Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования

«Новосибирский Государственный Технический Университет»

Факультет автоматики и вычислительной техники

Кафедра вычислительной техники

**Пояснительная записка**

**к курсовой работе по дисциплине "Теория формальных языков и компиляторов"**

*Тема: " Разработка интерпретатора для учебного языка программирования "*

**Факультет:** АВТФ

**Группа:** АВТ-610

**Студент:** Дунаев Н. Ю.

**Вариант:** 2131143

**Преподаватель:** Малявко А.А.

Новосибирск 2019

# Задание на курсовую работу

1.    Написать несколько простых тестовых программ, содержащих все заданные элементы и управляющие конструкции языка. Эти программы использовать впоследствии для проверки элементов разрабатываемого транслятора. Описать лексику, синтаксис и семантику заданного варианта языка.

2.    Разработать систему регулярных выражений, определяющую лексику заданного варианта языка. Используя пакет Вебтранслаб, построить автоматную реализацию лексического анализатора на выбранном инструментальном языке, добиться его работоспособности.

3.    Разработать формальную грамматику класса LL(1), определяющую синтаксис заданного языка.

**Или:**

Разработать формальную грамматику класса не выше, чем LALR(1) , определяющую синтаксис заданного языка.

Используя пакет Вебтранслаб, построить автоматную реализацию синтаксического акцептора, добиться его работоспособности на тестовых программах.

4.    Разработать совокупность действий для расширения синтаксического акцептора, выполняющего преобразование входной последовательности лексем в постфиксную форму записи (ПФЗ) или в абстрактное синтаксическое дерево (АСД).

5.    Разработать семантический анализатор, преобразователь ПФЗ (или АСД) в псевдокод  и интерпретатор псевдокода или ПФЗ.

6.    Оформить (в электронном виде) расчетно-пояснительную записку следующего содержания:

-        Данное задание.

-        Оглавление.

-        Введение.

-        Описание заданного варианта языка.

-        Тестовая программа (программы).

-        По пунктам 2 и 3:

-        система регулярных выражений ( п.2 ) или формальная грамматика ( п. 3 )

-        фрагменты управляющих таблиц конечных автоматов, построенных Вебтранслаб’ом. Описать функционирование автоматов по этим фрагментам

-        примеры результатов работы автоматов с тестовыми примерами; объяснение и анализ этих результатов.

-        По каждому из пунктов 4 и 5:

-        полное описание разработанного алгоритма, тексты разработанных программ (в случае их большого объема включать в записку фрагменты, содержащие наиболее важные части алгоритма; описание алгоритма можно не включать, если программы приводятся полностью и имеют детальные комментарии)

-        примеры результатов работы компонент транслятора с правильными и ошибочными входными программами; объяснение и анализ этих результатов.

-        Заключение.

Оглавление

[Задание на курсовую работу 2](#_Toc11236674)

[Введение 4](#_Toc11236675)

[Описание лексики языка 5](#_Toc11236676)

[Функции 6](#_Toc11236677)

[Примитивные типы 6](#_Toc11236678)

[Константы 6](#_Toc11236679)

[Блок операторов 6](#_Toc11236680)

[Логические выражения 6](#_Toc11236681)

[Операторы сравнения 7](#_Toc11236682)

[Лексические тонкости 7](#_Toc11236683)

[Тестовые программы 7](#_Toc11236684)

[Лексический анализ 11](#_Toc11236685)

[Фрагменты управляющих таблиц конечных автоматов 12](#_Toc11236686)

[Синтаксический анализ 13](#_Toc11236687)

[Таблицы идентификаторов 18](#_Toc11236688)

[Формирование постфиксной записи 19](#_Toc11236689)

[Описание алгоритма 19](#_Toc11236690)

[Преобразование ПФЗ в последовательность тетрад 22](#_Toc11236691)

[Реализация интерпретатора 22](#_Toc11236692)

[Заключение 24](#_Toc11236693)

# Введение

В современном мире языкам програмирования уделено большое внимание. Современные алгоритмы пишутся на многих языках программирования и отлично воспринимаются компьютером. Однако с помощью языка программирования создаётся не готовая программа, а только её текст, описывающий ранее разработанный алгоритм.

Процесс преобразования операторов исходного языка программирования в машинные коды микропроцессора называется трансляцией исходного текста. В настоящее время ручная трансляция программ практически не используется. Трансляция производится специальными программами-трансляторами.

Чтобы получить работающую программу, надо этот текст либо автоматически перевести в машинный код и затем использовать отдельно от исходного текста, либо сразу выполнять команды языка, указанные в тексте программы (этим занимаются программы-интерпретаторы).

В данной работе рассматривается реализация учебного языка программирования в соответствии с вариантом задания и написание интепретатора, который будет понимать реализованный язык программирования.

**Цель работы**

Практическое применение теоретических основ проектирования трансляторов с языков программирования; освоение средств автоматизации построения трансляторов; разработка элементов транслятора для учебного языка.

# Описание лексики языка

Заданием на курсовую работу были определены следующие конструкции языка:

*1 Идентификаторы*

|  |  |
| --- | --- |
| Вариант: | **2** |
|  | <бБ><пЦ>  ( d23, N1, …) |

*2      Константы*

|  |  |
| --- | --- |
| Вариант: | **1** |
| Константы | целые по основаниям 2, 8 и 10;  вещественные;  символьные |

*3      Объявления примитивных типов (целое, вещественное, символьное):*

|  |  |
| --- | --- |
| Вариант: | **3** |
|  | [u]integer  float  symbol |

*4      Оператор присваивания:*

|  |  |
| --- | --- |
| Вариант: | **3** |
|  | <И> **:=** <В> ; |

*5      Условный оператор:*

|  |  |
| --- | --- |
| Вариант: | **1** |
|  | **? (**<ЛВ>**)** <ОБ>**[? (**<ЛВ>**)** <ОБ>**]**[**:**<ОБ>] |

*6      Оператор цикла:*

|  |  |
| --- | --- |
| Вариант: | **4** |
|  | **for([**<ОП>**];[**<ЛВ>**];[**<О>**])**<ОБ> |

*7      Оператор переключателя*

|  |  |
| --- | --- |
| Вариант: | **3** |
|  | **case**<В> **{ when**<К> **then** <ОБ>    **[ exit; ]** …**[** **other**<ОБ> **]** **}** |

Обозначения:

**[...]** – необязательная часть конструкции;

**…** – предшествующая часть конструкции повторяется произвольное количество раз;

< > – описание сокращения:

<Б>|<б>|<бБ>|<бЦ>|<Ц>|<пБ>|<пЦ>|<пБЦ> – одна большая буква | одна маленькая буква | одна любая буква | одна маленькая буква или одна цифра одна цифра | непустая последовательность букв | непустая последовательность цифр | возможно пустая посл-ть букв и/или цифр

<И> – Имя переменной / объекта; <К> – Константа;

<В> – произвольное Выражение; <ЛВ> – Логическое Выражение;

<ОБ> – Оператор или Блок; <О> – одиночный оператор; <ОП> – оператор присваивания;

Другие конструкции могут быть определены самостоятельно и будут определяться по мере выполнения лабораторных работ.

Разрабатываемый язык является императивным процедурным статически типизируемым языком. Данный язык подобен языку C.

## Функции

Основным строительным элементом программ разрабатываемого языка являются функции. Объявление функций имеет следующий вид:

*тип возвращаемого значения* имя функции (*список аргументов*)

где *тип возвращаемого значения* – любой допустимый тип языка; имя функции – любой допустимый идентификатор языка; *список аргументов* – список допустимых типов (кроме *void)* и идентификаторов, разделенных запятыми.

Возврат значения из функции производится при помощи ключевого слова *return*.

## Примитивные типы

В языке определены следующие типы данных: *void* (для функций), *integer*, *uinteger*, *float*, *symbol*

## Константы

Допустимы константы следующего вида:

* Целые числа: 123(10), 2x001, 8x567
* Вещественные числа: 1., 0.123
* Символы: ‘Z’, ‘0’

## Блок операторов

Любой блок операторов разрабатываемого языка может быть определён в любом месте корректно определенной функции, за исключением случаев, противоречащих заданию. Начало блока операторов обозначается символом «{», окончание – символом «}».

## Логические выражения

В логических выражениях используются следующие логические операторы:

&&, ||

## Операторы сравнения

В корректной программе могут использоваться следующие операторы сравнения:

=, !=, <, >, <=, >=

## Лексические тонкости

Каждый корректный оператор языка должен завершаться символом «;», за исключением случаев, противоречащих заданию.

# Тестовые программы

Для проверки корректности реализации грамматик, преобразователя в постфиксную запись, а также семантического анализатора были написаны несколько тестовых программ.

Тестовая программа 1 (Демонстрация семантического анализатора):

**includes**

**integer s1(integer v1, integer v2){**

**return v1 + v2;**

**}**

**integer m1(integer x1, integer x2){**

**return x1 - x2;**

**}**

**void main(){**

**integer z1 let 10;**

**integer z2;**

**scanf(z2);**

**integer z3 let s1(z1, z2);**

**z2 let m1(z3, z2);**

**printf(z3);**

**printf(z2);**

**integer p1 let 0;**

**p1 let p1 + ((z1 > z3) || (z1 < z3)) && ((z1 != z3) || (z1 == z3));**

**printf(p1);**

**}**

Тестовая программа 1 целиком обрабатывается интерпретатором. Запрашиваемые значения:

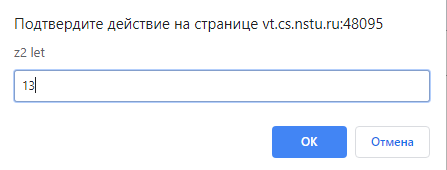
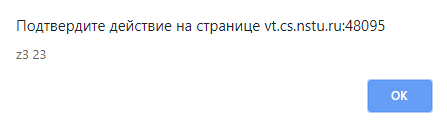
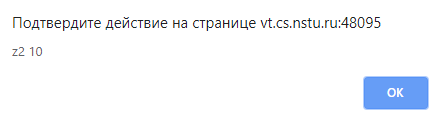


Рисунок 1 – Запрашиваемые значения для тестовой программы 1.

Результат выполнения:





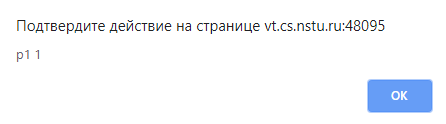


Рисунок 2 – Результат выполнения тестовой программы 1.

Тестовая программа 2 (Демонстрация ПФЗ):

**includes**

**integer c1(integer s1)**

**{**

**case(s1)**

**{**

**when 100 then**

**{**

**s1 let 0;**

**exit;**

**}**

**when 50 then**

**{**

**s1 let 1;**

**exit;**

**}**

**other**

**{**

**exit;**

**}**

**}**

**return s1;**

**}**

**float s9473(uinteger r4)**

**{**

**float p1;**

**p1 let 3.14159265359;**

**return 4\*p1\*r4\*r4;**

**}**

**integer s1(integer n2)**

**{**

**integer t1;**

**t21 let 2;**

**for(; t1 < n2; t1 let t1 + 1)**

**{**

**?(t1 < t2)**

**t1 let t1 + 1;**

**:?**

**}**

**return 1;**

**}**

**integer main(){**

**}**

Тестовая программа 2 исполняется интерпретатором не целиком, т. к. в функции интерпретации на данный момент не реализованы действия, соответствующие меткам структур *if-else*, *fo*r и *switch*

Тестовая программа 3 (Демонстрация семантического анализатора):

includes

integer s9473(integer r4) // Площадь сферы

{

float a0 let 4\*r4\*r4\*3.14159265359;

return a0;

}

void main(){

integer r1;

scanf(r1);

float s9 let s9473(r1);

printf(s9);

}

Тестовая программа 3 также целиком обрабатывается интерпретатором.

Запрашиваемые значения:

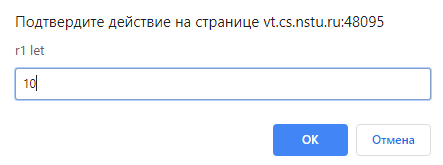


Рисунок 3 – Запрашиваемые значения для тестовой программы 2.

Результаты выполнения:

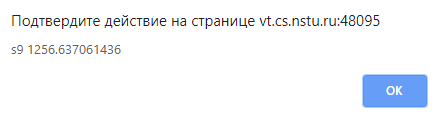


Рисунок 4 – Результат выполнения тестовой программы 2.

# Лексический анализ

В курсовой работе была реализована система регулярных выражений формального языка.

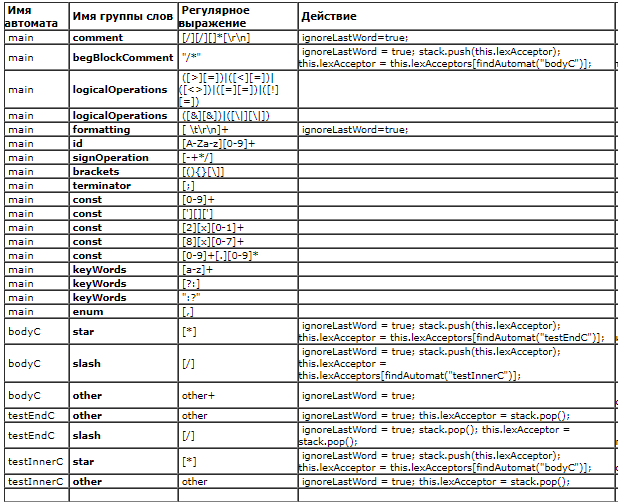


Рисунок 5 – Система лексических правил.

## Фрагменты управляющих таблиц конечных автоматов

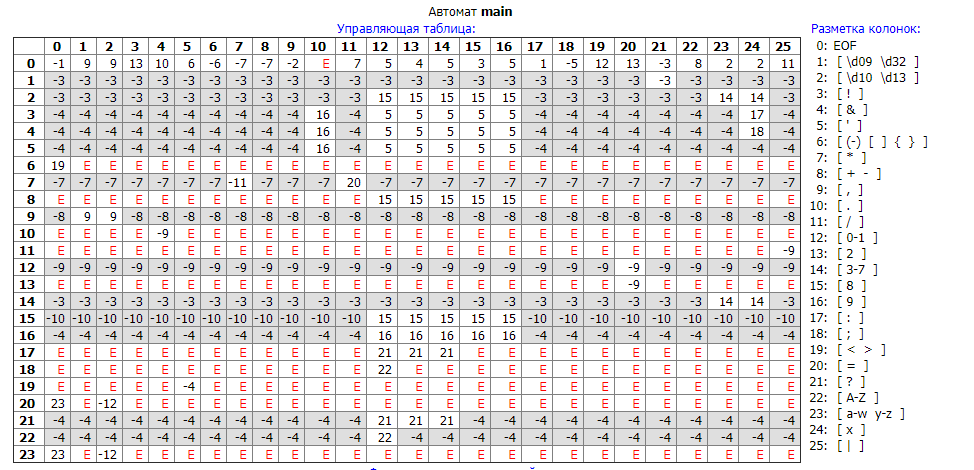


Рисунок 6 – Управляющая таблица (УТ) автомата main.

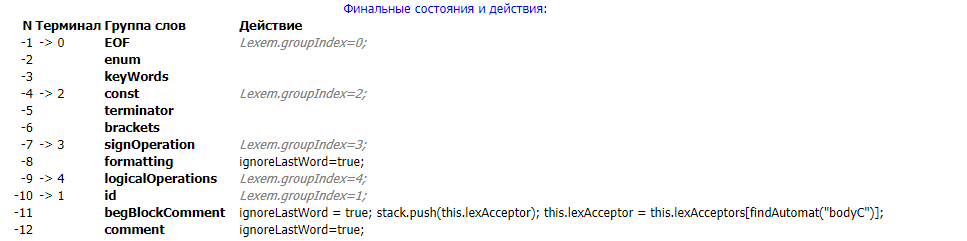


Рисунок 7 – Финальные состояния и действия автомата main.

В управляющей таблице основного автомата ***main*** (рис. 3) строки являются рабочими состояниями автомата, а столбцы – поступающими на вход символами. При нахождении автомата в одном из состояний **0**-**23** числа в соответствующей строке определяют переход автомата в следующее состояние в зависимости от поступившего на вход символа.

* Число > 0 : определяет переход в следующее рабочее состояние (номер строки УТ).
* Число < 0 на белом фоне : определяет переход в финальное состояние.
* Число < 0 на сером фоне : переход в финальное состояние с возвратом литеры на вход (т.е. последний поступивший символ будет подан на вход еще раз).

Для проверки работы на тестовой программе использовался следующий автомат: автомат, управляемый графом состояний и переходов.

Результат работы автомата с тестовой программой 3:

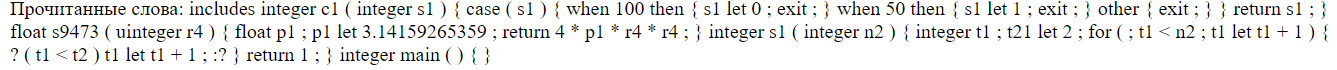


Рисунок 8 – Результаты запуска тестовой программы 3.

В «прочитанные слова» попадают все слова, которые не помещены внутрь конструкции «/\*…\*/». За игнорирование слов в конструкции «/\*…\*/» отвечат автоматы: bodyC, testInnerC, testEndC.

# Синтаксический анализ

Описанная в ходе работы грамматика относится к классу *LL*(1). *LL*(1) называется такая контекстно-свободная грамматика, у которой множества выбора правил с одинаковым нетерминалом в левой части попарно не пересекаются.

Любая такая грамматика может быть использована для нисходящего детерминированного восстановления дерева синтаксического разбора порождаемого языка. Другими словами, на основе любой *LL*(1)-грамматики может быть построен детерминированный нисходящий синтаксический акцептор, проверяющий правильность предложений языка.

Предложенная грамматика может быть легко преобразована в программу синтаксического акцептора, реализующую *рекурсивный спуск*.

Формальная грамматика учебного языка имеет следующий вид:

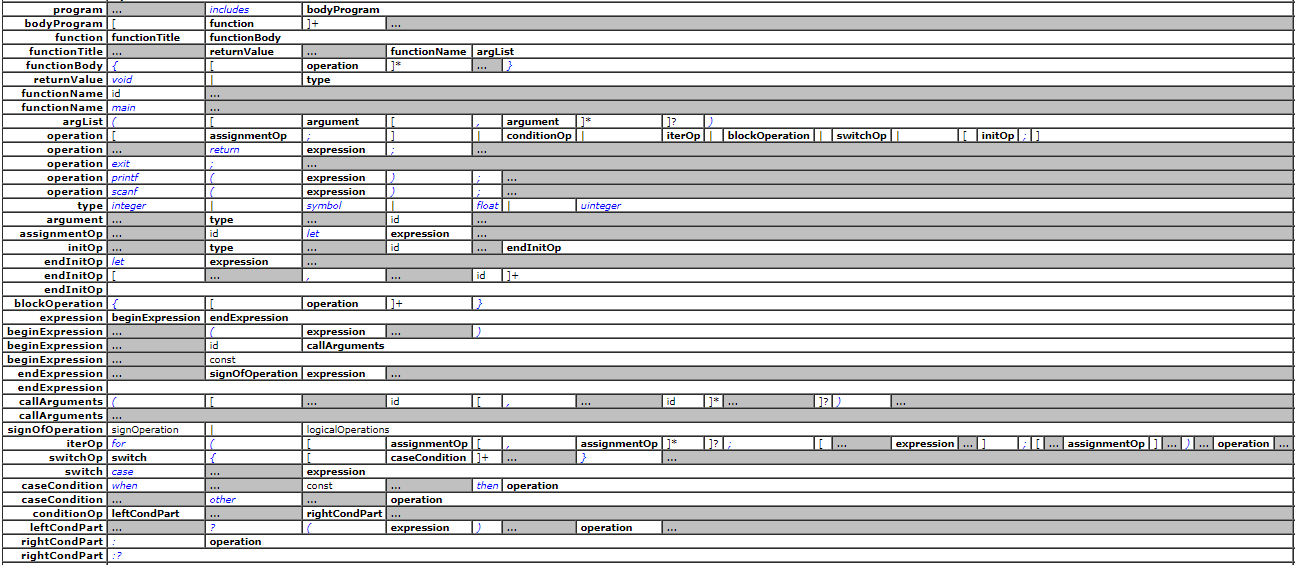


Рисунок 9 – Формальная грамматика языка (для компактности отображения действия скрыты).

Множества выбора правил имеют вид:

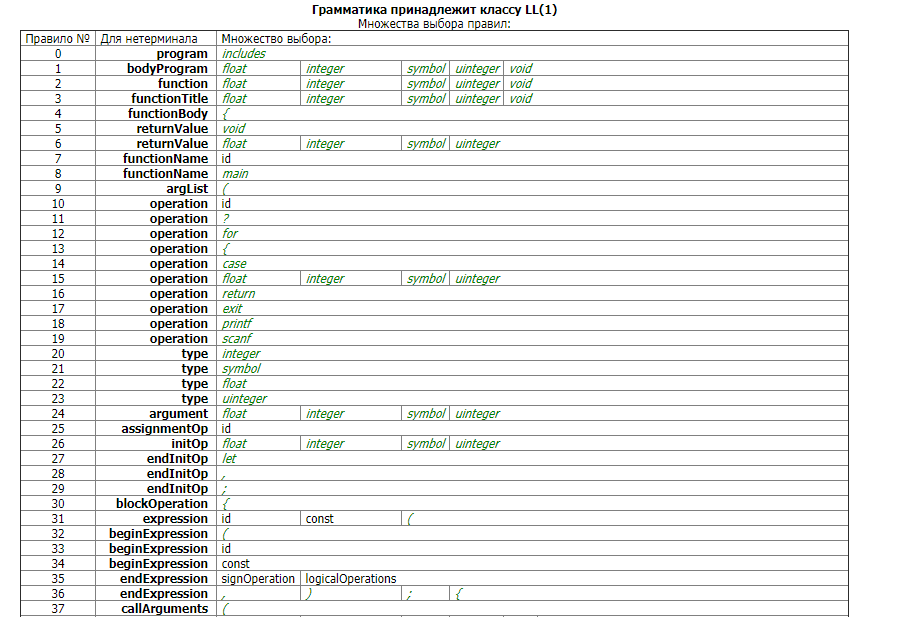


Рисунок 10 – Множества выбора правил (часть 1).

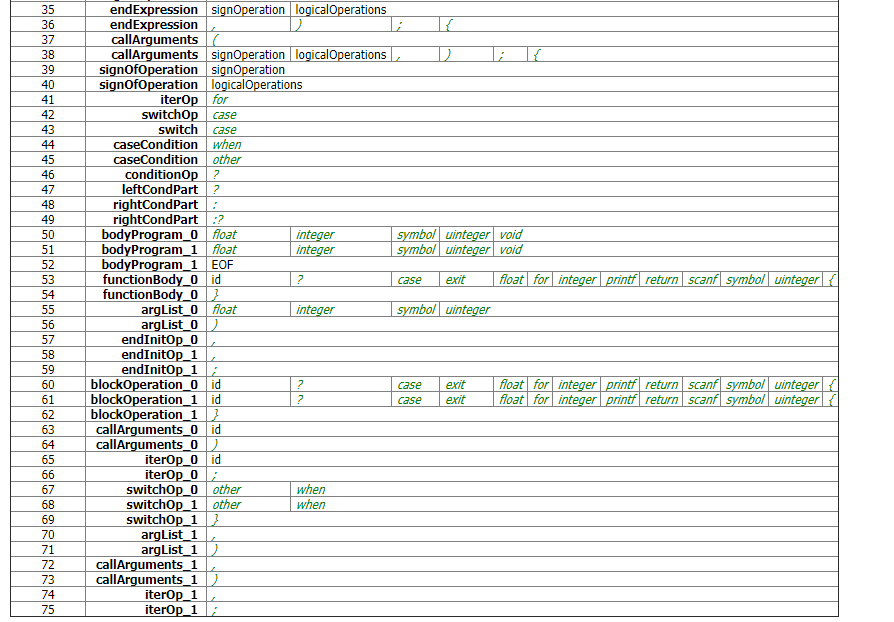


Рисунок 11 – Множества выбора правил (часть 2).

Для проверки работы на тестовой программе использовался следующий синтаксис: рекурсивный спуск, процедурная реализация.

Тестовая программа для проверки работы синтаксического анализатора:

includes

void main(){

integer r1;

}

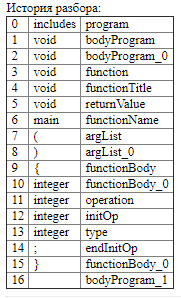


Рисунок 12 – История разбора.

Функционирование конечного автомата определяется управляющей таблицей. При запуске автомат оказывается в начальном состоянии. На каждом такте работы по входному символу и текущему состоянию автомат определяет и выполняет соответствующие операции над входным потоком.

Кратко опишем суть рекурсивного спуска в реализации данного автомата.

Синтаксический разбор начинается с первого правила грамматики (***program***).

1. По правым частям правил движение осуществляется посимвольно слева направо. Правую часть каждого правила завершает условный символ конца цепочки, что обеспечивает однотипную обработку в том числе и правил с пустой правой частью.
   * Обработка любого нетерминала состоит в переключении на первое правило для этого нетерминала.
   * Обработка терминального символа состоит в проверке его совпадения с текущим входным символом. При положительном результате происходит чтение следующего символа. При отрицательном результате происходит останов по ошибке.
2. По левым частям правил движение осуществляется сверху вниз. При этом движении используются только правила с одинаковыми нетерминалами в левой части. Для каждого правила прежде всего проверяется, содержит ли его множество выбора текущий входной символ.
   * Если множество выбора правила содержит текущий символ, происходит переключение на обработку правой части.
   * Если не содержит, то происходит переход к левой части следующего правила.
   * Если ни по одному правилу не было найдено совпадение, происходит останов по ошибке.

## Таблицы идентификаторов

Используемые в программе идентификаторы, функции и аргументы функций сохраняются в соответствующие таблицы. В дальнейшем формируемые таблицы будут необходимы для работы интерпретатора.

Программно формируются структуры, хранящие в себе идентификаторы и их значения и типы, функции и соответствующие им точки входа, Структуры реализованы с помощью одномерных стеков, контейнера table языка HTML и экземпляров класса Map языка JavaScript. Структуры также предоставляют возможность пополнения.

Далее приведем листинги соответствующих структур.

1. Словарь идентификаторов и типов (slovar).

var slovar=document.createElement("table");//создание контейнера slovar для хранения идентификаторов и их типов

//заполнение контейнера slovar идентификатором из стека table[2] и ее типом из стека table[1] (table – двумерный массив, в котором хранятся тетрады

var slovarRow=slovar.insertRow(k);

var slovarCell = slovarRow.insertCell(0);

slovarCell.width="100";

slovarCell.align="center";

slovarCell.innerHTML= top\_arr(table[2]);

var slovarCell = slovarRow.insertCell(1);

slovarCell.width="100";

slovarCell.align="center";

slovarCell.innerHTML= top\_arr(table[1]);

1. Словарь идентификаторов и их значений (values). Реализован с помощью класса Map языка JavaScript, где ключ – идентификатор переменной, а соответствующее ему значение – значение этой переменной.

var values = new Map();

function add\_value(iden, value){

values[iden]=value;

}

1. Словарь функций и точек входа в них (function\_points). Также реализован с помощью класса Map, где имя функции – ключ, точка входа (номер строки таблицы тетрад, в которой кранится определение этой функции) – значение.

var function\_points = new Map();

Рассмотрим заполнение таблицы идентификаторов и типов slovar на основе тестовой программы 1.

Была получена следующая таблица идентификаторов:

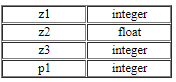


Рисунок 13 – Таблица slovar, хранящая идентификаторы и их типы.

# Формирование постфиксной записи

## Описание алгоритма

Преобразование последовательности лексем в постфиксную запись производится для всех конструкций программы таким образом, что по ПФЗ можно будет в точности восстановить исходный код.

В ПФЗ будут преобразованы следующие конструкции:

1. объявление переменных, функций;
2. передача и инициализация аргументов;
3. арифметические и логические выражения;
4. операция присваивания;
5. циклы, условные операторы, переключатели;
6. оператор возвратарезультата;
7. вызов функции.

При преобразовании в постфиксную запись весь код будет представлен в вид**е <*Оп*><*Оп*><*Функция*>**, т.е. операнды функций идут перед самой функцией. Например:

* инфиксная нотация: 1 + 2 (***<Оп><Функция><Оп>***)
* постфиксная нотация: 1 2 + (***<Оп><Оп><Функция>***)

Преобразование арифметических выражений в ПФЗ осуществляется на основе следующего алгоритма. Данный алгоритм реализует преобразование *LL*(1) выражений в ПФЗ, используя промежуточный стек операций. На вход **последовательно** подаются токены исходного выражения, после чего, опираясь на известные приоритеты операций, принимается решение, перекладывать токен в выходную очередь или во временный стек. Суть алгоритма:

* + прочитать токен;
  + если токен – операнд, сохранить его;
    - если последующие токены – отправляемые в функцию аргументы, значит сохраненный операнд – имя вызываемой функции, добавить аргументы поочередно в выходной поток, после этого отправить в выходной поток сохраненный операнд;
    - если после операнда не следует отправляемых аргументов, операнд отправляется в выходной поток;
  + если токен – оператор op1:
    - пока на вершине временного стека присутствует другой оператор op2 и его приоритет выше либо равен приоритету op1:
    - перекладывать оператор op2 из временного стека в выходную очередь;
    - добавить op1 во временный стек;
  + если токен – открывающая скобка, добавить токен во временный стек;
  + если токен – закрывающая скобка:
    - пока на вершине временного стека не открывающая скобка:
      * перекладывать операторы из стека на выход;
    - выкинуть открывающую скобку из стека;
  + если токен
  + если больше не осталось токенов на входе:
    - пока остались операторы в стеке:
      * перекладывать операторы на выход;
  + конец.

Функция, возвращающая приоритет той или иной операции:

function priority(s) {

if(s == "(") return 0;

if(s == "return") return 1;

if(s == "+") return 5;

if(s == "-") return 5;

if(s == "||") return 5;

if(s == "&&") return 7;

if(s == "!=") return 10;

if(s == ">") return 10;

if(s == "<") return 10;

if(s == "==") return 10;

if(s == "\*") return 10;

if(s == "/") return 10;

return 0;

}

Как видно, наименьший приоритет присвоен скобкам, для того чтобы одну скобку могла вытолкнуть только другая скобка.

Алгоритм реализуется посредством расширения формальной грамматики действиями.

Для демонстрации работы преобразователя возьмем тестовую программу 2.

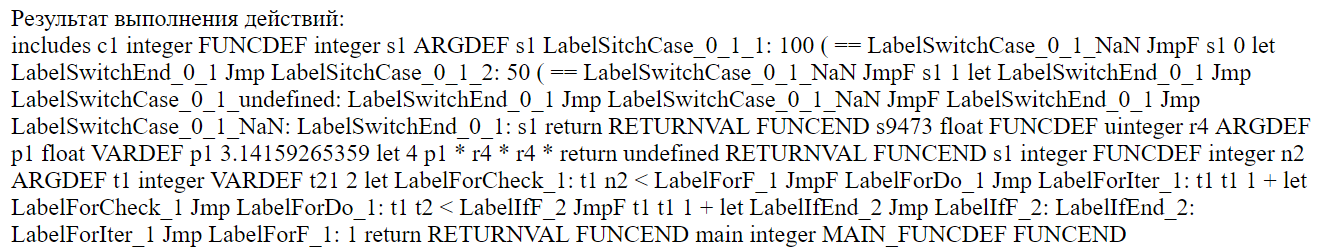


Рисунок 14 – Результат преобразования тестовой программы 2 в ПФЗ.

# Преобразование ПФЗ в последовательность тетрад

Для демонстрации работы преобразователя возьмем тестовую программу 1.

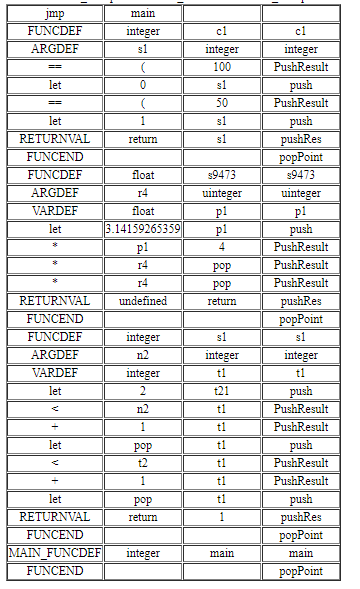


Рисунок 15 – Результат преобразования тестовой программы 1 в последовательность тетрад.

В первом столбце таблицы тетрад содержится метка, определяющая операцию, реализуемую строкой таблицы. Во втором и третьем столбце – операнды, а в третьем – результат.

# Реализация интерпретатора

Интерпретатор был реализован в виде функции interpret(). Функция обрабатывает таблицу тетрад. Далее приведен фрагмент функции, выполняющий определенные операции в зависимости от метки, содержащейся в первом столбце таблицы тетрад.

switch(table[0][i]) { //элемент 1-го столбца i-ой строки таблицы тетрад

case "PRINTF": //если метка – PRINTF, выводим диалоговое окно с соответствующим значением переменной и ее идентификатором

alert(table[1][i]+" "+values[table[1][i]]);

break;

case "SCANF": //если метка – SCANF, выводится диалоговое окно с полем ввода

values[table[1][i]]=prompt(table[1][i]+" := ");

break;

case ":=": //если метка – знак присваивания, определяем, нужно ли присвоить результат вычисления выражения (в таком случае второй операнд равен рор), либо значение переменной, в этом случае присваеваем значение переменной, извлекая его из словаря values, либо

if (table[1][i] == "pop")

values[table[2][i]] = results.pop();

var lex\_type = get\_type\_of\_lexem(table[1][i]);

if (lex\_type == "int" || lex\_type == "float")

values[table[2][i]] = table[1][i];

if (lex\_type == "id")

values[table[2][i]] = values[table[1][i]];

break;

case "jmp": //переходим на точку входа функции, указанной как первый операнд

var func\_name=table[1][i];

i = function\_points[func\_name] - 1;

break;

case "FUNCDEF"://обнуляем список аргументов функции для последующей их инициализации

arg\_num = 0;

break;

case "ARGDEF": //инициализируем аргументы функции

values[table[1][i]]=values[sent\_args[arg\_num++]];

break;

case "RETURNVAL": //кладем в стек результатов значение, возвращаемое функцией для последующего его извлечения в точке вызова функции

if (table[2][i] == "pop")

break;

results.push(values[table[2][i]]);

break;

case "FUNCEND": //возвращаемся в точку вызова функции

i = points\_to\_jmp.pop();

break;

case "FUNCCALL": //переходим в вызванную функцию со стеком аргументов sent\_args, запоминаем точку вызова функции

var func\_name=table[1][i];

var point = function\_points[func\_name] - 1;

sent\_args = table[2][i];

points\_to\_jmp.push(i);

i = point;

break;

//выполнение арифметических и логических операций

case "+": case "-": case "\*": case "/":

case "%": case "<": case ">": case "<=":

case ">=": case "!=": case "==": case "&&": case "||":

var oper\_type = table[0][i];

var op1 = table[1][i]; //1-ый столбец, i-я ячейка

var op2 = table[2][i];

if (op1 == "pop")

op1=results.pop();

if (get\_type\_of\_lexem(op1) == "id")

op1=values[op1];

var type = get\_type\_of\_lexem(op1);

if (type == "int")

op1=parseInt(op1);

if (type == "float")

op1=parseFloat(op1);

if (op2 == "pop")

op2=results.pop();

if (get\_type\_of\_lexem(op2) == "id")

op2=values[op2];

type = get\_type\_of\_lexem(op2);

if (type == "int")

op2=parseInt(op2);

if (type == "float")

op2=parseFloat(op2);

res = operate[table[0][i]](op2, op1);

results.push(res);

}

# Заключение

В результате выполнения данной курсовой работы был разработан язык программирования с заданными особенностями синтаксиса. Язык предоставляет возможность писать функции, выполнять арифметические вычисления, сохранять их результат в память и извлекать результат из памяти.

В ходе работы были изучены основные понятия и методы лексического и синтаксического анализа, построены автоматные реализации соответствующих анализаторов и изучены особенности их работы. Также были реализованы алгоритмы, преобразующие исходный код программы сначала в постфиксную запись, а затем в псевдокод в формате последовательности триад.

На всех этапах разработки языка управляющие конструкции создавались с расчетом на надёжность легкую масштабируемость, что позволяет легко расширять и изменять функциональность языка. Полученные знания по основам теории формальных языков, а также опыт разработки на языке *JavaScript* могут оказаться полезными в дальнейшей работе.