|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ  РОССИЙСКОЙ ФЕДЕРАЦИИ | | | |  |
|  | | Федеральное государственное бюджетное образовательное учреждение высшего образования  «Пермский государственный национальный исследовательский университет» | |  | |
| ОТЧЕТ  по практической работе  по дисциплине формальные грамматики и методы трансляции  «Разработка компилятора для языка Паскаль» | | | | | |
|  | | | Работу выполнил студент  группы ПМИ-1,2-2019 3 курса механико-математического факультета  \_\_\_\_\_\_\_\_ Балчиков В.И.  «\_\_\_» \_\_\_\_\_\_\_\_2021 г. | | |
| Проверил:  Ассистент кафедры  математического обеспечения  вычислительных систем  \_\_\_\_\_\_\_\_ Пономарёв Ф.А.  «\_\_\_» \_\_\_\_\_\_\_\_2021 г. | | |
|  | Пермь 2021 | | | |  |

Содержание

[Содержание 2](#_Toc90848543)

[Введение 4](#_Toc90848544)

[1 Модуль ввода-вывода 5](#_Toc90848545)

[1.1 Описание 5](#_Toc90848546)

[1.2 Проектирование 5](#_Toc90848547)

[1.3 Реализация 5](#_Toc90848548)

[2 Лексический анализатор 7](#_Toc90848549)

[2.1 Лексический анализатор 7](#_Toc90848550)

[1.1.2 Описание 7](#_Toc90848551)

[2.1.2 Проектирование 7](#_Toc90848552)

[3.1.2 Реализация 8](#_Toc90848553)

[2.2 Токены 9](#_Toc90848554)

[1.2.2 Проектирование 9](#_Toc90848555)

[2.2.2 Реализация 9](#_Toc90848556)

[2.3 Тестирование IO + Лексер 10](#_Toc90848557)

[3 Синтаксический анализатор 12](#_Toc90848558)

[3.1 Описание 12](#_Toc90848559)

[3.2 Проектирование 12](#_Toc90848560)

[3.3 Реализация 12](#_Toc90848561)

[3.4 Тестирование 13](#_Toc90848562)

[4 Семантический анализатор 16](#_Toc90848563)

[4.1 Описание 16](#_Toc90848564)

[4.2 Проектирование 16](#_Toc90848565)

[1.2.4 Общее описание 16](#_Toc90848566)

[2.2.4 Типы 16](#_Toc90848567)

[3.2.4 Идентификаторы 17](#_Toc90848568)

[4.2.4 Область видимости 17](#_Toc90848569)

[5.2.4 Модуль работы с областями видимости 18](#_Toc90848570)

[4.3 Реализация 18](#_Toc90848571)

[4.4 Тестирование 19](#_Toc90848572)

[5 Кодогенерация 23](#_Toc90848573)

[5.1 Описание 23](#_Toc90848574)

[5.2 Проектирование 23](#_Toc90848575)

[5.3 Реализация 23](#_Toc90848576)

[5.4 Тестирование 24](#_Toc90848577)

[Приложение B 26](#_Toc90848578)

Введение

В ходе курса формальные грамматики и методы трансляции мы постепенно знакомились с тем, как строится компилятор, из каких модулей он состоит и также получали теоретическую базу о грамматиках, языках.

И собственно главной задачей данного курса было не только понять теорию, но и попробовать самим реализовать компилятор для языка Паскаль. Конечно, не было необходимости писать полный компилятор, но нужно было познакомиться со всеми основными модулями компилятора и реализовать их.

Также в данном курсе было уделено внимание к самому процессу разработки. Так как компилятор сам по себе достаточно большая и объемная программа, то разработка велась последовательно. Каждый модуль компилятора разрабатывался в несколько этапов: проектирование, реализация и тестирование.

После нескольких месяцев разработки на выходе получился компилятор, который может совершать лексический, синтаксический, семантический анализ, находить ошибки в программе и конечно генерировать промежуточный код, который может быть выполнен CLR машиной. Перейдем к поэтапному и детальному рассмотрению всех модулей компилятора.

# Модуль ввода-вывода

## Описание

Модуль IO должен считывать исходные данные из файла и сохранять их в буфер и уже из буфера передавать литеры анализатору.

Также данный модуль отвечает за вывод ошибок. IO модуль должен иметь возможность принимать сообщения об ошибках из других модулей иметь интерфейс вывода.

## Проектирование

Модуль IO построен в виде отдельного класса IOModule.

Модуль должен уметь считывать входные файлы, работать со строками из данного файла и также работать с литерами, поэтому для класса были выделены следующие методы:

* получение текущей литеры
* получение позиции текущей литеры
* получение следующей литеры
* получение подстроки текущей строки
* добавление ошибки
* вывод всех ошибок

Данные методы являются публичными для того, чтобы другие модули в последствии могли обратиться к ним.

Также в классе содержатся следующие поля, необходимые для работы с файлом, строкой, литерой:

* поток чтения
* текущая строка
* номер текущей строки
* номер позиции в текущей строке
* перечислимый тип для хранения кодов ошибок
* словарь для хранения строк и ошибок в них

## Реализация

Модуль реализован следующим образом:

Модуль считывает исходную программу из файла построчно, одновременно сохраняя только одну строку. При вызове соответствующего метода возвращает литеру текущей строки анализатору. Если литеры текущей строки кончились, то модуль считывает следующую строку.

Модуль также занимается хранением и выводом ошибок, поэтому в нем присутствует отдельная структура данных, которая хранит ошибки. В качестве такой структуры данных я выбрал словарь, так как для одной строки может быть несколько ошибок из нескольких модулей. И сделано это так, что ключ словаря — это номер строки, а значение — это сама строка и список кодов ошибок для нее.

Вывод ошибок производится отдельным методом, в котором выводится строка, коды ошибок с пояснительным текстом. В конце выводится общее число найденных ошибок.

Коды ошибок представлены в виде перечисляемого типа CompilerError. Строки с пояснительной информацией ошибки хранятся в словарь, где ключ — это код ошибки, а значение – информация об ошибке.

# Лексический анализатор

## Лексический анализатор

### Описание

Задача лексического анализатора (сканера) формировать символы исходной программы и строить их внутреннее представление. Также лексический анализатор должен распознавать и исключать комментарии, которые не нужны для дальнейшей трансляции.

### Проектирование

Сканер реализован в виде отдельного класса Lexer.

Lexer содержит следующие методы и свойства:

* методы для игнорирования ненужных символов – пробел, табуляция, и комментариев
* метод для чтения целых, вещественных беззнаковых констант
* метод для чтения идентификатора и ключевого слова
* метод для получения токена
* метод для получения следующей литеры
* свойство для получения позиции текущей литеры
* свойство для получения текущей литеры

И Lexer содержит следующие поля:

* поле модуля IO
* словарь спец. символов

Выделив все необходимые методы и поля, можно описать, как все это все будет в целом взаимодействовать друг с другом. Для работы лексера, необходимо передать модуль IO, который будет считывать текст программы и посимвольно отдавать лексеру. Работа с лексером осуществляется через единственный публичный метод GetNextToken(). Для того, чтобы выделить лексему, необходимо сначала добраться для нее.

Метод ReadUnnecessary() занимается тем, что пропускает комментарии, литеры пробела, табуляции и все остальное, что не может являться частью лексемы и собственно позволяет добраться до лексемы.

Дальше GetNextToken() пытается определить лексему и в зависимости от полученных литер может вызвать методы ReadNum() или ReadIdentifier() для чтения целой константы или идентификатора соответственно.

И в итоге метод должен вернуть токен.

### Реализация

Работа модуля организована следующим образом:

При создании экземпляра лексического модуля, в конструкторе передается IO модуль.

Рассмотрим поля и свойства, необходимые для функционирования модуля в целом. В классе объявлены свойства, которые возвращают текущую позиции литеры в строке и саму литеру. Также объявлен метод для получения следующей литеры. Внутри свойства и методы обращаются к модулю IO для получения соответствующих значений.

Рассмотрим реализацию основного метода GetNextToken().

Метод для пропуска литер не нужных для последующего анализа обращается к методу ReadUnnecessary().

Метод ReadUnnecessary() устроен следующим образом. В цикле пропускаются все литеры, которые не могут являться частью лексемы. Это пробельные литеры, литеры табуляции, конца файла, литеры перевода каретки. Также метод пропускает комментарии, написанные в соответствии с правилами языка, которые не нужны для дальнейшего лексического анализа.

После этого можно перейти к обработке самих лексем. Обработка лексем устроена таким образом, что при получении первой литеры, нужно выбрать один из возможных способов дальнейшего анализа. Один из вариантов решения данной задачи это создание конечного автомата. Однако был выбран путь попроще – реализация через конструкцию "switch". Также для сканирования идентификаторов, целых и вещественных констант вызываются методы ReadIdentifier() или ReadNum() соответственно. Методы в цикле работают с литерами, пока не закончат анализ лексемы числовой константы или идентификатора. В итоге после сканирования лексемы на выходе метод GetNextToken() вернет токен.

Также стоит упомянуть то, что на пути лексера могут встретиться лексические ошибки, такие как: не закрытый мультистрочный комментарий, литера, которая не входит в список возможных литер языка, числовая константа, которая превышает максимально допустимое значение или просто числовая константа, написанная с ошибкой. В этом случае ошибка фиксируются с помощью модуля IO – происходит обращение к методу AddError(), в котором передается позиция ошибки и код ошибки из перечисляемого типа, описанного в модуле IO.

## Токены

### Проектирование

Токен представляет собой объект, созданный на основе лексемы во время лексического анализа. Токены могут быть различных типов, в зависимости от лексемы, которую они представляют. Все токены обладают некоторыми общими свойствами и при этом отличаются по смыслу, так как одним типам токенов нужно хранить лексему, другим нет. Поэтому удобно представить токены в виде иерархии классов, разбив их по смыслу – токены для идентификатора, спец. символа, констант и токен для всего остального.

### Реализация

Я реализовал это следующим образом. В качестве базового класса будет выступать абстрактный класс, содержащий одно поле – позиция начала лексемы.

Далее токены делятся на 4 класса, все из которых наследуются от базового.

* Первый класс IdentifierToken – предназначен для идентификаторов и содержит одно поле строкового типа для хранения имени идентификатора.
* Второй класс SpecialSymbolToken – предназначен для специальных символов и содержит одно поле для хранения идентификатора перечислимого типа спец. символов.
* Третий класс ConstToken – данный класс является обобщенным и предназначен для хранения констант различного типа (int, double, string, bool).
* Последний класс TriviaToken – предназначен для ошибочных токенов и также для токена конца файла, чтобы следующий модуль мог понять, когда чтение программы закончилось.

## Тестирование IO + Лексер

Так как работа модуля лексера напрямую связана с работой модуля ввода-вывода, то проведем тестирование двух модулей сразу.

Подадим на вход текст программы на языке Паскаль. И проверим корректность чтения файла модулем ввода-вывода, получение токенов из лексера и корректность их типов.

Текст программы:

program HelloWorld(output);

var s:string;

    b:real;

begin

    b:=2 \* 2 / 100 + 1 - 15;

    {comment section}

    b := 0.1234;

    s:= 's t';

    writeln('Hello, World!')

end.

В результате на выходе были получены следующие токены:

|  |  |
| --- | --- |
| **Tokentype** | **Value** |
| SpecialSymbolToken | ProgramToken |
| IdentifierToken | HelloWorld |
| SpecialSymbolToken | LeftRoundBracketToken |
| IdentifierToken | output |
| SpecialSymbolToken | RightRoundBracketToken |
| SpecialSymbolToken | SemicolonToken |
| SpecialSymbolToken | VarToken |
| IdentifierToken | s |
| SpecialSymbolToken | ColonToken |
| IdentifierToken | string |
| SpecialSymbolToken | SemicolonToken |
| IdentifierToken | b |
| SpecialSymbolToken | ColonToken |
| IdentifierToken | real |
| SpecialSymbolToken | SemicolonToken |
| SpecialSymbolToken | BeginToken |
| IdentifierToken | b |
| SpecialSymbolToken | AssignmentToken |
| Inttoken | 2 |
| SpecialSymbolToken | MultToken |
| Inttoken | 2 |
| SpecialSymbolToken | DivisionToken |
| Inttoken | 100 |
| SpecialSymbolToken | PlusToken |
| Inttoken | 1 |
| SpecialSymbolToken | MinusToken |
| Inttoken | 15 |
| SpecialSymbolToken | SemicolonToken |
| IdentifierToken | b |
| SpecialSymbolToken | AssignmentToken |
| Doubletoken | 0,1234 |
| SpecialSymbolToken | SemicolonToken |
| IdentifierToken | s |
| SpecialSymbolToken | AssignmentToken |
| Stringtoken | st |
| SpecialSymbolToken | SemicolonToken |
| IdentifierToken | writeln |
| SpecialSymbolToken | LeftRoundBracketToken |
| Stringtoken | Hello,World! |
| SpecialSymbolToken | RightRoundBracketToken |
| SpecialSymbolToken | EndToken |
| SpecialSymbolToken | DotToken |
| TriviaTokentoken | EndOfFileToken |

Можно убедиться, что файл считан верно – получены все ожидаемые токены различных типов, комментарии успешно пропущены.

Можно сделать вывод, что тестирование пройдено.

# Синтаксический анализатор

## Описание

Главная задач синтаксического анализатора проверить удовлетворяет ли программа формальным правилам.

Для описания формальных правил широко применяются формы Бэкуса-Наура, а также синтаксические диаграммы.

## Проектирование

Синтаксический анализатор реализован в виде отдельного класса SyntaxAnalyzer. Основная идея при анализе синтаксических конструкций будет заключатся в том, что для анализа каждого отдельного правила необходимо описать свой метод, который, собственно, будет анализировать данную конструкцию. Поэтому были выписаны все необходимые формальные правила в виде БНФ.

Работа синтаксического анализатора организована следующим образом. При запуске модуля, он начинает анализировать синтаксические конструкции языка. Анализ начинается с первой конструкции и далее идет по порядку. Анализ конструкций осуществляется по полученным лексемам в виде токенов. SyntaxAnalyzer работает напрямую с модулем лексического анализатора, из которого получает токены.

На этапе проектирование удалось выделить следующие поля для класса.

* поле для хранения экземпляра модуля IO
* поле для хранения экземпляра лексера
* поле для хранения текущего токена

Также практически для каждой БНФ выделен отдельный метод (не для всех БНФ, так как некоторые формы были объединены), перечислять их все не буду, так как можно ознакомиться с ними на диаграмме. И были выделены метод для проверки токена на заданный тип, метод получения следующего токена.

## Реализация

При создании экземпляра синтаксического анализатора в конструкторе передаются лексический модуль и модуль ввода-вывода.

В классе присутствует единственный публичный метод Start(), который запускает анализ. Метод получает первый токен и далее вызывает метод Program(), с которого и начинается анализ программы. Так как анализ различных конструкций практически одинаковый, рассмотрим несколько примеров для понимания работы. Program() анализирует конструкцию *<программа>::=program <имя>; <блок>*.

Сначала метод принимает ключевое слово program с помощью метода Accept().

Метод Accept(SpecialSymbolType symbolType) устроен следующим образом. Он принимает значение из перечислимого типа SpecialSymbolType и проверяет объект на класс SpecialSymbolToken и проверяет сходится ли текущее значение поля Type с ожидаемым значением, полученным в качестве аргумента. Если нет, то происходит обработка ошибки.

Метод имеет еще одну версию Accept<T>(), которая работает полностью аналогично, но проверяет является ли токен классом T.

Далее как раз с помощью второй версии метода Accept() проверяется является ли токен идентификатором, затем проверяется являются ли символ ';' и начинается анализ конструкции *<блок>*. После которой метод должен принять последний символ программы - точку.

Переходим к конструкции *<блок>*, которая состоит из 6 частей:

1. Раздел меток
2. Раздел констант
3. Раздел типов
4. Раздел переменных
5. Раздел процедур и функций
6. Раздел операторов

В моем случае реализованы "раздел переменных" и "раздел операторов". Для этих конструкций созданы соответствующие методы.

Рассмотрим анализ конструкции <раздел переменных>. Данная конструкция представлена следующей БНФ: *<раздел переменных>::= var <описание однотипных переменных>; {< описание однотипных переменных>; } | < пусто >*

Разбор данной конструкции был реализован через цикл do while, так как конструкция может содержать одно и более описание однотипных переменных.

Для *<описание однотипных переменных>* разработан отдельный метод.

Анализ остальных конструкция реализован схожим образом, поэтому перейдем к пункту с тестированием.

## Тестирование

Проверим корректность анализа различных конструкций:

* блок программы
* блок объявления переменных
* основной блок программы
* блок if else
* блок выражения
* блок цикла while

Подадим на вход следующий текст программы.

program TestProgram;

var

i, int1, int2, int3 :integer;

begin

    {if с веткой else}

    if real2 > 0 then begin

        r := 1.1;

    end

    else begin

        s := 'text from if statement';

    end;

    i:= ((81) - 18 \* (((8)) \* 59 - 14)) \* 0;

    while i < 10 do

    begin

        i:= i + 1;

    end;

end.

На выходе получим сообщение от компилятора, что ошибки не найдены. Данная программа полностью удовлетворят синтаксису языка Паскаль. Однако можно обратить внимание, что, например, в программе используется необъявленная переменная 'r', но ошибки не было. Данная ошибка относится к семантике программы, анализ которой рассмотрим в следующем разделе.

Попробуем нарушить некоторые конструкции, чтобы также проверить корректность работы синтаксического анализатора.

Пример 1:

{Забыли then в конструкции if then}

program TestProgram;

var

i, i2 :integer;

begin

    if i > 0 {then} begin

        i := 1;

    end

    else begin

        i := 2;

    end;

end.

Получим сообщение об ошибке:

Line 6:

if i > 0 {then} begin

Ожидался Then.

Пример 2:

{Оставили лишнюю запятую}

program TestProgram;

var

i, {i2} :integer;

begin

    i:=1000;

    writeln(i);

end.

Line 4:

i, {i2} :integer;

Ожидалась другая лексема

Пример 3:

{Забыли оператор присваивания}

program TestProgram;

var

i:integer;

begin

    i : 5;

    writeln(i);

end.

Line 6:

i : 5;

Ожидался оператор присваивания или оператор процедуры

# Семантический анализатор

## Описание

Главная задач семантического анализатора проверить удовлетворяет ли программа неформальным правилам языка.

Семантика работает с идентификаторами и типами. Ошибки, которые должен находить семантический анализатор: неописанные и повторно описанные идентификаторы, несоответствие типов, неправильный способ использования, ошибки с параметрами процедур и функций, с индексами массивов и т.п.

## Проектирование

### Общее описание

Данный модуль является встроенным в модуль синтаксического анализа. Семантика "встраивается" в уже реализованные функции, анализирующие синтаксические правила, добавляя проверку соответствующих неформальных правил.

Также семантическому модулю необходимо хранить информацию об идентификаторах:

* имя
* способ использования
* тип (ссылка на таблицу типов) – кроме процедур
* список параметров (для процедур и функций)

Хранить эту информацию необходимо в коллекции, позволяющей осуществлять быстрый поиск и добавление новых элементов.

Также необходимо хранить информацию о типах. Необходимо реализовать функционал проверки операций над различными типами, приводимости типов друг к другу. Типы и идентификаторы должны храниться в соответствующих им областях видимости.

### Типы

Для начала разберемся с тем, как хранить информацию о типах и их приводимости друг к другу. В нашем случае есть несколько стандартных типов в языке: integer, real, string, boolean. Все эти типы связаны тем, что надо, собственно, хранить информацию о типе и о приводимости. Логично создать абстрактный класс, от которого уже другие типы буду наследоваться.

Абстрактный класс содержит:

* readonly поле перечислимого типа для хранения типа (readonly так как тип не может измениться на ходу)
* абстрактный метод булевого типа, который отвечает за то, приводим ли данный тип к типу, переданному в аргументе метода

Остальные классы, которые наследуются от базового будут иметь свою реализацию метода и будут иметь отличные значения в поле типа.

### Идентификаторы

Далее необходимо разобраться с информацией об идентификаторах. Самое основное, что необходимо хранить об идентификаторе – это его назначение, тип. Для этого подойдет класс, от которого, можно будет унаследоваться в будущем при необходимости, например, для хранения информации о функциях, которые могут иметь один и более аргументов со своими типами.

Назначение идентификатора хранится в соответствующем поле перечислимого типа. Для типа, описанного ранее также предусмотрено поле в классе.

### Область видимости

Также необходимо позаботиться о хранении областей видимости в программе. Так их может быть несколько целесообразно описать класс для хранения информации о текущей области видимости. Необходимо хранить информацию о типах в данной области видимости – для этой задачи подойдет коллекция – список, так нет никаких строгих требований. Далее эту структуры данных буду обозначать как таблица типов (ТТ).

Для хранения информации об идентификаторах нужна коллекция с быстрым доступом, так как в ходе семантического анализа часто приходится обращаться к различным идентификаторам. Тут можно использовать различные структуры данных: сбалансированные деревья, нагруженные деревья, ассоциативный массив. Я остановился на последнем в виде структуры Dictionary в C#. Далее эту структуры данных буду обозначать как таблица идентификаторов (ТИ).

Описанные структуры ТТ и ТИ являются недоступными из вне, работа с ними осуществляется с помощью методов:

- добавить идентификатор

- добавить тип

- проверить содержится ли данный идентификатор в данной области видимости

- найти информацию об идентификаторе

### Модуль работы с областями видимости

И конечно же необходим модуль для работы с несколькими областями видимости. Модуль хранит области видимости в виде стека. Доступ к текущей области видимости осуществляется через соответствующее свойство. Работа с приватными полями ведется с помощью методов:

* добавление идентификатора
* проверка доступности идентификатора в областях видимости
* получение информации о заданном идентификаторе

## Реализация

Рассмотрим некоторые детали реализации всего описанного в проектирование.

В моем случае я описал следующие типы Паскаля в перечислимом типе: Integer, Real, String, Boolean.

Классы IntType, RealType, StringType, BooleanType, наследующиеся от абстрактного класса CType передают в конструктор базового класса соответствующие им типы Integer, Real, String, Boolean. Что касается метода проверки приводимости типов IsCastedTo, то проверка в нем реализована на конструкции switch. Единственный стандартный тип, который можно неявно привести к другому в Паскале – это integer к real, остальные типы являются не приводимыми друг к другу.

Способы использования идентификаторов описаны в перечислимом типе, среди них:

* константа
* переменная
* тип
* процедура
* функция

Модуль работы с областями видимости (ScopeManager).

При создании ScopeManager инициализируются таблица областей видимости, и создается первая, так называемая, фиктивная область видимости. В данную область видимости заносится информация о стандартных типах языках Паскаль.

Также в методах добавления идентификатора в текущую область видимости происходит предварительная проверка на случай дубликата. Если же данный идентификатору уже существует, то фиксируется ошибка. Также ошибка фиксируется при попытке получить информацию о несуществующем идентификаторе.

Для того, чтобы семантический модуль мог полноценно функционировать – модифицируем класс SyntaxAnalyzer.

Одно из главных изменений – это то, что теперь методы, работающие с выражениями, возвращают типы для того, чтобы можно было проверять корректность операций и приводить, если необходимо.

Добавим приватное поле для ScopeManager. Также добавим методы для приведения типов. В местах объявления типов и переменных допишем добавление типов и переменных в текущую область видимости (в данной реализации область одна, не считая фиктивной).

И заключительная и не менее важная часть – это проверка типов при различных операциях. Для этого я создал три метода, проверяющих аддитивные, мультипликативные операции и операции отношения. Эти методы проверят возможно ли для левого и правого аргументов различных типов совершить ту или иную операцию и по возможности привести к одному типу.

## Тестирование

Проведем тестирование семантического модуля.

Для начала подадим на вход семантически корректную программу.

Проверим совместимость типов констант, переменных в выражении. Проверим также все ли переменные объявлены перед их использованием.

program TestProgram;

var

i, int1, int2, int3 :integer;

r, a, real1, real2, real3: real;

s, str1, str2, str3 :string;

b, bool1, bool2, bool3 :boolean;

begin

    i:=1;

    if i > 0 then begin

        writeln('if statement');

    end

    else begin

        s := 'else statement';

        writeln(s);

    end;

    if 100 > 0 then begin

        writeln('if №2');

    end;

    i := 0;

    while i < 100 do

    begin

        writeln(i);

        i:= i + 1;

    end;

    {Тесты на совместимость типов в выражении}

    {int, int}

    i:= 1 + 2222;

    i:= 2 - 2222;

    i:= 3 \* 2222;

    i:= 4 / 2222;

    i:= 5 mod 2;

    i:= 6 div 3;

    writeln(i);

    writeln('hello world');

    {real, real}

    r:= 1.111 + 1.111;

    r:= 2.111 - 1.111;

    r:= 3.8 \* 3.5;

    r:= 7.4 / 3.7;

    writeln(r);

    {real, int}

    r:= 1.1 + 2;

    r:= 2.1 - 2;

    r:= 3.1 \* 2;

    r:= 4.1 / 2;

    {bool, bool}

    b:= true or false;

    b:= false and false;

    b:= 1 > 111;

    b:= 99999 >= 2222;

    b:= 3 < 33333;

    b:= 4 <= 444444;

    b:= 5 = 555555;

    b:= 6 <> 666666;

    b:= 'text 1' = 'text 2';

    writeln(b);

    {string, string}

    s:= 'Hello, ' + 'world!!!';

    writeln(s);

    {Сложные выражения bool, real}

    b:= ((13 > 10) = true) and ( ((13 \* 19) / 3) < 13 ) or (( 4 mod 2 ) = 0);

    s:= s + 'second string';

    r:= 1.999 - (13 \* 2 / 7 - 8) \* (11.1 - 2) + (114 div 3);

    i:= (55) - (54) \* 55 + 92 - 13 - ((36)); {-2872}

end.

В результате программа успешно считывается и анализируется.

Теперь проверим корректность нахождения ошибок.

На вход подадим следующую программу с ошибками типов.

program TestProgram;

var

i, int1, int2, int3 :integer;

r, a, real1, real2, real3: real;

s, str1, str2, str3 :string;

b, bool1, bool2, bool3 :boolean;

begin

    {Не приводимые типы}

    i:= 'text';

    i:= false;

    s:= 10;

    s:= true;

    b:= 'text';

    b:= 100;

    i:= 1 + 't';

    b:= (10 - 11) > 'sample text';

    b:= ((13 > 10) = true) \* ( ((13 \* 19) / 3) < 13 );

    s:= s + i;

    r:= s + i;

end.

В результате все ошибки найдены:

Line 9:

    i:= 'text';

Ожидался Integer

Line 10:

    i:= false;

Ожидался Integer

Line 11:

    s:= 10;

Ожидался String

Line 12:

    s:= true;

Ожидался String

Line 14:

    b:= 'text';

Ожидался Boolean

Line 15:

    b:= 100;

Ожидался Boolean

Line 17:

    i:= 1 + 't';

Операция + не допустима для типов: Integer, String

Line 18:

    b:= (10 - 11) > 'sample text';

Операции <, <=, >, >= не допустимы для типов: Integer, String

Line 20:

    b:= ((13 > 10) = true) \* ( ((13 \* 19) / 3) < 13 );

Операции \*, / не допустимы для типов: Boolean, Boolean

Line 22:

    s:= s + i;

Операция + не допустима для типов: String, Integer

Line 23:

    r:= s + i;

Операция + не допустима для типов: String, Integer

И еще одна программа для проверки несуществующих идентификаторов.

program TestProgram;

var

i, int1, int2, int3 :integer123;

begin

    newVar := 10;

end.

Идентификатор с именем integer123 не существует.

Идентификатор с именем newVar не существует.

# Кодогенерация

## Описание

Задача генератора кода преобразовать синтаксически корректную программу в последовательность инструкций, которые могут выполняться на машине. На выходе возможно несколько различных вариантов

* машинный код
* программа на языке ассемблера
* объектные код на промежуточном языке (например, MSIL, Java bytecode)

## Проектирование

Для кодогенерации был выбран промежуточный язык Common Intermediate Language (CIL), разработанный Microsoft для платформы .NET. Таким образом анализируя код программы на языке Паскаль, мы будем переводить его в промежуточный язык MSIL и сохранять в виде сборки .NET. Эту сборку возможно выполнять прямо на ходу или сохранить в DLL или EXE.

Для перевода программы на MSIL необходимо добавлять соответствующие команды IL на стек. Корректное добавление всех команд позволит сохранить и в дальнейшем выполнить программу.

Для работы с IL будем использовать пространство имен Reflection.Emit.

Все, что нам осталось, чтобы добавить генерацию кода – это добавить еще раз модифицировать методы SyntaxAnalyzer и дописать в нужных местах добавление соответствующих команд на стек IL команд и затем сохранить сборку.

Для генерации кода создадим отдельный класс CodeGenerator, который позволит хранить все данные для работы с текущей сборкой. Класс генератора кода содержит методы для добавления различных команд на стек: создание переменной, операция присвоения, добавления константы, добавление операции, печать в консоль и т.д. Также в CodeGenerator предусмотрены методы для работы с самой сборкой – сохранение, запуск сборки.

Помимо этого, нужно сохранять адреса добавленных переменных, для этого необходимо добавить дополнительное поле в класс IdentifierInfo.

## Реализация

Что касается реализации, то класс CodeGenerator не является чем-то примечательным, он скорее служит удобной оберткой для работы с Reflection.Emit.

Самое важное – это модификация методов SyntaxAnalyzer. Таким образом в метод VariableDeclaration() обработки блока *<описание однотипных переменных>* был добавлен вызов метода добавления переменной на стек MSIL кода - DeclareVar(); и также сохранение адреса данной переменной. В метод анализа конструкции *<множитель>* добавлены вызовы методов для добавления различных константы на стек. В блоке с операциями – вызов метода для добавления операции на стек; в блоке присвоения значений – вызов метода кодогенератора для добавления операции присвоения на стек и аналогично для остальных операций.

## Тестирование

Подадим на вход корректную программу и запустим ее.

program TestProgram;

var

i, int1, int2, int3 :integer;

r, a, real1, real2, real3: real;

s, str1, str2, str3 :string;

b, bool1, bool2, bool3 :boolean;

begin

    if r > -1.0 then begin

        writeln('if statement');

    end

    else begin

        s := 'else statement';

        writeln(s);

    end;

    writeln('after if №1 statement');

    if 100 > 0 then begin

        writeln('if №2');

    end;

    writeln('after if №2 statement');

    if -100 < 0 then begin

        writeln('this message should not be displayed');

    end;

    writeln('after if №3 statement');

    i := 0;

    while i < 10 do

    begin

        writeln(i);

        i:= i + 1;

    end;

    writeln('after while' + ' statement');

    {Сложные выражения bool, real}

    b:= ((13 > 10) = true) and ( ((13 \* 19) / 3) < 13 ) or (( 4 mod 2 ) = 0);

    writeln(b);

    s:= 's + ';

    s:= s + ' string 2';

    writeln(s);

end.

На выходе при запуске программы получим следующий вывод в консоль.

[Debug] Program starts from dll

[Debug] Program output:

if statement

after if №1 statement

if №2

after if №2 statement

this message should not be displayed

after if №3 statement

0

1

2

3

4

5

6

7

8

9

after while statement

True

s +  string 2

[Debug] Program finished

Кол-во ошибок: 0

Можно убедиться, код генерируется корректно.

Приложение B

Диаграмма классов компилятора в PlantUML.

К сожалению, из-за размера рисунок не помещается, однако его можно посмотреть в репозитории проекта.

И так блок рассказа про построение диаграмм некуда поместить, расскажу прямо здесь. Мне не удалось в короткий срок найти качественных средств для построения диаграмм классов для проекта на C#, поэтому пришлось воспользоваться расширением для Visual Studio Code «CSharp to PlantUML», которое позволяет строить PlantUML диаграммы для проектов на C#. Далее с помощью инструментария официального сайта PlantUML диаграммы преобразовались в рисунки формата PNG (однако есть и SVG). Диаграммы получились неидеальные, есть пересечения, слова местами налезают друг на друга, тем не менее диаграммы позволяют увидеть цельную картину проекта.