Пермский Государственный Национальный Исследовательский Университет

Формальные языки и методы трансляции

Отчёт по работе «Компилятор»

Паскар Алексей   
Владимирович   
ПМИ-2-2019

## Оглавление

Компилятор 4

Модуль ввода-вывода 6

1. Описание 6
2. Проектирование 6
3. Реализация 6
4. Тестирование 7

Лексический анализатор 8

1. Описание 8
2. Проектирование 8
3. Реализация 8
   1. Лексический анализатор 8
   2. Токен 9
   3. Токен-константа 9
   4. Токен-идентификатор 9
   5. Токен-оператор 9
4. Тестирование 10

Синтаксический и семантический анализ 16

1. Описание 16
2. Проектирование 16
3. Реализация 18
   1. Компилятор 18
   2. Синтаксический анализ 18
   3. Нейтрализация синтаксических ошибок 18
   4. Семантический анализатор 19
4. Тестирование 20
   1. Раздел описания переменных 20
   2. Раздел операторов 23

Генератор 27

1. Описание 27
2. Проектирование 27
3. Реализация 28
4. Тестирование 29

Проектирование компилятора

Компилятор — это программа, которая переводит программу на языке высокого уровня в эквивалентную программу на другом (объектном) языке. Обычно компилятор также выдает листинг, содержащий текст исходной программы и сообщения обо всех обнаруженных ошибках.

При вводе исходной программы и получении листинга мы имеем дело с конкретными устройствами ввода-вывода (клавиатура, ОП, экран, магнитные диски и т.д.). Чтобы легко адаптировать компилятор к различным внешним устройствам конкретной машины, отделим все действия по вводу-выводу данных от собственно процесса компиляции.

Модуль   
ввода-вывода

Компилятор

Исходная программа

Послед-ть

литер

Листинг

Объектная программа

Сам компилятор можно разделить на 4 основные части: лексический анализатор, синтаксический анализатор, семантический анализатор и генератор. Синтаксический и семантический анализаторы объединим для упрощения написания программы.

Послед-ть

литер

Листинг

Анализатор

Лексический анализатор

Синтаксический анализатор

Семантический анализатор

Послед-ть

символов

Доп. Информ.

Провер.

программа

Ошибки

Провер.

программа

Генератор

Объектная

программа

Для каждого блока будем использовать свой класс, в итоге определим 4 класса: ModuleIO, ModuleLexical, ModuleSyntax, ModuleGenerator.

ModuleIO

fileName

ModuleLexical

ModuleSyntax

ModuleGenerator

NextChar()

RecordError(ErrorInformation)

RecordError(ErrorInformation)

NextSymbol()

IdentifierTable

CommandList

Syntax

Semantic

Program

ErrorList

Listing

В ModuleIO основным методом будет NextChar, который будет возвращать следующую букву в программе. Также определим функцию RecordInformation для будущей записи ошибок.

В ModuleLexical определим метод NextSymbol, который будет возвращать информацию о текущем символе. Для этого он будет считывать символы из ModuleIO до тех пор, пока мы не сможем определить символ. Также модуль будет сохранять информацию о всех идентификаторах, которые были использованы в таблицу IdentifierTable.

ModuleLexical может возвращать только ошибки о несуществующих символах.

В ModuleSyntax основным методом будет Accept, который будет проверять, совпадает ли ожидаемый символ с вводимым.

ModuleSyntax будет проходить по всей программе и сохранять необходимую для генерации информацию, а также сохранять синтаксические и семантические ошибки.

ModuleGenerator будет использовать информацию, собранную на предыдущих шагах для того, чтобы генерировать объектный код.

Модуль ввода-вывода

# Описание

Модуль ввода-вывода отвечает за считывание программы и её посимвольную выдачу в лексический анализатор, и последующий вывод листинга программы. Также он будет сохранять информацию об ошибках.

# Проектирование

В данной программе модуль ввода-вывода будет считывать программу из файла, выдавать последовательно литеры в следующий модуль, принимать ошибки, а также выводить листинг программы и объектный код в файл.

ModuleIO

Имя файла

Ошибки

Ошибки

Listing.txt

Program.exe

Литеры

Объектная программа

# Реализация

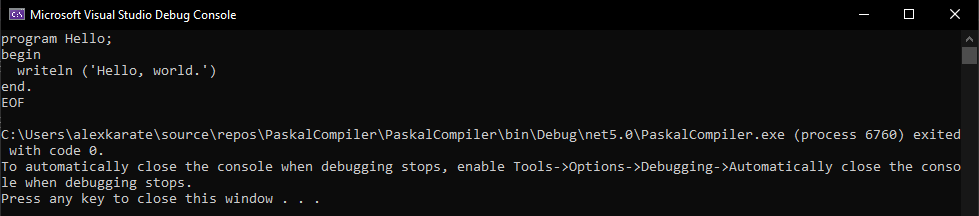
Вводить имя файла и открывать его будем через конструктор, а затем посимвольно выдавать текст программы благодаря методу NextChar.

Для оптимизации будем считывать из файла не один символ, а 1024 символов и сохранять их в буфер. Из файла будем считывать только по окончанию буфера.

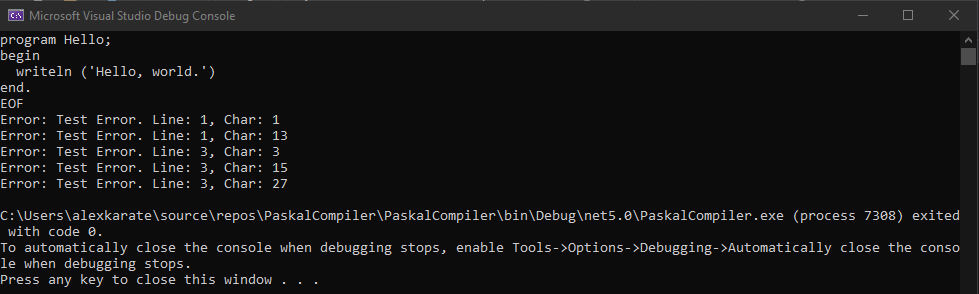
Дополнительно, определим класс Error, который будет хранить в себе информацию об ошибке, а также её позицию в коде. При вызове метода RecordError будем сохранять данную ошибку в списке.

Листинг будем создавать при помощи функции GenerateListing. Также создадим методы OutputFile, чтобы выводить файлы листинга и программы.

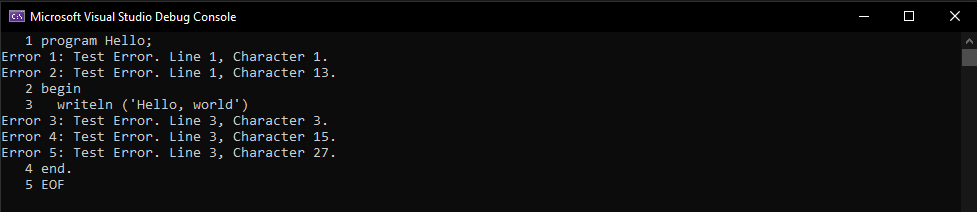
# Тестирование

Для тестирования модуля создадим простое консольное приложение, в котором будем выводить поочерёдно все символы из файла через метод NextChar.  
  


Файл выводится правильно.

Протестируем сохранение и вывод ошибок. Добавим код, который будет добавлять новую ошибку каждые 12 символов, а затем выведет их отдельно.  
  


Ошибки сохраняются. Создадим листинг данной программы.



Модуль ввода вывода работает корректно.

Лексический анализатор

# Описание

Лексический анализатор проводит лексический анализ программы. Он заключается в группировке литеров в символы, классификации данных символов и их последовательном выводе. Также лексический анализатор проверяет текст на невозможные литеры.

# Проектирование

Лексический модуль будет принимать литеры из предыдущего модуля, последовательно выдавать символы, и сохранять ошибки.

ModuleLexical

Литеры

Ошибки

Символы

Идентификатор, константа или операция

Определение символа будет происходить по мере считывания. По первому литеру будет определяться первоначальный тип символа. Затем, когда мы дойдём до литера, который не может входить в данный символ, мы окончательно определяем тип данного символа и выводим его.

# Реализация

Лексический модуль будет считывать литеры из модуля ввода-вывода и выводить последовательность символов при помощи метода NextSymbol.

При считывании первого литера, который может быть началом символа, добавляем его в строку currentSymbol и определяем predictedSymbol. Он может быть идентификатором, операцией или константой.

Если первый литер – буква, то тип является идентификатором, если цифра или литер «’», то константой, если один из разрешённых небуквенных символов, то операция. Иначе, записывается ошибка и пропускается данный литер.

Затем проверяется, может ли следующий литер входить в данный символ. Если нет, то идёт окончательное определение типа и сохранение дополнительной информации об символе.

# Токен

Создадим класс CToken для последующего вывода символов. Данный тип будет хранить в себе enum ETokenType, который будет однозначно определять тип данного токена.

Переопределим метод Equals, чтобы мы могли сравнивать два различных токена. Также переопределим метод ToString для последующего тестирования.

Определим 3 основных типа символов и соответствующие им классы: идентификатор (CIdent), операция (COperation) и константа (CValue), которые будут наследовать от основного класса CToken. Для каждого класса будем хранить дополнительную информацию.

# Токен-константа

Токен-константа имеет класс CValue, который наследуется от CToken. Также, в конструкторе класса поле ETokenType \_tt определяется, как константа.

Для токена-константы добавляется дополнительное поле EVarType \_vt, определяющий тип константы, а также object value, который определяет значение константы.

Также, как и для CToken, переопределяется ToString и Equals.

# Токен-идентификатор

Токен-идентификатор имеет класс CIdent, который наследуется от CToken. Аналогично, ETokenType \_tt определяется, как идентификатор.

Для идентификатора хранится поле identName, которая хранит в себе имя данного идентификатора.

В классе также переопределяется ToString и Equals.

# Токен-операция.

Токен-операция имеет класс COperation, который наследуется от CToken. Аналогично, ETokenType \_tt определяется, как идентификатор.

Для операции храниться поле EOperator \_vo, который хранит в себе код данной операции.

В классе определяются дополнительные методы, для определения типа операции. К ним относятся IsAdditive, который проверяет, является ли операция аддитивной, IsMultiplicative, который проверяет, является ли операция мультипликативной, а также IsRelative, который проверяет, является ли операция сравнивающей.

А также, аналогично, переопределяется ToString и Equals.

# Тестирование

Для тестирования модуля создадим консольную программу, которая будет выводить все символы в программе, а также сохранённые лексические ошибки.

Изначально протестируем все основные типы данных и констант.

Program Types;

var a:integer;

c:char;

cc: string;

d:real;

e:boolean;

begin

a:= -1-1;

c:= 'a';

cc:= 'aaa';

c := '''';

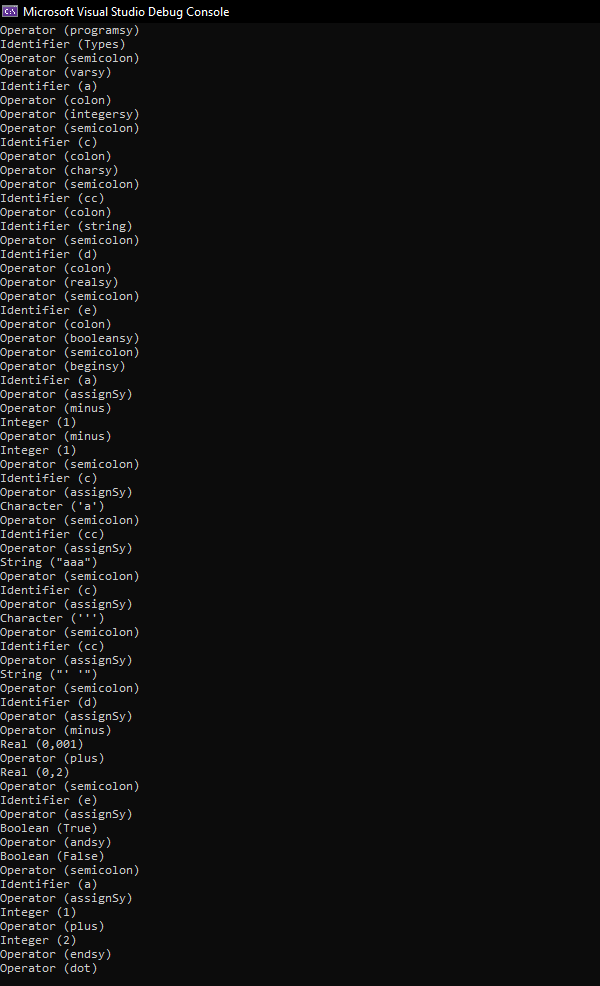
cc:= ''' ''';

d:= -1.0e-3 + 0.2;

e:= True and False;

a:=1+2

end.



Затем протестируем операторы if, составной оператор, и численные выражения.

Program Example;

var a,b:integer;

begin

a:=1+2;

b:=a div 3;

if b = 1 then

b := 3

else

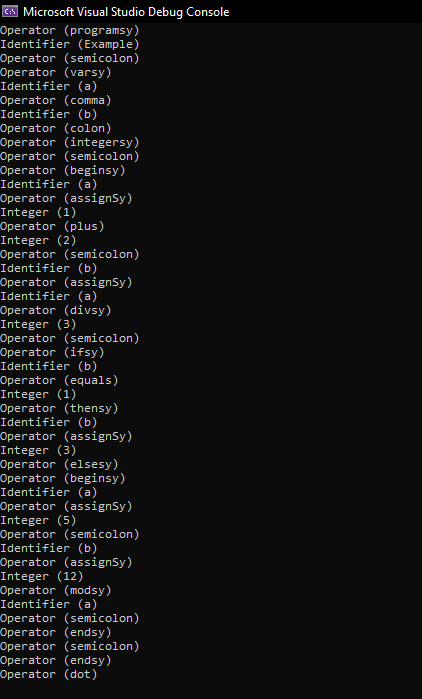
begin

a := 5;

b := 12 mod a

end

end.



Проверим цикл с предусловием (While) и выражения в скобках

Program Example;

var a,b:integer;

begin

a:=12;

b:=a div 3;

while b < a do

b:= (b + 1) \* 2

end.



Проверим ошибочные символы.

Program Example;

var a,b:&integer;

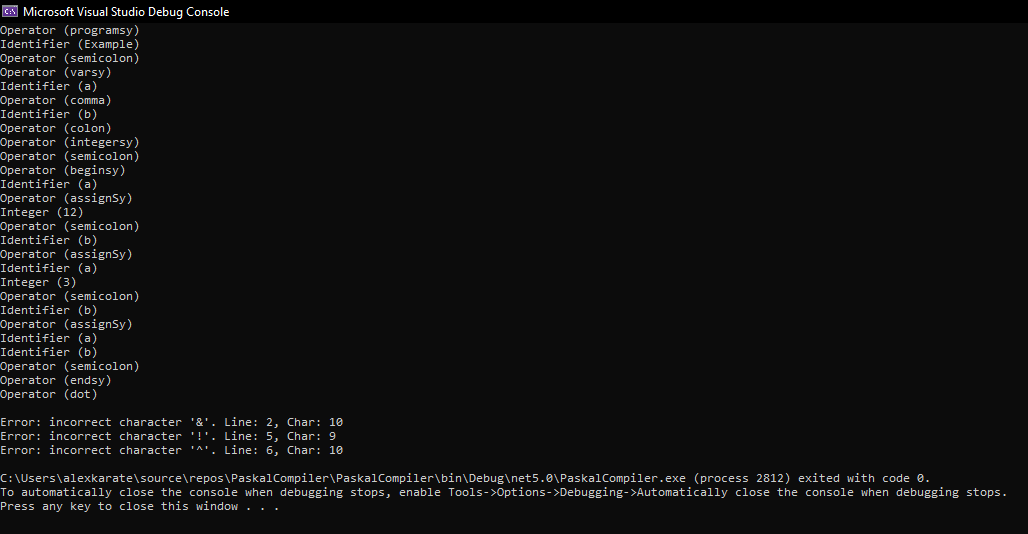
begin

a:=12;

b:=a ! 3;

b:= a ^ b;

end.



Лексический модуль работает корректно.

Синтаксический и семантический анализ

# Описание

Синтаксический и семантический анализ состоит в получении символов из предыдущего модуля, и последующей проверки данных символов на соответствие правилам языка.

В данном анализаторе, синтаксический анализ состоит в проверке того, что код написан на языке Паскаль, а также, что все синтаксические структуры (программа, область переменных, условные операторы, и так далее.) соответствуют данной структуре на языке Паскаль.

Семантический анализ состоит в проверке того, что программа семантически правильная, т.е. все идентификаторы были заранее объявлены, типы для выражений равны, либо один из них сводится к другому, а также допустимость операций над конкретными типами данных.

# Проектирование

Оба анализа будем производить в одном модуле. Данный модуль отличается от предыдущих тем, что у него нет последовательного вывода данных, он будет проводить анализ всей программы в файле, и выводить листинг программы.

ModuleSyntax

Символы

Ошибки

Листинг

Для проведения анализа будем вызывать методы, соответствующие синтаксическим конструкциям в языке Паскаль. Стандарт данного языка будем реализовывать по Формам Бэкуса-Натура для языка Паскаль.

<Программа>

Program

<Имя>

(<Имя ф-ла>)

;

<Блок>

<Раздел переменных> | <пусто>

<Раздел Операторов>

Var

,

<Описание однотипных переменных>;

<Составной оператор>

Begin

End

<Оператор>

.

;

…

…

Семантический анализатор будет реализоваться при помощи контекстов, в каждом из которых будем хранить таблицу типов и таблицу имён.

Для каждой программы создадим одинаковый фиктивный контекст, а также будем добавлять контексты при вызове функции. Это происходить не должно, поэтому, у нас есть только 2 контекста: фиктивный контекст и контекст программы.

# Реализация

Синтаксический модуль будет считывать символы из ModuleLexical и проверять, совпадает ли введённая программа со структурой программы на языке Paskal, а также проводить семантический анализ программы.

Основным методом в данном модуле будем считать метод CheckProgram, который будет запускать процесс анализа программы. В итоге в модуле будет сохранена необходимая информация о программе, а также список найденных ошибок. Также определим метод GenerateListing, который будет выводить данную информацию в виде листинга программы.

# Синтаксический анализатор

Основным методом синтаксического анализа является метод Accept, который проверяет, совпадает ли текущий символ с ожидаемым. Если был введён неверный символ, то кидается ошибка SyntaxException.

Процесс анализа запускается при вызове метода Program, которая определяет структуру всей программы на языке Paskal по его БНФ. Далее поочерёдно вызываются методы, соответствующие синтаксическим структурам языка.

В данном компиляторы были реализованы разделы переменных и операторов, под методами Variables и Operators.

Также были реализованы составной оператор (CompoundOperator), оператор присваивания, условный оператор (ChooseOperator), цикл с предусловием (PreLoopOperator).

Для работы данных операторов были реализованы выражения (Expression). Данные выражения могут состоять из базовых арифметических операций, операций сравнения, скобок, переменных и констант.

# Нейтрализация синтаксических ошибок

При выполнении метода CheckProgram, любая синтаксическая ошибка приводит к остановке анализа программы. Такое поведение допустимо, но предпочтительно продолжить анализ программы, даже если будут выводится ошибки, которые не существуют в программе.

Для продолжения анализа, будем добавлять операторы try-catch в ключевых точках программы, после которых мы предполагаем, что сможем продолжать анализ без проблем.

Если мы вошли в обработчик ошибок такого блока, то мы будем пропускать символы до тех пор, пока не найдём символ, при котором заканчивается данный блок или при котором начинается следующий.

Изначально, добавим обработчик ошибок в метод CheckProgram, чтобы отлавливать критические ошибки, после которых продолжать анализ нет возможности.

Далее, добавим обработчики ошибок для области объявления программы, а также для разделов переменных и операторов.

Также будем считать ключевыми точками вызов произвольного оператора и скобки в выражениях.

# Семантический анализатор

Семантический анализатор основывается на структуре, которую уже предоставил синтаксический анализатор. Нам остаётся только дополнить его функционал.

Для начала, создадим фиктивный контекст при помощи метода GenerateDefaultScope. Затем, добавим основной контекст для программы, который изначально пустой.

Затем, при выполнении раздела переменных, будем записывать все переменные в таблицу имён. Раздел типов у нас не реализуется, поэтому таблица типов в контексте программы будет пустым.

В разделе операторов будет проверяться 2 основные вещи: существование имени, соответствие вида использования имени и соответствие типов.

Существование имени осуществляется поиском соответствующего значения в таблице имён. Также, вместе с существованием проверяется соответствие вида применения имени, например, присвоить новое значение можно только переменной.

Если имя не существует, то выводим ошибку, а также добавляем данное имя как переменную в таблицу имён с неизвестным типом.

Соответствие типов проходит для каждого оператора по-своему.

Для оператора присваивания проверяем, что тип выражения справа приводится к типу переменной слева.

Для оператора выбора и цикла с предусловием проверяем, что выражение имеет булевый тип.

Для выражения мы имеет 2 типа: левый и правый тип (для типа результата слева и справа). Мы должны привести итоговое выражение к одному из данных типов. Если один из типов приводим к другому, то выводим наиболее общий тип.

Если один из типов в выражении – неизвестный, то возвращаем другой тип. Если оба типа в выражении неизвестные, то возвращаем неизвестный тип.

# Тестирование

Для тестирования программы будем создавать листинг проверочных программ и выводить их.

# Раздел описания переменных

Первоначально, создадим листинг программы без ошибок, в котором будут находиться все возможные типы для переменной.

1 Program Types;

2 var a:integer;

3 c:char;

4 cc: string;

5 d:real;

6 e:boolean;

7 begin

8 a:= -1-1;

9 c:= 'a';

10 cc:= 'aaa';

11 c := '''';

12 cc:= ''' ''';

13 d:= -1.0e-3;

14 e:= True;

15 end.

Создадим листинг программы, в которой неправильно написан тип одной из переменных.

1 Program Types;

2 var a:integer;

3 cc: strin;

Error 1: Could not find type strin. Line 3, Character 15.

4 c: char;

5 begin

6 a:= -1-1;

7 cc:= 'aaa';

8 c:= 'a';

9 end.

В данном тесте анализатор сохранил 3 переменные: ‘a’ до ошибки, ‘cc’ с неизвестным типом и ‘c’ после ошибки. Дальше, благодаря присваиванию на строке 7, переменной ‘cc’ был присвоен тип string.

Добавим новое присваивание для переменной ‘cc’ до данного оператора.

1 Program Types;

2 var a:integer;

3 cc: strin;

Error 1: Could not find type strin. Line 3, Character 15.

4 c: char;

5 begin

6 a:= -1-1;

7 cc := 2;

8 c := 'a';

9 cc:= 'aaa';

Error 2: Type et\_string is not derivable to et\_integer. Line 9, Character 14.

10 c:= 'a';

11 a:= 2;

12 end.

В список имён была добавлена переменная ‘cc’ с неизвестным типом, на строке 7 ей был присвоен тип integer, а затем ей была присвоена строка. Также, после первой ошибки анализ не был завершён.

Добавим операции с переменными, которые не были инициализированы.

1 Program Types;

2 var a:integer;

3 cc: strin;

Error 1: Could not find type strin. Line 3, Character 15.

4 c: char;

5 begin

6 a:= -1-1;

7 cc := 2;

8 c := 'a';

9 cc:= 'aaa';

Error 2: Type et\_string is not derivable to et\_integer. Line 9, Character 14.

10 c:= 'a';

11 a:= 2;

12 d:= 1e-5;

Error 3: Identificator d not found. Line 12, Character 5.

13 d:= d \* 2;

14 e:= True;

Error 4: Identificator e not found. Line 14, Character 5.

15 e:= False and e;

16 end.

Переменные d и e не были инициализированы, но их соответствующие типы были определены по результатам операции присваивания.

Создадим листинг программы, в котором есть ошибки в структуре раздела переменных.

1 Program Types;

2 var a, b, :integer;

Error 1: Expected Identifier, got Operator (colon). Line 2, Character 13.

3 c:char;

4 cc string;

Error 2: Expected Operator (colon), got Identifier (string). Line 4, Character 15.

5 d:real;

6 e:boolean

7 begin

Error 3: Expected Operator (semicolon), got Operator (beginsy). Line 7, Character 7.

8 a:= -1-1;

9 c:= 'a';

10 cc:= 'aaa';

Error 4: Identificator cc not found. Line 10, Character 6.

11 c := '''';

12 cc:= ''' ''';

13 d:= -1.0e-3;

14 e:= True;

15 end.

Во второй строке запятая была проигнорирована, так как после неё шёл символ двоеточия, после которого идёт тип.

В четвёртой строке, переменная ‘cc’ не была записана, так как в данной строке был нарушен порядок символов в определении типа.

В 6 строке не было точки с запятой, поэтому на 7 строке вышла ошибка о том, что анализатор ожидал точку с запятой.

При этом, анализ программы продолжился и вышла ошибка на 10 строке, так как переменная ‘cc’ не была записана.

# Раздел операторов

Создадим листинг программы без ошибок, в которой есть все необходимые операторы.

1 Program Example;

2 var a,b:integer;

3 begin

4 a:=1+2;

5 b:=a div 3;

6 if b = 1 then

7 b := 3

8 else

9 begin

10 a := 5;

11 b := 12 mod a;

12 end;

13

14 while b < a do

15 b:= (b + 1) \* 2;

16 a:=(a+b) div b + (a + b) \* 3 – 2;

17 end.

Уберём точки с запятой из некоторых операторов.

1 Program Example;

2 var a,b:integer;

3 begin

4 a:=1+2

5 b:=a div 3;

Error 1: Expected Operator (semicolon), got Identifier (b). Line 5, Character 5.

6 if b = 1 then

7 b := 3

8 else

9 begin

10 a := 5;

11 b := 12 mod a;

12 end;

13

14 while b < a do

15 b:= (b + 1) \* 2

16 a:=(a+b) div b + (a + b) \* 3 - 2;

Error 2: Expected Operator (semicolon), got Identifier (a). Line 16, Character 5.

17 end.

Если после оператора нет точки с запятой, или символа ‘end’, то анализатор пропускает все символы до точки с запятой или символа ‘end’.

Проверим программу, у которой нарушена структура операторов ‘if’ и ‘while’.

1 Program Example;

2 var a,b:integer;

3 begin

4 a:= 1;

5 b:= 2;

6 if b = 1

7 b := 3

Error 1: Expected Operator (thensy), got Identifier (b). Line 7, Character 8.

8 else

9 begin

10 a := 5;

11 b := 12 mod a;

12 end;

13 if b <> 1 then then

Error 2: Expected Operator (semicolon), got Operator (thensy). Line 13, Character 23.

14 b := 1;

15

16 while b < a

17 b:= (b + 1) \* 2;

Error 3: Expected Operator (dosy), got Identifier (b). Line 17, Character 8.

18 while a > b do do

Error 4: Expected Operator (semicolon), got Operator (dosy). Line 18, Character 21.

19 a := b - 1;

20 end.

Структура операторов then и if была нарушена, пропуском или дублированием символов then и do. При этом, анализ программы на этом не завершился.

Проверим программу, в которой составной оператор не имеет символа begin.

1 Program Example;

2 var a,b:integer;

3 begin

4 a:=1+2;

5 b:=a div 3;

6 if b = 1 then

7 begi

Error 1: Identificator begi not found. Line 7, Character 8.

8 b := 3;

Error 2: Expected Operator (assignSy), got Identifier (b). Line 8, Character 8.

9 end;

Error 3: Expected Operator (dot), got Operator (semicolon). Line 9, Character 7.

Error 4: More tokens than expected Line 9, Character 7.

10 a:= 12;

11 b:= 5;

12 a:= a div b;

13 end.

Анализатор считает, что мы закончили программу на строке 9, так как он нашёл символ ‘end’ после очередного оператора.

Проверим программу, в которой оператор не имеет символа end.

1 Program Example;

2 var a,b:integer;

3 begin

4 a:=1+2;

5 b:=a div 3;

6 if b = 1 then

7 begin

8 b := 3;

9 ending;

Error 1: Identificator ending not found. Line 9, Character 10.

Error 2: Expected Operator (assignSy), got Operator (semicolon). Line 9, Character 10.

10 a:= 12;

11 b:= 5;

12 a:= a div b;

13 end.

Error 3: Expected Operator (semicolon), got Operator (dot). Line 13, Character 5.

Error 4: Expected Operator (endsy), got Empty token. Line 13, Character 5.

Error 5: Expected Operator (dot), got Empty token. Line 13, Character 5.

В конце программы, анализатор не находит символов ‘end’ и ‘.’, а также находит символ ‘.’ ранее, чем ожидал.

Проверим программу, в которой выражение возвращает неправильный тип для оператора.

1 Program Example;

2 var a,b:integer;

3 begin

4 a:=1+2;

5 b:=a div 3;

6 a:=a + 0.5;

Error 1: Type et\_real is not derivable to et\_integer. Line 6, Character 14.

7 b:= 'c';

Error 2: Type et\_char is not derivable to et\_integer. Line 7, Character 11.

8 a:= True;

Error 3: Type et\_boolean is not derivable to et\_integer. Line 8, Character 12.

9 b:= 'string';

Error 4: Type et\_string is not derivable to et\_integer. Line 9, Character 16.

10

11 if b + 3 then

Error 5: Type et\_integer is not derivable to et\_boolean. Line 11, Character 17.

12 b := 3

13 else

14 begin

15 a := 5;

16 b := 12 mod a;

17 end;

18

19 while a + 'str' do

Error 6: Type et\_string is not derivable to et\_boolean. Line 19, Character 22.

20 b:= (b + 1) \* 2;

21 a:=(a+b) div b + (a + b) \* 3 - 2

22 end.

Проверим программу, у которой есть ошибки в выражении

1 Program Example;

2 var a,b:integer;

3 begin

4 a:= 1;

5 b:= 2;

6 a:=(a+b) div b + (a + b) \* 3 - 2;

7 b:= a+-b;

Error 1: Token Operator (minus) cannot be a part of expression. Line 7, Character 10.

8 a:= -b-a-;

Error 2: Token Operator (semicolon) cannot be a part of expression. Line 8, Character 13.

9 b:= a b;

Error 3: Expected Operator (semicolon), got Identifier (b). Line 9, Character 11.

10 b:= a \* b \* c;

Error 4: Identificator c not found. Line 10, Character 17.

11 end.

Генератор

# Описание

Генератор кода – часть компилятора, которая производит перевод программы из данного языка (Паскаля) в язык более низкого уровня или в машинный код.

Для данного компилятора мы будем использовать возможности .NET и переведём программу из языка Паскаль в Common Intermediate Language (CIL или MSIL) для последующего запуска при помощи Common Runtime Environment (CRE).

Генератор

Программа на языке Паскаль

Программа на языке CIL (MSIL)

CRE

Программа на машинном коде (в ОП)

# Проектирование

Для реализации генератора мы будем модифицировать уже существующий модуль синтаксического/семантического анализа. При этом, генератор кода будет работать до первой ошибки, и выводить в файл мы будем только при отсутствии ошибок в программе.

ModuleSyntax

Символы

Листинг

Program.exe

Для программы на языке MSIL будем создавать класс Program, в нём статический метод Main. Переменные в программе будут определены, как поля в классе Program.

Операторы будут поочерёдно перенесены в метод Main. Для этого будем добавлять в метод Main операции, которые будут иметь тот-же результат, что и на языке Паскаль. При этом также нужно проводить явные преобразования типов для выражений, включающих в себя различные типы.

Также добавим в компилятор возможность выводить значение в консоль через функцию writeln.

# Реализация

Чтобы проводить генерацию кода будем создавать объект встроенного класса AssemblyBuilder. Благодаря нему мы будем проводить вывод программы, а также создавать объекты других классов, с помощью которых можно проводить перевод программы.

В частности, данными классами является: ModuleBuilder, TypeBuilder, MethodBuilder и ILGenerator.

Для нашей программы, мы будем добавлять поля в TypeBuilder, добавим метод через MethodBuilder, определим его, как наш вход в программу, и будем выводить операции в этот метод при помощи ILGenerator.Emit.

Для упрощения работы, определим несколько методов для работы с ILGenerator.

EmitRelation будет выводить операции, связанные со сравнением двух значений.

EmitAdditive будет выводить операции сложения, вычитания, а также побитового ИЛИ.

EmitMultiplicative будет выводить операции, связанные с умножением, делением, а также побитовым И.

EmitConcat будет выводить конкатенацию двух значений в строку через метод String.Concat.

EmitWriteLine будет выводить в консоль значение через вызов метода Console.WriteLine.

EmitConvert будет проводить явное преобразование типов значений через вызов метода Convert.To<ТИП>.

Также определим методы CreateGenerator, который будет создавать объекты классов и OutputProgram, которая будет заканчивать создание типа Program, задавать метод Main, как вход в программу и сохранять программу в файл.

Для хранения переменных будем использовать статические поля в классе Program. Для этого будем использовать метод TypeBuilder.DefineField. Ссылку на данное поле храним в таблице имён, вместе с другой информацией об имени.

Выражения будут реализовываться через вышеописанные методы.

Оператор присваивания будет реализовываться через EmitConvert и OpCodes.Stsfld (Set static field).

Считывание переменных будет происходить через OpCodes.Ldsfld (Load static field).

Ввод констант будет проходить через OpCodes.Ldc или OpCodes.Ldstr (Load constant или Load string).

Операторы if и while будут реализовываться при помощи меток.

Оператор if определяет две метки: метку else, а также метку конца. Если выражение выдало значение true, то проходим один оператор и переходим к метке конца. Иначе, переходим к метке else, которая находится в блоке else, если он существует, либо в конце блока, если его не существует.

Оператор while также определяет две метки: метку начала и метку конца цикла. Если при очередном повторе выражение выдало значение true, то проходим один оператор и прыгаем в начало цикла. Иначе, прыгаем в метку конца цикла.

# Тестирование

Для тестирования программы, будем создавать .exe файл по Паскаль коду и проводить его выполнение при помощи программы cmd.exe.

Для начала, протестируем генерацию простейшей программы и вывод в консоль.

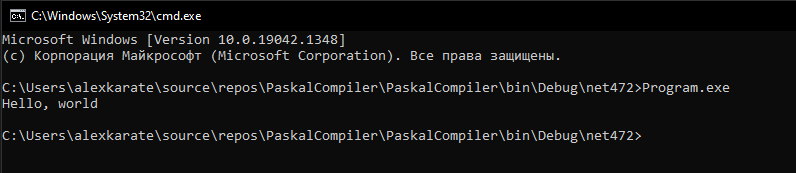
program Hello;

begin

writeln ('Hello, world')

end.

В результате, был создан файл Program.exe. При его запуске, в консоль было выведено:



Протестируем вывод в консоль различных типов данных.

Program Example;

begin

writeln('string');

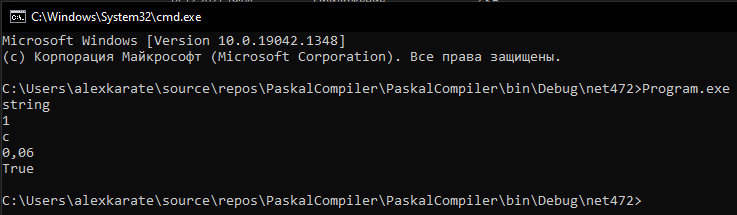
writeln(1);

writeln('c');

writeln(6e-2);

writeln(True);

end.



Протестируем работу выражений и переменных.

Program Types;

var a:integer;

c:char;

cc: string;

d:real;

e:boolean;

begin

a:= (-1 - 1) \* (5 mod 3);

writeln(a);

c:= 'c' - 'a' + 'A';

writeln(c);

cc:= a + ' a ' + a;

writeln(cc);

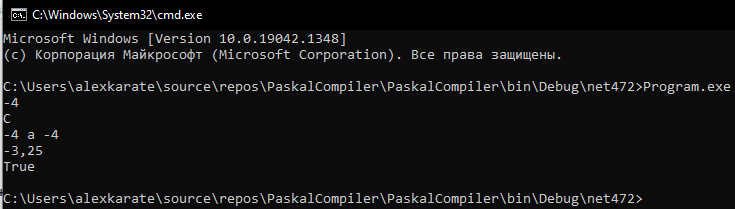
d:= 13 / a;

writeln(d);

e:= d > a;

writeln(e);

end.



Протестируем работу оператора if.

Program Example;

var a,b:integer;

begin

a:=2;

b:=10;

if b < a then

writeln(b + ' is less than ' + a);

else

writeln(b + ' is greater or equals to ' + a);

if b mod 3 = 0 then

begin

writeln(b);

writeln('is divisible by 3');

end;

else

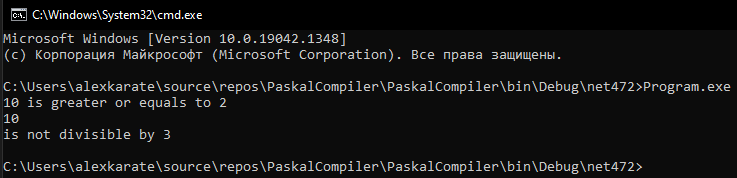
begin

writeln(b);

writeln('is not divisible by 3');

end;

end.



Протестируем работу оператора while

Program Example;

var a,b:integer;

begin

a:=2;

b:=100;

while a < b do

begin

writeln(a);

a:=a \* a;

end;

writeln(a + ' is greater than ' + b);

end.

