МИНОБРНАУКИ РОССИИ

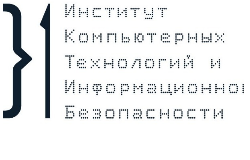
## Федеральное государственное автономное образовательное

## учреждение высшего образования

## «ЮЖНЫЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

**Институт компьютерных технологий и информационной безопасности**

**Кафедра Математического обеспечения и применения ЭВМ**

**ОТЧЕТ**

по индивидуальному заданию

по курсу «Проектирование трансляторов и интерпретаторов»

Вариант задания: Множества и операции над ними

Выполнила:

студентка группы КТмо1-8

Янкина Екатерина

Проверила:

ст. преподаватель каф. МОП ЭВМ

Балабаева И.Ю.

Оценка

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

«\_\_\_\_» \_\_\_\_\_\_\_\_\_\_ 2017 г.

Таганрог, 2017 г.

# Постановка задачи

Разработать синтаксис, семантику и транслятор для предметно-ориентированного языка в соответствии с вариантом.

Каждый предметно-ориентированный язык должен включать:

* + стандартные арифметические операции и типы данных;
  + не менее 2 типов данных для объектов предметной области (описание и использование);
  + операции над введенными типами данных (не менее 4);
  + операторы ввода и вывода для организации диалога с пользователем в текстовом или графическом режиме;
  + операторы условного, безусловного перехода и цикла.

# Описание синтаксиса предметно-ориентированного языка

## 2.1 Типы данных

### 2.1.1 Стандартные типы данных

Язык содержит пять стандартных типов данных:

|  |  |  |  |
| --- | --- | --- | --- |
| Формат | Тип | Диапазон значений | Значение по умолчанию |
| целые числа | byte | (0:255) | 0 |
| целые числа | int | (-2147483648: 2147483647) | 0 |
| числа с плавающей точкой | double | (5E-324 : 1,7Е+308) | 0 |
| логический тип данных | bool | (true, false) | false |
| строковый тип данных | string |  | “” |

### 2.1.2 Предметные типы данных

Предметно-ориентированный язык содержит в себе два предметных типа данных: непрерывное множество и дискретное множество.

Непрерывное множество - линейно упорядоченное множество X, все собственные сечения которого являются дедекиндовыми сечениями, то есть при любом разбиении X на два непустых подмножества A и B таком, что каждый элемент из A предшествует каждому элементу из B, либо в A есть наибольший элемент, но в B нет наименьшего элемента, либо в A нет наибольшего элемента, но в B есть наименьший элемент.

С геометрической точки зрения, непрерывное множество можно представить как интервал с двумя концами (крайними элементами множества). При этом следует учитывать, что концы интервала могут быть открытыми и закрытыми (рисунок 1).



Рисунок 1 – Виды интервалов

Таким образом, непрерывное множество представляет собой непрерывный интервал упорядоченных значений (от минимального до максимального) с включением или не включением крайних элементов.

Следует отметить, что элементами непрерывного множества при объявлении могут быть только целочисленные значения (положительные или отрицательные), причем между ними должно соблюдаться строгое неравенство: первое значение < второго значения.

Дискретное множество – это набор неповторяющихся дискретных элементов одного типа (целые числа, дробные числа, строки). Множество не может быть элементом другого множества!

## 2.2 Операции над предметными типами данных

Для работы с множествами введены следующие бинарные операции: объединение, пересечение, разность, симметричная разность.

|  |  |  |  |
| --- | --- | --- | --- |
| Операция | Демонстрация | Непрерывное множество | Дискретное множество |
| Объединение | https://upload.wikimedia.org/wikipedia/commons/thumb/3/30/Venn0111.svg/220px-Venn0111.svg.png | A=[-5..5], B=(3..100)  AUB=[-5..100) | A={0;1;2;3;5}, B={3;100}  AUB={0;1;2;3;5;100} |
| Пересечение | https://upload.wikimedia.org/wikipedia/commons/thumb/9/99/Venn0001.svg/220px-Venn0001.svg.png | A=[-5..5], B=(3..100)  A∩B=(3..5] | A={0;1;2;3;5}, B={3;100}  A∩B={3} |
| Разность | https://upload.wikimedia.org/wikipedia/commons/thumb/e/e6/Venn0100.svg/220px-Venn0100.svg.png | A=[-5..5], B=(3..100)  A/B=[-5..3] | A={0;1;2;3;5}, B={3;100}  A/B={0;1;2;5} |
| Симметричная разность | https://upload.wikimedia.org/wikipedia/commons/thumb/4/46/Venn0110.svg/220px-Venn0110.svg.png | A=[-5..5], B=(3..100)  A∆B=[-5..3] U (5..100) | A={0;1;2;3;5}, B={3;100}  A∆B={0;1;2;5}U{100} |

## 2.3 Синтаксис языка

Структура программы:

* заголовок программы;
* раздел меток;
* раздел констант;
* раздел описания переменных;
* раздел операторов.

Обязательными среди них являются: заголовок, раздел переменных, раздел операторов.

Пример структуры программы:

|  |  |
| --- | --- |
| Заголовок: | PROGRAM Test1  { |
| Раздел меток: | LABEL a,b; |
| Раздел констант: | CONST  int c=5;  string d="hello"; |
| Раздел переменных: | VAR  int var1; |
| Раздел операторов: | BEGIN  Readln( var1 );  a:WHILE (var1 < 0) {  var1=var1+1;  };  Writeln ("Bla-Bla-Bla", "IsNotTrue"  );  END;  } |

Тело программы заключено в фигурные скобки «{» … «}».

Раздел меток начинается с ключевого слова «LABEL», имена меток перечисляются при помощи запятых. Раздел меток заканчивается точкой с запятой «;».

Раздел констант начинается с ключевого слова «CONST». Описание константы состоит из типа, имени константы, знака «=», значения константы. Значение должно соответствовать типу!

Описания констант разделены точкой с запятой «;».

Раздел переменных начинается с ключевого слова «VAR». Описание переменной состоит из типа, названия переменной. Описания разделены между собой точкой с запятой «;».

Раздел операторов начинается с ключевого слова «BEGIN», завершается ключевым словом «END». Каждый оператор содержит свой собственный синтаксис. Каждый оператор должен быть завершен знаком «;».

Метка ставится перед оператором. Состоит из имени метки и двоеточия «:».

Раздел операторов может содержать в себе следующие операторы:

* присваивания;
* условного перехода;
* безусловного перехода;
* ввода/вывода;
* цикла.

### 2.3.1 Оператор присваивания

Общий вид оператора присваивания: Переменная «=» Значение «;».

Значение может быть константой (числом, строкой, константой из раздела констант, логической константой («TRUE», «FALSE»), явно заданным множеством(«{»…«}»)), переменной соответствующего типа, выражением.

Значение выражения должно быть соответствующего переменной типа. Выражение может состоять из логических операций «OR», «AND», знаков сравнения (==,!=,<,>,<=,>=), арифметических знаков (+,-,\*, /).

### 2.3.2 Оператор условного перехода

Общий вид оператора условного перехода:

IF (Выражение) {Операторы} [ELSE {Операторы}] ;

Результатом выражения должно быть значение типа bool.

Пример:

IF (a==b) {a=a+1; b=b-1;};

IF (a==b) {a=a+1; b=b-1;} ELSE {a=b;} ;

В разделах операторов обязательно должно присутствовать операторы!

### 2.3.3 Оператор безусловного перехода

Общий вид оператора безусловного перехода: GOTO Имя\_метки ;

Имя\_метки обязательно должно быть объявлено, метка должна быть установлена.

Пример:

LABEL a1;

VAR int a,b;

BEGIN

a1: a=a+b;

GOTO a1;

END …

### 2.3.4 Оператор ввода/вывода

Общий вид оператора ввода: READ (Список\_переменных);

READLN (Список\_переменных);

Общий вид оператора вывода: WRITE (Выражение);

WRITELN (Выражение);

Выражением может быть строка, переменная, число, арифметическое выражение.

Пример:

READLN (a,b,c);

WRITELN (“a=”, a);

WRITE (“d=”, a+b-c);

### 2.3.5 Оператор цикла

Общий вид оператора цикла: WHILE (Выражение) {Операторы};

В разделе операторов обязательно должен быть хотя бы один оператор!

Выражение должно быть логическим (результат типа bool)!

Пример:

WHILE (var1 < 0) {

var1=var1+1;

};

### 2.3.6 Предметные операторы языка

//TODO

# Разработка грамматики языка

## 3.1 До приведения к классу LL

Программа = Заголовок ";" Блок "."

Заголовок = "PROGRAM" Имя ["(" ИмяФайла ");"]

ИмяФайла = Имя

Имя = Буква { БукваИлиЦифра }

Буква = "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"| "z"

БукваИлиЦифра = Буква|Цифра

Цифра = "1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"

Блок = [РазделМеток][РазделКонстант] РазделПеременных РазделОператоров

РазделМеток = "LABEL" Метка {"," Метка} ";"

Метка = Имя

РазделКонстант = "CONST" ОпределениеКонстанты {"," ОпределениеКонстанты} ";"

ОпределениеКонстанты = Имя "=" Константа

Константа = Число|Имя|Знак Имя|"""Имя"""

Число = [Знак] ЧислоБезЗнака

Знак = "+"|"-"

ЧислоБезЗнака = ЦелоеБезЗнака|ВещественноеБезЗнака

ЦелоеБезЗнака = Цифра {Цифра}

ВещественноеБезЗнака = Цифра {Цифра} "." {Цифра} ["E" ["+"|"-"] Цифра {Цифра}]

РазделПеременных = "VAR" ОбъявлениеПеременной ";" {ОбъявлениеПеременной ";"}

ОбъявлениеПеременной = Имя {"," Имя} ":" Тип

Тип = ПростойТип|ПредметныйТип

ПростойТип = "int"|"byte"|"real"|"double"|"bool"|"char"

ПредметныйТип = НепрерывноеМножество | ПеречисляемоеМножество

НепрерывноеМножество = "Set of" ЦелоеБезЗнака ".." ЦелоеБезЗнака

ПеречисляемоеМножество = "Set of" ПростойТип

РазделОператоров = СоставнойОператор

СоставнойОператор = "BEGIN" Оператор {";" Оператор} "END;"

Оператор = [Метка ":"] НепомеченныйОператор

НепомеченныйОператор = ПростойОператор | СложныйОператор

ПростойОператор = ОператорПрисваивания | ОператорПерехода | ОператорВводаВывода

СложныйОператор = СоставнойОператор | УсловныйОператор | ОператорЦикла

ОператорПрисваивания = ИмяПеременной "=" Выражение

ИмяПеременной = Имя

Выражение = ЧисловоеВыражение|ЛогическоеВыражение|МножественноеВыражение

ЧисловоеВыражение = Слагаемое|Знак Слагаемое|ЧисловоеВыражение Знак Слагаемое

Слагаемое = Множитель|Слагаемое МультипликативныйЗнак Множитель

МультипликативныйЗнак = "\*"|"/"

Множитель = ИмяПеременной | Число | "(" ЧисловоеВыражение ")"

ЛогическоеВыражение = ЛогическоеЗначение | ЛогическоеВыражение ЛогическаяОперация ЛогическоеЗначение

ЛогическаяОперация = "AND"|"OR"

ЛогическоеЗначение = ЛогическаяКонстанта|ИмяПеременной|Отношение|"(" ЛогическоеВыражение ")"

ЛогическаяКонстанта = "TRUE"|"FALSE"

Отношение = ЧисловоеВыражение ЗнакОтношения ЧисловоеВыражение

ЗнакОтношения = "=="|"!="|"<"|">"|"<="|">="

МножественноеВыражение = МножественныйОперанд | МножественныйОперанд МножественнаяОперация Множество

МножественнаяОперация = "Union"|"Cros"|"Res"|"Diff"

МножественныйОперанд = Множество | "(" МножественноеВыражение ")"

Множество = ИмяМножества | "{" ЗначениеЭлемента {"," ЗначениеЭлемента} "}"

ЗначениеЭлемента = ИмяПеременной | Число | ЛогическаяКонстанта | Константа

ИмяМножества = Имя

ОператорПерехода = "GOTO" Метка

ОператорВводаВывода = ОператорВвода|ОператорВывода

ОператорВвода = ("Read"|"Readln") "(" СписокПеременных ")"

СписокПеременных = ИмяПеременной {"," ИмяПеременной}

ОператорВывода = ("Write"|"Writeln") "(" СписокВыражений ")"

СписокВыражений = {СтрокаИнформации} {ИмяПеременной}

СтрокаИнформации = Имя

УсловныйОператор = "IF" "(" ЛогическоеВыражение ")" "THEN" Оператор ["ELSE" Оператор]

ОператорЦикла = ЦиклСПредусловием | ЦиклСПостусловием | ЦиклСпараметром

ЦиклСПредусловием = "WHILE" Выражение "DO" Оператор

ЦиклСПостусловием = "REPEAT" Оператор {";" Оператор} "UNTIL" Выражение

ЦиклСпараметром = "FOR" ПараметрЦикла "=" СписокЦикла "DO" Оператор

ПараметрЦикла = Имя

СписокЦикла = НачЗнач "TO" КонечЗнач | НачЗнач "DOWNTO" КонечЗнач

НачЗнач = Выражение

КонечЗнач = Выражение

## 3.2 Приведение к классу LL

### 3.2.1 Факторизация

Множество:

ПредметныйТип = НепрерывноеМножество | ПеречисляемоеМножество

НепрерывноеМножество = **"Set of"** ЦелоеБезЗнака ".." ЦелоеБезЗнака

ПеречисляемоеМножество = **"Set of"** ПростойТип

->

ПредметныйТип = **"Set of"** (НепрерывноеМножество | ПеречисляемоеМножество)

НепрерывноеМножество = ЦелоеБезЗнака ".." ЦелоеБезЗнака

ПеречисляемоеМножество = ПростойТип

### 3.2.2 Левая рекурсия

Числовое выражение:

**ЧисловоеВыражение** = Слагаемое|Знак Слагаемое|**ЧисловоеВыражение** Знак Слагаемое

**->**

**ЧисловоеВыражение** = (Слагаемое|Знак Слагаемое){Знак Слагаемое}

Слагаемое:

**Слагаемое** = Множитель|**Слагаемое** МультипликативныйЗнак Множитель

->

**Слагаемое** = Множитель {МультипликативныйЗнак Множитель}

Логическое выражение:

**ЛогическоеВыражение** = ЛогическоеЗначение|**ЛогическоеВыражение** ЛогическаяОперация ЛогическоеЗначение

->

**ЛогическоеВыражение** = ЛогическоеЗначение {ЛогическаяОперация ЛогическоеЗначение}

### 3.2.3 Проверка следующего символа

Метка = Имя «:»

Переменная = Имя

Возникает проблема определения метки. Встречая имя в разделе операторов, анализатор переходит к определению Оператора\_Присваивания = Имя «=» Выражение.

Встречая после имени «:» внвлизатор выдает ошибку.

Для решения данной проблемы можно использовать проверку следующего символа scanner.Peek().

Функция проверки будет выглядеть следующим образом:

bool LabelOperation() {

Token next = scanner.Peek();

return next.val == ":";

}

Функция возвращает true, если следующий символ = «:».

Вызов функции в грамматике:

Operator<out Operator op1>=

[IF (LabelOperation()) Metka<out labelname>] UnlabeledOperator ";".

Если функция возвращает true, то «Metka». Затем идет оператор. Если функция возвращает false, то анализатор идет мимо «Metka» сразу к UnlabeledOperator.

## 3.3 После приведения к классу LL

COMPILER SET

const int

undef\_ = 0, int\_ = 1, bool\_ = 2, byte\_ = 3, double\_ = 5, string\_=6, label\_=7,

SetOfint = 8, SetOfbool = 9, SetOfbyte = 10, SetOfdouble = 12, SetOfstring = 13;

const int

var\_ = 0, proc\_ = 1, const\_ = 3;

public SymbolTable tab;

public CodeGenerator gen;

bool LabelOperation() {

Token next = scanner.Peek();

return next.val == ":";

}

/\*---------------------------------------------------------------------------------------------\*/

CHARACTERS

letter = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz".

digit = "0123456789".

verbatimStringstring = ANY -'"'.

cr = '\r'.

lf = '\n'.

tab = '\t'.

TOKENS

name\_ = letter {letter | digit}.

unsignedInteger = digit {digit}.

realNumber = digit {digit} "." {digit} ["E" ["+"|"-"] digit {digit}].

stringg = "\"" {verbatimStringstring} "\"".

COMMENTS FROM "/\*" TO "\*/" NESTED

COMMENTS FROM "//" TO lf

IGNORE cr + lf + tab

PRODUCTIONS

/\*---------------------------------------------------------------------------------------------\*/

SET="PROGRAM" Name

'{'[LabelSection][ConstantSection]VariablesSection OperatorsSection'}'

Name= name\_

LabelSection = "LABEL" Name{"," Name} ";" .

ConstantSection = "CONST" ConstantsDefinition";"{ConstantsDefinition ";"}.

ConstantsDefinition = Type Name "=" Constant.

Constant =([Sign](NumberWithoutSign|Name))|stringg|LogicalConst.

LogicalConst = "TRUE"|"FALSE".

NumberWithoutSign = unsignedInteger|realNumber.

Sign= "+"|"-".

VariablesSection ="VAR" VariablesDeclaration";"{VariablesDeclaration";"}.

VariablesDeclaration = Type Name {"," Name }

Type= SimpleType|SubjectType .

SimpleType= "int"| "byte" | "double"| "bool" | "string" .

SubjectType="Set" "Of" (ContinuousSet |EnumeratedSet ).

ContinuousSet = unsignedInteger ".." unsignedInteger .

EnumeratedSet= SimpleType

OperatorsSection = "BEGIN" {Operator (.tab.operSum++;.) } "END" ";".

Operator= [IF (LabelOperation()) Metka<out labelname> ] UnlabeledOperator";".

Metka= Name ":".

UnlabeledOperator=SimpleOperator | ComplicatedOperator

SimpleOperator=AssignmentsOperator| JumpOperator | InOutOperator.

ComplicatedOperator =IfElseOperator| CycleOperator

AssignmentsOperator = VariableName"=" Expression

VariableName= Name

Expression=Exp0{LogicSign Exp0}.

Exp0=Exp1[RatioSign Exp1].

Exp1=Exp2{Sign Exp2}.

Exp2=Operand { (MultiSign | VarietyOperation)}.

Operand = (["-"](Name|unsignedInteger |realNumber )|stringg |LogicalConst |Variety)

Variety= "{" ElementValue {"," ElementValue<out type2, out value2>} "}".

ElementValue=([Sign](unsignedInteger |realNumber|Name))|stringg|LogicalConst.

MultiSign ="\*"|"/"

RatioSign="=="|"!="|"<"|">"|"<="|">="

LogicSign="AND"|"OR"

VarietyOperation= "Union”|"Cros”|"Res”|"Diff”

JumpOperator= "GOTO" Name

InOutOperator=InOperator|OutOperator

InOperator= "Read"|"Readln" "(" VariableName {"," VariableName } ")"

OutOperator="Write"|"Writeln" "(" Expression {","Expression} ")"

IfElseOperator= "IF" "(" Expression")" "{" {Operator} "}" ["ELSE" "{"{Operator}"}"]

CycleOperator="WHILE" "(" Expression<out expr, out type> ")" "{"{Operator}"}"

END SET.

# Генерация сканера и парсера

После описания грамматики в файле .ATG необходимо сгенерировать парсер. Для этого надо запустить в консоли файл coco.exe с параметром в виде файла .ATG.   
Команда: coco Set.ATG -namespace Program.

После этого в папке расположения coco.exe появятся файлы Parser.cs и Scanner.cs.

Для того, чтобы проверить работу грамматики, необходимо сгенерировать .exe файл программы. Для этого удобнее всего создать проект в VisualStudio, добаить в него файлы Parser.cs и Scanner.cs. Вместо созданного основного файла программы (точки входа в программу) необходимо добавить файл Program.cs (название может отличаться).

Содержимое файла Program.cs:

using System;namespace Program {

class Set { public static void Main (string[] arg) { if (arg.Length > 0) { Scanner scanner = new Scanner(arg[0]); Parser parser = new Parser(scanner); parser.tab = new SymbolTable(parser); parser.gen = new CodeGenerator(); parser.Parse(); if (parser.errors.count == 0) { parser.gen.Decode(); parser.gen.Interpret("Set.IN"); } } else Console.WriteLine("-- No source file specified"); }}}

Если в папке coco есть файлы SymbolTable и CodeGenerator, их тоже следует добавить в проект (или не вызывать в Program.cs). Эти файлы не являются обязательными. В них можно реализовывать обработку синтаксиса грамматики.

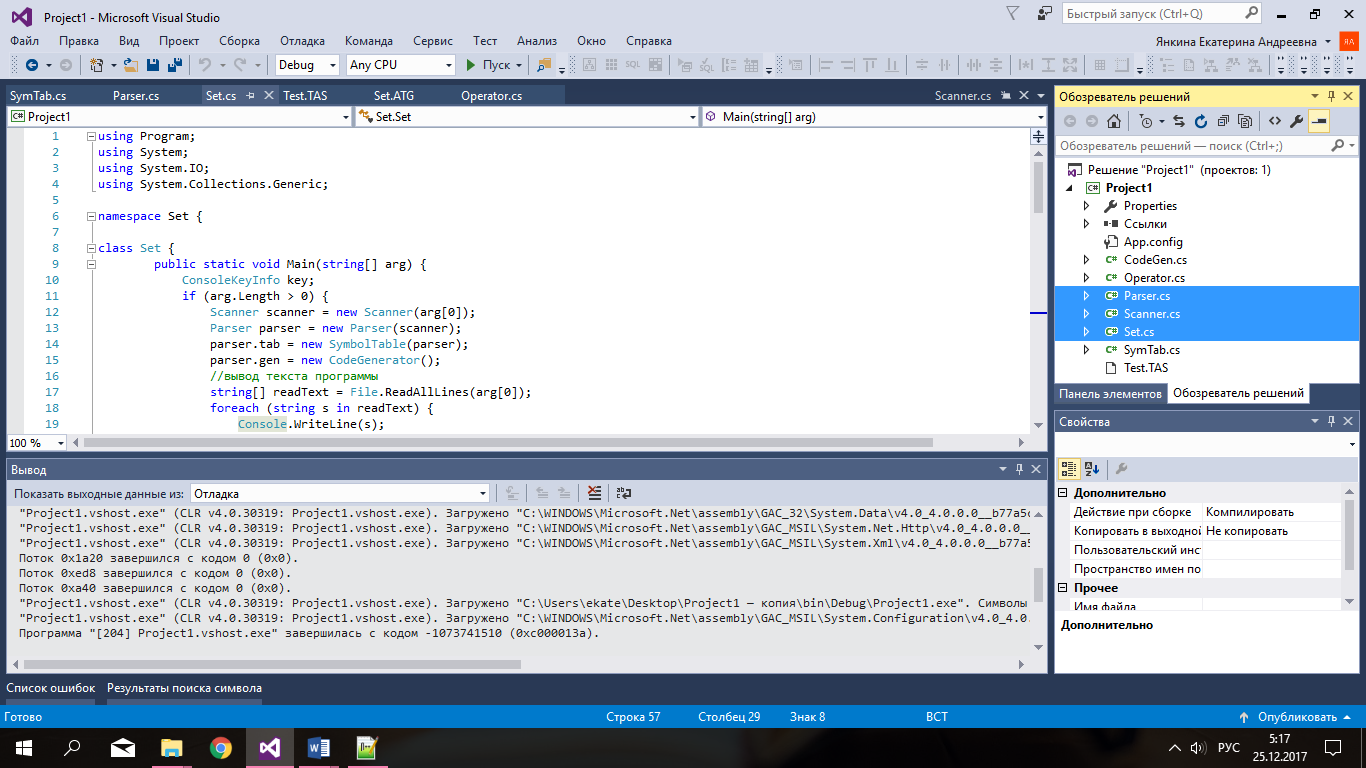


Рисунок2 – Компиляция синтаксического анализатора

Если в описании файла грамматики есть ошибки, то в файле parser.cs они будут подсвечены visual studio. Таким образом можно отлаживать программу. После изменения .ATG файла грамматики необходимо заново генерировать сканер и парсер, заменять их в проекте.

Некоторые правила описания грамматики:

После описания каждого правила необходимо ставить точку.

Токены не могут содержать пробелы. «Set of» -> «Set» «of».

Coco генерирует сканер (лексический анализатор) и парсер (синтаксический анализатор). Сканер строится как детерминированный конечный автомат, а парсер — рекурсивным спуском.

Парсер проходит по токенам от начала программы, «спускаясь вниз» по грамматике.

Каждый токен состоит из параметров:

class Token {

int kind; // тип токена

int pos; // позиция (в байтах )от начала разбираемой программы

int charPos; // позиция (в символах) от начала разбираемой программы

int col; // колонка (позиция от начала строки)

int line; // текущая строка разбираемой программы

string val; // значение текущего токена

Token next; // след.токен (т.к. они хранятся в связанном списке)

}

Ключевую роль играет параметр «Token.kind» - тип токена, т.к. именно по нему осуществляется проверка синтаксиса и переходы по раскрытию лексем.

Парсер присваивает токену определенный тип в зависимости от встреченной комбинации символов. Затем, в зависимости от этого типа, происходит переход далее по грамматике языка. Если же встреченный тип токена не позволяет совершить переход ни по одной из возможных ветвей, то парсер выдает ошибку.

Например проверка типа данных выглядит так:

void Type(out int type) {

{// если тип токена один из таб. «простых типов данных»

if (StartOf(2))

// то это простой тип данных (int, byte, bool, double…)

SimpleType(out type);

//если тип токена=24 («Set Of»)

} else if (la.kind == 24) {

// то это спец. тип данных - «множество»

SubjectType(out type);

// иначе - ошибка

} else SynErr(57);

}

Таким образом, парсер лишь обращается к таблице возможных типов токенов. Создается она в сканере. Полученная таблица терминальных символов:

|  |  |
| --- | --- |
| case "PROGRAM": t.kind = 5; break;  case "LABEL": t.kind = 8; break;  case "CONST": t.kind = 11; break;  case "TRUE": t.kind = 13; break;  case "FALSE": t.kind = 14; break;  case "VAR": t.kind = 17; break;  case "int": t.kind = 18; break;  case "byte": t.kind = 19; break;  case "real": t.kind = 20; break;  case "double": t.kind = 21; break;  case "bool": t.kind = 22; break;  case "string": t.kind = 23; break;  case "Set": t.kind = 24; break; | case "Of": t.kind = 25; break;  case "BEGIN": t.kind = 27; break;  case "END": t.kind = 28; break;  case "AND": t.kind = 38; break;  case "OR": t.kind = 39; break;  case "Union": t.kind = 40; break;  case "Cros": t.kind = 41; break;  case "Res": t.kind = 42; break;  case "Diff": t.kind = 43; break;  case "GOTO": t.kind = 44; break;  case "Read": t.kind = 45; break;  case "Readln": t.kind = 46; break;  case "Write": t.kind = 49; break; |

# Реализация интерпретатора

Парсер и интерпретатор выполняются последовательно друг за другом.

Во время работы парсера все метки, константы, переменные записываются в класс:

public class Obj : Value { // объект, описывающий объявленное имя

public string name;

public int type;

public int kind;

public Object value;

public override Object GetValue() {

return value;

}

}

Для реализации множества был написан класс Set:

public class Set{

public List<Object> list = new List<Object>();

public Kind kind;

public Set() {

}

public void Add(Kind k, Object obj) {

kind = k;

list.Add(obj);

}

}

Множество состоит из списка элементов типа Object, типа множества (kind): SetOfint = 8, SetOfbyte = 10, SetOfdouble = 12, SetOfstring = 13.

Создание множества:

public void NewNumericSet(string setName, int type, Value value) {

if (type == SetOfint) {

List <int> s = new List<int>();

s.Add(Convert.ToInt32(value));

}

else if (type == SetOfbyte) {

List<byte> s = new List<byte>();

s.Add(Convert.ToByte(value));

}

else if (type == SetOfdouble) {

List<double> s = new List<double>();

s.Add(Convert.ToDouble(value));

}

else if (type == SetOfstring) {

List<string> s = new List<string>();

s.Add(Convert.ToString(value));

}

}

Во время работы парсера все операторы последовательно записываются в **список операторов.** Работа интерпретатора заключается в последовательном выполнении операторов из списка.

Поскольку в языке перед операторами встречаются метки, то в начале своей работы программа проходит по всем операторам, добавляя встреченные по пути метки вместе с ссылками на соответствующие им операторы.

foreach (Operator op in parser.tab.operators) {

if (op is JumpOperator) {

string label = ((JumpOperator)op).labelName;

if (label == null) continue;

foreach (Operator labeledOp in parser.tab.operators)

{

if (labeledOp.label == label) {

((JumpOperator)op).oper = labeledOp;

break;

}

}

}

}

Затем происходит проход по всему списку операторов с проверкой типа оператора (цикл, условие, безусловный переход, ввод-вывод) и вызовом соответствующих функций обработки.

Для вычисления значений переменных (выражений) реализован метод

public static Object Calculate(Object v1, Op op, Object v2), который в зависимости от операции и типа вычисляет конечное значение.

Например, для реализации операций над множествами происходит проверка типа множества и типа операции:

if (kind == Kind.SetOfint) {

switch (op) {

case Op.UNION: {

List<int> int1 = new List<int>();

List<int> int2 = new List<int>();

foreach (var ob in s1.list) {

int a = (int)(((ConstValue)ob).GetValue());

int1.Add(a);

}

foreach (var ob in s2.list) {

int a = (int)(((ConstValue)ob).GetValue());

int2.Add(a);

}

List<int> res1 = new List<int>();

res1.AddRange(int1);

foreach (int ob in int2) {

if (!res1.Contains(ob)) res1.Add(ob);

}

foreach (int ob in res1) {

res.list.Add(new ConstValue(ob));

}

return (res);

}

# Тестирование интерпретатора

|  |  |
| --- | --- |
| Работа с множествами. Целые числа | |
| PROGRAM Test1  {  VAR  Set Of { 11 .. 100 } a1;  Set Of string s1,s5;  Set Of int s2, s6;  Set Of byte s3;  Set Of double s4;  BEGIN  s2 = {1,2,3,4,5};  s6 = {3,4,5,6,7};  s1 = {"1","2","3","4","5"};  Writeln ("Union:", s2 Union s6);  Writeln ("Cros:", s2 Cros s6);  Writeln ("Res:", s2 Res s6);  Writeln ("Diff:", s2 Diff s6);  END;  } |  |
| Работа с множествами. Строки | |
| PROGRAM Test1  {  VAR  Set Of { 11 .. 100 } a1;  Set Of string s1,s5;  Set Of double s2, s6;  Set Of byte s3;  Set Of double s4;  BEGIN  s2 = {1.5,2.2,3.0,4.0,5.3};  s6 = {3.0,4.0,5.3,6.8,7.77};  s1 = {"a","b","c","d","e"};  s5 = {"c","d","e","f","g"};  Writeln ("Union:", s1 Union s5);  Writeln ("Cros:", s1 Cros s5);  Writeln ("Res:", s1 Res s5);  Writeln ("Diff:", s1 Diff s5);  END;  } |  |
| Работа с множествами. Числа с плавающей точкой | |
| PROGRAM Test1  {  VAR  Set Of { 11 .. 100 } a1;  Set Of string s1,s5;  Set Of double s2, s6;  Set Of byte s3;  Set Of double s4;  BEGIN  s2 = {1.5,2.2,3.0,4.0,5.3};  s6 = {3.0,4.0,5.3,6.8,7.77};  s1 = {"a","b","c","d","e"};  s5 = {"c","d","e","f","g"};  Writeln ("Union:", s2 Union s6);  Writeln ("Cros:", s2 Cros s6);  Writeln ("Res:", s2 Res s6);  Writeln ("Diff:", s2 Diff s6);  END;  } |  |

Проверка синтаксиса написанной программы

|  |  |
| --- | --- |
| Тест1. Отсутствие операторов в программе | Результат |
| PROGRAM Test1  {  VAR  bool a1;  BEGIN  END;  } |  |
| PROGRAM Test1  {  VAR  int a;  int b;  int d;  double c;  BEGIN  a=2;  b=9;  c=a/b;  d=a\*b;  Writeln ("d=",d);  Write ("c=",c);  END;  } |  |
| PROGRAM Test1  {  VAR  int a;  int b;  bool d,f;  int c;  BEGIN  a=2; b=8;  d=b<a\*b;  Writeln ("d=",d);  f = FALSE;  f = f AND d;  Writeln ("f=",f);  f= f OR d;  Writeln ("f=",f);  END;  } |  |
| Отсутствие операторов в цикле | |
| PROGRAM Test1  {  LABEL a,b;  VAR  int var1;  BEGIN  Readln( var1 );  WHILE (var1 < 0) {  };  Writeln ("Bla-Bla-Bla", "IsNotTrue"  );  END;  } |  |

# Приложение

## Приложение А. SET.ATG

COMPILER SET

const int

undef\_ = 0, int\_ = 1, bool\_ = 2, byte\_ = 3, double\_ = 5, string\_=6, label\_=7,

SetOfint = 8, SetOfbool = 9, SetOfbyte = 10, SetOfdouble = 12, SetOfstring = 13;

const int

var\_ = 0, proc\_ = 1, const\_ = 3;

public SymbolTable tab;

public CodeGenerator gen;

bool InCondition = false;

bool LabelOperation() {

Token next = scanner.Peek();

return next.val == ":";

}

/\*------------------------------------------------------------------------------------\*/

CHARACTERS

letter = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz".

digit = "0123456789".

verbatimStringstring = ANY -'"'.

cr = '\r'.

lf = '\n'.

tab = '\t'.

TOKENS

name\_ = letter {letter | digit}.

unsignedInteger = digit {digit}.

realNumber = digit {digit} "." {digit} ["E" ["+"|"-"] digit {digit}].

stringg = "\"" {verbatimStringstring} "\"".

COMMENTS FROM "/\*" TO "\*/" NESTED

COMMENTS FROM "//" TO lf

IGNORE cr + lf + tab

PRODUCTIONS

/\*---------------------------------------------------------------------\*/

SET (. string name; .)

= "PROGRAM" Name<out name> (. tab.OpenScope(); .)

'{'

[LabelSection]

[ConstantSection]

VariablesSection

OperatorsSection

'}'

(. tab.CloseScope();

if (tab.operSum == -1) {

SemErr("No executed operators in the program :)");

ConsoleKeyInfo key = Console.ReadKey();

}

.).

/\*---------------------------------------------------------------------\*/

Name<out string name>

= name\_ (. name = t.val; .).

/\*---------------------------------------------------------------------\*/

LabelSection (. string name; int type=label\_; .)

= "LABEL" Name<out name> (. tab.NewObj(name, var\_, type); .)

{"," Name<out name> (. tab.NewObj(name, var\_, type); .)

} ";" .

/\*---------------------------------------------------------------------\*/

ConstantSection

= "CONST" ConstantsDefinition";"{ConstantsDefinition ";"}.

/\*---------------------------------------------------------------------\*/

ConstantsDefinition (. string name; int type; .)

= (. type=undef\_; .)

Type<out type> (. if (type>7) {

SemErr("Set can't be a constant"); ConsoleKeyInfo key = Console.ReadKey();}.)

Name<out name> (. tab.NewObj(name, const\_, type); .)

"=" Constant. //присвоить значение константе

/\*---------------------------------------------------------------------\*/

Constant (. string name; Op op; .)

=([Sign<out op>](NumberWithoutSign|Name<out name>))|stringg|LogicalConst.

LogicalConst = "TRUE"|"FALSE".

NumberWithoutSign = unsignedInteger|realNumber.

/\*---------------------------------------------------------------------\*/

Sign<out Op op> (. op = Op.ERR; .)

=

"+" (. op = Op.ADD; .)

|"-" (. op = Op.SUB; .)

.

/\*---------------------------------------------------------------------\*/

VariablesSection ="VAR" VariablesDeclaration";"{VariablesDeclaration";"}.

/\*---------------------------------------------------------------------\*/

VariablesDeclaration (. string name; int type; .)

= Type<out type>

Name<out name> (. tab.NewObj(name, var\_, type); .)

{"," Name<out name> (. tab.NewObj(name, var\_, type); .)

}.

/\*---------------------------------------------------------------------\*/

Type<out int type> (. type = undef\_; .)

= SimpleType<out type>|SubjectType<out type>.

/\*---------------------------------------------------------------------\*/

SimpleType<out int type> (. type = undef\_; .)

=

"int" (. type = int\_; .)

| "byte" (. type = byte\_; .)

| "double" (. type = double\_; .)

| "bool" (. type = bool\_; .)

| "string" (. type = string\_; .).

/\*---------------------------------------------------------------------\*/

SubjectType<out int type> (. int type1=0; type = undef\_;.)

=

"Set" "Of" (EnumeratedSet<out type1>|ContinuousSet<out type1>) (. type = type1;.).

/\*---------------------------------------------------------------------\*/

ContinuousSet<out int type> = "{" ["-"] unsignedInteger ".." ["-"] unsignedInteger "}" (. type = SetOfint; .).

/\*---------------------------------------------------------------------\*/

EnumeratedSet<out int type> = SimpleType<out type> (. if (type == int\_) type = SetOfint;

else if (type == byte\_) type = SetOfbyte;

else if (type == double\_) type = SetOfdouble;

else if (type == bool\_) type = SetOfbool;

else if (type == string\_) type = SetOfstring; .).

/\*---------------------------------------------------------------------\*/

OperatorsSection (. Operator op; .)

= "BEGIN" {Operator<out op> (.tab.operSum++;.) } "END" ";".

/\*---------------------------------------------------------------------\*/

Operator<out Operator op1> (. Operator op; string labelname=null; .)

= [IF (LabelOperation()) Metka<out labelname>

]

UnlabeledOperator<out op> (. if (labelname != null) {

if ((tab.FindLabel(labelname))==false){

if (InCondition==true) {

SemErr("You can't put a label here!");ConsoleKeyInfo key = Console.ReadKey();

}

else

{op.label = labelname; tab.labelsOp.Add(labelname); }

}

else {SemErr("the label is already set! ");ConsoleKeyInfo key = Console.ReadKey();}

}.)

";" (. op1=op; .).

/\*---------------------------------------------------------------------\*/

Metka<out string labelname> (. string name; Obj obj; .)

= Name<out name> ":" (. obj = tab.Find(name);

if (obj!=null) tab.NewLabel(name,t); labelname=name; .).

/\*---------------------------------------------------------------------\*/

UnlabeledOperator<out Operator op> (.op=null; Operator simop, compop;.)

=

SimpleOperator<out simop> (. op = simop; .)

| ComplicatedOperator<out compop> (. op = compop; .).

/\*---------------------------------------------------------------------\*/

SimpleOperator<out Operator op> (. op = null; OperAssign ass; JumpOperator jmp; InOutOperator inout; .)

=

(AssignmentsOperator<out ass> (. op = ass;.)

| JumpOperator <out jmp> (. op = jmp; .)

| InOutOperator<out inout> (. op = inout; .)

) (. tab.operators.AddLast(op); .).

/\*---------------------------------------------------------------------\*/

ComplicatedOperator<out Operator op> (. op = null; IfOperator ifop; CycleOperator cycop; .)

=

IfElseOperator<out ifop> (. op = ifop; .)

| CycleOperator<out cycop> (. op = cycop; .).

/\*---------------------------------------------------------------------\*/

AssignmentsOperator<out OperAssign assign> (. Kind type; string name; Obj obj; int adr; Object value1; Expression expr; .)

= (. assign = null; .)

VariableName<out name> (. obj = tab.Find(name);

if (obj == null) {SemErr("Undefined variable");ConsoleKeyInfo key = Console.ReadKey();}

if (obj.kind == const\_) {SemErr("Assignment to a constant"); ConsoleKeyInfo key = Console.ReadKey();}.)

"="

Expression<out expr, out type> (.

if (obj.type < 8) {

if (type != Process.KindOf(obj.value)) { SemErr("Incompatible types!"); ConsoleKeyInfo key = Console.ReadKey(); }

else assign = new OperAssign(obj, expr);

}

else {

if (obj.type == SetOfint) {

if (type != Kind.SetOfint) { SemErr("Incompatible types!"); ConsoleKeyInfo key = Console.ReadKey(); }

else assign = new OperAssign(obj, expr);

}

else if (obj.type == SetOfbyte) {

if (type != Kind.SetOfint) { SemErr("Incompatible types!"); ConsoleKeyInfo key = Console.ReadKey(); }

else assign = new OperAssign(obj, expr);

}

else if (obj.type == SetOfdouble) {

if (type != Kind.SetOfdouble) { SemErr("Incompatible types!"); ConsoleKeyInfo key = Console.ReadKey(); }

else assign = new OperAssign(obj, expr);

}

else if (obj.type == SetOfstring) {

if (type != Kind.SetOfstring) { SemErr("Incompatible types!"); ConsoleKeyInfo key = Console.ReadKey(); }

else assign = new OperAssign(obj, expr);

}

}

.) .

/\*---------------------------------------------------------------------\*/

VariableName<out string name> = Name<out name>.

/\*---------------------------------------------------------------------\*/

Expression<out Expression expr, out Kind type> (. Op op; Expression expr1, expr2; Kind type2; .)

=

Exp0<out expr1, out type> (. expr = new Expression(expr1); .)

{LogicSign<out op> Exp0<out expr2, out type2> (. if (type != type2) {SemErr("Types uncompabitle");ConsoleKeyInfo key = Console.ReadKey();}

if (type != Kind.Boolean) {SemErr("Type unexpected");ConsoleKeyInfo key = Console.ReadKey();}

expr.Add(op, expr2); .)

}.

/\*---------------------------------------------------------------------\*/

Exp0<out Expression expr, out Kind type> (. Op op; Expression expr1, expr2; Kind type2; .)

=

Exp1<out expr1, out type> (. expr = new Expression(expr1); .)

[RatioSign<out op> Exp1<out expr2, out type2> (. if (type != type2) {SemErr("Types uncompabitle");ConsoleKeyInfo key = Console.ReadKey();}

if (op != Op.EQU && op != Op.NEQU && type != Kind.Numeric) {SemErr("Type unexpected");ConsoleKeyInfo key = Console.ReadKey();}

expr.Add(op, expr2); type = Kind.Boolean; .)

].

/\*---------------------------------------------------------------------\*/

Exp1<out Expression expr, out Kind type> (. Op op; Expression expr1, expr2; Kind type2; .)

=

Exp2<out expr1, out type> (. expr = new Expression(expr1); .)

{Sign<out op> Exp2<out expr2, out type2> (. if (type != type2) {SemErr("Types uncompabitle");ConsoleKeyInfo key = Console.ReadKey();}

if (type != Kind.Numeric) {SemErr("Type unexpected");ConsoleKeyInfo key = Console.ReadKey();}

expr.Add(op, expr2); .)

}.

/\*---------------------------------------------------------------------\*/

Exp2<out Expression expr, out Kind type> (. Kind type2; Op op; Value value1, value2; bool mult=false; .)

=

Operand<out type, out value1> (. expr = new Expression(value1); .)

{ (MultiSign<out op> (. mult=true; .)

| VarietyOperation<out op>)

Operand<out type2, out value2>

(. if (type != type2) {SemErr("Types uncompabitle");ConsoleKeyInfo key = Console.ReadKey();}

if (mult==true){

if (type != Kind.Numeric) {SemErr("Type unexpected");ConsoleKeyInfo key = Console.ReadKey();}

}

else {

if (((type != Kind.SetOfbyte) && (type != Kind.SetOfdouble) && (type != Kind.SetOfint) && (type != Kind.SetOfstring))

|| ((type2 != Kind.SetOfbyte) && (type2 != Kind.SetOfdouble) && (type2 != Kind.SetOfint) && (type2 != Kind.SetOfstring)))

{SemErr("Set operation, but operand isn't set");ConsoleKeyInfo key = Console.ReadKey();}

}

expr.Add(op, value2); .)

}.

/\*---------------------------------------------------------------------\*/

Operand<out Kind kind, out Value val> (. string name, str; Op op; int type; Kind k; Object value, variety; Obj obj; bool v = false;

type = undef\_; value =0.0; int minus = 0; kind=0; .)

= (. obj = null; val = null; .)

(["-" (. minus=1; .)

]

(Name<out name> (. obj = tab.Find(name);

if (obj.kind != proc\_) {

if (obj.value != null) {

if (obj.type > 7) {

v=true;

if (minus == 1) { SemErr("Exception: set=-set"); ConsoleKeyInfo key = Console.ReadKey(); }

switch (obj.type) {

case SetOfint: kind = Kind.SetOfint; break;

case SetOfbyte: kind = Kind.SetOfbyte; break;

case SetOfdouble: kind = Kind.SetOfdouble; break;

case SetOfstring: kind = Kind.SetOfstring; break;

}

}

else{

if (minus==0) value = obj.value;

else if (minus==1) value = tab.Oper(obj.type, obj.type, 0, obj.value, Op.SUB);

}

}

} else {SemErr("Variable expected");ConsoleKeyInfo key = Console.ReadKey();}

val = obj; .)

|unsignedInteger (. type = int\_; if (minus==0) value = Convert.ToInt32(t.val); else value = -Convert.ToInt32(t.val);

val = new ConstValue(value); .)

|realNumber (. type = double\_; if (minus==0) value = Double.Parse(t.val, System.Globalization.CultureInfo.InvariantCulture.NumberFormat);

else value = - Double.Parse(t.val, System.Globalization.CultureInfo.InvariantCulture.NumberFormat);

val = new ConstValue(value); .)

)

|stringg (. type = string\_; str = t.val; value = str.Substring(1, str.Length - 2); val = new ConstValue(value); .)

|LogicalConst (. type = bool\_; value = Convert.ToBoolean(t.val); val = new ConstValue(value); .)

|Variety<out k, out variety> (. val = new ConstValue(variety); kind=k; v = true; .)

)

(. if (!v) kind = Process.KindOf(value); .).

/\*---------------------------------------------------------------------\*/

Variety<out Kind kind, out Object value> (. Kind type1, type2;

Object value1, value2; kind = 0; value=0; .)

= (. value = null; Set s = new Set();.)

"{" ElementValue<out type1, out value1> (. if (type1 == Kind.Ints) kind = Kind.SetOfint;

else if (type1 == Kind.Bytes) kind = Kind.SetOfbyte;

else if (type1 == Kind.Doubles) kind = Kind.SetOfdouble;

else if (type1 == Kind.String) kind = Kind.SetOfstring;

s.Add(kind, value1);

.)

{"," ElementValue<out type2, out value2> (. if (type1 != type2) {SemErr("Different types of elements in the set!"); ConsoleKeyInfo key = Console.ReadKey();}

else {

if (type2 == Kind.Ints) kind = Kind.SetOfint;

else if (type2 == Kind.Bytes) kind = Kind.SetOfbyte;

else if (type2 == Kind.Doubles) kind = Kind.SetOfdouble;

else if (type2 == Kind.String) kind = Kind.SetOfstring;

s.Add(kind, value2);

}.)

} "}" (. value=s; .).

/\*---------------------------------------------------------------------\*/

ElementValue<out Kind kind, out Object val> (. Obj obj; string name,str; Op op; int type=undef\_; Object value=0; op=Op.ADD;value =0.0; int minus = 0; .)

= (. val = null; kind = 0;.)

(["-" (. minus=1; .)

]

(Name<out name> (. obj = tab.Find(name);

if (obj.kind != proc\_) {

if (obj.value != null) {

if (minus==0) value = obj.value;

else if (minus==1) value = tab.Oper(obj.type, obj.type, 0, obj.value, Op.SUB);}

else {SemErr("Variable value = null");ConsoleKeyInfo key = Console.ReadKey();}

} else {SemErr("Variable expected");ConsoleKeyInfo key = Console.ReadKey();}

val = obj;

if (obj.type == int\_) kind = Kind.Ints;

else if (obj.type == double\_) kind = Kind.Doubles;

else if (obj.type == byte\_) kind = Kind.Bytes;

else if (obj.type == string\_) kind = Kind.String;

else { SemErr("Bad variable type for element of set"); ConsoleKeyInfo key = Console.ReadKey(); }

.)

|unsignedInteger (. kind = Kind.Ints; if (minus==0) value = Convert.ToInt32(t.val); else value = -Convert.ToInt32(t.val);

val = new ConstValue(value); .)

|realNumber (. kind = Kind.Doubles; if (minus==0) value = Double.Parse(t.val, System.Globalization.CultureInfo.InvariantCulture.NumberFormat);

else value = - Double.Parse(t.val, System.Globalization.CultureInfo.InvariantCulture.NumberFormat);

val = new ConstValue(value); .)

)

|stringg (. kind = Kind.String; str = t.val;value = str.Substring(1, str.Length - 2); val = new ConstValue(value); .)

).

/\*---------------------------------------------------------------------\*/

MultiSign<out Op op> (. op = Op.ERR; .)

=

"\*" (. op = Op.MUL; .)

|"/" (. op = Op.DIV; .)

.

/\*---------------------------------------------------------------------\*/

RatioSign<out Op op> (. op = Op.ERR; .)

=

("==" (. op = Op.EQU; .)

|"!=" (. op = Op.NEQU; .)

|"<" (. op = Op.LSS; .)

|">" (. op = Op.GTR; .)

|"<=" (. op = Op.LEQ; .)

|">=" (. op = Op.GEQ; .)

).

/\*---------------------------------------------------------------------\*/

LogicSign<out Op op> (. op = Op.ERR; .)

=

("AND" (. op = Op.AND; .)

|"OR" (. op = Op.OR; .)

).

/\*---------------------------------------------------------------------\*/

VarietyOperation<out Op op> (. op = Op.ERR; .)

= "Union" (. op = Op.UNION; .)

|"Cros" (. op = Op.CROS; .)

|"Res" (. op = Op.RES; .)

|"Diff" (. op = Op.DIFF; .).

/\*---------------------------------------------------------------------\*/

JumpOperator<out JumpOperator op> (. string name; Obj obj;.)

= "GOTO" Name<out name> (. obj = tab.Find(name);

op = new JumpOperator(name); .).

/\*---------------------------------------------------------------------\*/

InOutOperator<out InOutOperator op> (. InOutOperator inop,outop; op=null; .)

=

InOperator<out inop> (. op=inop; .)

|OutOperator<out outop> (. op=outop; .).

/\*---------------------------------------------------------------------\*/

InOperator<out InOutOperator op> (. Obj obj; string name; int w=0; op=null; .)

= ("Read" (. w=2; .)

|"Readln" (. w=3; .)

) (. op = new InOutOperator(w); op.elementToInOut = new System.Collections.Generic.List<Value>(); .)

"(" VariableName<out name> (. obj = tab.Find(name); op.Add(obj); .)

{"," VariableName<out name> (. obj = tab.Find(name); op.Add(obj); .)

} ")".

/\*---------------------------------------------------------------------\*/

OutOperator<out InOutOperator op> (. Expression expr; Kind type; int w=0; op=null; .)

= ("Write" (. w=0; .)

|"Writeln" (. w=1; .)

) (. op = new InOutOperator(w); op.elementToInOut = new System.Collections.Generic.List<Value>(); .)

"(" Expression <out expr, out type> (. op.Add(expr); .)

{","Expression <out expr, out type> (. op.Add(expr); .)

} ")".

/\*---------------------------------------------------------------------\*/

IfElseOperator<out IfOperator op> (. Kind type; Expression expr; op=null; Operator ifop=null,elseop=null; .)

= "IF" "(" Expression<out expr, out type> (. if (type != Kind.Boolean) {SemErr("Expected boolean");ConsoleKeyInfo key = Console.ReadKey();} .)

")" (. op = new IfOperator(expr); tab.operators.AddLast(op); .)

"{" (. op.ifOperator = new System.Collections.Generic.List<Operator>(); .)

{ (. InCondition= true;.)

Operator<out ifop> (. op.ifOperator.Add(ifop);.)

} "}"

(. JumpOperator gotoEnd = new JumpOperator(new NothingOperator());

tab.operators.AddLast(gotoEnd); .)

["ELSE" "{" (. op.elseOperator = new System.Collections.Generic.List<Operator>(); .)

{ (. InCondition= true;.)

Operator<out elseop> (. op.elseOperator.Add(elseop);.)

}"}"]

(. op.endOperator = (NothingOperator)gotoEnd.oper; tab.operators.AddLast(op.endOperator); InCondition= false;.).

/\*---------------------------------------------------------------------\*/

CycleOperator<out CycleOperator op> (. Kind type; Expression expr; Operator oper; .)

= (. op=null;.)

"WHILE"

"(" Expression<out expr, out type> ")" (. if (type != Kind.Boolean) {SemErr("Expected boolean");ConsoleKeyInfo key = Console.ReadKey();}

op = new CycleOperator(expr); tab.operators.AddLast(op); .)

"{"{ (. InCondition = true; .)

Operator<out oper> (. op.isEmpty=false; .)

}"}"

(. JumpOperator gotoBegin = new JumpOperator(op);

tab.operators.AddLast(gotoBegin);

NothingOperator endOperator = new NothingOperator();

tab.operators.AddLast(endOperator);

op.endOperator = endOperator; InCondition= false;.).

/\*---------------------------------------------------------------------\*/

END SET.

## Приложение В. Классы

**Файл Set.cs**

using Program;

using System;

using System.IO;

using System.Collections.Generic;

namespace Set {

class Set {

public static void Main(string[] arg) {

ConsoleKeyInfo key;

if (arg.Length > 0) {

Scanner scanner = new Scanner(arg[0]);

Parser parser = new Parser(scanner);

parser.tab = new SymbolTable(parser);

parser.gen = new CodeGenerator();

//вывод текста программы

string[] readText = File.ReadAllLines(arg[0]);

foreach (string s in readText) {

Console.WriteLine(s);

}

Console.WriteLine("-----------");

parser.Parse();

//init

foreach (Operator op in parser.tab.operators) {

if (op is JumpOperator) {

string label = ((JumpOperator)op).labelName;

if (label == null) continue;

foreach (Operator labeledOp in parser.tab.operators) {

if (labeledOp.label == label) {

((JumpOperator)op).oper = labeledOp;

break;

}

}

}

}

// end init

try {

LinkedListNode<Operator> node;

for (node = parser.tab.operators.First; node != null; node = node.Next) {

if (node.Value is JumpOperator) {

Operator link = ((JumpOperator)node.Value).oper;

node = parser.tab.operators.Find(link);

}

if (node.Value is IfOperator) {

IfOperator oper = (IfOperator)node.Value;

if (oper.IsTrue()) {

if (oper.ifOperator.Count > 0) node = parser.tab.operators.Find(oper.ifOperator[0]);

else Console.WriteLine("No operators in the If-construction");

}

else {

if (oper.elseOperator.Count > 0) node = parser.tab.operators.Find(oper.elseOperator[0]);

else Console.WriteLine("No operators in the Else-construction");

}

}

if (node.Value is CycleOperator) {

CycleOperator oper = (CycleOperator)node.Value;

if (oper.isEmpty) {

Console.WriteLine("No operators in the cycle"); break;

}

if (!oper.IsTrue()) {

node = parser.tab.operators.Find(oper.endOperator);

}

}

node.Value.Do();

}

//консоль ждет нажатия

key = Console.ReadKey();

if (parser.errors.count == 0) {

}

}

catch (Exception ex) { Console.WriteLine("-- "+ex.Message.ToString()); }

}

else { Console.WriteLine("-- No source file specified"); }

key = Console.ReadKey();

}

}

}

**SymTab.cs**

using System;

using System.Collections;

using System.Collections.Generic;

namespace Program {

public enum Op {

ADD, SUB, MUL, DIV, EQU, NEQU, LSS, GTR, LEQ, GEQ, NEG,

LOAD, LOADG, STO, STOG, CONST, AND, OR, UNION, CROS, RES, DIFF,

CALL, RET, ENTER, LEAVE, JMP, FJMP, READ, WRITE, ERR

}

public class Obj : Value { // объект, описывающий объявленное имя

public string name;

public int type;

public Obj next; // to next object in same scope

public int kind; // var, proc, scope

public Object value;

public int adr; // address in memory or start of proc

public int level; // nesting level; 0=global, 1=local

public Obj locals; // scopes: to locally declared objects

public int nextAdr; // scopes: next free address in this scope

public override Object GetValue() {

return value;

}

}

public class Set{

public List<Object> list = new List<Object>();

public Kind kind;

public Set() {

}

public void Add(Kind k, Object obj) {

kind = k;

list.Add(obj);

}

}

public class SymbolTable {

public static bool IsNumeric(int t) {

return t == int\_ || t == byte\_ || t == double\_;

}

const int

undef\_ = 0, int\_ = 1, bool\_ = 2, byte\_ = 3, double\_ = 5, string\_=6, label\_=7,

SetOfint = 8, SetOfbool = 9, SetOfbyte = 10, SetOfdouble = 12, SetOfstring = 13;

const int

var\_ = 0, proc\_ = 1, scope = 2, const\_ = 3;

public int operSum = 0; //Общее количество выполненных операторов

public int curLevel; // Уровень вложенности of current scope

public Obj undefObj; // объект-узел для ошибочных символов

public Obj topScope; // верхняя область процедуры

public Hashtable labels;

public List<string> labelsOp = new List<string>();

public Parser parser;

public LinkedList<Operator> operators = new LinkedList<Operator>();

public LinkedList<HashSet<Object>> numericSet = new LinkedList<HashSet<Object>>();

public SymbolTable(Parser parser) {

this.parser = parser;

List<Operator> Operations = new List<Operator>();

labels = new Hashtable();

topScope = null;

curLevel = -1;

undefObj = new Obj();

undefObj.name = "undef\_"; undefObj.type = undef\_; undefObj.kind = var\_;

undefObj.adr = 0; undefObj.level = 0; undefObj.next = null;

}

public Obj NewObj(string name, int kind, int type) {

Obj p, last, obj = new Obj();

obj.name = name; obj.kind = kind; obj.type = type;

obj.value = Process.GetValue(type);

obj.level = curLevel;

p = topScope.locals; last = null;

while (p != null) {

if (p.name == name) parser.SemErr("name declared twice");

last = p; p = p.next;

}

if (last == null) topScope.locals = obj; else last.next = obj;

if (kind == var\_) obj.adr = topScope.nextAdr++;

return obj;

}

public void NewLabel(string labelName, Token labelOperator) {

if (!(labels.Contains(labelName))) labels.Add(labelName,labelOperator);

}

public void NewNumericSet(string setName, int type, Value value) {

if (type == SetOfint) {

List <int> s = new List<int>();

s.Add(Convert.ToInt32(value));

}

else if (type == SetOfbyte) {

List<byte> s = new List<byte>();

s.Add(Convert.ToByte(value));

}

else if (type == SetOfdouble) {

List<double> s = new List<double>();

s.Add(Convert.ToDouble(value));

}

else if (type == SetOfstring) {

List<string> s = new List<string>();

s.Add(Convert.ToString(value));

}

}

public Token gotoRealization(string labelName) {

Token t=null;

if (labels.Contains(labelName)) t = (Token) labels[labelName];

return t;

}

public Object Oper(int type1, int type2, Object value1, Object value2, Op op ) {

Object value=0; int type=0;

int val1Int=0; byte val1Byte=0; double val1Double=0;

int val2Int=0; byte val2Byte=0; double val2Double=0;

string val1String=""; bool val1Bool=false;

string val2String=""; bool val2Bool=false;

switch (type1) {

case int\_: val1Int = (int)value1; break;

case byte\_: val1Byte = (byte)value1; break;

case double\_: val1Double = Convert.ToDouble(value1); break;

case string\_: val1String = Convert.ToString(value1); break;

case bool\_: val1Bool = Convert.ToBoolean(value1); break;

}

switch (type2) {

case int\_: val2Int = (int)value2; break;

case byte\_: val2Byte = (byte)value2; break;

case double\_: val2Double = Convert.ToDouble(value2); break;

case string\_: val2String = Convert.ToString(value2); break;

case bool\_: val2Bool = Convert.ToBoolean(value2); break;

}

switch (op) {

case Op.MUL:

switch (type1) {

case int\_:

if (type2 == int\_) { value = val1Int \* val2Int;}

else if (type2 == byte\_) { value = val1Int \* val2Byte;}

else if (type2 == double\_) { value = val1Int \* val2Double;}

break;

case byte\_:

if (type2 == int\_) { value = val1Byte \* val2Int; }

else if (type2 == byte\_) { value = val1Byte \* val2Byte; }

else if (type2 == double\_) { value = val1Byte \* val2Double; }

break;

case double\_:

if (type2 == int\_) { value = val1Double \* val2Int; }

else if (type2 == byte\_) { value = val1Double \* val2Byte; }

else if (type2 == double\_) { value = val1Double \* val2Double; }

break;

}

break;

case Op.DIV:

switch (type1) {

case int\_:

if (type2 == int\_) { value = (double) val1Int / (double)val2Int; }

else if (type2 == byte\_) { value = (double)val1Int / (double)val2Byte; }

else if (type2 == double\_) { value = (double) val1Int / (double)val2Double; }

break;

case byte\_:

if (type2 == int\_) { value = (double) val1Byte / (double) val2Int; }

else if (type2 == byte\_) { value = (double) val1Byte / (double) val2Byte; }

else if (type2 == double\_) { value = (double) val1Byte / (double) val2Double; }

break;

case double\_:

if (type2 == int\_) { value = (double) val1Double / (double) val2Int; }

else if (type2 == byte\_) { value = (double) val1Double / (double) val2Byte; }

else if (type2 == double\_) { value = (double) val1Double / (double) val2Double; }

break;

}

break;

case Op.UNION:

break;

case Op.CROS:

break;

case Op.RES:

break;

case Op.DIFF:

break;

case Op.ADD:

switch (type1) {

case int\_:

if (type2 == int\_) { value = val1Int + val2Int; }

else if (type2 == byte\_) { value = val1Int + val2Byte;}

else if (type2 == double\_) { value = val1Int + val2Double;}

break;

case byte\_:

if (type2 == int\_) { value = val1Byte + val2Int; }

else if (type2 == byte\_) { value = (byte) (val1Byte + val2Byte); }

else if (type2 == double\_) { value = val1Byte + val2Double;}

break;

case double\_:

if (type2 == int\_) { value = val1Double + val2Int; }

else if (type2 == byte\_) { value = val1Double + val2Byte; }

else if (type2 == double\_) { value = val1Double + val2Double; }

break;

}

break;

case Op.SUB:

switch (type1) {

case int\_:

if (type2 == int\_) { value = val1Int - val2Int; }

else if (type2 == byte\_) { value = val1Int - val2Byte; }

else if (type2 == double\_) { value = val1Int - val2Double; }

break;

case byte\_:

if (type2 == int\_) { value = val1Byte - val2Int; }

else if (type2 == byte\_) { value = (byte)(val1Byte - val2Byte); }

else if (type2 == double\_) { value = val1Byte - val2Double; }

break;

case double\_:

if (type2 == int\_) { value = val1Double - val2Int; }

else if (type2 == byte\_) { value = val1Double - val2Byte; }

else if (type2 == double\_) { value = val1Double - val2Double; }

break;

}

break;

case Op.EQU:

if ((type1 == int\_) && (type2 == int\_)) value = (val1Int == val2Int);

if ((type1 == byte\_) && (type2 == byte\_)) value = (val1Byte == val2Byte);

if ((type1 == double\_) && (type2 == double\_)) value = (val1Double == val2Double);

if ((type1 == bool\_) && (type2 == bool\_)) value = (val1Bool == val2Bool);

if ((type1 == string\_) && (type2 == string\_)) value = String.Equals(val1String, val2String);

break;

case Op.NEQU://!=

if ((type1 == int\_) && (type2 == int\_)) value = (val1Int != val2Int);

if ((type1 == byte\_) && (type2 == byte\_)) value = (val1Byte != val2Byte);

if ((type1 == double\_) && (type2 == double\_)) value = (val1Double != val2Double);

if ((type1 == bool\_) && (type2 == bool\_)) value = (val1Bool != val2Bool);

if ((type1 == string\_) && (type2 == string\_)) value = !(String.Equals(val1String, val2String));

break;

case Op.LSS://<

if ((type1 == int\_) && (type2 == int\_)) value = (val1Int < val2Int);

if ((type1 == byte\_) && (type2 == byte\_)) value = (val1Byte < val2Byte);

if ((type1 == double\_) && (type2 == double\_)) value = (val1Double < val2Double);

break;

case Op.GTR://>

if ((type1 == int\_) && (type2 == int\_)) value = (val1Int > val2Int);

if ((type1 == byte\_) && (type2 == byte\_)) value = (val1Byte > val2Byte);

if ((type1 == double\_) && (type2 == double\_)) value = (val1Double > val2Double);

break;

case Op.LEQ://<=

if ((type1 == int\_) && (type2 == int\_)) value = (val1Int <= val2Int);

if ((type1 == byte\_) && (type2 == byte\_)) value = (val1Byte <= val2Byte);

if ((type1 == double\_) && (type2 == double\_)) value = (val1Double <= val2Double);

break;

case Op.GEQ://>=

if ((type1 == int\_) && (type2 == int\_)) value = (val1Int >= val2Int);

if ((type1 == byte\_) && (type2 == byte\_)) value = (val1Byte >= val2Byte);

if ((type1 == double\_) && (type2 == double\_)) value = (val1Double >= val2Double);

break;

case Op.AND:

if ((type1 == bool\_) && (type2 == bool\_)) value = (val1Bool && val2Bool);

break;

case Op.OR:

if ((type1 == bool\_) && (type2 == bool\_)) value = (val1Bool || val2Bool);

break;

case Op.ERR:

Console.WriteLine("Error operation");

break;

}

return value;

}

public Obj Find (string name) {

Obj obj, scope;

scope = topScope;

while (scope != null) { //для всех открытых областей

obj = scope.locals;

while (obj != null) { // для всех объектов в этой области

if (obj.name == name) return obj;

obj = obj.next;

}

scope = scope.next;

}

parser.SemErr(name + " is undeclared");

return undefObj;

}

public bool FindLabel(string name) {

bool find = false;

foreach (string item in labelsOp) {

if (item.Contains(name))

return true;

}

return find;

}

// открыть новую область действия и сделать ее текущей областью (topScope)

public void OpenScope() {

Obj scop = new Obj();

scop.name = ""; scop.kind = scope;

scop.locals = null; scop.nextAdr = 0;

scop.next = topScope; topScope = scop;

curLevel++;

}

// закрыть текущую область действия

public void CloseScope() {

topScope = topScope.next; curLevel--;

}

} // end SymbolTable

} // end namespace

**Файл Operator.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Program {

public abstract class Operator {

public const int // types

undef\_ = 0, int\_ = 1, bool\_ = 2, byte\_ = 3, double\_ = 5, string\_ = 6, label\_ = 7,

SetOfint = 8, SetOfbool = 9, SetOfbyte = 10, SetOfreal = 11, SetOfdouble = 12, SetOfstring = 13;

public string label;

public abstract void Do();

}

public class NothingOperator : Operator {

public NothingOperator() { }

public override void Do() { }

}

public enum Kind { Numeric = 0, String = 1, Boolean = 2, Set = 3, Undefined = 4, Doubles=5, Ints=6, Bytes=7, SetOfint=8, SetOfbyte=9, SetOfdouble = 10, SetOfstring=11 }

public class Process {

public static Object GetValue(int type) {

switch (type) {

case Operator.undef\_: return null;

case Operator.int\_: return 0;

case Operator.bool\_: return false;

case Operator.byte\_: return (byte)0;

case Operator.double\_: return 0.0;

case Operator.string\_: return "";

case Operator.label\_: return "";

case Operator.SetOfbool:

case Operator.SetOfbyte:

case Operator.SetOfint:

case Operator.SetOfreal:

case Operator.SetOfstring:

case Operator.SetOfdouble:

return new Set();

default: return null;

}

}

public static Kind KindOf(Object val) {

if (val is int || val is double || val is byte) return Kind.Numeric;

else if (val is string) return Kind.String;

else if (val is bool) return Kind.Boolean;

else return Kind.Undefined;

}

public static Object Calculate(Object v1, Op op, Object v2) {

Kind kind = Process.KindOf(v1);

if (kind == Kind.Undefined) { Set s = (Set) v1; kind = s.kind; }

Kind kind2 = Process.KindOf(v2);

if (kind2 == Kind.Undefined) { Set s = (Set)v2; kind2 = s.kind; }

if (kind != kind2) {

Console.WriteLine("Errors in types: 'types uncompabitle'");

return null;

}

if (kind == Kind.Numeric) {

if (v1 is double || v2 is double) {

switch (op) {

case Op.ADD: return (double)v1 + (double)v2;

case Op.SUB: return (double)v1 - (double)v2;

case Op.DIV: return (double)v1 / (double)v2;

case Op.MUL: return (double)v1 \* (double)v2;

case Op.EQU: return (double)v1 == (double)v2;

case Op.NEQU: return (double)v1 != (double)v2;

case Op.GTR: return (double)v1 > (double)v2;

case Op.LSS: return (double)v1 < (double)v2;

case Op.GEQ: return (double)v1 >= (double)v2;

case Op.LEQ: return (double)v1 <= (double)v2;

}

} else {

switch (op) {

case Op.ADD: return Convert.ToInt32(v1) + Convert.ToInt32(v2);

case Op.SUB: return Convert.ToInt32(v1) - Convert.ToInt32(v2);

case Op.DIV: return Convert.ToInt32(v1) / Convert.ToInt32(v2);

case Op.MUL: return Convert.ToInt32(v1) \* Convert.ToInt32(v2);

case Op.EQU: return Convert.ToInt32(v1) == Convert.ToInt32(v2);

case Op.NEQU: return Convert.ToInt32(v1) != Convert.ToInt32(v2);

case Op.GTR: return Convert.ToInt32(v1) > Convert.ToInt32(v2);

case Op.LSS: return Convert.ToInt32(v1) < Convert.ToInt32(v2);

case Op.GEQ: return Convert.ToInt32(v1) >= Convert.ToInt32(v2);

case Op.LEQ: return Convert.ToInt32(v1) <= Convert.ToInt32(v2);

}

}

} else if (kind == Kind.Boolean) {

switch (op) {

case Op.EQU: return (bool)v1 == (bool)v2;

case Op.NEQU: return (bool)v1 != (bool)v2;

case Op.AND: return (bool)v1 && (bool)v2;

case Op.OR: return (bool)v1 || (bool)v2;

}

} else if (kind == Kind.String) {

switch (op) {

case Op.EQU: return (string)v1 == (string)v2;

case Op.NEQU: return (string)v1 != (string)v2;

}

} else if ((kind == Kind.SetOfbyte) || (kind == Kind.SetOfdouble) || (kind == Kind.SetOfint) || (kind == Kind.SetOfstring)) {

Set s1 = (Set)v1;

Set s2 = (Set)v2;

Set res = new Set();

if (kind == Kind.SetOfint) {

switch (op) {

case Op.UNION: {

List<int> int1 = new List<int>();

List<int> int2 = new List<int>();

foreach (var ob in s1.list) {

int a = (int)(((ConstValue)ob).GetValue());

int1.Add(a);

}

foreach (var ob in s2.list) {

int a = (int)(((ConstValue)ob).GetValue());

int2.Add(a);

}

//--

List<int> res1 = new List<int>();

res1.AddRange(int1);

foreach (int ob in int2) {

if (!res1.Contains(ob)) res1.Add(ob);

}

//--

foreach (int ob in res1) {

res.list.Add(new ConstValue(ob));

}

return (res);

}

case Op.CROS: {

List<int> int1 = new List<int>();

List<int> int2 = new List<int>();

foreach (var ob in s1.list) {

int a = (int)(((ConstValue)ob).GetValue());

int1.Add(a);

}

foreach (var ob in s2.list) {

int a = (int)(((ConstValue)ob).GetValue());

int2.Add(a);

}

foreach (int ob1 in int1) {

foreach (int ob2 in int2) {

if (ob1 == ob2) res.list.Add(new ConstValue(ob2));

}

}

return (res);

}

case Op.RES: {

List<int> int1 = new List<int>();

List<int> int2 = new List<int>();

foreach (var ob in s1.list) {

int a = (int)(((ConstValue)ob).GetValue());

int1.Add(a);

}

foreach (var ob in s2.list) {

int a = (int)(((ConstValue)ob).GetValue());

int2.Add(a);

}

List<int> res1 = new List<int>();

res1.AddRange(int1);

foreach (int ob1 in int1) {

foreach (int ob2 in int2) {

if (ob1 == ob2) res1.Remove(ob2);

}

}

foreach (int ob in res1) {

res.list.Add(new ConstValue(ob));

}

return (res);

}

case Op.DIFF: {

List<int> int1 = new List<int>();

List<int> int2 = new List<int>();

foreach (var ob in s1.list) {

int a = (int)(((ConstValue)ob).GetValue());

int1.Add(a);

}

foreach (var ob in s2.list) {

int a = (int)(((ConstValue)ob).GetValue());

int2.Add(a);

}

//Объединение

List<int> res1 = new List<int>();

res1.AddRange(int1);

foreach (int ob in int2) {

if (!res1.Contains(ob)) res1.Add(ob);

}

//пересечение

List<int> res2 = new List<int>();

foreach (int ob1 in int1) {

foreach (int ob2 in int2) {

if (ob1 == ob2) res2.Add(ob2);

}

}

//Вычитание из объединения пересечения

foreach (int ob2 in res2) {

res1.Remove(ob2);

}

//Обратное приведение типов

foreach (int ob in res1) {

res.list.Add(new ConstValue(ob));

}

return (res);

}

}

}

else if (kind == Kind.SetOfstring) {

switch (op) {

case Op.UNION: {

List<string> int1 = new List<string>();

List<string> int2 = new List<string>();

foreach (var ob in s1.list) {

string a = (string)(((ConstValue)ob).GetValue());

int1.Add(a);

}

foreach (var ob in s2.list) {

string a = (string)(((ConstValue)ob).GetValue());

int2.Add(a);

}

//--

List<string> res1 = new List<string>();

res1.AddRange(int1);

foreach (string ob in int2) {

if (!res1.Contains(ob)) res1.Add(ob);

}

//--

foreach (string ob in res1) {

res.list.Add(new ConstValue(ob));

}

return (res);

}

case Op.CROS: {

List<string> int1 = new List<string>();

List<string> int2 = new List<string>();

foreach (var ob in s1.list) {

string a = (string)(((ConstValue)ob).GetValue());

int1.Add(a);

}

foreach (var ob in s2.list) {

string a = (string)(((ConstValue)ob).GetValue());

int2.Add(a);

}

foreach (string ob1 in int1) {

foreach (string ob2 in int2) {

if (ob1 == ob2) res.list.Add(new ConstValue(ob2));

}

}

return (res);

}

case Op.RES: {

List<string> int1 = new List<string>();

List<string> int2 = new List<string>();

foreach (var ob in s1.list) {

string a = (string)(((ConstValue)ob).GetValue());

int1.Add(a);

}

foreach (var ob in s2.list) {

string a = (string)(((ConstValue)ob).GetValue());

int2.Add(a);

}

List<string> res1 = new List<string>();

res1.AddRange(int1);

foreach (string ob1 in int1) {

foreach (string ob2 in int2) {

if (ob1 == ob2) res1.Remove(ob2);

}

}

foreach (string ob in res1) {

res.list.Add(new ConstValue(ob));

}

return (res);

}

case Op.DIFF: {

List<string> int1 = new List<string>();

List<string> int2 = new List<string>();

foreach (var ob in s1.list) {

string a = (string)(((ConstValue)ob).GetValue());

int1.Add(a);

}

foreach (var ob in s2.list) {

string a = (string)(((ConstValue)ob).GetValue());

int2.Add(a);

}

//Объединение

List<string> res1 = new List<string>();

res1.AddRange(int1);

foreach (string ob in int2) {

if (!res1.Contains(ob)) res1.Add(ob);

}

//пересечение

List<string> res2 = new List<string>();

foreach (string ob1 in int1) {

foreach (string ob2 in int2) {

if (ob1 == ob2) res2.Add(ob2);

}

}

//Вычитание из объединения пересечения

foreach (string ob2 in res2) {

res1.Remove(ob2);

}

//Обратное приведение типов

foreach (string ob in res1) {

res.list.Add(new ConstValue(ob));

}

return (res);

}

}

}//else if строки

else if (kind == Kind.SetOfdouble) {

switch (op) {

case Op.UNION: {

List<double> double1 = new List<double>();

List<double> double2 = new List<double>();

foreach (var ob in s1.list) {

double a = (double)(((ConstValue)ob).GetValue());

double1.Add(a);

}

foreach (var ob in s2.list) {

double a = (double)(((ConstValue)ob).GetValue());

double2.Add(a);

}

//--

List<double> res1 = new List<double>();

res1.AddRange(double1);

foreach (double ob in double2) {

if (!res1.Contains(ob)) res1.Add(ob);

}

//--

foreach (double ob in res1) {

res.list.Add(new ConstValue(ob));

}

return (res);

}

case Op.CROS: {

List<double> double1 = new List<double>();

List<double> double2 = new List<double>();

foreach (var ob in s1.list) {

double a = (double)(((ConstValue)ob).GetValue());

double1.Add(a);

}

foreach (var ob in s2.list) {

double a = (double)(((ConstValue)ob).GetValue());

double2.Add(a);

}

foreach (double ob1 in double1) {

foreach (double ob2 in double2) {

if (ob1 == ob2) res.list.Add(new ConstValue(ob2));

}

}

return (res);

}

case Op.RES: {

List<double> double1 = new List<double>();

List<double> double2 = new List<double>();

foreach (var ob in s1.list) {

double a = (double)(((ConstValue)ob).GetValue());

double1.Add(a);

}

foreach (var ob in s2.list) {

double a = (double)(((ConstValue)ob).GetValue());

double2.Add(a);

}

List<double> res1 = new List<double>();

res1.AddRange(double1);

foreach (double ob1 in double1) {

foreach (double ob2 in double2) {

if (ob1 == ob2) res1.Remove(ob2);

}

}

foreach (double ob in res1) {

res.list.Add(new ConstValue(ob));

}

return (res);

}

case Op.DIFF: {

List<double> double1 = new List<double>();

List<double> double2 = new List<double>();

foreach (var ob in s1.list) {

double a = (double)(((ConstValue)ob).GetValue());

double1.Add(a);

}

foreach (var ob in s2.list) {

double a = (double)(((ConstValue)ob).GetValue());

double2.Add(a);

}

//Объединение

List<double> res1 = new List<double>();

res1.AddRange(double1);

foreach (double ob in double2) {

if (!res1.Contains(ob)) res1.Add(ob);

}

//пересечение

List<double> res2 = new List<double>();

foreach (double ob1 in double1) {

foreach (double ob2 in double2) {

if (ob1 == ob2) res2.Add(ob2);

}

}

//Вычитание из объединения пересечения

foreach (double ob2 in res2) {

res1.Remove(ob2);

}

//Обратное приведение типов

foreach (double ob in res1) {

res.list.Add(new ConstValue(ob));

}

return (res);

}

}

}//конец дабл

}

Console.WriteLine("Errors in types: 'operation and type uncompabitle'");

return null;

}

}

public abstract class Value {

public abstract Object GetValue();

}

public class ConstValue : Value {

Object value;

public ConstValue(Object val) {

value = val;

}

public override Object GetValue() {

return value;

}

}

public class Expression : Value {

public Value operand;

public List<Operation> operations;

public Expression(Value val) {

operand = val;

operations = new List<Operation>();

}

public bool IsEmpty() {

return operations.Count == 0;

}

public void Add(Op op, Value val) {

operations.Add(new Operation(op, val));

}

public override Object GetValue() {

Object value = operand.GetValue();

foreach(Operation op in operations) {

Object subVal = op.operand.GetValue();

value = Process.Calculate(value, op.operation, subVal);

}

return value;

}

}

public class Operation {

public Op operation;

public Value operand;

public Operation(Op op, Value val) {

operation = op;

operand = val;

}

}

public class OperAssign:Operator {

public Obj subject;

public Value value;

public OperAssign(Obj s, Value v) {

subject = s;

value = v;

}

public override void Do() {

subject.value = value.GetValue();

}

}

public class IfOperator:Operator {

public Value expression;

public List<Operator> ifOperator;

public List<Operator> elseOperator;

public NothingOperator endOperator;

public IfOperator(Value expression1) {

expression = expression1;

}

public bool IsTrue() {

return ((bool)expression.GetValue() == true);

}

public override void Do() {

}

}

public class CycleOperator : Operator {

public Value expression;

public List<Operator> operators;

public NothingOperator endOperator;

public Boolean isEmpty;

public CycleOperator(Value expression1) {

expression = expression1;

isEmpty = true;

}

public bool IsTrue() {

return ((bool)expression.GetValue() == true);

}

public override void Do() {

}

}

public class InOutOperator : Operator {

public int inOutType; //write/writeln/read/readln

public List<Value> elementToInOut; //то, что будем вводить или выводить

public InOutOperator(int t) {

inOutType = t;

}

public void Add(Value obj) {

elementToInOut.Add(obj);

}

public override void Do() {

if (inOutType == 0 || inOutType == 1) {

string output = "";

foreach (Value obj in elementToInOut) {

Object a = obj.GetValue();

if (obj.GetValue().ToString().Equals("Program.Set")) {

output += "{ ";

Set s = (Set) obj.GetValue();

foreach (Object ob in s.list) {

ConstValue cv = (ConstValue)ob;

output += cv.GetValue().ToString().ToString() + "; ";

}

output += "}";

}

else output += obj.GetValue().ToString() + " ";

}

if (inOutType == 0) Console.Write(output);

else Console.WriteLine(output);

} else {

if (inOutType == 2) {

foreach (Value obj in elementToInOut) {

Obj var = (Obj)obj;

int input = Console.Read();

Kind kind = Process.KindOf(var.value);

switch (kind) {

case Kind.Numeric: var.value = input; break;

case Kind.String: var.value = input.ToString(); break;

default:

Console.WriteLine("Errors in types: 'types uncompabitle'");

break;

}

}

} else {

foreach (Value obj in elementToInOut) {

Obj var = (Obj)obj;

string input = Console.ReadLine();

Kind kind = Process.KindOf(var.value);

switch (kind) {

case Kind.String: var.value = input; break;

case Kind.Numeric:

if (var.value is double) var.value = double.Parse(input, System.Globalization.CultureInfo.InvariantCulture.NumberFormat);

else var.value = int.Parse(input);

break;

default:

Console.WriteLine("Errors in types: 'types uncompabitle'");

break;

}

}

}

}

}

}

public class JumpOperator : Operator {

public string labelName;

public Operator oper;

public JumpOperator(string name) {

labelName = name;

}

public JumpOperator(Operator op) {

oper = op;

}

public override void Do() {

}

}

}