

**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»
Теплоенергетичний факультет
Кафедра автоматизації проектування енергетичних процесів і
систем**

КУРСОВА РОБОТА

**з дисципліни: «Основи розробки трансляторів»
на тему:**

**Мова програмування загального призначення
Crundras та її імплементація**

Студент 3 курсу групи ТР-72
спеціальності 122 “Комп’ютерні науки”
Куйбіда Павло Костянтинович
Керівник д.т.н, проф. Надашківський О.Л.

Кількість балів: _____ Оцінка: _____

Члени комісії:	_____	д.т.н, проф. Надашківський О.Л.
	(підпис)	(вчене звання, науковий ступінь, прізвище та ініціали)
	_____	к.т.н., доц. Стативка Ю.І.
	(підпис)	(вчене звання, науковий ступінь, прізвище та ініціали)

	(підпис)	(вчене звання, науковий ступінь, прізвище та ініціали)

Київ- 2020 рік

АНОТАЦІЯ

Курсову роботу виконано на 42 аркушах, вона містить 3 додатки та перелік посилань на використані джерела з 4 найменувань. У роботі наведено 45 рисунків та 4 таблиці.

Метою курсової роботи є розробка імперативної мови загального призначення та її імплементація засобами мови програмування C#.

Ключові слова: формальна граматика, специфікація мови програмування, лексичний аналіз, таблиця символів, автомат з магазинною пам'яттю та під-автоматами, діаграма станів, синтаксичний аналіз, трансляція.

ВСТУП.....	5
1 Завдання	6
2 Специфікація мови «Crundras»	7
2.1 Обробка.....	7
2.2 Нотація.....	7
2.3 Алфавіт	8
2.4 Лексика	8
2.4.1 Спеціальні символи	8
2.4.2 Ідентифікатори.....	9
2.4.3 Літерали	9
2.4.4 Ключові слова	10
2.4.5 Токени.....	10
2.5 Тип.....	10
2.6 Синтаксис	11
2.6.1 Вирази.....	11
2.6.2 Оператори.....	12
2.7 Програма.....	12
2.8 Оголошення.....	12
2.9 Інструкції	13
2.9.1 Оператор (інструкція) присвоювання.....	13
2.9.2 Інструкція введення.....	14
2.9.3 Інструкція виведення.....	14
2.9.4 Оператор вибору.....	14
2.9.5 Оператор циклу.....	15
3 Структура транслятора	16
4 Програмна реалізація.....	18
4.1 Лексичний аналізатор.....	18
4.2 Синтаксичний аналізатор.....	19
4.3 Проміжна форма подання програми	21
4.3.1 Expression	22
4.3.2 Negation	22
4.3.3 Assignment statement.....	22
4.3.4 Output statement.....	22
4.3.5 Input statement	22
4.3.6 Declaration Statement.....	23
4.3.7 Label statement.....	23
4.3.8 Jump statement	23
4.3.9 Conditional statement.....	23
4.3.10 Iteration Statement.....	24

4.4	Interpreting.....	25
4.4.1	label, identifier, int_literal, float_literal.....	25
4.4.2	goto	25
4.4.3	if.....	25
4.4.4	@	25
4.4.5	\$	25
4.4.6	=.....	25
4.4.7	NEG.....	25
4.4.8	+, -, *, **, /, %, <, <=, >, >=, ==, !=.....	25
5	Інструкція користувача по роботі з транслятором	26
6	Тестування	27
	Output explanation.....	27
6.1	General arithmetic example:	27
6.2	Power operation examples	28
6.2.1	Example 1	28
6.2.2	Example 2.....	29
6.2.3	Example 3	30
6.3	General logical operator examples	31
6.3.1	Example 1	31
6.3.2	Example 2.....	32
6.4	Assignment example.....	32
6.5	Input example	33
6.6	Output example.....	34
6.7	Undefined variable error example	34
1.1.	Negation example	35
6.8	If example	36
6.9	For example	37
6.10	Complex example	38
	ВИСНОВКИ.....	41
	СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ	42

ВСТУП

В наш час різноманітні комп'ютери використовуються всюди. Сучасний світ залежить від мов програмування, оскільки програмне забезпечення усіх комп'ютерів написано тою чи іншою мовою програмування. Задля коректного створення способів текстового опису, а згодом і їх виконання, використовують методи на межі лінгвістики, дискретної математики і інформатики.

Вже розроблені тисячі різноманітних мов та трансляторів для них, проте питання створення нових було і буде актуальним. Стрімкий розвиток технологій, постійна постановка все нових задач приводять до утворення нових підходів до методів комунікації спеціалістів з все більш «розумним» апаратним забезпеченням.

Також, принципи і методи проектування компіляторів застосовні у великому спектрі задач програмування, що навряд чи який вчений-кібернетик і тим паче програміст-практик не зустрінеться з ними купу разів під час своєї діяльності.

1 Завдання

Метою даної курсової роботи є розробити імперативну мову загального призначення і транслятор до неї. Обов'язковою вимогою є:

1. Дотримання структури оператора циклу:

```
for <identifier>=<expr1> to <expr2> by <expr3> while (<expr4>)  
    <operations block> rof;
```

2. Дотримання структури умовного оператора:

```
if (<expression>) <operations block>;
```

3. Обробка цілих та дійсних чисел — не менше чотирьох арифметичних операцій, унарний мінус, піднесення до степеня та дужки.

2 Специфікація мови «Crundras»

2.1 Обробка

Програма, написана мовою, подається на вхід транслятора (компілятора або інтерпретатора) для трансформації до цільової форми. Результат трансляції виконується у системі часу виконання (run-time system), для чого приймає вхідні дані та надає результат виконання програми. Трансляція передбачає фази лексичного та синтаксичного аналізу, а також фазу генерації проміжного коду. Кожна фаза здійснюється окремим проходом.

2.2 Нотація

Для опису мови Crundras використовується розширена форма Бекуса – Наура. Ланцюжки, що починаються з великої літери вважаються нетерміналами (нетермінальними символами). Термінали — ланцюжки, що починаються з маленької літери, або знаходяться між одинарними, або подвійними лапками. Для графічного представлення граматики використовуються синтаксичні діаграми Вірта.

Метасимвол	Значення
=	визначається як
	альтернатива
[x]	0 або 1 екземпляр x
{ x }	0 або більше екземплярів x
(x y)	групування: «будь-який з» (x або y)
Fgh	нетермінал
Ghj	термінал
'1'	термінал
"1"	термінал

Табл. 1: Прийнята нотація РБНФ

2.3 Алфавіт

Програма може містити текст з використанням таких символів (character) — літер, цифр, спеціальних знаків та ознаки кінця файлу:

```
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' | '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'.
Digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'.
SpecialSigns = '+' | '-' | '*' | '/' | '%' | '<' | '>' | '=' | '!' | ':' | ';'
              | '(' | ')' | '$' | '@' | '.'.
WhiteSpaces = ' ' | '\t'.
EndOfLine = '\n' | '\r' | '\n\r' | '\r\n'.
EndOfFile = '\u0000'.
```

2.4 Лексика

Лексичний аналіз виконується окремим проходом і не несе допоміжний функціонал синтаксичного чи семантичного аналізу. Лексичний аналізатор розбиває вхідний текст на лексеми. У програмі мовою C# можна використовуватись лексичні елементи, що класифікуються як спеціальні символи, ідентифікатори, беззнакові цілі константи, беззнакові дійсні константи та ключові слова.

2.4.1 Спеціальні символи

Синтаксис

```
1. SpecialSymbols = ArithmeticOperators | RelationalOperators |
                   BracketsOperators | Punctuation.
ArithmeticOperator = '+' | '-' | '*' | '**' | '/' | '%'.
RelationalOperator = '<' | '>' | '<=' | '>=' | '==' | '!='.
BracketsOperators = '(' | ')' | '{' | '}'.
Punctuation = ':' | ';' | '.'.
InputOutputOperators = '@' | '$'.
```


Опис

2. До спеціальних символів належать арифметичні оператори, оператори відношень, оператор присвоювання та знаки пунктуації.

Обмеження

3. Набір токенів див. додаток табл. 2

2.4.2 Ідентифікатори

Синтаксис

1. `Ident = Letter {Letter | Digit}.`

Опис

2. Першим символом ідентифікатора може бути тільки літера, наступні символи, якщо вони є, можуть бути цифрами або літерами. Довжина ідентифікатора не обмежена.

Обмеження

3. Жоден ідентифікатор не може збігатись із ключовим (вбудованим, зарезервованим) словом.

Семантика

4. Елемент, який у фазі лексичного аналізу може бути визначений як ідентифікатор, або як ключове слово, вважається ключовим словом.

Приклади

5. `a`, `x1`, `time24`

2.4.3 Літерали

Синтаксис

1. `Literal = IntegerLiteral | FloatingLiteral.`

`FloatingLiteral = IntegerLiteral '.' [IntegerLiteral].`

`IntegerLiteral = Digit {Digit}.`

Обмеження

2. Кожен літерал повинен мати тип, а величина літералу повинна знаходитись у діапазоні репрезентативних значень для її типу.

Семантика

3. Кожен літерал має тип, визначений її формою та значенням.

Приклади

4. 12, 234, 1.54, 34.567, 23.

2.4.4 Ключові слова

Синтаксис

1. `keywords = 'for' | 'by' | 'to' | 'while' | 'rof' | 'goto' | 'if' | 'int' | 'float'.`

2.4.5 Токени

З потоку символів вхідної програми на етапі лексичного аналізу виокремлюються послідовності символів з певним сукупним значенням, — токени. Список токенів див. табл. 2.

2.5 Тип

Мова Crundras обробляє значення двох типів: `int` та `float`.

1. Цілий тип `int` може бути представлений оголошеною змінною типу `int`, або константою `IntegerLiteral`. Діапазон значень залежить від реалізації.
2. Дійсний тип `float` може бути представлений оголошеною змінною типу `float` або константою `FloatingLiteral`.

2.6 Синтаксис

2.6.1 Вирази

Синтаксис

1. Expression = [Sign] ('(' Expression ')') | Literal | Identifier
{Operator [Sign] ('(' Expression ')') | Literal | Identifier}.

Опис

2. Вираз - це послідовність операторів і операндів, що визначає порядок обчислення значення.
3. Значення, обчислене за арифметичним виразом, має тип float або int.
4. Окрім піднесення до степені всі бінарні оператори у виразах є лівоасоціативними.
5. Піднесення до степені є правоасоціативним.
6. Послідовність двох або більше операторів з однаковим пріоритетом лівоасоціативна.
7. Пріоритети операцій:

Operator	Comment
<, <=, >, >=, !=, ==	Relation
+, -	Sum
*, /, %, **	Multiplication
+, -	Unary

Табл. 3: Таблиця пріоритетів операцій

Обмеження

8. Використання змінної, з не визначеним на момент обчислення виразу значенням, викликає помилку.

Семантика

9. Кожна константа має тип, визначений її формою та значенням.

Приклади

10. $x, 12, (a + 234), 32/(b + 786), -b, f1 + g,$
 $(a*x + b/z) >= (k ** t)$

2.6.2 Оператори

1. Типом результату бінарних операторів $+$, $-$, $*$, $**$ є тип обох операндів, якщо вони одного типу, інакше обирається більший тип.
2. Типом результату бінарного оператора $/$ завжди є `float`.
3. Типом результату бінарного оператора $\%$ завжди є `int`.
4. Тип результату унарних операторів завжди рівний типу операнду.
5. Тип результату операторів відношення завжди `int`.

2.7 Програма

Програма складається з набору виразів. Програма не може містити неоголошену змінну.

$$_Program = \{Statement\}.$$

2.8 Оголошення

Оголошення (декларації) специфікує інтерпретацію та атрибути набору ідентифікаторів.

Синтаксис

1. `DeclarationStatement = TypeSpecifier Identifier ';'.`
`TypeSpecifier = "int" | "float".`

Обмеження

2. Щонайменше один специфікатор типу повинен бути вказаний специфікаторами декларацій у кожній декларації.

Семантика

3. Декларація визначає інтерпретацію та атрибути ідентифікатора. Визначення ідентифікатора - це (єдине) оголошення ідентифікатора.

Приклад

4. `int a; double h;`

2.9 Інструкції

Інструкції (Statements) визначають алгоритмічні дії, які мають бути виконані. За винятком зазначених далі випадків, інструкції виконуються послідовно.

2.9.1 Оператор (інструкція) присвоювання

Синтаксис

1. `AssignmentStatement = AssignmentExpression ';'.`
`AssignmentExpression = Identifier '=' Expression.`

Обмеження

2. Якщо тип змінної з ідентифікатором Identifier не відповідає типу виразу праворуч оператора =, то вираз приводиться до типу змінної.

Семантика

3. Інструкція присвоювання заносить значення лівого виразу у змінну, що є лівим операндом.
4. Після виконання присвоєння, AssignmentExpression має значення лівого операнду.

Приклад

5. `foo = 35;`

2.9.2 Інструкція введення

Синтаксис

1. `InputStatement = '$' Identifier ';'.`

Приклад

2. `$d;`

2.9.3 Інструкція виведення

Синтаксис

1. `OutputStatement = '@' Expression ';'.`

Приклад

2. `@42;`

2.9.4 Оператор вибору

Синтаксис

1. `ConditionStatement = "if" '(' Expression ')' Statement.`

Обмеження

2. Керуючий вираз повинен бути скалярним.

Семантика

3. Якщо `Expression` не дорівнює 0, то виконується `Statement`.

2.9.5 Оператор циклу

Синтаксис

1. IterationStatement = "for" AssignmentExpression "to" Expression "by" Expression "while" '(' Expression ')' Statement "rof" ';'.

Обмеження

2. Керуючий вираз повинен бути скалярним.

Семантика

3. Інструкція:

```
for <assignment> to <expression-1> by <expression-2> while(<expression-3>)
    <statement>
rof;
```

3.1.<assignment> виконується до будь-яких інструкцій тілу цикла.

3.2.Кожного кроку вирази виконуються у такій послідовності:

```
<expression-3>
<expression-2>
<expression-1>
<statement>
```

3.3.Тіло циклу повторно виконується доки результат <expression-3> не дорівнює 0 або змінна, ініціалізована у <assignment>, та <expression-1> не рівні.

Приклад

4. for d = 15 to 73%3 by 0.34 while(d >= eps) a = d*3-7; rof;

3 Структура транслятора

Транслятор — це програма, що зчитує текст програми, написаної однією мовою — початковою, і транслює в еквівалентний текст іншою мовою — цільовою (рис. 1). Одна з найважливіших ролей транслятора у повідомленні про помилки в початковій програмі, знайдених в процесі трансляції. [3]

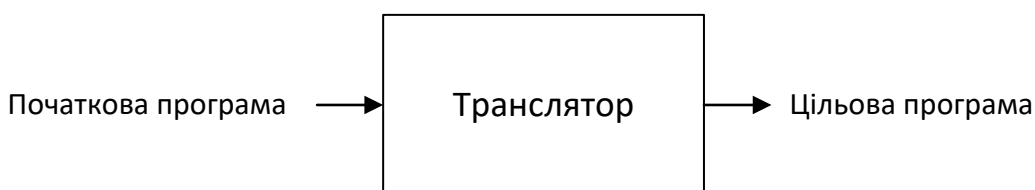


Рис. 1. Транслятор

Інтерпретатор представляє собою ще один поширений вид процесору мов. Замість отримання цільової програми — інтерпретатор безпосередньо виконує операції, вказані в початковій програмі, з даними, що надає користувач (рис. 2).

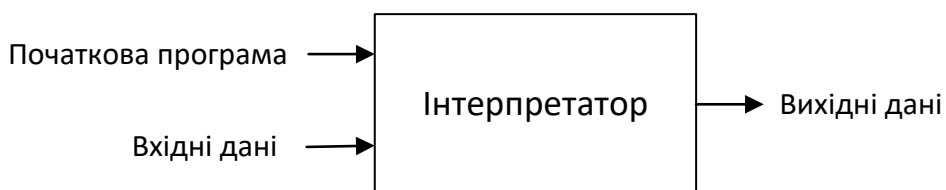


Рис. 2. Інтерпретатор

При детальному огляді процес трансляції можна умовно розбити на послідовність певних фаз, кожна з яких перетворює одно проміжне представлення програми на інше. Типове положення фаз трансляції показано на рис.3. На практиці деякі фази можуть бути об'єднані чи опущені, а проміжне

представлення між фазами не завжди будуватися явно. Таблиця символів, в якій зберігається інформація про всю програму, використовується усіма фазами трансляції.

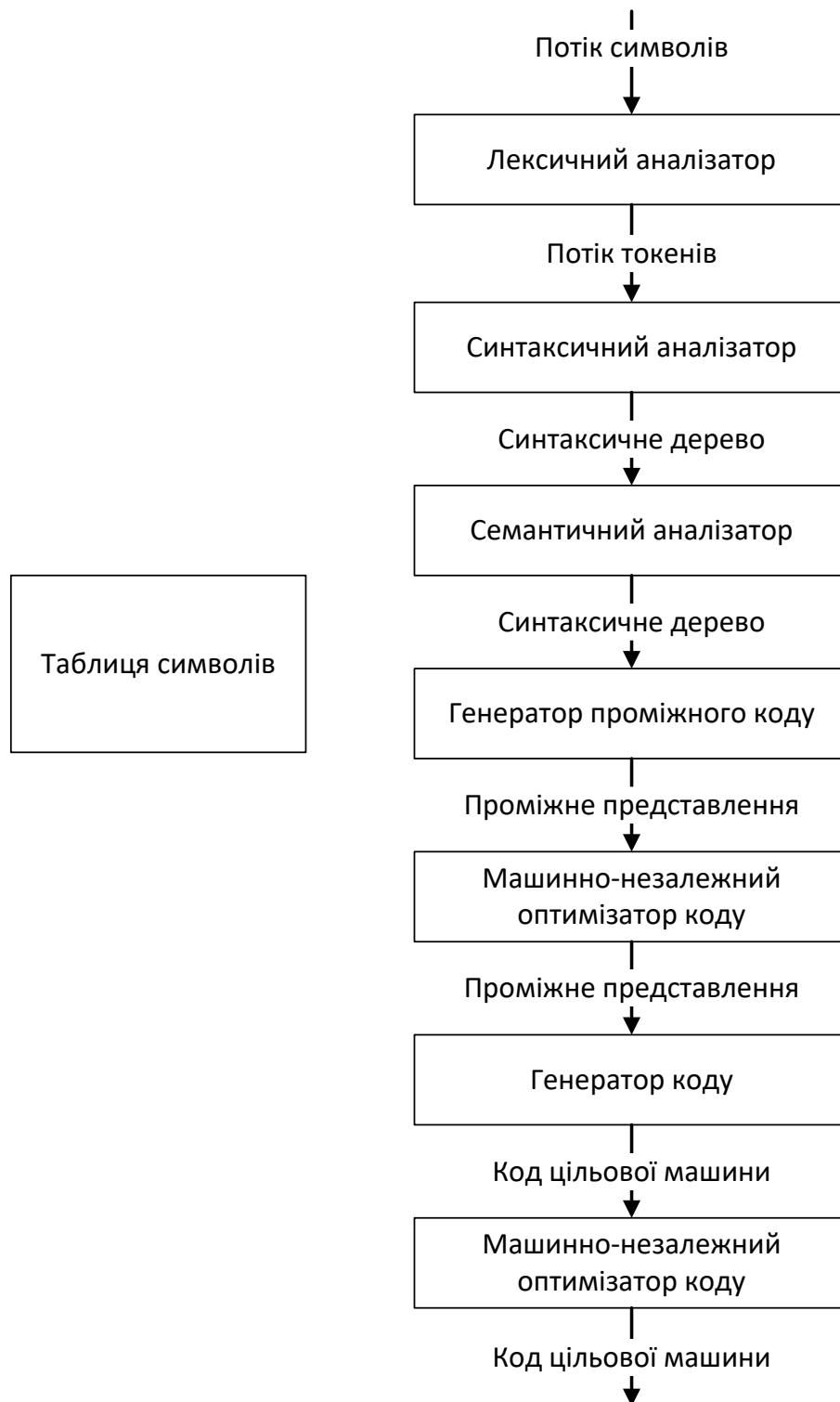


Рис 3. Фази транслятора

4 Програмна реалізація

4.1 Лексичний аналізатор

Лексичний аналізатор зчитує потік символів, з яких складається початкова програма, і групує їх в значущі послідовності, що називаються лексеми. Для кожної лексеми аналізатор будує вихідний токен вигляду:

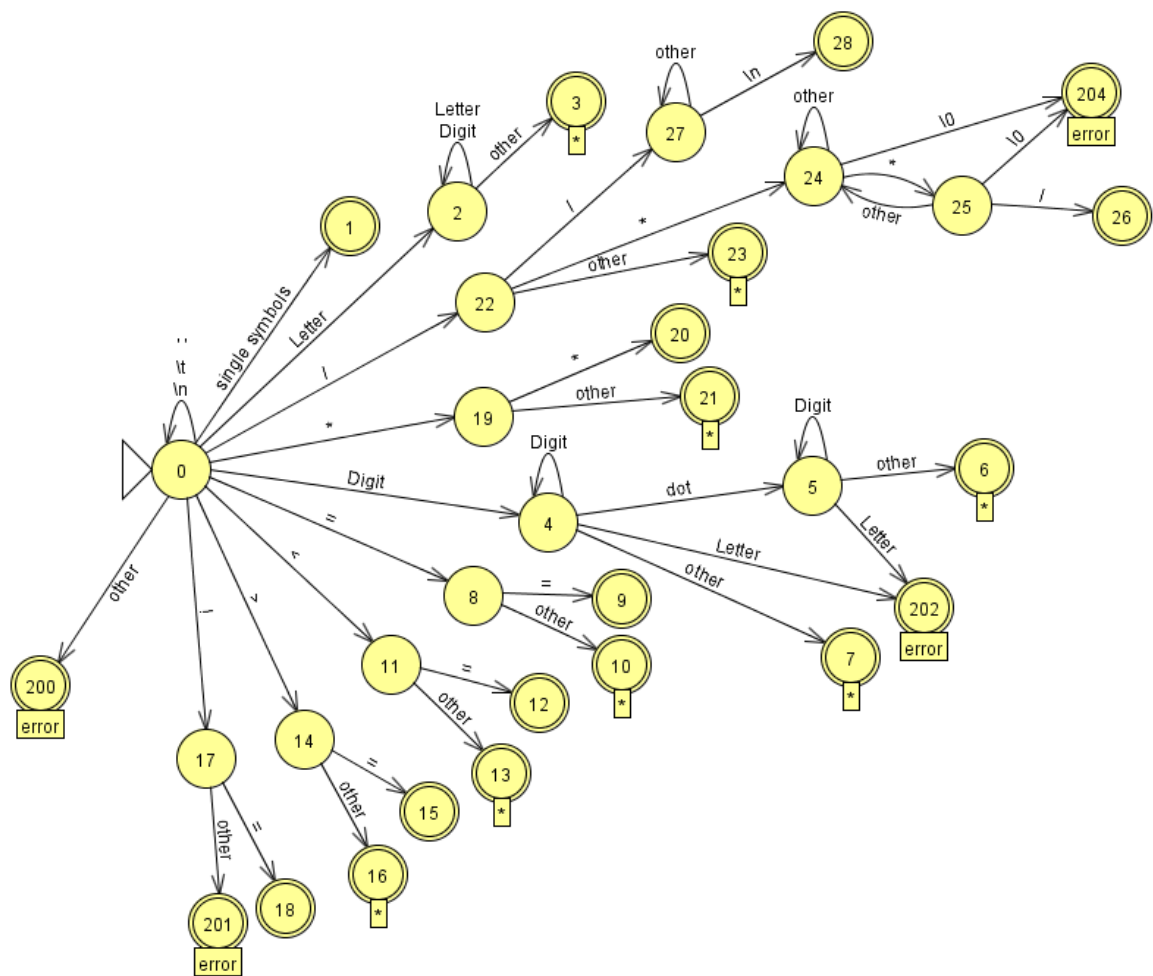
```
namespace Crundras.Common
{
    public struct Token
    {
        /// <summary>
        /// line of source file where was current token
        /// </summary>
        public uint Line { get; set; }

        /// <summary>
        /// token code in tokens table in specification
        /// </summary>
        public uint Code { get; set; }

        /// <summary>
        /// id of lexeme in otherwise table (or null)
        /// </summary>
        public uint? ForeignId { get; set; }
    }
}
```

Де Code – це код лексеми з Табл. 2., Line – це номер рядку в якому зустрівся токен та ForeignId – номер ідентифікатору чи літералу у їх таблицях.

Лексичний аналізатор реалізована через скінченний автомат. Для створення візуально відображення використовувалася програма JFLAP вер. 7. Схему приведено на рис. 4.



Умовне позначення ‘*’ позначає стан який не забирає символ з вхідного потоку, а ‘error’ – позначає стан з помилкою.

Синтаксичний аналіз – використовує токени виділені на етапі лексичного аналізу для побудови проміжного представлення деревовидного виду, що описує граматичну структуру потоку токенів. Типовими представленнями являються дерево розбору та абстрактне синтаксичне дерево.

Для реалізації транслятору мови Crundras було обрано абстрактне синтаксичне дерево оскільки воно відкидає токени, що не впливають на семантику програми. Саме дерево будується за допомогою алгоритму що використовує автомат з магазинною пам'яттю та під-автоматами. Схеми дивитись рис. 5 та 6.

Рис. 5: PDA for statement

Рис. 6: PDA for expression

4.3 Проміжна форма подання програми

Загалом працювати напряду з початковою програмою надзвичайно незручно від чого з'являється купа помилок. Мови програмування розробляються щоб бути зрозумілим людині і не залишають багатозначності місця, але не дуже зручні для обробки. Тому створюється проміжне представлення. У випадку транслятора, що розроблявся в рамках цієї курсової, як проміжне представлення було використано Польський Зворотній Запис.

Польський зворотній запис давно використовується при створенні мов програмування. Сам ПОЛІЗ був запропонований польським логіком Яном Лукашевичем у 1924р. з метою спрощення логіки висловлювань.

Поширення ПОЛІЗ набув з розвитком комп'ютерних наук. Широко використовувався у 1970-1980 роках такою відомою компанією як Hewlett-Packard в їх виробках і продовжували використовувати приблизно до 2010го року [4]. З його використанням були створенні граматики стек-орієнтовних мов програмування, таких як Forth, RPL та PostScript.

Для транслятору мови Crundras використовувалась реалізація перетворення арифметичних виразів за допомогою стеку і лінійна реалізація перетворення інших конструкцій мови. Пріоритети арифметичних виразів:

```
var priorityTable = new Dictionary<string, int> {
    { "(", 0 },
    { ")", 0 },
    { "<", 1 },
    { "<=", 1 },
    { ">", 1 },
    { ">=", 1 },
    { "=", 1 },
    { "!", 1 },
    { "+", 2 },
    { "-", 2 },
    { "*", 3 },
    { "/", 3 },
    { "%", 3 },
    { "**", 3 },
    { "NEG", 4 }
};
```

4.3.1 Expression

Tokens	Priority
(,)	0
<, <=, >, >=, ==, !=	1
+, -	2
*, /, %, **	3
Negation	4

Table 4: Priorities for RPN translator

Arithmetic expression is translated to RPN by Shunting-yard algorithm invented by Edsger W. Dijkstra.

4.3.2 Negation

Negation token is converted from '-' token while processing expression in case if previous token wasn't ')', literal or identifier.

4.3.3 Assignment statement

Syntax:

AssignmentExpression = Identifier '=' Expression.

RPN:

{identifier} Expression {=}

4.3.4 Output statement

Syntax:

OutputStatement = '@' Expression ';'.

RPN:

Expression {@}

4.3.5 Input statement

Syntax:

InputStatement = '\$' Identifier ';'.

RPN:

{identifier} {\$}

4.3.6 Declaration Statement

Syntax:

```
TypeSpecifier = "int" | "float".  
DeclarationStatement = TypeSpecifier Identifier ';'.
```

Comment:

On declaration statement translator doesn't create tokens. Just set types into identifiers table.

4.3.7 Label statement

Syntax:

```
LabelStatement = Identifier ':'.
```

RPN:

```
{label}
```

Comment:

Labels are separated from identifiers by semantic function while building syntax tree.

4.3.8 Jump statement

Syntax:

```
JumpStatement = 'goto' Identifier ';'.
```

RPN:

```
{label} {goto}
```

4.3.9 Conditional statement

Syntax:

```
ConditionStatement = "if" '(' Expression ')' Statement.
```

RPN:

```
Expression {label (1)} {if} Statement {label (1)}
```

Comment:

Labels names for conditional statement is created using symbols not allowed for user to avoid collisions.

4.3.10 Iteration Statement

Syntax:

IterationStatement = "for" AssignmentExpression "to" Expression1 "by" Expression2 "while" '(' Expression3 ')' Statement "rof" ';'.

RPN:

```
{identifier (1)} Expression {=}
{label (2)}
Expression {"0" (1)} {!=} {label (1)} {if}
Expression {identifier (1)} {>} {"0" (1)} {!=} {label (1)} {if}
Statement
Expression {identifier (1)} {+} {identifier (1)} {=}
{label (2)} {goto}
{label (1)}
```

Comments:

1. Simply saying cycle:

```
for <ident> = <expr-1> to <expr-2> by <expr-3> while(<expr-4>)
<statement>
rof;
```

is rewritten to such program:

```
<ident> = <expr-1>;
```

```
start:
```

```
if(<expr-4> != 0) goto end;
if(<ident> > <expr-2> != 0) goto end;
```

```
<statement>
```

```
<ident> = <ident> + <expr-3>;
goto start;
end:
```

2. Labels names for iteration statement is created using symbols not allowed for user to avoid collisions.

4.4 Interpreting

4.4.1 label, identifier, int_literal, float_literal

Each of them is just pushed to stack.

4.4.2 goto

Takes from stack one token which should be label and interpreting flow goes to label position.

4.4.3 if

Takes from stack two tokens: label and value. If value token is 0 then interpreting flow goes to label position.

4.4.4 @

Takes from stack one token which should be identifier, int literal or float literal and displays value from appropriate table into interpreter console.

4.4.5 \$

Takes from stack one token which should be identifier then reads from console value that try to parse depending on variable type and puts it into identifiers table.

4.4.6 =

Takes from stack identifier and value token. Checks types and if types matches - sets value into identifiers table.

4.4.7 NEG

Takes one value from stack (can be identifier, int or float) creates negated value of taken token and push created to stack.

4.4.8 +, -, *, **, /, %, <, <=, >, >=, ==, !=

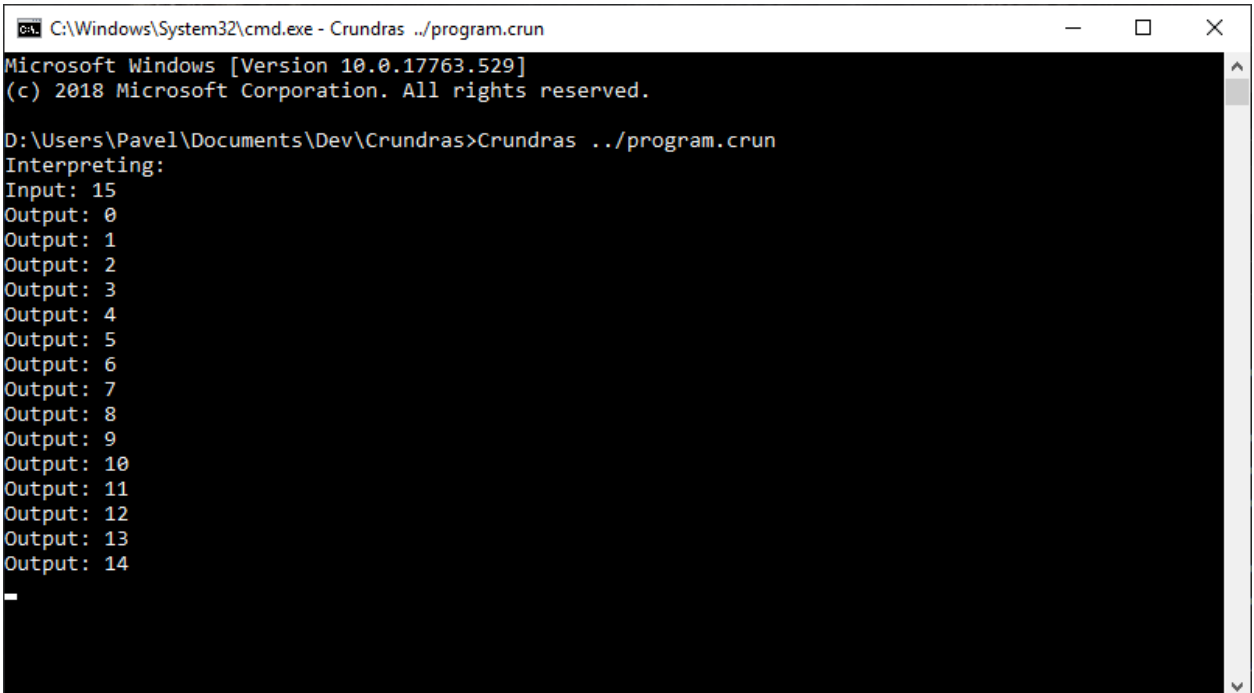
Takes two values from stack (each can be identifier, int or float) and calculates result by the rules specified for operator by language specification. Result is set in matching table and created RPN token that is pushed into stack.

5 Інструкція користувача по роботі з транслятором

Робота з розробленим транслятором здійснюється через командний рядок. Транслятор приймає ім'я файлу з програмою мовою Crundras як параметр. Тобто виклик транслятору буде виглядати так:

`crundras <filename>`

Приклад: рис. 9.



```
C:\Windows\System32\cmd.exe - Crundras ../program.crun
Microsoft Windows [Version 10.0.17763.529]
(c) 2018 Microsoft Corporation. All rights reserved.

D:\Users\Pavel\Documents\Dev\Crundras>Crundras ../program.crun
Interpreting:
Input: 15
Output: 0
Output: 1
Output: 2
Output: 3
Output: 4
Output: 5
Output: 6
Output: 7
Output: 8
Output: 9
Output: 10
Output: 11
Output: 12
Output: 13
Output: 14
-
```

Рис. 9: Приклад виклику транслятора з командного рядку

6 Тестування

Output explanation

Identifier tables consist of:

1. id
2. variable name
3. type id
4. value

Literals tables:

1. id
2. value

Labels table:

1. id
2. label name
3. label position

6.1 General arithmetic example:

Translating program:

```
@45 % 7 * 3 - 2**-2;
```

Expected reverse polish notation (RPN):

```
45 7 % 3 * 2 2 NEG ** - @
```

Calculating:

1. $45 \% 7 = 3$ (modulo operation)
2. $3 * 3 = 9$
3. $2 ** (-2) = 0.25$
4. $9 - 0.25 = 8.75$

As on step 3 power operation has float type, the type of output value should be float.

Translator output:

```
D:\Users\Pavel\Documents\KPI\Crundras\Crundras\RPNTTranslator\bin\Debug\netcoreapp3.1\RPNTTranslator.exe
RPN Tokens:
{"45" (1)} {"7" (2)} {"%" (3)} {"3" (3)} {"*" (4)} {"2" (4)} {"2" (4)} {NEG} {"**"} {"-"} {"@"}
Identifiers table:

IntLiteralsTable table:
  1      45
  2       7
  3       3
  4       2

FloatLiteralsTable table:

Labels table:
-
```

Interpreter output:

```
D:\Users\Pavel\Documents\KPI\Crundras\Crundras\RPNIInterpreter\bin\Debug\netcoreapp3.1\RPNIInterpreter.exe
Interpreting:
Output: 8.75
```

6.2 Power operation examples

6.2.1 Example 1

Translating program:

@7 ** 2.5;

Expected reverse polish notation (RPN):

7 2.5 ** @

Calculating:

7 ** 2.5 = 129.64181424216494

Translator output:

```
D:\Users\Pavel\Documents\KPI\Crundras\Crundras\RPNTTranslator\bin\Debug\netcoreapp3.1\RPNTTranslator.exe
RPN Tokens:
{"7" (1)} {"2.5" (1)} {"**"} {"@"}
Identifiers table:

IntLiteralsTable table:
  1      7

FloatLiteralsTable table:
  1      2.5

Labels table:
-
```

Interpreter output:

```
D:\Users\Pavel\Documents\KPI\Crundras\Crundras\RPNIInterpreter\bin\Debug\netcoreapp3.1\RPNIInterpreter.exe
Interpreting:
Output: 129.64
-
```

Here we have only two symbols after dot because of interpreter realization.

6.2.2 Example 2

Translating program:

@2 ** (5 - 2 ** -0.5);

Expected reverse polish notation (RPN):

2 5 2 0.5 NEG ** - ** @

Calculating:

1. $2 ** -0.5 = 0.7071067811865476$
2. $5 - 0.70 = 4.292893218813452$
3. $2 ** 4.3 = 19.601514449154106$

Translator output:

```
D:\Users\Pavel\Documents\KPI\Crundras\Crundras\RPNTranslator\bin\Debug\netcoreapp3.1\RPNTranslator.exe
RPN Tokens:
{"2" (1)} {"5" (2)} {"2" (1)} {"0.5" (1)} {NEG} {**} {-} {**} {@}
Identifiers table:

IntLiteralsTable table:
  1      2
  2      5

FloatLiteralsTable table:
  1      0.5

Labels table:
-
```

Interpreter output:

```
D:\Users\Pavel\Documents\KPI\Crundras\Crundras\RPNIInterpreter\bin\Debug\netcoreapp3.1\RPNIInterpreter.exe
Interpreting:
Output: 19.60
-
```

6.2.3 Example 3

Translating program:

@ (3.1 * 1.213 + 0.712) ** 4

Expected reverse polish notation (RPN):

3.1 1.213 * 0.712 + 4 ** @

Calculating:

1. $3.1 * 1.213 = 3.7603000000000004$
2. $(3.1 * 1.213 + 0.712) = 4.4723000000000001$
3. $4.4723000000000001 ** 4 = 400.05869375294014$

Translator output:

```
D:\Users\Pavel\Documents\KPI\Crundras\Crundras\RPNTranslator\bin\Debug\netcoreapp3.1\RPNTranslator.exe
RPN Tokens:
{"3.1" (1)} {"1.213" (2)} {"*"} {"0.712" (3)} {"+"} {"4" (1)} {"**"} {"@"}
Identifiers table:
IntLiteralsTable table:
  1      4
FloatLiteralsTable table:
  1      3.1
  2      1.213
  3      0.712
Labels table:
```

Interpreter output:

```
D:\Users\Pavel\Documents\KPI\Crundras\Crundras\RPNInterpreter\bin\Debug\netcoreapp3.1\RPNInterpreter.exe
Interpreting:
Output: 400.06
```

6.3 General logical operator examples

6.3.1 Example 1

Translating program:

```
@2 == 2;
```

```
@2 == 3;
```

```
@2 >= 3;
```

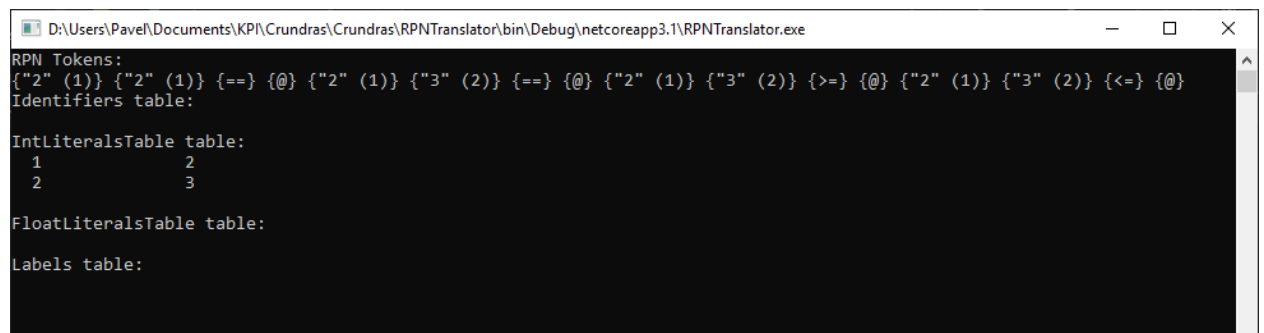
```
@2 <= 3;
```

Expected reverse polish notation (RPN):

```
2 2 == @ 2 3 == @ 2 3 >= @ 2 3 <= @
```

As it was declared in specification in case if expression is false result have to be 0, otherwise it is not 0 but, for example, 1.

Translator output:



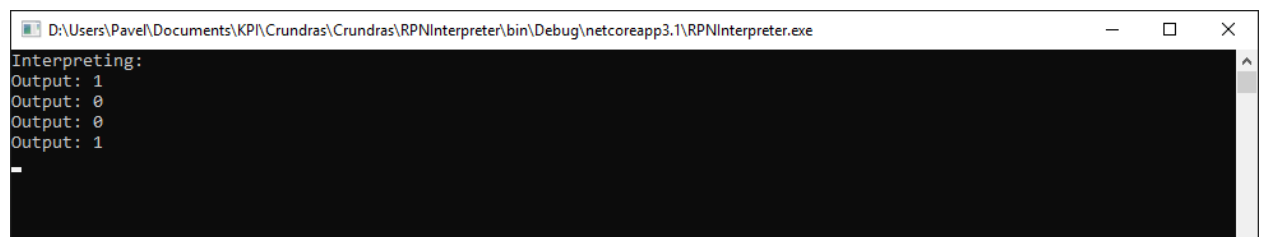
```
D:\Users\Pavel\Documents\KPI\Crundras\Crundras\RPNTranslator\bin\Debug\netcoreapp3.1\RPNTranslator.exe
RPN Tokens:
{"2" (1)} {"2" (1)} {"==" } {"@"} {"2" (1)} {"3" (2)} {"==" } {"@"} {"2" (1)} {"3" (2)} {">=" } {"@"} {"2" (1)} {"3" (2)} {"<=" } {"@"}
Identifiers table:

IntLiteralsTable table:
  1      2
  2      3

FloatLiteralsTable table:

Labels table:
```

Interpreter output:



```
D:\Users\Pavel\Documents\KPI\Crundras\Crundras\RPNInterpreter\bin\Debug\netcoreapp3.1\RPNInterpreter.exe
Interpreting:
Output: 1
Output: 0
Output: 0
Output: 1
```

6.3.2 Example 2

Translating program:

```
@16*16 == 2**4*2**4;
```

```
@16*16 != 2**4*2**4;
```

Translator output:

```
D:\Users\Pavel\Documents\KPI\Crundras\Crundras\RPNTTranslator\bin\Debug\netcoreapp3.1\RPNTTranslator.exe
RPN Tokens:
{"16" (1)} {"16" (1)} {"*" (2)} {"2" (2)} {"4" (3)} {"**" (2)} {"2" (2)} {"*" (2)} {"4" (3)} {"**" (2)} {"="} {"@"} {"16" (1)} {"16" (1)} {"*" (2)} {"2" (2)} {"4" (3)} {"**" (2)} {"2" (2)} {"*" (2)} {"4" (3)} {"**" (2)} {"!="} {"@"}
Identifiers table:
IntLiteralsTable table:
  1      16
  2       2
  3       4
FloatLiteralsTable table:
Labels table:
-
```

Interpreter output:

```
D:\Users\Pavel\Documents\KPI\Crundras\Crundras\RPNIInterpreter\bin\Debug\netcoreapp3.1\RPNIInterpreter.exe
Interpreting:
Output: 0
Output: 1
-
```

6.4 Assignment example

Translating program:

```
int a;
```

```
a = 46 + 95;
```

Translator output:

```
D:\Users\Pavel\Documents\KPI\Crundras\Crundras\RPNTTranslator\bin\Debug\netcoreapp3.1\RPNTTranslator.exe
RPN Tokens:
{"a" (1)} {"46" (1)} {"95" (2)} {"+" (2)} {"="}
Identifiers table:
  1      a  3
IntLiteralsTable table:
  1      46
  2      95
FloatLiteralsTable table:
Labels table:
-
```


Interpreter output:

```
D:\Users\Pavel\Documents\KPI\Crundras\Crundras\RPNIInterpreter\bin\Debug\netcoreapp3.1\RPNIInterpreter.exe
Interpreting:
Identifiers table:
  1          a  3      141

IntLiteralsTable table:
  1          46
  2          95
  3         141

FloatLiteralsTable table:

Labels table:
```

6.5 Input example

Translating program:

```
float d;
$d;
```

Translator output:

```
D:\Users\Pavel\Documents\KPI\Crundras\Crundras\RPNTranslator\bin\Debug\netcoreapp3.1\RPNTranslator.exe
RPN Tokens:
{"d" (1)} {$}
Identifiers table:
  1          d  4

IntLiteralsTable table:

FloatLiteralsTable table:

Labels table:
```

Interpreter output:

```
D:\Users\Pavel\Documents\KPI\Crundras\Crundras\RPNIInterpreter\bin\Debug\netcoreapp3.1\RPNIInterpreter.exe
Interpreting:
Input: 75
Identifiers table:
  1          d  4      75

IntLiteralsTable table:

FloatLiteralsTable table:

Labels table:
```

6.6 Output example

Translating program:

```
@75 ** 3;
```

Translator output:

```
D:\Users\Pavel\Documents\KPI\Crundras\Crundras\RPNTranslator\bin\Debug\netcoreapp3.1\RPNTranslator.exe
RPN Tokens:
{"75" (1)} {"3" (2)} {"**"} {"@"}
Identifiers table:

IntLiteralsTable table:
  1      75
  2      3
FloatLiteralsTable table:
Labels table:
```

Interpreter output:

```
D:\Users\Pavel\Documents\KPI\Crundras\Crundras\RPNInterpreter\bin\Debug\netcoreapp3.1\RPNInterpreter.exe
Interpreting:
Output: 421875.00
Identifiers table:

IntLiteralsTable table:
  1      75
  2      3
FloatLiteralsTable table:
  1      421875
Labels table:
```

6.7 Undefined variable error example

Translating program:

```
$d;
```

Translator output:

```
D:\Users\Pavel\Documents\KPI\Crundras\Crundras\RPNTranslator\bin\Debug\netcoreapp3.1\RPNTranslator.exe
RPN Tokens:
{"d" (1)} {"$"}
Identifiers table:
  1      d 0
IntLiteralsTable table:
FloatLiteralsTable table:
Labels table:
```

Interpreter output:

```
D:\Users\Pavel\Documents\KPI\Crundras\Crundras\RPNIInterpreter\bin\Debug\netcoreapp3.1\RPNIInterpreter.exe
Interpreting:
Undeclared variable "d".
```

1.1. Negation example

Translating program:

```
int a;
a = -42;
```

Translator output:

```
D:\Users\Pavel\Documents\KPI\Crundras\Crundras\RPNTranslator\bin\Debug\netcoreapp3.1\RPNTranslator.exe
RPN Tokens:
{"a" (1)} {"42" (1)} {NEG} {=}
Identifiers table:
1      a  3
IntLiteralsTable table:
1      42
FloatLiteralsTable table:
Labels table:
```

Interpreter output:

```
D:\Users\Pavel\Documents\KPI\Crundras\Crundras\RPNIInterpreter\bin\Debug\netcoreapp3.1\RPNIInterpreter.exe
Interpreting:
Identifiers table:
1      a  3      -42
IntLiteralsTable table:
1      42
2      -42
FloatLiteralsTable table:
Labels table:
```

6.8 If example

Translating program:

```
int a;  
$a;  
if (a) @a;
```

Translator output:

```
D:\Users\Pavel\Documents\KPI\Crundras\Crundras\RPNTTranslator\bin\Debug\netcoreapp3.1\RPNTTranslator.exe  
RPN Tokens:  
{"a" (1)} {$} {"a" (1)} {"_LB1" p(7) (1)} {if} {"a" (1)} {@} {"_LB1" p(7) (1)}  
Identifiers table:  
1 a 3  
IntLiteralsTable table:  
FloatLiteralsTable table:  
Labels table:  
1 _LB1 7
```

Interpreter output if 'a' not equal 0:

```
D:\Users\Pavel\Documents\KPI\Crundras\Crundras\RPNIInterpreter\bin\Debug\netcoreapp3.1\RPNIInterpreter.exe  
Interpreting:  
Input: 5  
Output: 5  
Identifiers table:  
1 a 3 5  
IntLiteralsTable table:  
FloatLiteralsTable table:  
Labels table:  
1 _LB1 7
```

Interpreter output if 'a' is equal 0:

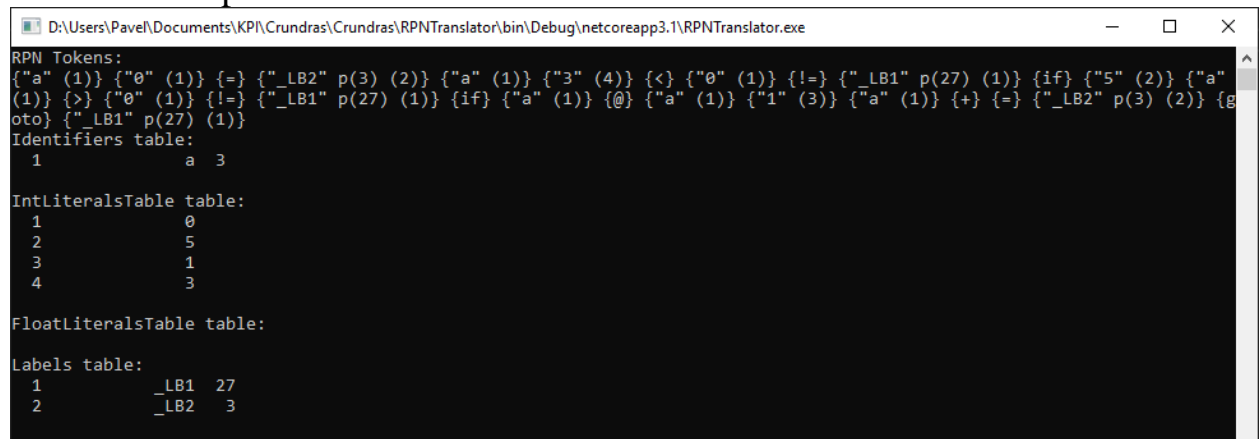
```
D:\Users\Pavel\Documents\KPI\Crundras\Crundras\RPNIInterpreter\bin\Debug\netcoreapp3.1\RPNIInterpreter.exe  
Interpreting:  
Input: 0  
Identifiers table:  
1 a 3 0  
IntLiteralsTable table:  
FloatLiteralsTable table:  
Labels table:  
1 _LB1 7
```

6.9 For example

Translating program:

```
int a;  
for a = 0 to 5 by 1 while(a < 3)  
    @a;  
rof;
```

Translator output:

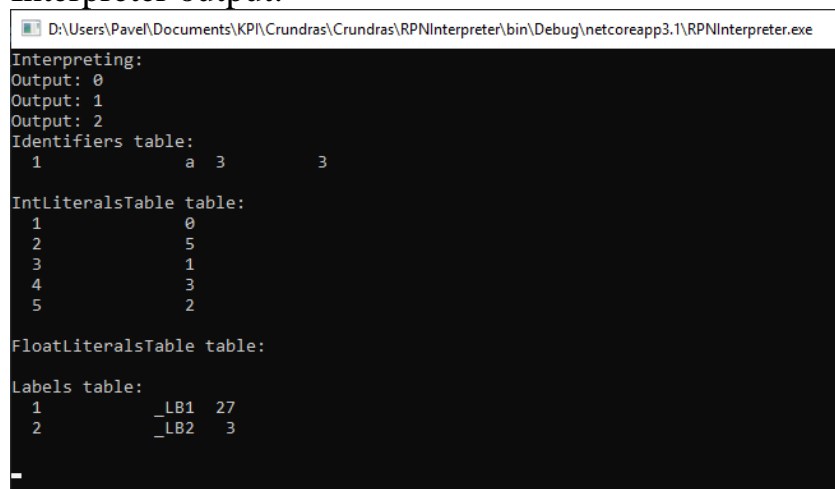


```
D:\Users\Pavel\Documents\KPI\Crundras\Crundras\RPNTTranslator\bin\Debug\netcoreapp3.1\RPNTTranslator.exe  
RPN Tokens:  
{ "a" (1) } { "0" (1) } { = } { "_LB2" p(3) (2) } { "a" (1) } { "3" (4) } { < } { "0" (1) } { != } { "_LB1" p(27) (1) } { if } { "5" (2) } { "a" (1) } { > } { "0" (1) } { != } { "_LB1" p(27) (1) } { if } { "a" (1) } { @ } { "a" (1) } { "1" (3) } { "a" (1) } { + } { = } { "_LB2" p(3) (2) } { goto } { "_LB1" p(27) (1) }  
Identifiers table:  
1 a 3  
IntLiteralsTable table:  
1 0  
2 5  
3 1  
4 3  
FloatLiteralsTable table:  
Labels table:  
1 _LB1 27  
2 _LB2 3
```

RPN tokens:

```
{ "a" (1) } { "0" (1) } { = } (assignment)  
{ "_LB2" p(3) (2) } (start label)  
{ "a" (1) } { "3" (4) } { < } { "0" (1) } { != } { "_LB1" p(27) (1) } { if } (goto end if)  
{ "5" (2) } { "a" (1) } { > } { "0" (1) } { != } { "_LB1" p(27) (1) } { if } (goto end if)  
{ "a" (1) } { @ } (statement)  
{ "a" (1) } { "1" (3) } { "a" (1) } { + } { = } (iteration)  
{ "_LB2" p(3) (2) } { goto } (goto start)  
{ "_LB1" p(27) (1) } (end label)
```

Interpreter output:



```
D:\Users\Pavel\Documents\KPI\Crundras\Crundras\RPNInterpreter\bin\Debug\netcoreapp3.1\RPNInterpreter.exe  
Interpreting:  
Output: 0  
Output: 1  
Output: 2  
Identifiers table:  
1 a 3 3  
IntLiteralsTable table:  
1 0  
2 5  
3 1  
4 3  
5 2  
FloatLiteralsTable table:  
Labels table:  
1 _LB1 27  
2 _LB2 3
```


Different interpreter output on input data:

Here we have 42 number in output as if branch (d == 23) is true. And don't have 34 before 42 as it was skipped by goto:

```
D:\Users\Pavel\Documents\KPI\Crundras\Crundras\RPNInterpreter\bin\Debug\netcoreapp3.1\RPNInterpreter.exe
Interpreting:
Output: 45
Input: 23
Output: 42
Output: 44
Output: 67
Output: 42
Output: 44
Output: 68
Output: 42
Output: 44
Output: 69
Output: 24.50
Output: 24.50
Output: 24.50
Output: 24.50
Output: 24.50
Output: 24.50
Output: 24.50
Output: 24.50
Output: 24.50
Output: 24.50
Output: 42
Input: 19
Output: 19
```

Tables after interpretation:

Identifiers table:

1	a	3	19
2	d	4	23
3	label	0	

Labels table:

1	label	100
2	_LB2	10
3	_LB3	58
4	_LB4	18
5	_LB5	46
6	_LB6	90
7	_LB7	62
8	_LB8	103

IntLiteralsTable table:

1	46
2	95
3	45
4	75
5	8
6	140
7	2
8	0
9	22
10	23
11	42
12	44
13	15
14	3
15	34
16	141
17	67
18	24
19	1

FloatLiteralsTable table:

1	0.5
2	1.5
3	70
4	1
5	68
6	69
7	5
8	24.5
9	2
10	2.5
11	3
12	3.5
13	4
14	4.5

Here we have non cycle execution as condition in while $(75 < 22 + 2)$ is false.

```
D:\Users\Pavel\Documents\KPI\Crundras\Crundras\RPNInterpreter\bin\Debug\netcoreapp3.1\RPNInterpreter...
Interpreting:
Output: 45
Input: 75
Input: 5
Output: 5
Identifiers table:
1          a  3          5
2          d  4          75
3          label 0

IntLiteralsTable table:
1          46
2          95
3          45
4          75
5           8
6         140
7           2
8           0
9          22
10         23
11         42
12         44
13         15
14          3
15         34
16        141
17         67
18         24

FloatLiteralsTable table:
1          0.5
2          1.5

Labels table:
1          label 100
2          _LB2  10
3          _LB3  58
4          _LB4  18
5          _LB5  46
6          _LB6  90
7          _LB7  62
8          _LB8 103
```

In case of inputting incorrect data we get error message.

```
D:\Users\Pavel\Documents\KPI\Crundras\Crundras\RPNInterpreter\bin\Debug\netcoreapp3.1\RPNInterpreter.exe
Interpreting:
Output: 45
Input: test
Input type error
```

Also in case trying to input float to int variable we have an error.

```
D:\Users\Pavel\Documents\KPI\Crundras\Crundras\RPNInterpreter\bin\Debug\netcoreapp3.1\RPNInterpreter.exe
Interpreting:
Output: 45
Input: 90
Input: 5.9
Input type error
```


ВИСНОВКИ

В даній курсовій роботі було розроблено Сі-подібну мову програмування Crundras, що підтримує арифметичні операції, виведення, введення, операторі умови та циклу.

Також до цієї мови було розроблено чотири-прохідний транслятор з такими фазами як лексичний аналіз, синтаксичний аналіз, генерація проміжного представлення та інтерпретація проміжного представлення. Для проміжного представлення програми було використано зворотною польську нотацію. Було наведено приклади роботи транслятора з різними вхідними програмами розробленої мови.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Медведєва, В. М. Транслятори: лексичний та синтаксичний аналізатори / В. М. Медведєва, В. А. Третяк. – К. : НТУУ "КПІ", 2012. – 148 с.
2. Медведєва, В. М. Транслятори: внутрішнє подання програм та інтерпретація / В. М. Медведєва, В. А. Третяк. – К. : НТУУ "КПІ", 2015. – 144 с.
3. Ахо Альфред, Лам Моника, Сети Рави, Ульман Джеффри Д. Компиляторы: принципы, технологии и инструментарий, 2-е изд. : Пер. с англ. – М. : ООО „И.Д. Вильямс”, 2008. – 1184 с.
4. Osborne, Thomas E. (2010) [1994]. "Tom Osborne's Story in His Own Words". Steve Leibson

ДОДАТОК

Повна граматика мови Crundras

УКР.НТУУ"КПІ"_ТЕФ_АПЕПС.ТР72262

Київ – 2020

`_Program = {Statement}.`

`Statement = InputStatement | OutputStatement | CompoundStatement |
ConditionStatement | IterationStatement | DeclarationStatement |
AssignmentStatement | JumpStatement | LabelStatement.
ConditionStatement = "if" '(' Expression ')' Statement.
IterationStatement = "for" AssignmentExpression "to" Expression "by" Expression
"while" '(' Expression ')' Statement "rof" ';'.
CompoundStatement = '{' {Statement} '}'.
DeclarationStatement = TypeSpecifier Identifier ';'.
AssignmentStatement = AssignmentExpression ';'.
Expression = [Sign] ('(' Expression ')' | Literal | Identifier) { Operator
[Sign] ('(' Expression ')' | Literal | Identifier) }.`

`AssignmentExpression = Identifier '=' Expression.`

`InputStatement = '$' Identifier ';'.
OutputStatement = '@' Expression ';'.
LabelStatement = Identifier ':'.
JumpStatement = 'goto' Identifier ';'.
TypeSpecifier = "int" | "float".
Identifier = Letter {Letter | Digit}.
Literal = IntegerLiteral | FloatingLiteral.
FloatingLiteral = IntegerLiteral '.' [IntegerLiteral].
IntegerLiteral = Digit {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'
| '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'.
Digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'.
Sign = '+' | '-'.
Operator = ArithmeticOperator | RelationalOperator.
ArithmeticOperator = '+' | '-' | '*' | "**" | '/' | '%'.
RelationalOperator = '<' | '>' | "<=" | ">=" | "==" | "!=".`

ДОДАТОК

Таблиця токенів мови Crundras

УКР.НТУУ”КПІ”_ТЕФ_АПЕПС.ТР72262

Київ – 2020

Code	Lexeme example	Token	Informal description
1	a, er4s2, qwe	identifier	ідентифікатор
2	asd, ew4, a43	label	назва мітки переходу
3	15, 413, 1000, 0	int_literal	ціле без знаку
4	3.75, 8., 78.987	float_literal	дійсне без знаку
5	int	int	термінал int
6	float	float	термінал float
7	if	if	термінал if
8	for	for	термінал for
9	to	to	термінал to
10	by	by	термінал by
11	while	while	термінал while
12	rof	rof	термінал rof
13	goto	goto	термінал goto
14	\$	\$	термінал \$
15	@	@	термінал @
16	=	=	термінал =
17	+	+	термінал +
18	-	-	термінал -
19	*	*	термінал *
20	**	**	термінал **
21	/	/	термінал /
22	%	%	термінал %
23	<	<	термінал <
24	>	>	термінал >
25	<=	<=	термінал <=
26	==	==	термінал =
27	>=	>=	термінал >=
28	!=	!=	термінал !=
29	((термінал (
30))	термінал)
31	{	{	термінал {
32	}	}	термінал }
33	;	;	термінал ;
34	:	:	термінал :
35	' ', \t	white_space	пробільний символ
36	\n, \r\n, \r	eol	end of line
37	\u0000	eof	end of file

Табл. 2: Таблица токенів мови Cgrundras

ДОДАТОК

Лістинг програмного продукту

УКР.НТУУ”КП”_ТЕФ_АПЕПС.ТР72262

Листів 33

Київ – 2020

Full source code can be found on:

<https://github.com/Dilorfin/Crundras/tree/coursework>

Token.cs

```
namespace Crundras.Common
{
    public struct Token
    {
        /// <summary>
        /// line of source file where was current token
        /// </summary>
        public uint Line { get; set; }

        /// <summary>
        /// token code in tokens table in specification
        /// </summary>
        public uint Code { get; set; }

        /// <summary>
        /// id of lexeme in otherwise table (or null)
        /// </summary>
        public uint? ForeignId { get; set; }

        public static bool IsIntLiteral(uint code)
        {
            return code == 3;
        }
        public static bool IsFloatLiteral(uint code)
        {
            return code == 4;
        }
        public static bool IsLiteral(uint code)
        {
            return IsFloatLiteral(code) || IsIntLiteral(code);
        }
        public static bool IsIdentifier(uint code)
        {
            return code == 1;
        }
        public static bool IsIdentifierOrLiteral(uint code)
        {
            return IsLiteral(code) || IsIdentifier(code);
        }
        public static bool IsLabel(uint code)
        {
            return code == 2;
        }
    }
}
```

SyntaxTreeNode.cs

```
using System.Collections.Generic;

namespace Crundras.Common
{
    public class SyntaxTreeNode
    {

```



```

    /// <summary>
    /// parent node
    /// </summary>
    public SyntaxTreeNode Parent { get; private set; }

    /// <summary>
    /// list of child nodes
    /// </summary>
    public List<SyntaxTreeNode> Children { get; private set; }

    /// <summary>
    /// lexeme code
    /// </summary>
    public uint LexemeCode { get; }

    /// <summary>
    /// id of identifier or literal
    /// </summary>
    public uint? Id { get; }

    public SyntaxTreeNode(uint lexemeCode, uint? id = null)
    {
        LexemeCode = lexemeCode;
        Id = id;
    }

    public SyntaxTreeNode(Token token)
    {
        LexemeCode = token.Code;
        Id = token.ForeignId;
    }

    public void AddChild(SyntaxTreeNode node)
    {
        Children ??= new List<SyntaxTreeNode>();

        node.SetParent(this);
        Children.Add(node);
    }

    public void AddChildren(IEnumerable<SyntaxTreeNode> nodes)
    {
        foreach (var node in nodes)
        {
            AddChild(node);
        }
    }

    private void SetParent(SyntaxTreeNode parent)
    {
        Parent = parent;
    }
}

```

LiteralsTable.cs

```

using System;
using System.Collections.Generic;

```

```

namespace Crundras.Common.Tables.Literals
{
    public abstract class LiteralsTable<TValue>
    {
        protected readonly Dictionary<uint, TValue> Values = new Dictionary<uint,
TValue>();

        public TValue this[uint index]
        {
            get => Values[index];
            set => Values[index] = value;
        }

        public abstract uint GetId(string lexeme);
        public abstract uint GetId(TValue value);

        public void Display()
        {
            Console.WriteLine($"{GetType().Name} table:");
            foreach (var value in Values)
            {
                Console.WriteLine($"{value.Key,3}{value.Value,15}");
            }
            Console.WriteLine();
        }
    }
}

```

IntLiteralTable.cs

```

using System.Linq;

namespace Crundras.Common.Tables.Literals
{
    public class IntLiteralTable : LiteralsTable<int>
    {
        public override uint GetId(string lexeme)
        {
            return int.TryParse(lexeme, out int value) ? GetId(value) : 0;
        }

        public override uint GetId(int value)
        {
            if (Values.ContainsValue(value))
            {
                return Values.First(
                    v => v.Value == value
                ).Key;
            }

            var id = (uint)(Values.Count + 1);
            Values[id] = value;
            return id;
        }
    }
}

```

FloatLiteralTable.cs

```
using System;
using System.Linq;

namespace Crundras.Common.Tables.Literals
{
    public class FloatLiteralTable : LiteralsTable<float>
    {
        public override uint GetId(string lexeme)
        {
            return float.TryParse(lexeme, out float value) ? GetId(value) : 0;
        }

        public override uint GetId(float value)
        {
            if (Values.ContainsValue(value))
            {
                return Values.First(
                    v => Math.Abs(v.Value - value) < float.Epsilon
                ).Key;
            }

            var id = (uint)(Values.Count + 1);
            Values[id] = value;
            return id;
        }
    }
}
```

IdentifiersTable.cs

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace Crundras.Common.Tables
{
    public class IdentifiersTable
    {
        public class Identifier
        {
            public uint Type { get; set; }
            public string Name { get; set; }
            public string Value { get; set; }
        }

        private readonly Dictionary<uint, Identifier> identifiers = new Dictionary<uint, Identifier>();

        public Identifier this[uint index] => identifiers.ContainsKey(index) ? identifiers[index] : null;

        public uint GetId(string name)
        {
            var (key, value) = identifiers.FirstOrDefault(p => p.Value.Name == name);
            if (value != null)
            {
                return key;
            }
        }
    }
}
```

```

    }

    var index = (uint)(identifiers.Count + 1);
    identifiers[index] = new Identifier { Name = name };
    return index;
}

public void Display()
{
    Console.WriteLine("Identifiers table:");
    foreach (var (key, value) in identifiers)
    {
        Console.WriteLine($"{key,3}{value.Name,15}{value.Type,3}{value.Value,10}");
    }
    Console.WriteLine();
}
}
}

```

LabelsTable.cs

```

using System;
using System.Collections.Generic;
using System.Linq;

namespace Crundras.Common.Tables
{
    public class LabelsTable
    {
        public class Label
        {
            public string Name { get; set; }
            public uint Position { get; set; }
        }

        private readonly Dictionary<uint, Label> labels = new Dictionary<uint, Label>();

        public Label this[uint index] => labels.ContainsKey(index) ? labels[index] :
        null;

        public int Count => labels.Count;

        public uint GetId(string lexeme)
        {
            var label = labels.FirstOrDefault(p => p.Value.Name == lexeme);
            if (label.Value != null)
            {
                return label.Key;
            }

            var index = (uint)(labels.Count + 1);
            labels[index] = new Label { Name = lexeme };
            return index;
        }

        public void Display()
        {
            Console.WriteLine("Identifiers table:");

```

```

        foreach (var (key, value) in labels)
        {
            Console.WriteLine($"{key,3}{value.Name,15}{value.Position,3}");
        }
        Console.WriteLine();
    }
}

```

LexemsTable.cs

```

using System.Collections.Generic;
using System.Linq;

namespace Crundras.Common.Tables
{
    public static class LexemesTable
    {
        private static readonly Dictionary<uint, string> CodesTable = new Dictionary<uint, string>
        {
            { 1, "identifier"},
            { 2, "label"},
            { 3, "int_literal"},
            { 4, "float_literal"},
            { 5, "int"},
            { 6, "float"},
            { 7, "if"},
            { 8, "for"},
            { 9, "to"},
            { 10, "by"},
            { 11, "while"},
            { 12, "rof"},
            { 13, "goto"},
            { 14, "$"},
            { 15, "@"},
            { 16, "="},
            { 17, "+"},
            { 18, "-"},
            { 19, "*"},
            { 20, "**"},
            { 21, "/"},
            { 22, "%"},
            { 23, "<"},
            { 24, ">"},
            { 25, "<="},
            { 26, "=="},
            { 27, ">="},
            { 28, "!="},
            { 29, "("},
            { 30, ")"},
            { 31, "{"},
            { 32, "}"},
            { 33, ";"},
            { 34, ":"}
        };

        public static bool IsKeyword(string lexeme)
        {

```

```

        var lexemeId = GetLexemeId(lexeme);
        return lexemeId >= 5 && lexemeId <= 34;
    }

    public static string GetLexemeName(uint code)
    {
        return CodesTable[code];
    }

    public static uint GetLexemeId(string lexeme)
    {
        if (!CodesTable.ContainsValue(lexeme))
        {
            return 0;
        }

        return CodesTable.First(p => p.Value == lexeme).Key;
    }
}

```

TablesCollection.cs

```

using Crundras.Common.Tables.Literals;

namespace Crundras.Common.Tables
{
    public class TablesCollection
    {
        public TokenTable TokenTable { get; }
        public FloatLiteralsTable FloatLiteralsTable { get; }
        public IntLiteralsTable IntLiteralsTable { get; }
        public IdentifiersTable IdentifiersTable { get; }
        public LabelsTable LabelsTable { get; }

        public TablesCollection()
        {
            this.FloatLiteralsTable = new FloatLiteralsTable();
            this.IntLiteralsTable = new IntLiteralsTable();
            this.IdentifiersTable = new IdentifiersTable();
            this.TokenTable = new TokenTable();
            this.LabelsTable = new LabelsTable();
        }
    }
}

```

TokenTable.cs

```

using System.Collections.Generic;

namespace Crundras.Common.Tables
{
    public class TokenTable : LinkedList<Token>
    {
    }
}

```

LexicalAnalyzer.cs

```
using Crundras.Common;
using Crundras.Common.Tables;
using Crundras.LexicalAnalyzer.FSM;
using System;
using System.IO;
using System.Text;
using System.Text.RegularExpressions;

namespace Crundras.LexicalAnalyzer
{
    public class LexicalAnalyzer
    {
        private readonly StringBuilder stringBuilder;
        private readonly StateMachine stateMachine;
        private State currentState => stateMachine.CurrentState;

        public TablesCollection Tables { get; }

        private uint line = 1;

        private LexicalAnalyzer()
        {
            this.Tables = new TablesCollection();

            this.stateMachine = new StateMachine();
            this.stringBuilder = new StringBuilder();
        }

        private void NextChar(char nextChar)
        {
            do
            {
                int charClass = GetCharClass(nextChar);
                // transiting state machine to next state
                stateMachine.NextState(charClass);

                // checking if error has occurred
                if (currentState.IsError)
                {
                    // character escaping
                    string character = Regex.Escape(new string(nextChar, 1));
                    // displaying error message
                    string message = (currentState as ErrorState)?.Message;

                    throw new Exception($"{message}. Character: '{character}'. Line:
{line}." );
                }

                // counting lines
                if (nextChar == '\n')
                {
                    line++;
                }

                // checking for '*' states
                if (currentState.TakeCharacter)

```

```

        {
            stringBuilder.Append(nextChar);
        }
        // adding lexeme to table in final states
        if (CurrentState.IsFinal)
        {
            var lexeme = stringBuilder.ToString();
            AddLexeme(lexeme, CurrentState.Id);
            stringBuilder.Clear();
        }

        // clearing builder at 0 state, needed in case of self transiting
        if (CurrentState.Id == 0)
        {
            stringBuilder.Clear();
        }
    }
    while (!CurrentState.TakeCharacter);
}

public static TablesCollection AnalyzeFile(string fileName)
{
    using var file = new StreamReader(fileName, true);
    var analyzer = new LexicalAnalyzer();

    while (!file.EndOfStream)
    {
        analyzer.NextChar((char)file.Read());
    }

    // '\0' - last character
    analyzer.NextChar('\0');

    file.Close();

    return analyzer.Tables;
}

private void AddLexeme(string lexeme, int stateId)
{
    // skip comments
    if (stateId == 26 || stateId == 28)
    {
        return;
    }

    var token = new Token { Line = line, ForeignId = null };

    // checking if lexeme is language specific
    if (LexemesTable.IsKeyword(lexeme))
    {
        token.Code = LexemesTable.GetLexemeId(lexeme);
    }
    else
    {
        // identifiers
        if (stateId == 3)
        {
            token.Code = LexemesTable.GetLexemeId("identifier");
            token.ForeignId = Tables.IdentifiersTable.GetId(lexeme);
        }
    }
}

```



```

        // int literal
        else if (stateId == 7)
        {
            token.Code = LexemesTable.GetLexemeId("int_literal");
            token.ForeignId = Tables.IntLiteralsTable.GetId(lexeme);
        }
        // float literal
        else
        {
            token.Code = LexemesTable.GetLexemeId("float_literal");
            token.ForeignId = Tables.FloatLiteralsTable.GetId(lexeme);
        }
    }

    Tables.TokenTable.AddLast(token);
}

private static int GetCharClass(char c)
{
    int charClass = c;
    if (char.IsLetter(c))
    {
        charClass = 1;
    }
    else if (char.IsDigit(c))
    {
        charClass = 2;
    }
    else if (char.IsWhiteSpace(c) && !("\r\n").Contains(c))
    {
        charClass = 3;
    }

    return charClass;
}
}
}

```

ErrorState.cs

```

namespace Crundras.LexicalAnalyzer.FSM
{
    internal class ErrorState : State
    {
        public string Message { get; }

        public ErrorState(int id, string message)
            : base(id, true, false)
        {
            this.Message = message;
        }
    }
}

```

State.cs

```

using System.Collections.Generic;

namespace Crundras.LexicalAnalyzer.FSM

```

```

{
    internal class State
    {
        public int Id { get; }

        private readonly Dictionary<int, int> transitions = new Dictionary<int, int>();
        private int? otherwiseId;

        public State(int id, bool isFinal = false, bool takeCharacter = true)
        {
            this.Id = id;
            this.IsFinal = isFinal;
            this.TakeCharacter = takeCharacter;
        }

        public State ConfigureTransition(int charClass, int nextStateId)
        {
            if (!CanTransit(charClass))
            {
                transitions.Add(charClass, nextStateId);
            }

            return this;
        }

        public State ConfigureOtherwiseTransition(int nextStateId)
        {
            otherwiseId = nextStateId;
            return this;
        }

        public State ConfigureSelfTransition(int charClass)
        {
            return ConfigureTransition(charClass, this.Id);
        }

        public State ConfigureOtherwiseSelfTransition()
        {
            return ConfigureOtherwiseTransition(this.Id);
        }

        public int? Transit(int charClass)
        {
            if (CanTransit(charClass))
            {
                return transitions[charClass];
            }

            return otherwiseId;
        }

        private bool CanTransit(int charClass)
        {
            return transitions.ContainsKey(charClass);
        }

        public bool IsError => this.GetType() == typeof(ErrorState);
        public bool IsFinal { get; }
        public bool TakeCharacter { get; }
    }
}

```

StateMachine.cs

```
using System;
using System.Collections.Generic;

namespace Crundras.LexicalAnalyzer.FSM
{
    internal partial class StateMachine
    {
        private readonly Dictionary<int, State> states = new Dictionary<int, State>();
        public State CurrentState { get; private set; }

        public StateMachine()
        {
            Config();
        }

        public void NextState(int charClass)
        {
            // refreshing state in final state
            if (CurrentState.IsFinal)
            {
                CurrentState = states[0];
            }

            var nextStateId = CurrentState.Transit(charClass);

            if (!nextStateId.HasValue)
            {
                throw new Exception("Unconfigured transition");
            }

            // transiting to next state
            CurrentState = states[nextStateId.Value];
        }
    }
}
```

StateMachine.cs

```
namespace Crundras.LexicalAnalyzer.FSM
{
    internal partial class StateMachine
    {
        private void Config()
        {
            // final for single character lexemes
            states[1] = new State(1, true);

            // numbers
            states[6] = new State(6, true, false);
            states[7] = new State(7, true, false);

            states[202] = new ErrorState(202, "Unexpected character");

            states[5] = new State(5)
                .ConfigureSelfTransition(2)
                .ConfigureTransition(1, 202)

```

```

        .ConfigureOtherwiseTransition(6);

states[4] = new State(4)
    .ConfigureSelfTransition(2)
    .ConfigureTransition('.', 5)
    .ConfigureTransition(1, 202)
    .ConfigureOtherwiseTransition(7);

// identifiers
states[3] = new State(3, true, false);

states[2] = new State(2)
    .ConfigureSelfTransition(1)
    .ConfigureSelfTransition(2)
    .ConfigureOtherwiseTransition(3);

// == & =
states[9] = new State(9, true);
states[10] = new State(10, true, false);

states[8] = new State(8)
    .ConfigureTransition('=', 9)
    .ConfigureOtherwiseTransition(10);

// <= & <
states[12] = new State(12, true);
states[13] = new State(13, true, false);

states[11] = new State(11)
    .ConfigureTransition('=', 12)
    .ConfigureOtherwiseTransition(13);

// >= & >
states[15] = new State(15, true);
states[16] = new State(16, true, false);

states[14] = new State(14)
    .ConfigureTransition('=', 15)
    .ConfigureOtherwiseTransition(16);

// !=
states[18] = new State(18, true);

states[201] = new ErrorState(201, "Expected '='");
states[17] = new State(17)
    .ConfigureTransition('=', 18)
    .ConfigureOtherwiseTransition(201);

// ** & *
states[20] = new State(20, true);
states[21] = new State(21, true, false);

states[19] = new State(19)
    .ConfigureTransition('*', 20)
    .ConfigureOtherwiseTransition(21);

// comments
states[204] = new ErrorState(204, "Expected end of the comment (\"*/\")");

states[28] = new State(28, true);

```

```

states[27] = new State(27)
    .ConfigureTransition('\n', 28)
    .ConfigureTransition('\r', 28)
    .ConfigureOtherwiseSelfTransition();

states[26] = new State(26, true);

states[25] = new State(25)
    .ConfigureTransition('\0', 204)
    .ConfigureTransition('/', 26)
    .ConfigureOtherwiseTransition(24);

states[24] = new State(24)
    .ConfigureTransition('\0', 204)
    .ConfigureTransition('*', 25)
    .ConfigureOtherwiseSelfTransition();

states[23] = new State(23, true, false);

states[22] = new State(22)
    .ConfigureTransition('*', 24)
    .ConfigureTransition('/', 27)
    .ConfigureOtherwiseTransition(23);

states[200] = new ErrorState(200, "Unknown symbol");

// I am root
states[0] = new State(0)
    .ConfigureSelfTransition(3)
    .ConfigureSelfTransition('\0')
    .ConfigureSelfTransition('\r')
    .ConfigureSelfTransition('\n')
    .ConfigureTransition('$', 1)
    .ConfigureTransition('@', 1)
    .ConfigureTransition('+', 1)
    .ConfigureTransition('-', 1)
    .ConfigureTransition('%', 1)
    .ConfigureTransition('(', 1)
    .ConfigureTransition(')', 1)
    .ConfigureTransition('{', 1)
    .ConfigureTransition('}', 1)
    .ConfigureTransition('; ', 1)
    .ConfigureTransition(':', 1)
    .ConfigureTransition(1, 2)
    .ConfigureTransition(2, 4)
    .ConfigureTransition('=', 8)
    .ConfigureTransition('<', 11)
    .ConfigureTransition('>', 14)
    .ConfigureTransition('!', 17)
    .ConfigureTransition('*', 19)
    .ConfigureTransition('/', 22)
    .ConfigureOtherwiseTransition(200);

CurrentState = states[0];
    }
}
}

```

SyntaxAnalyzerPDA.cs

```
using Crundras.Common;
using Crundras.Common.Tables;
using SyntaxAnalyzerPDA.PDA;
using System;
using System.Collections.Generic;

namespace SyntaxAnalyzerPDA
{
    public class SyntaxAnalyzer
    {
        private readonly StateMachine stateMachine;
        private readonly TablesCollection tables;

        private LinkedListNode<Token> tokenListNode;

        public SyntaxTreeNode RootTreeNode { get; }
        public SyntaxTreeNode CurrentTreeNode { get; private set; }

        public SyntaxAnalyzer(TablesCollection tables)
        {
            this.tables = tables;
            tokenListNode = tables.TokenTable.First;

            RootTreeNode = new SyntaxTreeNode(uint.MaxValue);
            CurrentTreeNode = RootTreeNode;

            stateMachine = new StateMachine();
        }

        public static SyntaxTreeNode Analyze(TablesCollection tables)
        {
            var analyzer = new SyntaxAnalyzer(tables);
            analyzer.Analyze();
            return analyzer.RootTreeNode;
        }

        private void Analyze()
        {
            while (tokenListNode != null)
            {
                stateMachine.NextState(tokenListNode.Value.Code);

                if (stateMachine.CurrentState.IsError)
                {
                    var message = (stateMachine.CurrentState as ErrorState)?.Message;
                    throw new Exception(message);
                }

                if ((stateMachine.CurrentState.Id == 11
                    && tokenListNode?.Next?.Value.Code == LexemesTable.GetLexemeId(":"))
                    || stateMachine.CurrentState.Id == 8)
                {
                    var value = tokenListNode.Value;

                    value.Code = LexemesTable.GetLexemeId("label");

                    var identifier = tables.IdentifiersTable[value.ForeignId.Value];
                    value.ForeignId = tables.LabelsTable.GetId(identifier.Name);
                }
            }
        }
    }
}
```

```

        tokenListNode.Value = value;
    }

    SyntaxTreeBuilder();

    if (stateMachine.CurrentState.TakeToken)
    {
        tokenListNode = tokenListNode.Next;
    }
}

public void SyntaxTreeBuilder()
{
    if (stateMachine.CurrentState.Id == 105)
    {
        CurrentTreeNode = CurrentTreeNode.Parent;
        if (CurrentTreeNode.LexemeCode != LexemesTable.GetLexemeId("if"))
        {
            CurrentTreeNode = CurrentTreeNode.Parent;
        }
    }

    if (CurrentTreeNode.LexemeCode == LexemesTable.GetLexemeId("if")
        && CurrentTreeNode.Children?.Count == 2)
    {
        CurrentTreeNode = CurrentTreeNode.Parent;
    }

    if (!stateMachine.CurrentState.TakeToken)
    {
        return;
    }

    if (tokenListNode.Value.Code == LexemesTable.GetLexemeId(":"))
    {
        CurrentTreeNode.AddChild(new SyntaxTreeNode(tokenListNode.Previous.Value));
    }
    else if (tokenListNode.Value.Code == LexemesTable.GetLexemeId("="))
    {
        CurrentTreeNode.AddChild(new SyntaxTreeNode(tokenListNode.Value));
        CurrentTreeNode = CurrentTreeNode.Children[^1];
        CurrentTreeNode.AddChild(new SyntaxTreeNode(tokenListNode.Previous.Value));
    }
    else if (new List<uint> {
        LexemesTable.GetLexemeId("if"),
        LexemesTable.GetLexemeId("{"),
        LexemesTable.GetLexemeId "@"),
        LexemesTable.GetLexemeId("for"),
        LexemesTable.GetLexemeId("to"),
        LexemesTable.GetLexemeId("by"),
        LexemesTable.GetLexemeId("while")
    }.Contains(tokenListNode.Value.Code))
    {
        CurrentTreeNode.AddChild(new SyntaxTreeNode(tokenListNode.Value));
        CurrentTreeNode = CurrentTreeNode.Children[^1];
    }
    else if (tokenListNode.Value.Code == LexemesTable.GetLexemeId("}"))

```

```

        {
            CurrentTreeNode = CurrentTreeNode.Parent;
        }
        else if (tokenListNode.Value.Code == LexemesTable.GetLexemeId("rof"))
        {
            CurrentTreeNode = CurrentTreeNode.Parent;
        }

        if (new List<int>{ 1, 4, 7 }.Contains(stateMachine.CurrentState.Id))
        {
            CurrentTreeNode.AddChild(new SyntaxTreeNode(tokenListNode.Value));
            CurrentTreeNode = CurrentTreeNode.Children[^1];
        }
        else if (new List<int>{ 2, 5, 8 }.Contains(stateMachine.CurrentState.Id))
        {
            CurrentTreeNode.AddChild(new SyntaxTreeNode(tokenListNode.Value));
            CurrentTreeNode = CurrentTreeNode.Parent;
        }
        else if (stateMachine.CurrentState.Id == 100)
        {
            CurrentTreeNode.AddChild(new SyntaxTreeNode(100));
            CurrentTreeNode = CurrentTreeNode.Children[^1];
        }
        else if (stateMachine.CurrentState.Id > 100)
        {
            CurrentTreeNode.AddChild(new SyntaxTreeNode(tokenListNode.Value));
        }
    }
}
}

```

ErrorState.cs

```

namespace SyntaxAnalyzerPDA.PDA
{
    public class ErrorState : State
    {
        public string Message { get; }

        public ErrorState(int id, string message)
            : base(id, null, true)
        {
            this.Message = message;
        }
    }
}

```

State.cs

```

using System;
using System.Collections.Generic;

namespace SyntaxAnalyzerPDA.PDA
{
    public class State
    {
        private class TransitionUnit
        {
            public TransitionUnit(int? pop, int? push, int stateId)

```



```

        {
            Pop = pop;
            Push = push;
            StateId = stateId;
        }

        public int? Pop { get; }
        public int? Push { get; }

        public int StateId { get; }
    }

    public int Id { get; }

    private readonly Stack<int> stack;

    private readonly Dictionary<uint, TransitionUnit> transitions = new Dictionary<uint, TransitionUnit>();
    private TransitionUnit otherwise = null;

    public State(int id, Stack<int> stack, bool isFinal = false, bool takeToken =
true)
    {
        this.Id = id;
        this.stack = stack;
        this.TakeToken = takeToken;
        this.IsFinal = isFinal;
    }

    #region CONFIGS
    public State ConfigureTransition(uint tokenType, int nextStateId, int? pop =
null, int? push = null)
    {
        if (CanTransit(tokenType))
        {
            return this;
        }

        var transitionUnit = new TransitionUnit(pop, push, nextStateId);
        transitions.Add(tokenType, transitionUnit);

        return this;
    }
    public State ConfigureSelfTransition(uint tokenType, int? pop = null, int? push =
null)
    {
        if (CanTransit(tokenType))
        {
            return this;
        }

        var transitionUnit = new TransitionUnit(pop, push, this.Id);
        transitions.Add(tokenType, transitionUnit);

        return this;
    }
    public State ConfigureOtherwiseTransition(int stateId, int? pop = null, int? push
= null)
    {

```

```

        otherwise = new TransitionUnit(pop, push, stateId);
        return this;
    }
    #endregion

    public int Transit(uint tokenType)
    {
        var transitionUnit = CanTransit(tokenType) ? transitions[tokenType] : otherwise;
    }

    #if DEBUG
        if (transitionUnit is null)
        {
            throw new Exception($"Unconfigured transaction. State: {Id}. " +
                                $"Token: {Crundras.Common.Tables.LexemesTable.GetLexemeName(tokenType)}");
        }
    #endif

    if (transitionUnit.Pop.HasValue)
    {
        if (stack.Peek() != transitionUnit.Pop.Value)
        {
            return otherwise.StateId;
        }
        stack.Pop();
    }

    if (transitionUnit.Push.HasValue)
    {
        stack.Push(transitionUnit.Push.Value);
    }

    return transitionUnit.StateId;
}

private bool CanTransit(uint tokenType)
{
    return transitions.ContainsKey(tokenType);
}

public bool IsError => this.GetType() == typeof(ErrorState);
public bool IsFinal { get; }
public bool TakeToken { get; }
}
}

```

StateMachine.cs

```

using Crundras.Common.Tables;
using System;
using System.Collections.Generic;

namespace SyntaxAnalyzerPDA.PDA
{
    public class StateMachine
    {
        private readonly Dictionary<int, State> states = new Dictionary<int, State>();
        private readonly Stack<int> stack = new Stack<int>();
    }
}

```

```

public State CurrentState { get; private set; }

public StateMachine()
{
    #region EXPRESSION
    states[105] = new State(105, stack, true, false);

    states[103] = new State(103, stack)
        .ConfigureTransition(LexemesTable.GetLexemeId("("), 106, null, ')')
        .ConfigureTransition(LexemesTable.GetLexemeId("+"), 102)
        .ConfigureTransition(LexemesTable.GetLexemeId("-"), 102)
        .ConfigureTransition(LexemesTable.GetLexemeId("identifiant"), 104)
        .ConfigureTransition(LexemesTable.GetLexemeId("int_literal"), 104)
        .ConfigureTransition(LexemesTable.GetLexemeId("float_literal"), 104);

    states[104] = new State(104, stack)
        .ConfigureSelfTransition(LexemesTable.GetLexemeId("("), ')')
        .ConfigureTransition(17, 103)
        .ConfigureTransition(18, 103)
        .ConfigureTransition(19, 103)
        .ConfigureTransition(20, 103)
        .ConfigureTransition(21, 103)
        .ConfigureTransition(22, 103)
        .ConfigureTransition(23, 103)
        .ConfigureTransition(24, 103)
        .ConfigureTransition(25, 103)
        .ConfigureTransition(26, 103)
        .ConfigureTransition(27, 103)
        .ConfigureTransition(28, 103)
        .ConfigureOtherwiseTransition(105);

    states[102] = new State(102, stack)
        .ConfigureTransition(LexemesTable.GetLexemeId("("), 106, null, ')')
        .ConfigureTransition(LexemesTable.GetLexemeId("identifiant"), 104)
        .ConfigureTransition(LexemesTable.GetLexemeId("int_literal"), 104)
        .ConfigureTransition(LexemesTable.GetLexemeId("float_literal"), 104);

    states[106] = new State(106, stack)
        .ConfigureTransition(LexemesTable.GetLexemeId("+"), 102)
        .ConfigureTransition(LexemesTable.GetLexemeId("-"), 102)
        .ConfigureTransition(LexemesTable.GetLexemeId("identifiant"), 101)
        .ConfigureTransition(LexemesTable.GetLexemeId("int_literal"), 101)
        .ConfigureTransition(LexemesTable.GetLexemeId("float_literal"), 101)
        .ConfigureSelfTransition(LexemesTable.GetLexemeId("("), null, ')');

    states[101] = new State(101, stack, false, false)
        .ConfigureTransition(LexemesTable.GetLexemeId("+"), 102)
        .ConfigureTransition(LexemesTable.GetLexemeId("-"), 102)
        .ConfigureTransition(LexemesTable.GetLexemeId("identifiant"), 104)
        .ConfigureTransition(LexemesTable.GetLexemeId("int_literal"), 104)
        .ConfigureTransition(LexemesTable.GetLexemeId("float_literal"), 104)
        .ConfigureTransition(LexemesTable.GetLexemeId("("), 106, null, ')');

    states[100] = new State(100, stack)
        .ConfigureTransition(LexemesTable.GetLexemeId("+"), 101)
        .ConfigureTransition(LexemesTable.GetLexemeId("-"), 101)
        .ConfigureTransition(LexemesTable.GetLexemeId("identifiant"), 101)
        .ConfigureTransition(LexemesTable.GetLexemeId("int_literal"), 101)
        .ConfigureTransition(LexemesTable.GetLexemeId("float_literal"), 101)
        .ConfigureTransition(LexemesTable.GetLexemeId("("), 101);

```

```

#endregion

#region STATEMENT
states[3] = new State(3, stack, true);

// for
states[21] = new State(21, stack)
    .ConfigureTransition(LexemesTable.GetLexemeId(";"), 3);

states[20] = new State(20, stack)
    .ConfigureTransition(LexemesTable.GetLexemeId("rof"), 21);

states[19] = new State(19, stack)
    .ConfigureTransition(LexemesTable.GetLexemeId(")"), 0, null, 20);

states[18] = new State(18, stack)
    .ConfigureTransition(LexemesTable.GetLexemeId("("), 100, null, 19);

states[17] = new State(17, stack)
    .ConfigureTransition(LexemesTable.GetLexemeId("while"), 18);

states[16] = new State(16, stack)
    .ConfigureTransition(LexemesTable.GetLexemeId("by"), 100, null, 17);

states[15] = new State(15, stack)
    .ConfigureTransition(LexemesTable.GetLexemeId("to"), 100, null, 16);

states[14] = new State(14, stack)
    .ConfigureTransition(LexemesTable.GetLexemeId("="), 100, null, 15);

states[13] = new State(13, stack)
    .ConfigureTransition(LexemesTable.GetLexemeId("identifiant"), 14);

// assignment
states[12] = new State(12, stack)
    .ConfigureTransition(LexemesTable.GetLexemeId(";"), 3);
// goto & assignment
states[11] = new State(11, stack)
    .ConfigureTransition(LexemesTable.GetLexemeId("="), 100, null, 12)
    .ConfigureTransition(LexemesTable.GetLexemeId(":"), 3);

// if
states[10] = new State(10, stack)
    .ConfigureTransition(LexemesTable.GetLexemeId(")"), 0, null, 3);

states[9] = new State(9, stack)
    .ConfigureTransition(LexemesTable.GetLexemeId("("), 100, null, 10);

// goto
states[8] = new State(8, stack)
    .ConfigureTransition(LexemesTable.GetLexemeId(";"), 3);

states[7] = new State(7, stack)
    .ConfigureTransition(LexemesTable.GetLexemeId("identifiant"), 8);

// output
states[6] = new State(6, stack)
    .ConfigureTransition(LexemesTable.GetLexemeId(";"), 3);

// input
states[5] = new State(5, stack)

```

```

        .ConfigureTransition(LexemesTable.GetLexemeId(";"), 3);

states[4] = new State(4, stack)
    .ConfigureTransition(LexemesTable.GetLexemeId("identifier"), 5);

// declaration
states[2] = new State(2, stack)
    .ConfigureTransition(LexemesTable.GetLexemeId(";"), 3);

states[1] = new State(1, stack)
    .ConfigureTransition(LexemesTable.GetLexemeId("identifier"), 2);

// root
states[43] = new State(43, stack, false, false)
    .ConfigureTransition(LexemesTable.GetLexemeId("int"), 1)
    .ConfigureTransition(LexemesTable.GetLexemeId("float"), 1)
    .ConfigureTransition(LexemesTable.GetLexemeId("$"), 4)
    .ConfigureTransition(LexemesTable.GetLexemeId("@"), 100, null, 6)
    .ConfigureTransition(LexemesTable.GetLexemeId("goto"), 7)
    .ConfigureTransition(LexemesTable.GetLexemeId("identifier"), 11)
    .ConfigureTransition(LexemesTable.GetLexemeId("if"), 9)
    .ConfigureTransition(LexemesTable.GetLexemeId("for"), 13)
    .ConfigureTransition(LexemesTable.GetLexemeId("{"), 0, null, 42);

states[42] = new State(42, stack)
    .ConfigureTransition(LexemesTable.GetLexemeId("}"), 3)
    .ConfigureOtherwiseTransition(43, null, 42);

states[0] = new State(0, stack)
    .ConfigureOtherwiseTransition(43);

#endregion

CurrentState = states[0];
}

public void NextState(uint tokenType)
{
    int nextStateId;
    while (CurrentState.IsFinal)
    {
        if (stack.Count == 0)
        {
            CurrentState = states[0];
            break;
        }

        nextStateId = stack.Pop();
        if (!states.ContainsKey(nextStateId))
        {
            throw new Exception($"Something forgotten in stack:
\\'{{(char)nextStateId}\\'");
        }

        CurrentState = states[nextStateId];
    }

    nextStateId = CurrentState.Transit(tokenType);

    if (!states.ContainsKey(nextStateId))
    {

```

```

        throw new Exception($"lexeme: {LexemesTable.GetLexemeName(tokenType)}.
state: {CurrentState.Id}");
    }
    CurrentState = states[nextStateId];
}
}
}

```

RPNToken.cs

```

using Crundras.Common;
using Crundras.Common.Tables;

namespace RPNTTranslator
{
    public class RPNToken
    {
        public string Name { get; set; }

        public uint LexemeCode { get; set; }

        public uint? Id { get; set; }

        public RPNToken(uint lexemeCode, string name, uint? id = null)
        {
            Name = name;
            LexemeCode = lexemeCode;
            Id = id;
        }

        public RPNToken(uint lexemeCode, uint? id = null)
        {
            Name = LexemesTable.GetLexemeName(lexemeCode);
            LexemeCode = lexemeCode;
            Id = id;
        }

        public RPNToken(SyntaxTreeNode treeNode)
        {
            Name = LexemesTable.GetLexemeName(treeNode.LexemeCode);
            LexemeCode = treeNode.LexemeCode;
            Id = treeNode.Id;
        }
    }
}

```

RPNTTranslator.cs

```

using Crundras.Common;
using Crundras.Common.Tables;
using System;
using System.Collections.Generic;

namespace RPNTTranslator
{
    public class RPNTTranslator
    {
        private readonly LinkedList<RPNToken> result = new LinkedList<RPNToken>();
        private readonly TablesCollection tables;
    }
}

```

```

private RPNTTranslator(TablesCollection tables)
{
    this.tables = tables;
}

private void ArithmeticExpression(List<SyntaxTreeNode> nodes)
{
    if (nodes.Count == 0) return;

    var stack = new Stack<RPNTToken>();
    var priorityTable = new Dictionary<string, int>
    {
        { "(", 0 },
        { ")", 0 },
        { "<", 1 },
        { "<=", 1 },
        { ">", 1 },
        { ">=", 1 },
        { "==", 1 },
        { "!=", 1 },
        { "+", 2 },
        { "-", 2 },
        { "*", 3 },
        { "/", 3 },
        { "%", 3 },
        { "**", 3 },
        { "NEG", 4 }
    };

    for (int i = 0; i < nodes.Count; i++)
    {
        if (Token.IsIdentifierOrLiteral(nodes[i].LexemeCode))
        {
            result.AddLast(new RPNTToken(nodes[i]));
            continue;
        }

        if (nodes[i].LexemeCode == LexemesTable.GetLexemeId("("))
        {
            stack.Push(new RPNTToken(nodes[i]));
        }
        else if (nodes[i].LexemeCode == LexemesTable.GetLexemeId(")"))
        {
            var token = stack.Pop();
            while (token.Name != "(")
            {
                result.AddLast(token);
                token = stack.Pop();
            }
        }
        else
        {
            var token = new RPNTToken(nodes[i]);

            if ((i == 0) || (nodes[i - 1].LexemeCode != LexemesTable.GetLex-
emeId("(")
&& !Token.IsIdentifierOrLiteral(nodes[i - 1].Lex-
emeCode)))
            {
                if (nodes[i].LexemeCode == LexemesTable.GetLexemeId("-"))

```

```

        {
            token.Name = "NEG";
        }
        else if (nodes[i].LexemeCode == LexemesTable.GetLexemeId("+"))
        {
            continue;
        }
    }
    while ((stack.Count > 0) && priorityTable[token.Name] <= priori-
tyTable[stack.Peek().Name])
    {
        result.AddLast(stack.Pop());
    }
    stack.Push(token);
}

while (stack.Count > 0)
{
    result.AddLast(stack.Pop());
}

private void SetIdentifierType(SyntaxTreeNode identifierNode, uint type)
{
    if (!identifierNode.Id.HasValue)
    {
        throw new Exception("Expected identifier.");
    }

    var identifier = tables.IdentifiersTable[identifierNode.Id.Value];
    identifier.Type = type;
}

private void IfStatement(SyntaxTreeNode node)
{
    var endLabelId = tables.LabelsTable.GetId($"_LB{tables.LabelsTable.Count +
1}");
    var endLabel = tables.LabelsTable[endLabelId];
    var endLabelToken = new RPNToken(LexemesTable.GetLexemeId("label"), endLa-
bel.Name, endLabelId);

    result.AddLast(endLabelToken);

    ArithmeticExpression(node.Children[0].Children);
    result.AddLast(new RPNToken(LexemesTable.GetLexemeId("if")));

    Statement(node.Children[1]);

    endLabel.Position = (uint)result.Count;
    result.AddLast(endLabelToken);
}

private void ForStatement(SyntaxTreeNode node)
{
    var endLabelId = tables.LabelsTable.GetId($"_LB{tables.LabelsTable.Count +
1}");
    var endLabel = tables.LabelsTable[endLabelId];
    var endLabelToken = new RPNToken(LexemesTable.GetLexemeId("label"), endLa-
bel.Name, endLabelId);

```



```

        var startLabelId = tables.LabelsTable.GetId($"_LB{tables.LabelsTable.Count +
1}");
        var startLabelToken = new RPNToken(LexemesTable.GetLexemeId("label"), ta-
bles.LabelsTable[startLabelId].Name, startLabelId);

        // stm 1 (assign)
        Assign(node.Children[0]);

        var iterationVariable = node.Children[0].Children[0];

        // start label
        tables.LabelsTable[startLabelId].Position = (uint)result.Count;
        result.AddLast(startLabelToken);

        // stm 4 (while)
        result.AddLast(endLabelToken);
        ArithmeticExpression(node.Children[3].Children[0].Children);
        var id = tables.IntLiteralsTable.GetId(0);
        result.AddLast(new RPNToken(LexemesTable.GetLexemeId("int_literal"),
"int_literal", id));
        result.AddLast(new RPNToken(LexemesTable.GetLexemeId("!="), "!="));
        result.AddLast(new RPNToken(LexemesTable.GetLexemeId("if")));

        // stm 2 (to)
        result.AddLast(endLabelToken); // goto end
        ArithmeticExpression(node.Children[1].Children[0].Children);
        result.AddLast(new RPNToken(iterationVariable));
        result.AddLast(new RPNToken(LexemesTable.GetLexemeId(">")));
        id = tables.IntLiteralsTable.GetId(0);
        result.AddLast(new RPNToken(LexemesTable.GetLexemeId("int_literal"),
"int_literal", id));
        result.AddLast(new RPNToken(LexemesTable.GetLexemeId("!="), "!="));
        result.AddLast(new RPNToken(LexemesTable.GetLexemeId("if")));

        // body
        Statement(node.Children[4]);

        // stm 3 (by - iteration)
        ArithmeticExpression(node.Children[3].Children[0].Children);
        result.AddLast(new RPNToken(iterationVariable));
        result.AddLast(new RPNToken(LexemesTable.GetLexemeId("+")));
        result.AddLast(new RPNToken(iterationVariable));
        result.AddLast(new RPNToken(LexemesTable.GetLexemeId("=")));

        // goto start
        result.AddLast(startLabelToken);
        result.AddLast(new RPNToken(LexemesTable.GetLexemeId("goto")));
        // end
        endLabel.Position = (uint)result.Count;
        result.AddLast(endLabelToken);
    }

    private void Assign(SyntaxTreeNode node)
    {
        ArithmeticExpression(node.Children[1].Children);
        result.AddLast(new RPNToken(node.Children[0]));
        result.AddLast(new RPNToken(node));
    }

    private void Statement(SyntaxTreeNode node)

```

```

{
    switch (node.LexemeCode)
    {
        // '='
        case 16:
            Assign(node);
            break;
        // 'label'
        case 2:
            tables.LabelsTable[node.Id.Value].Position = (uint)result.Count;
            result.AddLast(new RPNToken(node));
            break;
        // '$'
        case 14:
            result.AddLast(new RPNToken(node.Children[0]));
            result.AddLast(new RPNToken(LexemesTable.GetLexemeId("$")));
            break;
        // '@'
        case 15:
            ArithmeticExpression(node.Children[0].Children);
            result.AddLast(new RPNToken(node));
            break;
        // '{'
        case 31:
            Statements(node);
            break;
        // 'if'
        case 7:
            IfStatement(node);
            break;
        // 'for'
        case 8:
            ForStatement(node);
            break;
        // 'int'
        case 5:
            SetIdentifierType(node.Children[0], LexemesTable.GetLex-
emeId("int_literal"));
            break;
        // 'float'
        case 6:
            SetIdentifierType(node.Children[0], LexemesTable.GetLex-
emeId("float_literal"));
            break;
        // 'goto'
        case 13:
            result.AddLast(new RPNToken(node.Children[0]));
            result.AddLast(new RPNToken(node));
            break;
    }
}

private void Statements(SyntaxTreeNode node)
{
    foreach (var child in node.Children)
    {
        Statement(child);
    }
}

```

```

        public static LinkedList<RPNToken> Analyze(TablesCollection tables, Syntax-
TreeNode root)
        {
            var syntaxAnalyzerRpn = new RPNTranslator(tables);
            syntaxAnalyzerRpn.Statements(root);
            return syntaxAnalyzerRpn.result;
        }
    }
}

```

RPNIInterpreter.cs

```

using Crundras.Common;
using Crundras.Common.Tables;
using RPNTranslator;
using System;
using System.Collections.Generic;
using System.Collections.Immutable;

namespace RPNIInterpreter
{
    public class RPNIInterpreter
    {
        public void Interpret(TablesCollection tables, LinkedList<RPNToken> tokens)
        {
            var stack = new Stack<RPNToken>();
            var tokensArray = tokens.ToImmutableArray();

            for (int i = 0; i < tokensArray.Length; i++)
            {
                RPNToken token = tokensArray[i];
                if (Token.IsIdentifierOrLiteral(token.LexemeCode)
                    || Token.IsLabel(token.LexemeCode))
                {
                    stack.Push(token);
                }
                else
                {
                    if (token.LexemeCode == LexemesTable.GetLexemeId("@"))
                    {
                        Console.WriteLine("Output: ");
                        var rpnToken = stack.Pop();
                        if (!rpnToken.Id.HasValue)
                        {
                            throw new Exception("");
                        }

                        if (Token.IsIdentifier(rpnToken.LexemeCode))
                        {
                            Console.WriteLine(tables.IdentifiersTable[rpnToken.Id.Value].Value);
                        }
                        else if (Token.IsIntLiteral(rpnToken.LexemeCode))
                        {
                            Console.WriteLine(tables.IntLiteralsTable[rpnToken.Id.Value]);
                        }
                        else if (Token.IsFloatLiteral(rpnToken.LexemeCode))
                        {
                            Console.WriteLine(tables.FloatLiteralsTable[rpnToken.Id.Value]);
                        }
                    }
                }
            }
        }
    }
}

```

```

    }
}
else if (token.LexemeCode == LexemesTable.GetLexemeId("$"))
{
    Console.WriteLine("Input: ");
    var identToken = stack.Pop();
    if (!identToken.Id.HasValue || !Token.IsIdentifier(identTo-
ken.LexemeCode))
    {
        throw new Exception("");
    }

    var inputLine = Console.ReadLine();

    if (Token.IsFloatLiteral.tables.IdentifiersTable[identTo-
ken.Id.Value].Type)
        && !float.TryParse(inputLine, out _)
    {
        throw new Exception("");
    }
    if (Token.IsIntLiteral.tables.IdentifiersTable[identTo-
ken.Id.Value].Type)
        && !int.TryParse(inputLine, out _)
    {
        throw new Exception("");
    }
    tables.IdentifiersTable[identToken.Id.Value].Value = inputLine;
    stack.Push(identToken);
}
else if (token.LexemeCode == LexemesTable.GetLexemeId("goto"))
{
    var labelToken = stack.Pop();
    if (!labelToken.Id.HasValue || !Token.IsLabel(labelToken.Lex-
emeCode))
    {
        throw new Exception($"Expected label but got '{LexemesTa-
ble.GetLexemeName(labelToken.LexemeCode)}'");
    }

    i = (int)tables.LabelsTable[labelToken.Id.Value].Position;
}
else if (token.LexemeCode == LexemesTable.GetLexemeId("if"))
{
    var valueToken = stack.Pop();
    if (!valueToken.Id.HasValue ||
        (!Token.IsLiteral(valueToken.LexemeCode) && !Token.IsIdenti-
fier(valueToken.LexemeCode)))
    {
        throw new Exception($"Expected literal or identifier but got
'{LexemesTable.GetLexemeName(valueToken.LexemeCode)}'");
    }

    var labelToken = stack.Pop();
    if (!labelToken.Id.HasValue || !Token.IsLabel(labelToken.Lex-
emeCode))
    {
        throw new Exception($"Expected label but got '{LexemesTa-
ble.GetLexemeName(labelToken.LexemeCode)}'");
    }

    bool value = false;

```

```

        if (Token.IsIntLiteral(valueToken.LexemeCode))
        {
            value = tables.IntLiteralsTable[valueToken.Id.Value] == 0;
        }
        else if (Token.IsFloatLiteral(valueToken.LexemeCode))
        {
            value = Math.Abs(tables.FloatLiteralsTable[valueToken.Id.Value]) < double.Epsilon;
        }
        else if (Token.IsIdentifier(valueToken.LexemeCode))
        {
            value = Math.Abs(double.Parse(tables.IdentifiersTable[valueToken.Id.Value].Value)) < double.Epsilon;
        }
        else
        {
            throw new Exception("???");
        }

        if (value)
        {
            i = (int)tables.LabelsTable[labelToken.Id.Value].Position;
        }
    }
    else if (token.Name == "NEG")
    {
        var rpnToken = stack.Pop();
        if (!rpnToken.Id.HasValue)
        {
            throw new Exception("");
        }

        if (Token.IsIdentifier(rpnToken.LexemeCode))
        {
            var value = tables.IdentifiersTable[rpnToken.Id.Value].Value;
            if (value.StartsWith('-'))
            {
                value.TrimStart('-');
            }
            else value = '-' + value;

            tables.IdentifiersTable[rpnToken.Id.Value].Value = value;
        }
        else if (Token.IsIntLiteral(rpnToken.LexemeCode))
        {
            var value = -tables.IntLiteralsTable[rpnToken.Id.Value];
            rpnToken.Id = tables.IntLiteralsTable.GetId(value);
        }
        else if (Token.IsFloatLiteral(rpnToken.LexemeCode))
        {
            var value = -tables.FloatLiteralsTable[rpnToken.Id.Value];
            rpnToken.Id = tables.FloatLiteralsTable.GetId(value);
        }

        stack.Push(rpnToken);
    }
    else if (token.Name == "=")
    {
        var identToken = stack.Pop();
        if (!Token.IsIdentifier(identToken.LexemeCode))
        {

```

```

        throw new Exception("");
    }
    if (!identToken.Id.HasValue)
    {
        throw new Exception("");
    }

    var type = tables.IdentifiersTable[identToken.Id.Value].Type;
    if (type == 0)
    {
        throw new Exception($"Undefined variable: '{tables.IdentifiersTable[identToken.Id.Value].Name}'");
    }

    var valueToken = stack.Pop();

    string value;
    if (Token.IsIntLiteral(valueToken.LexemeCode))
    {
        value = tables.IntLiteralsTable[valueToken.Id.Value].ToString();
    }
    else if (Token.IsFloatLiteral(valueToken.LexemeCode))
    {
        value = tables.FloatLiteralsTable[valueToken.Id.Value].ToString();
    }
    else if (Token.IsIdentifier(valueToken.LexemeCode))
    {
        value = tables.IdentifiersTable[valueToken.Id.Value].Value;
    }
    else
    {
        throw new Exception("");
    }

    tables.IdentifiersTable[identToken.Id.Value].Value = value;
    stack.Push(identToken);
}
else if (stack.Count >= 2)
{
    var a = stack.Pop();
    var b = stack.Pop();

    if (!a.Id.HasValue || !b.Id.HasValue)
    {
        throw new Exception("");
    }

    double firstValue = 0;
    double secondValue = 0;
    uint firstType = 0;
    uint secondType = 0;

    if (Token.IsIdentifier(a.LexemeCode))
    {
        var value = tables.IdentifiersTable[a.Id.Value].Value;
        if (string.IsNullOrEmpty(value))
        {
            throw new Exception($"Undefined variable: '{tables.IdentifiersTable[a.Id.Value].Name}'");
        }
    }

```

```

    }
    firstValue = double.Parse(value);
    firstType = tables.IdentifiersTable[a.Id.Value].Type;
}
else if (Token.IsIntLiteral(a.LexemeCode))
{
    firstValue = tables.IntLiteralsTable[a.Id.Value];
    firstType = a.LexemeCode;
}
else if (Token.IsFloatLiteral(a.LexemeCode))
{
    firstValue = tables.FloatLiteralsTable[a.Id.Value];
    firstType = a.LexemeCode;
}

if (Token.IsIdentifier(b.LexemeCode))
{
    var value = tables.IdentifiersTable[b.Id.Value].Value;
    if (string.IsNullOrEmpty(value))
    {
        throw new Exception($"Undefined variable: '{tables.IdentifiersTable[b.Id.Value].Name}'");
    }
    secondValue = double.Parse(value);
    secondType = tables.IdentifiersTable[b.Id.Value].Type;
}
else if (Token.IsIntLiteral(b.LexemeCode))
{
    secondValue = tables.IntLiteralsTable[b.Id.Value];
    secondType = b.LexemeCode;
}
else if (Token.IsFloatLiteral(b.LexemeCode))
{
    secondValue = tables.FloatLiteralsTable[b.Id.Value];
    secondType = b.LexemeCode;
}

if (firstType == 0 || secondType == 0)
{
    throw new Exception("");
}

double resultValue = 0;
switch (token.LexemeCode)
{
    case 17: // +
        resultValue = secondValue + firstValue;
        break;
    case 18: // -
        resultValue = secondValue - firstValue;
        break;
    case 19: // *
        resultValue = secondValue * firstValue;
        break;
    case 20: // **
        resultValue = Math.Pow(secondValue, firstValue);
        break;
    case 21: // /
        resultValue = secondValue / firstValue;
        break;
    case 22: // %

```



```

        return;
    }

    if (!File.Exists(args[0]))
    {
        Console.WriteLine($"File \"{args[0]}\" doesn't seem to exist.");
        return;
    }

    try
    {
        var tables = LexicalAnalyzer.AnalyzeFile(args[0]);

        var syntaxTree = SyntaxAnalyzer.Analyze(tables);
        var rpntokens = RPNTTranslator.RPNTTranslator.Analyze(tables, syntaxTree);

        Console.WriteLine("Interpreting:");
        new RPNInterpreter().Interpret(tables, rpntokens);
    }
    catch (Exception e)
    {
        Console.WriteLine(e.Message);
    }

    Console.ReadKey();
}
}
}

```