ГУАП

КАФЕДРА № 44

ОТЧЕТ
ЗАЩИЩЕН С ОЦЕНКОЙ

ПРЕПОДАВАТЕЛЬ

канд. техн. наук, доцент		Н.В. Кучин
должность, уч. степень, звание	подпись, дата	инициалы, фамилия

ОТЧЕТ О ЛАБОРАТОРНОЙ РАБОТЕ №2

ПОСТРОЕНИЕ СИНТАКСИЧЕСКОГО АНАЛИЗАТОРА И ПРОСТЕЙШЕГО ДЕРЕВА ВЫВОДА

по курсу: Системное программное обеспечение

РАБОТУ ВЫПОЛНИЛ

СТУДЕНТ гр. № 4142		К.С. Некрасов
	подпись, дата	инициалы, фамилия

Санкт-Петербург 2024

Цель работы:

Изучение основных понятий теории грамматик простого и операторного предшествования, ознакомление с алгоритмами синтаксического анализа (разбора) для некоторых классов КС-грамматик, получение практических навыков создания простейшего синтаксического анализатора для заданной грамматики операторного предшествования.

Получение практических навыков создания простейшего синтаксического анализатора для заданной грамматики операторного предшествования, обработка и представление результатов синтаксического анализа.

Задание

Требуется написать программу, которая выполняет лексический анализ входного текста в соответствии с заданием, порождает таблицу лексем и выполняет синтаксический разбор текста по заданной грамматике. Текст на входном языке задается в виде символьного (текстового) файла. Допускается исходить из условия, что текст содержит не более одного предложения входного языка.

Индивидуальное задание

Вариант: 14
Вариант грамматики: 2

$$S \to \textbf{a} := F;$$

$$F \to F \text{ or } T \mid F \text{ xor } T \mid T$$

$$T \to T \text{ and } E \mid E$$

$$E \to (F) \mid \text{not } (F) \mid a$$

Терминальные символы: **a**, **or**, **xor**, **and**, **not**, **(**, **)**

Выполнение задания

Построение левых и правых множеств:

Левые

1. Шаг 1
    1. L(S): a

2. L(F): F T
3. L(T): T E
4. L(E): (not a

2. Результат

    1. L(S): a
    2. L(F): F T E (not a
    3. L(T): T E (not a
    4. L(E): (not a

3. Терминальные

    1. L'(S): a
    2. L'(F): or xor and (not a
    3. L'(T): and (not a
    4. L'(E): (not a

Правые

1. Шаг 1

    1. R(S): ;
    2. R(F): T
    3. R(T): E
    4. R(E):) a

2. Результат

    1. R(S): ;
    2. R(F): T E) a
    3. R(T): E) a
    4. R(E):) a

3. Терминальные

    1. R'(S): ;
    2. R'(F): or xor and) a
    3. R'(T): and) a
    4. R'(E):) a

Матрица предшествования

Таблица 1 – Матрица предшествования

	a	:=	(	)	not	or	xor	and	;
a		=		>		>	>	>	>
:=	<		<	<	<	<	<	<	=
(	<		<	=	<	<	<	<	
)				>		>	>	>	>
not			=						
or	<		<	>	<	>	>	<	>
xor	<		<	>	<	>	>	<	>
and	<		<	>	<	>	>	>	>
;									

Пример разбора простейшего предложения

Предложение: `a := 0×FFF and (b or c);`

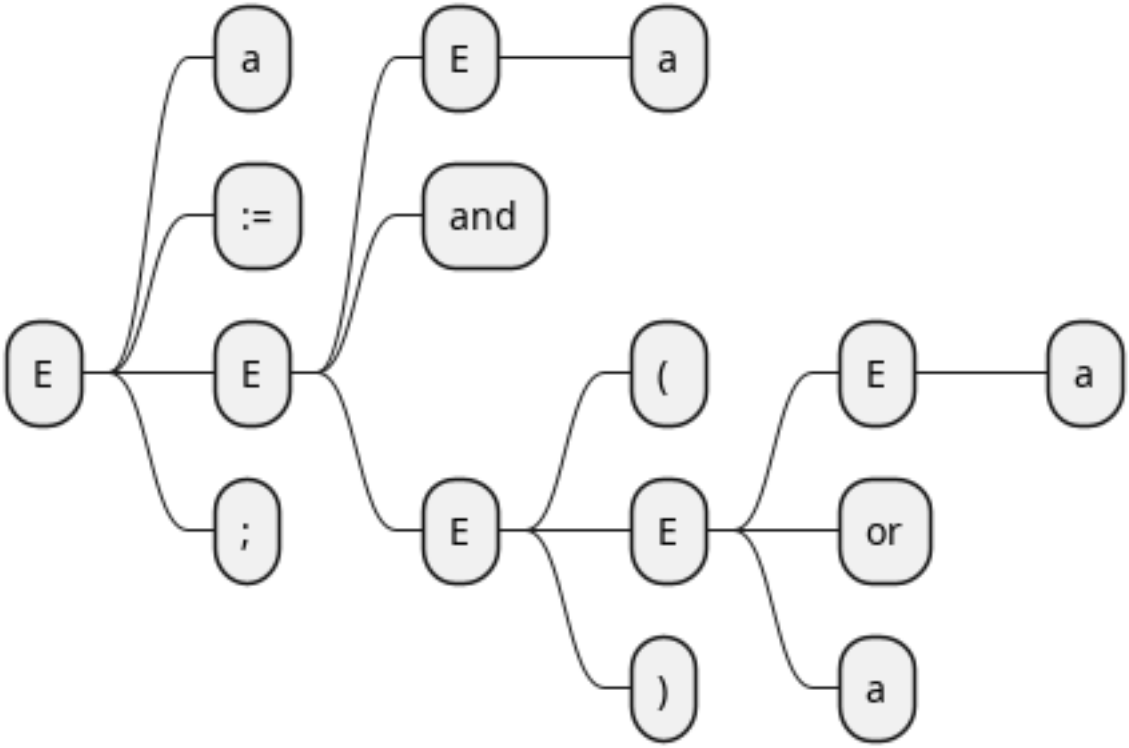
Таблица 2 – Пример разбора простейшего предложения

Входная строка	Стек	Действие
a := a and (a or a); к	н	п
:= a and (a or a); к	н a	п
a and (a or a); к	н a:=	п
and (a or a); к	н a:=a	с
and (a or a); к	н a:=E	п
(a or a); к	н a:=E and	п
a or a); к	н a:=E and (	п
or a); к	н a:=E and (a	п
or a); к	н a:=E and (E	с
a); к	н a:=E and (E or	п
); к	н a:=E and (E or a	п
); к	н a:=E and (E	с
; к	н a:=E and (E)	с
; к	н a:=E and E	с
к	н a:=E;	с
к	н E	-

Построение дерева вывода для простейшего примера

Предложение: `a := 0×FFF and (b or c);`

E -> a :=E; -> a :=E and E; -> a :=E and (E); -> a := E and (E or a); -> a := E and (a or a); -> a := a and (a or a);



Текст программы

```
package main

import (
        "fmt"
        "os"

        "goodhumored/lr2_syntax_analyzer/syntax_analyzer"
        "goodhumored/lr2_syntax_analyzer/syntax_analyzer/nonterminal"
        "goodhumored/lr2_syntax_analyzer/syntax_analyzer/precedence"
        "goodhumored/lr2_syntax_analyzer/syntax_analyzer/rule"
        "goodhumored/lr2_syntax_analyzer/token"
        "goodhumored/lr2_syntax_analyzer/token_analyzer"
)

// Правила грамматики
var rulesTable = rule.RuleTable{Rules: []rule.Rule{
        {Left: nonterminal.E, Right: []rule.Symbol{token.IdentifierType, token.Assig
        {Left: nonterminal.E, Right: []rule.Symbol{nonterminal.E, token.OrType, nont
```

```
        {Left: nonterminal.E, Right: []rule.Symbol{nonterminal.E, token.XorType, nor
        {Left: nonterminal.E, Right: []rule.Symbol{nonterminal.E, token.AndType, nor
        {Left: nonterminal.E, Right: []rule.Symbol{token.NotType, token.LeftParenthT
        {Left: nonterminal.E, Right: []rule.Symbol{token.LeftParenthType, nontermina
        {Left: nonterminal.E, Right: []rule.Symbol{token.IdentifierType}},
}}

// Матрица предшествования
var precedenceMatrix = precedence.PrecedenceMatrix{
        token.IdentifierType:   map[token.TokenType]precedence.PrecedenceType{token.
        token.AssignmentType:   map[token.TokenType]precedence.PrecedenceType{token.
        token.LeftParenthType:  map[token.TokenType]precedence.PrecedenceType{token.
        token.RightParenthType: map[token.TokenType]precedence.PrecedenceType{token.
        token.NotType:          map[token.TokenType]precedence.PrecedenceType{token.
        token.OrType:           map[token.TokenType]precedence.PrecedenceType{token.
        token.XorType:          map[token.TokenType]precedence.PrecedenceType{token.
        token.AndType:          map[token.TokenType]precedence.PrecedenceType{token.
}

func main() {
        source := getInput("./input.txt") // читаем файл

        // выводим содержимое
        println("Содержимое входного файла:\n")
        fmt.Println(source)

        // запускаем распознание лексем
        tokenTable := token_analyzer.RecogniseTokens(source)

        // выводим лексемы
        fmt.Println("Таблица лексем:")
        fmt.Println(tokenTable)

        // Проверяем на ошибки
        if errors := tokenTable.GetErrors(); len(errors) > 0 {
                fmt.Printf("Во время лексического анализа было обнаружено: %d ошибок
                for _, error := range errors {
                        fmt.Printf("Неожиданный символ '%s'\n", error.Value)
                }
                return
```

```
        }

        // запускаем синтаксический анализатор
        tree, error := syntax_analyzer.AnalyzeSyntax(rulesTable, *tokenTable, preced
        if error ≠ nil {
                fmt.Printf("Ошибка при синтаксическом анализе строки: %s", error)
        } else {
                fmt.Println("Строка принята!!!")
                tree.Print()
        }
}

// Читает файл с входными данными, вызывает панику в случае неудачи
func getInput(path string) string {
        data, err := os.ReadFile(path)
        if err ≠ nil {
                panic(err)
        }
        return string(data)
}
```

```
package token_analyzer

import (
        "regexp"

        "goodhumored/lr2_syntax_analyzer/token"
)

// Вспомогатеьлная структура для установки соответствия шаблонов лексем
// с их фабричными функциями
type TokenPattern struct {
        Pattern *regexp.Regexp
        Type    func(string, token.Position) token.Token
}

// Массив соответствий шаблонов лексем
var tokenPatterns = []TokenPattern{
        {regex("or"), token.Or},
        {regex("xor"), token.Xor},
        {regex("and"), token.And},
```

```
        {regex("not"), token.Not},
        {regex("(0x|[0-9$])[0-9a-fA-F]+"), token.Identifier},
        {regex("[a-zA-Z][a-zA-Z0-9]+"), token.Identifier},
        {regex(":="), token.Assignment},
        {regex("#.*"), token.Comment},
        {regex("[("), token.LeftParenth},
        {regex("[)]"), token.RightParenth},
        {regex(";"), token.Delimiter},
}

// вспомогательная функция создающая объект регулярного выражения
// добавляющая в начале шаблона признак начала строки
func regex(pattern string) *regexp.Regexp {
        return regexp.MustCompile("^" + pattern)
}
```

```
package token_analyzer

import (
        "strings"

        "goodhumored/lr2_syntax_analyzer/token"
        "goodhumored/lr2_syntax_analyzer/token_table"
)

// Распознаёт токены в данной строке построчно и записывает в таблицу
func RecogniseTokens(source string) *token_table.TokenTable {
        tokenTable := &token_table.TokenTable{}
        tokenTable.Add(token.Start)
        for _, line := range strings.Split(source, "\n") {
                recogniseTokensLine(line, tokenTable)
        }
        tokenTable.Add(token.EOF)
        return tokenTable
}

// Распознаёт лексемы в данной строке и записывает в таблицу
func recogniseTokensLine(line string, tokenTable *token_table.TokenTable) {
        for {
                line = strings.Trim(line, " ") // обрезаем пробельные символы в стро
                if len(line) == 0 {            // если строка пустая - завершаем обр
```

```
                        return
                }
                nextToken := getNextToken(line)      // ищем очередную лексему
                tokenTable.Add(nextToken)            // добавляем лексему в таблицу
                line = line[nextToken.Position.End:] // вырезаем обработанную часть
        }
}

// Ищет очередную лексему в строке
func getNextToken(str string) token.Token {
        // проходим по всем шаблонам лексем
        for _, tokenPattern := range tokenPatterns {
                res := tokenPattern.Pattern.FindStringIndex(str)
                if res ≠ nil {
                        return tokenPattern.Type(str[res[0]:res[1]], token.Position{
                }
        }
        return token.Error(str[0:1], token.Position{0, 1})
}

package main

import (
        "fmt"
        "os"

        "goodhumored/lr2_syntax_analyzer/syntax_analyzer"
        "goodhumored/lr2_syntax_analyzer/syntax_analyzer/nonterminal"
        "goodhumored/lr2_syntax_analyzer/syntax_analyzer/precedence"
        "goodhumored/lr2_syntax_analyzer/syntax_analyzer/rule"
        "goodhumored/lr2_syntax_analyzer/token"
        "goodhumored/lr2_syntax_analyzer/token_analyzer"
)

// Правила грамматики
var rulesTable = rule.RuleTable{Rules: []rule.Rule{
        {Left: nonterminal.E, Right: []rule.Symbol{token.IdentifierType, token.Assig
        {Left: nonterminal.E, Right: []rule.Symbol{nonterminal.E, token.OrType, nont
        {Left: nonterminal.E, Right: []rule.Symbol{nonterminal.E, token.XorType, nor
        {Left: nonterminal.E, Right: []rule.Symbol{nonterminal.E, token.AndType, nor
        {Left: nonterminal.E, Right: []rule.Symbol{token.NotType, token.LeftParenthT
```

```
        {Left: nonterminal.E, Right: []rule.Symbol{token.LeftParenthType, nontermina
        {Left: nonterminal.E, Right: []rule.Symbol{token.IdentifierType}},
}}

// Матрица предшествования
var precedenceMatrix = precedence.PrecedenceMatrix{
        token.IdentifierType:   map[token.TokenType]precedence.PrecedenceType{token.
        token.AssignmentType:   map[token.TokenType]precedence.PrecedenceType{token.
        token.LeftParenthType:  map[token.TokenType]precedence.PrecedenceType{token.
        token.RightParenthType: map[token.TokenType]precedence.PrecedenceType{token.
        token.NotType:          map[token.TokenType]precedence.PrecedenceType{token.
        token.OrType:           map[token.TokenType]precedence.PrecedenceType{token.
        token.XorType:          map[token.TokenType]precedence.PrecedenceType{token.
        token.AndType:          map[token.TokenType]precedence.PrecedenceType{token.
}

func main() {
        source := getInput("./input.txt") // читаем файл

        // выводим содержимое
        println("Содержимое входного файла:\n")
        fmt.Println(source)

        // запускаем распознание лексем
        tokenTable := token_analyzer.RecogniseTokens(source)

        // выводим лексемы
        fmt.Println("Таблица лексем:")
        fmt.Println(tokenTable)

        // Проверяем на ошибки
        if errors := tokenTable.GetErrors(); len(errors) > 0 {
                fmt.Printf("Во время лексического анализа было обнаружено: %d ошибок
                for _, error := range errors {
                        fmt.Printf("Неожиданный символ '%s'\n", error.Value)
                }
                return
        }

        // запускаем синтаксический анализатор
```

```
        tree, error := syntax_analyzer.AnalyzeSyntax(rulesTable, *tokenTable, preced
        if error ≠ nil {
                fmt.Printf("Ошибка при синтаксическом анализе строки: %s", error)
        } else {
                fmt.Println("Строка принята!!!")
                tree.Print()
        }
}

// Читает файл с входными данными, вызывает панику в случае неудачи
func getInput(path string) string {
        data, err := os.ReadFile(path)
        if err ≠ nil {
                panic(err)
        }
        return string(data)
}
```

```
package token

import "fmt"

// Структура Position представляет положение лексемы в строке
type Position struct {
        Start int
        End   int
}

// Структура Token представляет лексему с ее типом и значением
type Token struct {
        Type     TokenType // Тип
        Value    string    // Значение
        Position Position  // Положение лексемы
}

// Функция получения имени токена, для соответствия интерфейсу символа
func (token Token) GetName() string {
        return token.Type.GetName()
}

// Фабричная функция для токенов, возвращающая замкнутую лямбда функцию для создания
```

```
func tokenFactory(tokenType TokenType) func(string, Position) Token {
        return func(value string, position Position) Token {
                return Token{
                        Value:    value,
                        Type:     tokenType,
                        Position: position,
                }
        }
}

// Функция определяющая как токен преобразуется в строку
func (token Token) String() string {
        return fmt.Sprintf("%s (%s)", token.Type, token.Value)
}

// Функции создания лексем определённых типов
var (
        Delimiter    = tokenFactory(DelimiterType)           // Разделиитель
        Identifier   = tokenFactory(IdentifierType)          // Идентификатор
        Assignment   = tokenFactory(AssignmentType)          // Присваивание
        And          = tokenFactory(AndType)                 // И
        Or           = tokenFactory(OrType)                  // Или
        Xor          = tokenFactory(XorType)                 // Исключающее или
        Not          = tokenFactory(NotType)                 // Не
        LeftParenth  = tokenFactory(LeftParenthType)         // Левая скобка
        RightParenth = tokenFactory(RightParenthType)        // Правая скобка
        Error        = tokenFactory(ErrorType)               // Ошибка
        Comment      = tokenFactory(CommentType)             // Комментарий
        Start        = Token{StartType, "", Position{0, 0}}  // Начало строки
        EOF          = Token{EOFType, "EOF", Position{0, 0}} // Конец
)

package token

type TokenType struct {
        Name string
}

func (tokenType TokenType) GetName() string {
        return tokenType.Name
}
```

```
var (
        DelimiterType    = TokenType{"delimiter"}         // Разделитель
        IdentifierType   = TokenType{"identifier"}        // Идентификатор
        HexType          = TokenType{"hex_number"}        // Шестнадцатиричное число
        AssignmentType   = TokenType{"assignment"}        // Присваивание
        AndType          = TokenType{"and"}               // and
        OrType           = TokenType{"or"}                // or
        XorType          = TokenType{"xor"}               // xor
        NotType          = TokenType{"not"}               // not
        LeftParenthType  = TokenType{"left_parentheses"}  // Скобки
        RightParenthType = TokenType{"right_parentheses"} // Скобки
        ErrorType        = TokenType{"error"}             // Ошибка
        CommentType      = TokenType{"comment"}           // Комментарий
        StartType        = TokenType{"start"}             // Начало
        EOFType          = TokenType{"EOF"}               // Конец
)
```

```
package token_table

import (
        "fmt"
        "strings"

        "goodhumored/lr2_syntax_analyzer/token"
)

// Таблица лексем
type TokenTable struct {
        tokens []token.Token
}

// Метод добавления лексемы в таблицу
func (tt *TokenTable) Add(token token.Token) {
        tt.tokens = append(tt.tokens, token)
}

// Метод получения списка найденных лексем
func (tt TokenTable) GetTokens() []token.Token {
        return tt.tokens
}
```

```
// Вспомогательная функция для вывода таблицы
func (tt *TokenTable) Print() {
        errors := tt.GetErrors()
        if len(errors) > 0 {
                errorMsg := ""
                for _, error := range errors {
                        errorMsg += fmt.Sprintf("Неизвестный символ: %s \n", error.V
                }
                fmt.Println(fmt.Errorf(errorMsg))
        }
        fmt.Println(tt.String())
}

// Вспомогательная функция для генерации строки с таблицей лексем
func (tt *TokenTable) String() string {
        if len(tt.tokens) == 0 {
                return "Ни одного токена не найдено"
        }

        // Определяем максимальную ширину столбца
        maxTypeLen := len("Тип")
        maxValueLen := len("Значение")
        for _, token := range tt.tokens {
                if len(token.Type.Name) > maxTypeLen {
                        maxTypeLen = len(token.Type.Name)
                }
                if len(token.Value) > maxValueLen {
                        maxValueLen = len(token.Value)
                }
        }

        // создаем шапку и рамки
        header := fmt.Sprintf("| %-*s | %-*s |", maxTypeLen, "Тип", maxValueLen, "Зн
        border := fmt.Sprintf("+-%s-+-%s-+", strings.Repeat("-", maxTypeLen), string

        // Собираем таблицу
        res := border + "\n" + header + "\n" + border + "\n"
        for _, token := range tt.tokens {
                res += fmt.Sprintf("| %-*s | %-*s |\n", maxTypeLen, token.GetName(),
```

```
        }
        res += border

        return res
}

// Функция возвращающая все ошибки в таблице
func (tt TokenTable) GetErrors() []token.Token {
        tokens := []token.Token{}
        for _, recognisedToken := range tt.tokens {
                if recognisedToken.Type == token.ErrorType {
                        tokens = append(tokens, recognisedToken)
                }
        }
        return tokens
}
```

```
package parse_tree

import (
        "fmt"

        "goodhumored/lr2_syntax_analyzer/syntax_analyzer/rule"
)

// Узел дерева вывода
type Node struct {
        Symbol   rule.Symbol
        Children []*Node
}

// Вспомогательная функция для создания пустого узла
func CreateNode(s rule.Symbol) Node {
        return Node{s, []*Node{}}
}

// Метод, добавляющий дочерний узел
func (n *Node) AddChild(child *Node) {
        n.Children = append(n.Children, child)
}
```

```
// Метод свёртки узла дерева
func (node *Node) Reduce(rule rule.Rule) bool {
	// Если не можем применить правило к текущему узлу - уходим
	if !node.CanApplyRule(rule) {
		return false
	}
	// считаем разницу длин правой части правила и детей узла
	lenDiff := len(node.Children) - len(rule.Right)

	// копируем слайс с нужными нам узлами, которые собираемся заменять
	nodes := make([]*Node, len(rule.Right))
	copy(nodes, node.Children[lenDiff:])

	// перезаписываем дочерние узлы узла
	node.Children = append(node.Children[:lenDiff], &Node{rule.Left, nodes})
	return true
}

// Функция проверки возможности применения правила к дочерним узлам узла
func (node Node) CanApplyRule(rule rule.Rule) bool {
	lenDiff := len(node.Children) - len(rule.Right)
	// Если в правиле больше элементов чем в узле - уходим
	if lenDiff < 0 {
		return false
	}
	// Проходимся по символам правила и сравниваем с дочерними символами
	for i, rule := range rule.Right {
		if rule.GetName() ≠ node.Children[i+lenDiff].Symbol.GetName() {
			return false
		}
	}
	return true
}

// Метод для рекурсивного вывода узлов дерева в консоль
func (node *Node) Print(prefix string, isTail bool) {
	// Выводим символ узла с отступом
	var branch, prefixSuffix string
	if isTail {
		prefixSuffix = "    "
```

```
                branch = "└── "
        } else {
                branch = "├── "
                prefixSuffix = "│   "
        }
        fmt.Println(prefix + branch + node.Symbol.GetName())

        // Рекурсивно выводим дочерние узлы
        for i := 0; i < len(node.Children)-1; i++ {
                node.Children[i].Print(prefix+prefixSuffix, false)
        }
        if len(node.Children) > 0 {
                node.Children[len(node.Children)-1].Print(prefix+prefixSuffix, true)
        }
}
```

```
package syntax_analyzer

import (
        "fmt"

        "goodhumored/lr2_syntax_analyzer/syntax_analyzer/rule"
        "goodhumored/lr2_syntax_analyzer/token"
)

// Стек символов
type symbolStack []rule.Symbol

// Добавление символа в стек
func (s symbolStack) Push(e rule.Symbol) symbolStack {
        return append(s, e)
}

// Просмотр верхнего элемента стека
func (s symbolStack) Peek() rule.Symbol {
        length := len(s)
        if length == 0 {
                return nil
        }
        return s[length-1]
}
```

```
// Просмотр n-ного элемента стека
func (s symbolStack) PeekN(n int) rule.Symbol {
        length := len(s)
        if length == 0 {
                return nil
        }
        return s[length-n-1]
}

// Поиск ближайшего к вершине терминала в стеке
func (s symbolStack) PeekNextTerminal() *token.Token {
        for i := range s {
                symbol := s.PeekN(i)
                if token, ok := symbol.(token.Token); ok {
                        return &token
                }
        }
        return nil
}

// Вспомогательный метод преобразования стека символов в строку
func (s symbolStack) String() string {
        str := ""
        for _, i := range s {
                str += fmt.Sprintf("%s ", i.GetName())
        }
        return str
}

// Вспомогательный метод вывода стека символов
func (s symbolStack) Print() {
        fmt.Print(s.String())
}
```

```
package rule

import (
        "fmt"

        "goodhumored/lr2_syntax_analyzer/syntax_analyzer/nonterminal"
```

```
)

// Интерфейс для представления символа
type Symbol interface {
        GetName() string
}

// Правило
type Rule struct {
        Left  nonterminal.NonTerminal
        Right []Symbol
}

// Метод получения строки из правила
func (r Rule) String() string {
        return fmt.Sprintf("%s → %s", r.Left.GetName(), r.Right)
}

package rule

// Таблица правил
type RuleTable struct {
        Rules []Rule
}

// Метод поиска правила по правой части
func (ruleTable RuleTable) GetRuleByRightSide(tokenTypes []Symbol) *Rule {
        for _, rule := range ruleTable.Rules {
                if isApplyable(rule.Right, tokenTypes) {
                        return &rule
                }
        }
        return nil
}

// Проверка на применимость правила к целевым символам
func isApplyable(ruleSymbols, targetSymbols []Symbol) bool {
        // Проверяем длины
        lenDiff := len(targetSymbols) - len(ruleSymbols)
        if lenDiff < 0 {
                return false
```

```
        }
        // Сравниваем последние символы цепочки символов и символы правила
        for i, ruleSymbol := range ruleSymbols {
                if ruleSymbol.GetName() ≠ targetSymbols[i+lenDiff].GetName() {
                        return false
                }
        }
        return true
}
```

```
package syntax_analyzer

import (
        "fmt"

        "goodhumored/lr2_syntax_analyzer/syntax_analyzer/nonterminal"
        "goodhumored/lr2_syntax_analyzer/syntax_analyzer/parse_tree"
        "goodhumored/lr2_syntax_analyzer/syntax_analyzer/precedence"
        "goodhumored/lr2_syntax_analyzer/syntax_analyzer/rule"
        "goodhumored/lr2_syntax_analyzer/token"
        "goodhumored/lr2_syntax_analyzer/token_table"
)

// Функция для анализа синтаксиса, принимает таблицу токенов, список правил и матриц
func AnalyzeSyntax(ruleTable rule.RuleTable, tokenTable token_table.TokenTable, matr
        // Создаём дерево
        rootNode := parse_tree.CreateNode(nonterminal.Null)
        tree := parse_tree.ParseTree{Root: &rootNode}
        // Получаем лексемы из таблицы
        tokens := tokenTable.GetTokens()
        tokenIndex := 1
        // Создаём стек
        stack := symbolStack{tokens[0]}

        for {
                // Берём ближайший к вершине терминал
                stackTerminal := stack.PeekNextTerminal()
                // Берём текущий символ входной строки
                inputToken := tokens[tokenIndex]
                // Если строка принята, значит возвращаем дерево вывода
                if isInputAccepted(inputToken, stack) {
```

```
                return tree, nil
        }
        // Если комментарий - пропускаем
        if inputToken.Type == token.CommentType {
                tokenIndex += 1
                continue
        }

        fmt.Printf("Лексема: '%s' \n", tokens[tokenIndex].Value)
        fmt.Printf("Стек: %s \n", stack)

        // Получаем предшествование из матрицы
        prec := matrix.GetPrecedence(stackTerminal.Type, inputToken.Type)

        // Если предшествование или =, тогда сдвигаем
        if prec == precedence.Lt || prec == precedence.Eq {
                print("Сдвигаем\n")
                tree.AddNode(&parse_tree.Node{Symbol: inputToken, Children:
                stack = stack.Push(inputToken)
                tokenIndex += 1
        } else if prec == precedence.Gt { // Иначе сворачиваем
                print("Сворачиваем\n")
                // сворачиваем стек
                newStack, rule, err := reduce(stack, ruleTable)
                if err != nil {
                        return tree, err
                }
                stack = newStack
                // сворачиваем дерево
                tree.Reduce(*rule)
        } else {
                // Если предшествование не определено - выдаем ошибку
                return tree, fmt.Errorf("Ошибка в синтексе, неожиданное соче
        }
        println("═══════════")
    }
}

// Проверка на завершённость
func isInputAccepted(currentToken token.Token, stack symbolStack) bool {
```

```
        nextTerminal := stack.PeekNextTerminal()
        nextSymbol := stack.Peek()
        return currentToken.Type == token.EOFType && // Если дошли до конца строки
                nextTerminal != nil &&
                nextTerminal.Type == token.Start.Type && // Если ближайший терминал
                nextSymbol != nil &&
                nextSymbol == nonterminal.E // А на вершине строки - целевой символ
}

// Функция свёртки стека
func reduce(stack symbolStack, ruleTable rule.RuleTable) (symbolStack, *rule.Rule, e
        for {
                // Если есть применимое к стеку правило
                if rule := ruleTable.GetRuleByRightSide(stack); rule != nil {
                        fmt.Printf("Нашлось правило: %v, пушим %s в стек\n", rule, r
                        // обновляем стек
                        stack = append(stack[:len(stack)-len(rule.Right)], rule.Left
                        return stack, rule, nil
                } else {
                        // Если нет выдаем ошибку
                        return stack, nil, fmt.Errorf("Не найдено правил для свёртки
                }
        }
}

package parse_tree

import (
        "fmt"

        "goodhumored/lr2_syntax_analyzer/syntax_analyzer/rule"
)

// Дерево вывода
type ParseTree struct {
        Root *Node
}

// Метод добавления узлов в дерево
func (tree *ParseTree) AddNode(node *Node) {
        tree.Root.AddChild(node)
```

```
}

// Метод для свёртки дерева по правилу
func (tree *ParseTree) Reduce(rule rule.Rule) {
        fmt.Printf("Применяем правило %s к дереву\n", rule)
        if tree.Root.Reduce(rule) {
                fmt.Printf("Успешно применено\n")
        } else {
                fmt.Printf("Правило %s применить не удалось\n", rule)
        }
}

// Метод для вывода дерева
func (tree ParseTree) Print() {
        tree.Root.Print("", true)
}
```

```
package nonterminal

// Структура представляющая нетерминалы
type NonTerminal struct {
        Name string
}

// Метод для соответствия нетерменалов интерфейсу символ
func (nt NonTerminal) GetName() string {
        return nt.Name
}

var (
        E    = NonTerminal{"E"} // Стандартный нетерминал
        Null = NonTerminal{"/"} // Корневой нетерминал
)
```

```
package precedence

// Тип предшествования
type PrecedenceType struct {
        Name string
}

// Типы предшествования
```

```
var (
        Lt        = PrecedenceType{"<"} // Предшествует
        Eq        = PrecedenceType{"="} // Составляет основу
        Gt        = PrecedenceType{">"} // Следует
        Undefined = PrecedenceType{"-"} // Неопределено
)
```

```
package precedence

import (
        "goodhumored/lr2_syntax_analyzer/token"
)

// Матрица предшествования
type PrecedenceMatrix map[token.TokenType]map[token.TokenType]PrecedenceType

// Метод для поиска типа предшествования для двух терминалов
func (matrix PrecedenceMatrix) GetPrecedence(left, right token.TokenType) Precedence
        // Если левый символ - начало файла, возвращаем предшествие
        if left == token.StartType {
                return Lt
        }
        // Если правый символ - конец файла, возвращаем следствие
        if right == token.EOFType {
                return Gt
        }
        // Если находится - возвращаем
        if val, ok := matrix[left]; ok {
                if precedence, ok := val[right]; ok {
                        return precedence
                }
        }
        // Если не находится - возвращаем неопределённость
        return Undefined
}
```

Примеры работы программы

Простое предложение

```
# Простой пример с одним идентификатором и числом
id1 := 0x1A;
```

Рисунок 1 – предложение 1

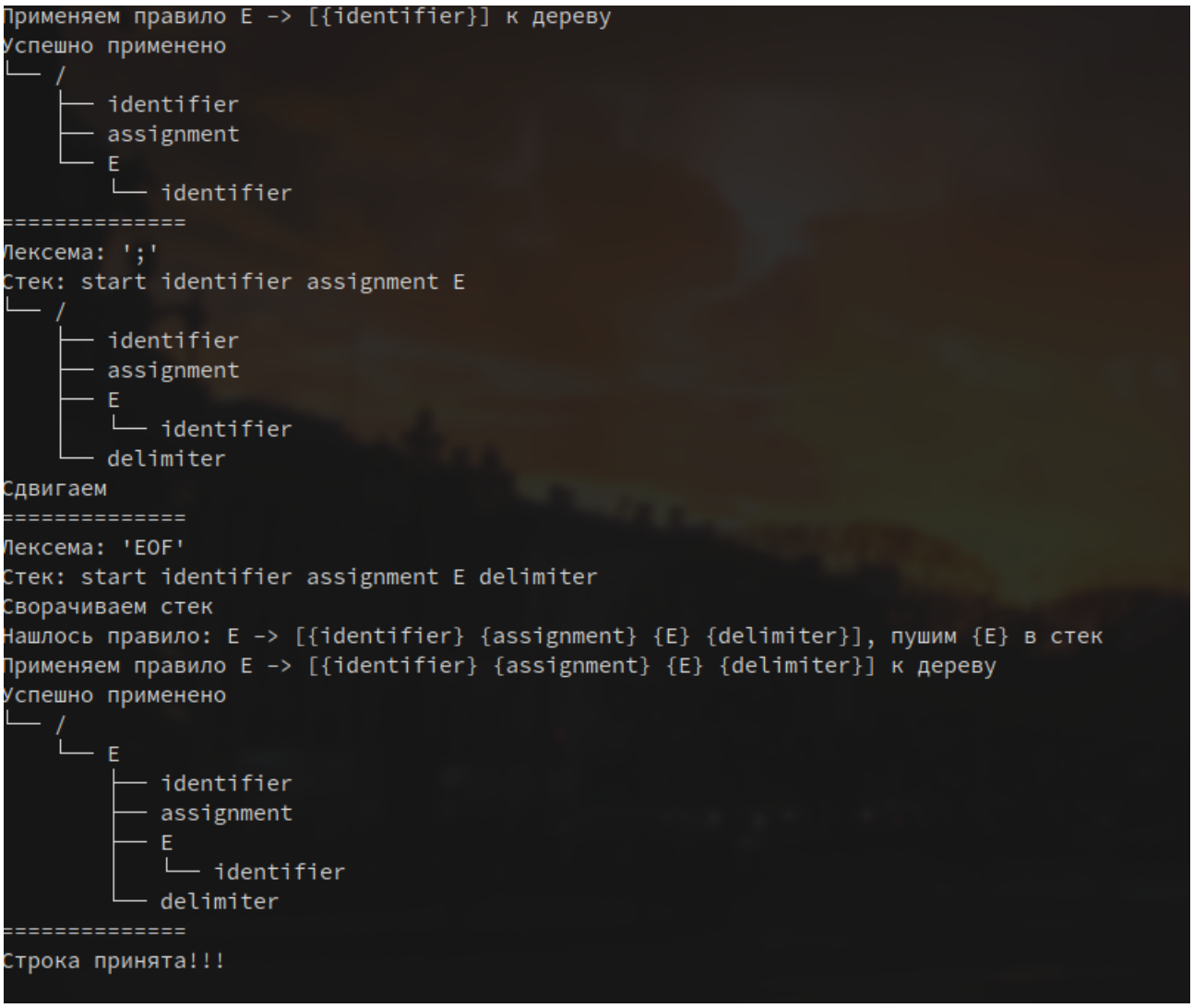


Рисунок 2 – предложение 1 продолжение

Сложное предложение

```
# Комментарий на отдельной строке
# Следующая строка содержит сложное выражение
complex := id13 and (not (id14) or (id15 xor 0×9A)) and 0×1B; # Комментарий на той же стр
# Комментарий снизу
```

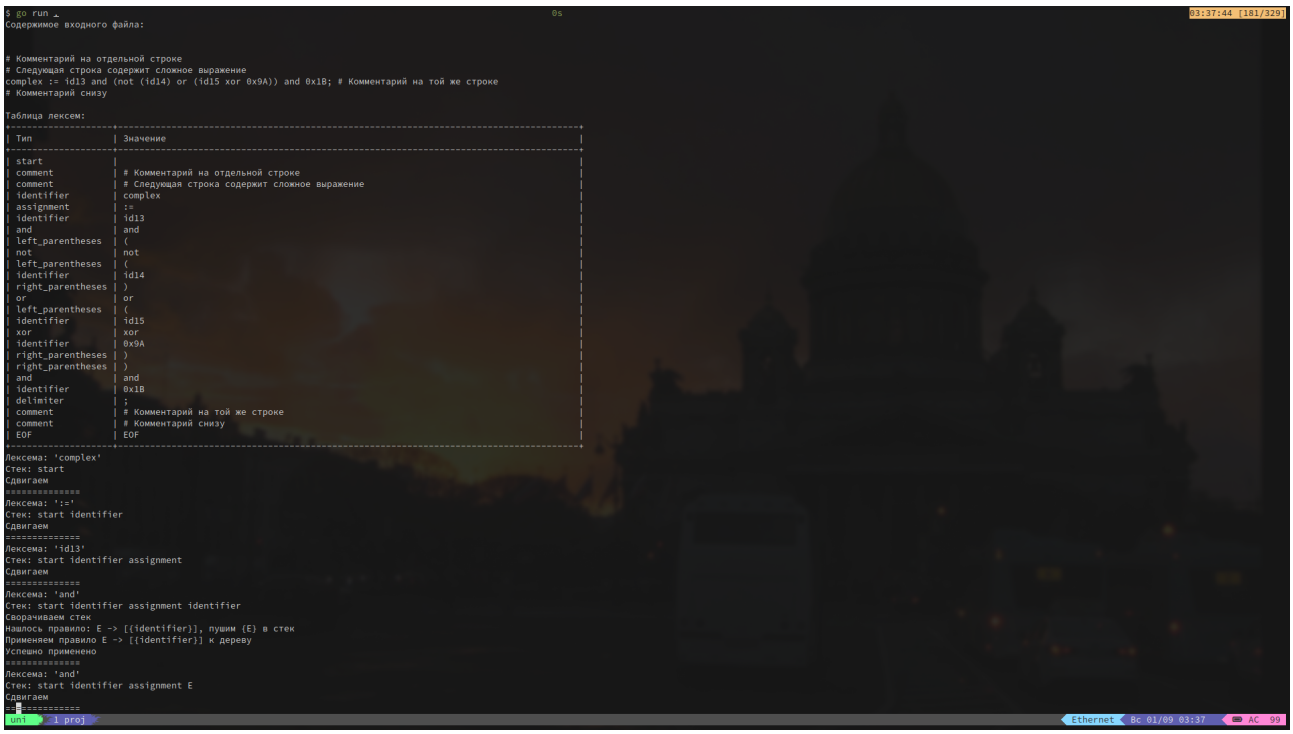


Рисунок 3 – предложение 2 часть 1

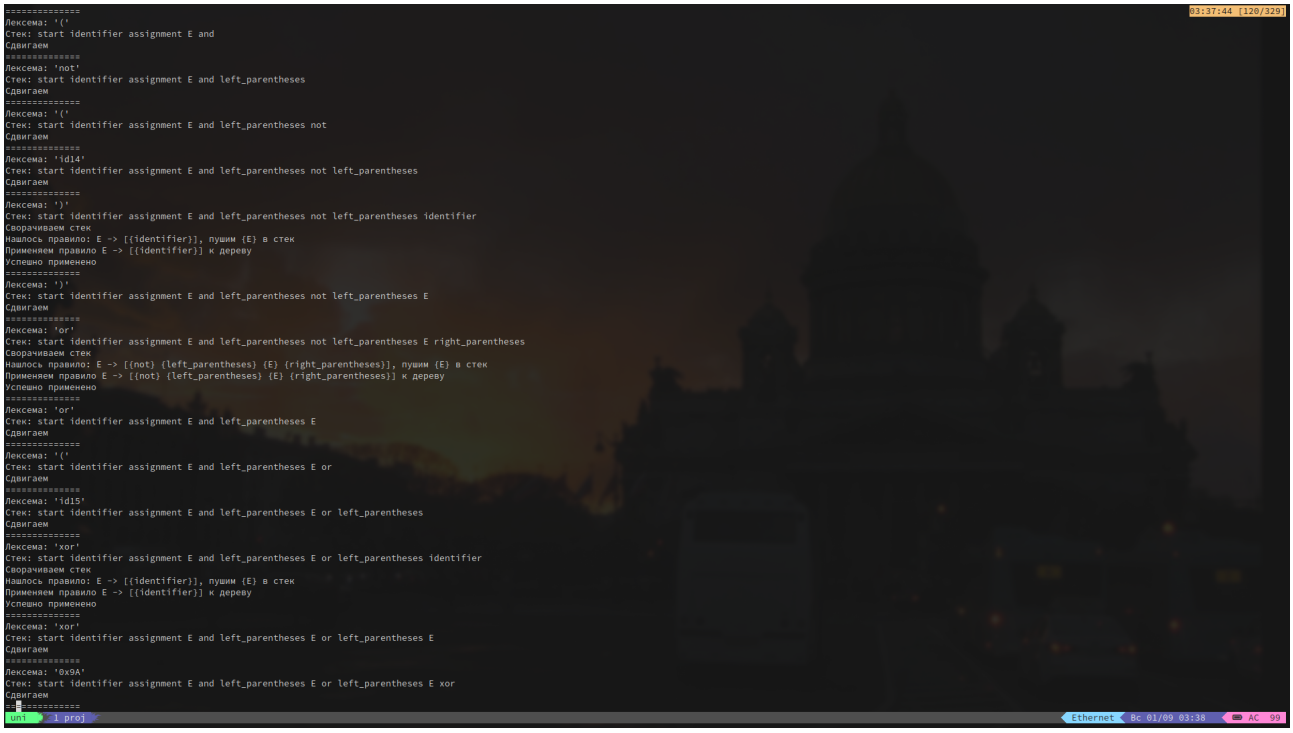


Рисунок 4 – предложение 2 часть 2

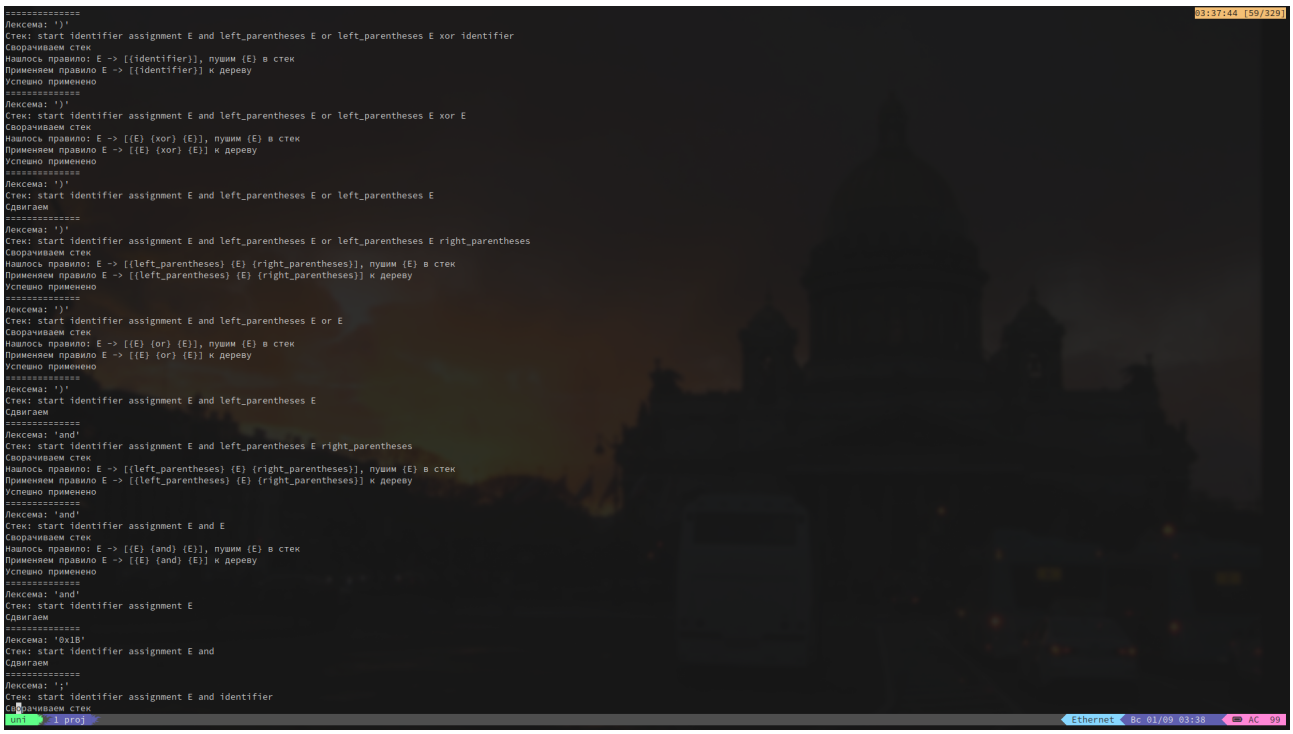


Рисунок 5 – предложение 2 часть 3

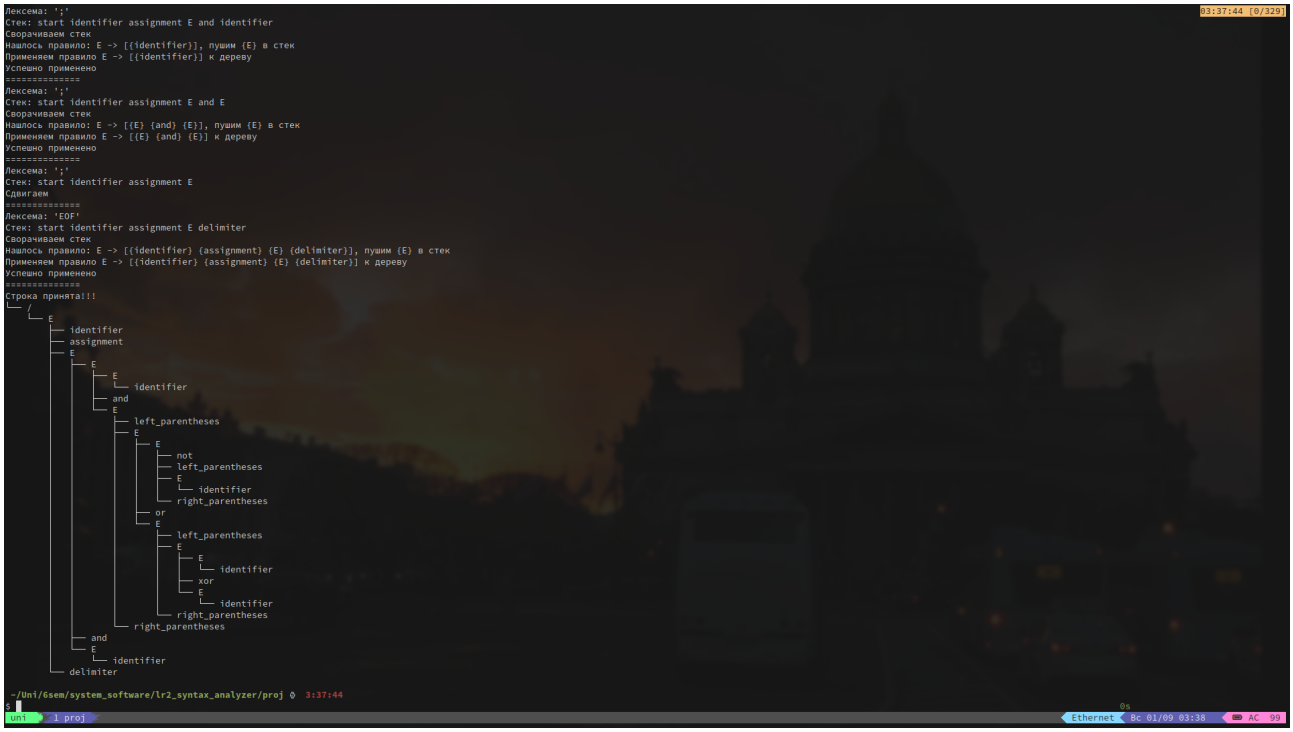


Рисунок 6 – предложение 2 часть 4

Примеры с ошибками

Ошибка на этапе лексического анализа

```
# Неправильный идентификатор
1invalid := 0×2C;
```

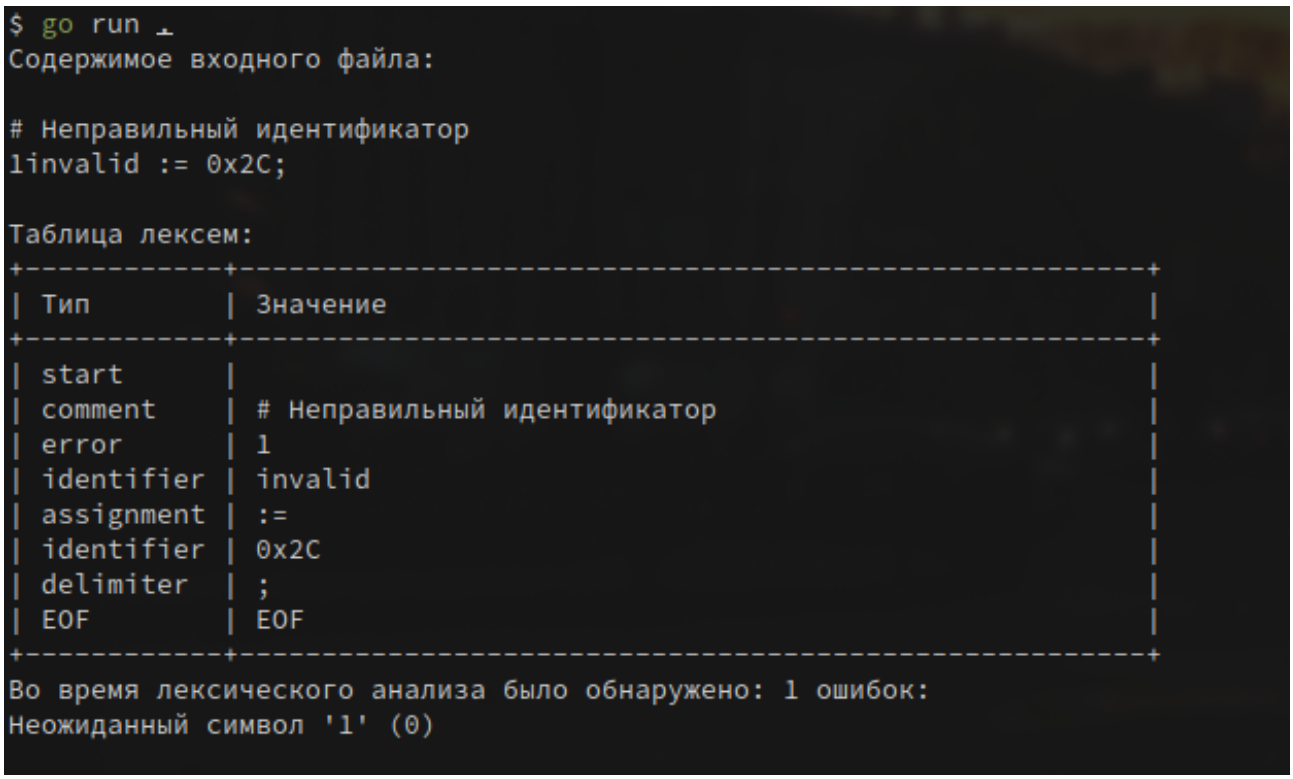


Рисунок 7 – лексическая ошибка пример 1

```
# Пример много ошибок
a := b and (not (1id14) orr (id15 xor 0×9A)) and0×1B;
```

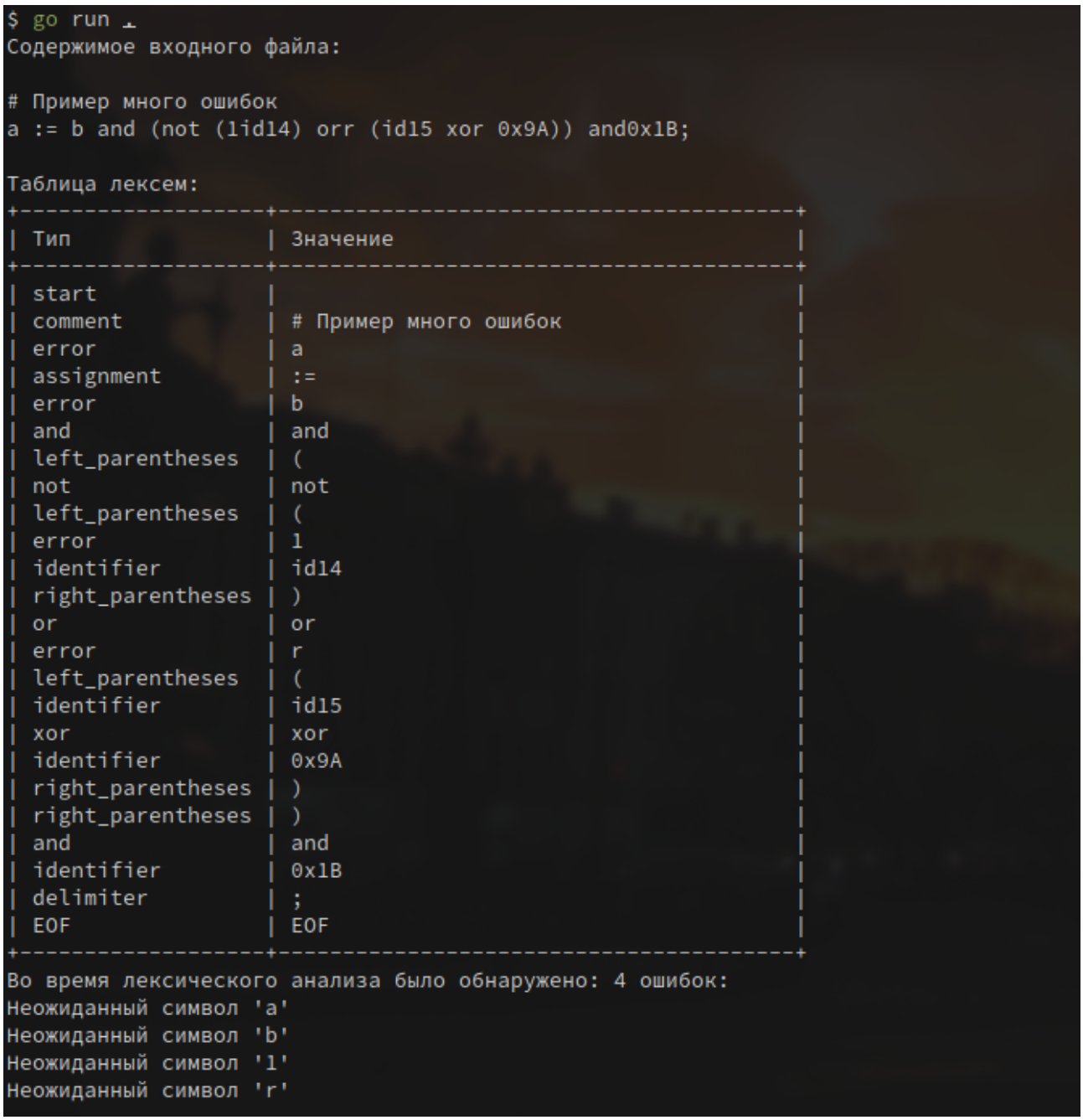


Рисунок 8 – лексическая ошибка пример 2

Ошибка на этапе синтаксического анализа

```
# Пример с пустым значением
empty := ;
```

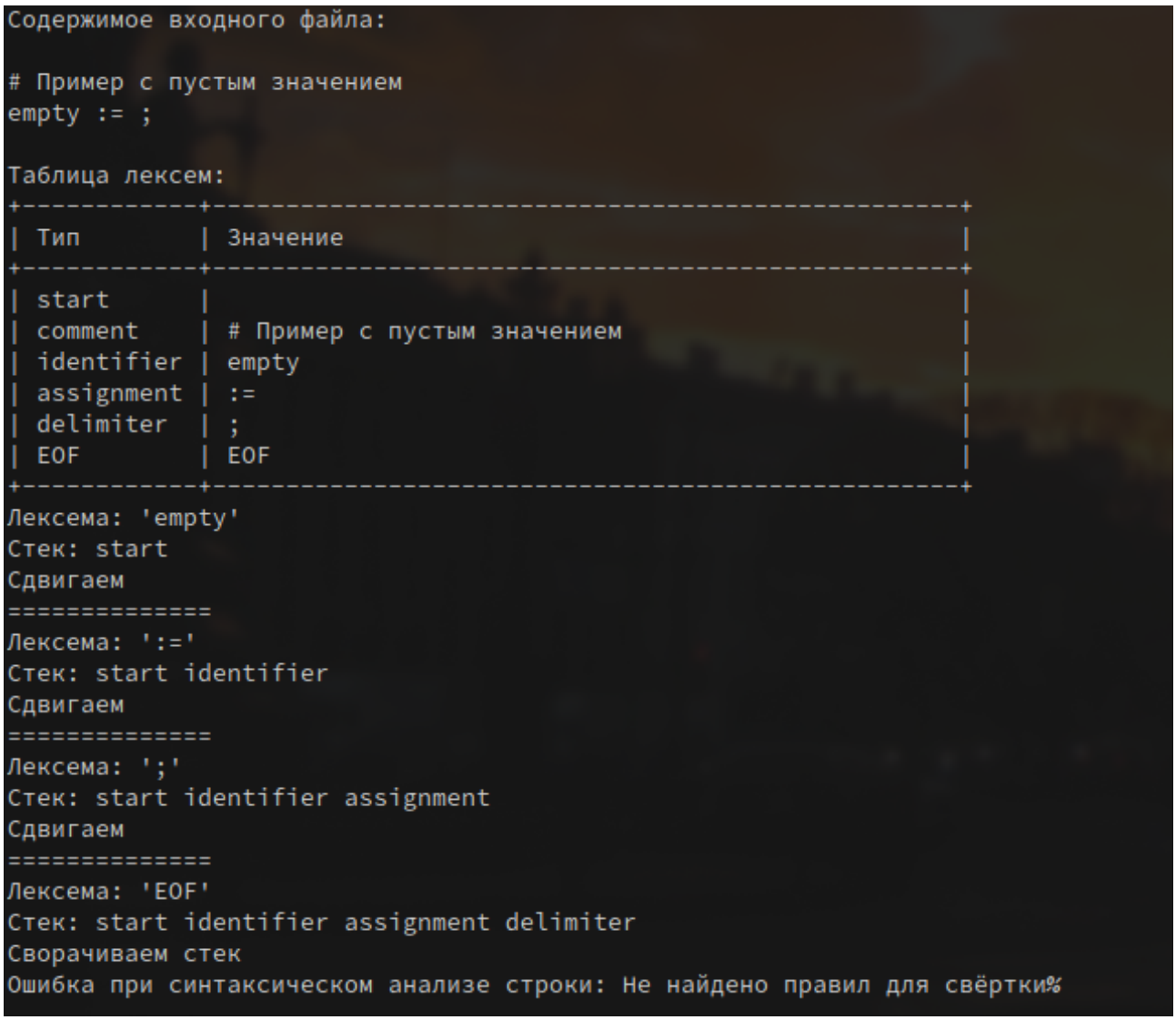


Рисунок 9 – предложение с синтаксической ошибкой

```
# Несоответствие скобок
id := ((ab);
```

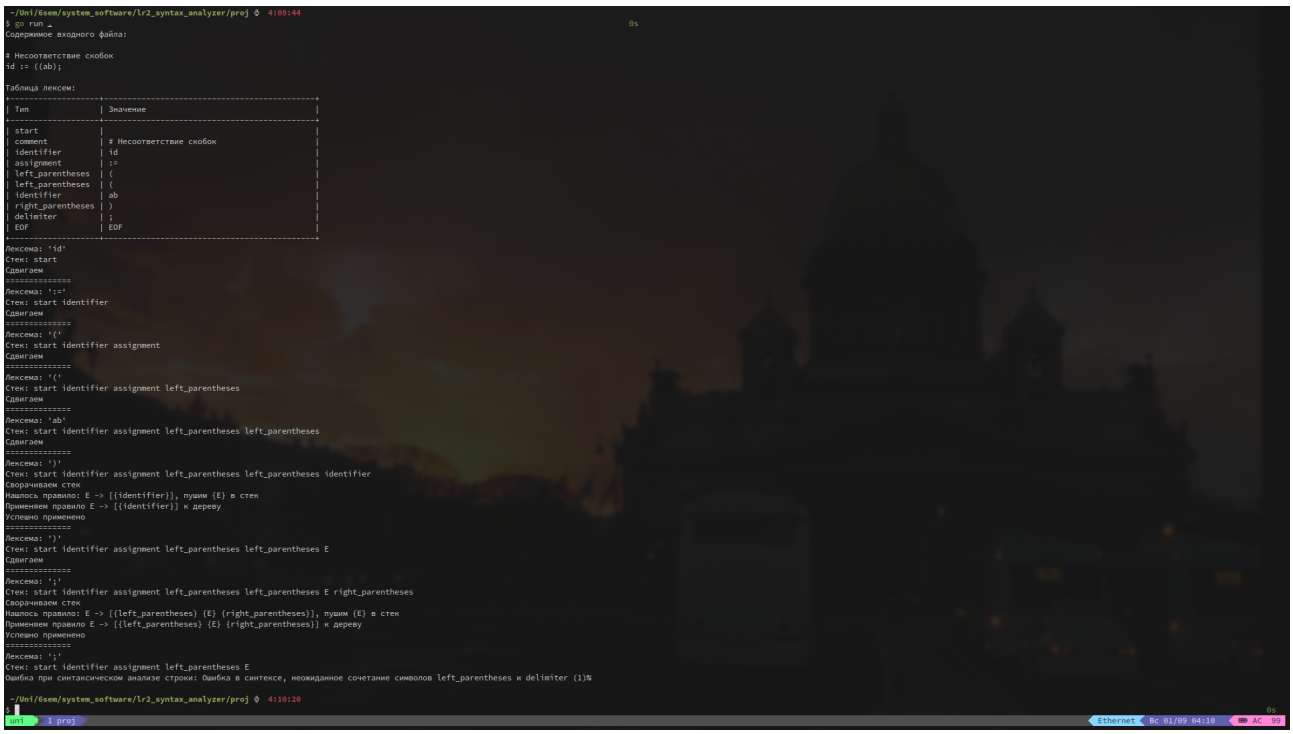


Рисунок 10 – предложение с синтаксической ошибкой. Пример 2

Вывод

Изучены основные понятия теории грамматик простого и операторного предшествования, ознакомился с алгоритмами синтаксического анализа (разбора) для некоторых классов КС-грамматик, получены практические навыки создания простейшего синтаксического анализатора для заданной грамматики операторного предшествования.

Получены практические навыки создания простейшего синтаксического анализатора для заданной грамматики операторного предшествования, обработи и представления результатов синтаксического анализа.