Пермский Государственный Национальный Исследовательский Университет

Формальные языки и методы трансляции

Отчёт

Паскар Алексей   
Владимирович   
ПМИ-2-2019

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, дисплей, магнитные диски). 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.exe

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.

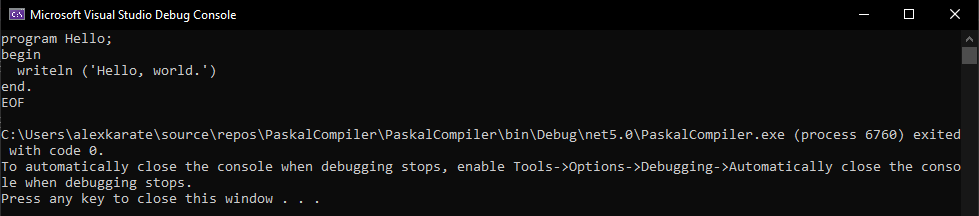
ModuleGenerator будет использовать информацию, собранную на предыдущих шагах для того, чтобы генерировать объективный код.

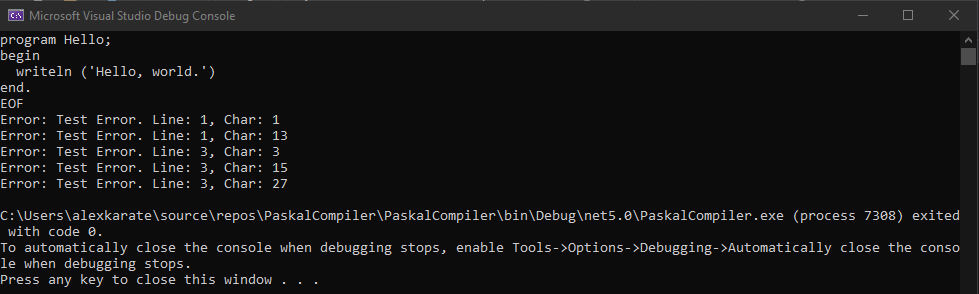
ModuleIO

Модуль ввода вывода будет считывать файл и, благодаря методу NextChar, посимвольно выводить текст программы.

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.

Определим класс Error, который будет хранить в себе информацию об ошибке, а также позицию в коде. При вызове RecordError будем добавлять новую ошибку в список и сохранять текущую позицию в тексте программы.

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

Добавим код, который будет добавлять новую ошибку каждые 12 символов.

Модуль ввода вывода работает корректно.

ModuleLexical

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

Определим 3 основных типа символов и соответствующие им классы: идентификатор (CIdent), операция (COperation) и константа (CValue), которые наследуют от вспомогательного класса CToken. Для символа будем однозначно хранить их тип при помощи enum ETokenType, а также дополнительную информацию для каждого типа.

По первому литеру в символе будем определять вид считываемого символа. Если очередной литер не может входить в данный символ, то сохраняем данный литер для следующего символа и окончательно определяем символ.

При этом, окончательный тип может отличаться от изначального. Например, резервированные слова изначально определяются как идентификаторы.

После определения символа возвращаем найденный символ. Если файл закончился, то возвращаем пустой символ.

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 := '''';

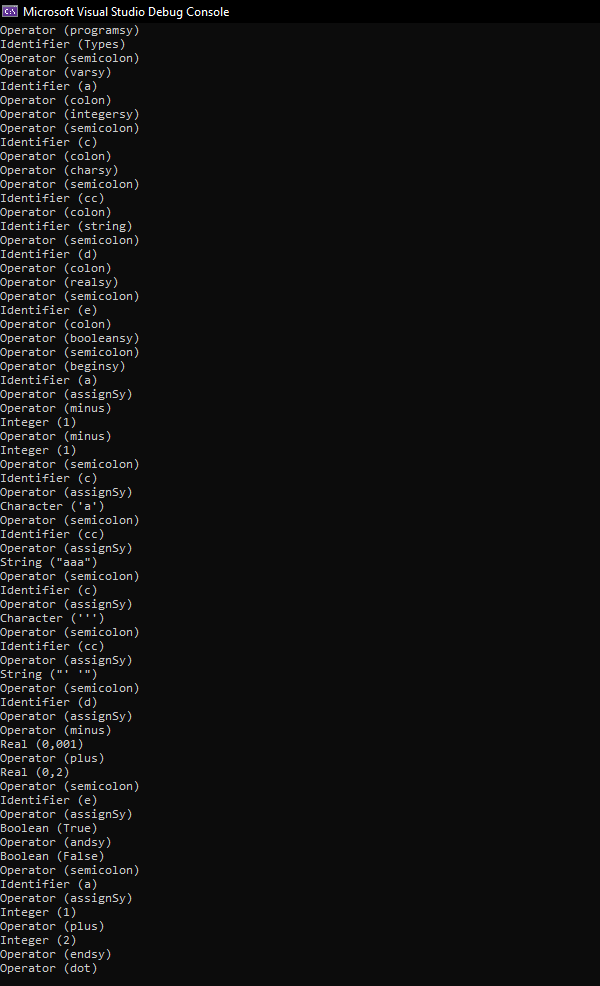
cc:= ''' ''';

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

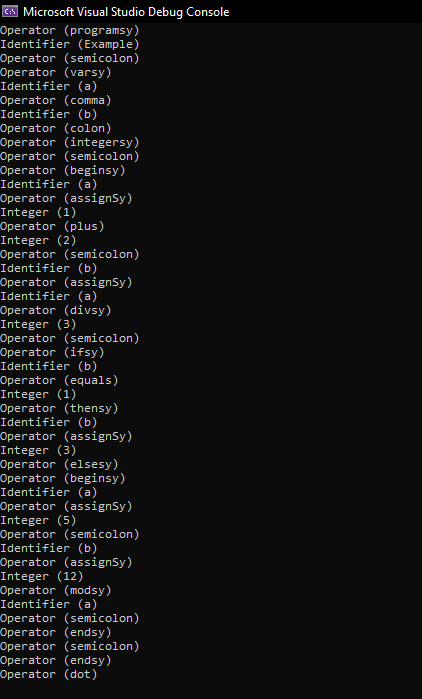
begin

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;

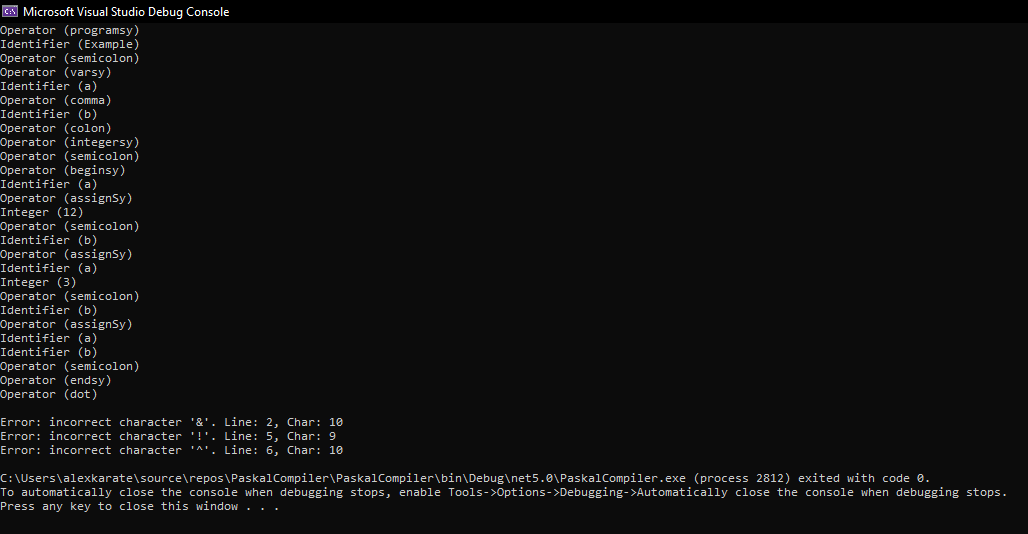
begin

a:=12;

b:=a ! 3;

b:= a ^ b;

end.



Лексический модуль работает корректно.