Министерство науки и высшего образования РФ

ФГАОУ ВО Пермский национальный исследовательский

политехнический университет

Кафедра «Вычислительная математика, механика и биомеханика»

КУРСОВАЯ РАБОТА

Тема: «Язык управления роботом»

по дисциплине «Языки программирования и методы трансляции»

Выполнил: студентка группы ИСТ-22-1б Афонина А.М.

Проверил: Доцент С.Е. Батин.

Пермь, 2023

Содержание

[**Введение** 3](#_Toc155499669)

[**Теоретическая часть** 4](#_Toc155499670)

[Лексический анализатор 4](#_Toc155499671)

[Генератор лексических анализаторов Flex 4](#_Toc155499672)

[Структура программ Flex 5](#_Toc155499673)

[Синтаксический анализатор 5](#_Toc155499674)

[Генератор синтаксических анализаторов Bison 5](#_Toc155499675)

[Структура программ Bison 6](#_Toc155499676)

[**Практическая** **часть** 7](#_Toc155499677)

[Описание языка управления 7](#_Toc155499678)

[Тестирование работы кода 8](#_Toc155499679)

[Анализ ограничений применимости языка 9](#_Toc155499680)

[Заключение 10](#_Toc155499681)

[**Приложения** 10](#_Toc155499682)

[**Список** **литературы** 15](#_Toc155499683)

# **Введение**

Курсовая работа направлена в создании языка программирования, который способен решать простые практические задачи. В работе должны быть представлены грамматика этого языка и инструменты, использованные при его разработке. Также необходимо включить лексический и синтаксический анализаторы, которые отвечают за разбор и анализ исходного кода языка. Важной частью работы является анализ ограничений, связанных с использованием созданного языка программирования. Это позволит определить, где и как язык может быть применен с наибольшей эффективностью.

Цель курсовой работы - создать язык управления роботом, с помощью которого робот может двигаться в произвольном направлении, выполнять заданные действия. Транслятор должен записать последовательность действий робота при передаче ему программы.

Задачи:

* Создание лексического анализатора
* Создание синтаксического анализатора
* Тестирование
* Анализ ограничений применимости языка

# **Теоретическая часть**

**Компилятор** — это программа, которая переводит исходный код на языке программирования в машинный код. Если этого не сделать, компьютер не поймёт, как выполнить инструкции разработчика. Поэтому мы отдаём компилятору строки кода, а он сравнивает их со своим словарём, учитывает контекст и выдаёт набор из нулей и единиц.

В компилировании программы одни из важных этапов являются создание лексического (Lexer) и синтаксического (Parser) анализаторов. В данной работе подробно расписаны данные этапы.

## Лексический анализатор

**Лексический анализатор** - первая фаза компиляции, которая считывает поток символов исходной программы и группирует их в лексемы. Каждая лексема затем преобразуется в токен, который передается следующей фазе компиляции, синтаксическому анализу.

Справочник по словам:

**Токен** - это объект, который создается из лексемы в процессе лексического анализа. Он представляет собой конкретный тип данных или символ, имеющий смысл для компилятора или интерпретатора.

**Лексема** - это последовательность символов языка программирования, которая имеет определенное значение для трансляции. Лексема может быть идентификатором, ключевым словом, числом, оператором или другой частью исходного кода, которая имеет смысл для анализа программы.

## Генератор лексических анализаторов Flex

**Flex** используется по следующей схеме:

* Создается файл с расширением .l, написанный на языке lex, в котором описывается генерируемый лексический анализатор.
* При компиляции файла .l генерируется файл lex.yy.c
* Файл lex.yy.c содержит реализацию лексического анализатора, который был описан с помощью правил сопоставления шаблонов в файле .l.
* Файл lex.yy.c может быть интегрирован с другими компонентами программы. Например, сгенерированный лексический анализатор может быть частью синтаксического анализатора (парсера), который обрабатывает токены, полученные от лексического анализатора.

## Структура программ Flex

Создаваемая программа должна иметь следующий вид :

***Определения***

***%%***

***Правила***

***%%***

***Пользовательский код***

Раздел определений включается в себя объявление переменных, именованые константы и регулярные определения.

Раздел правила имеет следующий вид:

***Шаблон{Действие}***

Шаблон является регулярным выражением, которое может использовать регулярные определения из раздела определений. Действия представляют собой фрагменты кода на языке программирования С.

Раздел пользовательского кода содержит в себе допольнительные функции, используемые в действиях.

## Синтаксический анализатор

**Синтаксический анализ**, вторая фаза компиляции, использует токены, полученные в результате лексического анализа, для создания синтаксического дерева. Синтаксическое дерево описывает грамматическую структуру входного потока токенов и помогает понять его семантику. При этом синтаксический анализатор может также обнаруживать и сообщать об ошибках в структуре или последовательности токенов, если они не соответствуют грамматике языка программирования или других форматов данных.

## Генератор синтаксических анализаторов Bison

**Bison** используется по следующей схеме:

* Создается файл с расширением .y
* При компиляции файла "parser.y", Bison анализирует грамматику, определенную внутри файла. Грамматика обычно содержит правила, которые описывают различные синтаксические конструкции языка (например, условные выражения, циклы и т.д.) и специальные действия, связанные с этими конструкциями.
* При успешной компиляции, Bison создает два файла: "parser.tab.c" и "parser.tab.h". "parser.tab.c" содержит сгенерированный код на языке программирования, который реализует синтаксический анализатор, основанный на заданной грамматике, "parser.tab.h" содержит определения токенов, используемых синтаксическим анализатором.
* Файл "parser.tab.c" может быть затем скомпилирован с помощью выбранного компилятора вместе с другими файлами проекта или библиотеками, чтобы создать окончательный исполняемый файл или модуль.
* Файл "parser.tab.h" может быть включен в другие файлы проекта, чтобы использовать определения токенов в коде.

## Структура программ Bison

Основная часть описания грамматики в Bison - описание последовательности ожидаемых лексем.

Создаваемая программа должна иметь следующий вид:

***Объявления***

***%%***

***Правила трансляции***

***%%***

***С-программы поддержки***

В программе на Bison есть два раздела объявлений.

В первом разделе можно разместить обычные объявления на языке С, заключенные между символами % { и % }. В этом разделе обычно размещаются объявления временных переменных, которые будут использоваться в правилах трансляции или в других частях программы. Также в этой части можно объявить токены грамматики.

Во второй части программы, после первой пары символов %%, размещаются правила трансляции. Каждое правило состоит из продукции грамматики и связанных семантических действий. Семантические действия в Bison представляют собой последовательности инструкций на языке С, которые будут выполнены при срабатывании соответствующего правила.

Третья часть спецификации Bison содержит С-подпрограммы поддержки. Один из обязательных элементов в этой части - это функция лексического анализатора, которая обычно называется yylex(). Лексический анализатор yylex() генерирует токены, каждый из которых состоит из имени токена и связанного значения атрибута.

Таким образом, в Bison-программе можно разместить объявления, правила трансляции и С-подпрограммы поддержки, такие как лексический анализатор yylex(). Каждая часть программы выполняет свою роль в процессе синтаксического анализа и трансляции.

# **Практическая** **часть**

Для реализации цели курсовой работы необходимо решить ряд задач. К ним относятся создание лексического и синтаксического анализаторов, тестирование работы создаваемого языка управления, а также анализ ограничений его работы.

## Описание языка управления

С помощью генераторов лексического и синтаксического анализов lex и bison был создан язык программирования для управления роботом. Размер робота составляет одну клетку. Робот может передвигаться направо, налево, вверх и вниз ( up, down, right, left) по координатам x и y. Поле, на котором находится робот, ограничено, его размер 100х100. Если робот выйдет за пределы, программа объявляет пользователю об этом и прекращает свою работу.

В качестве мини-заданий пользователю предлагается: поставить 2 кубика по пути: t1 (поставить первый кубик) , t2 (поставить второй кубик); а также сделать «прыжок» ( jumpd – «прыжок» вниз, jumpr-«прыжок» вправо, jumpup - «прыжок» вверх,jimpl - «прыжок» влево). «прыжок» означает, что количество шагов, который задает пользователь, будет сделано роботом в 2 раза больше. Например, пользователь задает команду «прыжок вправо 5», робот ,в свою очередь сделает ,10 шагов вправо.

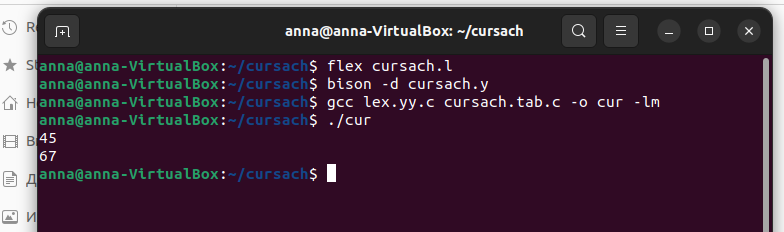
У нас доступны 2 "cursach\_int.txt","cursach\_out.txt". В первом файле пользователь записывает задачи для робота( сделать n количество шагов вправо, влево и тд; поставить первый кубик, второй кубик, сделать прыжок).Во втором файле описывается каждое движение робота (Робот движется вправо на n количество шагов, Робот поставил первый кубик, Робот сделал супер прыжок на n количество шагов и пр.), каждые его действия подсчитываются.

В самом начале программа просит нас написать изначальные координаты робота, при каждом дейтсвии робота также обновляются его координаты и печатаются в файле "cursach\_out.txt". После команды exit робот прекращает свою работу. В файле указываются конечные координаты робота, а также сколько действий он выполнил.

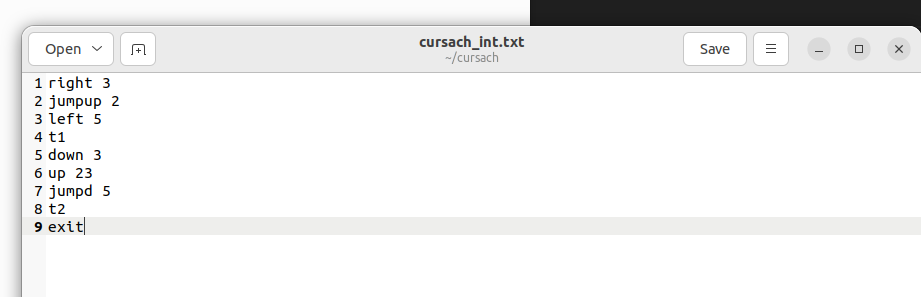
## Тестирование работы кода

Тестирование показало, что робот выполняет все указанные нами задания.

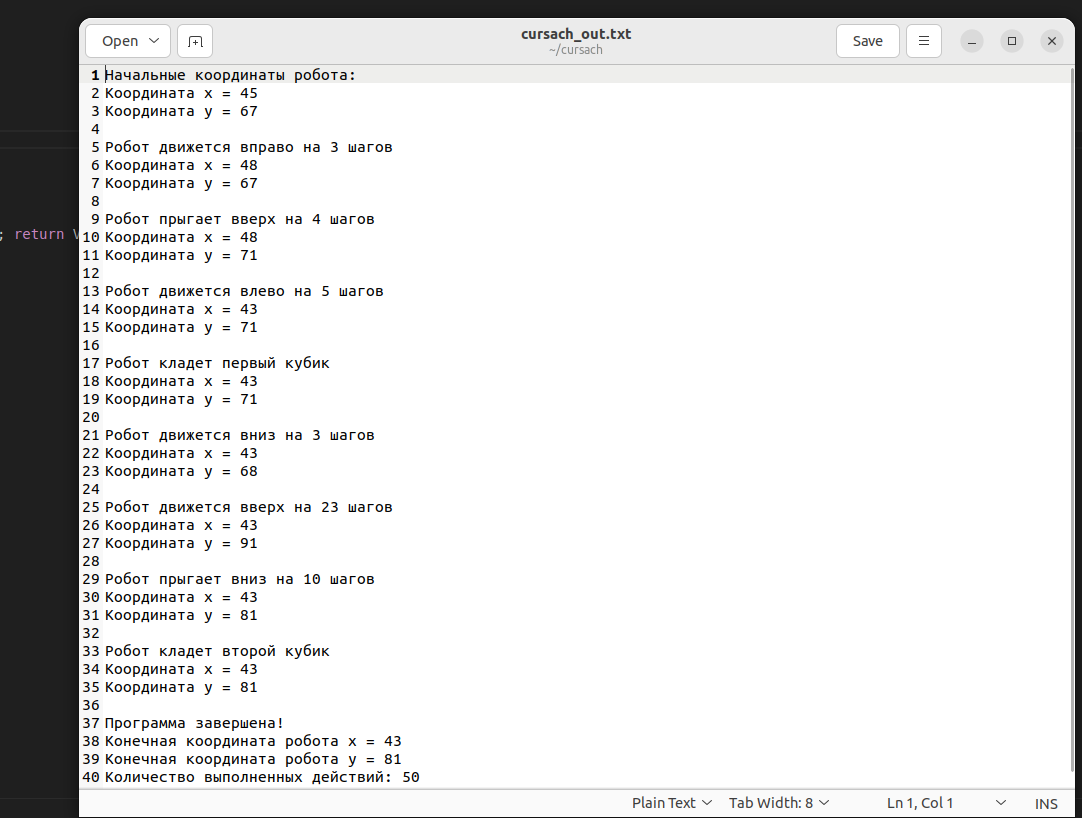
Компиляция и запуск программы:



Входной файл: cursach\_int.txt



Выходной файл: cursach\_out.txt



## Анализ ограничений применимости языка

Данный язык программирования имеет ограничения, которые могут сделать его неудобным для реализации сложных программ работы робота. Например, отсутствие возможности создавать циклы требует прописывать каждый шаг вручную, что может быть неэффективно и неудобно при работе с более сложными задачами. Кроме того, язык не поддерживает числовые константы, такие как число π и экспонента e, и не может оперировать числами с плавающей точкой. Размеры поля также могут быть изменены, но это требует изменений в синтаксическом анализаторе языка.

## Заключение

Был разработан язык программирования для управления роботом, включая создание лексического и синтаксического анализаторов. Проведены тесты программ, результаты которых были представлены выше. Однако, при анализе ограничений было выявлено, что не удалось полностью реализовать задачи, включая возможность работы с условиями и циклами в созданном языке программирования. Тем не менее, в дальнейшем возможно развитие языка путем добавления функциональности для работы с переменными, условиями и циклами.

# **Приложения**

Приложение 1:

Код лексического анализатора:

Файл: cursach.l

1. %{
3. #include <stdio.h>
4. #include <stdlib.h>
5. #include"cursach.tab.h"
6. #include <math.h>
7. **extern** **int** yylval;
9. %}
11. %%
12. [0-9]+  {  yylval=atoi(yytext); **return** VAL; }
13. jumpd **return** JUMP\_DOWN;
14. jumpr **return** JUMP\_RIGHT;
15. jumpup **return** JUMP\_UP;
16. jumpl **return** JUMP\_LEFT;
17. right **return** RIGHT;
18. left **return** LEFT;
19. up **return** UP;
20. down **return** DOWN;
21. t1 **return** T1;
22. t2 **return** T2;
24. "exit"   { **return** EXIT; }
25. \n
26. [ \t]+ ;
28. %%
29. **int** yywrap() {
30. **return** 1;
31. }

Приложение 2:

Код синтаксического анализаора:

Файл: cursach.y

1. %{
2. #include <stdio.h>
3. #include <stdlib.h>
4. #include"cursach.tab.h"
5. **extern** FILE \*yyin, \*yyout;
7. **void** yyerror();
8. **int** yylex(**void**);
9. **int** x;  // начальная координата по оси x
10. **int** y;  // начальная координата по оси y
11. **int** counter = 0;  // счетчик выполненных действий
12. %}

15. %token VAL UP DOWN RIGHT LEFT EXIT T1 T2 JUMP\_DOWN JUMP\_LEFT JUMP\_RIGHT JUMP\_UP
16. %left '+' '-'
17. %left '\*' '/'
18. %left '(' ')'
20. %%
22. commands: //описывает последовательность команд
23. | commands command
24. ;
26. command: movement  // движение
27. | exit  // выход
28. | action  // действие
29. | jump  // прыжок
30. ;
31. // command  представляет собой 4 типа команд : movement, exit,action,jump
33. movement: UP VAL {  // движение вверх на заданное количество шагов
34. y = y + $2;
35. **if** (y > 100 || y < 0) {
36. fprintf(yyout, "Невозможное действие: превышение границы по оси y\n");
37. exit(0);
38. } **else** {
39. counter = counter + $2;
40. fprintf(yyout, "Робот движется вверх на %d шагов\n", $2);
41. fprintf(yyout, "Координата x = %d\n", x);
42. fprintf(yyout, "Координата y = %d\n\n", y);
43. }
44. }
45. | DOWN VAL {  // движение вниз на заданное количество шагов
46. y = y - $2;
47. **if** (y < 0 || y > 100) {
48. fprintf(yyout, "Невозможное действие: превышение границы по оси y\n");
49. exit(0);
50. } **else** {
51. counter = counter + $2;
52. fprintf(yyout, "Робот движется вниз на %d шагов\n", $2);
53. fprintf(yyout, "Координата x = %d\n", x);
54. fprintf(yyout, "Координата y = %d\n\n", y);
55. }
56. }
57. | RIGHT VAL {  // движение вправо на заданное количество шагов
58. x = x + $2;
59. **if** (x > 100 || x < 0) {
60. fprintf(yyout, "Невозможное действие: превышение границы по оси x\n");
61. exit(0);
62. } **else** {
63. counter = counter + $2;
64. fprintf(yyout, "Робот движется вправо на %d шагов\n", $2);
65. fprintf(yyout, "Координата x = %d\n", x);
66. fprintf(yyout, "Координата y = %d\n\n", y);
67. }
68. }
69. | LEFT VAL {  // движение влево на заданное количество шагов
70. x = x - $2;
71. **if** (x < 0 || x > 100) {
72. fprintf(yyout, "Невозможное действие: превышение границы по оси x\n");
73. exit(0);
74. } **else** {
75. counter = counter + $2;
76. fprintf(yyout, "Робот движется влево на %d шагов\n", $2);
77. fprintf(yyout, "Координата x = %d\n", x);
78. fprintf(yyout, "Координата y = %d\n\n", y);
79. }
80. };
82. exit: EXIT {  // выход из программы
83. fprintf(yyout, "Программа завершена!\n");
84. };
86. action: T1 {  // действие 1
87. fprintf(yyout, "Робот кладет первый кубик\n");
88. fprintf(yyout, "Координата x = %d\n", x);
89. fprintf(yyout, "Координата y = %d\n\n", y);
90. counter++;
91. }
92. | T2 {  // действие 2
93. fprintf(yyout, "Робот кладет второй кубик\n");
94. fprintf(yyout, "Координата x = %d\n", x);
95. fprintf(yyout, "Координата y = %d\n\n", y);
96. counter++;
97. };
99. jump: JUMP\_UP VAL {  // прыжок вверх на заданное количество шагов
100. y = y + (2 \* $2);
101. **if** (y > 100 || y < 0) {
102. fprintf(yyout, "Невозможное действие: превышение границы по оси y\n");
103. exit(0);
104. } **else** {
105. counter = counter + (2 \* $2);
106. fprintf(yyout, "Робот прыгает вверх на %d шагов\n", (2 \* $2));
107. fprintf(yyout, "Координата x = %d\n", x);
108. fprintf(yyout, "Координата y = %d\n\n", y);
109. }
110. }
111. | JUMP\_DOWN VAL {  // прыжок вниз на заданное количество шагов
112. y = y - (2 \* $2);
113. **if** (y < 0 || y > 100) {
114. fprintf(yyout, "Невозможное действие: превышение границы по оси y\n");
115. exit(0);
116. } **else** {
117. counter = counter + (2 \* $2);
118. fprintf(yyout, "Робот прыгает вниз на %d шагов\n", (2 \* $2));
119. fprintf(yyout, "Координата x = %d\n", x);
120. fprintf(yyout, "Координата y = %d\n\n", y);
121. }
122. }
123. | JUMP\_RIGHT VAL {  // прыжок вправо на заданное количество шагов
124. x = x + (2 \* $2);
125. **if** (x > 100 || x < 0) {
126. fprintf(yyout, "Невозможное действие: превышение границы по оси x\n");
127. exit(0);
128. } **else** {
129. counter = counter + (2 \* $2);
130. fprintf(yyout, "Робот прыгает вправо на %d шагов\n", (2 \* $2));
131. fprintf(yyout, "Координата x = %d\n", x);
132. fprintf(yyout, "Координата y = %d\n\n", y);
133. }
134. }
135. | JUMP\_LEFT VAL {  // прыжок влево на заданное количество шагов
136. x = x - (2 \* $2);
137. **if** (x < 0 || x > 100) {
138. fprintf(yyout, "Невозможное действие: превышение границы по оси x\n");
139. exit(0);
140. } **else** {
141. counter = counter + (2 \* $2);
142. fprintf(yyout, "Робот прыгает влево на %d шагов\n", (2 \* $2));
143. fprintf(yyout, "Координата x = %d\n", x);
144. fprintf(yyout, "Координата y = %d\n\n", y);
145. }
146. };
148. %%
150. **void** main() {
151. yyin = fopen("cursach\_int.txt", "r");
152. yyout = fopen("cursach\_out.txt", "w");
153. scanf("%d", &x);  // ввод начальной координаты x
154. scanf("%d", &y);  // ввод начальной координаты y
156. fprintf(yyout, "Начальные координаты робота:\nКоордината x = %d\nКоордината y = %d\n", x, y);
158. fprintf(yyout, "\n");
159. yyparse();
160. fprintf(yyout, "Конечная координата робота x = %d\n", x);
161. fprintf(yyout, "Конечная координата робота y = %d\n", y);
163. fprintf(yyout, "Количество выполненных действий: %d\n", counter);
164. fprintf(yyout, "\n");
165. }
167. **void** yyerror() {
168. printf("\nВыражение недопустимо\n");
169. }

# **Список** **литературы**

1. **flex & bison /.** John Levin / August 2009
2. Разработка компиляторов и интерпретаторов

<https://ipc.susu.ru/38277-3.html>

1. Lexical Analysis with Flex

<https://epaperpress.com/lexandyacc/download/flex.pdf>