|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ | | | | | | | | | | |
|  | | |  | | | | |  | | |
| ФГБОУ ВО «Пермский государственный национальный  исследовательский университет» | | | | | | | | | | |
|  | | |  | | | | |  | | |
|  | | ОТЧЕТ  по дисциплине «Формальные грамматики и методы трансляции» | | | | | | |  | |
|  | | |  | | | | |  | | |
|  | Работу выполнил  студент гр. ПМИ-1,2  Жуков А.Е. \_\_\_\_\_\_\_  (подпись)  «\_\_\_» \_\_\_\_\_\_\_\_\_ 2021 | | | |  | Работу проверил  ассистент кафедры МОВС  Пономарев Ф.А. \_\_\_\_\_\_\_\_\_  (подпись)  «\_\_\_» \_\_\_\_\_\_\_\_ 2021 | | | |  |
|  |  | | | |  |  | | | |  |
|  | | | | Пермь 2021 | | |  | | | |

# Техническая документация

1. Компиляторы: принципы, технологии и инструментарий, 2 е изд . : Пер . с англ. — М. : ООО “И.Д. Вильямс”, 2018. — 1184 с. : ил. — Парал. тит. Англ (на английском не осилил, много времени в перевод уходит☺)
2. Разработка Паскаль-компилятора / Л.А. Залогова.

# Оглавление

[Техническая документация 2](#_Toc90638193)

[Оглавление 3](#_Toc90638194)

[Лексический анализатор 4](#_Toc90638195)

[Проектирование 4](#_Toc90638196)

[Разработка 8](#_Toc90638197)

[Тестирование 11](#_Toc90638198)

[Синтаксический анализатор 14](#_Toc90638199)

[Проектирование 14](#_Toc90638200)

[Разработка 15](#_Toc90638201)

[Тестирование 16](#_Toc90638202)

[Семантический анализатор 18](#_Toc90638203)

[Проектирование 18](#_Toc90638204)

[Разработка 20](#_Toc90638205)

[Тестирование 22](#_Toc90638206)

[Нейтрализация синтаксических ошибок 23](#_Toc90638207)

[Проектирование 23](#_Toc90638208)

[Разработка 24](#_Toc90638209)

[Тестирование 26](#_Toc90638210)

[Нейтрализация Семантических ошибок 30](#_Toc90638211)

[Проектирование 30](#_Toc90638212)

[Разработка 31](#_Toc90638213)

[Тестирование 32](#_Toc90638214)

[Послесловие 34](#_Toc90638215)

# Лексический анализатор

## Проектирование

Я долго думал, много читал, чтобы как можно лучше спроектировать все сущности, обдумывая много идей и решений, но так и не пришел ни к чему, как мне кажется, нужному. Поэтому решил разобраться в самом процессе.

Сразу выделю класс IO, в котором будет метод getNextToken(), по поводу полей, пока не определился, но нужен, как минимум текущий токен, которым управляет класс CToken.

В CToken несколько типов полей: TokenType, EOperationKeyWords, String и CVariant. EOperationKeyWords содержит все ключевые слова, служебные символы и знаки операций. Вся структура следующая:

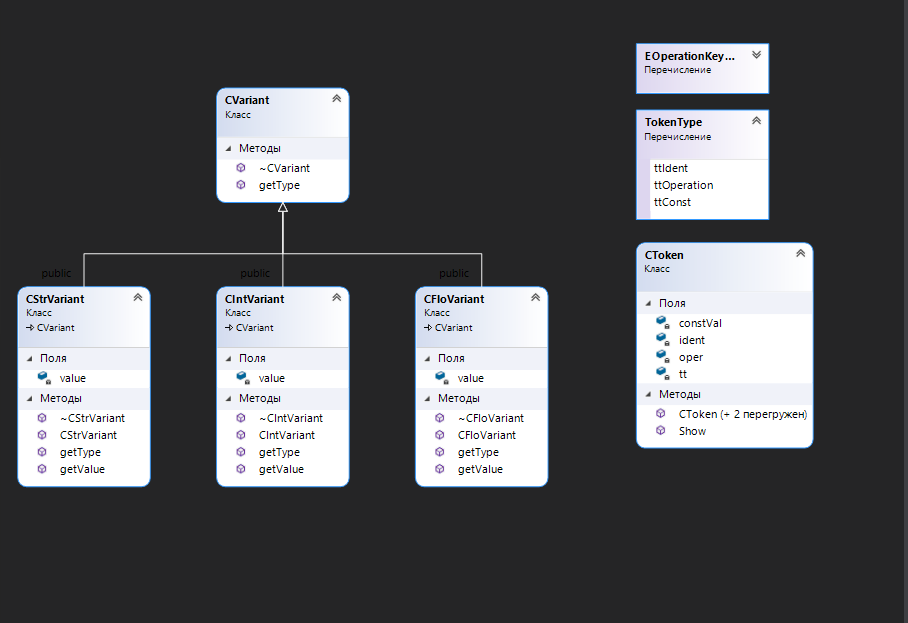


Рисунок 1 – Структуры CToken и CVariant.

Класс IO имеет следующую структуру:

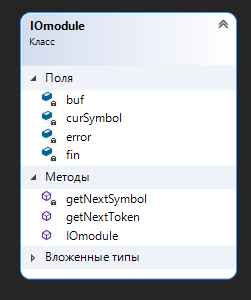


Рисунок 2 – Структура класса IO.

Для получения токенов в методе getNextToken() был реализован автомат. Я посчитал, что он отлично подходит для данной задачи, а также хорошо дополняем в будущем.

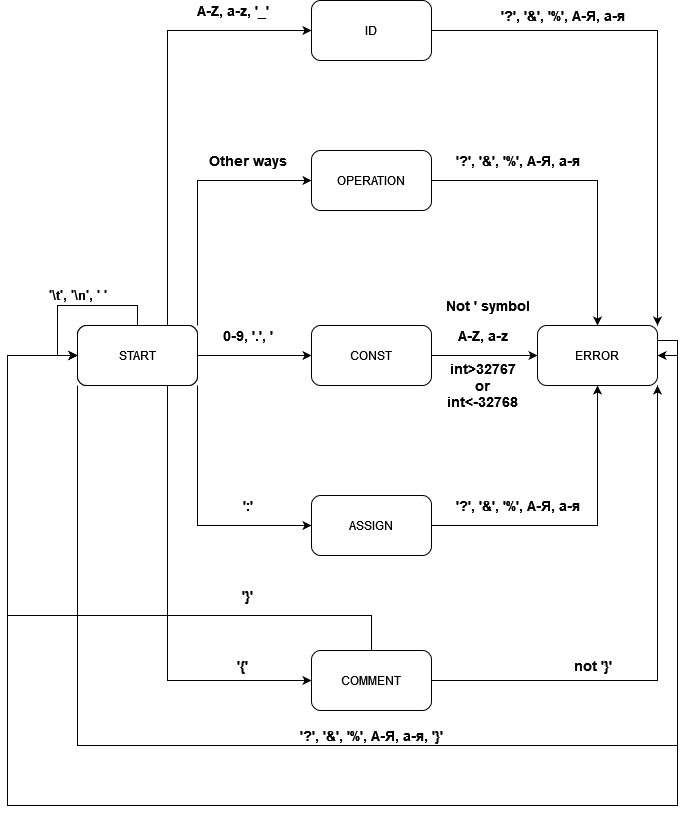


Рисунок 3 – Структура автомата

В модуль ввода-вывода добавил класс lexErrors, который будет хранить информацию о лексических ошибках. В этом классе есть перечисление с именами ошибок, метод show(), для вывода строки, в которой находится ошибка, а так же текст самой ошибки.

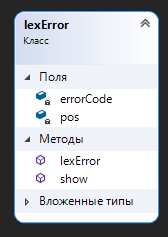


Рисунок 4 – Структура класса lexError.

## Разработка

Начал с описания CVariant. Все методы тут аналогичны, в конструкторе присваивается значение. Метод getValue() возвращает значение каждого варианта, getType() – тип. Так же прописал виртуальные деструкторы, чтобы память освободилась точно, все-таки си два креста, шутки с памятью плохи.

Далее идет CToken. В заголовочном файле имеется перечисление всех ключевых слов и операторов, закинул все в одно, чтобы не усложнять. Создал два map: в одном ключ перечисление, значение – само ключевое слово и операция; в другой поменяны местами ключ-значение. Вторую по факту не использовал в итоге, но может доведу до ума в конце, чтобы быстрее работало. По методам по факту 3 конструктора, принимающие тип токена, и в зависимости от типа значение: ключевое слово или операция, строку с идентификатором, константа, которую храню в ранее описанном классе CVariant. Метод Show() выводит информацию о токене: тип и значение. Для констант использую dynamic\_cast, чтобы привести базовый абстрактный класс к дочерним и вытянуть данные.

Сам модуль ввода-вывода содержит, кроме себя самого класс с лексическими ошибками, про них чуть позже. По факту самый существенный метод в модуле ввода-вывода – получение следующего токена. Конструктор получает на вход имя файла, открывает его и заполняет им строку. Я не думаю, что будет настолько большая программа, что std::string выдаст overflow, по крайней мере, в данной работе это точно сработает. Прописаны две вспомогательные функции: первая сдвигает указатель с текущего символа на следующий и возвращает символ; вторая ищет в map индекс оператора или ключевого слова, если не находит, то -1. Казалось бы, в map 58 элементов, искать должно быстро, но есть проблема, что идет сравнение строк, что не очень быстрая операция, но пока пусть будет так, опять же можно вернуться и замутить оптимизацию.

Получение токенов происходит через метод getNextToken(). Это автомат с семью состояниями. Я обернул его в while(true) в котором switch(state). Все программируется согласно спроектированному автомату. Есть строка для токена, которая может приводится к константам. Все состояния, за исключением ERROR и COMMENT возвращают новый токен.

Состояние ID для идентификаторов. Считываем до того момента, пока поступают допустимые символы идентификатора. Проверка на ключевое слово, если идентификатор ключевое слово – возвращаем, что токен ttOperation, иначе ttIdent.

Состояние OPERATION для операций. Сначала проверка на знаки сравнения, потому что есть <>, <=, >=. Проверка по первому символу, дальше в зависимости от второго возвращаем ту операцию, которая была. Если же это не оператор сравнения, то может быть ключевое слово. Считываем, пока на вход подаются буквы. Так, если пришел +, - либо что-то из похожего, то цикл просто скипнется. Ищем индекс ключевого слова в map, возвращаем токен.

Состояние CONST для константных значений. Первым делом проверка на символ 39 из ASCII, что является одинарной кавычкой, то есть в случае строковой константы. Строковая константа заканчивается так же одинарной кавычкой. Если же это не строковая константа, то значит, что числовая, считываем пока на входе цифры, при получении точки запоминаем, что это вещественная константа. Как только цифры заканчиваются возвращаем токен.

Состояние ASSIGN для присваивания. Проверяем следующий символ после «:», если это =, то возвращаем токен оператора присваивания, иначе просто «:».

Состояние COMMENT для пропуска комментариев. Просто скипаем все символы, пока не получим }.

Состояние ERROR для вывода ошибок.

Для хранения ошибок используется класс lexError. Конструктор получает на вход код ошибки и позицию. Я не храню список всех ошибок, а только одной. То есть получив ошибку, я создаю экземпляр этого класса, вывожу ошибку, удаляю созданный экземпляр. Выделил на этапе лексического анализа 7 ошибок: запрещенный символ, неправильное именование идентификатора, превышение максимальной длины идентификатора, неоткрытый комментарий, незакрытый комментарий, превышение максимального значения для integer, незакрытая строковая константа. Так, метод show() класса lexError выводит строку, в которой произошла ошибка, указывает на символ, в котором ошибка, выводит текст ошибки. Единственное, что мне не нравится в этом всем, что я использую глобальную переменную с текстом программы.

Почти в каждом состоянии идет проверка на запрещенные символы. START может вернуть ошибку, что комментарий не открыт, при получении символа }.

ID может вернуть ошибку о том, что превышена максимальная длина идентификатора.

CONST может вернуть несколько ошибок. Строка не закрыта, если достигнут конец текста программы, а это все еще строковая константа. Неправильное имя идентификатора: в это состояние проваливаемся, когда получаем число, идентификатор не может начинаться с цифры, таким образом, получая букву в этом состоянии, делаем вывод, что идентификатор имеет неверное имя. Здесь же идет проверка на превышение допустимого значения для integer.

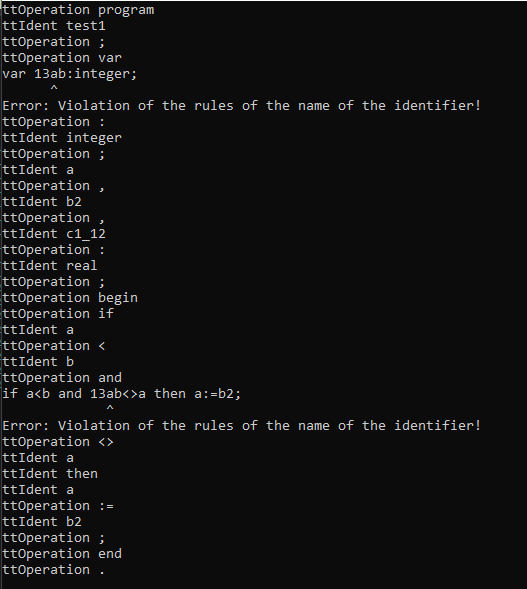
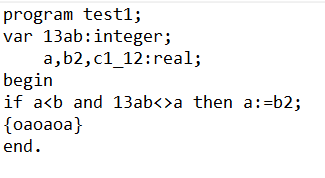
COMMENT возвращает только одну ошибку, когда комментарий не закрыт, то есть текст программы закончился, а символа } так и не поступило.

В состоянии ERROR происходит вывод информации об ошибке, удаление экземпляра класса и переход в стартовое состояние.

## Тестирование

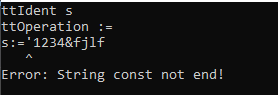
Я просто написал некоторые программки, которые все случаи охватывают, все ошибки, большинство ключевых слов.

1. Неверное именование идентификатора, операции сравнения, пропуск комментариев.

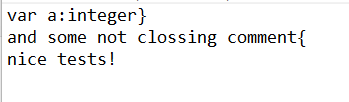


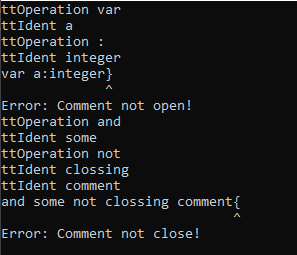
1. Не закрытая строка.

s:='1234&fjlf

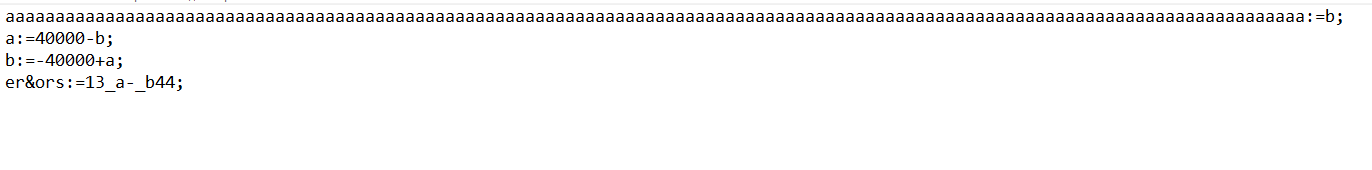


1. Неоткрытый комментарий и незакрытый комментарий.





1. Превышение длины идентификатора, выход за пределы integer, запрещенные символы.





# Синтаксический анализатор

## Проектирование

Синтаксический анализатор довольно прост. По факту это запись всех БНФ. Класс Syntax, имеющий поля CIO типа IOModule\* и curToken типа CToken\*. Конструктор, получающий на вход имя файла, инициализирует модуль ввода-вывода с данного файла, получает первый токен и вызывает первую БНФ. Несколько вспомогательных функций: получение следующего токена, подтверждение типа токена, подтверждение типа операции. На этом все, все остальное это БНФ-ки.

## Разработка

Изначально я просто забивал БНФ-ки последовательно, не задумываясь о выборе альтернатив для некоторых из них, считая, что этим займется семантический анализатор. Однако после меня осенило, что я могу выбирать альтернативы в зависимости от того, с какого токена она начинается. То есть по факту я начал почти все заново, анализируя каждую БНФ-ку, в которой есть альтернативы и сами эти БНФ-ки.

До этого переломного момента я думал, что синтаксический анализатор — это максимально скучно, но альтернативы заинтересовали меня. Учитывая упрощенный функционал компилятора, я смог прописать выбор альтернатив для всех БНФ-ок.

По факту все, описывать тут больше особо нечего☺

## Тестирование

На данный момент я провел лишь один тест, в котором программа не содержит никаких ошибок, получив на выходе “Good!”, я обрадовался и пошел спать.

1. Тот самый радостный тест, по факту он определяет работу цикла while и условного оператора.

program Evklid;

{ определение наибольшего общего делителя }

var M, N :integer;

begin

while M <> N do

begin

if M>N

then M := M-N

else N := N - M

end;

end.

1. Работа типов, описание однотипных переменных.

program syntaxtest2;

type mylim=1..10;

var x:integer;

integer:real;

m,q:array[1..10] of boolean;

begin

x:=5;

q:=m;

w:=m;

m:=6;

end.

1. Раздел констант, вложенные циклы while, новые арифметические операции.

program SyntaxTest3;

const LIMIT = 10000;

var n,i,j,s,lim,c,d : integer;

begin { основная программа }

while(i<LIMIT) do begin

s:=1; lim:=i;

while(j<lim) do begin

c:=i mod j;

d:=i div j;

if c = 0 then begin

s:=s+j;

if (j<>d) then s:=s+d; {дважды не складывать корень числа}

end;

end;

if s=i then i:=0;

end;

end.

1. If-else много

program SyntaxTest4;

var a,b,c,d,x:real;

begin

d:=b-4\*a\*c;

if d<0 then begin

x:=-1;

end else if d=0 then begin

x:=(-b)/2\*a;

end else begin

x:=(-b+d)/2\*a;

end

end.

1. Перечислимый тип

program DEC\_BIN;

var digits:(a,b);

res:string;

d:0..1;

begin

res:='';

while (x<>0) do begin

d:=x mod 2; res:=digits+res;

x:=x div 2;

end;

DEC\_BIN:=res;

end.

В целом, тут особо нечего тестировать, БНФ, если правильно написаны, то они и будут правильно работать. Постарался охватить максимум для основных операторов.

# Семантический анализатор

## Проектирование

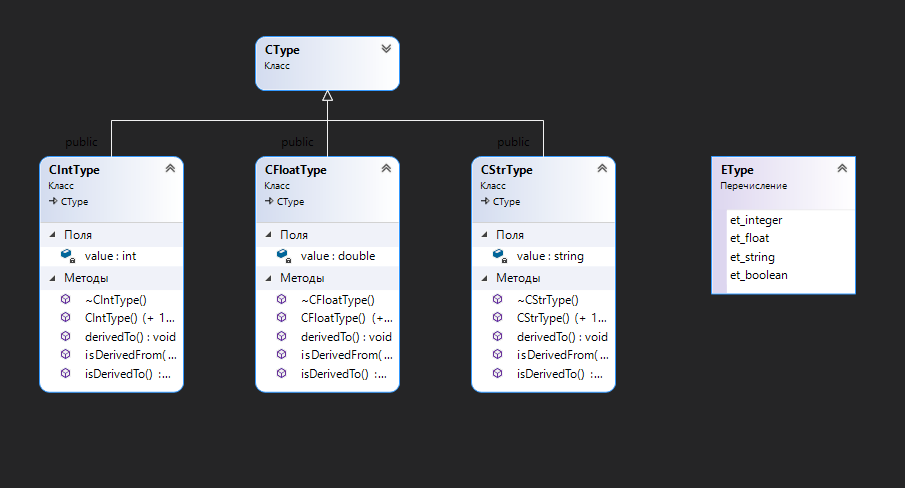
Первое, что я понял, семантика должна быть непосредственно связана с синтаксисом. Семантический анализатор должен работать параллельно с синтаксическим.

Далее, скорее всего, нужна таблица идентификаторов, для хранения переменных и их значений, что будет очень актуально для выражений. Так же стоит хранить и тип переменной.

Чтобы выражения работали в общем случае, необходим специальный класс, хранящий допустимые типы данных, в нашем случае это 4 типа: integer, real, string, boolean. Просто всем выражениям будет приписывать условно тип CType. В нем же пропишем проверки на приводимость типов и само приведение типов, если оно возможно. Так же добавим вспомогательные методы для получения значения и типа.

Для хранения доступных переменных воспользуемся map, где ключ строка, а элемент один из экземпляров класса.

По факту, я усложнил себе задачу, семантика – это проверка типов, никаких значений. Удалил метод с получением значения, добавил флаг для константных значений, доопределил еще один конструктор, в который можно положить значение и флаг о константности. Прописал метод isDerivedFrom(), который будет использоваться для присваивания, но об этом подробнее в разработке. В итоге получилось следующее:



Немного переделал deriveTo(), теперь у него есть возвращаемое значение CType\*, а принимает он два аргумента CType\*, логика его не изменилась, просто теперь добавляется return.

Добавляю поле isConst для фиксирования констант.

## Разработка

С самого начала попал в проблему с плюсовыми абстрактными классами. По итогу в базовом классе 4 чисто виртуальных и 2 обычных метода. Виртуальные функции: функция проверки приводимости типов, функция приведения типов, виртуальный деструктор, проверка приводимости типов для присвания. Обычные функции для получения и установки типа.

Конструкторы всех классов инициализируют свой тип и значение по умолчанию: для чисел – 0, строки – пустая строка, boolean – false.

Функция isDerivedTo принимает на вход некоторый экземпляр класса CType. В зависимости от типов возвращает информацию о том, приводимы типы или нет. Так, только целочисленный тип можно привести к вещественному и целочисленному, остальные типы приводимы только к самим себе.

deriveTo принимает два CType, для первого меняет поле типа на тип правого. Дальше просто ретерн лефт.

isDerivedFrom принимает на вход CType, логика почти та же что и у isDerivedTo, но есть отличия для int и float. Если слева float, то справа может быть, как int, так и float. Если же слева от знака присваивания int, то может быть только float.

Описал хранение переменных. У нас упрощенный вариант, поэтому сделал следующим образом. Запоминаю названия переменных в отдельный массив. После определения их типа уже непосредственно добавляю в map.

Далее описываю для всего, что связано с выражениями проверку приведения типов через isDerivedTo(). Там, где есть операторы сравнения, выполняю проверку, чтобы выражение слева и справа были приводимы, иначе как их сравнить? Далее возвращаю в таких случаях новый булевый тип.

Для присваивания использую isDerivedFrom. Здесь важно, чтобы правая часть была приводима к левой, поэтому немного противоположная логика.

Для ifStatement() после if и получения выражения выполняю проверку на то, булевое ли выражение вообще. Аналогично для while.

factor() теперь проверяя идентификаторы, возвращает тип идентификатора, для констант аналогично. Так же добавил возвращения для «true/false», тоже ловиться в идентификаторе.

Совсем забыл про константы. Они используются так же, как переменные, только флаг isConst у них в значении true, а тип выводится из полученного токена.

## Тестирование

Я не стал особо ничего придумывать большого здесь, закинул лишь общий тест на все, постарался охватить все ньюансы. Основное тестирование при получении ошибок. Здесь мы живем в мире радости и счастья, в котором нет ошибок!

program semanticTest1;

const a=3.14;

b=5;

c='avc';

var r,l:integer;

p:real;

begin

while(p>0) do begin

if(r=b) then r:=r+b+1;

else l:=b+l;

p:=p+a;

end;

end.

# Нейтрализация синтаксических ошибок

## Проектирование

Только сейчас осознал, что это как-то странно, ловить ошибки в блоке с нейтрализацией, но логика в этом все-таки есть. Поэтому пусть про ошибки будет здесь, а не в самом анализаторе.

Надо учесть ошибки для лексического анализатора. На ум приходит хранить все ошибки, например, в массиве, а после окончания выводить их. Чтобы ничего не ломалось и нужна нейтрализация, но о ней, думаю, разговор пойдет в разработке, потому что, в целом, проектировать нечего.

Создал ErrorManager, создал массив пар, в котором первое – код ошибки, второе – позиция. Для получения позиции, прописал метод получения текущей позиции в модуле ввода-вывода. При инициализации Syntax добавил инициализацию ErrorManager. Прописал поле isError для CType, чтобы не выводить повторные семантические ошибки.

Добавил функцию skipTo, принимающую список токенов, до которых будет выполняться пропуск.

## Разработка

Начну со skipTo. Здесь бесконечный цикл. В зависимости от типа токена выполняется поиск в поданном массиве. Если токен найден, то цикл останавливается.

Что касается менеджера ошибок, то здесь интересен вывод. Я решил выводить в конце текст всей программы и параллельно с этим ошибки, если они имеются. Единственная проблема, что позиция ошибки обычно соответствует концу следующего токена, что немного нелогично, но главное, что суть в целом ясна по поводу ошибок.

Отлавливаю ошибки через try-catch. Не знаю, насколько я правильно это делаю, но у меня теперь куча этих try-catch, причем вложенных, но главное, что оно работает!

Сложность состоит лишь в том, чтобы понять, до каких токенов пропускать. В самом простом случае, можно скипать большинство до следующих разделов. Например, константы до переменных, а переменные до операторов. Однако я стараюсь пропускать не так грубо.

Первая БНФ – это программа. Так, если само ключевое слово program упущено, то скипаю до следующих разделов. При отсутствии имени, аналогично, как и при отсутствии точки с запятой.

Далее константы. Я не провожу проверку на правильность написания const, т.к. в случае ошибки, она вылезет в операторах. Заходим в эту БНФ по ключевому слову const, логично, что, если оно неверно, то и никак не зайдешь. Дверь есть – ключа нет. После ключевого слова должен идти идентификатор, если его нет, то скип до следующей БНФ. Если пропущено =, то скип до точки с запятой, в надежде, что следующая константа будет нормально описана, и скип до следующей БНФ. Если после равно не константа, то скип до следующей БНФ или точки с запятой. Если пропущена точка с запятой, то скип до идентификатора или до следующей БНФ.

С переменными сложнее. Если получен не идентификатор, то скип до запятой, точки с запятой или до следующей БНФ. Дальше, если же все-таки получена запятая, то нужен снова идентификатор. В случае его отсутствия скип до двоеточия, чтобы записать правильные переменные, или до запятой. Если не получено двоеточие, то скип до точки с запятой или до идентификатора. Если идентификатор не получен, то выходим из объявления однотипных переменных.

Дальше немного для типов, точнее для простого типа, куда мы попадаем при описании переменных. Если мы получаем не идентификатор, то скип до точки с запятой или до следующей БНФ. Если же получен идентификатор, который не соответствует стандартным типам, то аналогичный скип.

Самое сложное, как мне кажется – операторы! Сразу же падаем в составной оператор. Если begin не получен, то скип до if, while, end. Если в конце не получен end, то просто ошибку добавляем, это последний end в программе.

Оператор присваивания. Если не ловим присваивание, то скип до идентификатора, константы, точки с запятой, левой скобки. Такие пропуски следуют из выражений. Вера в то, что просто пропущен оператор присваивания, либо в нем ошибка, но выражение верное.

Чтобы не запутаться, я сразу прописал для выражений. По факту тут только в factor прописать. Если идентификатор какой-то не такой, то скип до точки с запятой, do, then, end. Если открывалась левая скобка, то в случае потери правой скобки, просто переход к следующему токену.

Условный оператор. Тут можем упустить только then, остальное уже описано. Если then нет, то скип до идентификатора, например, для оператора присваивания, и скип до begin, while, if, для составного оператора. Тут есть какой-то прекол с оператором после else. Там не надо после него точку с запятой, если он один. Я не знаю, насколько это правильно, но если что, то это фича, а не баг!

Оператор while. Тоже только для do, скип аналогичный для if.

## Тестирование

1. Вариации ошибок в program

Рассмотрел все случаи, поэтому сейчас будет суперкрасивый коллаж!

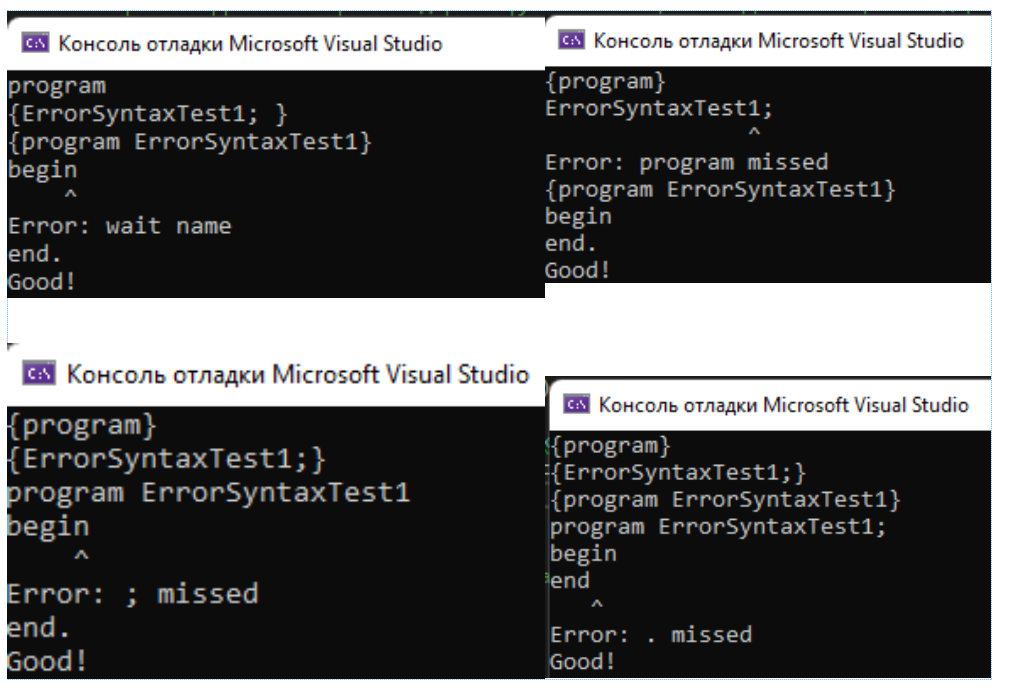
{program}

{ErrorSyntaxTest1; }

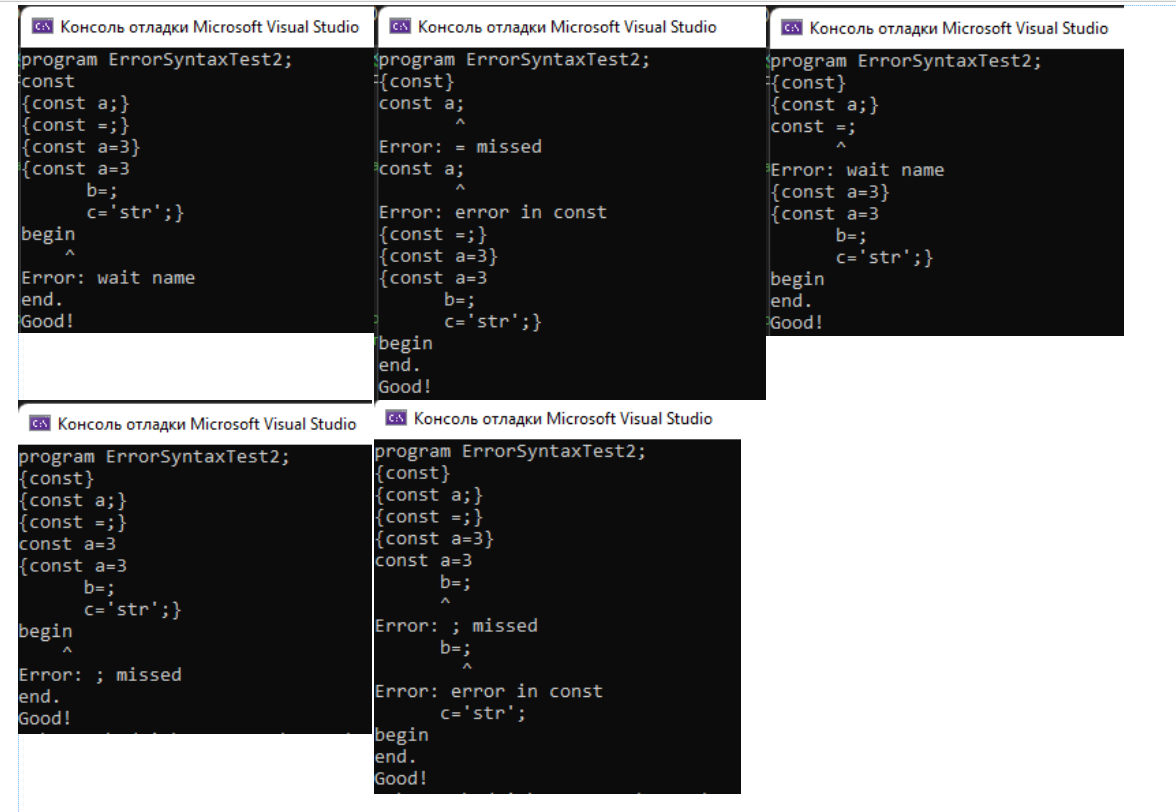
**program** ErrorSyntaxTest1

**begin**

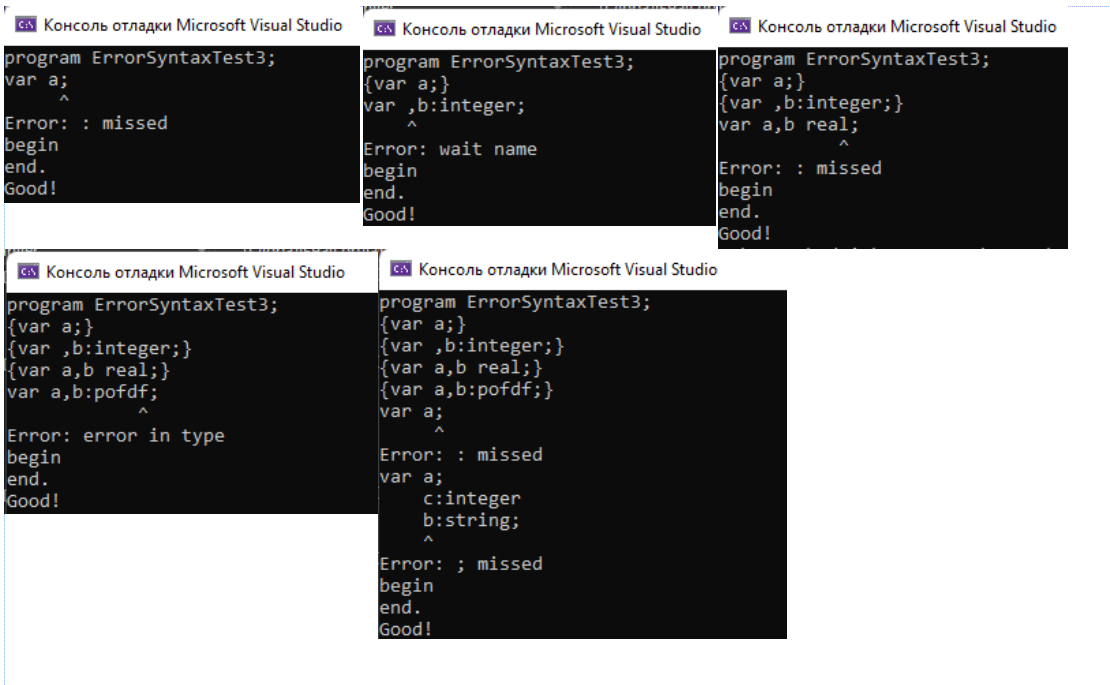
**end**



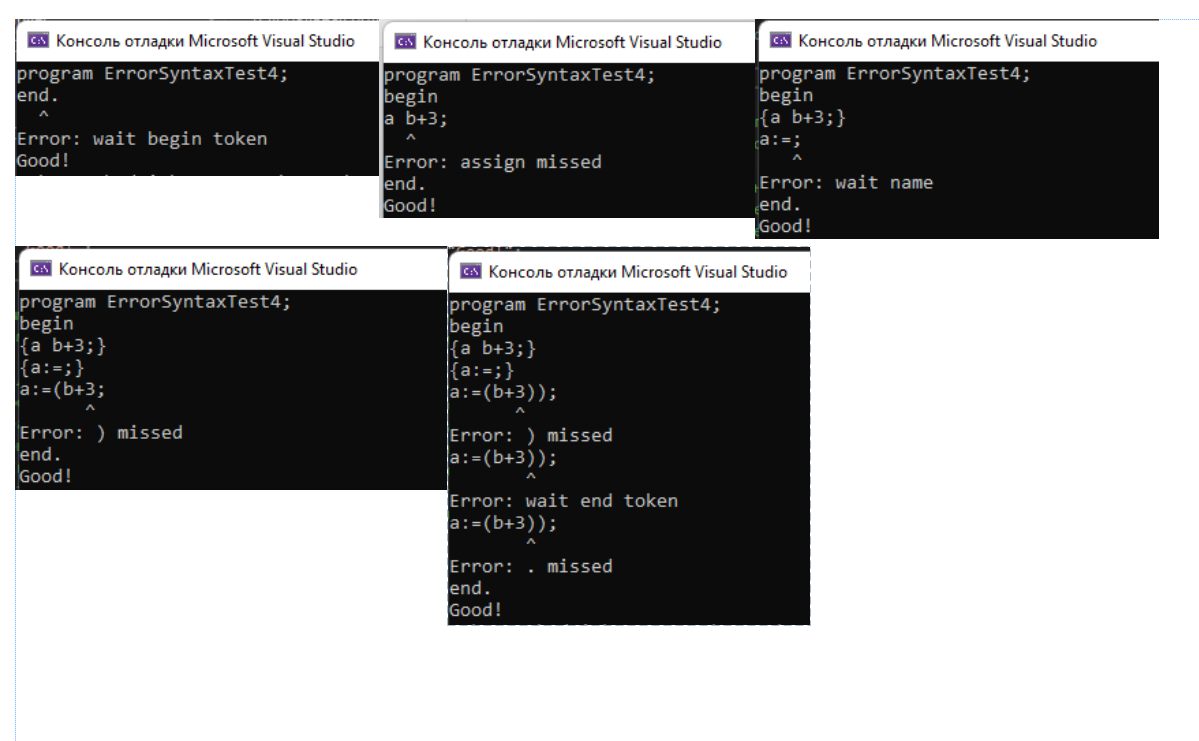
1. Константы. Тут тоже будет коллажик! Классно кстати в Win11 сделали округленные окна, перфекционисты после таких выделений рыдают.



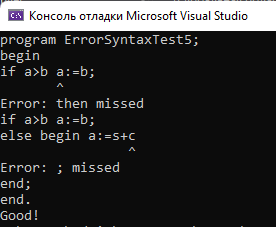
1. Раздел переменных



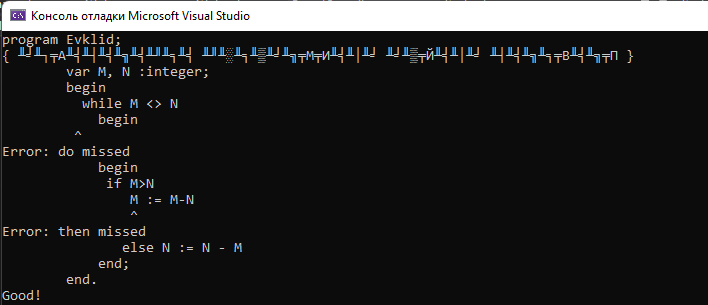
1. Операторы. Начнем с присваивания, тем самым охватив большую часть операторов в принципе. Интересно получилось с встречающейся ). В целом, в БНФ никак не учитывается, что может быть такое. Так что оно просто выходит из простого оператора, оставляя текущим токеном эту скобку.



1. Условный оператор. Тут особо издеваться не будем, примем, что выражения верны. В них все равно скип до then, begin, ;, end.



1. В целом, нет смысла проводить для while, там все аналогично. Поэтому взял просто код и сделал в нем рандомные ошибки.



# Нейтрализация Семантических ошибок

## Проектирование

Здесь воспользуемся уже написанным менеджером ошибок. Изначально хотел хранить сведения по переменной, были ли ошибки про нее. Решил отказаться от этой идеи. Пусть просто много раз пишет про нее, ничего страшного))

Так же, наверное, стоит скипать, если есть ошибки в выражении, определить токены, до которых нужно скипать, наверное, не составит труда, учитывая опыт синтаксиса.

## Разработка

Решил начать с констант, там один тип ошибок, что описывается повторная переменная. Решаем просто, если встречается уже описанная переменная, то просто кидаем сообщение об этой ошибке и скипаем до следующей константы или БНФ.

Аналогичная проблема есть для переменных. Решение аналогичное. Разве что скипаем немного до других токенов. Так, если идентификатор до запятой уже используется, то скип до запятой, чтобы описать следующие переменные, либо до следующих однотипных переменных, либо вообще до раздела операторов. Если же после запятой, то аналогичный скип, разве что добавляется двоеточие. Даже если до запятой ничего не получили, то ошибки не будет, потому что в массив переменную не добавили.

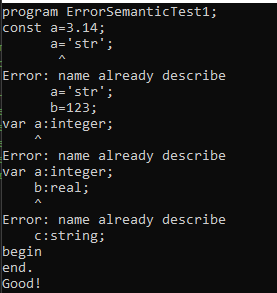
Далее присваивание. Как только в него заходим, запоминаем имя переменной, к которой выполняется присвоение. После анализа выражения, проверяем, есть ли вообще такая переменная в доступных. Если нет, то кидаем ошибку, что переменная не описана. Иначе проверяем приводимость типов, если что-то идет не так, то снова кидаем ошибку, что типы не приводимы.

Выражения – тут аналогичные ошибки. Так, если в factor встречаем не описанный идентификатор или что-то вообще не то, то скипаем до конца конструкции и возвращаем nullptr. Таким образом, нужно обернуть с учетом этого все выше и оператор присваивания, а далее держать в голове, что может прийти nullptr. В term и simpleExpression только ошибки на приводимость типов.

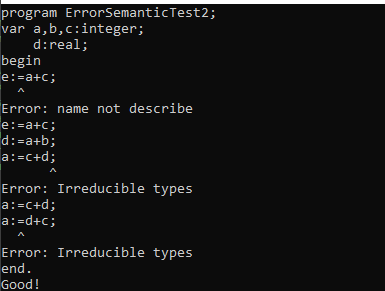
Условный оператор может дать семантическую ошибку, если выражение после if не логического типа. Аналогично для while, где после while может быть не логический тип, то это семантика, кидаем ошибку.

## Тестирование

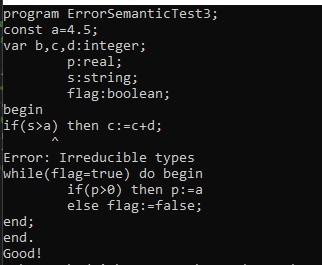
1. Проверка на повторно описанных переменных.



1. Приводимость типов.



1. If, while. Немного проверки приводимости еще.



# Послесловие

Какие проблемы остались: встречаем лексическую ошибку – кидаем сообщение и выходим; при выводе ошибок в менеджере дублируется строка, точнее она иногда дублируется, иногда нет☺; если после в if в операторе после then одно присваивание, то там не надо ставить точку с запятой, будем считать, что переосмыслили условный оператор!