Минобрнауки России

Федеральное государственное бюджетное образовательное учреждение

высшего образования

**«САРАТОВСКИЙ национальный исследовательский  
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМЕНИ**

**Н. Г. ЧЕРНЫШЕВСКОГО»**

Факультет компьютерных наук и информационных технологий

**Основы трансляции языков программирования.**

Лабораторная работа №1. Построение синтаксического анализатора методом рекурсивного спуска.

Студента 4 курса 441 группы

направления 02.03.03—Математическое обеспечение и администрирование

информационных систем

факультета КНиИТ

Новоярчикова Михаила Андреевича

Саратов 2024

# Задание на лабораторную работу.

С использованием метода рекурсивного спуска реализовать синтаксический анализатор языка, цепочки которого имеют вид, указанный в задании на первую лабораторную работу. Синтаксический анализатор должен принимать на вход последовательность лексем, сформированную лексическим анализатором и восстанавливать дерево разбора цепочки. В случае ошибок во входной цепочке анализатор должен дать пользователю информативное сообщение с указанием причины и места возникновения ошибки.

# Описание грамматики исходного языка.

Грамматика языка, реализованная в синтаксическом анализаторе, может быть представлена следующими правилами:

**1. Присваивание (Assignment)**

<assignment> ::= <identifier> = <expression> ;

Здесь <identifier> — это переменная, которой присваивается значение выражения <expression>. Выражение может быть арифметическим.

**2. Арифметические выражения (Arithmetic Expression)**

<expression> ::= <term> { ("+" | "-") <term> }

<term> ::= <factor> { ("\*" | "/") <factor> }

<factor> ::= <identifier> | <constant> | "(" <expression> ")"

Арифметическое выражение состоит из одного или нескольких термов, разделенных операторами сложения и вычитания. Терм может состоять из множества факторов, разделенных операторами умножения и деления. Фактором может быть идентификатор, константа или выражение в скобках.

**3. Логические выражения (Logical Expression)**

<logical-expression> ::= <relational-expression> { ("and" | "or") <relational-expression> }

<relational-expression> ::= <operand> <relational-operator> <operand>

<relational-operator> ::= "==" | "<" | ">" | "<=" | ">="

<operand> ::= <identifier> | <constant>

Логическое выражение состоит из одного или нескольких реляционных выражений, соединенных логическими операторами and и or. Реляционное выражение сравнивает два операнда с помощью операторов сравнения, таких как ==, <, > и т.д.

**4. Цикл while (While Loop)**

<while-loop> ::= "while" <logical-expression> "do" <statement> "end"

Цикл while содержит логическое выражение в качестве условия. Если условие истинно, выполняется оператор <statement>, который может быть либо присваиванием, либо другим вложенным оператором (например, циклом).

**5. Цикл for (For Loop)**

<for-loop> ::= "for" "(" <declaration-or-assignment> ";" <relational-expression> ";" <increment> ")" <statement>

Цикл for начинается с инициализации переменной (либо объявления, либо присваивания), затем следует условие (реляционное выражение) и шаг инкрементации переменной. После выполнения тела цикла выполняется шаг инкрементации до тех пор, пока условие истинно.

**6. Операторы (Statements)**

<statement> ::= <assignment> | <for-loop> | <while-loop> | <block>

<block> ::= "{" <statement>\* "}"

Оператор может быть присваиванием, циклом for, циклом while или блоком кода, состоящим из нескольких операторов.

# Описание всех преобразований исходной грамматики и грамматику в окончательном виде, представленную в расширенной бэкусовой нормальной форме.

**1. Устранение левой рекурсии**

Левая рекурсия возникает, когда нетерминал ссылается на самого себя как на первое выражение в правиле. Это вызывает зацикливание в процессе рекурсивного разбора. Для корректной работы метода рекурсивного спуска мы должны устранить левую рекурсию в грамматике.

Пример исходной левой рекурсии:

<expression> ::= <expression> "+" <term> | <term>

Здесь <expression> ссылается на самого себя в первом правиле, что является левой рекурсией.

Преобразование:

Мы заменяем левую рекурсию на правую рекурсию с использованием нового нетерминала:

<expression> ::= <term> <expression'>

<expression'> ::= "+" <term> <expression'> | ε

Теперь грамматика не содержит левой рекурсии и может быть обработана рекурсивным спуском.

**2. Устранение неоднозначности**

Неоднозначность возникает, когда для одного и того же выражения существует несколько возможных деревьев разбора. Чтобы синтаксический анализатор мог однозначно разбирать цепочки, мы вводим строгие правила приоритета операторов и ассоциативности.

Пример исходной неоднозначной грамматики:

<term> ::= <term> "\*" <factor> | <factor>

Здесь могут быть проблемы с неоднозначностью при разборе выражений с несколькими операторами умножения.

Преобразование:

Уточняем порядок разбора, вводя четкие правила для операций:

<term> ::= <factor> { ("\*" | "/") <factor> }

Теперь выражения с операторами умножения и деления обрабатываются последовательно.

**3. Добавление скобок для приоритета операций**

В EBNF добавляются скобки {} и [] для представления опциональных и повторяющихся элементов. Это позволяет сократить количество правил и упростить грамматику.

**Грамматика в окончательном виде, представленная в расширенной форме Бэкуса — Наура (EBNF)**

**1. Присваивание:**

<assignment> ::= <identifier> "=" <expression> ";"

**2. Арифметическое выражение:**

<expression> ::= <term> { ("+" | "-") <term> }

<term> ::= <factor> { ("\*" | "/") <factor> }

<factor> ::= <identifier> | <constant> | "(" <expression> ")"

**3. Логическое выражение:**

<logical-expression> ::= <relational-expression> { ("and" | "or") <relational-expression> }

<relational-expression> ::= <operand> <relational-operator> <operand>

<relational-operator> ::= "==" | "<" | ">" | "<=" | ">="

<operand> ::= <identifier> | <constant>

**4. Цикл while:**

<while-loop> ::= "while" <logical-expression> "do" <statement> "end"

**5. Цикл for:**

<for-loop> ::= "for" "(" <declaration-or-assignment> ";" <relational-expression> ";" <increment> ")" <statement>

**6. Операторы:**

<statement> ::= <assignment> | <for-loop> | <while-loop> | <block>

<block> ::= "{" <statement>\* "}"

**Использование скобок в EBNF:**

{ } — для описания повторений (ноль или больше раз).

[ ] — для описания опциональных элементов.

| — для описания выбора (или).

# Описание основных алгоритмов и структур данных, используемых в программе

**А) описание процедур и функций рекурсивного спуска**

Метод рекурсивного спуска был реализован через методы класса Parser. Каждый метод отвечает за разбор одной из грамматических конструкций исходного языка. Эти методы рекурсивно вызывают другие методы для разбора вложенных структур, что и является основой метода рекурсивного спуска.

Примеры ключевых процедур:

**ParseAssignment**

Разбирает присваивание вида <identifier> = <expression>;.

После сопоставления идентификатора и знака =, вызывается ParseExpression для разбора выражения, которое должно быть присвоено переменной.

Затем ожидается символ ;, который завершает выражение.

**ParseExpression**

Разбирает арифметические выражения. Выражение состоит из одного или нескольких термов, которые соединяются операторами сложения и вычитания.

Внутри ParseExpression вызывается ParseTerm, который обрабатывает факторы (например, числа или переменные), разделенные операторами умножения и деления.

**ParseLogicalExpression**

Разбирает логические выражения, содержащие операторы and и or.

Каждое логическое выражение состоит из реляционных выражений, которые сравниваются с помощью операторов ==, <, >, <=, >=.

**ParseForLoop и ParseWhileLoop**

Эти методы разбирают циклы for и while.

ParseForLoop обрабатывает инициализацию, условие и инкремент в цикле for, а затем вызывает метод для разбора тела цикла.

ParseWhileLoop обрабатывает условие и тело цикла while.

**Б) Описание типов данных, используемых для представления последовательности лексем**

Типы данных, используемые для представления лексем, описаны в классе Token. Лексема состоит из двух полей: Type и Value.

**Token**

Поле **Type** хранит тип лексемы (например, идентификатор, оператор, ключевое слово).

Поле **Value** хранит строковое представление самой лексемы (например, имя переменной, значение числа).

**В) Описание процедур и функций, отвечающих за работу с последовательностью лексем и вывод синтаксических ошибок**

Работа с последовательностью лексем

* Лексический анализатор (Lexer) отвечает за разбор исходной строки символов и преобразование её в список лексем (List<Token>).
* Основная функция лексического анализатора — это метод Tokenize, который проходит по каждому символу строки, определяет тип символа и создает соответствующую лексему.

Вывод синтаксических ошибок

* Синтаксический анализатор выводит ошибки в том случае, если текущий токен не соответствует ожидаемому. Это происходит с помощью метода Match, который проверяет соответствие текущего токена ожидаемому типу.
* Если токен не соответствует ожидаемому, программа генерирует исключение с сообщением об ошибке, которое затем выводится пользователю.

# Описание интерфейса пользователя программы

Программа от пользователя ожидает выражение для анализа или exit для завершения.

# Контрольный пример и результаты тестирования.

Примеры:

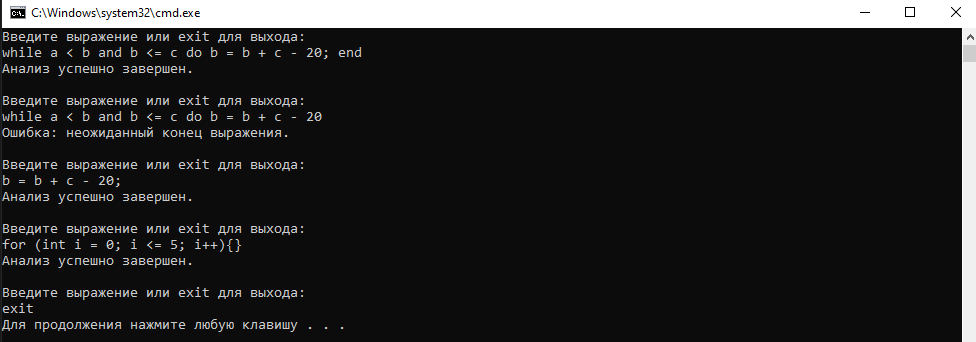
while a < b and b <= c do b = b + c - 20; end

b = b + c - 20;

for (int i = 0; i <= 5; i++){}

Пример для ошибки:

while a < b and b <= c do b = b + c - 20



# Листинг программы

using System;

namespace RcursiveDescentParser

{

public class Program

{

static void Main(string[] args)

{

while (true)

{

Console.WriteLine("Введите выражение или exit для выхода:");

string input = Console.ReadLine();

if (input.ToLower() == "exit")

{

Console.WriteLine();

break;

}

try

{

var lexer = new Lexer.Lexer(input);

var tokens = lexer.Tokenize();

var parser = new Parser.Parser(tokens);

parser.ParseStatement();

Console.WriteLine("Анализ успешно завершен.");

}

catch (Exception ex)

{

Console.WriteLine(ex.Message);

}

Console.WriteLine();

}

}

}

}

using RcursiveDescentParser.Grammar;

namespace RcursiveDescentParser.Lexer

{

public class Token

{

public TokenType Type { get; }

public string Value { get; }

public Token(TokenType type, string value)

{

Type = type;

Value = value;

}

public override string ToString()

{

return $"{Type} ({Value})";

}

}

}

namespace RcursiveDescentParser.Grammar

{

public enum TokenType

{

ID,

CONST,

ASSIGN,

SEMICOLON,

COLON,

PLUS,

MULT,

OPENPAREN,

CLOSEPAREN,

WHILE,

DO,

END,

AND,

OR,

REL,

CLOSEBRACE,

OPENBRACE,

INCREMENT,

FOR

}

}

using RcursiveDescentParser.Grammar;

using System;

using System.Collections.Generic;

namespace RcursiveDescentParser.Lexer

{

public class Lexer

{

private readonly string \_input;

private int \_position;

public Lexer(string input)

{

\_input = input;

\_position = 0;

}

public List<Token> Tokenize()

{

var tokens = new List<Token>();

while (\_position < \_input.Length)

{

var currentChar = \_input[\_position];

if (char.IsWhiteSpace(currentChar))

{

\_position++;

continue;

}

if (char.IsLetter(currentChar))

{

var identifier = ReadIdentifier();

if (identifier == "for")

{

tokens.Add(new Token(TokenType.FOR, identifier));

}

else if (identifier == "while")

{

tokens.Add(new Token(TokenType.WHILE, identifier));

}

else if (identifier == "and")

{

tokens.Add(new Token(TokenType.AND, identifier));

}

else if (identifier == "or")

{

tokens.Add(new Token(TokenType.OR, identifier));

}

else if (identifier == "do")

{

tokens.Add(new Token(TokenType.DO, identifier));

}

else if (identifier == "end")

{

tokens.Add(new Token(TokenType.END, identifier));

}

else

{

tokens.Add(new Token(TokenType.ID, identifier));

}

continue;

}

if (char.IsDigit(currentChar))

{

var number = ReadNumber();

tokens.Add(new Token(TokenType.CONST, number));

continue;

}

switch (currentChar)

{

case '=':

if (\_position + 1 < \_input.Length && \_input[\_position + 1] == '=')

{

tokens.Add(new Token(TokenType.REL, "=="));

\_position += 2;

}

else

{

tokens.Add(new Token(TokenType.ASSIGN, "="));

\_position++;

}

break;

case '<':

if (\_position + 1 < \_input.Length && \_input[\_position + 1] == '=')

{

tokens.Add(new Token(TokenType.REL, "<="));

\_position += 2;

}

else

{

tokens.Add(new Token(TokenType.REL, "<"));

\_position++;

}

break;

case '>':

if (\_position + 1 < \_input.Length && \_input[\_position + 1] == '=')

{

tokens.Add(new Token(TokenType.REL, ">="));

\_position += 2;

}

else

{

tokens.Add(new Token(TokenType.REL, ">"));

\_position++;

}

break;

case '+':

if (\_position + 1 < \_input.Length && \_input[\_position + 1] == '+')

{

tokens.Add(new Token(TokenType.INCREMENT, "++"));

\_position += 2;

}

else

{

tokens.Add(new Token(TokenType.PLUS, "+"));

\_position++;

}

break;

case '-':

tokens.Add(new Token(TokenType.PLUS, "-"));

\_position++;

break;

case '\*':

tokens.Add(new Token(TokenType.MULT, "\*"));

\_position++;

break;

case '(':

tokens.Add(new Token(TokenType.OPENPAREN, "("));

\_position++;

break;

case ')':

tokens.Add(new Token(TokenType.CLOSEPAREN, ")"));

\_position++;

break;

case '{':

tokens.Add(new Token(TokenType.OPENBRACE, "{"));

\_position++;

break;

case '}':

tokens.Add(new Token(TokenType.CLOSEBRACE, "}"));

\_position++;

break;

case ';':

tokens.Add(new Token(TokenType.SEMICOLON, ";"));

\_position++;

break;

case ':':

tokens.Add(new Token(TokenType.COLON, ":"));

\_position++;

break;

default:

throw new Exception($"Неизвестный символ: {currentChar}");

}

}

return tokens;

}

private string ReadIdentifier()

{

var start = \_position;

while (\_position < \_input.Length && char.IsLetterOrDigit(\_input[\_position]))

{

\_position++;

}

return \_input.Substring(start, \_position - start);

}

private string ReadNumber()

{

var start = \_position;

while (\_position < \_input.Length && char.IsDigit(\_input[\_position]))

{

\_position++;

}

return \_input.Substring(start, \_position - start);

}

}

}

using RcursiveDescentParser.Grammar;

using RcursiveDescentParser.Lexer;

using System;

using System.Collections.Generic;

namespace RcursiveDescentParser.Parser

{

public class Parser

{

private readonly List<Token> \_tokens;

private int \_currentTokenIndex;

#region Constructor and Helpers

public Parser(List<Token> tokens)

{

\_tokens = tokens;

\_currentTokenIndex = 0;

}

private Token CurrentToken

{

get

{

if (HasMoreTokens())

{

return \_tokens[\_currentTokenIndex];

}

else

{

throw new Exception("Ошибка: неожиданный конец выражения.");

}

}

}

private bool HasMoreTokens()

{

return \_currentTokenIndex < \_tokens.Count;

}

private void Match(TokenType expectedType)

{

if (\_currentTokenIndex >= \_tokens.Count)

{

throw new Exception($"Ошибка: ожидался {expectedType}, но входная строка завершилась.");

}

if (CurrentToken.Type == expectedType)

{

\_currentTokenIndex++;

}

else

{

throw new Exception($"Ошибка: ожидалось {expectedType}, но найдено {CurrentToken.Type}");

}

}

#endregion

#region Assignment and Expression Parsing

public void ParseAssignment()

{

if (HasMoreTokens() && CurrentToken.Type == TokenType.ID)

{

Match(TokenType.ID);

Match(TokenType.ASSIGN);

ParseExpression();

if (HasMoreTokens() && CurrentToken.Type == TokenType.SEMICOLON)

{

Match(TokenType.SEMICOLON);

}

else

{

throw new Exception("Ошибка: ожидается ';' в конце выражения.");

}

}

else

{

throw new Exception("Ошибка: ожидался идентификатор перед присваиванием.");

}

}

private void ParseExpression()

{

ParseTerm();

while (HasMoreTokens() && CurrentToken.Type == TokenType.PLUS)

{

Match(TokenType.PLUS);

ParseTerm();

}

}

private void ParseTerm()

{

ParseFactor();

while (CurrentToken.Type == TokenType.MULT)

{

Match(TokenType.MULT);

ParseFactor();

}

}

private void ParseFactor()

{

if (HasMoreTokens() && CurrentToken.Type == TokenType.ID)

{

Match(TokenType.ID);

if (HasMoreTokens() && CurrentToken.Type == TokenType.OPENPAREN)

{

Match(TokenType.OPENPAREN);

ParseParameters();

Match(TokenType.CLOSEPAREN);

}

}

else if (HasMoreTokens() && CurrentToken.Type == TokenType.CONST)

{

Match(TokenType.CONST);

}

else if (HasMoreTokens() && CurrentToken.Type == TokenType.OPENPAREN)

{

Match(TokenType.OPENPAREN);

ParseExpression();

Match(TokenType.CLOSEPAREN);

}

else

{

throw new Exception("Ошибка: ожидался идентификатор, константа или выражение в скобках");

}

}

private void ParseParameters()

{

ParseExpression();

while (CurrentToken.Type == TokenType.COLON)

{

Match(TokenType.COLON);

ParseExpression();

}

}

public void ParseDeclarationOrAssignment()

{

if (HasMoreTokens() && IsTypeKeyword(CurrentToken))

{

Match(CurrentToken.Type);

Match(TokenType.ID);

Match(TokenType.ASSIGN);

ParseExpression();

if (HasMoreTokens() && CurrentToken.Type == TokenType.SEMICOLON)

{

Match(TokenType.SEMICOLON);

}

else

{

throw new Exception("Ошибка: ожидается ';' в конце объявления переменной.");

}

}

else

{

ParseAssignment();

}

}

private bool IsTypeKeyword(Token token)

{

return token.Value == "int" || token.Value == "float" || token.Value == "double";

}

#endregion

#region Logical and Relational Expressions

private void ParseLogicalExpression()

{

ParseRelationalExpression();

while (CurrentToken.Type == TokenType.AND || CurrentToken.Type == TokenType.OR)

{

if (CurrentToken.Type == TokenType.AND)

{

Match(TokenType.AND);

}

else

{

Match(TokenType.OR);

}

ParseRelationalExpression();

}

}

private void ParseRelationalExpression()

{

ParseOperand();

if (CurrentToken.Type == TokenType.REL)

{

Match(TokenType.REL);

ParseOperand();

}

}

private void ParseOperand()

{

if (CurrentToken.Type == TokenType.ID)

{

Match(TokenType.ID);

}

else if (CurrentToken.Type == TokenType.CONST)

{

Match(TokenType.CONST);

}

else

{

throw new Exception("Ошибка: ожидался операнд");

}

}

#endregion

#region Loop and Statement Parsing

public void ParseForLoop()

{

if (CurrentToken.Type == TokenType.FOR)

{

Match(TokenType.FOR);

Match(TokenType.OPENPAREN);

ParseDeclarationOrAssignment();

ParseRelationalExpression();

Match(TokenType.SEMICOLON);

ParseIncrement();

Match(TokenType.CLOSEPAREN);

ParseStatement();

}

else

{

throw new Exception("Ошибка: ожидался оператор for.");

}

}

public void ParseWhileLoop()

{

if (CurrentToken.Type == TokenType.WHILE)

{

Match(TokenType.WHILE);

ParseLogicalExpression();

Match(TokenType.DO);

ParseStatement();

Match(TokenType.END);

}

else

{

throw new Exception("Ошибка: ожидался оператор while.");

}

}

private void ParseIncrement()

{

if (CurrentToken.Type == TokenType.ID)

{

Match(TokenType.ID);

if (HasMoreTokens() && CurrentToken.Type == TokenType.INCREMENT)

{

Match(TokenType.INCREMENT);

}

else if (HasMoreTokens() && CurrentToken.Type == TokenType.ASSIGN)

{

Match(TokenType.ASSIGN);

ParseExpression();

}

else

{

throw new Exception("Ошибка: ожидался инкремент или присваивание.");

}

}

else

{

throw new Exception("Ошибка: ожидался идентификатор в инкременте цикла for.");

}

}

public void ParseStatement()

{

if (CurrentToken.Type == TokenType.ID)

{

ParseAssignment();

}

else if (CurrentToken.Type == TokenType.FOR)

{

ParseForLoop();

}

else if (CurrentToken.Type == TokenType.WHILE)

{

ParseWhileLoop();

}

else if (CurrentToken.Type == TokenType.OPENBRACE)

{

Match(TokenType.OPENBRACE);

while (HasMoreTokens() && CurrentToken.Type != TokenType.CLOSEBRACE)

{

ParseStatement();

}

Match(TokenType.CLOSEBRACE);

}

else

{

throw new Exception("Ошибка: неизвестный оператор.");

}

}

#endregion

}

}

# Контрольные вопросы.

**1) Определение контекстно-свободной грамматики (CFG):**

Контекстно-свободная грамматика — это формальная грамматика, в которой правила вывода имеют вид 𝐴→𝛼, где A — это один нетерминальный символ, а α — произвольная строка, состоящая из терминальных и/или нетерминальных символов. Такие грамматики определяют язык, где структура выражений не зависит от контекста, в котором они встречаются.

**2) Класс грамматик, допускающий построение анализаторов методом рекурсивного спуска:**

Метод рекурсивного спуска может быть применен к грамматикам класса LL(1), то есть к контекстно-свободным грамматикам, которые можно анализировать слева направо (Left-to-right), с выводом с левым упорядочением (Leftmost derivation), и где анализатор принимает решение на основе одного символа вперед (1 lookahead).

**Терминал** — это символ, который не может быть заменён на другие символы в процессе вывода по правилам грамматики. Терминальные символы являются конечными "строительными блоками" строк языка, которые грамматика порождает. В контексте синтаксического анализа терминалы представляют собой конкретные символы или лексемы (например, ключевые слова, операторы, идентификаторы), которые встречаются в исходном коде или входных данных программы.

Пример терминалов для языка программирования может включать такие символы, как if, else, +, ;, = и т. д. Терминалы не подлежат дальнейшему разбору или разложению — они воспринимаются анализатором как единые целые.

Противоположностью терминалов являются нетерминальные символы, которые можно разложить на терминалы и другие нетерминалы, согласно правилам грамматики.

**3) Определение функции First1(…):**

Функция First1(X) для символа X (либо терминала, либо нетерминала) — это множество терминалов, которые могут быть первыми символами в любой строке, порожденной символом X. Если X — терминал, то First1(X) равно самому терминалу X. Если X — нетерминал, то First1(X) — это множество всех терминалов, которые могут быть первыми символами строк, порождаемых этим нетерминалом.

**Пример**: Для правила A→aB∣bCA \rightarrow aB \mid bCA→aB∣bC, First1(A) будет {a,b}\{a, b\}{a,b}.

**4) Как строится анализатор методом рекурсивного спуска**:

Анализатор методом рекурсивного спуска представляет собой набор рекурсивных функций, каждая из которых соответствует одному правилу грамматики. Эти функции вызывают друг друга для анализа соответствующих частей входной строки, рекурсивно обрабатывая каждый нетерминал. Для каждого нетерминала функция проверяет текущее входное выражение и пытается сопоставить его с одной из альтернатив грамматических правил. Если сопоставление успешно, анализ продолжается, в противном случае возвращается ошибка.

**5)** **Элементы, используемые в расширенной бэкусовой нормальной форме (EBNF)**:

Расширенная бэкусовая нормальная форма (EBNF) — это расширение обычной Бэкусовой нормальной формы (BNF), позволяющее более компактно записывать правила грамматики. В EBNF используются следующие элементы:

* Квадратные скобки [ ] — для обозначения необязательных элементов.
* Фигурные скобки { } — для обозначения повторяющихся элементов (ноль или более раз).
* Альтернатива ∣ — для обозначения вариантов.
* Символы группировки ( ) — для группирования выражений.