**Синтаксичний аналізатор методом рекурсивного спуску**

1. **Початкові дані для розробки синтаксичного аналізатора**
   1. **Завдання**

Варіант №19

**Арифметика**: цілі та дійсні числа, основні чотири арифметичні операції (додавання, віднімання, ділення та множення), піднесення до степеня(правоасоціативна операція), дужки

**Особливості**: експоненційна форма дійсного числа

**Інструкція повторення**: do <список операторів> while <логічний вираз> **Інструкція розгалуження:** if (<відношення>) {<сп. операторів>}

* 1. **Граматика мови**

Program = program Ident '{' DeclarList DoSection '}'.

DeclarList = '{' Declaration { Declaration } '}'.

Declaration = Type IdentList ';'.

IdentList = Ident{',' Ident }.

Type = int | float | boolean.

DoSection = '{' StatementList '}'.

StatementList = Statement {Statement}.

Statement = Assign | Inp | Out | ForStatement | IfStatement | LabelStatement | WhileStatement.

Assign = Ident '=' Expr ';'.

Expr = ArithmExpr | BooleanExpr.

BooleanExpr = Expr RelOp Expr | true | false.

ArithmExpr = [Sign] Term {AddOp Term}.

Term = Chunk {MultOp Chunk}.

Chunk = Factor {PowerOp Factor}.

Factor = Ident | Const | '(' Expr ')'.

Inp = input '(' IdenttList ')' ';'.

Out = print '(' IdenttList ')' ';'.

IfStatement = if '(' BooleanExpr ')' DoSection.

WhileStatement = do DoSection while '(' BooleanExpr ')'.

Const = IntNumber | FloatNumber | BoolConst.

IntNumber = [Sign] UnsignedInt.

FloatNumber = [Sign] UnsignedFloat.

Sign = '+' | '-'.

UnsignedInt = Digit {Digit}.

UnsignedFloat = UnsignedInt '.' UnsignedInt

| UnsignedInt '.' UnsignedInt 'E' IntNumber.

BoolConst = true | false.

Ident = Letter { Letter | Digit }.

Letter = 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' | 'z'.

Digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'.

SpecialSign = '.' | ',' | ':' | ';' | '(' | ')' | '{' | '}' | '=' | '+' | '-' | '\*' | '/' | '^' | '<' | '>' | WhiteSpace | EndOfLine.

WhiteSpace = ' ' | '\t'.

EndOfLine = '\n' | '\r' | '\n\r'.

SpecSymbols = ArithOp | RelOp | BracketsOp | AssignOp | Punct.

ArithOp = AddOp | MultOp | PowerOp.

AddOp = '+' | '-'.

MultOp = '\*' | '/'.

PowerOp = '^'.

RelOp = '==' | '<=' | '<' | '>' | '>=' | '!='.

BracketsOp = '(' | ')'.

AssignOp = '='.

Punct = '.' | ',' | ':' | ';'.

* 1. **Лексичний аналізатор мови**

В даній роботі використовується лексичний аналізатор, створений в ході лабораторної роботи №2, написаний мовою C#. Лексичний аналізатор та таблиця символів програми ініціалізуються так:

var lexicalAnalyzer = new LexicalAnalyzer();

var currentState = 0;

var filePath = "path";

foreach (var lineOfText in TextReader.GetLineOfText(filePath))

{

lexicalAnalyzer.Analyze(lineOfText, ref currentState);

}

* 1. **Формат таблиці символів**

Таблиця символів отримується внаслідок роботи лексичного аналізатора. Ключами таблиці є цілі числа, а значення об’єкти класу (record) у вигляді:

public record SymbolInformation(int LineNumber, string Lexeme, string LexemeToken, int? Index); де

LineNumber – номер рядка вхідної програми;

Lexeme – лексема;

LexemeToken – токен лексеми;

Index – індекс ідентифікатора або константи у таблиці ідентифікаторів та констант відповідно; для інших лексем - null.

* 1. **Аналізатор дужок у програмі**

Для більш коректного аналізу дужок у програмі було розроблено клас BracketsProcessor, який на основі стеку аналізує чи коректними є дужки у коді програми

public class BracketsProcessor

{

private Stack<string> BracketsStack = new(4);

public bool ControlBracketsFlow(SymbolInformation bracket, string expectedBracket)

{

var result = false;

if (bracket.LexemeToken == "par\_op" && bracket.Lexeme == expectedBracket)

{

Console.WriteLine(ParserMessages.Information, bracket.LineNumber, bracket.Lexeme, bracket.LexemeToken);

if (bracket.Lexeme is "{" or "(")

{

BracketsStack.Push(bracket.Lexeme);

result = true;

}

else if (BracketsStack.TryPop(out var bracketInStack))

{

if ((bracketInStack == "{" && bracket.Lexeme == "}") ||

(bracketInStack == "(" && bracket.Lexeme == ")"))

{

result = true;

}

}

}

if (!result)

{

throw new Exception($"Parser: unexpected bracket '{bracket.Lexeme}', expected - {expectedBracket}");

}

return result;

}

public void CheckStackStatus()

{

if (BracketsStack.Count != 0)

{

throw new Exception("Parser: some brackets wasn't closed");

}

}

}

* 1. **Код програми**

Код класу SyntaxAnalyzer реалізованого мовою С#

public class SyntaxAnalyzer

{

// ReSharper disable once InconsistentNaming

private readonly List<SymbolInformation>? AnalysisInformation;

private readonly BracketsProcessor \_bracketsProcessor;

private int \_iterator;

private int Iterator

{

get

{

if (++\_iterator > AnalysisInformation!.Count - 1)

throw new Exception("Parser: ");

return \_iterator;

}

set => \_iterator = value;

}

public SyntaxAnalyzer(AnalysisInformation analysisInformation, BracketsProcessor bracketsProcessor)

{

AnalysisInformation = analysisInformation.SymbolsInformation.Values.ToList();

\_bracketsProcessor = bracketsProcessor;

\_iterator = -1;

}

public List<string> ParseProgram()

{

var resultLists = new List<string>();

if (AnalysisInformation is not null)

{

if (ProcessProgramStartSection())

{

\_bracketsProcessor.ControlBracketsFlow(AnalysisInformation[Iterator], "{");

ProcessDeclarationSection();

ProcessDoSection();

\_bracketsProcessor.ControlBracketsFlow(AnalysisInformation[Iterator], "}");

\_bracketsProcessor.CheckStackStatus();

}

else

{

throw new Exception("Parser: unexpected error");

}

}

return resultLists;

}

#region Token Parsers

private static bool ParseToken((string, string) expectedToken, SymbolInformation currentInformation)

{

if (expectedToken == (currentInformation.Lexeme, currentInformation.LexemeToken))

{

Console.WriteLine(ParserMessages.Information,

currentInformation.LineNumber,

currentInformation.Lexeme,

currentInformation.LexemeToken);

}

else

{

throw new Exception(string.Format(ParserMessages.ErrorWithUnexpectedElements,

currentInformation.LineNumber,

currentInformation.Lexeme,

currentInformation.LexemeToken,

expectedToken.Item1,

expectedToken.Item2));

}

return true;

}

private static bool ParseToken(string expectedToken, SymbolInformation currentInformation)

{

if (expectedToken == currentInformation.LexemeToken)

{

Console.WriteLine(ParserMessages.Information,

currentInformation.LineNumber,

currentInformation.Lexeme,

currentInformation.LexemeToken);

}

else

{

throw new Exception(string.Format(ParserMessages.ErrorWithUnexpectedToken,

currentInformation.LineNumber,

currentInformation.LexemeToken,

expectedToken));

}

return true;

}

private static bool ParseIdentToken(SymbolInformation symbolInformation) =>

symbolInformation.LexemeToken == "ident" && ParseToken("ident", symbolInformation);

private static bool ParseTypeToken(SymbolInformation symbolInformation) =>

new[] { "int", "float", "boolean" }.Contains(symbolInformation.Lexeme) &&

ParseToken("keyword", symbolInformation);

#endregion

#region Main Parts Processors

private bool ProcessProgramStartSection()

{

var result = false;

var symbolInformation = AnalysisInformation![Iterator];

if (ParseToken(("program", "keyword"), symbolInformation))

{

symbolInformation = AnalysisInformation![Iterator];

if (ParseIdentToken(symbolInformation))

{

result = true;

}

}

return result;

}

private void ProcessDeclarationSection()

{

\_bracketsProcessor.ControlBracketsFlow(AnalysisInformation![Iterator], "{");

var symbolInformation = ProcessIdentList();

\_bracketsProcessor.ControlBracketsFlow(symbolInformation, "}");

}

private SymbolInformation ProcessIdentList()

{

SymbolInformation symbolInformation;

while ((symbolInformation = AnalysisInformation![Iterator]).Lexeme != "}")

{

if (ParseTypeToken(symbolInformation))

{

symbolInformation = AnalysisInformation[Iterator];

symbolInformation = ParseIdentList(symbolInformation, ";");

ParseToken((";", "punct"), symbolInformation);

}

}

return symbolInformation;

}

private SymbolInformation ParseIdentList(SymbolInformation symbolInformation, string endOfIdent)

{

if (ParseIdentToken(symbolInformation))

{

symbolInformation = AnalysisInformation![Iterator];

while (symbolInformation.Lexeme != endOfIdent)

{

if (ParseToken((",", "punct"), symbolInformation))

{

ParseIdentToken(AnalysisInformation![Iterator]);

}

symbolInformation = AnalysisInformation[Iterator];

}

}

return symbolInformation;

}

private void ProcessDoSection()

{

\_bracketsProcessor.ControlBracketsFlow(AnalysisInformation![Iterator], "{");

while (ProcessStatementList()) { }

\_bracketsProcessor.ControlBracketsFlow(AnalysisInformation![\_iterator], "}");

}

private bool ProcessStatementList()

{

var symbolInformation = AnalysisInformation![Iterator];

var result = symbolInformation switch

{

{ LexemeToken: "ident" } => ProcessIdentExpression(symbolInformation),

{ LexemeToken: "keyword", Lexeme: "input" } => ProcessInputExpression(symbolInformation),

{ LexemeToken: "keyword", Lexeme: "print" } => ProcessPrintExpression(symbolInformation),

{ LexemeToken: "keyword", Lexeme: "if" } => ProcessIfExpression(symbolInformation),

{ LexemeToken: "keyword", Lexeme: "do" } => ProcessDoWhileExpression(symbolInformation),

{ LexemeToken: "par\_op", Lexeme: "}" } => false,

\_ => throw new Exception($"Parser: unexpected token {symbolInformation.Lexeme}")

};

return result;

}

private bool ProcessIdentExpression(SymbolInformation symbolInformation)

{

var result = false;

if (ParseIdentToken(symbolInformation))

{

symbolInformation = AnalysisInformation![Iterator];

if (ParseToken(("=", "assign\_op"), symbolInformation))

{

result = ParseExpression();

if (result)

{

result = ParseToken((";", "punct"), AnalysisInformation![\_iterator]);

}

}

else

{

throw new Exception(string.Format(ParserMessages.ErrorExpectedAssignToken,

symbolInformation.LineNumber));

}

}

return result;

}

private bool ParseExpression()

{

var temp = \_iterator;

if (ParseBooleanExpression())

{

return true;

}

Iterator = temp;

if (ParseArithmeticExpression(AnalysisInformation![Iterator]))

{

return true;

}

throw new Exception("This kinds of expression isn’t allwoed there");

}

private bool ParseBooleanExpression()

{

var symbol = AnalysisInformation![Iterator];

if (symbol.Lexeme is "true" or "false" && ParseToken("boolval", symbol))

{

return true;

}

if (symbol.Lexeme is "(" && \_bracketsProcessor.ControlBracketsFlow(symbol, "("))

{

if (ParseArithmeticExpression(AnalysisInformation[Iterator]))

{

symbol = AnalysisInformation[\_iterator];

if (ParseRelExpression(symbol))

{

symbol = AnalysisInformation[Iterator];

if (ParseArithmeticExpression(symbol))

{

if (\_bracketsProcessor.ControlBracketsFlow(AnalysisInformation[\_iterator], ")"))

{

\_iterator++;

return true;

}

}

}

}

}

else if (ParseArithmeticExpression(symbol))

{

symbol = AnalysisInformation[\_iterator];

if (ParseRelExpression(symbol))

{

symbol = AnalysisInformation[Iterator];

if (ParseArithmeticExpression(symbol))

{

return true;

}

}

}

return false;

}

private bool ParseRelExpression(SymbolInformation symbolInformation)

=> symbolInformation.LexemeToken is "rel\_op" && ParseToken("rel\_op", symbolInformation);

private bool ParseArithmeticExpression(SymbolInformation symbolInformation)

{

if (symbolInformation.Lexeme is "+" or "-")

ParseToken("add\_op", symbolInformation);

if (ParseTerm(symbolInformation))

{

var symbol = AnalysisInformation![\_iterator];

while (symbol.Lexeme is "+" or "-" && ParseToken("add\_op", symbol))

{

if (!ParseTerm(AnalysisInformation[Iterator]))

return false;

symbol = AnalysisInformation![\_iterator];

}

return true;

}

return false;

}

private bool ParseTerm(SymbolInformation symbolInformation)

{

if (ParseChunk(symbolInformation))

{

symbolInformation = AnalysisInformation![\_iterator];

while (symbolInformation.Lexeme is "\*" or "/" && ParseToken("mult\_op", symbolInformation))

{

if (!ParseChunk(AnalysisInformation[Iterator]))

return false;

symbolInformation = AnalysisInformation![\_iterator];

}

return true;

}

return false;

}

private bool ParseChunk(SymbolInformation symbolInformation)

{

if (ParseFactor(symbolInformation))

{

var currentInformation = AnalysisInformation![\_iterator];

while (currentInformation.Lexeme == "^" && ParseToken(("^", "pow\_op"), currentInformation))

{

if (!ParseFactor(AnalysisInformation![Iterator]))

{

return false;

}

currentInformation = AnalysisInformation![\_iterator];

}

return true;

}

return false;

}

private bool ParseFactor(SymbolInformation symbol)

{

if (ParseConst(symbol) || ParseIdentToken(symbol))

{

\_iterator++;

return true;

}

if (symbol.Lexeme is "(" && \_bracketsProcessor.ControlBracketsFlow(symbol, "("))

{

if (ParseArithmeticExpression(AnalysisInformation![Iterator]))

{

return \_bracketsProcessor.ControlBracketsFlow(AnalysisInformation![Iterator], ")");

}

}

return false;

}

private bool ParseConst(SymbolInformation symbol)

=> (symbol.LexemeToken is "boolval" && ParseToken("boolval", symbol)) ||

(symbol.LexemeToken is "int" && ParseNumber(symbol, "int")) ||

(symbol.LexemeToken is "exp" && ParseNumber(symbol, "exp")) ||

(symbol.LexemeToken is "float" && ParseNumber(symbol, "float"));

private bool ParseNumber(SymbolInformation symbol, string typeOfNumber)

{

if (symbol.Lexeme is "+" or "-")

{

if (!ParseToken("add\_op", symbol))

return false;

}

return ParseToken(typeOfNumber, symbol);

}

private bool ProcessDoWhileExpression(SymbolInformation symbolInformation)

{

if (ParseToken(("do", "keyword"), symbolInformation))

{

if (\_bracketsProcessor.ControlBracketsFlow(AnalysisInformation![Iterator], "{"))

{

while (ProcessStatementList()) { }

if (\_bracketsProcessor.ControlBracketsFlow(AnalysisInformation![\_iterator], "}"))

{

if (ParseToken(("while", "keyword"), AnalysisInformation![Iterator]))

{

if (\_bracketsProcessor.ControlBracketsFlow(AnalysisInformation[Iterator], "("))

{

if (ParseBooleanExpression())

{

return \_bracketsProcessor.ControlBracketsFlow(AnalysisInformation[\_iterator], ")");

}

}

}

}

}

}

return false;

}

private bool ProcessIfExpression(SymbolInformation symbolInformation)

{

if (ParseToken(("if", "keyword"), symbolInformation))

{

if (\_bracketsProcessor.ControlBracketsFlow(AnalysisInformation![Iterator], "("))

{

if (ParseBooleanExpression())

{

if (\_bracketsProcessor.ControlBracketsFlow(AnalysisInformation![\_iterator], ")"))

{

\_bracketsProcessor.ControlBracketsFlow(AnalysisInformation![Iterator], "{");

while (ProcessStatementList()) { }

return \_bracketsProcessor.ControlBracketsFlow(AnalysisInformation![\_iterator], "}");

}

}

}

}

return false;

}

private bool ProcessPrintExpression(SymbolInformation symbolInformation)

{

if (ParseToken(("print", "keyword"), symbolInformation))

{

if (\_bracketsProcessor.ControlBracketsFlow(AnalysisInformation![Iterator], "("))

{

symbolInformation = ParseIdentList(AnalysisInformation[Iterator], ")");

if (\_bracketsProcessor.ControlBracketsFlow(symbolInformation, ")"))

{

return ParseToken((";", "punct"), AnalysisInformation[Iterator]);

}

}

}

return false;

}

private bool ProcessInputExpression(SymbolInformation symbolInformation)

{

if (ParseToken(("input", "keyword"), symbolInformation))

{

if (\_bracketsProcessor.ControlBracketsFlow(AnalysisInformation![Iterator], "("))

{

symbolInformation = ParseIdentList(AnalysisInformation[Iterator], ")");

if (\_bracketsProcessor.ControlBracketsFlow(symbolInformation, ")"))

{

return ParseToken((";", "punct"), AnalysisInformation[Iterator]);

}

}

}

return false;

}

#endregion

}

1. **Програмна реалізація парсера**

Синтаксичний аналізатор, як і лексичний, реалізований мовою С#. Робота проводиться над таблицею символів, що була отримана внаслідок роботи лексичного аналізатора. Для основних нетерміналів мови створено методи з відповідними назвами, які дозволяють опрацювати граматику мови. Тому методи, що відповідають певним нетерміналам, які містять інші нетермінали, викликають методи, які відповідають їм. Таким чином рекурсивно ми приходимо до найпростіших конструкцій мови. Наявність важливих терміналів мови у необхідних місцях ми перевіряємо за допомогою методу ParseToken(), який має дві реалізації. В першому випадку наявність лексеми й токена є обов’язковою, тому їх відсутність викличе помилку в програмі. В другому – перевіряємо лише чи відповідає токен вказаному та повертаємо значення типу boolean. Це дозволяє скоротити кількість одноманітного коду, що повторюється у різних місцях.

1. **Тестування роботи синтаксичного аналізатора**

Наводжу приклад лише початку та кінця аналізу, оскільки вони цілком коректно відображають роботу реалізованого нами аналізатора

* 1. Базовий приклад

program myprogram {

{

int x,y,z, i;

float a,b;

boolean c;

}

{

a = x ^ y;

print(x, y);

input(x,y);

x = x + 3;

y = y \* 2;

x = x - 2;

y = y / 3;

a = x ^ y;

b = 1.3E02;

c = (5 <= 3);

i = 4;

do {

if (i / 2 != 0) {

print(i);

}

i = i - 1;

} while (i >= 0)

}

}

Результат роботи:

ParseToken: in row 1 lexeme - 'program'|token - 'keyword'

ParseToken: in row 1 lexeme - 'myprogram'|token - 'ident'

ParseToken: in row 1 lexeme - '{'|token - 'par\_op'

ParseToken: in row 2 lexeme - '{'|token - 'par\_op'

ParseToken: in row 3 lexeme - 'int'|token - 'keyword'

ParseToken: in row 3 lexeme - 'x'|token - 'ident'

ParseToken: in row 3 lexeme - ','|token - 'punct'

ParseToken: in row 3 lexeme - 'y'|token - 'ident'

ParseToken: in row 3 lexeme - ','|token - 'punct'

ParseToken: in row 3 lexeme - 'z'|token - 'ident'

ParseToken: in row 3 lexeme - ','|token - 'punct'

ParseToken: in row 3 lexeme - 'i'|token - 'ident'

ParseToken: in row 3 lexeme - ';'|token - 'punct'

.....

ParseToken: in row 23 lexeme - '1'|token - 'int'

ParseToken: in row 23 lexeme - ';'|token - 'punct'

ParseToken: in row 24 lexeme - '}'|token - 'par\_op'

ParseToken: in row 24 lexeme - 'while'|token - 'keyword'

ParseToken: in row 24 lexeme - '('|token - 'par\_op'

ParseToken: in row 24 lexeme - 'i'|token - 'ident'

ParseToken: in row 24 lexeme - '>='|token - 'rel\_op'

ParseToken: in row 24 lexeme - '0'|token - 'int'

ParseToken: in row 24 lexeme - ')'|token - 'par\_op'

ParseToken: in row 25 lexeme - '}'|token - 'par\_op'

ParseToken: in row 26 lexeme - '}'|token - 'par\_op'

* 1. Відсутність розділів оголошень та інструкцій

Вихідний код:

program myprogram {

{

}

{

}

}

Результат роботи:

1 program keyword

1 myprogram ident 1

1 { par\_op

2 { par\_op

3 } par\_op

4 { par\_op

5 } par\_op

6 } par\_op

Lexer: Lexical analyzer was successfully completed

ParseToken: in row 1 lexeme - 'program'|token - 'keyword'

ParseToken: in row 1 lexeme - 'myprogram'|token - 'ident'

ParseToken: in row 1 lexeme - '{'|token - 'par\_op'

ParseToken: in row 2 lexeme - '{'|token - 'par\_op'

ParseToken: in row 3 lexeme - '}'|token - 'par\_op'

ParseToken: in row 4 lexeme - '{'|token - 'par\_op'

ParseToken: in row 5 lexeme - '}'|token - 'par\_op'

ParseToken: in row 6 lexeme - '}'|token - 'par\_op'

* 1. Програма має некоректну структуру

За граматикою кожна програма має містити зону оголошень та зону виконання і також починатись з ключового слова program й ідентифікатору

Приклад з некоректним ключовим словом

progr myprogram {

{

}

{

}

}

Результат роботи:

Parser Error:

Line 1 has unexpected elements - ('progr' 'ident'). Expected - ('program' 'keyword')

Приклад з пропущеною дужкою:

program myprogram {

}

{

}

}

Результат роботи

ParseToken: in row 1 lexeme - 'program'|token - 'keyword'

ParseToken: in row 1 lexeme - 'myprogram'|token - 'ident'

ParseToken: in row 1 lexeme - '{'|token - 'par\_op'

Parser: unexpected bracket '}', expected - {

* 1. Присвоєння значень змінній в розділі оголошень

program myprogram {

{

int a = 4;

}

{

}

}

Результат роботи:

ParseToken: in row 1 lexeme - 'program'|token - 'keyword'

ParseToken: in row 1 lexeme - 'myprogram'|token - 'ident'

ParseToken: in row 1 lexeme - '{'|token - 'par\_op'

ParseToken: in row 2 lexeme - '{'|token - 'par\_op'

ParseToken: in row 3 lexeme - 'int'|token - 'keyword'

ParseToken: in row 3 lexeme - 'a'|token - 'ident'

Parser Error:

Line 3 has unexpected elements - ('=' 'assign\_op'). Expected - (',' 'punct')

* 1. Відсутність ідентифікаторів, де вони очікуються

program myprogram {

{

int a;

}

{

print();

}

}

Результат роботи:

ParseToken: in row 1 lexeme - 'program'|token - 'keyword'

ParseToken: in row 1 lexeme - 'myprogram'|token - 'ident'

ParseToken: in row 1 lexeme - '{'|token - 'par\_op'

ParseToken: in row 2 lexeme - '{'|token - 'par\_op'

ParseToken: in row 3 lexeme - 'int'|token - 'keyword'

ParseToken: in row 3 lexeme - 'a'|token - 'ident'

ParseToken: in row 3 lexeme - ';'|token - 'punct'

ParseToken: in row 4 lexeme - '}'|token - 'par\_op'

ParseToken: in row 5 lexeme - '{'|token - 'par\_op'

ParseToken: in row 6 lexeme - 'print'|token - 'keyword'

ParseToken: in row 6 lexeme - '('|token - 'par\_op'

Parser error:

Parser: expected list of identifiers

1. Висновки

В ході даної лабораторної роботи було створено синтаксичний аналізатор, що на основі лексичного аналізатора здатний обробляти таблицю символів та перевіряти коректність програми згідно з її граматикою, тобто щоб усі отримані лексеми були на своїх місцях та не порушували існуючі правила. Отриманий застосунок було перевірено на кількох тестових випадках та в цілому працює так, як очікується.