**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 28 аркушах, вона містить 3 додатки та перелік посилань на використані джерела з 4 найменувань. У роботі наведено 13 рисунків та 3 таблиці.

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

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 4](#_Toc41662153)

[1 Завдання 5](#_Toc41662154)

[2 Специфікація мови «Crundras» 6](#_Toc41662155)

[2.1 Обробка 6](#_Toc41662156)

[2.2 Нотація 6](#_Toc41662157)

[2.3 Алфавіт 7](#_Toc41662158)

[2.4 Лексика 7](#_Toc41662159)

[2.4.1 Спеціальні символи 7](#_Toc41662160)

[2.4.2 Ідентифікатори 8](#_Toc41662161)

[2.4.3 Літерали 8](#_Toc41662162)

[2.4.4 Ключові слова 9](#_Toc41662163)

[2.4.5 Токени 9](#_Toc41662164)

[2.5 Тип 9](#_Toc41662165)

[2.6 Синтаксис 10](#_Toc41662166)

[2.6.1 Вирази 10](#_Toc41662167)

[2.6.2 Оператори 11](#_Toc41662168)

[2.7 Програма 11](#_Toc41662169)

[2.8 Оголошення 11](#_Toc41662170)

[2.9 Iнструкцiї 12](#_Toc41662171)

[2.9.1 Оператор (iнструкцiя) присвоювання 12](#_Toc41662172)

[2.9.2 Iнструкцiя введення 13](#_Toc41662173)

[2.9.3 Iнструкцiя виведення 13](#_Toc41662174)

[2.9.4 Оператор вибору 13](#_Toc41662175)

[2.9.5 Оператор циклу 14](#_Toc41662176)

[3 Структура транслятора 15](#_Toc41662177)

[4 Програмна реалізація 17](#_Toc41662178)

[4.1 Лексичний аналізатор 17](#_Toc41662179)

[4.2 Синтаксичний аналізатор 18](#_Toc41662180)

[4.3 Проміжна форма подання програми 20](#_Toc41662181)

[5 Інструкція користувача по роботі з транслятором 21](#_Toc41662182)

[6 Тестування 22](#_Toc41662183)

[ВИСНОВКИ 27](#_Toc41662184)

[СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 28](#_Toc41662185)

# 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.

# 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;

1. Дотримання структури умовного оператора:

if (<expression>) <operations block>;

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

# Специфікація мови «Crundras»

## 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 здійснюються окремим проходом.

## Нотація

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

|  |  |
| --- | --- |
| Метасимвол | Значення |
| = | визначається як |
| | | альтернатива |
| [ x ] | 0 або 1 екземпляр x |
| { x } | 0 або бiльше екземплярiв x |
| ( x | y ) | групування: «будь-який з»  (x або y) |
| Fgh | нетермiнал |
| Ghj | термiнал |
| '1' | термiнал |
| "1" | термiнал |

Табл. 1: Прийнята нотацiя РБНФ

## Алфавіт

Программа меже мiстити текст з використанням таких символiв (character) — лiтер, цифр, спецiальних знакiв та ознаки кiнця 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'.

## Лексика

Лексичний аналіз виконується окремим проходом і не несе вспоміжний функціонал синтаксичного чи семантичного аналізу. Лексичний аналізатор розбиває вхідний текст на лексеми. У програмі мовою Crundras можуть використовуватись лексичнi елементи, що класифiкуються як спецiальнi символи, iдентифiкатори, беззнаковi цiлi константи, беззнаковi дiйснi константи та ключовi слова.

### Спеціальні символи

Синтаксис

1. SpecialSymbols = ArithmeticOperators | RelationalOperators |

BracketsOperators | Punctuation.

ArithmeticOperator = '+' | '-' | '\*' | "\*\*" | '/' | '%'.

RelationalOperator = '<' | '>' | "<=" | ">=" | "==" | "!=".

BracketsOperators = '(' | ')' | '{'| '}'.

Punctuation = ':' | ';' | '.'.

InputOutpuOperators = '@' | '$'.

Опис

2. До спецiальних символiв належать арифметичнi оператори, оператори вiдношень, оператор присвоювання та знаки пунктуацiї.

Обмеження

3. Набiр токенiв див. додаток табл. 2

### Ідентифікатори

Синтаксис

1. Ident = Letter {Letter | Digit}.

Опис

1. Першим символом iдентифiкатора може бути тiльки лiтера, наступнi символи, якщо вони є, можуть бути цифрами або лiтерами. Довжина iдентифiкатора не обмежена.

Обмеження

1. Жоден iдентифiкатор не може збiгатись iз ключовим (вбудованим, зарезервованим) словом.

Семантика

1. Елемент, який у фазi лексичного аналiзу може бути визначений як iдентифiкатор, або як ключове слово, вважається ключовим словом.

Приклади

1. a, x1, time24

### Літерали

Синтаксис

1. Literal = IntegerLiteral | FloatingLiteral.

FloatingLiteral = IntegerLiteral '.' [IntegerLiteral].

IntegerLiteral = Digit {Digit}.

Обмеження

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

Семантика

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

Приклади

1. 12, 234, 1.54, 34.567, 23.

### Ключові слова

Синтаксис

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

### Токени

З потоку символів вхiдної програми на етапi лексичного аналiзу виокремлюються послідовності символів з певним сукупним значенням, — токени. Список токенів див. табл. 2.

## Тип

Мова Crundras обробляє значення двох типів: int та float.

1. Цілий тип int може бути представлений оголошеною змінною типу int, або константою IntegerLiteral. Дiапазон значень залежить вiд реалiзацiї.
2. Дiйсний тип float може бути представлений оголошеною змiнною типу float або константою FloatingLiteral.

## Синтаксис

### Вирази

Синтаксис

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

Опис

1. Вираз - це послiдовнiсть операторiв i операндiв, що визначає порядок обчислення значення.
2. Значення, обчислене за арифметичним виразом, має тип float або int.
3. Окрім піднесення до степені всі бінарні оператори у виразах є лiвоасоцiативними.
4. Піднесення до степені є правоасоціативним.
5. Послідовність двох або більше операторів з однаковим пріоритетом лівоасоцiативна.
6. Пріоритети операцій:

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

Табл. 3: Таблиця пріоритетів операцій

Обмеження

1. Використання змiнної, з не визначеним на момент обчислення виразу значенням, викликає помилку.

Семантика

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

Приклади

1. x, 12, (a + 234), 32/(b + 786), -b, f1 + g,

(a\*x + b/z) >= (k \*\* t)

### Оператори

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

## Програма

Програма складається з набору виразів. Програма не може мiстити неоголошену змiнну.

\_Program = {Statement}.

## Оголошення

Оголошення (декларацiї) специфiкує iнтерпретацiю та атрибути набору iдентифiкаторiв.

Синтаксис

* 1. DeclarationStatement = TypeSpecifier Identifier ';'.

TypeSpecifier = "int" | "float".

Обмеження

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

Семантика

* 1. Декларація визначає інтерпретацію та атрибути ідентифікатору. Визначення ідентифікатора - це (єдине) оголошення ідентифікатора.

Приклад

* 1. int a; double h;

## Iнструкцiї

Iнструкцiї (Statements) визначають алгоритмiчнi дiї, якi мають бути виконанi. За винятком зазначених далi випадкiв, iнструкцiї виконуються послiдовно.

### Оператор (iнструкцiя) присвоювання

Синтаксис

1. AssignmentStatement = AssignmentExpression ';'.

AssigmentExpression = Identifier '=' Expression.

Обмеження

1. Якщо тип змiнної з iдентифiкатором Identifier не вiдповiдає типу виразу праворуч оператора =, то вираз приводиться до типу змінної.

Семантика

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

Приклад

1. foo = 35;

### Iнструкцiя введення

Синтаксис

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

Приклад

1. $d;

### Iнструкцiя виведення

Синтаксис

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

Приклад

1. @42;

### Оператор вибору

Синтаксис

1. SelectionStatement = "if" '('Expression')' Statement.

Обмеження

1. Керуючий вираз повинен бути скалярним.

Семантика

1. Якщо Expression не дорівнює 0, то виконується Statement.

### Оператор циклу

Синтаксис

1. IterationStatement = "for" AssigmentExpression "to" Expression "by"

Expression "while" '(' Expression ')' Statement "rof" ';'.

Обмеження

1. Керуючий вираз повинен бути скалярним.

Семантика

1. Інструкція:

for <assignment> to <expression-1> by <expression-2> while(<expression-3>)

<statement>

rof;

* 1. <assignment> виконується до будь-яких інструкцій тілу цикла.
  2. Кожного кроку вирази виконуються у такій послідовності:

<expression-3>

<expression-2>

<expression-1>

<statement>

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

Приклад

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

# 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

# Програмна реалізація

## 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.

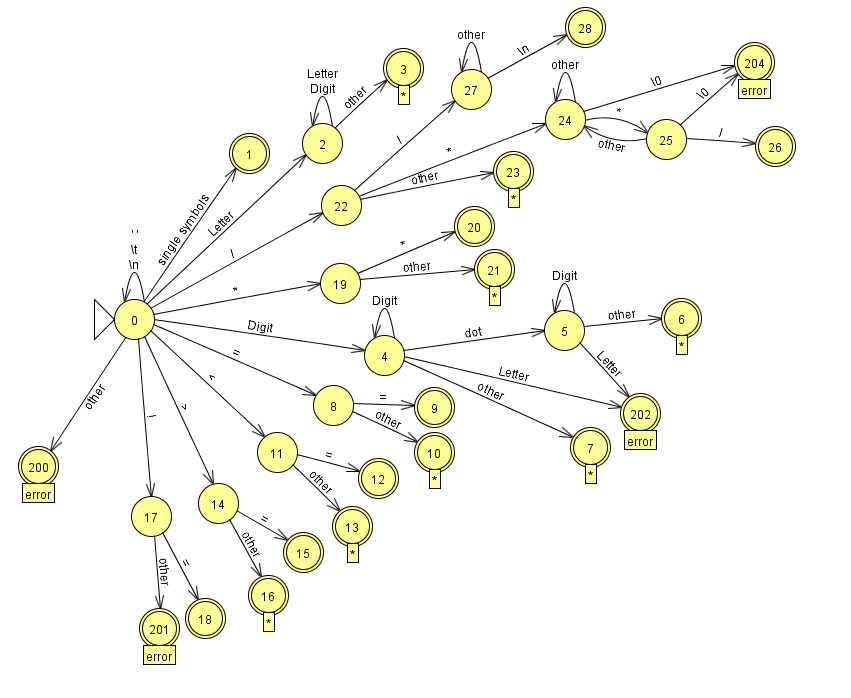


Рис. 4: Діаграма станів лексичного аналізатора

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

## Синтаксичний аналізатор

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, шо 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.

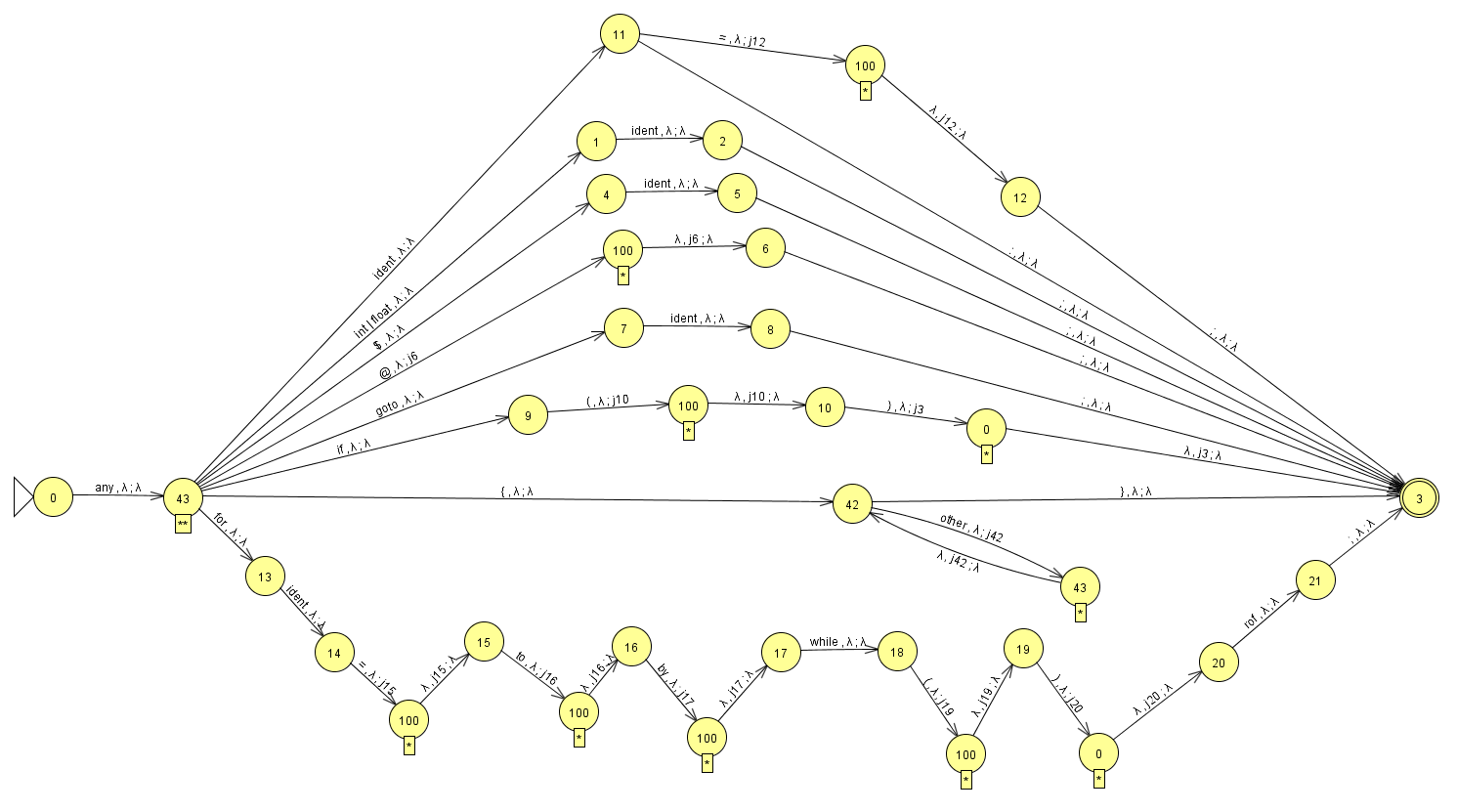


Рис. 5: PDA for statement

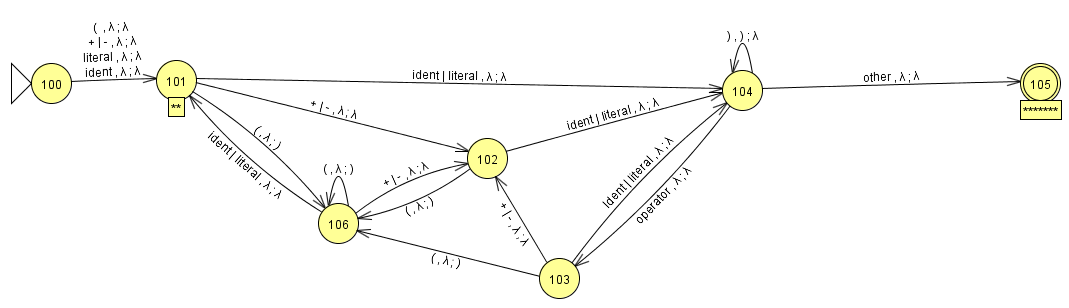


Рис. 6: PDA for expression

## Проміжна форма подання програми

Загалом працювати напряму 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 виробах 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 виразив:

var priorityTable = new Dictionary<string, int> {

    { "(", 0 },

    { ")", 0 },

    { "<", 1 },

    { "<=", 1 },

    { ">", 1 },

    { ">=", 1 },

    { "==", 1 },

    { "!=", 1 },

    { "+", 2 },

    { "-", 2 },

    { "\*", 3 },

    { "/", 3 },

    { "%", 3 },

    { "\*\*", 3 },

    { "NEG", 4 }

};

# 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.

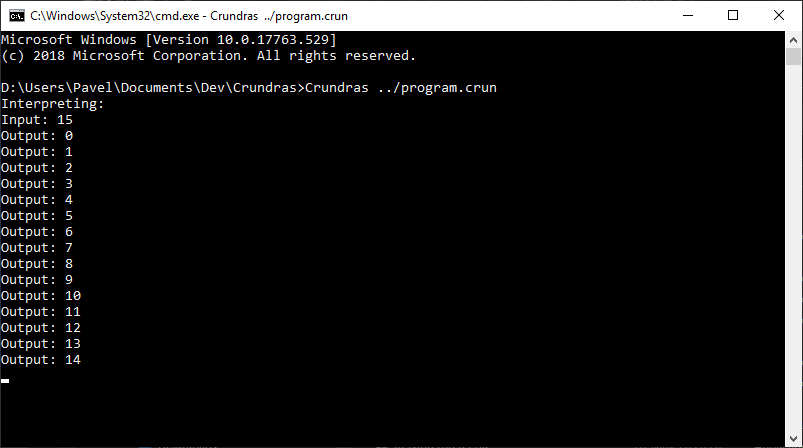


Рис. 9: Приклад виклику транслятора з командного рядку

# Тестування

1. Приклад 1

Код програми мовою Crundras:

int a;

float d;

a = 46 + 95;

if(a) @45;

$d;

for a=75-8 to 140/2 by 2\*\*0 while(d<22+2) {

if(d == 23)

{

@42;

@44;

}

@a;

} rof;

for a = 0 to 15/3 by 0.5 while(d<=23)

@d + 1.5;

rof;

if(d == 23)

{

goto label;

@34;

label:

@42;

}

$a;

@a;

Зворотній польський запис – рис. 8.

Результат виконання – рис. 9, 10.

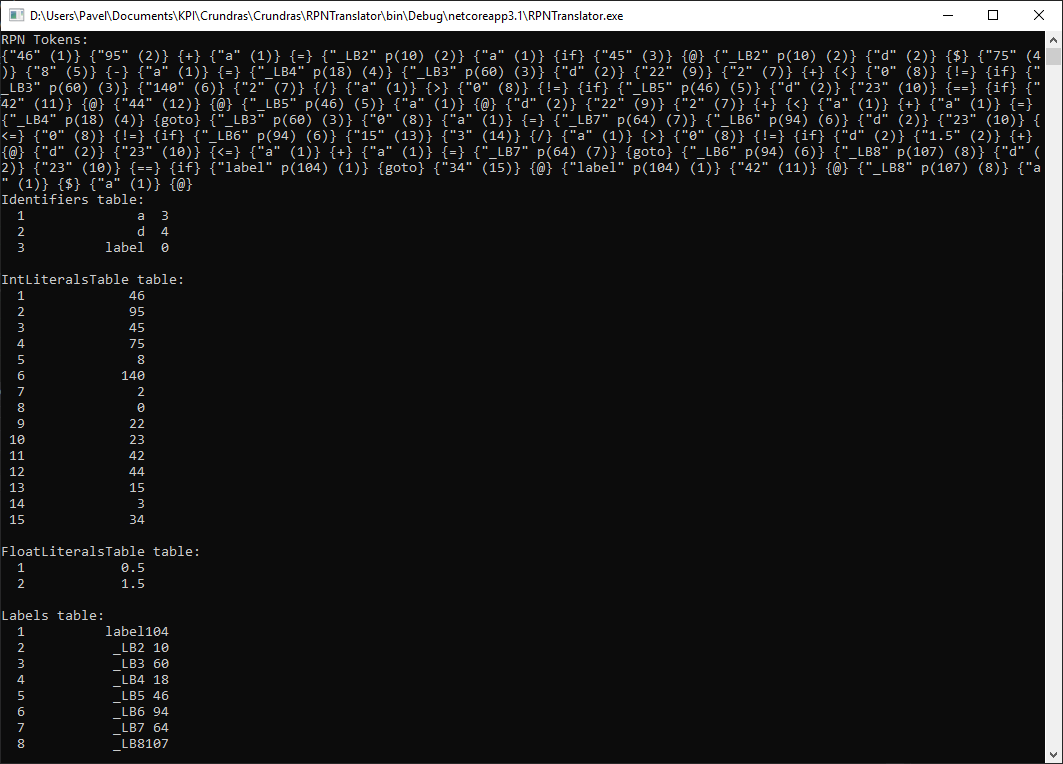


Рис. 8: ПОЛІЗ програми з прикладу 1

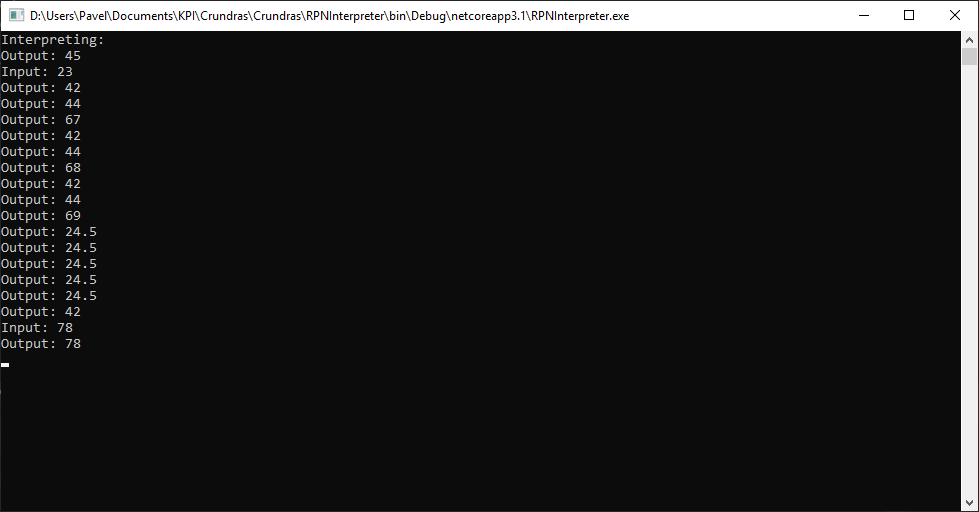


Рис. 9: Результат інтерпретації прикладу 1

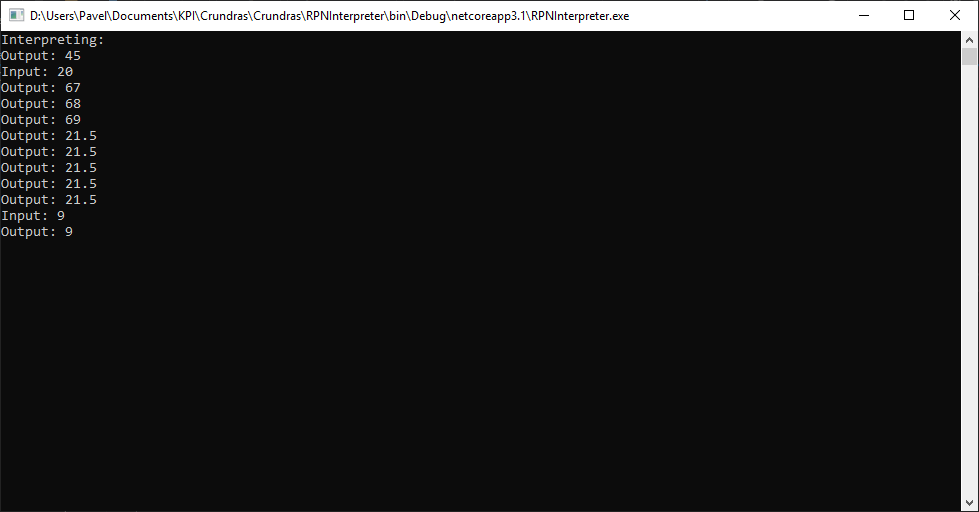


Рис. 10: Результат інтерпретації прикладу 1

1. Приклад 2

Код програми мовою Crundras:

int a;

float (ノ｀Д´)ノ彡┻━┻;

(ノ｀Д´)ノ彡┻━┻ = 2;

@(ノ｀Д´)ノ彡┻━┻;

Результат виконання – рис. 11.

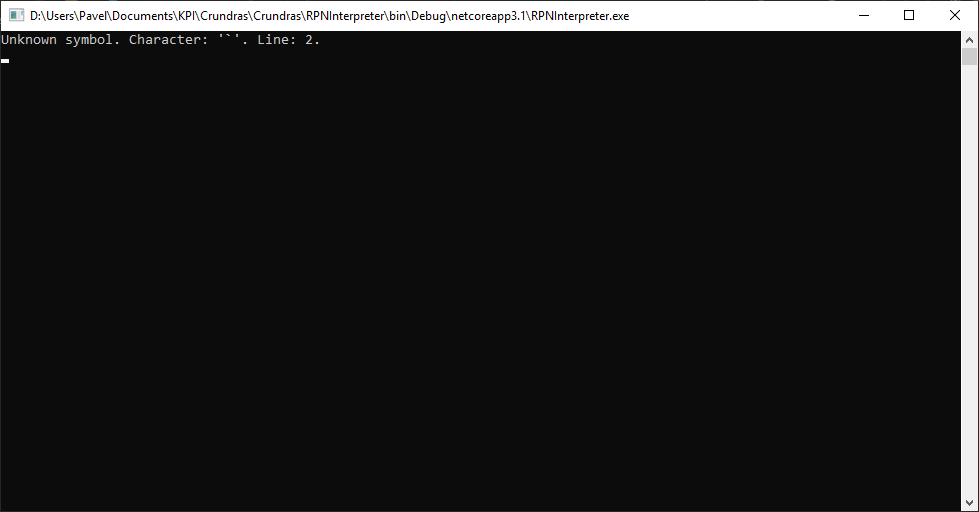


Рис. 11: Результат інтерпретації прикладу 2

1. Приклад 3

Код програми мовою Crundras:

float variable;

variable = 1\*2+-(42%5 – 7 \* 9 \*\* 0.5) \*\* 0;

@variable;

Зворотній польський запис – рис. 12.

Результат виконання – рис. 13.

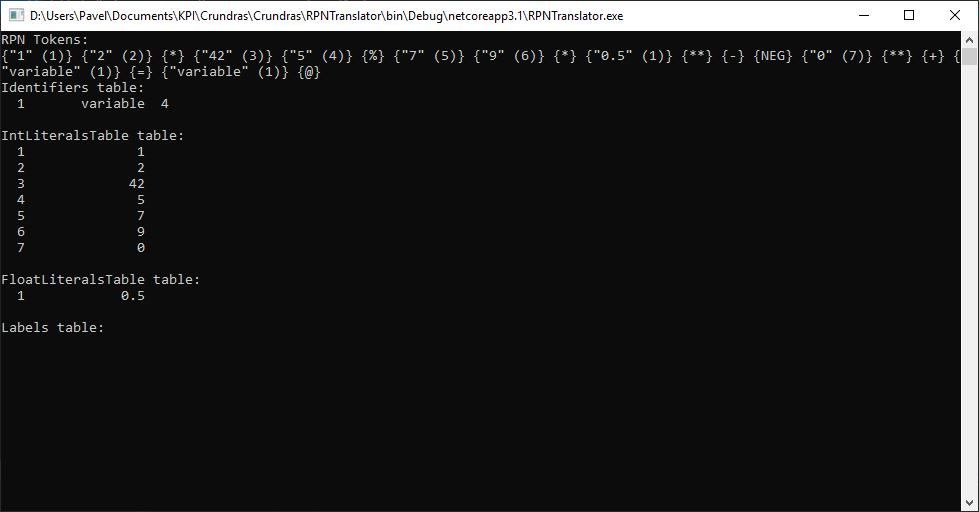


Рис. 12: ПОЛІЗ програми з прикладу 3

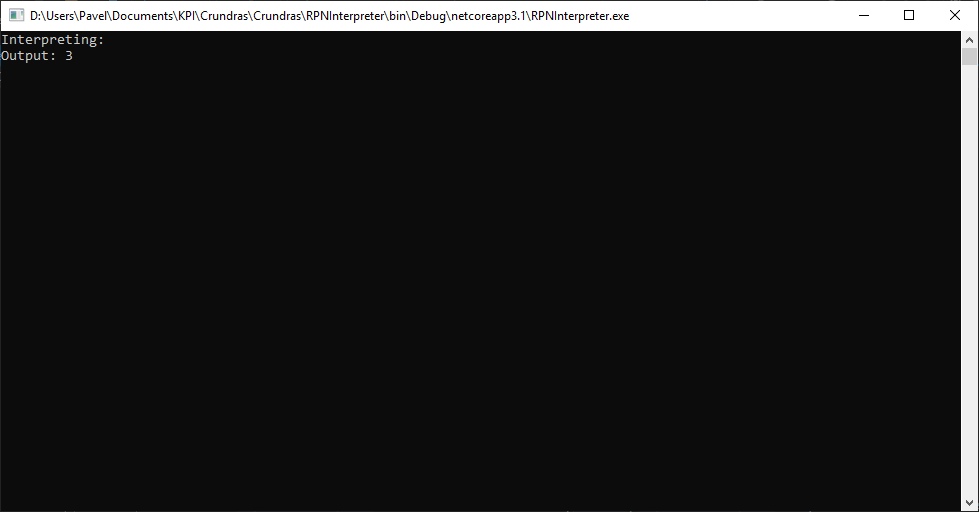


Рис. 13: Результат інтерпретації прикладу 3

# 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 Джефри 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 | SelectionStatement | IterationStatement | DeclarationStatement | AssignmentStatement | JumpStatement | LabelStatement.

SelectionStatement = "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 | iдентифiкатор |
| 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 | термiнал int |
| 6 | float | float | термiнал float |
| 7 | if | if | термiнал if |
| 8 | for | for | термiнал for |
| 9 | to | to | термiнал to |
| 10 | by | by | термiнал by |
| 11 | while | while | термiнал while |
| 12 | rof | rof | термiнал rof |
| 13 | goto | goto | термiнал goto |
| 14 | $ | $ | термiнал $ |
| 15 | @ | @ | термiнал @ |
| 16 | = | = | термiнал = |
| 17 | + | + | термiнал + |
| 18 | - | - | термiнал - |
| 19 | \* | \* | термiнал \* |
| 20 | \*\* | \*\* | термiнал \*\* |
| 21 | / | / | термiнал / |
| 22 | % | % | термiнал % |
| 23 | < | < | термiнал < |
| 24 | > | > | термiнал > |
| 25 | <= | <= | термiнал <= |
| 26 | == | == | термiнал = |
| 27 | >= | >= | термiнал >= |
| 28 | != | != | термiнал != |
| 29 | ( | ( | термiнал ( |
| 30 | ) | ) | термiнал ) |
| 31 | { | { | термiнал { |
| 32 | } | } | термiнал } |
| 33 | ; | ; | термiнал ; |
| 34 | : | : | термiнал : |
| 35 | ‘ ‘, \t | white\_space | пробільний символ |
| 36 | \n, \r\n, \r | eol | end of line |
| 37 | \u0000 | eof | end of file |

Табл. 2: Таблиця токенів мови Crundras

ДОДАТОК

Лістинг програмного продукту

**УКР.НТУУ”КПІ”\_ТЕФ\_АПЕПС.ТР72262**

**Листів 33**

**Київ – 2020**

**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 IntLiteralsTable : 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 FloatLiteralsTable : 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("identifier"), 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("identifier"), 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("identifier"), 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("identifier"), 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("identifier"), 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("identifier"), 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("identifier"), 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 RPNTranslator

{

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;

}

}

}

RPNTranslator.cs

using Crundras.Common;

using Crundras.Common.Tables;

using System;

using System.Collections.Generic;

namespace RPNTranslator

{

public class RPNTranslator

{

private readonly LinkedList<RPNToken> result = new LinkedList<RPNToken>();

private readonly TablesCollection tables;

private RPNTranslator(TablesCollection tables)

{

this.tables = tables;

}

private void ArithmeticExpression(List<SyntaxTreeNode> nodes)

{

if (nodes.Count == 0) return;

var stack = new Stack<RPNToken>();

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 RPNToken(nodes[i]));

continue;

}

if (nodes[i].LexemeCode == LexemesTable.GetLexemeId("("))

{

stack.Push(new RPNToken(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 RPNToken(nodes[i]);

if ((i == 0) || (nodes[i - 1].LexemeCode != LexemesTable.GetLexemeId(")")

&& !Token.IsIdentifierOrLiteral(nodes[i - 1].LexemeCode)))

{

if (nodes[i].LexemeCode == LexemesTable.GetLexemeId("-"))

{

token.Name = "NEG";

}

else if (nodes[i].LexemeCode == LexemesTable.GetLexemeId("+"))

{

continue;

}

}

while ((stack.Count > 0) && priorityTable[token.Name] <= priorityTable[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"), endLabel.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"), endLabel.Name, endLabelId);

var startLabelId = tables.LabelsTable.GetId($"\_LB{tables.LabelsTable.Count + 1}");

var startLabelToken = new RPNToken(LexemesTable.GetLexemeId("label"), tables.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.GetLexemeId("int\_literal"));

break;

// 'float'

case 6:

SetIdentifierType(node.Children[0], LexemesTable.GetLexemeId("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, SyntaxTreeNode root)

{

var syntaxAnalyzerRpn = new RPNTranslator(tables);

syntaxAnalyzerRpn.Statements(root);

return syntaxAnalyzerRpn.result;

}

}

}

RPNInterpretator.cs

using Crundras.Common;

using Crundras.Common.Tables;

using RPNTranslator;

using System;

using System.Collections.Generic;

using System.Collections.Immutable;

namespace RPNInterpreter

{

public class RPNInterpreter

{

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.Write("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.Write("Input: ");

var identToken = stack.Pop();

if (!identToken.Id.HasValue || !Token.IsIdentifier(identToken.LexemeCode))

{

throw new Exception("");

}

var inputLine = Console.ReadLine();

if (Token.IsFloatLiteral(tables.IdentifiersTable[identToken.Id.Value].Type)

&& !float.TryParse(inputLine, out \_))

{

throw new Exception("");

}

if (Token.IsIntLiteral(tables.IdentifiersTable[identToken.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.LexemeCode))

{

throw new Exception($"Expected label but got '{LexemesTable.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.IsIdentifier(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.LexemeCode))

{

throw new Exception($"Expected label but got '{LexemesTable.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.IsNullOrWhiteSpace(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.IsNullOrWhiteSpace(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: // %

resultValue = secondValue % firstValue;

break;

case 23: // <

resultValue = secondValue < firstValue ? 1 : 0;

break;

case 24: // >

resultValue = secondValue > firstValue ? 1 : 0;

break;

case 25: // <=

resultValue = secondValue <= firstValue ? 1 : 0;

break;

case 26: // ==

resultValue = Math.Abs(firstValue - secondValue) < double.Epsilon ? 1 : 0;

break;

case 27: // >=

resultValue = secondValue >= firstValue ? 1 : 0;

break;

case 28: // !=

resultValue = Math.Abs(firstValue - secondValue) > double.Epsilon ? 1 : 0;

break;

}

uint resultType = Math.Max(firstType, secondType);

uint id;

if (Token.IsIntLiteral(resultType))

{

id = tables.IntLiteralsTable.GetId((int)resultValue);

}

else

{

id = tables.FloatLiteralsTable.GetId((float)resultValue);

}

stack.Push(new RPNToken(resultType, id));

}

}

}

}

}

}

Program.cs

using Crundras.LexicalAnalyzer;

using SyntaxAnalyzerPDA;

using System;

using System.IO;

namespace RPNInterpreter

{

class Program

{

static void Main(string[] args)

{

if (args.Length != 1)

{

Console.WriteLine("Crundras.exe %filename%");

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 = RPNTranslator.RPNTranslator.Analyze(tables, syntaxTree);

Console.WriteLine("Interpreting:");

new RPNInterpreter().Interpret(tables, rpnTokens);

}

catch (Exception e)

{

Console.WriteLine(e.Message);

}

Console.ReadKey();

}

}

}