|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Поволжский Государственный Университет Телекоммуникаций и Информатики** Кафедра «ПОУТС»     |  |  | | --- | --- | | Сдана на проверку  «\_\_\_» \_\_\_\_\_\_\_\_ 2017 г. | Допустить к защите  «\_\_\_» \_\_\_\_\_\_\_\_ 2017 г.  Защищена с оценкой  «\_\_\_» \_\_\_\_\_\_\_\_ 2017 г. |   **КУРСОВАЯ РАБОТА** По дисциплине: «Конструирование компиляторов» На тему: «Разработка компилятора модельного языка»  Пояснительная записка     |  |  | | --- | --- | | Студент группы ИВТм-61 \_\_\_\_\_\_\_\_\_  (подпись) (ФИО)  Руководитель \_\_\_\_\_\_\_\_\_\_\_\_\_\_ д.т.н. Бахарева Н. Ф.  (подпись) (ФИО) | \_\_\_\_\_\_\_\_\_\_\_\_\_  (№ зачетной книжки) |     Самара, 2017 |

**Рецензия**

**Разработка лексического, синтаксического и семантического анализатора модельного языка программирования.**

Цель курсовой работы:

1. В соответствии с номером варианта составить формальное описание модельного языка программирования с помощью:

а) РБНФ;

б) диаграмм Вирта;

в) формальных грамматик.

1. Разработать программное средство для проведения анализа программ, написанных на данном языке, при помощи генератора синтаксических анализаторов.
2. Написать пять содержательных примеров программ, раскрывающих особенности конструкций учебного языка программирования, отразив в этих примерах все его функциональные возможности.

**Реферат**

Курсовая работа.

Пояснительная записка: 48 с., 54 рис., 0 табл., 1 прил.

Разработка компилятора модельного языка программирования

Разработана система лексического, синтаксического и семантического анализа кода написанного на модельном языке программирования, а также его интерпретация.

Язык программирования: Java, Oracle JDK 1.8.

**Постановка задачи**

Разработать лексический и синтаксический анализатор модельного языка при помощи генератора синтаксических анализаторов.

Синтаксис модельного языка в форме Бэкуса-Наура (далее БНФ) и в виде регулярных выражений с наглядной иллюстрацией в виде диаграмм Вирта можно представить в следующем виде:

Операции групп «сложение», «умножение», «отношение» и унарная операция:

<операции\_группы\_отношения>::= *NE* | *EQ* | *LT* | *LE* | *GT* | *GE*

<операции\_группы\_отношения>: *NE* | *EQ* | *LT* | *LE* | *GT* | *GE*

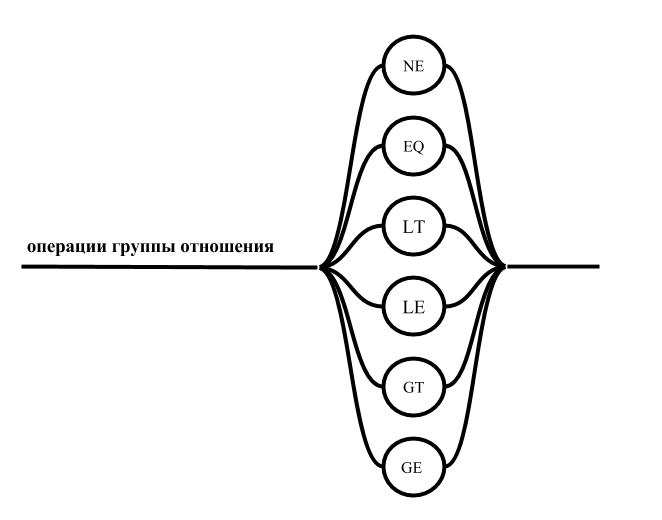
**

Рис. 1 - Диаграмма операций группы отношение.

<операции\_группы\_сложения>:: = *plus* | *min* | *or*

<операции\_группы\_сложения>: *plus* | *min* | *or*

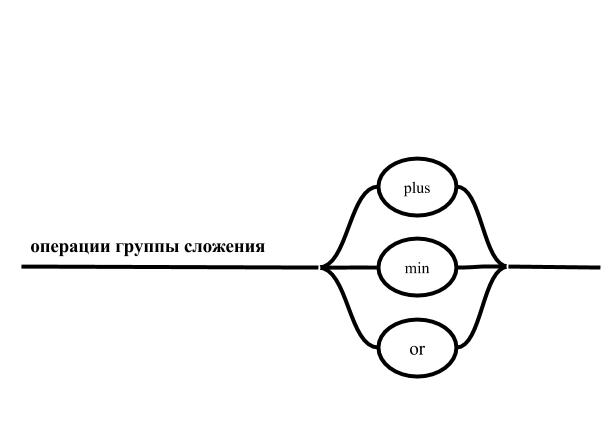
**

Рис. 2 - Диаграмма операций группы сложение.

<операции\_группы\_умножения>::= *mult* | *div* | *and*

<операции\_группы\_умножения>: *mult* | *div* | *and*

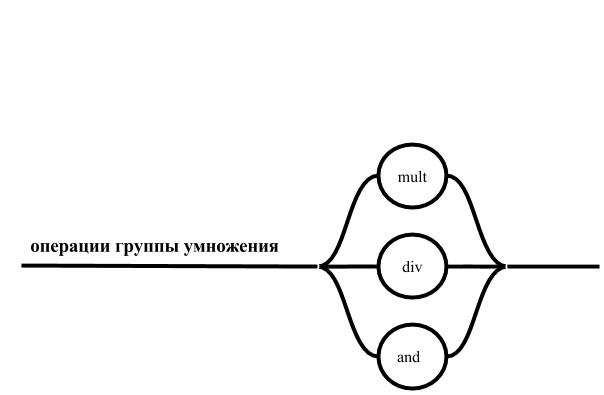
**

Рис. 3 - Диаграмма операций группы умножение.

<унарная\_операция>::= ~

<унарная\_операция>: ~

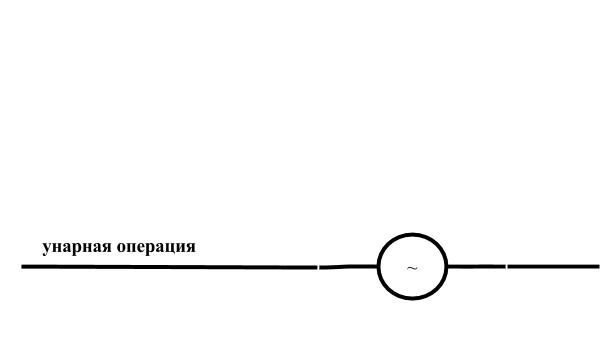


Рис. 4 - Диаграмма унарной операции.

Структура программы:

<программа>::= «{» {/ (<описание> | <оператор>) ; /} «}»

<программа>: \{(<описание> | <оператор>;)+\}

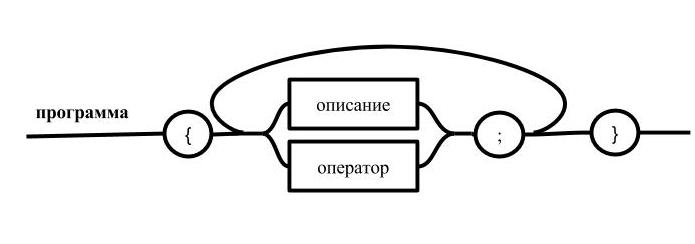


Рис. 5 - Диаграмма структуры программы.

Команды описания данных:

<описание>::= {<идентификатор> {, <идентификатор> } : <тип>;}

<описание>: <идентификатор> (, <идентификатор>)\* \ : <тип>

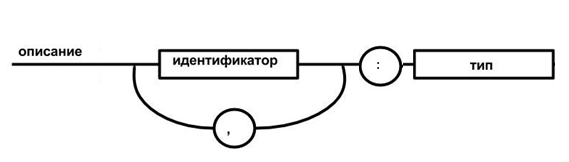


Рис. 6 - Диаграмма структуры описания идентификаторов.

Типы данных:

<тип>::= integer | real | boolean

<тип>: integer | real | boolean

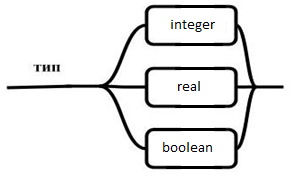


Рис. 7 - Диаграмма типов данных.

Операторы:

<оператор>::= <составной> | <присваивания> | <условный> |

<фиксированного\_цикла> | <условного\_цикла> | <ввода> | <вывода>

<оператор>: <составной> | <присваивания> | <условный> |

<фиксированного\_цикла> | <условного\_цикла> | <ввода> | <вывода>

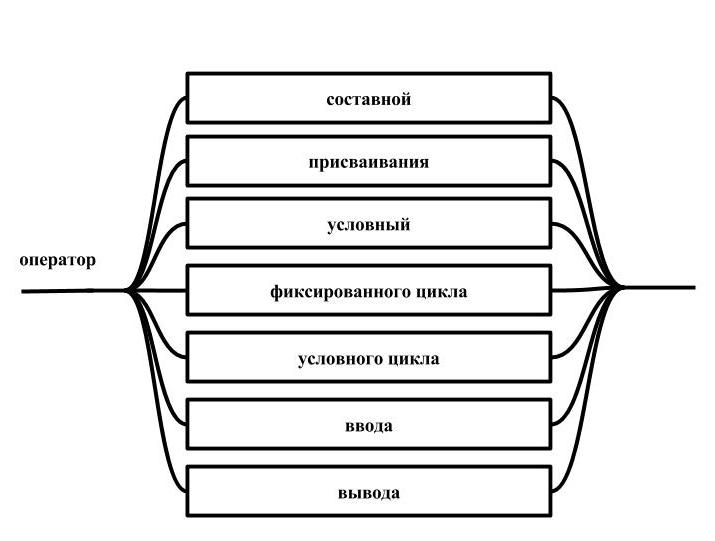


Рис. 8 - Диаграмма операторов.

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

<составной>: *begin* <оператор> (; <оператор>)\* *end*

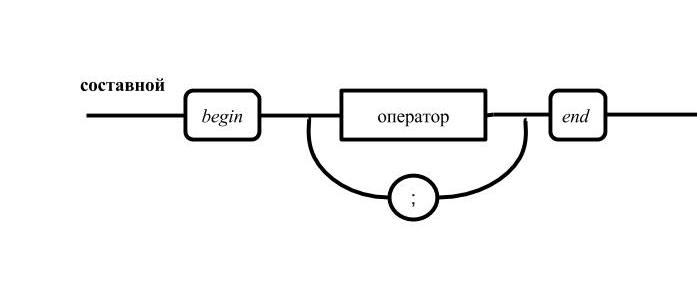
**

Рис. 9 - Диаграмма составного оператора.

<присваивания>::= <идентификатор> := <выражение>

<присваивания>: <идентификатор> := <выражение>

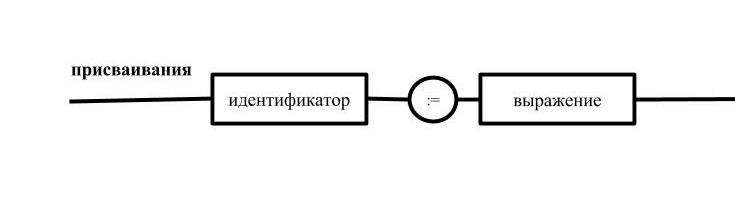


Рис. 10 - Диаграмма оператора присваивания.

<условный>::= *if* (<выражение>) <оператор> [*else* <оператор>]

<условный>: *if* \(<выражение>\) <оператор> (*else* <оператор>)?

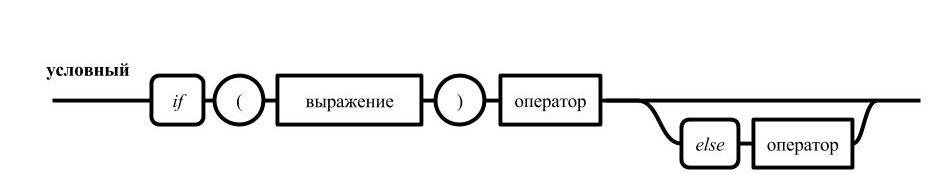


Рис. 11 - Диаграмма условного оператора.

<фиксированного\_цикла>::= *for* <присваивания> *to* <выражение> [*step* <выражение>] <оператор> *next*

<фиксированного\_цикла>: *for* <присваивания> *to* <выражение> (*step* <выражение>)? <оператор> *next*

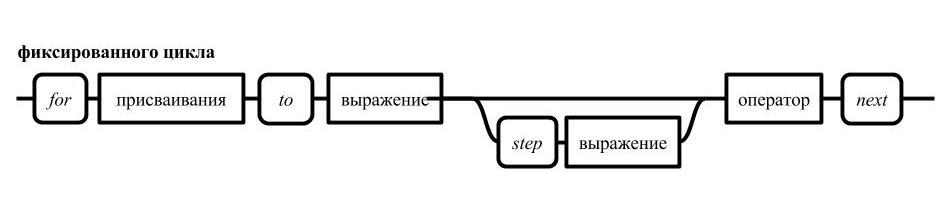
**

Рис. 12 - Диаграмма оператора фиксированного цикла.

<условного\_цикла>::= *while* (<выражение>) <оператор>

<условного\_цикла>: *while* \(<выражение>\) <оператор>

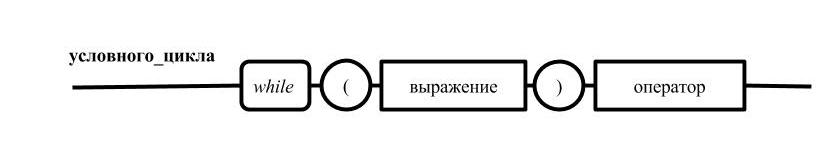


Рис. 13 - Диаграмма оператора условного цикла.

<ввода>::= *readln* идентификатор {, <идентификатор> }

<ввода>: *readln* идентификатор (, <идентификатор> )\*

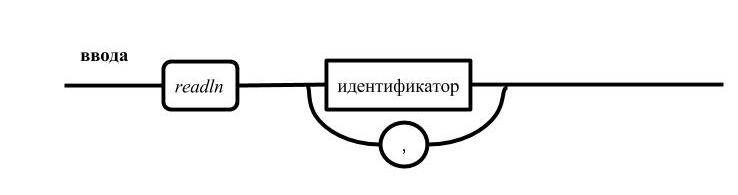


Рис. 14 - Диаграмма оператора ввода.

<вывода>::= *writeln* <выражение> {, <выражение> }

<вывода>: *writeln* <выражение> (, <выражение> )\*

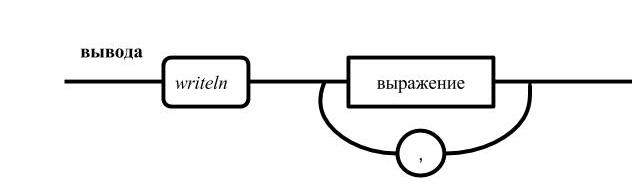


Рис.15 - Диаграмма оператора вывода.

Комментарии:

Признак начала комментария: «/\*»;

Признак конца комментария: «\*/»

<комментарий>::=/\* {<символ>} \*/

<комментарий>::=\/\\* (<символ>)\* \\*\/

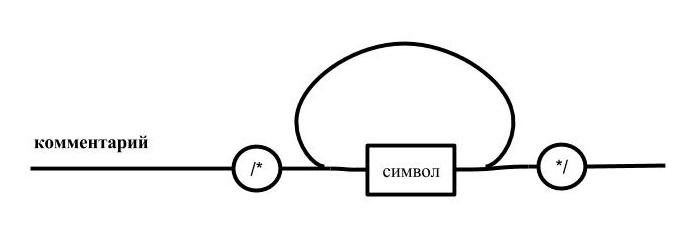


Рис. 16 - Диаграмма комментария.

Выражения языказадаются правилами:

<выражение>::= <операнд>{<операции\_группы\_отношения> <операнд>}

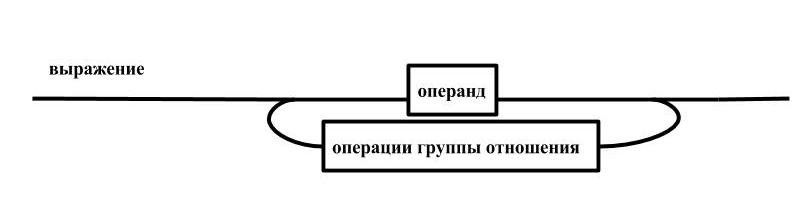
<выражение>: <операнд>(<операции\_группы\_отношения> <операнд>)\*

Рис. 17 - Диаграмма выражения.

<операнд>::= <слагаемое> {<операции\_группы\_сложения> <слагаемое>}

<операнд>: <слагаемое> (<операции\_группы\_сложения> <слагаемое>)\*

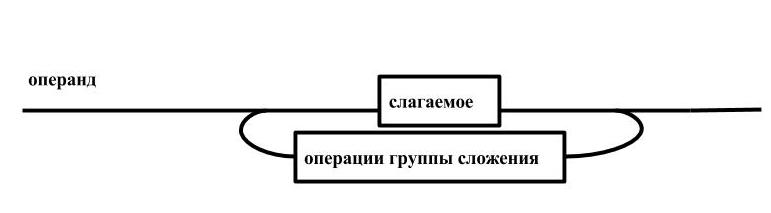


Рис. 18 - Диаграмма операнда.

<слагаемое>::= <множитель> {<операции\_группы\_умножения> <множитель>}

<слагаемое>: <множитель> (<операции\_группы\_умножения> <множитель>)\*



Рис. 19 - Диаграмма слагаемого.

<множитель>::= <идентификатор> | <число> | <логическая\_константа> |

<унарная\_операция> <множитель> | (<выражение>)

<множитель>: <идентификатор> | <число> | <логическая\_константа> |

<унарная\_операция> <множитель> | \(<выражение>\)

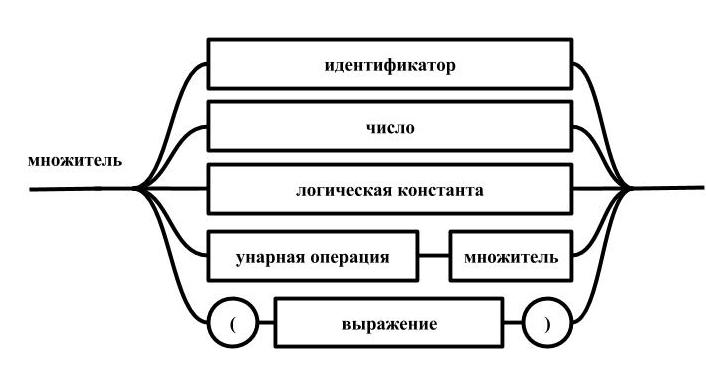


Рис. 20 - Диаграмма множителя.

<число>::= <целое> | <действительное>

<число>: <целое> | <действительное>

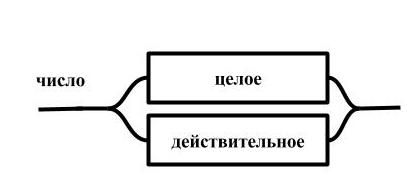


Рис. 21 - Диаграмма числа.

<логическая\_константа>::= *true* | *false*

<логическая\_константа>: *true* | *false*

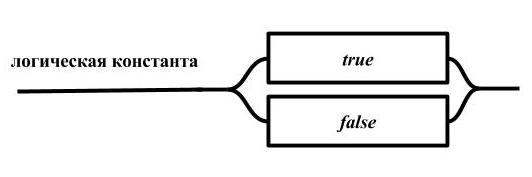


Рис. 22 - Диаграмма логической константы.

## Правила, определяющие идентификатор, букву и цифру:

<идентификатор>::= <буква> {<буква> | <цифра>}

<идентификатор>: <буква> (<буква> | <цифра>)\*

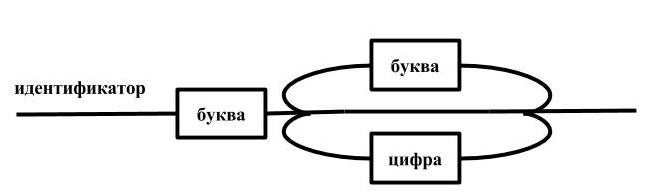


Рис. 23 - Диаграмма идентификатора.

<буква>::= *A* | *B* | *C* | *D* | *E* | *F* | *G* | *H* | *I* | *J* | *K* | *L* | *M* | *N* | *O* | *P* | *Q* | *R* | *S* | *T* |

*U* | *V* | *W* | *X* | *Y* | *Z* | *a* | *b* | *c* | *d* | *e* | *f* | *g* | *h* | *i* | *j* | *k* | *l* | *m* | *n* | *o* | *p* *q* | *r* | *s* | *t* | *u* | *v* | *w* | *x* | *y*

<буква>: *A* | *B* | *C* | *D* | *E* | *F* | *G* | *H* | *I* | *J* | *K* | *L* | *M* | *N* | *O* | *P* | *Q* | *R* | *S* | *T* |

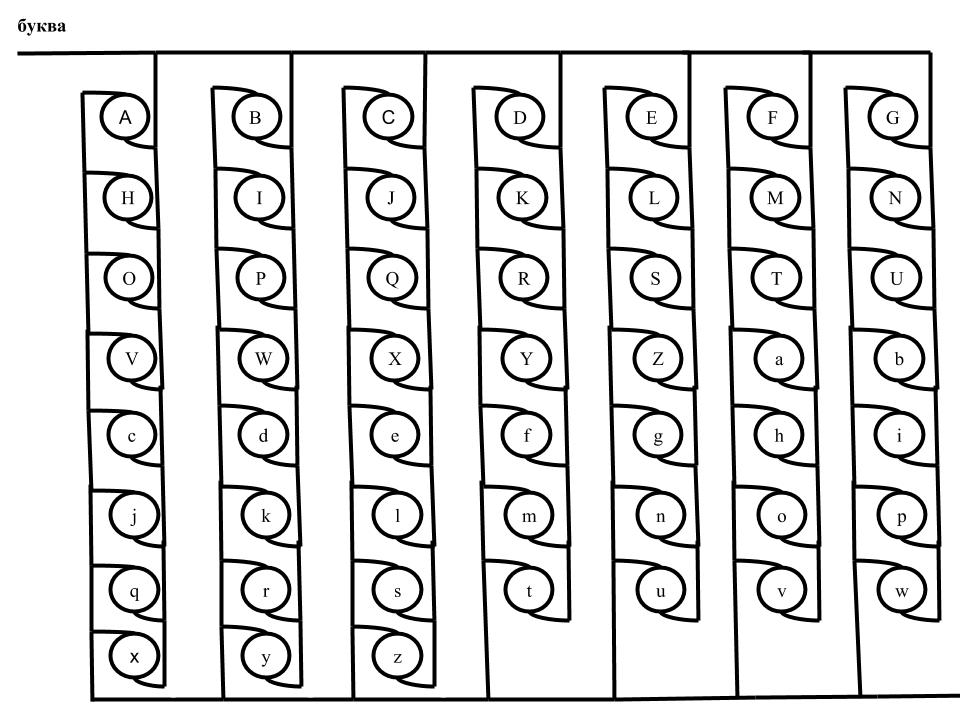
*U* | *V* | *W* | *X* | *Y* | *Z* | *a* | *b* | *c* | *d* | *e* | *f* | *g* | *h* | *i* | *j* | *k* | *l* | *m* | *n* | *o* | *p* *q* | *r* | *s* | *t* | *u* | *v* | *w* | *x* | *y* 

Рис. 24 - Диаграмма буквы.

<цифра>::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<цифра>: 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

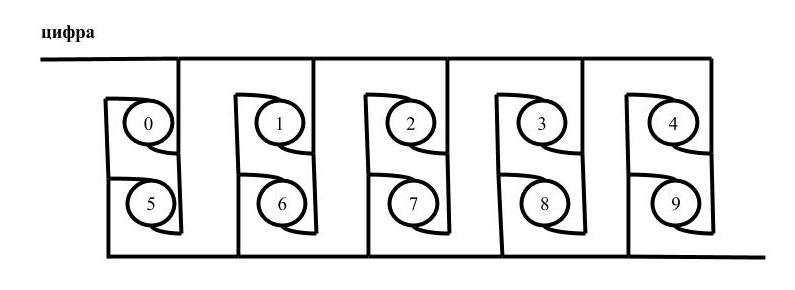


Рис. 25 - Диаграмма цифры.

Правила, определяющие целые числа:

<целое>::= <двоичное> | <восьмеричное> | <десятичное> |

<шестнадцатеричное>

<целое>: <двоичное> | <восьмеричное> | <десятичное> |

<шестнадцатеричное>

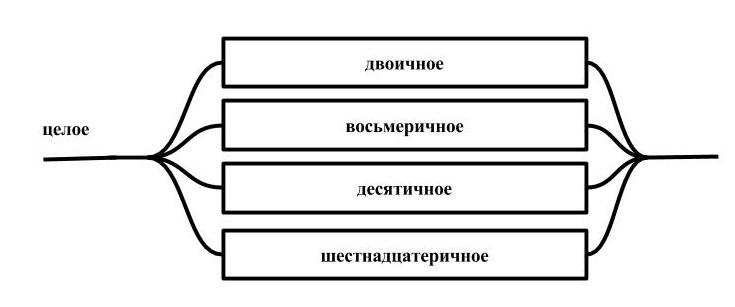


Рис. 26 - Диаграмма целого числа.

<двоичное>::= {/ 0 | 1 /} (*B* | *b*)

<двоичное>: ( 0 | 1)+ (*B* | *b*)

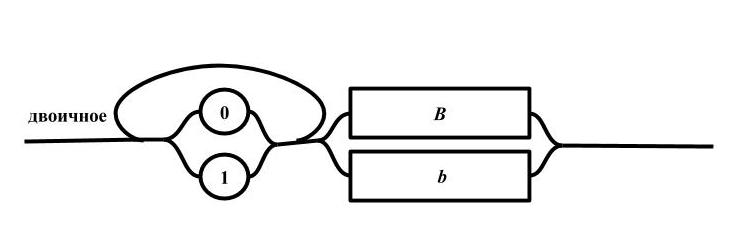
****

Рис. 27 - Диаграмма двоичного числа.

<восьмеричное>::= {/ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 /} (*O* | *o)*

<восьмеричное>: ( 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 )+ (*O* | *o)*

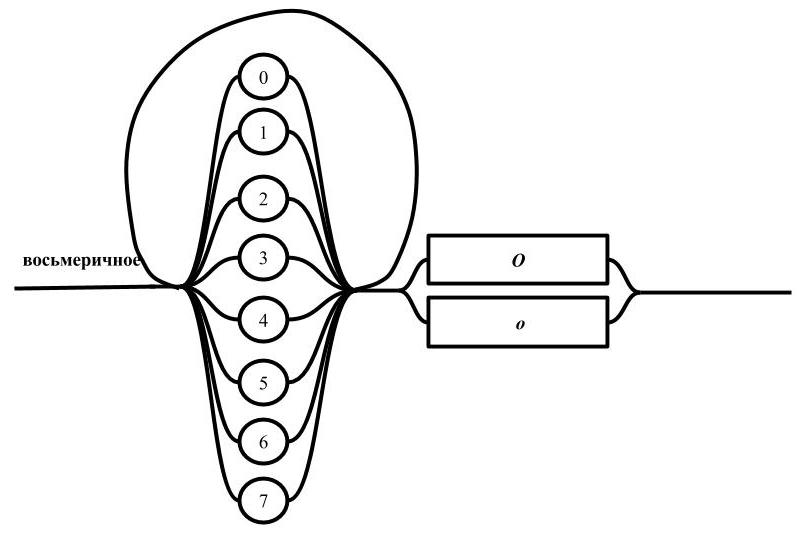


Рис. 28 - Диаграмма восьмеричного числа.

<десятичное>::= {/ <цифра> /} [*D* | *d*]

<десятичное>: ( <цифра> /} (*D* | *d*)?

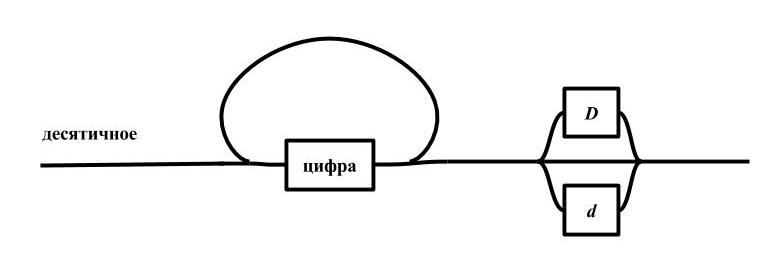


Рис. 29 - Диаграмма десятичного числа.

<шестнадцатеричное>::= <цифра> {<цифра> | *A* | *B* | *C* | *D* | *E* | *F* | *a* | *b* |

*c* | *d* | *e* | *f*} (*H* | *h*)

<шестнадцатеричное>: <цифра> (<цифра> | *A* | *B* | *C* | *D* | *E* | *F* | *a* | *b* |

*c* | *d* | *e* | *f*)\* (*H* | *h*)

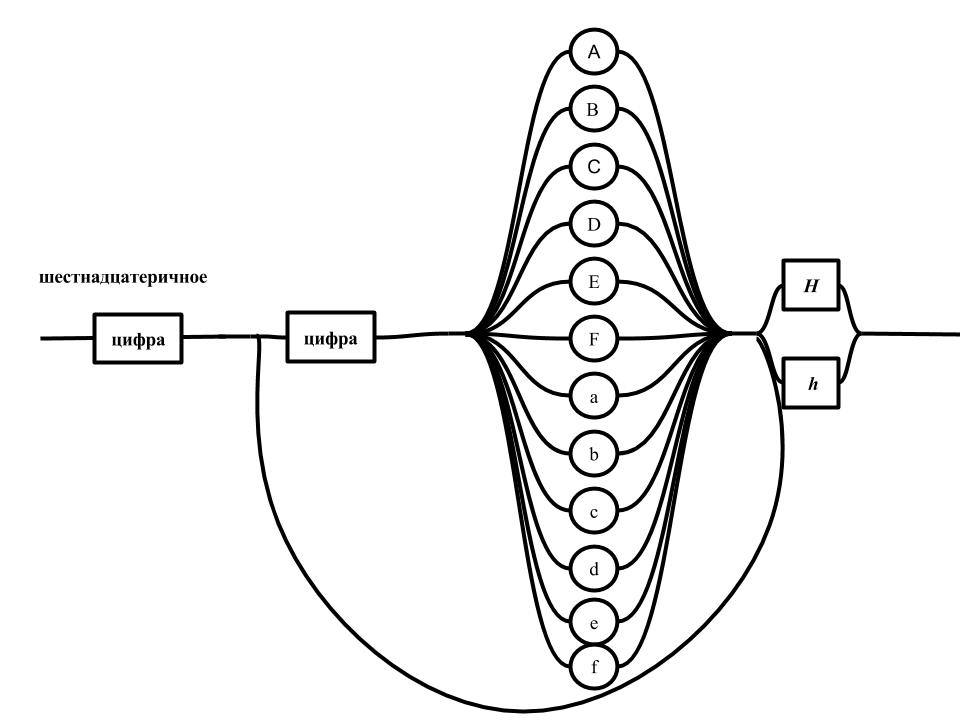


Рис. 30 - Диаграмма шестнадцатеричного числа.

Правила, описывающие действительные числа:

<действительное>::= <числовая\_строка> <порядок> |

[<числовая\_строка>] . <числовая\_строка> [порядок]

<действительное>: <числовая\_строка> <порядок> |

(<числовая\_строка>)?\ . <числовая\_строка> (порядок)?

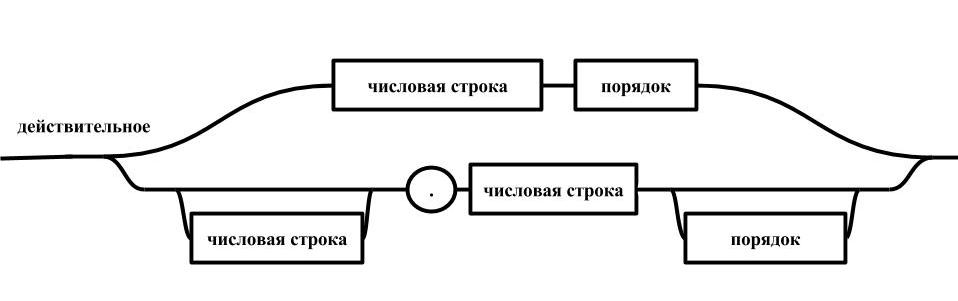


Рис. 31 - Диаграмма действительного числа.

<числовая\_строка>::= {/ <цифра> /}

<числовая\_строка>: ( <цифра> )+

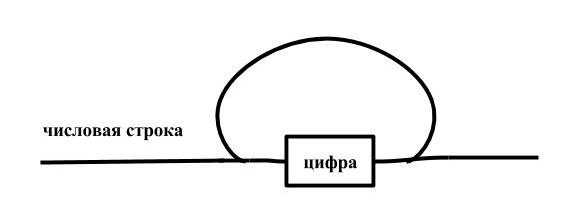


Рис. 32 - Диаграмма числовой строки.

<порядок>::= ( E | e )[+ | -] <числовая\_строка>

<порядок>: ( E | e )(+ | -)? <числовая\_строка>

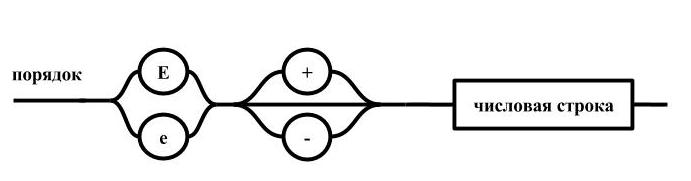


Рис. 33 - Диаграмма порядка.

Разделители:

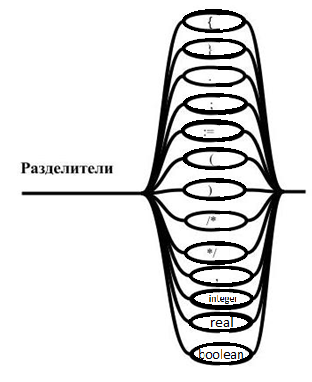


Рис. 34 - Диаграмма разделителей.

Ключевые слова:

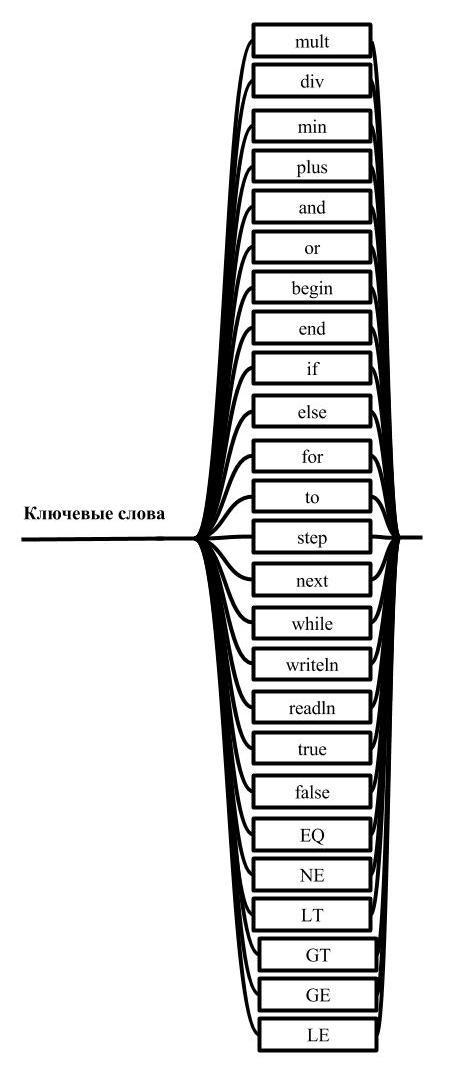


Рис. 35 - Диаграмма ключевых слов.

**Описание программы**

Разработанная программа состоит из нескольких модулей:

1. Лексический анализатор, созданный генератором синтаксических анализаторов;
2. Синтаксический анализатор, созданный генератором синтаксических анализаторов;
3. Семантический анализатор и интерпретатор, реализующий абстрактный класс, созданный генератором синтаксических анализаторов.
4. Описание синтаксиса модельного языка;
5. Модуль, использующий сгенерированные лексический и синтаксический анализаторы.
6. Модуль, отвечающие за тестирование каждого из вышеописанных модулей.

**Описание настройки и начала работы с ANTLR**

ANTLR (ANother Tool for Language Recognition) — генератор нисходящих анализаторов для формальных языков. Он используется для разработки компиляторов, интерпретаторов и трансляторов. Генератор широко используется в науке и промышленности для создания всевозможных языков, инструментов и механизмов. Поиск Twitter использует ANTLR для синтаксического анализа запросов, с более чем 2 миллиардами запросов в день. Oracle использует ANTLR в интегрированной среде разработки SQL Developer. IDE NetBeans разбирает C ++ с ANTLR. Так же ANTLR используется в реализации языков программирования Groovy, Jython, Processing, Apex.

ANTLR имеет визуальную среду разработки ANTLR Studio, плагин к среде разработки Eclipse, а также плагин для платформы Maven, что упрощает разработку.

В данной курсовой работе использовался плагин для платформы Maven. Для настройки и использования ANTLR с его помощью необходимо выполнить следующие действия:

1. Добавить в pom.xml проекта зависимость для библиотеки ANTLR:

<**dependency**>  
 <**groupId**>org.antlr</**groupId**>  
 <**artifactId**>antlr4</**artifactId**>  
 <**version**>4.3</**version**>  
</**dependency**>

1. Добавить в pom.xml проекта конфигурацию плагина ANTLR:

<**plugin**>  
 <**groupId**>org.antlr</**groupId**>  
 <**artifactId**>antlr4-maven-plugin</**artifactId**>  
 <**version**>4.3</**version**>  
 <**executions**>  
 <**execution**>  
 <**id**>antlr</**id**>  
 <**goals**>  
 <**goal**>antlr4</**goal**>  
 </**goals**>  
 </**execution**>  
 </**executions**>  
 <**configuration**>  
 <**arguments**>  
 <**argument**>-visitor</**argument**>  
 </**arguments**>  
 </**configuration**>  
</**plugin**>

1. Добавить описание грамматики модельного языка в файл с расширением g4: antlr4\Language.g4
2. Выполнить с помощью командной строки из корня проекта команду «mvn antlr4:antlr4». После удачного выполнения команды будут сгенерированны классы лексического и синтаксического анализаторов.
3. Использовать анализаторы для проведения анализа.

**Принципы работы ANTLR**

Из формального описания языка, называемого грамматикой, ANTLR генерирует синтаксический анализатор (парсер) для этого языка, который может автоматически строить деревья синтаксического анализа. Данные деревья представляют собой структуры данных, отображающие то, как вводимые лексемы сопоставляются с грамматикой. ANTLR также автоматически генерирует инструменты для обхода деревьев, которые вы можете использовать для посещения узлов этих деревьев для выполнения кода приложения.

Данный генератор парсеров является LL(\*)-парсеров. Он анализирует входной поток слева направо, и строит левый вывод грамматики.

ANTLR генерируется также и лексические анализаторы (лексеры). Лексер из из потока символов выделяет лексемы(токены), а парсер проверяет соответствие последовательности лексем грамматике языка. Результатом чего и является дерево разбора (синтаксическое дерево).

ANTLR на вход принимает файл с описанной в нем грамматикой языка для генерации лексера и парсера. Т.е. данный файл содержит как лексические, так и грамматические правила.

Каждое лексическое правило описывает лексему с помощью регулярного выражения. Например:

BIN : ('0'|'1')+ ('b'|'B').

Лексические правила начинаются с заглавной буквы.

Синтаксические правила более сложные. Они включают в себя другие правила. Синтаксические правила начинаются со строчной буквы. Пример:

parse

: '{' block '}' EOF

;

block

: stat\*

;

В каждом файле с описанием грамматики должно быть как минимум одно лексическое правило и одно грамматическое.

После генерации классов лексера и парсера, их можно использовать для анализа входного текста. С их помощью можно:

1. Провести лексический анализ для разбиения текста на лексемы
2. Построить абстрактное синтаксическое дерево из лексем
3. Получить доступ к абстрактному синтаксическому дереву
4. Вывести абстрактное синтаксическое дерево в консоль или схематично его построить при использовании специальных инструментов. Пример:

0 null

-- 4 Hello word

-- 5 !

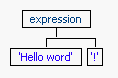


Рис. 36 – Построение абстрактного синтаксического дерева ANTLR

ANTLR также генерирует абстрактный класс LanguageVisitor, который может быть расширен для создания «посетителя» для обработки всех грамматический конструкций. С помощью реализации этого класса можно произвести семантический анализ и интерпретацию. Например, для семантического анализа и интерпретации цикла for необходимо переопределить метод visitFor\_stat, в котором и будут проводиться все необходимые операции.

**Структура классов**

Анализаторы, сгенерированные ANTLR, будут доступны в папке target\generated-sources\antlr4.

Для каждой грамматики, описанной в файлах g4, будут сгенерированы классы анализаторов с префиксами, соответствующими названию грамматики.

Класс MyLangLexer содержит лексический анализатор.

Класс MyLangParser содержит синтаксический анализатор.

Класс MyLangBaseVisitor представляет собой абстрактный класс, который может использоваться для дальнейшей обработки всех встреченный во входном тексте грамматических конструкций.

**Алгоритм решения задачи**

Логическая схема работы с использованием генератора синтаксических анализаторов:

**Семантический анализатор**

В работе реализованы следующие виды семантических проверок:

* Проверка инициализации переменной.
* Проверка объявления переменной.
* Проверка объявления уже существующей переменной.
* Проверка совместимости типов, участвующих в выражении.
* Проверка типа значения присваиваемого переменной. При этом есть ряд особенностей:
  + Переменным вещественного типа могут присваиваться числа и переменные типа целочисленного.
  + Переменным целочисленного типа могут присваиваться переменные вещественного типа, но значения которых являются целочисленными.
* Анализ бесконечного цикла. Компилятор сообщает о том есть ли в цикле for условие выполнения, влекущего за собой бесконечную петлю.
* Область видимости переменных.

**Особенности формального языка**

Архитектура формального языка небезупречна, в следствии чего приходится предпринимать ряд допущений. В цикле for у нас имеется оператор присваивания значения переменной управляющей циклом, но при этом конструкция не подразумевает её инициализацию, что, например, в языке программирования java реализовано. Исходя из этого мы может пойти двумя путями: заставлять пользователя инициализировать переменную в блоке описания или инициализировать переменную в блоке присваивания цикла for автоматически в виде исключения из правил. В работе используется второй вариант, поскольку заставлять пользователя думать о чём-то – плохая идея. Но отсюда рождается другое ограничение: нельзя в цикле for в качестве управляющей использовать уже существующую переменную, поскольку она будет повторно объявлена, что за собой повлечёт, срабатывание правила о повторном объявлении уже существующей переменной. Несмотря на это, второй вариант остаётся более предпочтительным поскольку, если мы знаем, что переменная будет использоваться только внутри цикла, то без проблем можем сделать для неё область видимости только внутри цикла. Делается это простым сокрытием переменной после того как прекращается выполнение цикла.

Числа представленные в системах счисления отличных от десятеричной, по умолчанию обрабатываются и заносятся в память как десятеричные. Например, если присвоить a:=101b и вывести переменную а на экран с помощью команды «writeln a», то результат будет равен 5.

**Пример работы программы**

Для проверки сгенерированных анализаторов было составлено несколько примеров кода модельного языка. Примеры и результаты работы программы представлены на рисунках.

**Лексический анализ**

1. Текст программы:

{

a,b,c,perimetr, hex, notreal: real;  
a:=171o;  
a:=171O;  
real:=141e1;  
real:=min .3e-1;  
hex:=min A13A4h;  
hex:=min123h;  
a:=101010b;  
a:=101010B;  
perimetr:=34.3e-1b;  
perimetr:=191o;  
perimetr:=123b;  
perimetr:=123B;  
perimetr:=12A;  
perimetr:=12Ad;  
}

Результат работы:

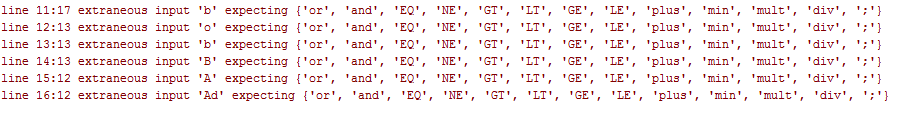


Рис. 37 – Результат тестирования лексического анализа.

1. Текст программы:

{  
1abc,b,c,2perimetr, hex, notreal: real;  
}

Результат работы:



Рис. 38 – Результат тестирования лексического анализа.

1. Текст программы:

{  
1abc,b,c,2perimetr, hex, notreal: real,  
1abc,b,c,2perimetr, hex, notreal: real)  
}

Результат работы:

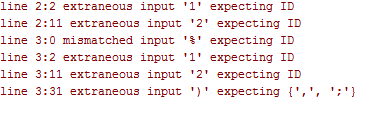


Рис. 39 – Результат тестирования лексического анализа.

**Синтаксический анализ**

1. Текст программы:

{  
a,b: boolean;  
c,perimeter: real;  
d integer;  
}

Результат работы:



Рис. 40 – Результат тестирования синтаксического анализа.

1. Текст программы:

{  
a,b :real;  
c,perimetr:boolean;  
  
a := ~true;  
b := false;  
  
while (b EQ a )  
begin  
 writeln Done;  
 writeln a;  
  
 begin  
 writeln a plus b;  
 a:=b;  
 end  
  
}

Результат работы:



Рис. 41 – Результат тестирования синтаксического анализа.

1. Текст программы:

{  
a,b: real;  
c,perimeter: boolean;  
  
a := ~true;  
b := false;  
  
if a and b  
 writeln b;  
else  
begin  
 writeln a plus b;  
 a:=b;  
end  
else  
begin  
 writeln a plus b;  
 a:=b;  
end  
  
}

Результат работы:



Рис. 42 – Результат тестирования синтаксического анализа.

1. Текст программы:

{  
a,b: real;  
c,perimeter: boolean;  
  
a := ~true;  
b := false;  
  
for a:=1 to 5 plus 1 step 1  
 writeln a;  
 writeln Done;  
  
 while (b GT a)  
 writeln b;  
  
 for a:=1 to 5 plus 1 step 1  
 writeln a;  
next  
  
}

Результат работы:



Рис. 43 – Результат тестирования синтаксического анализа.

1. Текст программы:

{  
for a to 5 plus 1 step 1  
 writeln a;  
next  
}

Результат работы:



Рис. 44 – Результат тестирования синтаксического анализа.

**Семантический анализ**

1. Текст программы:

{  
a: booblean;  
b: real;  
  
a:=true;  
b:=11;  
a:=b;  
  
}

Проверка на содержание ошибки:

@Test  
**public void** testIncorrectAssignment() {  
 setUp(**"semantics/testIncorrectAssignment.txt"**);  
 **try** {  
 **visitor**.visit(**tree**);  
 } **catch** (RuntimeException ex) {  
 Assert.*assertTrue*(ex.getMessage().contains(**"Incompatible value"**));  
 }  
}

Результат работы:



Рис. 45 – Результат тестирования семантического анализа.

1. Текст программы:

{  
a: integer;  
b: real;  
  
b:=11;  
  
for a:=1 to 5 plus 1 step 1  
 writeln a;  
next  
  
}

Проверка на содержание ошибки:

@Test(expectedExceptions = SemanticAnalyzerException.**class**, expectedExceptionsMessageRegExp  
 = **"The variable .\* already exists"**)  
**public void** testAlreadyExistingVariableInForCircle() {  
 setUp(**"semantics/testAlreadyExistingVariableInForCircle.txt"**);  
 **visitor**.visit(**tree**);  
}

Результат работы:



Рис. 46 – Результат тестирования семантического анализа.

1. Текст программы:

{  
  
b:integer;  
  
a:=true;  
b:=11;  
  
}

Проверка на содержание ошибки:

@Test  
**public void** testUseUndefinedVariable() {  
 setUp(**"semantics/testUseUndefinedVariable.txt"**);  
 **try** {  
 **visitor**.visit(**tree**);  
 } **catch** (RuntimeException ex) {  
 Assert.*assertTrue*(ex.getMessage().contains(**"use of an undefined variable"**));  
 }  
}

Результат работы:



Рис. 47 – Результат тестирования семантического анализа.

1. Текст программы:

{  
a, b: real;  
  
a:= b plus 10;  
}

Проверка на содержание ошибки:

@Test  
**public void** testUseUninitializedVariable() {  
 setUp(**"semantics/testUseUninitializedVariable.txt"**);  
 **try** {  
 **visitor**.visit(**tree**);  
 } **catch** (RuntimeException ex) {  
 Assert.*assertTrue*(ex.getMessage().contains(**"use of an uninitialized variable"**));  
 }  
}

Результат работы:



Рис. 48 – Результат тестирования семантического анализа.

1. Текст программы:

{  
for a:=10 to 1 step 1  
 writeln a;  
next  
}

Проверка на содержание ошибки:

@Test  
**public void** testInfinityLoopInPositiveDirection() {  
 setUp(**"semantics/testInfinityLoopInPositiveDirection.txt"**);  
 **visitor**.visit(**tree**);  
 **visitor**.printErrors();  
 Assert.*assertTrue*(**visitor**.getErrors().get(0).contains(**"infinite loop"**));  
}

Результат работы:



Рис. 49 – Результат тестирования семантического анализа.

**Интерпретация**

1. Текст программы:

{  
a:integer;  
b:integer;  
c, d: real;  
  
  
a:=123;  
b:=(a plus 150) mult 10 plus ((127 min 65) div 1);  
c:=12.3;  
d:=(c plus 15.0) mult 10 plus ((12.7 min 6.5) div 1.0);  
}

Проверка на корректность вычислений:

@Test  
**public void** testIntegerCalculation() {  
 setUp(**"interpretation/testIntegerCalculation.txt"**);  
 **visitor**.visit(**tree**);  
  
 Assert.*assertEquals*(**visitor**.**IDValueMemory**.get(**"b"**).toString(), **"2792"**);  
 Assert.*assertEquals*(**visitor**.**IDValueMemory**.get(**"d"**).toString(), **"279.2"**);  
}

Результат работы:



Рис. 50 – Результат тестирования интерпретации.

1. Текст программы:

{  
a, b, c, d, e:integer;  
  
a:=123;  
b:=99;  
c:=32;  
d:= 56;  
  
if ((a plus c) EQ (b plus d))  
 begin  
 e:=155;  
 end  
else  
 begin  
 e:=0;  
 end  
}

Проверка на корректность вычислений:

@Test  
**public void** testIfStatement() {  
 setUp(**"interpretation/testIfStatement.txt"**);  
 **visitor**.visit(**tree**);  
  
 Assert.*assertEquals*(**visitor**.**IDValueMemory**.get(**"e"**).toString(), **"155"**);  
}

Результат работы:



Рис. 51 – Результат тестирования интерпретации.

1. Текст программы:

{  
i, number, factorial:integer;  
i:=1;  
number:=4;  
factorial:=1;  
  
while ( i LE number)  
 begin  
 factorial:= factorial mult i;  
 i:=i plus 1;  
 end  
}

Проверка на корректность вычислений:

@Test  
**public void** testWhileStatement() {  
 setUp(**"interpretation/testWhileStatement.txt"**);  
 **visitor**.visit(**tree**);  
  
 Assert.*assertEquals*(**visitor**.**IDValueMemory**.get(**"factorial"**).toString(), **"24"**);  
}

Результат работы:



Рис. 52 – Результат тестирования интерпретации.

1. Текст программы:

{  
number, factorial:real;  
number:=4;  
factorial:=1;  
  
for i:=1 to number step 1  
 begin  
 factorial:= factorial mult i;  
 end  
next  
}

Проверка на корректность вычислений:

@Test  
**public void** testForStatement() {  
 setUp(**"interpretation/testForStatement.txt"**);  
 **visitor**.visit(**tree**);  
  
 Assert.*assertEquals*(**visitor**.**IDValueMemory**.get(**"factorial"**).toString(), **"24.0"**);  
}

Результат работы:



Рис. 53 – Результат тестирования интерпретации.

5) Текст программы:

{  
a, b, c, d:boolean;  
  
a:=13 NE 15;  
b:=14.4 NE 14.4;  
  
c:= a and b;  
d:=a or b;  
}

Проверка на корректность вычислений:

@Test  
**public void** testLogicalOperators() {  
 setUp(**"interpretation/testLogicalOperators.txt"**);  
 **visitor**.visit(**tree**);  
  
 Assert.*assertEquals*(**visitor**.**IDValueMemory**.get(**"c"**).toString(), **"false"**);  
 Assert.*assertEquals*(**visitor**.**IDValueMemory**.get(**"d"**).toString(), **"true"**);  
}

Результат работы:



Рис. 54 – Результат тестирования интерпретации.

**Вывод**

В ходе выполнения курсовой работы были детально рассмотрены этапы работы компилятора. Были изучены принципы работы и возможности генераторов синтаксических анализаторов. Были сгенерированы лексический, синтаксический, семантический анализаторы кода и интерпретатор с помощью генератора синтаксических анализаторов ANTLR.

Для описания модельного языка были использованы формы Бэкуса-Наура, регулярные выражения и диаграммы Вирта.

**Приложение**

Листинг класса MyLang.g4:

**grammar MyLang;**

**parse**

**: '{' (stat | definition)+ '}' EOF**

**;**

**block**

**: stat\***

**;**

**stat**

**: assignment SCOL**

**| composite\_operator**

**| if\_stat**

**| while\_stat**

**| for\_stat**

**| output**

**| input**

**| OTHER {System.err.println("unknown char: " + $OTHER.text);}**

**;**

**composite\_operator**

**: begin=BEGIN block end=END**

**;**

**definition**

**: var1=ID (',' var2=ID)\* ':' type SCOL**

**;**

**assignment**

**: ID ASSIGN expr**

**;**

**if\_stat**

**: IF condition\_block (ELSE stat)?**

**;**

**condition\_block**

**: expr stat**

**;**

**while\_stat**

**: WHILE LPAR expr RPAR stat**

**;**

**for\_stat**

**: FOR assignment TO expr (STEP expr)? block NEXT**

**;**

**output**

**: WRITELN expr SCOL**

**;**

**input**

**: READLN ID SCOL**

**;**

**expr**

**: MINUS expr #unaryMinusExpr**

**| NOT expr #notExpr**

**| expr op=(MULT | DIV ) expr #multiplicationExpr**

**| expr op=(PLUS | MINUS) expr #additiveExpr**

**| expr op=(LTEQ | GTEQ | LT | GT) expr #relationalExpr**

**| expr op=(EQ | NEQ) expr #equalityExpr**

**| expr AND expr #andExpr**

**| expr OR expr #orExpr**

**| atom #atomExpr**

**;**

**atom**

**: LPAR expr RPAR #parExpr**

**| (n=NUMBER) #numberAtom**

**| (TRUE | FALSE) #booleanAtom**

**| ID #idAtom**

**;**

**OR : 'or';**

**AND : 'and';**

**EQ : 'EQ';**

**NEQ : 'NE';**

**GT : 'GT';**

**LT : 'LT';**

**GTEQ : 'GE';**

**LTEQ : 'LE';**

**PLUS : 'plus';**

**MINUS : 'min';**

**MULT : 'mult';**

**DIV : 'div';**

**NOT : '~';**

**SCOL : ';';**

**ASSIGN : ':=';**

**LPAR : '(';**

**RPAR : ')';**

**BEGIN : 'begin';**

**END : 'end';**

**TRUE : 'true';**

**FALSE : 'false';**

**IF : 'if';**

**ELSE : 'else';**

**WHILE : 'while';**

**WRITELN : 'writeln';**

**READLN : 'readln';**

**TO : 'to';**

**NEXT : 'next';**

**STEP : 'step';**

**FOR : 'for';**

**ID**

**: [a-zA-Z\_] [a-zA-Z\_0-9]\***

**;**

**type**

**: 'real'**

**| 'boolean'**

**| 'integer'**

**;**

**NUMBER**

**: INTEGER**

**| REAL**

**;**

**INTEGER**

**: DEC**

**| BIN**

**| OCT**

**| HEX**

**;**

**BIN : ('0'|'1')+ ('b'|'B');**

**OCT : ('0'..'7')+ ('o'|'O');**

**DEC : ('0'..'9')+ ;**

**HEX : ('0'..'9'|'a'..'f'|'A'..'F')+ ('h'|'H');**

**REAL**

**: DEC? ('.' DEC)**

**| DEC? ('.' DEC) MANTISSA**

**| DEC MANTISSA**

**;**

**MANTISSA : E ('+'|'-')? DEC ;**

**E : 'E' | 'e' ;**

**COMMENT : '/\*' (.)\*? '\*/' -> skip;**

**SPACE**

**: [ \t\r\n] -> skip**

**;**

**OTHER**

**: .**

**;**

Листинг класса Types:

**public enum** Types {  
  
 ***REAL***, ***BOOLEAN***, ***INTEGER***;  
  
  
 **public static** Types getType(String name) {  
 **if** (name.equals(EvalVisitor.***REAL\_TYPE***)) {  
 **return *REAL***;  
 } **else if** (name.equals(EvalVisitor.***INT\_TYPE***)) {  
 **return *INTEGER***;  
 } **else** {  
 **return *BOOLEAN***;  
 }  
 }  
}

Листинг класса Value:

**public class** Value {  
  
 **public static** Value *VOID* = **new** Value(**new** Object());  
  
 **final** Object **value**;  
  
 **public** Value(Object value) {  
 **this**.**value** = value;  
 }  
  
 **public** Boolean asBoolean() {  
 **return** (Boolean) **value**;  
 }  
  
 **public** Double asDouble() {  
 **return** Double.*valueOf*(**value**.toString());  
 }  
  
 **public** Integer asInteger() {  
 **return** (Integer) **value**;  
 }  
  
 **public** String asString() {  
 **return** String.*valueOf*(**value**);  
 }  
  
 **public boolean** isDouble() {  
 **return value instanceof** Double;  
 }  
  
 **public boolean** isBoolean() {  
 **return value instanceof** Boolean;  
 }  
  
 **public boolean** isInteger() {  
 **return value instanceof** Integer;  
 }  
  
 **public boolean** lessThan(Double than) {  
 **return** (Double) **value** < than;  
 }  
  
 @Override  
 **public int** hashCode() {  
  
 **if** (**value** == **null**) {  
 **return** 0;  
 }  
  
 **return this**.**value**.hashCode();  
 }  
  
 @Override  
 **public boolean** equals(Object o) {  
  
 **if** (**value** == o) {  
 **return true**;  
 }  
  
 **if** (**value** == **null** || o == **null** || o.getClass() != **value**.getClass()) {  
 **return false**;  
 }  
  
 Value that = (Value) o;  
  
 **return this**.**value**.equals(that.**value**);  
 }  
  
 @Override  
 **public** String toString() {  
 **return** String.*valueOf*(**value**);  
 }  
}

Листинг класса EvalVisitor:

import exceptions.SemanticAnalyzerException;

import org.antlr.v4.runtime.misc.NotNull;

import org.antlr.v4.runtime.tree.TerminalNode;

import org.mvel2.MVEL;

import java.util.\*;

import java.util.regex.Matcher;

import java.util.regex.Pattern;

public class EvalVisitor extends MyLangBaseVisitor<Value> {

// used to compare floating point numbers

public static final String WARNING = "WARNING: ";

public static final String ERROR = "ERROR: ";

public static final String DEBUG = "DEBUG: ";

public static final String THE\_USE\_OF\_AN\_UNINITIALIZED\_VARIABLE = " use of an uninitialized variable: ";

public static final String THE\_USE\_OF\_AN\_UNDEFINED\_VARIABLE = " use of an undefined variable: ";

public static final String REAL\_TYPE = "real";

public static final String INT\_TYPE = "integer";

public static final String BOOLEAN\_TYPE = "boolean";

protected Map<String, Value> IDValueMemory = new HashMap<String, Value>();

protected Map<String, Types> IDTypeMemory = new HashMap<String, Types>();

protected Map<String, Integer> IDScopeMemory = new HashMap<String, Integer>();

private List<String> errors = new ArrayList<String>();

private List<String> debug = new ArrayList<String>();

private boolean forAssignment;

private int scope = 0;

public static boolean isInteger(String s) {

return isInteger(s, 10);

}

public static boolean isInteger(String s, int radix) {

if (s.isEmpty()) return false;

for (int i = 0; i < s.length(); i++) {

if (i == 0 && s.charAt(i) == '-') {

if (s.length() == 1) return false;

else continue;

}

if (Character.digit(s.charAt(i), radix) < 0) return false;

}

return true;

}

// assignment/id overrides

@Override

public Value visitAssignment(MyLangParser.AssignmentContext ctx) {

String id = ctx.ID().getText();

Value value = this.visit(ctx.expr());

if (forAssignment) {

debug.add(DEBUG + "definition: id=" + id + " type - " + REAL\_TYPE + " scope - " + scope);

forAssignment = false;

checkIDExistence(id);

IDTypeMemory.put(id, Types.REAL);

} else {

checkIDDefinition(id);

}

debug.add(DEBUG + "id: " + id + " value:" + value);

validateType(id, value);

IDValueMemory.put(id, value);

return value;

}

private void validateType(String id, Value value) {

boolean isValid = false;

Types type = IDTypeMemory.get(id);

switch (type) {

case INTEGER:

isValid = value.isInteger();

break;

case BOOLEAN:

isValid = value.isBoolean();

break;

case REAL:

isValid = value.isDouble() || value.isInteger();

break;

}

if (!isValid) {

throw new SemanticAnalyzerException("Incompatible value " + value.toString() + " for ID " + id);

}

}

private void checkIDExistence(String id) {

if (IDTypeMemory.get(id) != null) {

throw new SemanticAnalyzerException("The variable \"" + id + "\" already exists");

}

}

private void checkIDDefinition(String id) {

if (IDTypeMemory.get(id) == null) {

errors.add(ERROR + THE\_USE\_OF\_AN\_UNDEFINED\_VARIABLE + id);

throw new SemanticAnalyzerException(THE\_USE\_OF\_AN\_UNDEFINED\_VARIABLE + id);

}

}

private Value checkIDInitialize(String id) {

Value value = IDValueMemory.get(id);

if (value == null) {

errors.add(ERROR + THE\_USE\_OF\_AN\_UNINITIALIZED\_VARIABLE + id);

throw new SemanticAnalyzerException(ERROR + THE\_USE\_OF\_AN\_UNINITIALIZED\_VARIABLE + id);

}

return value;

}

@Override

public Value visitComposite\_operator(MyLangParser.Composite\_operatorContext ctx) {

scope++;

visitChildren(ctx);

removeOverdueID();

scope--;

return Value.VOID;

}

@Override

public Value visitDefinition(MyLangParser.DefinitionContext ctx) {

for (TerminalNode id : ctx.ID()) {

String idName = id.getText();

String idType = ctx.type().getText();

checkIDExistence(idName);

IDTypeMemory.put(idName, Types.getType(idType));

IDScopeMemory.put(idName, scope);

debug.add(DEBUG + "definition: id=" + idName + " type - " + idType + " scope - " + scope);

}

return Value.VOID;

}

@Override

public Value visitIdAtom(MyLangParser.IdAtomContext ctx) {

String id = ctx.getText();

checkIDDefinition(id);

return checkIDInitialize(id);

}

// atom overrides

@Override

public Value visitNumberAtom(MyLangParser.NumberAtomContext ctx) {

String value = ctx.getText();

if (isBin(value)) {

return new Value(Integer.parseInt(value.substring(0, value.length() - 1), 2));

}

if (isOct(value)) {

return new Value(Integer.parseInt(value.substring(0, value.length() - 1), 8));

}

if (isHex(value)) {

return new Value(Integer.parseInt(value.substring(0, value.length() - 1), 16));

}

if (isInteger(value)) {

return new Value(Integer.valueOf(value));

}

return new Value(Double.valueOf(value));

}

private boolean isBin(String value) {

Matcher matcher = Pattern.compile("^[0-1]\*(B|b)$").matcher(value);

return matcher.matches();

}

private boolean isOct(String value) {

Matcher matcher = Pattern.compile("^[0-7]\*(O|o)$").matcher(value);

return matcher.matches();

}

private boolean isHex(String value) {

Matcher matcher = Pattern.compile("^[0-9A-Fa-f]\*(H|h)$").matcher(value);

return matcher.matches();

}

@Override

public Value visitBooleanAtom(MyLangParser.BooleanAtomContext ctx) {

return new Value(Boolean.valueOf(ctx.getText()));

}

// expr overrides

@Override

public Value visitParExpr(MyLangParser.ParExprContext ctx) {

return this.visit(ctx.expr());

}

@Override

public Value visitUnaryMinusExpr(MyLangParser.UnaryMinusExprContext ctx) {

Value value = this.visit(ctx.expr());

return new Value(-value.asDouble());

}

@Override

public Value visitNotExpr(MyLangParser.NotExprContext ctx) {

Value value = this.visit(ctx.expr());

return new Value(!value.asBoolean());

}

@Override

public Value visitMultiplicationExpr(@NotNull MyLangParser.MultiplicationExprContext ctx) {

Value left = this.visit(ctx.expr(0));

Value right = this.visit(ctx.expr(1));

if (!(isValidType(left) & isValidType(right))) {

throw new SemanticAnalyzerException("wrong type in expr: " + ctx.op.getLine());

}

switch (ctx.op.getType()) {

case MyLangParser.MULT:

return new Value(MVEL.eval(left.toString() + '\*' + right.toString()));

case MyLangParser.DIV:

if (left.isInteger() && right.isInteger()) {

return new Value(left.asInteger() / right.asInteger());

}

return new Value(MVEL.eval(left.toString() + '/' + right.toString()));

default:

throw new SemanticAnalyzerException("unknown operator: " + MyLangParser.tokenNames[ctx.op.getType()]);

}

}

private boolean isValidType(Value value) {

return value.isDouble() || value.isInteger();

}

@Override

public Value visitAdditiveExpr(@NotNull MyLangParser.AdditiveExprContext ctx) {

Value left = this.visit(ctx.expr(0));

Value right = this.visit(ctx.expr(1));

if (!(isValidType(left) & isValidType(right))) {

throw new SemanticAnalyzerException("wrong type in expr, line: " + ctx.op.getLine());

}

switch (ctx.op.getType()) {

case MyLangParser.PLUS:

return new Value(MVEL.eval(left.toString() + '+' + right.toString()));

case MyLangParser.MINUS:

return new Value(MVEL.eval(left.toString() + '-' + right.toString()));

default:

throw new SemanticAnalyzerException("unknown operator: " + MyLangParser.tokenNames[ctx.op.getType()]);

}

}

@Override

public Value visitRelationalExpr(@NotNull MyLangParser.RelationalExprContext ctx) {

Value left = this.visit(ctx.expr(0));

Value right = this.visit(ctx.expr(1));

switch (ctx.op.getType()) {

case MyLangParser.LT:

return new Value(left.asDouble() < right.asDouble());

case MyLangParser.LTEQ:

return new Value(left.asDouble() <= right.asDouble());

case MyLangParser.GT:

return new Value(left.asDouble() > right.asDouble());

case MyLangParser.GTEQ:

return new Value(left.asDouble() >= right.asDouble());

default:

throw new SemanticAnalyzerException("unknown operator: " + MyLangParser.tokenNames[ctx.op.getType()]);

}

}

@Override

public Value visitEqualityExpr(@NotNull MyLangParser.EqualityExprContext ctx) {

Value left = this.visit(ctx.expr(0));

Value right = this.visit(ctx.expr(1));

switch (ctx.op.getType()) {

case MyLangParser.EQ:

return new Value(MVEL.eval(left.toString() + "==" + right.toString()));

case MyLangParser.NEQ:

return new Value(MVEL.eval(left.toString() + "!=" + right.toString()));

default:

throw new SemanticAnalyzerException("unknown operator: " + MyLangParser.tokenNames[ctx.op.getType()]);

}

}

@Override

public Value visitAndExpr(MyLangParser.AndExprContext ctx) {

Value left = this.visit(ctx.expr(0));

Value right = this.visit(ctx.expr(1));

return new Value(left.asBoolean() && right.asBoolean());

}

@Override

public Value visitOrExpr(MyLangParser.OrExprContext ctx) {

Value left = this.visit(ctx.expr(0));

Value right = this.visit(ctx.expr(1));

return new Value(left.asBoolean() || right.asBoolean());

}

// writeln override

@Override

public Value visitOutput(MyLangParser.OutputContext ctx) {

Value value = this.visit(ctx.expr());

System.out.println(value);

return value;

}

@Override

public Value visitInput(MyLangParser.InputContext ctx) {

String id = ctx.ID().getText();

Scanner keyboard = new Scanner(System.in);

String idValue = keyboard.next();

Value value = Value.VOID;

switch (IDTypeMemory.get(id)) {

case INTEGER:

value = new Value(Integer.valueOf(idValue));

break;

case BOOLEAN:

value = new Value(Boolean.valueOf(idValue));

break;

case REAL:

value = new Value(Double.valueOf(idValue));

break;

}

IDValueMemory.put(id, value);

return value;

}

// if override

@Override

public Value visitIf\_stat(MyLangParser.If\_statContext ctx) {

MyLangParser.Condition\_blockContext condition = ctx.condition\_block();

boolean evaluatedBlock = false;

Value evaluated = this.visit(condition.expr());

if (evaluated.asBoolean()) {

evaluatedBlock = true;

// evaluate this block whose expr==true

this.visit(condition.stat());

}

if (!evaluatedBlock && ctx.stat() != null) {

// evaluate the else-stat\_block (if present == not null)

this.visit(ctx.stat());

}

return Value.VOID;

}

// while override

@Override

public Value visitWhile\_stat(MyLangParser.While\_statContext ctx) {

Value value = this.visit(ctx.expr());

while (value.asBoolean()) {

// evaluate the code block

this.visit(ctx.stat());

// evaluate the expression

value = this.visit(ctx.expr());

}

return Value.VOID;

}

@Override

public Value visitFor\_stat(@NotNull MyLangParser.For\_statContext ctx) {

scope++;

forAssignment = true;

MyLangParser.AssignmentContext assignmentCtx = ctx.assignment();

List<MyLangParser.ExprContext> exprContextList = ctx.expr();

Double value = this.visit(assignmentCtx).asDouble();

Double step = exprContextList.size() > 1 ? this.visit(exprContextList.get(1)).asDouble() : 1;

Double to = this.visit(exprContextList.get(0)).asDouble();

if ((value < to && step <= 0) || (value > to && step >= 0)) {

getErrors().add(WARNING + "infinite loop ");

return Value.VOID;

}

String forID = assignmentCtx.ID().getText();

if (value > to) {

for (; value >= to; value += step) {

IDValueMemory.put(forID, new Value(value));

this.visit(ctx.block());

removeOverdueID();

}

} else if (value < to) {

for (; value <= to; value += step) {

IDValueMemory.put(forID, new Value(value));

this.visit(ctx.block());

removeOverdueID();

}

}

debug.add(DEBUG + "remove: id=" + forID);

IDTypeMemory.remove(forID);

IDValueMemory.remove(forID);

scope--;

return Value.VOID;

}

void removeOverdueID() {

Map<String, Integer> buffer = new HashMap<String, Integer>();

buffer.putAll(IDScopeMemory);

for (Map.Entry<String, Integer> entry : buffer.entrySet()) {

if (scope > 0 & entry.getValue() >= scope) {

IDScopeMemory.remove(entry.getKey());

IDTypeMemory.remove(entry.getKey());

IDValueMemory.remove(entry.getKey());

debug.add(DEBUG + "remove: id=" + entry.getKey());

}

}

}

public List<String> getErrors() {

return errors;

}

public void setErrors(List<String> errors) {

this.errors = errors;

}

public List<String> getDebug() {

return debug;

}

public void setDebug(List<String> debug) {

this.debug = debug;

}

public void printErrors() {

System.out.println();

System.out.println("Errors:");

for (String error : errors) {

System.out.println(error);

}

}

public void printDebug() {

System.out.println();

for (String entry : debug) {

System.out.println(entry);

}

}

}