Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное 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 № 7

Студент ИУ7-22М (Группа) _____ (Подпись, дата) Е. О. Карпова (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 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>

Рисунок 1.1 – Блок в стиле Алгол-Паскаль

Грамматика по варианту:

<выражение> ->
<арифметическое выражение> <операция отношения> <арифметическое выражение> |
<арифметическое выражение>

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

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

<фактор> -> <идентификатор> |

<константа> |
(<арифметическое выражение>)

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

<операция типа сложения> -> +|-

<операция типа умножения> -> *|/

Рисунок 1.2 – Грамматика по варианту

Грамматика по варианту после удаления левой рекурсии и добавления блока:

```
<программа> -> <блок>
<блок> -> begin <список_операторов> end
<список_операторов> -> <оператор>
<список_операторов> -> <оператор> <список_операторов>'
<оператор> -> <идентификатор> = <выражение>
<выражение> -> <арифметическое_выражение> <операция_отношения>
<арифметическое_выражение>
<выражение> -> <арифметическое_выражение>
<арифметическое_выражение> -> <терм>
<арифметическое_выражение> -> <терм> <арифметическое_выражение>'
<терм> -> <фактор>
<терм> -> <фактор> <терм>'
<фактор> -> <идентификатор>
<фактор> -> <константа>
<фактор> -> ( <арифметическое_выражение> )
<операция_отношения> -> <
<операция_отношения> -> <=
<операция_отношения> -> ==
<операция_отношения> -> <>
<операция_отношения> -> >
<операция_отношения> -> >=
<операция_типа_сложения> -> +
<операция_типа_сложения> -> -
<операция_типа_умножения> -> *
<операция_типа_умножения> -> /
<список_операторов>' -> ; <оператор>
<список_операторов>' -> ; <оператор> <список_операторов>'
<арифметическое_выражение>' -> <операция_типа_сложения> <терм>
<арифметическое_выражение>' -> <операция_типа_сложения> <терм>
<арифметическое_выражение>'
<терм>' -> <операция_типа_умножения> <фактор>
<терм>' -> <операция_типа_умножения> <фактор> <терм>'
```

Рисунок 1.3 – Грамматика по варианту после удаления левой рекурсии и добавления блока

2 Практическая часть

2.1 Листинг

Листинг 2.1 – Исходный код моделей для AST

```
type Program struct {
    Block *Block
}

type Block struct {
    OperatorList *OperatorList
}

type OperatorList struct {
    Operator     *Operator
    OperatorList *OperatorListX
}

type Operator struct {
    Identifier *Identifier
    Expression *Expression
}

type Expression struct {
    Left               *ArithmeticalExpression
    RelationOperation *RelationOperation
    Right              *ArithmeticalExpression
}

type ArithmeticalExpression struct {
    Term                    *Term
    ArithmeticalExpression *ArithmeticalExpressionX
}

type Term struct {
    Factor *Factor
    Term   *TermX
}

type Factor struct {
    Identifier              *Identifier
```

```
    Constant                *Constant
    ArithmeticalExpression *ArithmeticalExpression
}

type RelationOperation struct {
    Value string
}

type SumOperation struct {
    Value string
}

type MulOperation struct {
    Value string
}

type OperatorListX struct {
    Operator     *Operator
    OperatorList *OperatorListX
}

type ArithmeticalExpressionX struct {
    SumOperation           *SumOperation
    Term                   *Term
    ArithmeticalExpression *ArithmeticalExpressionX
}

type TermX struct {
    MulOperation *MulOperation
    Factor       *Factor
    Term         *TermX
}

type Identifier struct {
    Value string
}

type Constant struct {
    Value string
}
```

Листинг 2.2 – Исходный код алгоритма рекурсивного спуска

```go
type parser struct {
    tokens  []token.Token
    pos     int
    current token.Token
}

func newParser(tokens []token.Token) *parser {
    return &parser{
        tokens:  tokens,
        pos:     0,
        current: tokens[0],
    }
}

func (p *parser) next() {
    p.pos++
    if p.pos < len(p.tokens) {
        p.current = p.tokens[p.pos]
    } else {
        p.current = token.Token{Type: token.EOF, Value: ""}
    }
}

func (p *parser) parseProgram() (*Program, error) {
    block, err := p.parseBlock()
    if err != nil {
        return nil, fmt.Errorf("invalid block: %w", err)
    }

    return &Program{
        Block: block,
    }, nil
}

func (p *parser) parseBlock() (*Block, error) {
    if p.current.Type != token.StartBlock {
        return nil, fmt.Errorf("no start block")
    }

    p.next()

```

```
42      operatorList, err := p.parseOperatorList()
43      if err != nil {
44          return nil, fmt.Errorf("invalid operator list: %w", err)
45      }
46
47      if p.current.Type != token.EndBlock {
48          return nil, fmt.Errorf("no end block")
49      }
50
51      p.next()
52
53      return &Block{
54          OperatorList: operatorList,
55      }, nil
56  }
57
58  func (p *parser) parseOperatorList() (*OperatorList, error) {
59      operator, err := p.parseOperator()
60      if err != nil {
61          return nil, fmt.Errorf("invalid operator: %w", err)
62      }
63
64      ol := &OperatorList{
65          Operator: operator,
66      }
67
68      if p.current.Type == token.EndBlock {
69          return ol, nil
70      }
71
72      operatorListX, err := p.parseOperatorListX()
73      if err != nil {
74          return nil, fmt.Errorf("invalid operator list x: %w",
                err)
75      }
76
77      ol.OperatorList = operatorListX
78
79      return ol, nil
80  }
81
```

```
82  func (p *parser) parseOperator() (*Operator, error) {
83      identifier, err := p.parseIdentifier()
84      if err != nil {
85          return nil, fmt.Errorf("invalid identifier: %w", err)
86      }
87
88      if p.current.Type != token.Assignment {
89          return nil, fmt.Errorf("no assign")
90      }
91
92      p.next()
93
94      expression, err := p.parseExpression()
95      if err != nil {
96          return nil, fmt.Errorf("invalid expression: %w", err)
97      }
98
99      return &Operator{
100         Identifier: identifier,
101         Expression: expression,
102     }, nil
103 }
104
105 func (p *parser) parseIdentifier() (*Identifier, error) {
106     if p.current.Type != token.Identifier {
107         return nil, fmt.Errorf("invalid identifier")
108     }
109
110     value := p.current.Value
111     p.next()
112
113     return &Identifier{
114         Value: value,
115     }, nil
116 }
117
118 func (p *parser) parseExpression() (*Expression, error) {
119     left, err := p.parseArithmeticalExpression()
120     if err != nil {
121         return nil, fmt.Errorf("invalid left arithmetical
              expression: %w", err)
```

```
122     }
123
124     expr := &Expression{
125         Left: left,
126     }
127
128     relation, err := p.parseRelationOperation()
129     if err != nil {
130         return expr, nil
131     }
132
133     expr.RelationOperation = relation
134
135     right, err := p.parseArithmeticalExpression()
136     if err != nil {
137         return nil, fmt.Errorf("invalid right arithmetical
              expression: %w", err)
138     }
139
140     expr.Right = right
141
142     return expr, nil
143 }
144
145 func (p *parser) parseRelationOperation() (*RelationOperation,
      error) {
146     switch p.current.Type {
147     case token.Less, token.LessOrEqual, token.Greater,
          token.GreaterOrEqual, token.Equal, token.NotEqual:
148         value := p.current.Value
149         p.next()
150         return &RelationOperation{
151             Value: value,
152         }, nil
153     default:
154         return nil, fmt.Errorf("invalid relation operation")
155     }
156 }
157
158 func (p *parser) parseArithmeticalExpression()
      (*ArithmeticalExpression, error) {
```

```
159     term, err := p.parseTerm()
160     if err != nil {
161         return nil, fmt.Errorf("invalid term: %w", err)
162     }
163 
164     ae := &ArithmeticalExpression{
165         Term: term,
166     }
167 
168     if p.current.Type != token.Plus && p.current.Type !=
        token.Minus {
169         return ae, nil
170     }
171 
172     aex, err := p.parseArithmeticalExpressionX()
173     if err != nil {
174         return nil, fmt.Errorf("invalid aex: %w", err)
175     }
176 
177     ae.ArithmeticalExpression = aex
178 
179     return ae, nil
180 }
181 
182 func (p *parser) parseTerm() (*Term, error) {
183     factor, err := p.parseFactor()
184     if err != nil {
185         return nil, fmt.Errorf("invalid factor: %w", err)
186     }
187 
188     term := &Term{
189         Factor: factor,
190     }
191 
192     if p.current.Type != token.Multiply && p.current.Type !=
        token.Divide {
193         return term, nil
194     }
195 
196     termX, err := p.parseTermX()
197     if err != nil {
```

```
198             return nil, fmt.Errorf("invalid term x: %w", err)
199         }
200
201         term.Term = termX
202
203         return term, nil
204     }
205
206     func (p *parser) parseFactor() (*Factor, error) {
207         f := &Factor{}
208
209         switch p.current.Type {
210         case token.Identifier:
211             identifier, err := p.parseIdentifier()
212             if err != nil {
213                 return nil, fmt.Errorf("invalid identifier: %w", err)
214             }
215
216             f.Identifier = identifier
217         case token.Constant:
218             constant, err := p.parseConstant()
219             if err != nil {
220                 return nil, fmt.Errorf("invalid constant: %w", err)
221             }
222
223             f.Constant = constant
224         case token.LeftParen:
225             p.next()
226
227             ae, err := p.parseArithmeticalExpression()
228             if err != nil {
229                 return nil, fmt.Errorf("invalid arithmetical
                        expression: %w", err)
230             }
231
232             if p.current.Type != token.RightParen {
233                 return nil, fmt.Errorf("no right paren")
234             }
235
236             p.next()
237
```

```
238             f.ArithmeticalExpression = ae
239         default:
240             return nil, fmt.Errorf("invalid factor content")
241         }
242
243         return f, nil
244 }
245
246 func (p *parser) parseConstant() (*Constant, error) {
247         if p.current.Type != token.Constant {
248             return nil, fmt.Errorf("invalid constant")
249         }
250
251         value := p.current.Value
252         p.next()
253
254         return &Constant{
255             Value: value,
256         }, nil
257 }
258
259 func (p *parser) parseOperatorListX() (*OperatorListX, error) {
260         if p.current.Type != token.Semicolon {
261             return nil, fmt.Errorf("no semicolon")
262         }
263
264         p.next()
265
266         operator, err := p.parseOperator()
267         if err != nil {
268             return nil, fmt.Errorf("invalid operator: %w", err)
269         }
270
271         olx := &OperatorListX{
272             Operator: operator,
273         }
274
275         if p.current.Type != token.Semicolon {
276             return olx, nil
277         }
278
```

```
279     operatorListX, err := p.parseOperatorListX()
280     if err != nil {
281         return nil, fmt.Errorf("invalid operator list x: %w",
            err)
282     }
283
284     olx.OperatorList = operatorListX
285
286     return olx, nil
287 }
288
289 func (p *parser) parseArithmeticalExpressionX()
    (*ArithmeticalExpressionX, error) {
290     aex := &ArithmeticalExpressionX{}
291
292     sum, err := p.parseSumOperation()
293     if err != nil {
294         return nil, fmt.Errorf("invalid sum operation: %w", err)
295     }
296
297     aex.SumOperation = sum
298
299     term, err := p.parseTerm()
300     if err != nil {
301         return nil, fmt.Errorf("invalid term: %w", err)
302     }
303
304     aex.Term = term
305
306     if p.current.Type != token.Plus && p.current.Type !=
        token.Minus {
307         return aex, nil
308     }
309
310     aexIn, err := p.parseArithmeticalExpressionX()
311     if err != nil {
312         return nil, fmt.Errorf("invalid aex: %w", err)
313     }
314
315     aex.ArithmeticalExpression = aexIn
316
```

```
317     return aex, nil
318 }
319
320 func (p *parser) parseSumOperation() (*SumOperation, error) {
321     so := &SumOperation{}
322
323     switch p.current.Type {
324     case token.Plus:
325     case token.Minus:
326     default:
327         return nil, fmt.Errorf("unknown sum operation")
328     }
329
330     so.Value = p.current.Value
331
332     p.next()
333
334     return so, nil
335 }
336
337 func (p *parser) parseTermX() (*TermX, error) {
338     termX := &TermX{}
339
340     mul, err := p.parseMulOperation()
341     if err != nil {
342         return nil, fmt.Errorf("invalid mul operation: %w", err)
343     }
344
345     termX.MulOperation = mul
346
347     factor, err := p.parseFactor()
348     if err != nil {
349         return nil, fmt.Errorf("invalid tefactorrm: %w", err)
350     }
351
352     termX.Factor = factor
353
354     if p.current.Type != token.Multiply && p.current.Type !=
        token.Divide {
355         return termX, nil
356     }
```

```
357
358     termIn, err := p.parseTermX()
359     if err != nil {
360         return nil, fmt.Errorf("invalid term x: %w", err)
361     }
362
363     termX.Term = termIn
364
365     return termX, nil
366 }
367
368 func (p *parser) parseMulOperation() (*MulOperation, error) {
369     mo := &MulOperation{}
370
371     switch p.current.Type {
372     case token.Multiply:
373     case token.Divide:
374     default:
375         return nil, fmt.Errorf("unknown multiply operation")
376     }
377
378     mo.Value = p.current.Value
379
380     p.next()
381
382     return mo, nil
383 }
```

2.2 Результаты выполнения программы

Исходная программа:

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

Рисунок 2.1 – Исходная программа

Построенное дерево:

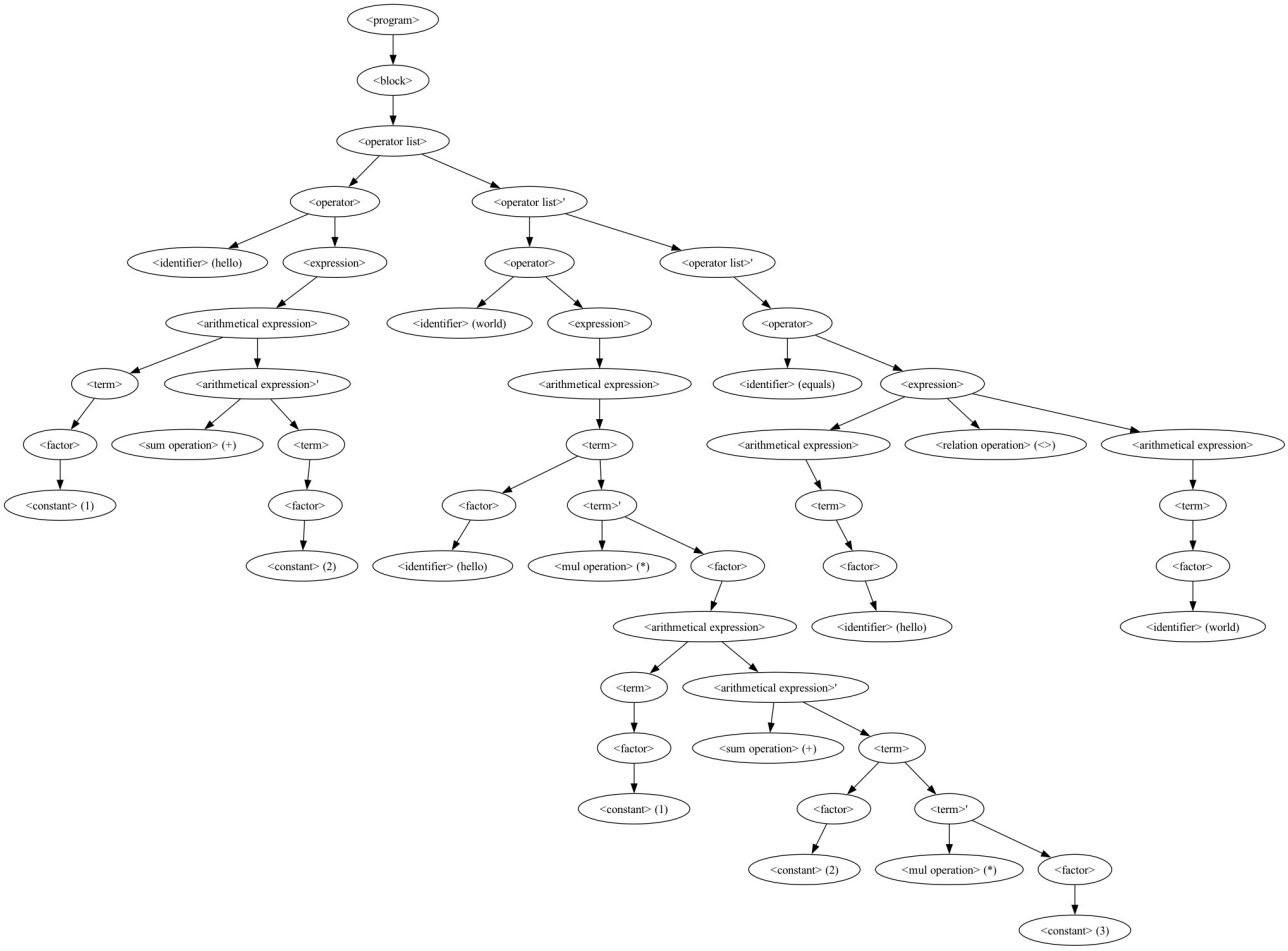

Рисунок 2.2 – Построенное дерево