|  |  |
| --- | --- |
| МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ  РОССИЙСКОЙ ФЕДЕРАЦИИ  Федеральное государственное автономное образовательное учреждение высшего образования  «Пермский государственный национальный  исследовательский университет» | |
| *Кафедра математического обеспечения  вычислительных систем* | |
|  | |
| ОТЧЕТ  по дисциплине Формальный грамматики и методы трансляции  «Компилятор для Pascal» | |
| Работу выполнил студент группы *ПМИ-1* *3* курса механико-математического факультета  Шестаков А.А.  18 декабря 2021г. | Принял:  ассистент каф. МОВС  Пономарев Ф.А.  “\_\_\_\_\_” 2021 г. |
| Пермь 2021 | |

Оглавление

[Модуль ввода-вывода 3](#_Toc90787380)

[1. Описание 3](#_Toc90787381)

[2. Проектирование 4](#_Toc90787382)

[3. Реализация 6](#_Toc90787383)

[4. Тестирование 10](#_Toc90787384)

[Лексический анализатор 12](#_Toc90787385)

[1. Описание 12](#_Toc90787386)

[2. Проектирование 13](#_Toc90787387)

[3. Реализация 15](#_Toc90787388)

[4. Тестирование 21](#_Toc90787389)

[Синтаксический и семантический анализ 29](#_Toc90787390)

[1. Описание 29](#_Toc90787391)

[2. Проектирование 30](#_Toc90787392)

[3. Реализация 32](#_Toc90787393)

[4. Тестирование 39](#_Toc90787394)

[Генератор 42](#_Toc90787395)

[1. Описание 42](#_Toc90787396)

[2. Проектирование 43](#_Toc90787397)

[3. Реализация 44](#_Toc90787398)

[4. Тестирование 51](#_Toc90787399)

# Модуль ввода-вывода

## Описание

Необходимо разработать модуль, который будет выдавать литеры из входного потока данных (исходный текст программы на языке Pascal), а также выводить ошибки, которые встретились по ходу анализирования программы.

## Проектирование

Для реализации модуля ввода-вывода, будем разрабатывать класс CIO, который будет содержать следующие поля и методы.

Класс CIO:

Поля:

* Поток ввода,
* Позиция литеры,
* Текст программы.

Методы:

* Получение следующей литеры,
* Получение позиции текущей литеры,
* Получение позиции предыдущей литеры,
* Вывод ошибок.

Текст программы необходимо хранить, чтобы можно было красиво выводить ошибки с указанием их позиции.

Вывод ошибок будет принимать объект класса ErrorManager, в котором будут храниться ошибки, накопленные в ходе анализирования программы.

Разберём ErrorManager, который должен хранить ошибки, но для начала определим класс Error:

Класс Error:

Поля:

* Код ошибки,
* Позиция ошибки,
* Словарь, в котором по коду ошибки, можно получить описание ошибки.

Методы:

* Получение позиции,
* Получение кода ошибки,
* Получение описания ошибки.

Класс ErrorManager:

Поля:

* Список ошибок.

Методы:

* Добавление ошибки,
* Получение ошибки,
* Получение количество ошибок,
* Сортировка ошибок.

Поскольку у нас однопроходный компилятор, то в конце анализа программы, будем запускать вывод ошибок и для удобства, метод получение ошибки будет возвращать первую ошибку в списке и удалять её оттуда.

Сортировка ошибок нужна для того, чтобы отсортировать ошибки по позициям, ведь при семантическом анализе мы можем сначала получить ошибку в 6 символе, и только потом в 3, поэтому было принято решение отсортировать ошибки по позициям перед выводом на консоль.

## Реализация

Начнём с класса CIO:

На Рисунок 1 представлен заголовочный файл класса CIO.

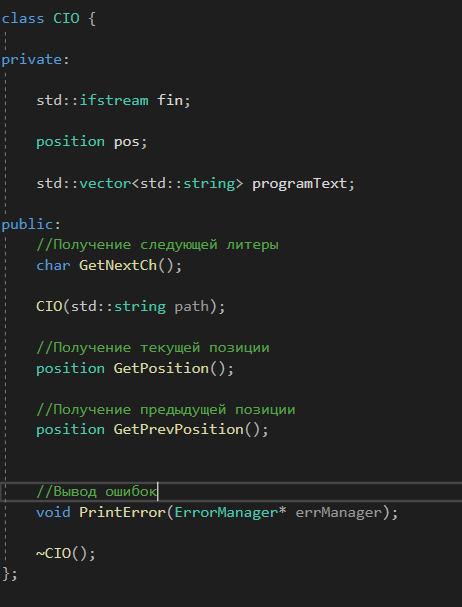


Рисунок 1. Заголовочный файл класса CIO.

На Рисунок 2 представлена реализация получения следующей литеры.

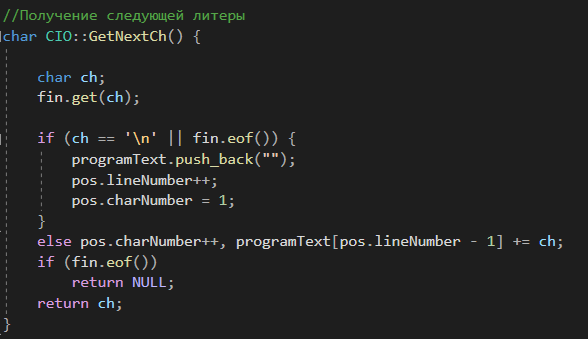
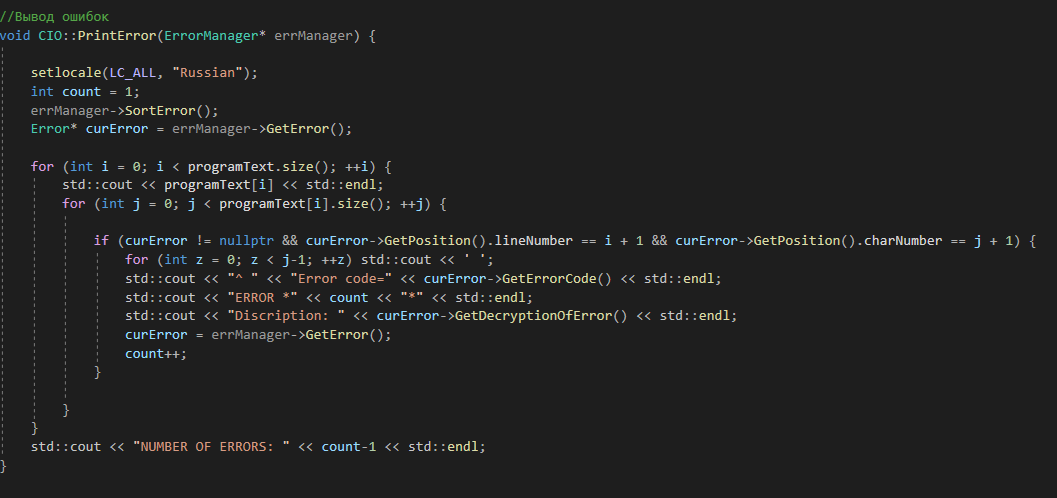


Рисунок 2. Реализация получения следующей литеры.

В этом методе идёт получение следующей литеры из fin, подсчёт позиции и также проверяется на перевод строки и конец файла.

Реализация метода вывода ошибок:



Сначала сортируем ошибки по позиции, дальше выводим строку, пробегаем по каждому символу в этой строке и проверяем есть ли там ошибка, если есть, то выводим ошибку и получаем следующую.

Класс Error:

На Рисунок 3 представлен заголовочный файл класса Error.

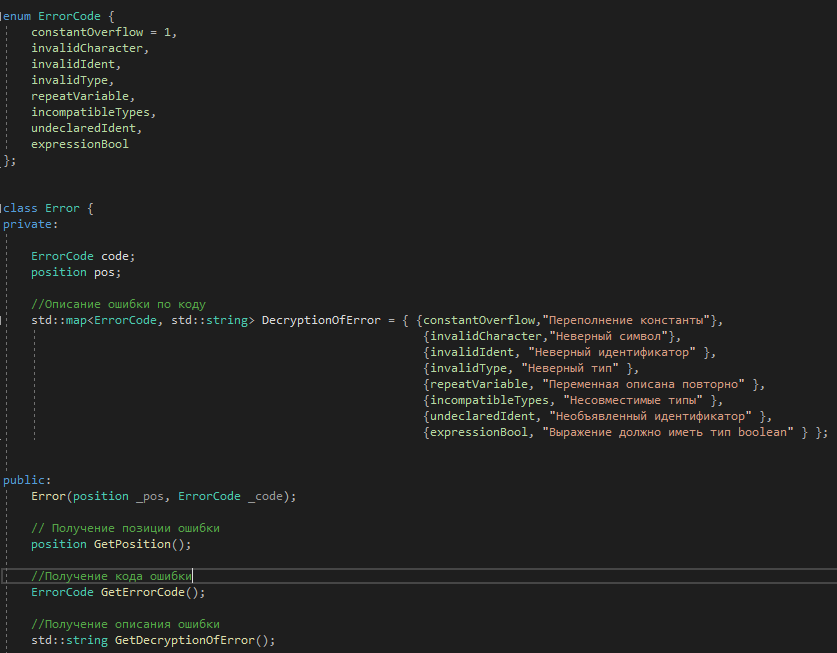


Рисунок 3. Заголовочный файл класса Error.

Класс ErrorManager:

На Рисунок 4 представлен заголовочный файл класса ErrorManager.

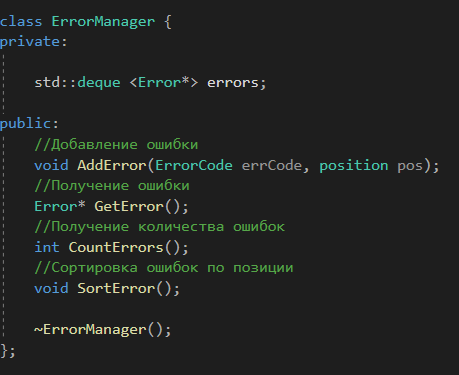
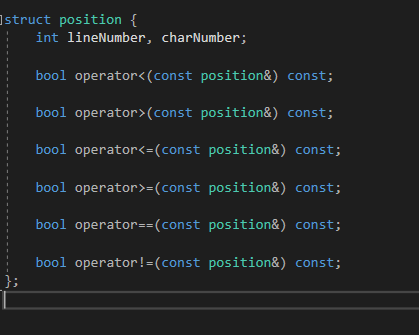


Рисунок 4. Заголовочный файл класса ErrorManager.

Для хранения ошибок была выбрана структура данных deque, чтобы можно было быстро получать первый элемент списка и нормально работала сортировка.

Заголовочный файл Position.



Переопределение операторов необходимо для сортировки ошибок по позициям.

## Тестирование

Поскольку ошибок ещё нет, то протестируем просто вывод текста исходно программы на консоль.

На Рисунок 5 представлен исходный текст программы на языке Pascal.

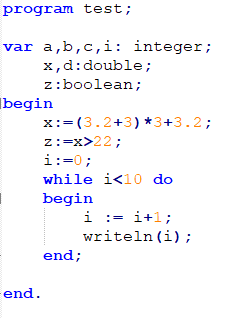


Рисунок 5. Исходный текст программы.

На Рисунок 6 представлен код, который вызывается в функции main.

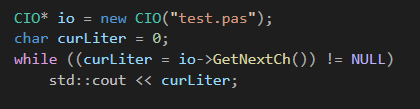


Рисунок 6. Код в main.

На Рисунок 7 представлен результат выполнения программы.

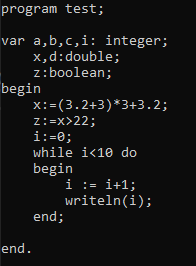


Рисунок 7. Результат выполнения программы.

Как мы видим, программы вывелась корректно.

Вывод ошибок будет рассмотрен в следующих разделах.

# Лексический анализатор

## Описание

Необходимо разработать модуль, который разбивает программу на токены, а также добавляет лексические ошибки в менеджер ошибок.

## Проектирование

Для реализации данного модуля, будем разрабатывать следующие классы CLexer (Лексический анализатор), CToken (Токен), CVariant (Константа).

Рассмотрим класс CLexer:

Класс CLexer:

Поля:

* Текущая литера,
* Модуль ввода-вывода,
* Словарь, в котором по ключевому слову можно получить символ операции,
* Множество ключевых слов,
* Менеджер ошибок.

Методы:

* Проверка на букву,
* Проверка на цифру,
* Проверка слова, на наличие ошибки,
* Проверка числовой константы,
* Добавление неверного идентификатора,
* Проверка на переполнение константы,
* Получение следующего токена.

Сам анализ будет происходить через конструкцию switch, в которой будет проверятся текущая литера, например: если к нам приходит символ «:», то смотрим на следующую литеру, если там «=», то это символ присваивания, иначе это символ для указания типа у описываемых переменных.

Класс CToken:

Поля:

* Словарь, в котором по символу операции, можно получить её строковое описание,
* Лексема,
* Позиция,
* Константа,
* Тип токена,
* Символ операции.

Методы:

* Инициализация словаря с символами операции их описанием,
* Получение строкового названия токена,
* Получение позиции,
* Получение лексемы,
* Получение типа токена,
* Получение символа операции,
* Вывод токена,
* Получение константы.

Методы по получению операции и константы нужны будут в синтаксическом и семантических анализаторах. Получение строкового названия токена нужно для вывода токена на консоль.

Будем рассматривать следующие типы токенов:

1) Идентификатор,

2) Константа,

3) Операция,

4) Специальный.

Специальный токен нужен для того, чтобы при нахождении лексической ошибки в программе, не ломалась логика синтаксического анализатора, то есть этот ошибочный токен нельзя просто пропустить.

Класс CVariant:

В этом классе будет один виртуальный метод для получения типа константы.

CVariant будет виртуальным классом от которого будут наследоваться классы: СIntVariant, CDoubleVariant, CStringVariant.

В каждом из этих классах будет храниться значение константы и переопределение метода для получения типа константы.

## Реализация

Начнём с класса CLexer:

На Рисунок 8 представлен заголовочный файл класса CLexer.

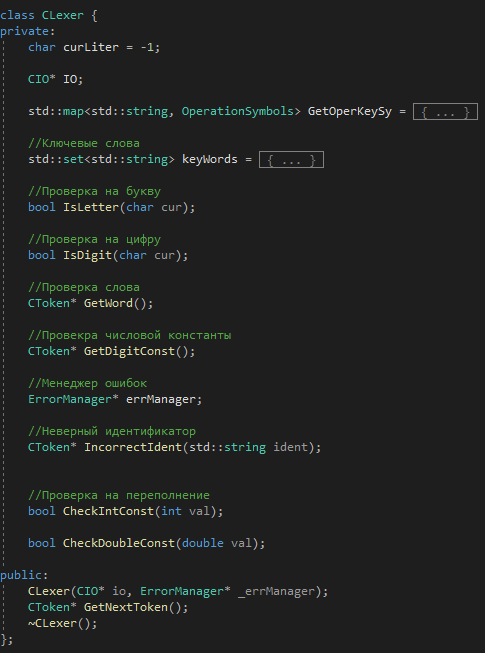


Рисунок 8. Заголовочный файл класса CLexer.

В main будет создаваться экземпляры класса ErrorManager и CIO, которые будут передаваться CLexer. CIO будет служить для получения следующей литеры, а в ErrorManager будем добавлять лексические ошибки.

Рассмотрим часть реализацию по получению следующего токена.

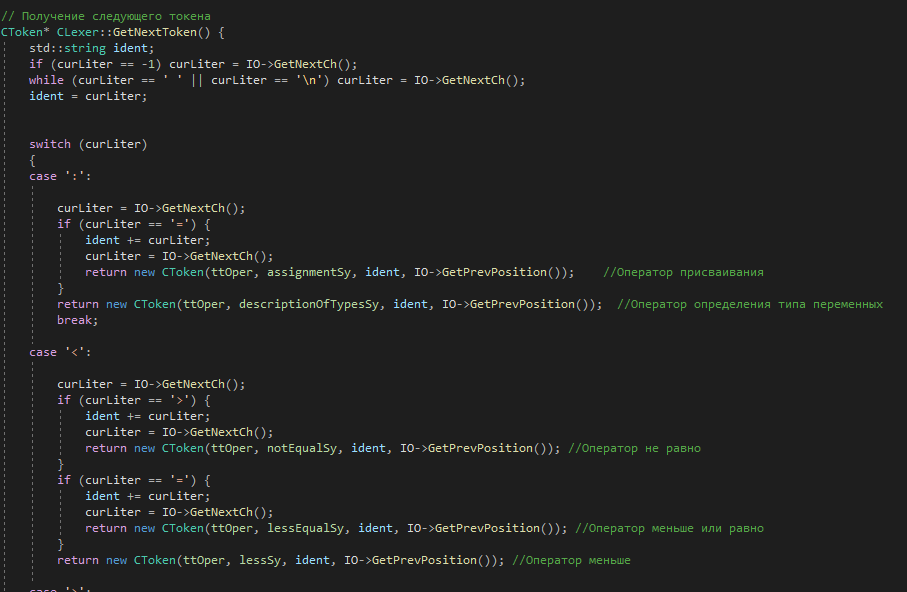


Рисунок 9. Часть реализации метода GetNextToken.

В начале идёт пропуск ненужных символов(пробелы и переводы строки), а дальше идёт анализ токена с использованием следующей литеры.

Также здесь реализована логика пропуска комментариев.

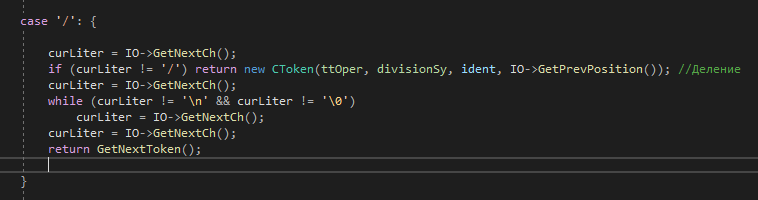


Рисунок 10. Пропуск комментариев.

На примере однострочного комментария, мы пропускаем символы до конца строки и дальше возвращаем следующий токен путём повторного вызова метода GetNextToken.

Если мы прошлись по всему switch и не нашли нужного символа и при этом это не конец файла, то формируем ошибку о недопустимом символе и возвращаем специальный токен.

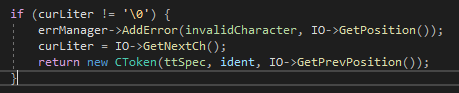


Рисунок 11. Неверный символ.

Аналогичным образом добавляется ошибка и о переполненной константе, и при ошибочном идентификаторе.

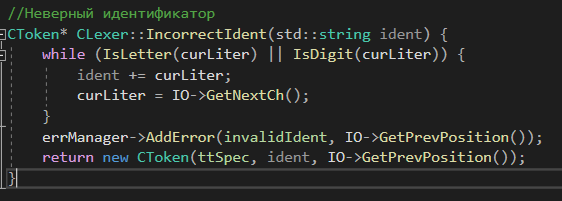


Рисунок 12. Неверный идентификатор.

Мы вызываем данный метод, если при чтении цифр, мы натыкаем на букву, тогда идёт неверный идентификатор, например: 1abc.

Класс CToken:

Заголовочный файл:

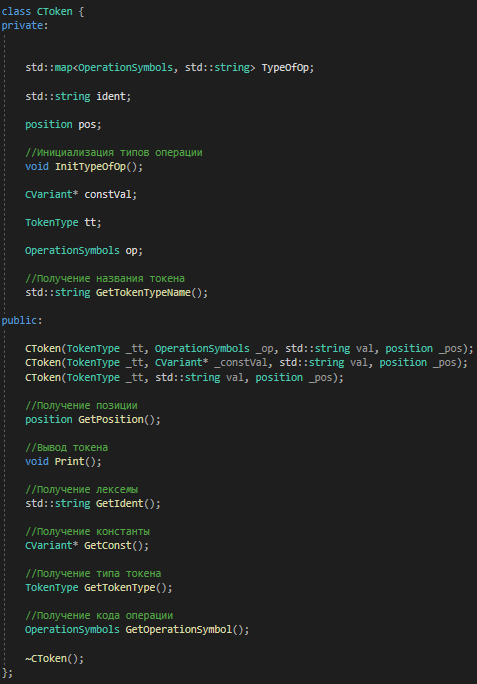


Рисунок 13. Заголовочный файл класса CToken.

Для удобного создания экземпляра данного класса было принято решение реализовать несколько конструкторов, чтобы не передавать все поля.

Ниже приведены все операции:

null,

assignmentSy, //:=

descriptionOfTypesSy, //:

notEqualSy, //<>

lessEqualSy, //<=

lessSy, //<

moreEqualSy,//>=

moreSy,//>

semicolonSy, //;

pointSy,//.

ellipsisSy,//..

sumSy,//+

minusSy,//-

divisionSy,// /

equalSy,// =

mulSy,//\*

openParSy,//(

closeParSy,//)

openIndSy,//[

closeIndSy,//]

commaSy, //,

ifsy,

dosy,

ofsy,

orsy,

insy,

tosy,

endsy,

varsy,

divsy,

andsy,

notsy,

forsy,

modsy,

setsy,

thensy,

elsesy,

casesy,

filesy,

gotosy,

typesy,

withsy,

beginsy,

whilesy,

arraysy,

constsy,

labelsy,

untilsy,

downtosy,

packedsy,

recordsy,

repeatsy,

lengthsy,

writelnsy,

readlnsy,

copysy,

programsy,

fuctionsy,

proceduresy.

Класс CVariant:

Ниже приведен заголовочный файл:

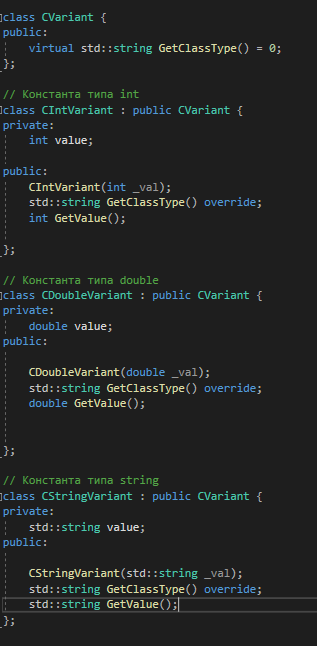


Рисунок 14. Заголовочный файл класса CVariant.

Для описания констант было решение применить наследование.

## Тестирование

Для начала проверим, какие токены выдаёт программа по исходному тексту.

Исходный текст программы:

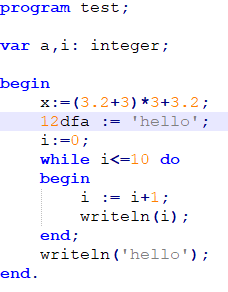


Рисунок 15. Исходный текст программы.

Функция main:

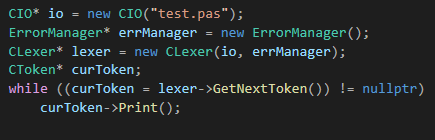


Рисунок 16. Функция main.

Вот вывод программы, так как он очень большой, то просто вставлю текст.

Type: KeyWord

Value: program

Type: Variable

Value: test

Type: Separator

Value: ;

Type: KeyWord

Value: var

Type: Variable

Value: a

Type: Separator

Value: ,

Type: Variable

Value: i

Type: Separator

Value: :

Type: Variable

Value: integer

Type: Separator

Value: ;

Type: KeyWord

Value: begin

Type: Variable

Value: x

Type: Assignment

Value: :=

Type: Separator

Value: (

Type: doubleConst

Value: 3.2

Type: Arithmetic Operators

Value: +

Type: integerConst

Value: 3

Type: Separator

Value: )

Type: Arithmetic Operators

Value: \*

Type: integerConst

Value: 3

Type: Arithmetic Operators

Value: +

Type: doubleConst

Value: 3.2

Type: Separator

Value: ;

Type: Special

Value: 12dfa

Type: Assignment

Value: :=

Type: stringConst

Value: 'hello'

Type: Separator

Value: ;

Type: Variable

Value: i

Type: Assignment

Value: :=

Type: integerConst

Value: 0

Type: Separator

Value: ;

Type: KeyWord

Value: while

Type: Variable

Value: i

Type: Comparison

Value: <=

Type: integerConst

Value: 10

Type: KeyWord

Value: do

Type: KeyWord

Value: begin

Type: Variable

Value: i

Type: Assignment

Value: :=

Type: Variable

Value: i

Type: Arithmetic Operators

Value: +

Type: integerConst

Value: 1

Type: Separator

Value: ;

Type: KeyWord

Value: writeln

Type: Separator

Value: (

Type: Variable

Value: i

Type: Separator

Value: )

Type: Separator

Value: ;

Type: KeyWord

Value: end

Type: Separator

Value: ;

Type: KeyWord

Value: writeln

Type: Separator

Value: (

Type: stringConst

Value: 'hello'

Type: Separator

Value: )

Type: Separator

Value: ;

Type: KeyWord

Value: end

Type: Separator

Value: .

Как мы видим, типы токенов определены верно.

Дальше рассмотрим следующие ошибки:

1. Неверный идентификатор,
2. Неверный символ,
3. Переполнение константы.

Допустим сразу все типы ошибок в исходной программе:

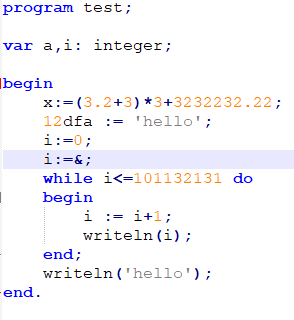


Рисунок 17. Текст исходной программы.

Допущенные ошибки:

1. 3232232.22 переполнение константы,
2. 12dfa неверный идентификатор,
3. & недопустимый символ,
4. 101132131 переполнение константы.

Получаем следующий результат работы компилятора:

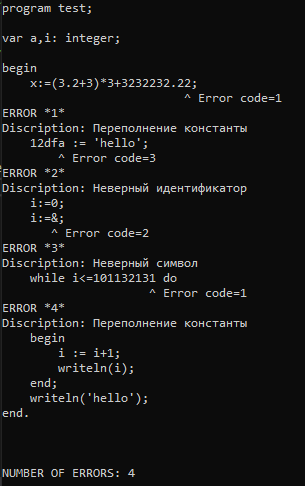


Рисунок 18. Результат работы с лексическими ошибками.

Как мы видим, все ошибки были определены верно.

# Синтаксический и семантический анализ

## Описание

Необходимо разработать модуль, который по исходной программе проводи синтаксический анализ (преобразует в структурированный формат) и семантический анализ (семантические правила языка). А также добавляет семантические ошибки в менеджер ошибок.

## Проектирование

Для реализации данного модуля, будем разрабатывать класс CCompiler и CType.

Класс CCompiler:

Поля:

* Лексический анализатор,
* Текущий токен,
* Менеджер ошибок,
* Словарь с допустимыми типами,
* Допустимые переменные с их типом,
* Индексы переменных (для генератора),
* Словарь для переменных, в которых мы помечаем ругались ли мы уже на эту переменную.

Методы:

* Результат приведения двух типов,
* Проверка на одну из операций или типа токена,
* Проверка операции,
* Блок,
* Раздел переменных,
* Описание однотипных переменных,
* Тип,
* Раздел операторов,
* Составной оператор,
* Оператор,
* Непомеченный оператор,
* Простой оператор,
* Сложный оператор,
* Оператор присваивания,
* Выражение,
* Простое выражение,
* Слагаемое,
* Множитель,
* Выбирающий оператор,
* Оператор цикла,
* Условный оператор,
* Цикл с предусловием,
* Оператор процедуры,
* Параметр,
* Проверка программы.

Синтаксический анализ будет проходить по БНФ для языка Pascal, которые были взяты из книги: «Разработка ПАСКАЛЬ-компилятора» Л.Залогова.

Будем рекурсивно ходить по БНФкам по соответствующим методам.

Рассмотрим абстрактный класс CType.

Класс CType:

Поля:

* Тип.

Методы:

* Виртуальный проверка на приводимость.

В типах также будем использовать наследования, как для констант.

Наследовать будут CIntType, CDoubleType, CStringType, CBoolType.

В которых будет переопределен метод «Приводим к», данный метод будет проверять приводим ли текущий тип к необходимому, который будет передаваться в параметре.

## Реализация

Начнём с класса CType.

Заголовочный файл:

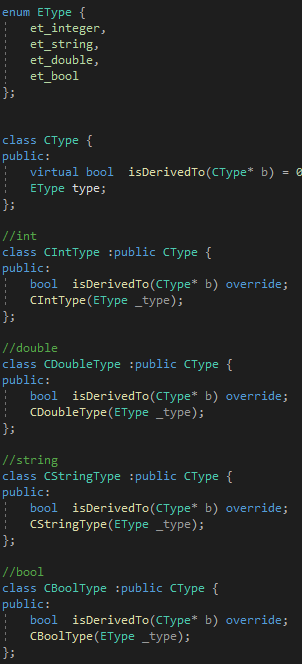


Рисунок 19. Заголовочный файл CType.

Метод isDerivedTo будет возвращать bool приводим ли данный тип к типу параметра b.

Класс CCompiler:

Заголовочный файл:

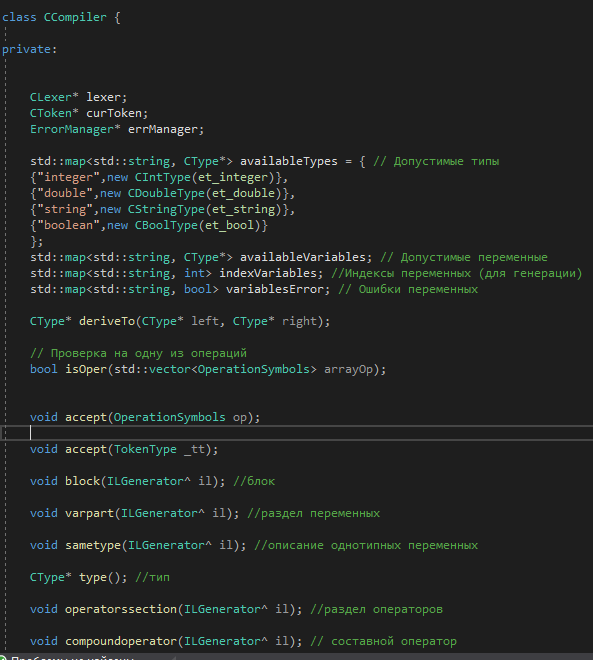


Рисунок 20. Заголовочный файл класса CCompiler.

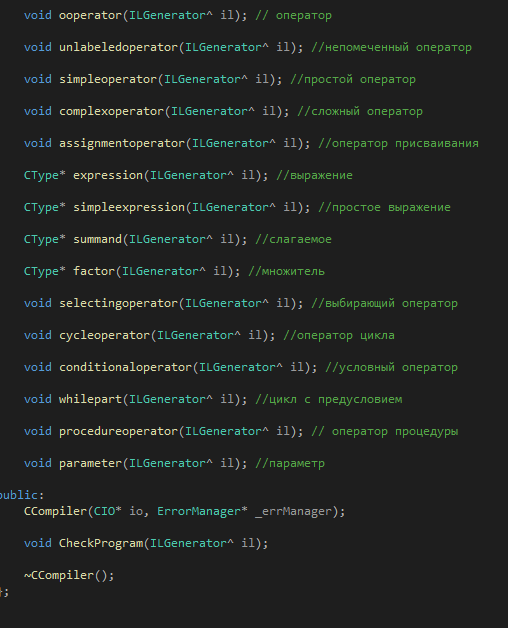


Рисунок 21. Заголовочный файл класса CCompiler.

Все выражения возвращаю CType\* для семантического анализа.

Теперь рассмотрим поподробнее сами методы.

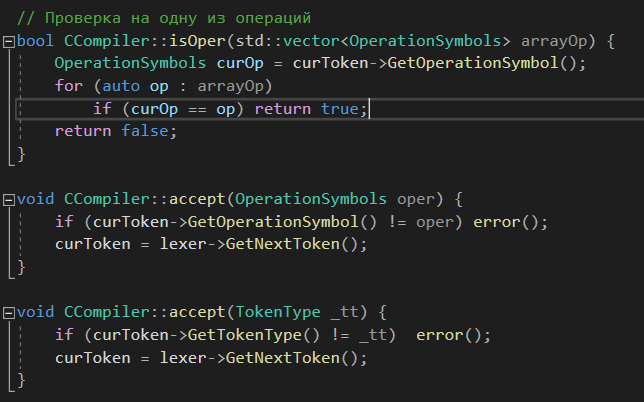


Рисунок 22. Методы для проверки

Выше представлена реализация основных проверок в синтаксическом анализаторе.

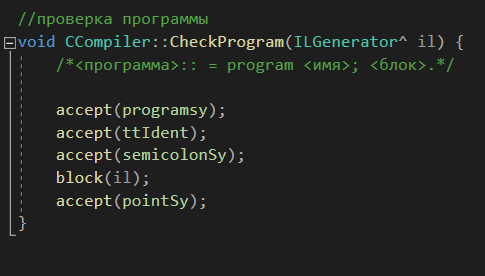


Рисунок 23. Начала синтаксической проверки программы.

Здесь происходят следующие проверки:

1. Проверка на Program,
2. Проверка на идентификатор,
3. Проверка на (;),
4. Уходим в Блок,
5. Проверка на точку (.).

Рассмотрим разбор простого выражения:



Рисунок 24. Разбор простого выражения.

Действия:

1. Проверка на наличие знака,
2. Вычисляем тип левой части выражения (left),
3. Проверяем, есть ли дальше аддитивная операция,
4. Если есть, то вычисляем тип правой части выражения(right),
5. Дальше проверяем на приводимость типов, если не получается их привести, то формируем сообщение об ошибке.
6. Возвращаем тип выражения.

Аналогичный действия проводятся и со слагаемым.

Также во время выполнения происходят следующие проверки:

1)Проверка на необъявленный идентификатор.

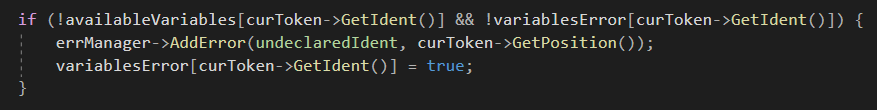


Рисунок 25. Проверка на необъявленный идентификатор.

Проверяем, если этой переменной нет в списке допустимых переменных и на неё ещё не ругались, то заносим ошибку в менеджер ошибок.

2)Проверка на повторно описанную переменную.

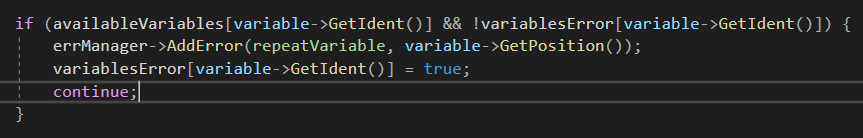


Рисунок 26. Проверка на повторно описанную переменную.

Проверяем, если переменная уже есть в списке доступных и при этом мы на неё не ругались, то заносим ошибку в менеджер ошибок.

Сами переменные заносятся в словарь в блоке описания однотипных переменных, который находится в блоке var.

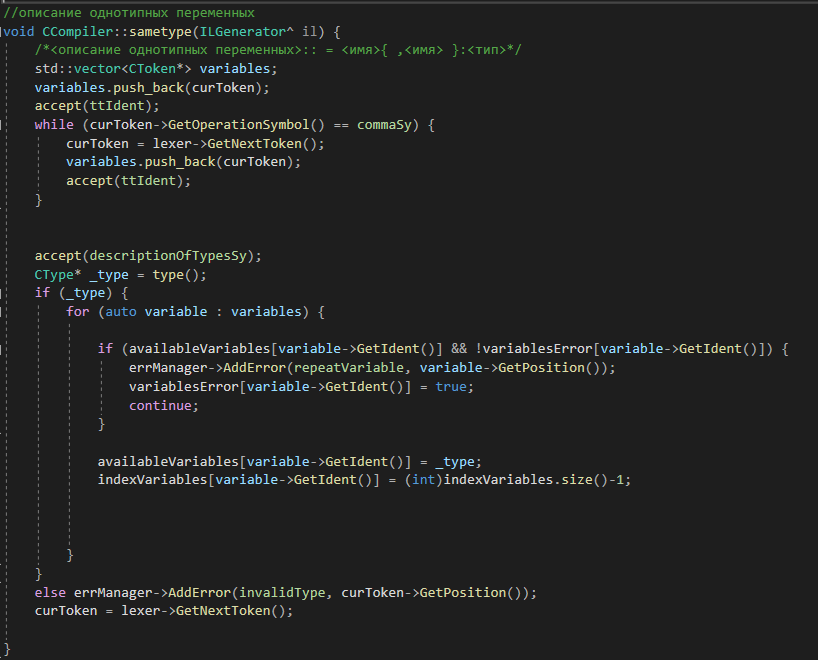


Рисунок 27. Описание однотипных переменных.

Здесь мы формируем список переменных, дальше попадаем на тип и заносим в словарь эти переменные с советующим типом.

В выражении мы спускаемся до самого низкого уровня «множитель», который представляет собой переменную, константу или выражение.



Рисунок 28. Реализация множителя.

Здесь возвращаем тип множителя.

## Тестирование

В семантическом анализе будем рассматривать следующие ошибки:

1. Неверный тип,
2. Переменная описана повторно,
3. Несовместимые типы,
4. Необъявленный идентификатор,
5. Выражение должно иметь тип Boolean.

Для начала проверим на правильной программе.

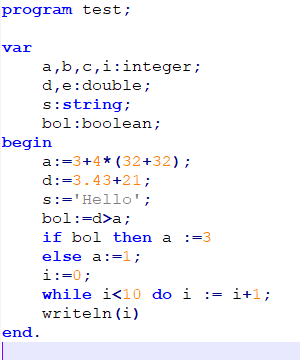


Рисунок 29. Исходный текст программы.

Результат работы программы:

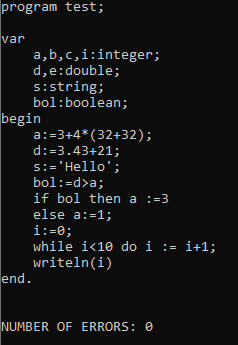


Рисунок 30. Результат работы программы.

Как мы видим, ошибок не найдено.

Теперь сделаем все, вышеописанные ошибки.

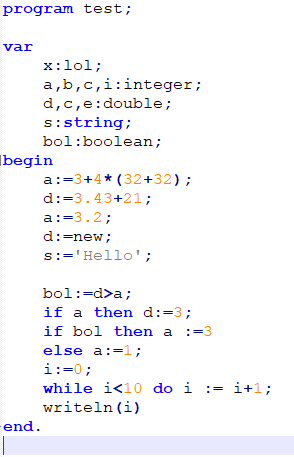


Рисунок 31.Исходная программа.

Допустили следующие ошибки:

1. Типа lol нет,
2. Переменную (с) описали повторно,
3. В переменную (а) типа int положили 3.2(double),
4. Переменная (new) не объявлена,
5. Несоответствие типов d:=new, так как new не объявлена,
6. If а then, тип переменной (a) не boolean.

Результат программы:

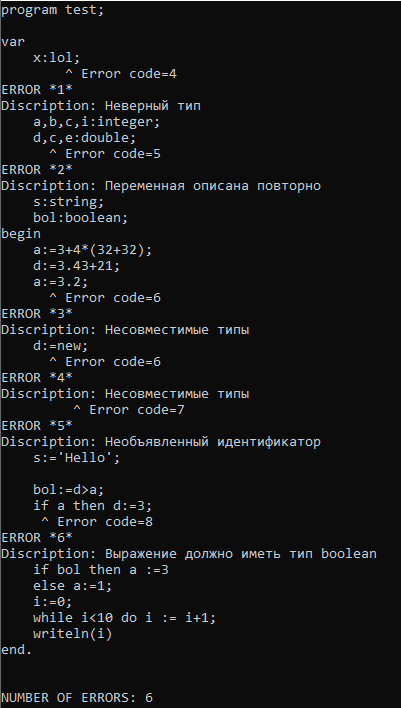


Рисунок 32. Результат программы.

Как мы видим, все ошибки были обнаружены.

# Генератор

## Описание

Необходимо написать модуль, который текст исходной программы переводит в IL код.

## Проектирование

Для реализации генератора, реализуем класс GeneratorMSIL.

Класс GeneratorMSIL.

Поля:

* Компилятор,
* Менеджер ошибок.

Методы:

* Компиляция.

В методе «Компиляция» мы будем подавать компилятору ilGenerator, и во время синтаксического и семантического анализа будем добавлять инструкции на стек.

Менеджер ошибок нужен для того, после анализа посмотреть есть ли ошибки в программе, если есть, то заканчиваем компиляцию.

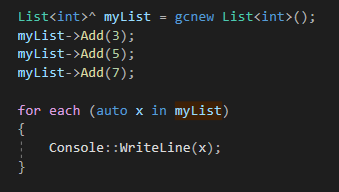
## Реализация

Для начала, так как я разрабатывал на C++, а IL код нужно генерировать через System::Reflection::Emit, а это .NET, было решено использовать CIL, чтобы можно было использовать .NET библиотеки в C++ коде.

Начал тестировать запустил в main Console::WriteLine("hello");

и это работает!

Дальше попробовал поработать с управляемыми объектами:



Результат:



Но это была разминка, перейдём к генератору.

Заголовочный файл:

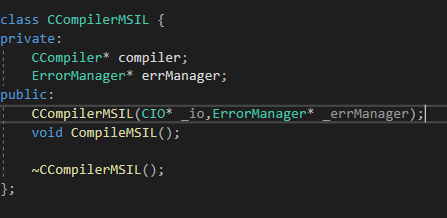


Рисунок 33. Заголовочный файл класса GeneratorMSIL.

Реализация метода CompileMSIL:

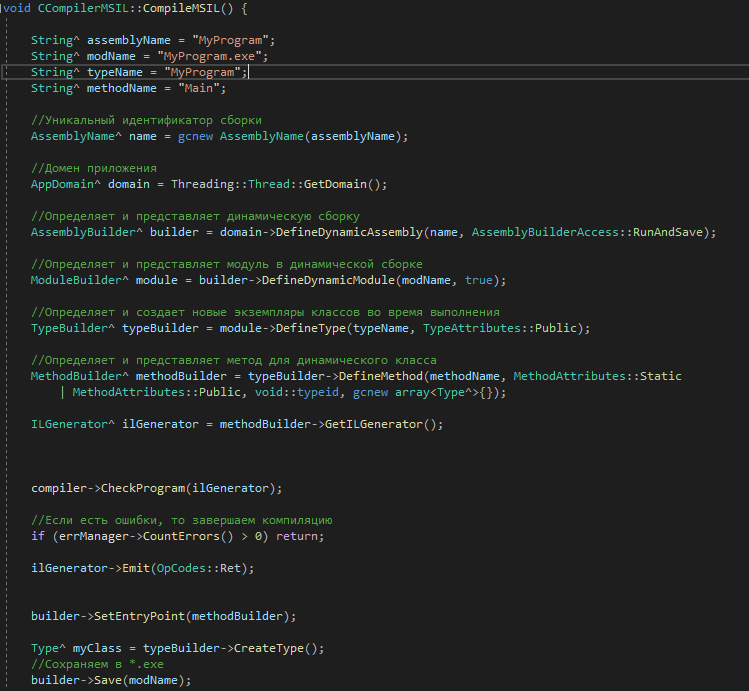


Рисунок 34. Реализация метода CompileMSIL.

Сам стек формируется по аналогии с обратно польской записью.

То есть мы создаём метод Main, в который уже будем класть инструкции из исходной программы.

После этого сохраняем с расширением \*.exe.

Для начала опишем функцию, по добавлению операции в стек:

void AddOper(OperationSymbols op, ILGenerator^ il) {

switch (op)

{

case notEqualSy:

il->Emit(OpCodes::Ceq);

il->Emit(OpCodes::Ldc\_I4\_0);

il->Emit(OpCodes::Ceq);

break;

case lessEqualSy:

il->Emit(OpCodes::Cgt);

il->Emit(OpCodes::Ldc\_I4\_0);

il->Emit(OpCodes::Ceq);

break;

case lessSy:

il->Emit(OpCodes::Clt);

break;

case moreEqualSy:

il->Emit(OpCodes::Clt);

il->Emit(OpCodes::Ldc\_I4\_0);

il->Emit(OpCodes::Ceq);

break;

case moreSy:

il->Emit(OpCodes::Cgt);

break;

case sumSy:

il->Emit(OpCodes::Add);

break;

case divsy:

il->Emit(OpCodes::Div);

break;

case minusSy:

il->Emit(OpCodes::Sub);

break;

case divisionSy:

il->Emit(OpCodes::Div);

break;

case modsy:

il->Emit(OpCodes::Rem);

break;

case equalSy:

il->Emit(OpCodes::Ceq);

break;

case mulSy:

il->Emit(OpCodes::Mul);

break;

case orsy:

il->Emit(OpCodes::Or);

break;

case andsy:

il->Emit(OpCodes::And);

break;

default:

break;

}

}

Добавление переменной:

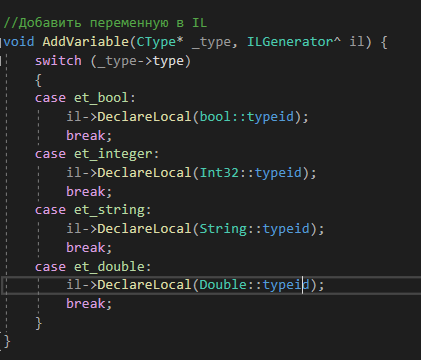


Рисунок 35. Добавление переменной.

Также нужно было в БНФ добавить процедуру writeln, чтобы можно было выводить на консоль.



Рисунок 36. Вывод на консоль.

Для этого мы создаём методы, за основу которых берём метод у Console, а именно WriteLine. И опишем несколько методов, под разный тип выводимого значения.

И ещё опишем приведения типов:

А именно int к double.

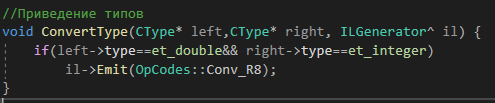


Рисунок 37. Приведение типов.

Теперь будем это использовать, начнём с добавления переменных:



Рисунок 38. Добавление переменной.

В описании однотипных переменных, добавляем вызов функции AddVariable.

Теперь рассмотрим множитель:



Рисунок 39. Множитель.

В зависимости от типа(переменная или константа) добавляем её в стек.

Сам анализ выражения измениться следующим образом:



Рисунок 40. Простое выражение.

То есть делаем приведение типов, если это необходимо и добавляем операцию.

С If уже поинтереснее, нам нужно метки.

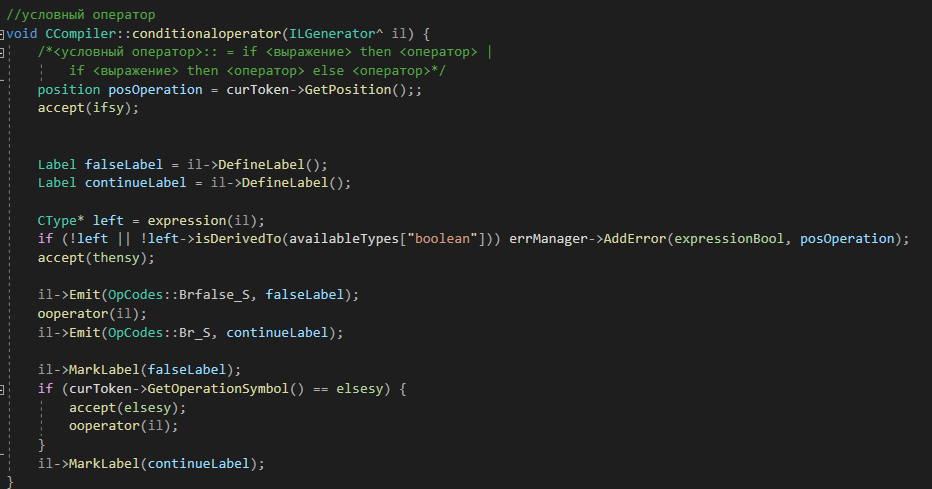


Рисунок 41. Анализ условного оператора.

В этом случае, мы создаём метки и инициализируем их в нужном месте, чтобы программа знала, куда ей перейти в зависимости от результата выражения(true или false).

С While поступаем аналогично:

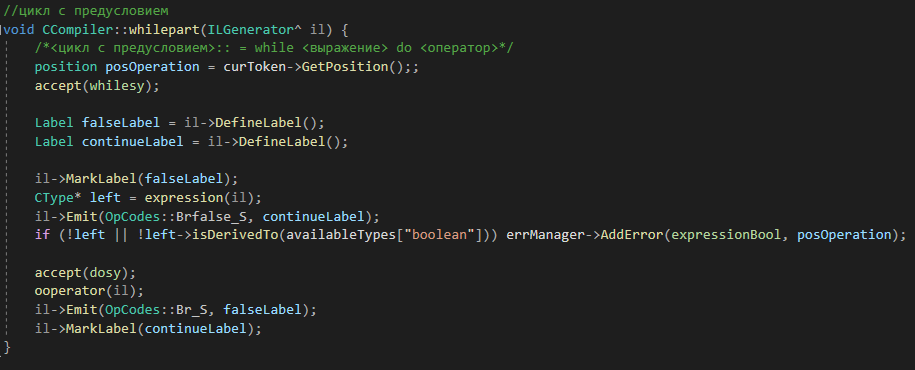


Рисунок 42. Цикл с предусловием.

## Тестирование

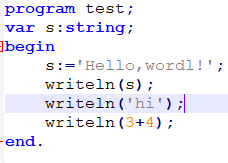
Для просмотра il кода будем использовать ildasm.exe.

Сохранять будем файл MyProgram.exe

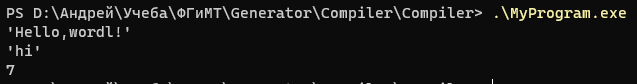
Попробуем с простого.

Exeшник будем запускать из консоль, потому что иначе он сразу закрывается.

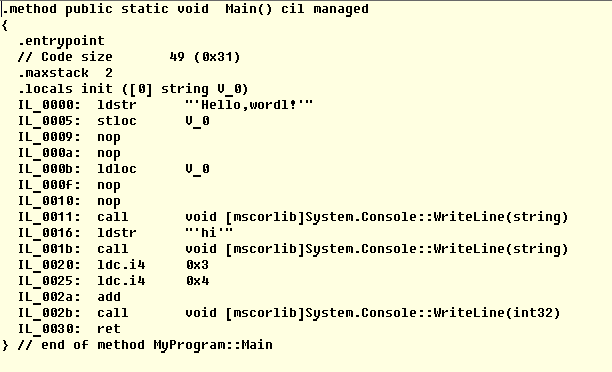
Исходная программа:



Результат программы:



IL код:



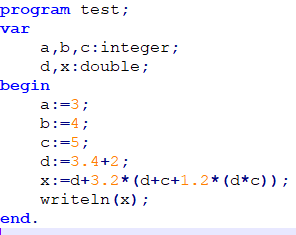
Разберем поподробнее:

* Создаём переменную типа string,
* Кладём в стек строку ‘Hello,wordl!’,
* После этого кладём в это в переменную V\_0,
* Кладём в стек переменную V\_0,
* Дальше вызываем WriteLine
* Кладём в стек строку ‘hi’,
* Дальше вызываем WriteLine
* Кладём в стек константу 3(int)
* Кладём в стек константу 4(int)
* Кладём в стек операцию Add
* Дальше вызываем WriteLine

И всё работает!

Возьмём пример по сложнее с приведением типов.

Исходная программа:



Результат:



Но так ли это?

Заменим переменные на их значения:

X:=5.4+3.2\*(5.4+5+1.2\*(5.4\*5));

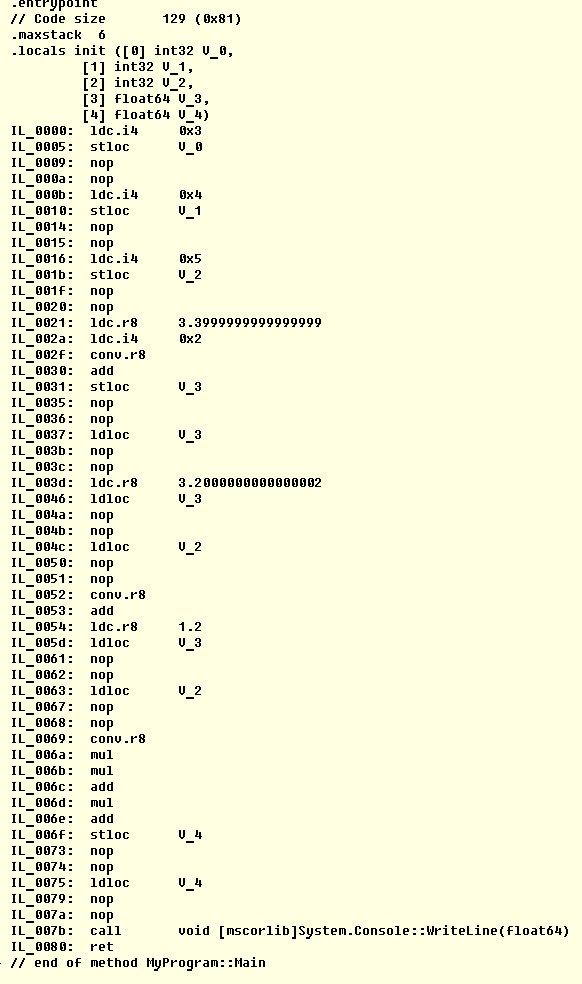
Загоним в Python



И да, результаты сошлись.

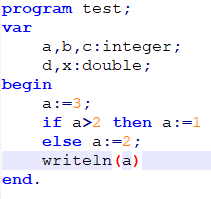
Приведение работает!

IL код:



Перейдём к интересному If:

Исходная программа

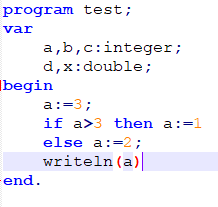


Результат



Рассмотрим else

Исходная программа

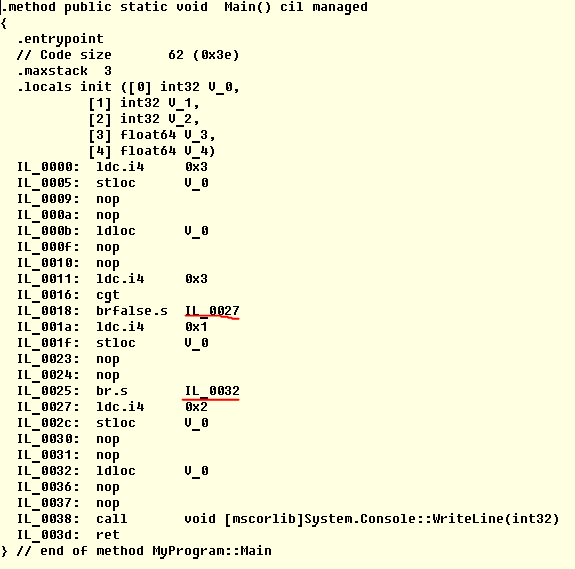


Результат



Работает.

IL код



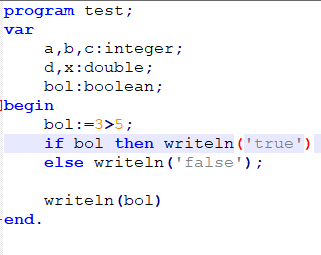
А вот и те метки(помечены красным), которые мы создавали в конструкции условного оператора, для перехода.

Означает, что если значение выражение false перейди к команде IL\_0027

А если было true, то мы в конце действий переходим на команду IL\_0032, чтобы обойти ветку else.

Попробуем с bool переменными:

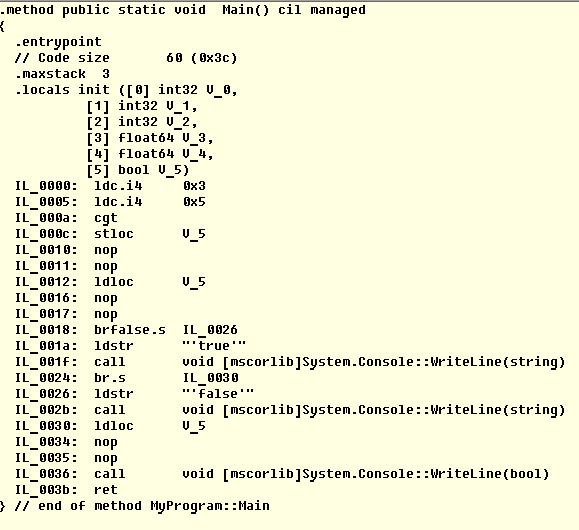
Исходная программа:



Результат:

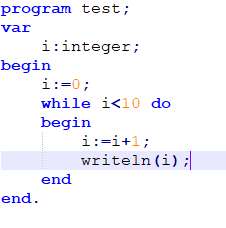


IL код:

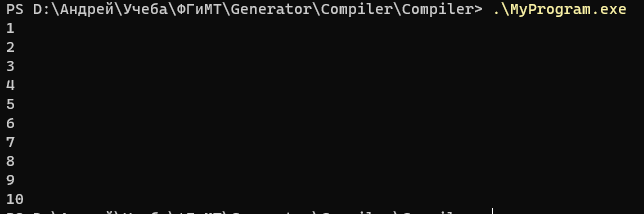


Попробуем while:

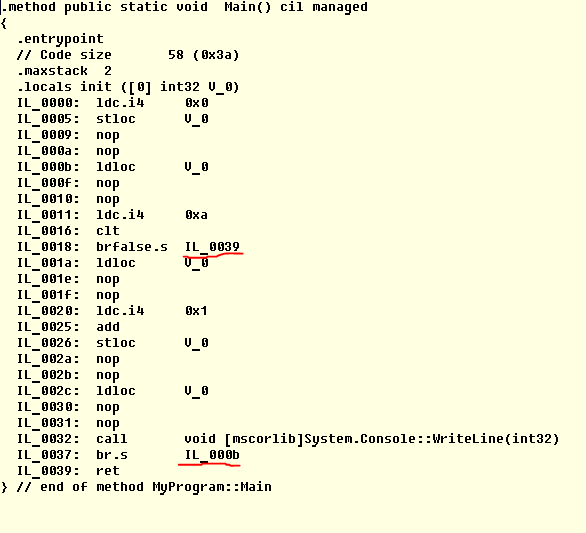
Исходная программа:



Результат:



IL код:

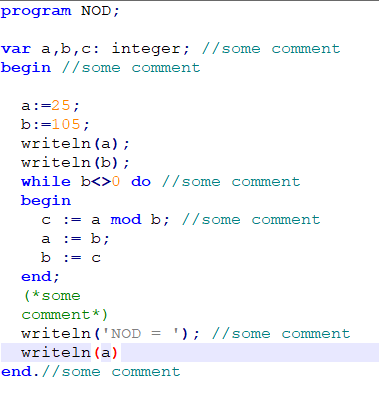


Здесь также появляются метки.

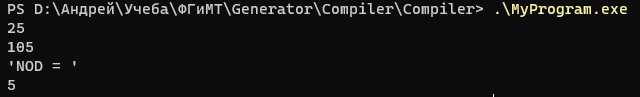
Если выражение false, то переходим к команде IL\_0039, если выражение true, то присваиваем i =i+1; и выводим, после этого переходи на команду IL\_000b, для повторной проверки условия.

Теперь запустим алгоритм по нахождению НОД двух целых чисел:

Исходная программа:

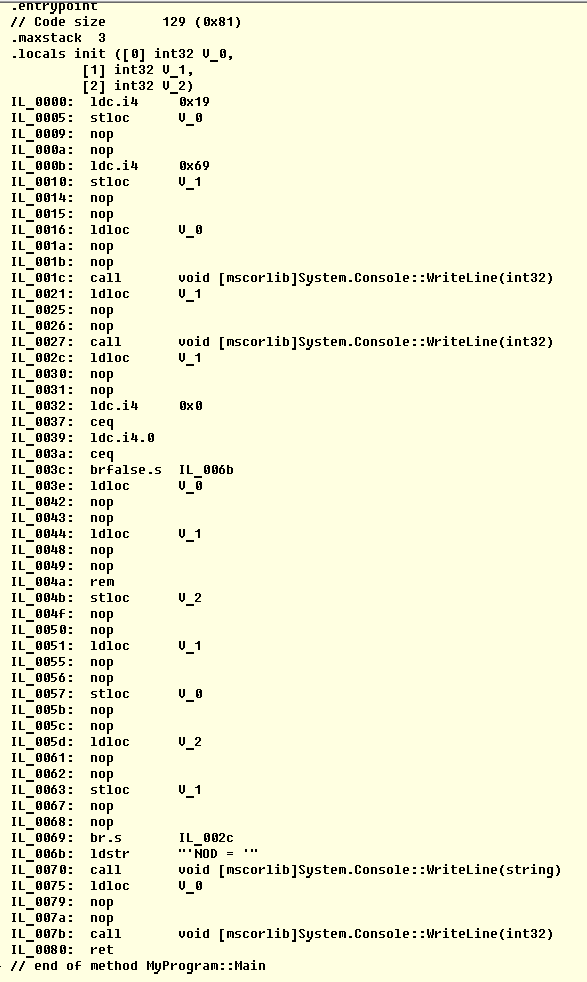


Результат:



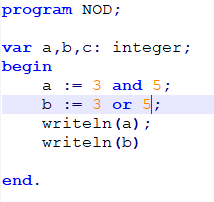
И получаем, что NOD(25,105) =5 , что правда.

IL код:

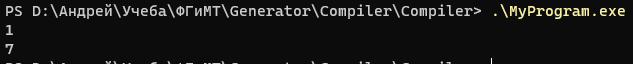


Также ещё проверим побитовые операции

Исходная программа:



Результат:



3 = 011

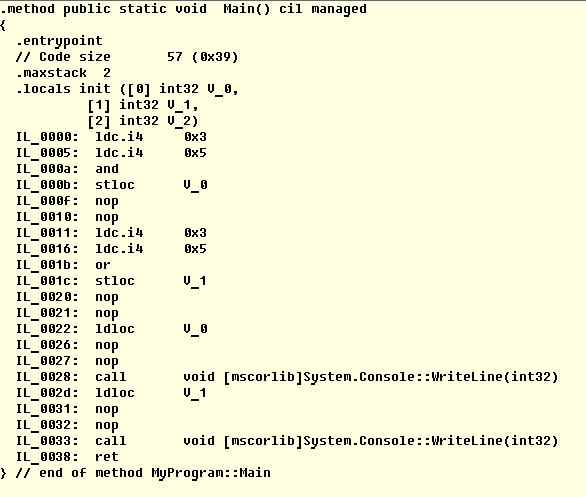
5 = 101

3&5 = 1

3|5 = 7

Правильно.

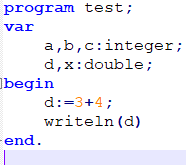
IL код:



Также ещё один тест на приведение типов:

В double положим int

Исходная программа:



Результат:



IL код:

