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 №4

по курсу «Конструирование компиляторов»

на тему: «Синтаксический управляемый перевод»

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 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> -> <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> -> /
<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 1.1 – Грамматика по варианту для выражений

2 Практическая часть

2.1 Листинг

Листинг 2.1 – Исходный код моделей для AST и функций формирования постфиксной записи для поддерева

```
type Node interface {
    draw(file *os.File) error
    Pass() []string
}

func DepthPass(root Node) []string {
    return root.Pass()
}

type Expression struct {
    Left               *ArithmeticalExpression
    RelationOperation  *RelationOperation
    Right              *ArithmeticalExpression
    Postfix            []string
}

func (e *Expression) Pass() []string {
    if e == nil {
        return nil
    }

    postfix := e.Left.Pass()
    postfix = append(postfix, e.Right.Pass()...)
    postfix = append(postfix, e.RelationOperation.Pass()...)

    e.Postfix = postfix

    return postfix
}

type ArithmeticalExpression struct {
    Term                   *Term
    ArithmeticalExpression *ArithmeticalExpressionX
    Postfix                []string
}
```

```
36
37 func (e *ArithmeticalExpression) Pass() []string {
38     if e == nil {
39         return nil
40     }
41
42     postfix := e.Term.Pass()
43     postfix = append(postfix, e.ArithmeticalExpression.Pass()...)
44
45     e.Postfix = postfix
46
47     return postfix
48 }
49
50 type Term struct {
51     Factor  *Factor
52     Term    *TermX
53     Postfix []string
54 }
55
56 func (t *Term) Pass() []string {
57     if t == nil {
58         return nil
59     }
60
61     postfix := t.Factor.Pass()
62     postfix = append(postfix, t.Term.Pass()...)
63
64     t.Postfix = postfix
65
66     return postfix
67 }
68
69 type Factor struct {
70     Identifier             *Identifier
71     Constant               *Constant
72     ArithmeticalExpression *ArithmeticalExpression
73     Postfix                []string
74 }
75
76 func (f *Factor) Pass() []string {
```

```
77      if f == nil {
78          return nil
79      }
80
81      postfix := f.Identifier.Pass()
82      postfix = append(postfix, f.Constant.Pass()...)
83      postfix = append(postfix, f.ArithmeticalExpression.Pass()...)
84
85      f.Postfix = postfix
86
87      return postfix
88  }
89
90  type RelationOperation struct {
91      Value   string
92      Postfix []string
93  }
94
95  func (r *RelationOperation) Pass() []string {
96      if r == nil {
97          return nil
98      }
99
100     r.Postfix = []string{r.Value}
101
102     return r.Postfix
103 }
104
105 type SumOperation struct {
106     Value   string
107     Postfix []string
108 }
109
110 func (r *SumOperation) Pass() []string {
111     if r == nil {
112         return nil
113     }
114
115     r.Postfix = []string{r.Value}
116
117     return r.Postfix
```

```
118 }
119
120 type MulOperation struct {
121     Value   string
122     Postfix []string
123 }
124
125 func (r *MulOperation) Pass() []string {
126     if r == nil {
127         return nil
128     }
129
130     r.Postfix = []string{r.Value}
131
132     return r.Postfix
133 }
134
135 type ArithmeticalExpressionX struct {
136     SumOperation           *SumOperation
137     Term                   *Term
138     ArithmeticalExpression *ArithmeticalExpressionX
139     Postfix                []string
140 }
141
142 func (r *ArithmeticalExpressionX) Pass() []string {
143     if r == nil {
144         return nil
145     }
146
147     postfix := r.Term.Pass()
148     postfix = append(postfix, r.ArithmeticalExpression.Pass()...)
149     postfix = append(postfix, r.SumOperation.Pass()...)
150
151     r.Postfix = postfix
152
153     return postfix
154 }
155
156 type TermX struct {
157     MulOperation *MulOperation
158     Factor       *Factor
```

```
159     Term          *TermX
160     Postfix       []string
161 }
162
163 func (r *TermX) Pass() []string {
164     if r == nil {
165         return nil
166     }
167
168     postfix := r.Factor.Pass()
169     postfix = append(postfix, r.MulOperation.Pass()...)
170     postfix = append(postfix, r.Term.Pass()...)
171
172     r.Postfix = postfix
173
174     return postfix
175 }
176
177 type Identifier struct {
178     Value   string
179     Postfix []string
180 }
181
182 func (r *Identifier) Pass() []string {
183     if r == nil {
184         return nil
185     }
186
187     r.Postfix = []string{r.Value}
188
189     return r.Postfix
190 }
191
192 type Constant struct {
193     Value   string
194     Postfix []string
195 }
196
197 func (r *Constant) Pass() []string {
198     if r == nil {
199         return nil
```

```
200     }
201
202     r.Postfix = []string{r.Value}
203
204     return r.Postfix
205 }
```

2.2 Результаты выполнения программы

Исходная программа:

$hello * (2 - (3 + 5)/7) > 0$

Построенное дерево:

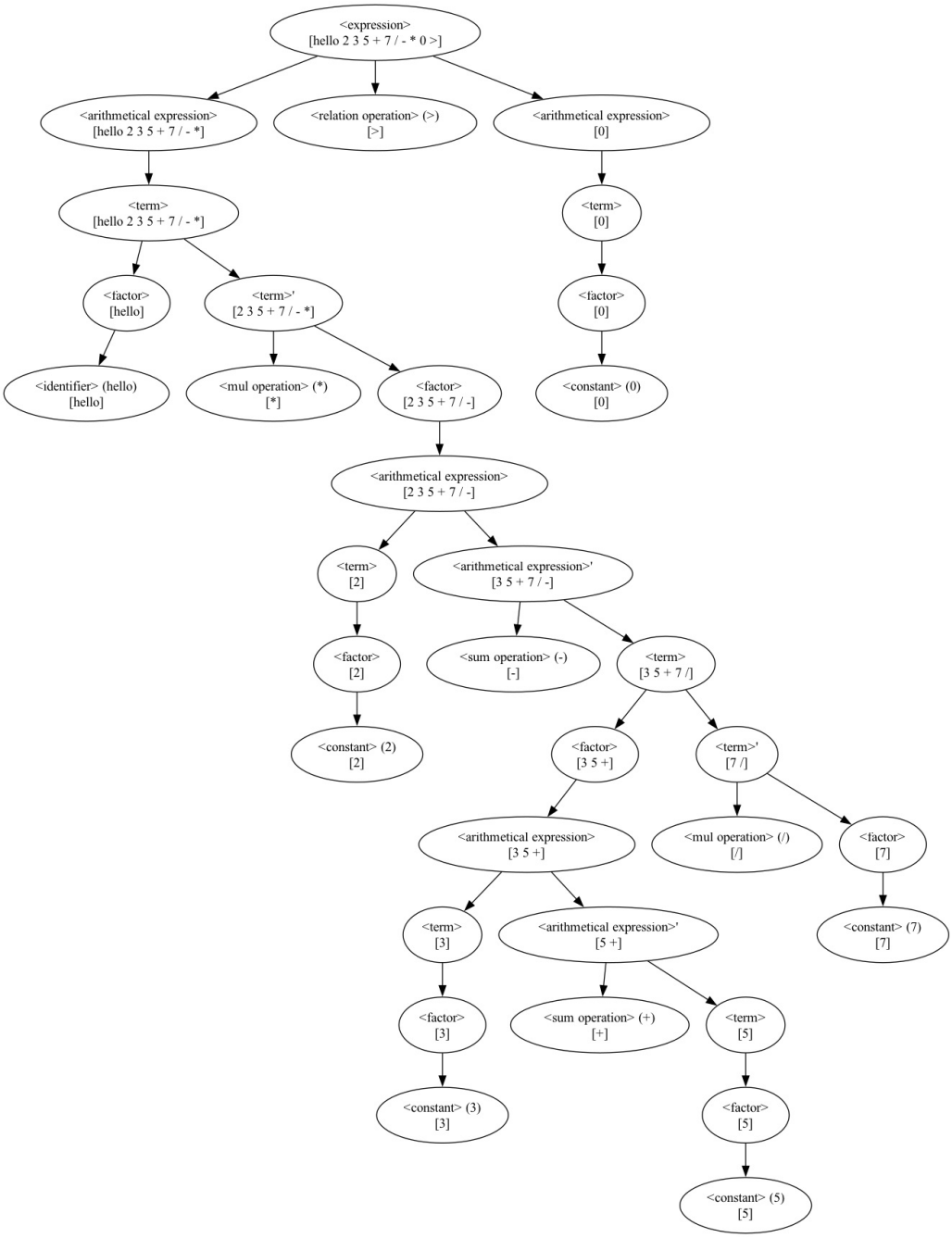

Рисунок 2.1 – Построенное дерево

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(nodeE, nodeT)`.