \delta$, где $\gamma \in N \cup \{\epsilon\}$.
3. $a \gtrdot b$, если $A \to \alpha B b \beta \in P$ и $B \implies \delta a \gamma$, где $\gamma \in N \cup \{\epsilon\}$.
4. $\$ \lessdot a$, если $S \implies \gamma a \alpha$ и $\gamma \in N \cup \{\epsilon\}$.
5. $a \gtrdot \$$, если $S \implies \alpha a \gamma$ и $\gamma \in N \cup \{\epsilon\}$.

3.4 Как выделяется основа в процессе синтаксического разбора операторного предшествования?

Основу правовыводимой цепочки грамматики можно выделить, просматривая эту цепочку слева направо до тех пор, пока впервые не встретится отношение $\gtrdot$. Для нахождения левого конца основы надо возвращаться назад, пока не встретится отношение $\lessdot$. Цепочка, заключенная между $\lessdot$ и $\gtrdot$, будет основой. Если грамматика предполагается обратимой, то основу можно однозначно свернуть. Этот процесс продолжается до тех пор, пока входная цепочка не свернется к начальному символу (либо пока дальнейшие свертки окажутся невозможными).

3.5 Какие виды синтаксических ошибок не обнаруживаются в предложенном примере?

1. Ошибки, связанные с ограниченным размером контекста. Пример: если указать после последнего оператора в блоке невалидный идентификатор

вместо точки с запятой, то будет ошибка "отсутствует end блок".

2. Ошибки, связанные с неоднозначностью. Пример: если вместо валидной операции отношения указать неизвестный символ, то, будет ошибка, связанная с другим нетерминалом, а не с операцией отношения.
3. Пропуск множественных ошибок в одном выражении.
4. Ошибки, связанные с контекстом.

3.6 Какие действия надо предпринять для обнаружения всех синтаксических ошибок в предложенном примере?

1. Увеличить количество просматриваемых символов.
2. Ручное восстановление после ошибок.
3. Реализовать метод правого разбора.

3.7 Как сформулировать синтаксически управляемые определения для перевода инфиксного выражения в последовательность команд стековой машины?

Установить последовательность команд, которые будут установлены относительно правил грамматики. Например, для правила $E \to E + T$ может соответствовать команда `ADD;`, правилу $T \to T * F$ соответствует команда `MUL;`, правилу $F \to a$ соответствует команда `LOAD a;`.

3.8 Как сформулировать синтаксически управляемые определения для перевода инфиксного выражения в абстрактное синтаксическое дерево?

Правилам будет соответствовать создание узлов дерева. Например, для правила $E \to E+T$ может соответствовать команда создания узла `NewSumNode(node nodeT)`.