|  |  |
| --- | --- |
| МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ | |
| Федеральное государственное бюджетное образовательное учреждение высшего образования  «Пермский государственный национальный исследовательский университет» | |
| **Формальные грамматики и методы трансляции** *Лабораторная работа №1*  **«Разработка компилятора для языка Pascal»** | |
|  | |
|  | Работу выполнил студент группы ПМИ-2:  Писцов Сергей |
|  | Проверил:  Ассистент кафедры математического обеспечения вычислительных систем  Ф.А. Пономарёв  “\_\_\_\_” 2021 г. |
| Пермь 2021 | |

Оглавление

[Задание 4](#_Toc90773668)

[Структура компилятора 5](#_Toc90773669)

[Модуль ввода-вывода 6](#_Toc90773670)

[Описание 6](#_Toc90773671)

[Проектирование 7](#_Toc90773672)

[Реализация 8](#_Toc90773673)

[Тестирование 9](#_Toc90773674)

[Вывод всех литер подряд 9](#_Toc90773675)

[Лексический анализ 13](#_Toc90773676)

[Описание 13](#_Toc90773677)

[Проектирование 13](#_Toc90773678)

[Реализация 15](#_Toc90773679)

[Тестирование 16](#_Toc90773680)

[Вывод всех лексем подряд 16](#_Toc90773681)

[Вывод ошибки о незаконченном комментарии 17](#_Toc90773682)

[Вывод ошибки о незаконченной строке/литере 18](#_Toc90773683)

[Вывод ошибки о неправильной форме вещественной константы 19](#_Toc90773684)

[Вывод ошибки о непредвиденном символе 20](#_Toc90773685)

[Синтаксический анализатор 22](#_Toc90773686)

[Описание 22](#_Toc90773687)

[Проектирование 22](#_Toc90773688)

[Тестирование 28](#_Toc90773689)

[Проверка ошибки при описании переменных 28](#_Toc90773690)

[Проверка ошибки на наличие «;» 28](#_Toc90773691)

[Проверка ошибки в условном операторе 28](#_Toc90773692)

[Проверка ошибки в операторе цикла 29](#_Toc90773693)

[Проверка ошибки в арифметическом выражении 29](#_Toc90773694)

[Проверка ошибки в строковом выражении 29](#_Toc90773695)

[Проверка ошибки в логическом выражении 30](#_Toc90773696)

[Семантические анализатор 31](#_Toc90773697)

[Описание 31](#_Toc90773698)

[Проектирование 31](#_Toc90773699)

[Реализация 32](#_Toc90773700)

[Тестирование 34](#_Toc90773701)

[Проверка ошибки неопределенного идентификатора 34](#_Toc90773702)

[Проверка ошибки на повторное объявление идентификатора 34](#_Toc90773703)

[Проверка ошибки на несоответствие типов арифметической операции 34](#_Toc90773704)

[Проверка ошибки на несоответствие типов при присваивании 35](#_Toc90773705)

[Проверка ошибки на несоответствие логических операндов 35](#_Toc90773706)

Задание

Разработать компилятор для программ на языке Pascal. Язык программирования, на котором должен быть написан компилятор, должен быть объектно-ориентированным.

Структура компилятора

Компиляция состоит из следующих фаз: модуль ввода-вывода, анализатор, генератор. Анализатор, в свою очередь, состоит из 3 модулей: лексический, синтаксический и семантический анализаторы. Весь компилятор принимает следующую структуру:

Модуль ввода-вывода получает исходный код программы и преобразует в последовательность литер

Лексический анализатор осуществляет лексический анализ, строит идентификаторы, ключевые слова, разделители, числа

Синтаксический анализатор проверяет, удовлетворяет ли программа формальным правилам

Семантический анализатор проверяет смысловую правильность программы

Генератор генерирует объектную программу

Модуль ввода-вывода

Описание

Модуль ввода-вывода предназначения для считывания символов из исходной программы. Также модуль включается в себя вывод позиции символа в текущий момент считывания: порядковый номер символа, номер строки, номер символа в строке.

Проектирование

Модуль ввода-вывода содержит класс IO

public IO(string path).

В конструктор класса IO передается путь к файлу с текстом программы на языке Pascal, в строковое поле ProgramText.

public string ProgramText { get; private set; }

считывается текст всей программы построчно. Из модуля ввода-вывода лексический анализатор берет очередной символ, для этого есть метод. В данном методе обновляется позиция символа по порядку, по номеру строки и позиция в самой строке

public char Nextch()

Для проверки лексем с длиной более 1 необходимо брать следующий символ. В случае, если он не подошел к лексеме, нужно вернуться на символ обратно, для этого есть метод

public void Back()

Следующие поля хранят соответствующие значения состояния каретки:

public int Line\_Number { get; set; }

public int Line\_Position { get; set; }

public int Last\_Line\_Number { get; set; }

public int Last\_Line\_Position { get; set; }

public int Count { get; set; }\

Реализация

Метод Nextch() последовательно считывает символы, обновляя количество строк при достижении символа переноса. Чтение заканчивается, если текущая позиция достигла длины текста всей программы. Также метод обновляет значения позиции предыдущего символа

Для возврата каретки в начало программы метод Set\_Start() зануляет поля, вроде номер строки, позиция в строке

Для отката символа на предыдущий метод Back() возвращает значение, позицию прошлого символа.

Тестирование

Вывод всех литер подряд

Текст исходной программы:

**program** showRelations;

**var**

a, b: integer;

**begin**

a := 21;

b := 10;

**if** a = b **then**

writeln('Line 1 - a is equal to b' )

**end**.

Результат работы модуля ввода-вывода, включая все невидимые символы табуляции:

Value: p| Position: 0| Line: 2| Count: 1

Value: r| Position: 0| Line: 3| Count: 2

Value: o| Position: 0| Line: 4| Count: 3

Value: g| Position: 0| Line: 5| Count: 4

Value: r| Position: 0| Line: 6| Count: 5

Value: a| Position: 0| Line: 7| Count: 6

Value: m| Position: 0| Line: 8| Count: 7

Value: | Position: 0| Line: 9| Count: 8

Value: s| Position: 0| Line: 10| Count: 9

Value: h| Position: 0| Line: 11| Count: 10

Value: o| Position: 0| Line: 12| Count: 11

Value: w| Position: 0| Line: 13| Count: 12

Value: R| Position: 0| Line: 14| Count: 13

Value: e| Position: 0| Line: 15| Count: 14

Value: l| Position: 0| Line: 16| Count: 15

Value: a| Position: 0| Line: 17| Count: 16

Value: t| Position: 0| Line: 18| Count: 17

Value: i| Position: 0| Line: 19| Count: 18

Value: o| Position: 0| Line: 20| Count: 19

Value: n| Position: 0| Line: 21| Count: 20

Value: s| Position: 0| Line: 22| Count: 21

Value: ;| Position: 0| Line: 23| Count: 22

| Position: 0| Line: 24| Count: 23

Value:

| Position: 1| Line: 1| Count: 24

Value: v| Position: 1| Line: 2| Count: 25

Value: a| Position: 1| Line: 3| Count: 26

Value: r| Position: 1| Line: 4| Count: 27

| Position: 1| Line: 5| Count: 28

Value:

| Position: 2| Line: 1| Count: 29

Value: a| Position: 2| Line: 2| Count: 30

Value: ,| Position: 2| Line: 3| Count: 31

Value: | Position: 2| Line: 4| Count: 32

Value: b| Position: 2| Line: 5| Count: 33

Value: :| Position: 2| Line: 6| Count: 34

Value: | Position: 2| Line: 7| Count: 35

Value: i| Position: 2| Line: 8| Count: 36

Value: n| Position: 2| Line: 9| Count: 37

Value: t| Position: 2| Line: 10| Count: 38

Value: e| Position: 2| Line: 11| Count: 39

Value: g| Position: 2| Line: 12| Count: 40

Value: e| Position: 2| Line: 13| Count: 41

Value: r| Position: 2| Line: 14| Count: 42

Value: ;| Position: 2| Line: 15| Count: 43

| Position: 2| Line: 16| Count: 44

Value:

| Position: 3| Line: 1| Count: 45

Value: b| Position: 3| Line: 2| Count: 46

Value: e| Position: 3| Line: 3| Count: 47

Value: g| Position: 3| Line: 4| Count: 48

Value: i| Position: 3| Line: 5| Count: 49

Value: n| Position: 3| Line: 6| Count: 50

| Position: 3| Line: 7| Count: 51

Value:

| Position: 4| Line: 1| Count: 52

Value: | Position: 4| Line: 2| Count: 53

Value: a| Position: 4| Line: 3| Count: 54

Value: | Position: 4| Line: 4| Count: 55

Value: :| Position: 4| Line: 5| Count: 56

Value: =| Position: 4| Line: 6| Count: 57

Value: | Position: 4| Line: 7| Count: 58

Value: 2| Position: 4| Line: 8| Count: 59

Value: 1| Position: 4| Line: 9| Count: 60

Value: ;| Position: 4| Line: 10| Count: 61

| Position: 4| Line: 11| Count: 62

Value:

| Position: 5| Line: 1| Count: 63

Value: | Position: 5| Line: 2| Count: 64

Value: b| Position: 5| Line: 3| Count: 65

Value: | Position: 5| Line: 4| Count: 66

Value: :| Position: 5| Line: 5| Count: 67

Value: =| Position: 5| Line: 6| Count: 68

Value: | Position: 5| Line: 7| Count: 69

Value: 1| Position: 5| Line: 8| Count: 70

Value: 0| Position: 5| Line: 9| Count: 71

Value: ;| Position: 5| Line: 10| Count: 72

| Position: 5| Line: 11| Count: 73

Value:

| Position: 6| Line: 1| Count: 74

Value: (| Position: 6| Line: 2| Count: 75

Value: \*| Position: 6| Line: 3| Count: 76

| Position: 6| Line: 4| Count: 77

Value:

| Position: 7| Line: 1| Count: 78

Value: | Position: 7| Line: 2| Count: 79

Value: e| Position: 7| Line: 3| Count: 80

Value: n| Position: 7| Line: 4| Count: 81

Value: d| Position: 7| Line: 5| Count: 82

Value: .| Position: 7| Line: 6| Count: 83

Лексический анализ

Описание

Лексический анализатор разделяет полученные из модуля ввода-вывода литеры и собирает их в токены: идентификаторы, ключевые слова, простые типы, операции, разделители, спецификаторы, строки, литеры, целочисленные и вещественные константы.

Каждый токен имеет собственный класс с соответствующим типом значения, причем каждый класс наследуется от базового класса «Лексема».

Для передачи полученной лексемы синтаксическому анализатору создается ссылка базового класса на объект производного

Для проверки нужной конструкции в лексическом анализаторе необходимо хранить состояние модуля чтения для возможности отката читающей каретки.

На данном этапе исключаются комментарии, неотображаемые символы (пробел, символы табуляции).

Проектирование

Анализатор содержит соответствующий класс

public Lexical(IO Input\_Reader)

Конструктор лексического анализатора получает экземпляр класса ввода вывода

Базовый класс «Лексема» содержит информацию о позиции лексемы в тексте программы

public class Lexem

Производные классы содержат поле value соответствующего типа (вещественные, целые, строковые и т.д.)

public class Keyword : Lexem

public class Simple\_Type : Lexem

public class Real : Lexem

public class Int : Lexem

public class Operation : Lexem

public class Id : Lexem

public class Limiter : Lexem

public class Specifier : Lexem

public class String : Lexem

public class Char : Lexem

public class True\_False : Lexem

Получение следующей лексемы происходит в методе

public Lexem NextSym(out string raw\_value)

Для сохранения позиции каретки, необходима структура

public struct Position\_L

Метод для сохранения позиции

public Position\_L Save\_Position()

Метод для восстановления позиции

public void Set\_Position(Position\_L Backup)

Форматированный вывод ошибок содержится в методе

public void Raise\_Error(int code)

Список встроенных ошибок содержится в методе

public static string Get\_Error(int code)

Где code – номер ошибки

Поле To\_Accept содержит не алфавитные и не числовые символы, которые программа распознает (строки, литеры, комментарии их могу содержать)

Метод для определения запрещенных символов:

public void Is\_Acceptable(char symbol)

Реализация

Ключевые слова (для сокращения кода массив содержит константы true/false), простые типы, спецификаторы, разделители и операции содержатся в массивах:

string[] operation = { "+", "-", "\*", "/", "%", "=", "<", ">", "@" };

string[] limiter = { ",", ".", "'", "(", ")", "[", "]", "{", "}", ":", ";" };

string[] specifier = { "^", "#", "$" };

string[] keywords\_ext = { "if", "do", "of", "or", "in", "to", "end", "var", "div", "and", "not", "for", "mod", "nil", "set", "then", "else", "case", "file", "goto", "type", "with", "begin", "while", "downto", "packed", "record", "repeat", "until", "program", "function", "procedure", "true", "false"};

string[] simple\_type = { "integer", "shortint", "longint", "byte", "word", "real", "single", "double", "exntended", "char", "string", "boolean" };

Определение идентификатора: первый символ из латинского алфавита, либо нижнее подчеркивание. Остальные могут быть теми же + цифры.

Определение ключевого слова: все символы должны быть латинскими, список ключевых слов находится внутри соответствующего массива

Определение целочисленной беззнаковой лексемы: все символы – цифры

Определение вещественной беззнаковой лексемы: получена константа в экспоненциальной форме или с фиксированной точкой

Определение операции, разделителя, спецификатора: если список операций содержит данный набор символов

Идентификаторы и ключевые слова начинаются с латиницы, константы с цифры.

Тестирование

Вывод всех лексем подряд

Исходная программа:

**program** showRelations;

**var**

a, b: integer;

**begin**

a := 21;

b := 10;

**if** a = b **then**

writeln('Line 1 - a is equal to b' )

**end**.

Результат вывода всех лексем:

Token: Keyword| Value: program| Position: 8| Line: 1| Count: 7

Token: Lexem| Value: | Position: 7| Line: 1| Count: 18

Token: Id| Value: ons| Position: 22| Line: 1| Count: 21

Token: Limiter| Value: ;| Position: 22| Line: 1| Count: 22

Token: Keyword| Value: var| Position: 4| Line: 2| Count: 27

Token: Id| Value: a| Position: 2| Line: 3| Count: 30

Token: Limiter| Value: ,| Position: 2| Line: 3| Count: 31

Token: Id| Value: b| Position: 5| Line: 3| Count: 33

Token: Limiter| Value: :| Position: 5| Line: 3| Count: 34

Token: Simple\_Type| Value: integer| Position: 13| Line: 3| Count: 42

Token: Limiter| Value: ;| Position: 14| Line: 3| Count: 43

Token: Keyword| Value: begin| Position: 6| Line: 4| Count: 50

Token: Id| Value: a| Position: 3| Line: 5| Count: 54

Token: Operation| Value: :=| Position: 5| Line: 5| Count: 57

Token: Int| Value: 21| Position: 9| Line: 5| Count: 60

Token: Limiter| Value: ;| Position: 9| Line: 5| Count: 61

Token: Id| Value: b| Position: 3| Line: 6| Count: 65

Token: Operation| Value: :=| Position: 5| Line: 6| Count: 68

Token: Int| Value: 10| Position: 9| Line: 6| Count: 71

Token: Limiter| Value: ;| Position: 9| Line: 6| Count: 72

Token: Limiter| Value: (| Position: 1| Line: 7| Count: 75

Token: Operation| Value: \*| Position: 2| Line: 7| Count: 76

Token: Keyword| Value: end| Position: 5| Line: 8| Count: 82

Token: Limiter| Value: .| Position: 5| Line: 8| Count: 83

Вывод ошибки о незаконченном комментарии

Исходная программа:

**program** showRelations;

**var**

a, b: integer;

**begin**

a := 21;

b := 10;

(\*Комментарий

end.

Результат вывода всех лексем:

Token: Keyword| Value: program| Position: 8| Line: 1| Count: 7

Token: Lexem| Value: | Position: 7| Line: 1| Count: 18

Token: Id| Value: ons| Position: 22| Line: 1| Count: 21

Token: Limiter| Value: ;| Position: 22| Line: 1| Count: 22

Token: Keyword| Value: var| Position: 4| Line: 2| Count: 27

Token: Id| Value: a| Position: 2| Line: 3| Count: 30

Token: Limiter| Value: ,| Position: 2| Line: 3| Count: 31

Token: Id| Value: b| Position: 5| Line: 3| Count: 33

Token: Limiter| Value: :| Position: 5| Line: 3| Count: 34

Token: Simple\_Type| Value: integer| Position: 13| Line: 3| Count: 42

Token: Limiter| Value: ;| Position: 14| Line: 3| Count: 43

Token: Keyword| Value: begin| Position: 6| Line: 4| Count: 50

Token: Id| Value: a| Position: 3| Line: 5| Count: 54

Token: Operation| Value: :=| Position: 5| Line: 5| Count: 57

Token: Int| Value: 21| Position: 9| Line: 5| Count: 60

Token: Limiter| Value: ;| Position: 9| Line: 5| Count: 61

Token: Id| Value: b| Position: 3| Line: 6| Count: 65

Token: Operation| Value: :=| Position: 5| Line: 6| Count: 68

Token: Int| Value: 10| Position: 9| Line: 6| Count: 71

Token: Limiter| Value: ;| Position: 9| Line: 6| Count: 72

Col: 5 Ln: 7 Description: Комментарий не закрыт

Вывод ошибки о незаконченной строке/литере

Текст исходной программы:

**program** showRelations;

**var**

a, b: integer;

st: string;

ch :char;

**begin**

a := 21;

b := 10;

//Комментарий

ch := 'a';

st := 'string;

**end**.

Результат вывода всех лексем:

Token: Keyword| Value: program| Position: 8| Line: 1| Count: 7

Token: Lexem| Value: | Position: 7| Line: 1| Count: 18

Token: Id| Value: ons| Position: 22| Line: 1| Count: 21

Token: Limiter| Value: ;| Position: 22| Line: 1| Count: 22

Token: Keyword| Value: var| Position: 4| Line: 2| Count: 27

Token: Id| Value: a| Position: 2| Line: 3| Count: 30

Token: Limiter| Value: ,| Position: 2| Line: 3| Count: 31

Token: Id| Value: b| Position: 5| Line: 3| Count: 33

Token: Limiter| Value: :| Position: 5| Line: 3| Count: 34

Token: Simple\_Type| Value: integer| Position: 13| Line: 3| Count: 42

Token: Limiter| Value: ;| Position: 14| Line: 3| Count: 43

Token: Id| Value: st| Position: 3| Line: 4| Count: 47

Token: Limiter| Value: :| Position: 3| Line: 4| Count: 48

Token: Simple\_Type| Value: string| Position: 10| Line: 4| Count: 55

Token: Limiter| Value: ;| Position: 11| Line: 4| Count: 56

Token: Id| Value: ch| Position: 3| Line: 5| Count: 60

Token: Limiter| Value: :| Position: 4| Line: 5| Count: 62

Token: Simple\_Type| Value: char| Position: 8| Line: 5| Count: 66

Token: Limiter| Value: ;| Position: 9| Line: 5| Count: 67

Token: Keyword| Value: begin| Position: 6| Line: 6| Count: 74

Token: Id| Value: a| Position: 3| Line: 7| Count: 78

Token: Operation| Value: :=| Position: 5| Line: 7| Count: 81

Token: Int| Value: 21| Position: 9| Line: 7| Count: 84

Token: Limiter| Value: ;| Position: 9| Line: 7| Count: 85

Token: Id| Value: b| Position: 3| Line: 8| Count: 89

Token: Operation| Value: :=| Position: 5| Line: 8| Count: 92

Token: Int| Value: 10| Position: 9| Line: 8| Count: 95

Token: Limiter| Value: ;| Position: 9| Line: 8| Count: 96

Token: Id| Value: ch| Position: 4| Line: 10| Count: 117

Token: Operation| Value: :=| Position: 6| Line: 10| Count: 120

Token: Char| Value: 'a'| Position: 10| Line: 10| Count: 124

Token: Limiter| Value: ;| Position: 11| Line: 10| Count: 125

Token: Id| Value: st| Position: 4| Line: 11| Count: 130

Token: Operation| Value: :=| Position: 6| Line: 11| Count: 133

Col: 4 Ln: 11 Description: Ошибка в константе

Вывод ошибки о неправильной форме вещественной константы

Текст исходной программы:

**program** showRelations;

**var**

a, b: integer;

re, ab :real;

**begin**

a := 21;

b := 10;

ab := -0.17E3;

re := 0.;

end.

Результат вывода всех лексем:

Token: Keyword| Value: program| Position: 8| Line: 1| Count: 7

Token: Lexem| Value: | Position: 7| Line: 1| Count: 18

Token: Id| Value: ons| Position: 22| Line: 1| Count: 21

Token: Limiter| Value: ;| Position: 22| Line: 1| Count: 22

Token: Keyword| Value: var| Position: 4| Line: 2| Count: 27

Token: Id| Value: a| Position: 2| Line: 3| Count: 30

Token: Limiter| Value: ,| Position: 2| Line: 3| Count: 31

Token: Id| Value: b| Position: 5| Line: 3| Count: 33

Token: Limiter| Value: :| Position: 5| Line: 3| Count: 34

Token: Simple\_Type| Value: integer| Position: 13| Line: 3| Count: 42

Token: Limiter| Value: ;| Position: 14| Line: 3| Count: 43

Token: Id| Value: re| Position: 3| Line: 4| Count: 47

Token: Limiter| Value: ,| Position: 3| Line: 4| Count: 48

Token: Id| Value: ab| Position: 7| Line: 4| Count: 51

Token: Limiter| Value: :| Position: 8| Line: 4| Count: 53

Token: Simple\_Type| Value: real| Position: 12| Line: 4| Count: 57

Token: Limiter| Value: ;| Position: 13| Line: 4| Count: 58

Token: Keyword| Value: begin| Position: 6| Line: 5| Count: 65

Token: Id| Value: a| Position: 3| Line: 6| Count: 69

Token: Operation| Value: :=| Position: 5| Line: 6| Count: 72

Token: Int| Value: 21| Position: 9| Line: 6| Count: 75

Token: Limiter| Value: ;| Position: 9| Line: 6| Count: 76

Token: Id| Value: b| Position: 3| Line: 7| Count: 80

Token: Operation| Value: :=| Position: 5| Line: 7| Count: 83

Token: Int| Value: 10| Position: 9| Line: 7| Count: 86

Token: Limiter| Value: ;| Position: 9| Line: 7| Count: 87

Token: Id| Value: ab| Position: 4| Line: 8| Count: 92

Token: Operation| Value: :=| Position: 6| Line: 8| Count: 95

Token: Operation| Value: -| Position: 8| Line: 8| Count: 97

Token: Real| Value: 0.17E3| Position: 15| Line: 8| Count: 103

Token: Limiter| Value: ;| Position: 15| Line: 8| Count: 104

Token: Id| Value: re| Position: 4| Line: 9| Count: 109

Token: Operation| Value: :=| Position: 6| Line: 9| Count: 112

Col: 10 Ln: 8 Description: Ошибка в константе

Вывод ошибки о непредвиденном символе

Текст исходной программы:

**program** showRelations;

**var**

a, b: integer;

re, ab :real;

**begin**

a№ := 21;

b := 10;

ab := -0.17E3;

**end**.

Результат вывода всех лексем:

Token: Keyword| Value: program| Position: 8| Line: 1| Count: 7

Token: Lexem| Value: | Position: 7| Line: 1| Count: 18

Token: Id| Value: ons| Position: 22| Line: 1| Count: 21

Token: Limiter| Value: ;| Position: 22| Line: 1| Count: 22

Token: Keyword| Value: var| Position: 4| Line: 2| Count: 27

Token: Id| Value: a| Position: 2| Line: 3| Count: 30

Token: Limiter| Value: ,| Position: 2| Line: 3| Count: 31

Token: Id| Value: b| Position: 5| Line: 3| Count: 33

Token: Limiter| Value: :| Position: 5| Line: 3| Count: 34

Token: Simple\_Type| Value: integer| Position: 13| Line: 3| Count: 42

Token: Limiter| Value: ;| Position: 14| Line: 3| Count: 43

Token: Id| Value: re| Position: 3| Line: 4| Count: 47

Token: Limiter| Value: ,| Position: 3| Line: 4| Count: 48

Token: Id| Value: ab| Position: 7| Line: 4| Count: 51

Token: Limiter| Value: :| Position: 8| Line: 4| Count: 53

Token: Simple\_Type| Value: real| Position: 12| Line: 4| Count: 57

Token: Limiter| Value: ;| Position: 13| Line: 4| Count: 58

Token: Keyword| Value: begin| Position: 6| Line: 5| Count: 65

Col: 3 Ln: 5 Description: Запрещенный символ

Синтаксический анализатор

Описание

Данный анализатор работает одновременно с лексическим анализатором и модулем ввода-вывода для отката ошибок.

Описание синтаксиса языка Паскаль в виде диаграмм Бэкуса-Наура, взяты в справочнике (Паскаль: руководство для пользователя. Йенсен К., Вирт Н.).

Лексемы программы последовательно проверяются на соответствие конструкциям, в зависимости от наличия ключевых слов, типов констант и остальных знаков.

На данном этапе необходимо найти лексемы, которые нарушают синтаксические правила построения программы.

Программа состоит шапки (название), описания переменных, описания операторов.

Реализована дополнительная часть: выражения, условный оператор, оператор цикла с предусловием

Проектирование

Для вывода нестандартных ошибок реализован метод

public void Unhandled\_Error(string expected)

Который получает на вход строку ошибки конструкции

Метод для обновления текущей лексемы, которую выдает модуль лексического анализатор:

public void NextSym()

Метод для обработки строковых лексем. Возвращает true в случае совпадения ожидаемой и полученной лексем:

public bool Accept\_Raw(string raw\_value)

Далее идет реализация обработки конструкций через диаграммы Бэкуса-Наура:

Общая структура программы: шапка, блок

public void Accept\_Program()

Блок, содержит описания, операторную часть программы:

public void Accept\_Block()

Часть описаний, содержит описание переменных:

public void Accept\_Description\_Chapter()

Часть описания операторов, содержит составной оператор:

public bool Accept\_Operators\_Chapter()

Часть описания переменных. Обращается к методу «описание переменных»

public void Accept\_Variables\_Chapter()

Составной оператор. Имеет следующую форму

< составной оператор > ::= begin < оператор > { ; < оператор > } end

public bool Accept\_Compound\_Operator()

Оператор, включает в себя основной или производный оператор:

public bool Accept\_Operator()

Основной оператор, включает в себя пустой оператор, оператор процедуры, оператор присваивания:

public bool Accept\_Main\_Operator()

Оператор процедуры, имеет следующую форму

< оператор процедуры > ::= < имя процедуры > | < имя процедуры > ( < фактический параметр > { , < фактический параметр > } )

public bool Accept\_Procedure\_Operator()

Производный оператор, состоит из составного оператора, оператора выбора, оператора цикла:

public bool Accept\_Derivative\_Operator()

Описание переменных, имеет следующую форму

< раздел переменных >::= | var<описание переменных> ; {<описание переменных> ; }:

public bool Accept\_Variables\_Description()

Оператор присваивания, имеет следующую форму

< оператор присваивания >::= < переменная > := < выражение > | < имя функции > := < выражение >

public bool Accept\_Assignment\_Operator()

Пустой оператор, состоит в проверке, что следующая лексема – ;

public bool Accept\_Void\_Operator()

Оператор выбора, обращается к условному оператору:

public bool Accept\_Selecting\_Operator()

Условный оператор, имеет следующую форму

< условный оператор >::= if < логическое выражение > then < оператор > | if < логическое выражение > then < оператор > else < оператор >

public bool Accept\_Conditional\_Operator()

Оператор цикла, обращается к оператору цикла с предусловием:

public bool Accept\_Cycle\_Operator()

Оператор цикла с предусловием, имеет следующую форму

< оператор цикла с предусловием > ::= while <логическое выражение

> do <оператор>

public bool Accept\_Loop\_Operator\_With\_Precondition()

Параметр цикла, проверяет наличие имени переменной:

public bool Accept\_Cycle\_Parameter()

Имя переменной, обращается к методу «имя»:

public bool Accept\_Variable\_Name()

Простой тип, обращается к имени типа:

public bool Accept\_Type()

Выражение, может принимать арифметическое, литерное, логическое выражения:

public bool Accept\_Expression()

Переменная, обращается к имени переменной:

public bool Accept\_Variable()

Арифметическое выражение, имеет следующую структуру

< арифметическое выражение >::= <слагаемое> { < операции сложения > <слагаемое> } | < операция сложения > <слагаемое> { <операция сложения> <слагаемое>}

< операция сложения >::= + | –

public bool Accept\_Arithmetic\_Expression()

Слагаемое, имеет структуру

< слагаемое >::= <множитель> { < операция умножения > <множитель> }

< операция умножения >::= \* | / | mod | div

public bool Accept\_Summand()

Множитель, принимает беззнаковое вещественное, беззнаковое целочисленное, переменные, другое арифметическое выражение:

public bool Accept\_Multiplier()

Литерное выражение, принимает константы или переменные литерного типа

public bool Accept\_Literal\_Expression()

Литерный множитель, имеет структуру арифметического множителя, но принимает литерное выражение, литерные переменные и константы, строковые переменные:

public bool Accept\_Literal\_Multiplier()

Фактический параметр, принимает выражение, переменные:

public bool Accept\_Actual\_Parameter()

Литерный токен, принимает литерные константы и переменные, строковые константы и переменные:

public bool Accept\_Literal\_Token()

Имя процедуры, проверяет, что имя – writeln():

public bool Accept\_Procedure\_Name()

Логическое выражение, принимает отношение или простое логическое выражение:

public bool Accept\_Logical\_Expression()

Простое логическое выражение, имеет структуру

<простое логическое выражение> ::= <логическое слагаемое> { or <логическое слагаемое> }

public bool Accept\_Simple\_Logical\_Expression()

Логическое слагаемое, имеет форму

< логическое слагаемое >::=< логический множитель > |< логическое слагаемое > and < логический множитель >

public bool Accept\_Logical\_Summand()

Логический множитель, имеет структуру

< логический множитель >::= < константа логического типа > | < переменная логического типа > | < функция логического типа > | not < логический множитель > | ( < логическое выражение > )

public bool Accept\_Logical\_Multiplier()

Отношение, может принимать скалярное или строковое отношения:

public bool Accept\_Relationship()

Скалярное отношение. Общая форма: <тип выражения> операция сравнения < тип выражения >. Выражения могут быть арифметическими, простыми логическими, литерными

public bool Accept\_Scalar\_Relationship()

Строковое отношение, имеет форму

< строковое отношение >::= < строковая константа >< операция сравнения > < строковая константа > | < переменная >< операция сравнения > < строковая константа > | < строковая константа >< операция сравнения >< переменная > | < переменная >< операция сравнения >< переменная >

public bool Accept\_String\_Relastionship()

Класс лексемы, получает название класса лексемы:

public string Get\_Class()

Оператор сравнения, имеет форму

< операция сравнения >::= = | <> | <= | < | >= | >

public bool Accept\_Comparison\_Operaion()

Беззнаковое вещественное, проверяет, что класс лексемы - Real

public bool Accept\_Real\_Without\_Sign()

Беззнаковое целочисленное, проверяет, что класс лексемы – Int

public bool Accept\_Unsigned\_Integer()

Имя, , проверяет, что класс лексемы – Id (идентификатор)

public bool Accept\_Name()

Простой тип, проверяет, что класс лексемы - Simple\_Type

public bool Accept\_Type\_Name()

Строковая константа, проверяет, что класс лексемы - String

public bool Accept\_String\_Const()

Литерная константа, проверяет, что класс лексемы - Char

public bool Accept\_Literal\_Const()

True/False, проверяет, что класс лексемы - True\_False

public bool Accept\_True\_False()

Тестирование

Проверка ошибки при описании переменных

Текст исходной программы:

**program** prog;

**var** a, b: ;

re, ab :real;

**begin**

a := 21;

b := 10;

ab := -0.17E3;

**end**.

Результат работы синтаксического анализатора:

Col: 11 Ln: 1 Получено ;, а ожидалось тип

Проверка ошибки на наличие «;»

Текст исходной программы:

**program** prog;

**var** a, b: integer;

re, ab :real;

**begin**

a := 21

b := 10;

ab := -0.17E3;

**end**.

Результат работы синтаксического анализатора:

Col: 2 Ln: 5 Description: Должен идти символ ';'

Проверка ошибки в условном операторе

Текст исходной программы:

**program** prog;

**var** a, b: integer;

re, ab :real;

**begin**

a := 21;

b := 10;

**if** a>5 writeln('истина');

ab := -0.17E3;

**end**.

Результат работы синтаксического анализатора:

Col: 14 Ln: 6 Description: Должно идти слово THEN

Проверка ошибки в операторе цикла

Текст исходной программы:

**program** prog;

**var** a, b: integer;

re, ab :real;

**begin**

a := 21;

b := 10;

**while** true

**begin**

writeln('text');

**end**;

ab := -0.17E3;

**end**.

Результат работы синтаксического анализатора:

Col: 8 Ln: 7 Description: Должно идти слово DO

Проверка ошибки в арифметическом выражении

Текст исходной программы:

**program** prog;

**var** a, b: integer;

re, ab :real;

**begin**

a := 21;

b := 10;

ab := (63+200\*(76+24));

a := (90-(400/4.07)-);

**end**.

Результат работы синтаксического анализатора:

Col: 7 Ln: 7 Получено ), а ожидалось слагаемое

Проверка ошибки в строковом выражении

Текст исходной программы:

**program** prog;

**var** a, b: string;

re, ab :char;

**begin**

a := 'text1';

b := 'text2';

re := 't';

ab := 'u';

a := a + re;

a := a \* re;

**end**.

Результат работы синтаксического анализатора:

Col: 14 Ln: 9 Description: Операция не применима к типам string и char

Проверка ошибки в логическом выражении

Текст исходной программы:

**program** prog;

**var** a, b: boolean;

**begin**

**if** true **or** 'text' **then**

writeln('text');

**end**.

Результат работы синтаксического анализатора:

Col: 3 Ln: 3 Получено 'text', а ожидалось логическое слагаемое

Семантические анализатор

Описание

Для проверки корректности программы недостаточно пройтись по формальным правилам БНФ. Например, в выражении идентификатор-слагаемое может иметь тип, не соответствующий второму слагаемому. То есть нельзя сложить строку с вещественным числом. Для решения этой проблемы необходимо создать таблицу для области видимости (т.к. в моей реализации нет функций и процедур, область всего одна => одна таблица). Т.к. таблица одна, я буду использовать словарь. Также в каждом выражении необходимо сделать проверки на приводимость типов слагаемых.

Необходимо для каждого типа добавить поля, позволяющие определить возможность приведения данного типа. Также нужно учесть, что один тип может быть приводим арифметически к другому, но не может быть присвоен (например, char можно сложить со string, но присвоить char string нельзя)

Проектирование

Базовый абстрактный класс для хранения информации о приводимости типа: 3 списка (сравнение, арифметические операции, присвоение) к каждому типу

public abstract class CType

Перечисление для работы с типами.

public enum EType

{

Integer,

Real,

String,

Char,

Boolean,

NotDerivable

}

Для хранения таблицы типов необходимо создать словарь

public Dictionary<string, EType> Types\_Table = new Dictionary<string, EType>();

Для хранения информации о приводимости необходимо создать структуры, каждый класс будет содержать свои структуры приводимости типов

public struct DerividedTypeList

public struct DerividedBoolList

Далее производные для базового CType классы для каждого основного типа:

public class IntCast: CType

public class RealCast : CType

public class StringCast : CType

public class CharCast : CType

public class BooleanCast : CType

Метод для проверки приводимости типов при сравнении:

public bool isComparable(EType Left, EType Right)

Метод для проверки приводимости типов при присваивании:

public EType IsAssignable(EType Left, EType Right)

Метод для проверки приводимости типов при выполнении арифметических операций

public EType IsArithmeticDerivided(EType Left, EType Right)

Для добавления нового идентификатора в словарь необходимо реализовать метод

public bool AddIdent(string Ident\_Name, string Type)

В процессе работы с типами необходимо преобразовывать строки в тип перечисления и наоборот, для этого необходимо реализовать методы

public EType ConvertToType(string raw\_type)

public string ConvertToString(EType rawtype)

Реализация

Семантический анализатор встраивается в синтаксический, его экземпляр нужен для конструктора синтаксического

Для добавления идентификаторов необходимо из метода принятия имени переменной возвращать кортеж <bool, string>, где 1 значение указывает на успешность принятия конструкции, а в string хранится тип выражения.

Например, в синтаксическом анализаторе необходимо добавить список всех имен переменных, затем в словарь занести имена и тип.

List<string> Ids = new List<string>();

Tuple<bool, Lexical.Lexem> temp = Accept\_Variable\_Name();

Tuple<bool, string> Bool\_EType = new Tuple<bool, string>(temp.Item1, temp.Item2.value);

if (!Bool\_EType.Item1)

{

return false;

}

Ids.Add(Bool\_EType.Item2);

while (Accept\_Raw(","))

{

temp = Accept\_Variable\_Name();

Bool\_EType = new Tuple<bool, string>(temp.Item1, temp.Item2.value); ;

if (!Bool\_EType.Item1)

{

Unhandled\_Error("имя переменной");

}

Ids.Add(Bool\_EType.Item2);

}

if (!Accept\_Raw(":")) Raise\_Error(5);

Bool\_EType = Accept\_Type();

if (!Bool\_EType.Item1)

{

Unhandled\_Error("тип");

}

foreach(string id in Ids)

{

if (!Semantic\_Analyzer.AddIdent(id, Bool\_EType.Item2)) Raise\_Error(101);

}

В выражениях каждое слагаемое должно провериться на соответствие типа.

Bool\_EType\_2 = Accept\_Summand();

if (Bool\_EType\_2.Item2.ToLower() != "integer" && Bool\_EType\_2.Item2.ToLower() != "real")

{

return new Tuple<bool, string>(false, "");

}

if (!Bool\_EType\_2.Item1) Unhandled\_Error("слагаемое");

.WriteLine("First: " + Bool\_EType.Item2 + " Second: " + Bool\_EType\_2.Item2);

derividedtype = Semantic\_Analyzer.IsArithmeticDerivided(Semantic\_Analyzer.ConvertToType(Bool\_EType.Item2), Semantic\_Analyzer.ConvertToType(Bool\_EType\_2.Item2));

if (derividedtype == EType.NotDerivable) Raise\_Error(145);

else Bool\_EType = new Tuple<bool, string>(true, Semantic\_Analyzer.ConvertToString(derividedtype));

Тестирование

Проверка ошибки неопределенного идентификатора

Текст исходной программы:

**program** prog;

**var** a, b: boolean;

e, u : char;

**as**, ag: string;

i, i\_ : integer;

r : real;

**begin**

i:=5;

i\_ :=7;

pp := 99;

a := False;

**if** 'asdf' < b **then** writeln();

**end**.

Результат работы семантического анализатора:

Col: 10 Ln: 9 Description: Переменная не объявлена

Проверка ошибки на повторное объявление идентификатора

Текст исходной программы:

**program** prog;

**var** a, b: boolean;

e, u : char;

**as**, u: string;

i, i\_ : integer;

r : real;

**begin**

i:=5;

i\_ :=7;

a := False;

**end**.

Результат работы семантического анализатора:

Col: 14 Ln: 3 Description: Имя описано повторно

Проверка ошибки на несоответствие типов арифметической операции

Текст исходной программы:

**program** prog;

**var** a, b: boolean;

e : char;

**as**, u: string;

i, i\_ : integer;

r : real;

**begin**

i:= 5 + a;

i\_ := 7;

a := False;

**if** 'asdf' < b **then** writeln();

**end**.

Результат работы семантического анализатора:

Col: 7 Ln: 7 Description: Ошибка типа/ожидалось выражение

Проверка ошибки на несоответствие типов при присваивании

Текст исходной программы:

**program** prog;

**var** a, b: boolean;

e : char;

**as**, u: string;

i, i\_ : integer;

r : real;

**begin**

i:= e;

i\_ :=7;

a := False;

**end**.

Результат работы семантического анализатора:

Col: 8 Ln: 7 Description: Тип результата функции не соответствует описанию

Проверка ошибки на несоответствие логических операндов

Текст исходной программы:

**program** prog;

**var** a, b: boolean;

e : char;

**as**, u: string;

i, i\_ : integer;

r : real;

**begin**

i\_ :=7;

a := False;

**if** 'asdf' < b **then** writeln();

end.

Результат работы семантического анализатора:

Col: 15 Ln: 9 Description: Конфликт типов