Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное образовательное
учреждение высшего образования
Пермский национальный исследовательский политехнический университет
(ПНИПУ)

Факультет: Электротехнический (ЭТФ)
Направление: 09.03.04 – Программная инженерия (ПИ)
Профиль: Разработка программно-информационной систем (РИС)
Кафедра информационных технологий и автоматизированных систем (ИТАС)

УТВЕРЖДАЮ
Зав. кафедрой ИТАС: д-р экон. наук, проф.
_____ Р.А. Файзрахманов
«_____»_____ 2025 г.

КУРСОВАЯ РАБОТА

по дисциплине
«Системное программирование»
на тему
«Разработка компилятора для языка программирования MedievalGibberish»

Студент: _____ Серебряков Егор Константинович
(подпись, дата)
Группа: РИС-23-2б

Оценка _____

Руководитель КР: Стар. Преп. Каф. ИТАС Кузнецов Д. Б.
_____
(подпись, дата)

Пермь 2025

Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное образовательное
учреждение высшего образования
Пермский национальный исследовательский политехнический университет
(ПНИПУ)

Факультет: Электротехнический (ЭТФ)
Направление: 09.03.04 – Программная инженерия (ПИ)
Профиль: Разработка программно-информационной систем (РИС)
Кафедра информационных технологий и автоматизированных систем (ИТАС)

УТВЕРЖДАЮ
Зав. кафедрой ИТАС: д-р экон. наук, проф.
_____ Р.А. Файзрахманов
«_____»_____ 2025 г.

ЗАДАНИЕ
на выполнение курсовой работы

Фамилия, имя, отчество: Серебряков Егор Константинович
Факультет Электротехнический Группа РИС-23-2б
Начало выполнения работы: 12.11.2025
Контрольные сроки просмотра работы: 23.11, 30.11, 12.12
Защита работы: 16.12.2025
1. Наименование темы: «Разработка компилятора для собственного языка программирования».
2. Исходные данные к работе (проекта):
Объект исследования – Языки программирования.
Предмет исследования – Разработка языка программирования.
Цель работы (проекта) – Разработка собственного язык программирования MedievalGibberish и создание для него компилятора
3. Содержание:
3.1 Исследование предметной области
3.1.1 Основы анализа кода программного языка
3.1.2 Лексический анализатор
3.1.2 Синтаксический анализатор
3.1.3 Дерево разбора
3.2 Разработка компилятора для собственного языка программирования
3.2.1 Выбор целевой платформы
3.2.2 Разработка собственного языка
3.2.3 Разработка лексического анализатора
3.2.4 Разработка синтаксического анализатора
3.2.5 Компиляция компилятора и создание MakeFile
3.3 Примеры использования

Руководитель КР: _____ Стар. Преп. Каф. ИТАС Кузнецов Д. Б.
(подпись, дата)
Задание получил: _____ Е.К. Серебряков
(подпись, дата)

КАЛЕНДАРНЫЙ ГРАФИК ВЫПОЛНЕНИЯ
КУРСОВОЙ РАБОТЫ

№ пп	Этапы работы	Объём этапа, %	Сроки выполнения	
			Начало	Конец
1.	Исследование предметной области	16.6	12.11.25	18.11.25
2.	Анализ исходных данных	9.5	18.11.25	21.11.25
3.	Разработка языка программирования	11.9	21.11.25	25.11.25
4.	Создание лексического анализатора в программе FLEX	9.5	25.11.25	28.11.25
5.	Создание синтаксического анализатора в программе GNU Bison	11.9	28.11.25	02.12.25
6.	Создание MakeFile	9.5	02.12.25	05.12.25
7.	Устранение ошибок и отладка	26.2	05.12.25	15.12.25
8.	Написание отчета	2.3	16.12.25	16.12.25
9.	Защита курсовой работы	2.3	16.12.25	16.12.25

Руководитель КР: _____ Стар. Преп. Каф. ИТАС Кузнецов Д. Б.
(подпись, дата)

Задание получил: _____ Серебряков Егор Константинович
(подпись, дата)

РЕФЕРАТ

Отчет 49 с., 42 рис., 4 источн., 5 прил.

ЛЕКСИЧЕСКИЙ АНАЛИЗАТОР, СИНТАКСИЧЕСКИЙ АНАЛИЗАТОР, ТРАНСЛИРОВАНИЕ ЯЗЫКА ПРОГРАММИРОВАНИЯ, КОМПИЛЯТОР

Цель работы – разработка компилятора для собственного языка программирования.

Объект исследования – языки программирования.

Предмет исследования – разработка языка программирования.

Разработка велась в несколько этапов: анализ предметной области, выбор языка C в качестве целевой платформы для транслирования языка с разработанного на известный, создание лексического анализатора на FLEX, создание синтаксического анализатора на BISON, автоматизация компилирования компилятора, состоящего из лексического и синтаксического анализаторов при помощи MakeFile.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Современная индустрия разработки программного обеспечения характеризуется широким распространением высокоуровневых языков и готовых фреймворков, абстрагирующих разработчика от низкоуровневых процессов, повышает продуктивность, но имеет и обратную сторону. Это ведёт к постепенному забыванию фундаментальных принципов, на которых построена вся вычислительная техника: работы с формальными грамматиками, лексического и синтаксического анализа, управления памятью и трансляции кода. Таким образом, актуальность данной работы заключается в практическом изучении одного из низкоуровневых процессов, освоении инструментов для автоматизации формирования компиляторов.

В качестве объекта исследования выступает процесс проектирования и реализации компилятора.

Предметом исследования является разработка компилятора для собственного языка программирования MedievalGibberish.

Целью работы является закрепление теоретических знаний на практике путем проектирования и реализации работоспособного компилятора для языка MedievalGibberish, транслирующего исходный код в код на языке программирования C.

Для достижения целы выполняются следующие задачи:

1. Анализ предметной области;
2. Разработка собственного языка программирования;
3. Разработка лексического анализатора;
4. Разработка синтаксического анализатора;
5. Компиляция компилятора и автоматизация компиляции при помощи MakeFile.

1 Исследование предметной области

1.1 Основы анализа кода программного языка

Любые инструкции, написанные на программном языке, проходят через три основных этапа обработки: лексический анализ, синтаксический анализ и семантический анализ.

Лексический анализ является первым этапом обработки исходного кода в процессе компиляции. Его основная задача — преобразование входного потока символов, представляющего собой текст программы, в упорядоченную последовательность значимых элементов, называемых лексемами или токенами.

Исходный код рассматривается на данном этапе как простая текстовая строка. Лексический анализатор последовательно читает эту строку, группируя символы в минимальные, неделимые с точки зрения языка единицы — лексемы.

Синтаксический анализатор исследует комбинации лексем. Он определяет, образует ли их последовательность правильно сформированные конструкции, соответствующие формальным правилам грамматики разрабатываемого языка. Этими конструкциями являются операторы, выражения, объявления функций, условные переходы, циклы и прочее.

Семантический анализ представляет собой этап компиляции, основной задачей которого является проверка смысловой корректности программы. Семантический анализ проверяет, имеет ли код программы логический смысл в заданном контексте при помощи анализа лексем в группах лексем, сформированных синтаксическим анализатором.

Таким образом, любой код на языке программирования проходит три стадии анализа. Первая стадия разбивает код на минимальные единицы, вторая стадия комбинирует минимальные единицы в группы, третья стадия анализирует является ли набор минимальных единиц в группе корректным.

1.2 Лексический анализатор

На этапе лексического анализа происходит работа с исходным текстом как с последовательностью символов. Задача анализатора — прочитать эту последовательность и сгруппировать символы в осмысленные элементарные блоки, которые называются лексемами или токенами.

Анализатор распознаёт, где начинается и заканчивается имя переменной (идентификатор), числовая константа, строковый литерал, ключевое слово языка (например, if, while, return) или оператор (+, =, ==). При этом он сознательно игнорирует всё, что не несёт смысловой нагрузки для последующих этапов: пробелы, табуляции, переводы строк и комментарии.

Результатом работы лексического анализатора является чистый, структурированный поток токенов, где каждый токен имеет свой тип и, при необходимости, значение. Этот поток уже готов для проверки структуры.

Написание анализатора с нуля является трудоемким процессом, поэтому существуют различные программные инструменты для написания программ для лексического анализа. Из известных: lex и flex. Они позволяют по шаблонам написать в файле правила по преобразованию входного потока в токены. После компиляции файла появится новый файл «lex.yy.c», содержащий функцию «yylex ()», используемую синтаксическим анализатором на втором этапе обработки кода.

1.3 Синтаксический анализатор

Синтаксический анализ, или парсинг, берёт на себя задачу проверки структуры программы. Получив от лексического анализатора поток токенов, синтаксический анализатор проверяет, можно ли из этой последовательности построить грамматически правильные предложения на целевом языке программирования. Он руководствуется формальным описанием языка — его грамматикой, которая определяет, как более мелкие конструкции (токены)

могут сочетаться в более крупные (выражения, операторы, объявления функций).

Например, грамматика диктует, что за ключевым словом if должна следовать условие в круглых скобках, а затем — оператор или блок. Если последовательность токенов нарушает эти правила, анализатор фиксирует синтаксическую ошибку. При успешном разборе парсер строит абстрактное синтаксическое дерево (АСД) — иерархическое представление программы, которое наглядно отражает вложенность и взаимосвязь всех её элементов. Этот этап отвечает исключительно за форму, а не за содержание.

Для обозначения минимальных по размеру токенов используют слово «терминал», для обозначения далее следующих по возрастающей иерархии токенов – «нетерминалы».

Написание синтаксического анализатора с нуля является трудоемким процессом, поэтому существуют различные программные инструменты для написания программ для синтаксического анализа. Из известных: yacc и bison.

Файл yacc/bison имеет определенный шаблон написания. После компиляции yacc/bison программ формируются два файла.

Первый файл «bison.tab.c» является самим синтаксическим анализатором, использующим функцию «yylex ()» из первого этапа. Синтаксический анализатор вызывает указанную функцию для получения токена от лексического анализатора. Как только токенов становится достаточно для какого-либо правила синтаксический анализатор применяет правило и заменяет несколько терминалов одним нетерминалом. Нетерминалы тоже могут быть сгруппированы в более высокий по иерархии нетерминал.

Второй файл «bison.tab.h» содержит численные обозначения токенов, используемых для разбора грамматики. Другими словами, данный файл

формирует перечисление «ENUM». Перечисление используется лексическим анализатором для корректной передачи токенов.

На рисунке 1 изображен пример синтаксического разбора математического выражения.

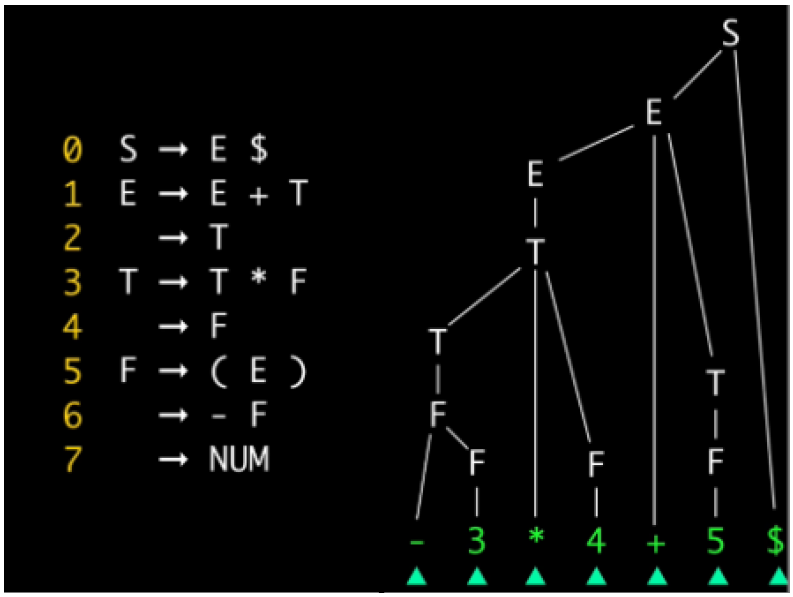

Рисунок 1 – разбор выражение синтаксическим анализатором

Из рисунка 1 видно, что построение «дерева» руководствуется некоторыми правилами замены комбинации токенов. Такая запись называется «нотацией Бэкуса-Наура» или «BNF-notation». Нотация BNF используется для краткого описания синтаксического анализатора. Используя нотацию, правила возможно перенести

1.4 Дерево разбора

Абстрактное синтаксическое дерево — это иерархическое представление структуры программы, которое строится синтаксическим анализатором после проверки грамматической корректности. В этом дереве корень соответствует всей программе, внутренние узлы — составным конструкциям вроде выражений или операторов, а листья — элементарным единицам, таким как идентификаторы или константы.

Синтаксический анализ, в процессе которого такое дерево строится, может выполняться двумя принципиально разными способами в зависимости от направления построения.

Нисходящий анализ работает от общего к частному. Он начинает со стартового символа грамматики, представляющего всю программу, и последовательно применяет правила, предсказывая и разворачивая её структуру до тех пор, пока не будет достигнут уровень входных токенов. Этот подход интуитивно понятен и часто реализуется методом рекурсивного спуска, но накладывает структурные ограничения на грамматику, требуя её предварительного преобразования для устранения неоднозначностей.

Восходящий анализ движется в обратном направлении — от частного к общему. Он начинает с чтения токенов и их накопления, постепенно комбинируя последовательности, которые соответствуют правым частям правил грамматики, и сводя их к более крупным нетерминальным символам. Этот процесс продолжается, пока вся входная последовательность не будет сведена к корневому стартовому символу. Наиболее известным методом такого разбора является LR-анализ, который, будучи более сложным в реализации, способен работать с более широким классом грамматик, чем нисходящий подход. Генераторы вроде Bison используют именно восходящие LR-методы для построения анализаторов.

2 Разработка компилятора для собственного языка программирования

2.1 Выбор целевой платформы

В качестве целевого языка трансляции выбран язык C. Этот выбор обусловлен двумя ключевыми факторами.

Первый фактор - универсальность и переносимость. Язык C является стандартом для системного программирования. Его компиляторы существуют

для практически любой вычислительной платформы, от микроконтроллеров до суперкомпьютеров.

С другой стороны, выбор языка C обусловлен его близостью к аппаратному уровню и минимальными накладными расходами на выполнение. Статическая типизация, явное управление памятью позволяют генерируемому коду работать с предсказуемой и высокой производительностью.

2.2 Разработка собственного языка

Язык программирования «MedievalGibberish» представляет из себя язык программирования, основой которого является тематика средневековья и магии. Отличительной особенностью языка, исходя из названия «MedievalGibberish» (Средневековая тарабарщина), является наличие сложно читаемых или странных перегруженных синтаксических конструкций в рамках средневековой и магической лексики.

Язык поддерживает объявление переменных, написание математических выражений, написание условных выражений, переопределение переменных, объявление функций, вызов функций, написание трех видов циклов, группировку кода, а также ветвления и подключение директив.

Главными особенностями языка являются:

— Обрамление значений разных типов данных,

— Обратный порядок объявления, инициализации и присваивания переменных,

— Порядок условных операций изменен: операция операнд операнд,

— Наличие специальных излишних конструкций для создания и вызова функций

— Наличие цикла с предусловием под названием «precycle»

Код на разработанном языке — приложение А.

2.3 Разработка лексического анализатора

Для создания лексического анализатора выбрана программа flex. Основными аргументами для выбора flex вместо lex и других анализаторов стали: некоммерческая основа распространения программы, постоянные обновления и исправления ошибок, быстродействие (в сравнении с lex, flex выполняет функцию анализа быстрее).

Написание лексического анализатора с помощью flex состоит из нескольких этапов.

Первым этапом является указание в файле лексера (лексического анализатора) кода на языке C. Код помещается в начало файла и обрамляется следующим синтаксисом: «%{ код на языке C %}». C-код, находящийся в начале файла используется для определения всех требуемых директив, создания переменных и функций для обработки и подготовки данных к отправлению через токены.

На рисунке 2 представлен готовый C-код с указанием используемых директив и функцией, удаляющей обрамление в поступившей для обработки строке.

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "BISON.tab.h"

char* strip_delimiters(const char* yytext)
{
    int len = strlen(yytext);
    if (len < 2) return strdup("");

    char* result = (char*)malloc(len - 1);  // len-2 символа + '\0'
    if (!result) return NULL;

    strncpy(result, yytext + 1, len - 2);
    result[len - 2] = '\0';
    return result;
}
%}
```

Рисунок 2 – C-код в начале файла лексического анализатора

Далее следует написание лексических правил, по которым анализатор поймет при встрече каких комбинаций символов какие токены отправлять. Лексические правила следует обрамлять по следующему шаблону: «%% правила %%»

На рисунке 3 изображена часть лексических правил.

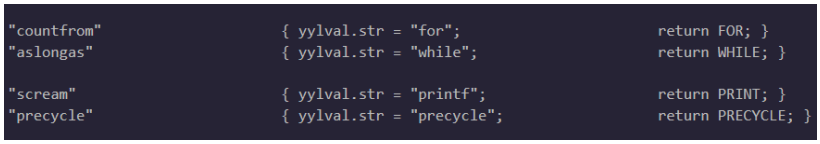

Рисунок 3 – лексические правила в файле лексера

Правила состоят из левой части, называемой «паттерном» и правой – «действием».

Паттерн — это правило сопоставления между конкретной комбинацией символов в исходном коде и типом токена, который должен быть создан. Действие – блок кода, содержащий инструкции при нахождении паттерна.

В случае с текущим лексическим анализатором в каждом действии лексер передает через специальную переменную «yylval» значение токена и возвращает сам токен.

Рассмотрим подробнее каждый блок правил.

— Регулярные выражения с различными обрамлениями (#int-число#, @строка@, &bool-значение&, `char-значение`, !float-число!) превращаются в токены значений определенного типа данных: «INTVAL», «STRVAL», «BOOLVAL», «CHARVAL», «FLOATVAL».

— Паттерны «flotless», «flot», «bell», «bard», «bull», «void» превращаются в токены типов данных: «INT_TYPE», «FLOAT_TYPE», «CHAR_TYPE», «STRING_TYPE», «BOOL_TYPE», «VOID_TYPE».

— Паттерны «dominion», «submission», «congruence», «dispersion», «ascendancy», «subjugation» превращаются в токены условных операций: «GE», «LE», «EQ», «NE», «GT», «LT».

— Паттерны «boost», «cut», «grow», «bamboozle» превращаются в токены математических операций: «ADD», «SUB», «MUL», «DIV».

— Паттерны «norkfork», «fork», «nork» превращаются в токены ветвлений: «ELSEIF», «IF», «ELSE».

— Паттерны «[learnspell]», «spell», «throw», «->» превращаются в токены, относящиеся к созданию и вызову функций: «LEARNSPELL», «SPELL», «RETURN», «STAFF».

— Паттерн «countfrom» и «aslongas», а также «precycle» используются для создания циклов, они превращаются в следующие токены: «FOR», «WHILE», «PRECYCLE».

— Паттерн «SCREAM» превращается в токен «PRINT» и используется для вызова функции «printf» для вывода консоль.

— Паттерн «=» применяется для присваивания значений переменным, превращается в токен «ASSIGN».

— Паттерн «~» применяется для отделения законченных команд на собственном языке и превращается в токен «SEPARATOR».

— Паттерн «,» применяется для отделения аргументов в функциях, превращается в токен «ENUMERATOR».

— Паттерны «<» и «>» являются заменой обычных скобочек из языков программирования (используются в функциях, циклах и ветвлениях), заменяются на токены «LARC», «RARC».

— Паттерны «{» и «}» остаются прежними и предназначены для группировки кода, заменяются на токены «LBRACE» и «RBRACE».

— Паттерн с регулярным выражением «[любые символы]» предназначен для указания директив и заменяется на токен «INCLUDE».

— Паттерн [CALL HERCULES] является сокращением основных директив, используемых в языке С. Заменяется на токен «CALLHERCULES».

После указания всех лексем переопределяются встроенные в flex функции. В случае текущей работы требуется переопределить функцию «yywrap ()». Функция «yywrap» предназначена для определения, конечный ли файл лексем обрабатывается или нет. При возврате нуля, цикл, читающий файлы лексеров продолжит чтение правил, иначе остановит работу.

Полный код лексического анализатора — приложение Б.

2.4 Разработка синтаксического анализатора

Для создания синтаксического анализатора выбрана программа bison. Bison является современной версией уасс и имеет бесшовную интеграцию для работы в паре с лексическим анализатором flex.

Написание синтаксического анализатора состоит из нескольких этапов. Этапы схожи с этапами лексического анализатора.

Первый этап – указание вспомогательного С-кода в обрамлении из «%{» и «%}». С-код, находящийся в начале файла используется для определения всех требуемых директив, создания переменных и функций для обработки и подготовки данных к транслированию на целевую платформу. Также обязательно требуется ссылаться на два метода, требуемых Bison’ом: «yylex()» и «yyerror()». Без ссылки на эти методы компиляция файла синтаксического анализатора выдаст ошибку. «yylex()» - метод из лексического анализатора, требуется для получения токенов из анализируемого текста. «yyerror()» требуется для вывода ошибок.

На рисунке 4 изображен код создания первого этапа синтаксического анализатора Bison.

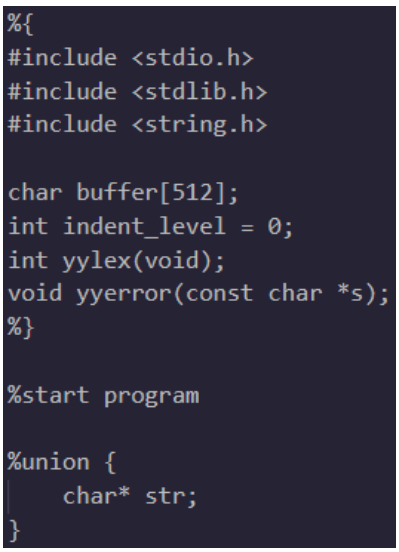

Рисунок 4 – вспомогательный код файла Bison

После подключения вспомогательного кода указывается корневой нетерминал грамматики. По алгоритму работы Bison должен преобразовать терминалы в нетерминалы, которые в итоге сложатся в указанный корневой нетерминал. Указание происходит следующим образом: «%start program».

Для передачи данных с токенами используется переменная «yylval», используемая в лексическом анализе в блоке действия у лексического правила. Чтобы определить, какой тип данных будет хранится в yylval вместе с конкретным токеном, используется конструкция «%union { типы данных название}».

Далее через «%token <тип_данных> имя_токена» определяется список токенов, используемый Bison и Flex.

После указывается какие нетерминалы будут участвовать в синтаксическом анализе: «%type <тип_данных> имя_нетерминала».

Перед блоком с правилами дополнительно возможно указать приоритет токенов при построении дерева разбора через «%left имена_токенов» и «%nonassoc имена_токенов».

Самая главная часть – синтаксические правила по группировке терминалов (токенов) и нетерминалов по «восходящему» принципу.

На рисунке 5 изображена часть синтаксических правил.

```
STATEMENTS:
    STATEMENT
    { $$ = $1; }
    |
    STATEMENTS STATEMENT
    { sprintf(buffer, "%s\n%s", $1, $2); $$ = strdup(buffer); }
```

Рисунок 5 – синтаксическое правило STATEMENTS

Правила помещаются в точно такое же обрамление, как и в случае с лексическим анализатором, в «%%». Каждое правило состоит из заголовка (например, STATEMENTS:) и набора терминалов и нетерминалов ниже, которые будут группированы в один нетерминал (например, STATEMENTS SEPARATOR STATEMENT будут преобразованы в STATEMENTS). После каждого набора терминалов и нетерминалов возможно действие. Действие обрамляется фигурными скобочками. В действиях возможно обращение к данным, сохраненным в терминалах и нетерминалах посредством указания символа «$» и порядкового номера в группе терминалов и нетерминалов. Для обращения к нетерминалу, в который группируется группа терминалов и нетерминалов, используется комбинация символов «$$».

Последней частью синтаксического анализатора является определение метода «yyerror()» для обработки ошибок и определение главной функции программы «int main()». Главная функция обязательно должна содержать вызов метода yyparse(), контролирующего выполнение разбора токенов (парсинга) и вызывающего метод «yylex()».

В случае с разрабатываемым компилятором принято решение по транслированию собственного языка на язык C следующим образом: лексер читает файл с языком MedievalGibberfish и разбивает его на токены, токены содержат строчные данные, строчные данные содержат информацию о самих себе, но в синтаксисе языка C; Парсер собирает токены в нетерминалы и

соединяет данные вместе; Таким образом из нескольких токенов языка MedievalGibberfish получается транслирование на язык C.

Для объединения строчных данных используется функция «sprint» по объединению строк в буфере, «strdup» используется для копирования данных из буфера в кучу (heap), ссылка на которые будет содержаться в новом создаваемом нетерминале. Для получения результата используется функция «printf» по выводу текста в консоль. Поток вывода переводится при помощи перевода потока «pipe» в clang и компилируется с новый исполняемый файл, который далее вызывается для получения результата работы кода.

Код синтаксического анализатора — приложение В.

2.5 Компиляция компилятора и создание MakeFile

Для компиляции компилятора вручную требуется выполнить несколько команд:

1. Команда генерации парсера: bison –d BISON.y (создаст файл синтаксического анализатора на языке C с расширением «.c», а также заголовочный файл с ENUM всех токенов с расширением «.h»).

2. Команда генерации лексера: flex –o lex.c LEXER.l (создаст файл лексического анализатора на языке C с расширением «.c»).

3. Команда компиляции исполняемого файла компилятора: clang lex.c BISON.tab.c –o myCompiler

В итоге получится исполняемый файл компилятора, на вход которого доступно подать текстовые данные.

На рисунке 6 представлен процесс ручной компиляции компилятора и проверка его работоспособности путем передачи файла с кодом на разработанном языке программирования.

```
bibub@bibub-VirtualBox:~/COURSE$ bison -d BISON.y
BISON.y: warning: 1 shift/reduce conflict [-Wconflicts-sr]
BISON.y: note: rerun with option '-Wcounterexamples' to generate conflict counte
rexamples
bibub@bibub-VirtualBox:~/COURSE$ flex -o lex.c LEXER.l
bibub@bibub-VirtualBox:~/COURSE$ clang BISON.tab.c lex.c -o myCompiler
bibub@bibub-VirtualBox:~/COURSE$ myCompiler < code.mg
myCompiler: command not found
bibub@bibub-VirtualBox:~/COURSE$ ./myCompiler < code.mg
#include "somelibrary.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <stdbool.h>
int main()
{
bool cat=true;
if(cat)
{
cat=false;
}
if(cat==false)
```

Рисунок 6 – ручная компиляция компилятора и проверка работоспособности

Для автоматизации процесса и упрощения сборки компилятора используется утилита «make», использующая файл с именем «makefile». Makefile позволяет при помощи синтаксиса, похожего на синтаксис синтаксического анализатора bison, указать как и в каком порядке выполнять указанные команды.

На рисунке 7 изображен файл «makefile» для автоматизации процесса

```
1  COMP = clang
2
3  all: compiler
4
5  compiler: BISON.tab.c BISON.tab.h lex.c
6      $(COMP) BISON.tab.c lex.c -o myCompiler
7
8  lex.c: LEXER.l
9      flex -o lex.c LEXER.l
10
11 BISON.tab.c: BISON.y
12     bison -d BISON.y
13
14 clear:
15     rm myCompiler BISON.tab.h BISON.tab.c lex.c
```

Рисунок 7 – автоматизация сборки компилятора при помощи утилиты make

Код makefile — приложение Г.

3 Контрольный пример

Для проверки работоспособности компилятора и правильности трансляции кода на язык C, напишем контрольный пример и выполним его после компиляции исполняемого файла.

На рисунке 8 представлен исходный код на разработанном языке.

```
[CALL HERCULES]

[learnspell]flotless->?returnPowered?<?num? flotless, ?ctr? flotless>
{
    #1# = ?result? flotless~
        countfrom<?ctr? = ?counter? flotless ~ dominion<?counter?,#1#> ~ ?counter? cut #1# = ?counter?>
        {
            ?result? grow ?num? = ?result?~
        }
        throw ?result?~
}

[learnspell]flotless->?main?<>
{
    &truth& = ?cat? bull~
    fork<?cat?>
    {
        scream<@Cat has been detected@ bard>~
        &bluf& = ?cat?~
    }
    precycle< congruence<?cat?,&bluf&> ~ #0# =?ctr? flotless ~ submission<?ctr?,#4#> ~ ?ctr? boost #1# = ?ctr?>
    {
        scream<@Meow@ bard>~
    }

    spell->?returnPowered?<#2#, #3#> = ?poweredNumber? flotless~
    scream<@Powered number is@ bard>~
    scream<?poweredNumber? flotless>~

    #5# = ?ctrForWhile? flotless~
    aslongas<dominion<?ctrForWhile?,#1#>>
    {
        ?ctrForWhile? cut #1# = ?ctrForWhile?~
        scream<?ctrForWhile? flotless>~

        fork<congruence<?ctrForWhile?,#4#>>
        {
            scream<@This is four@ bard>~
        }
                norkfork<congruence<?ctrForWhile?,#3#>>
                {
                        scream<@Number three@ bard>~
                }
                nork
                {
                        scream<@The number is too small Cant read@ bard>~
                }

    }
    throw #0#~
}
```

Рисунок 8 – код на разработанном языке

Код исходного файла программы — приложение А.

Транслируем пример, сохраним результат в отдельный файл с расширением «C».

На рисунке 9 представлен код, транслированный на C с исходного разработанного программного языка программирования.

```
B:\GIT\CompilerFLEXBISON\main>myCompiler < code.mg
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <stdbool.h>
int returnPowered(int num,int ctr)
{
int result=1;
for(int counter=ctr;counter>=1;counter=counter-1)
{
result=result*num;
}
return result;
}
int main()
{
bool cat=true;
if(cat)
{
printf("%s\n", "Cat has been detected");
cat=false;
}
if(cat==false)
{
for(int ctr=0;ctr<=4;ctr=ctr+1)
{
printf("%s\n", "Meow");
}
}
int poweredNumber=returnPowered(2,3);
printf("%s\n", "Powered number is");
printf("%d\n", poweredNumber);
int ctrForWhile=5;
while(ctrForWhile>=1)
{
ctrForWhile=ctrForWhile-1;
printf("%d\n", ctrForWhile);
if(ctrForWhile==4)
{
printf("%s\n", "This is four");
}
else if(ctrForWhile==3)
{
printf("%s\n", "Number three");
}
else
{
printf("%s\n", "The number is too small Cant read");
}
}
return 0;
}
B:\GIT\CompilerFLEXBISON\main>
```

Рисунок 9 – транслированный код с разработанного языка программирования

Код транслированного кода исходной программы — приложение Д.

Скомпилируем код в исполняемый файл и попробуем выполнить его.

На рисунке 10 изображена компиляция в исполняемый файл.

```
bibub@bibub-VirtualBox:~/COURSE$ ./myCompiler < code.mg > out.c; clang out.c -o pr
```

Рисунок 10 – компиляция в исполняемый файл

На рисунке 11 изображено выполнение исполняемого файла и вывод кода программы, написанной на разработанном языке.

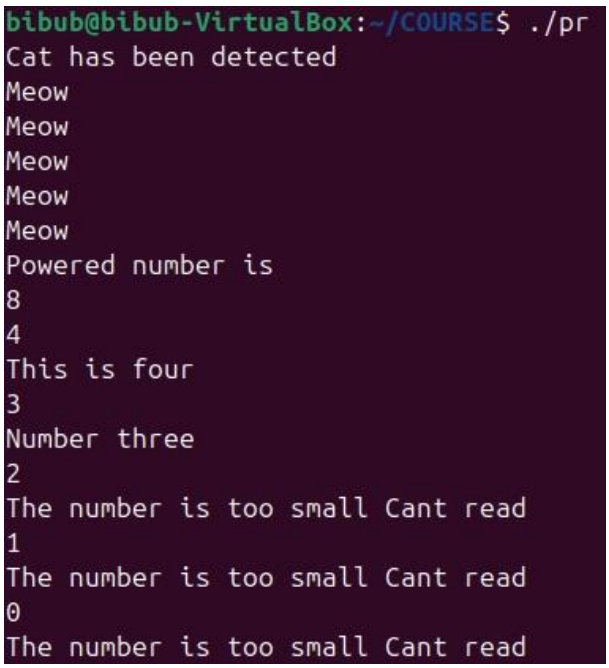

Рисунок 11 – запуск исполняемого файла и вывод

Из результатов работы программы видно, что контрольный пример успешно выполнился. Разработанный язык успешно позволяет объявлять переменные, переопределять переменные, выполнять логические и математические операции создавать функции, циклы, а также ветвления и циклы с предусловием.

ЗАКЛЮЧЕНИЕ

В ходе выполнения курсовой работы был разработан оригинальный компилируемый язык программирования MedievalGibberish, основанный на тематике средневековья и магии, и реализован полнофункциональный компилятор для него. Язык сочетает в себе нестандартную, стилизованную лексику и синтаксические конструкции с чёткой архитектурной основой, что позволяет транслировать программы в корректный и исполняемый код на языке C.

Все поставленные задачи были успешно решены: проведён анализ предметной области и этапов компиляции, спроектирован синтаксис собственного языка, разработаны лексический и синтаксический анализаторы с использованием инструментов Flex и Bison, реализована генерация кода на C, создана система автоматической сборки с помощью Makefile и выполнено тестирование на контрольных примерах.

Работоспособность компилятора подтверждена корректной трансляцией исходного кода на языке MedievalGibberish и последующей успешной компиляцией в исполняемые программы с помощью компилятора Clang.

Практическая значимость работы заключается в углублённом освоении принципов построения компиляторов, методов работы с формальными грамматиками и автоматической генерации анализаторов. Разработанный компилятор служит наглядным примером применения теории трансляции кода на практике.

В перспективе язык может быть расширен новыми синтаксическими конструкциями, типами данных и семантическим анализом для повышения выразительности и надёжности.

СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ

1. Конова, Е. А. Основы программирования на языке С : учебное пособие / Е. А. Конова ; Е. А. Конова, Г. А. Поллак, А. М. Ткачев ; под ред. А. М. Ткачева ; М-во образования и науки Российской Федерации, Федеральное агентство по образованию, Южно-Уральский гос. ун-т, Каф. информатики. – Челябинск : Изд-во ЮУрГУ, 2004. – ISBN 5-696-03601-5. – EDN QMQYTR.

2. Теоретические основы компиляции. – Новосибирск : Новосибирский государственный университет экономики и управления "НИНХ", 1980. – 172 с. – EDN UIREIR.

3. Корзун, Д. Ж. Lex и Yacc: генераторы программ лексического и синтаксического анализа : учебное пособие для математических специальностей университетов , для студентов высших учебных заведений, обучающихся по группе математических и механических направлений и специальностей / Д. Ж. Корзун, Ю. А. Богоявленский ; редкол. : А. В. Воронин [и др.]. – Петрозаводск : Изд-во ПетрГУ, 2007. – 100 с. – (Серия "Информатика: основы и приложения"). – ISBN 978-5-8021-0701-0. – EDN QMRFIL.

4. Shankel, J. Little languages with Lex, Yacc, and MFC / J. Shankel // Dr. Dobb's Journal: Software Tools for the Professional Programmer. – 1999. – Vol. 24, No. 1. – P. 28-31. – EDN CZBSSP.

ПРИЛОЖЕНИЕ А

Код на разработанном языке программирования

```
[CALL HERCULES]


[learnspell]flotless->?returnPowered?<?num? flotless, ?ctr? flotless>
{
#1# = ?result? flotless~
    countfrom<?ctr? = ?counter? flotless ~ dominion<?counter?,#1#> ~ ?counter? cut #1# = ?counter?>
    {
        ?result? grow ?num? = ?result?~
    }
    throw ?result?~
}


[learnspell]flotless->?main?<>
{
&truth& = ?cat? bull~
fork<?cat?>
{
     scream<@Cat has been detected@ bard>~
     &bluf& = ?cat?~
}
precycle< congruence<?cat?,&bluf&> ~ #0# =?ctr? flotless ~ submission<?ctr?,#4#> ~ ?ctr? boost #1# = ?ctr?>
{
     scream<@Meow@ bard>~
}


spell->?returnPowered?<#2#, #3#> = ?poweredNumber? flotless~
```

```
scream<@Powered number is@ bard>~
scream<?poweredNumber? flotless>~

#5# = ?ctrForWhile? flotless~
aslongas<dominion<?ctrForWhile?,#1#>>
{
        ?ctrForWhile? cut #1# = ?ctrForWhile?~
        scream<?ctrForWhile? flotless>~

        fork<congruence<?ctrForWhile?,#4#>>
        {
            scream<@This is four@ bard>~
        }
            norkfork<congruence<?ctrForWhile?,#3#>>
            {
                scream<@Number three@ bard>~
            }
            nork
            {
                scream<@The number is too small Cant read@ bard>~
            }

}
```

throw #0#~

ПРИЛОЖЕНИЕ Б

Полный код лексического анализатора

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "BISON.tab.h"

char* strip_delimiters(const char* yytext)
{
	int len = strlen(yytext);
	if (len < 2) return strdup("");

	char* result = (char*)malloc(len - 1);  // len-2 символа + '\0'
	if (!result) return NULL;

	strncpy(result, yytext + 1, len - 2);
	result[len - 2] = '\0';
	return result;
}
%}

%%
"[CALL HERCULES]" { yylval.str = strdup("#include <stdio.h>\n#include <stdlib.h>\n#include <string.h>\n#include <ctype.h>\n#include <stdbool.h>"); return CALLHERCULES; }

\#[0-9]+\# { yylval.str = strdup(strip_delimiters(yytext)); return INTVAL; }
```

```
\![0-9]+(\.[0-9]+)?\!        { yylval.str = strdup(strip_delimiters(yytext));
        return FLOATVAL; }
\`[A-Za-z0-9_]{1}\`          { yylval.str =
strdup(strip_delimiters(yytext));       return CHARVAL; }
\@.*\@          {
                                        char*          stripped          =
strip_delimiters(yytext);
                                        // Добавляем кавычки
                                        yylval.str                       =
malloc(strlen(stripped) + 3);
                                        sprintf(yylval.str,          "\"%s\"",
stripped);
                                        free(stripped);   // Освобождаем
промежуточную строку
                                        return STRVAL;
    }
\?[a-zA-Z_][a-zA-Z0-9_]*\?      { yylval.str =
strdup(strip_delimiters(yytext));       return VARIABLE; }
    "&truth&"                   { yylval.str = strdup("true");
            return BOOLVAL; }
    "&bluf&"                    { yylval.str = strdup("false");
        return BOOLVAL; }



    "countfrom"                 { yylval.str = strdup("for");
            return FOR; }
    "aslongas"                  { yylval.str = strdup("while");
            return WHILE; }


    "scream"                    { yylval.str = strdup("printf");
        return PRINT; }
    "precycle"                  { yylval.str = strdup("precycle");
            return PRECYCLE; }
```

```
"dominion"                          { yylval.str = strdup(">=");
                    return GE; }

"submission"                    { yylval.str = strdup("<=");
          return LE; }

"congruence"                 { yylval.str = strdup("==");
          return EQ; }

"dispersion"                { yylval.str = strdup("!=");
     return NE; }

"ascendacy"                     { yylval.str = strdup(">");
                    return GT; }

"subjugation"                { yylval.str = strdup("<");
          return LT; }


"boost"                             { yylval.str = strdup("+");
                    return ADD; }

"cut"                       { yylval.str = strdup("-");
          return SUB; }

"grow"                              { yylval.str = strdup("*");
                    return MUL; }

"bamboozle"                     { yylval.str = strdup("/");
          return DIV; }


"flotless"                          {  yylval.str = strdup("int");
     return INT_TYPE; }

"flot"                      {  yylval.str = strdup("float");
     return FLOAT_TYPE; }

"bell"                      {  yylval.str = strdup("char");
     return CHAR_TYPE; }

"bard"                              {  yylval.str = strdup("char*");
     return STRING_TYPE; }

"bull"                              {  yylval.str = strdup("bool");
     return BOOL_TYPE; }

"void"                              {  yylval.str = strdup("void");
     return VOID_TYPE; }
```

```
"norkfork"                                  { yylval.str = strdup("else if");
                        return ELSEIF; }
"fork"                      { yylval.str = strdup("if");
        return IF; }
"nork"                      { yylval.str = strdup("else");
        return ELSE; }


"\[learnspell\]"                    { yylval.str = strdup("[learnspell]");
        return LEARNSPELL; }
"spell"                     { yylval.str = strdup("spell");
        return SPELL; }
"throw"                     { yylval.str = strdup("return");
    return RETURN; }


"->"                    { yylval.str = strdup("wizardstaff");
    return STAFF; }
"="                     { yylval.str = strdup("=");
        return ASSIGN; }
~                       { yylval.str = strdup(";");
    return SEPARATOR; }
\,                      { yylval.str = strdup(",");
    return ENUMERATOR; }
:                                           { yylval.str = strdup(":");
                            return COLON;}
"<"                     { yylval.str = strdup("(");
        return LARC; }
">"                     { yylval.str = strdup(")");
        return RARC; }
"{"                     { yylval.str = strdup("{");
        return LBRACE; }
"}"                     { yylval.str = strdup("}");
        return RBRACE; }


\[[^\]]+\]              { yylval.str = strdup(strip_delimiters(yytext));    return
INCLUDE; }
```

```
[ \t\r\n]+                          { /* skip */ }
.                                   { yylval.str = strdup("unknown");
        return UNKNOWN; }


%%


int yywrap() { return 1; }
```

ПРИЛОЖЕНИЕ В

Код синтаксического анализатора

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char buffer[512];
int indent_level = 0;
int yylex(void);
void yyerror(const char *s);
%}

%start program

%union {
        char* str;
}



%token <str> CALLHERCULES INCLUDE
%token <str> INTVAL FLOATVAL CHARVAL STRVAL BOOLVAL VARIABLE
%token <str> INT_TYPE FLOAT_TYPE STRING_TYPE CHAR_TYPE BOOL_TYPE VOID_TYPE
%token <str> IF ELSEIF ELSE RET CALL
%token <str> LARC RARC LBRACE RBRACE
%token <str> WHILE FOR PRINT PRECYCLE POWER
%token <str> GE LE EQ NE GT LT
%token <str> ASSIGN ADD SUB MUL DIV
```

```
%token <str> SEPARATOR ENUMERATOR COLON SPELL RETURN STAFF
LEARNSPELL

%token <str> UNKNOWN


%type <str> program BLOCK STATEMENTS STATEMENT

%type <str> CONSOLE_OUTPUT

%type <str> PYCLE

%type <str> LOOP COUNTFROM ASLONGAS

%type <str> NORKFORK NORK FORK CONDITION

%type <str> FUNC_HEADER_WITH_PARAMS FUNC_HEADER
FUNC_HEADER_PARAMS RET_VAL

%type <str> FUNC_CALL_WITH_ARGS FUNC_CALL_ARGS
FUNC_CALL_PART

%type <str> FULL_PARTIAL_INITS PARTIAL_INITS FULL_INIT
PARTIAL_INIT VAR_DECLARE VALUE_ASSIGN

%type <str> EXPR

%type <str> MATH_EXPR SUBADD MULDIV

%type <str> BOOL_EXPR

%type <str> ATOM ARCS

%type <str> VALUE TYPE COMPARE_OPER


%left ADD SUB

%left MUL DIV

%nonassoc GE LE EQ NE GT LT

%nonassoc LARC RARC


%%

program:

        STATEMENTS

        { printf("%s", $1); }
```

```
//BLOCK - это максимальные ГРУППИРОВАННЫЕ по размеру блоки программы, подающиеся на вывод
BLOCK:
    LBRACE STATEMENTS RBRACE
    { sprintf(buffer, "%s\n%s\n%s", $1, $2, $3); $$ = strdup(buffer); }
//Statements - это максимальные по размеру блоки программы, подающиеся на вывод
STATEMENTS:
    STATEMENT
    { $$ = $1; }
    |
    STATEMENTS STATEMENT
    { sprintf(buffer, "%s\n%s", $1, $2); $$ = strdup(buffer); }
STATEMENT:
    EXPR SEPARATOR
    { sprintf(buffer, "%s%s", $1,$2); $$ = strdup(buffer); }
    |
    FULL_INIT SEPARATOR
    { sprintf(buffer, "%s%s", $1,$2); $$ = strdup(buffer); }
    |
    VAR_DECLARE SEPARATOR
    { sprintf(buffer, "%s%s", $1,$2); $$ = strdup(buffer); }
    |
    PARTIAL_INIT SEPARATOR
    { sprintf(buffer, "%s%s", $1,$2); $$ = strdup(buffer); }
    |
    FUNC_CALL_WITH_ARGS SEPARATOR
    { sprintf(buffer, "%s%s", $1,$2); $$ = strdup(buffer); }
    |
    FUNC_HEADER_WITH_PARAMS
```

```
{ $$ = $1; }
|
NORKFORK
{ $$ = $1; }
|
NORK
{ $$ = $1; }
|
FORK
{ $$ = $1; }
|
LOOP
{ $$ = $1; }
|
BLOCK
{ $$ = $1; }
|
PYCLE
{ $$ = $1; }
|
CONSOLE_OUTPUT SEPARATOR
{ sprintf(buffer, "%s%s", $1,$2); $$ = strdup(buffer); }
|
CALLHERCULES
{ $$ = $1; }
|
INCLUDE
{ sprintf(buffer, "#include %s", $1); $$ = strdup(buffer); }
|
RET_VAL SEPARATOR
```

```
        { sprintf(buffer, "%s%s", $1,$2); $$ = strdup(buffer); }


//Вывод в консоль
CONSOLE_OUTPUT:
        PRINT LARC EXPR INT_TYPE RARC
        {
                sprintf(buffer, "printf(\"%%d\\n\", %s)", $3); $$ = strdup(buffer);
        }
        |
        PRINT LARC EXPR FLOAT_TYPE RARC
        {
                sprintf(buffer, "printf(\"%%f\\n\", %s)", $3); $$ = strdup(buffer);
        }
        |
        PRINT LARC EXPR STRING_TYPE RARC
        {
                sprintf(buffer, "printf(\"%%s\\n\", %s)", $3); $$ = strdup(buffer);
        }
        |
        PRINT LARC EXPR CHAR_TYPE RARC
        {
                sprintf(buffer, "printf(\"%%c\\n\", %s)", $3); $$ = strdup(buffer);
        }
        |
        PRINT LARC EXPR BOOL_TYPE RARC
        {
                sprintf(buffer, "printf(\"%%d\\n\", %s)", $3); $$ = strdup(buffer);
        }
//Цикл с предусловием
```

PYCLE:

// 1 2 3 4 5 6 7 8 9 10 11

PRECYCLE LARC CONDITION SEPARATOR FULL_PARTIAL_INITS SEPARATOR CONDITION SEPARATOR PARTIAL_INITS RARC BLOCK

{ sprintf(buffer, "if%s%s%s\n{\nfor%s%s%s%s%s%s%s\n%s\n}", $2, $3, $10, $2, $5, $6, $7, $8, $9, $10, $11); $$ = strdup(buffer); }

//Циклы

LOOP:

COUNTFROM

{ $$ = $1; }

|

ASLONGAS

{ $$ = $1; }

COUNTFROM:

FOR LARC FULL_PARTIAL_INITS SEPARATOR CONDITION SEPARATOR PARTIAL_INITS RARC

{ sprintf(buffer, "%s%s%s%s%s%s%s%s", $1, $2, $3, $4, $5, $6, $7, $8); $$ = strdup(buffer); }

ASLONGAS:

WHILE LARC CONDITION RARC

{ sprintf(buffer, "%s%s%s%s", $1, $2, $3, $4); $$ = strdup(buffer); }

//Операторы ветвлений

NORKFORK:

ELSEIF LARC CONDITION RARC

{ sprintf(buffer, "%s%s%s%s", $1, $2, $3, $4); $$ = strdup(buffer); }

NORK:

ELSE

```
        { $$ = $1; }
FORK:
        IF LARC CONDITION RARC
        { sprintf(buffer, "%s%s%s%s", $1, $2, $3, $4); $$ = strdup(buffer); }
CONDITION:
        EXPR
        { $$ = $1; }


//Возврат значения
RET_VAL:
        RETURN VOID_TYPE
        { $$ = $1; }
        |
        RETURN FUNC_CALL_WITH_ARGS
        { sprintf(buffer, "%s %s", $1,$2); $$ = strdup(buffer); }
        |
        RETURN EXPR
        { sprintf(buffer, "%s %s", $1,$2); $$ = strdup(buffer); }
//Создание заголовка функции
FUNC_HEADER_WITH_PARAMS:
        FUNC_HEADER FUNC_HEADER_PARAMS
        { sprintf(buffer, "%s%s", $1, $2); $$ = strdup(buffer); }
FUNC_HEADER:
        LEARNSPELL TYPE STAFF VARIABLE
        { sprintf(buffer, "%s %s", $2, $4); $$ = strdup(buffer); }
FUNC_HEADER_PARAMS:
        ARCS
        { $$ = $1; }
        |
```

```
        LARC VAR_DECLARE
        { sprintf(buffer, "%s%s", $1, $2); $$ = strdup(buffer); }
        |
        FUNC_HEADER_PARAMS ENUMERATOR VAR_DECLARE
        { sprintf(buffer, "%s%s%s", $1, $2, $3); $$ = strdup(buffer); }
        |
        FUNC_HEADER_PARAMS RARC
        { sprintf(buffer, "%s%s", $1, $2); $$ = strdup(buffer); }

//Создание вызова функции
FUNC_CALL_WITH_ARGS:
        FUNC_CALL_PART FUNC_CALL_ARGS
        { sprintf(buffer, "%s%s", $1, $2); $$ = strdup(buffer); }
FUNC_CALL_PART:
        SPELL STAFF VARIABLE
        { sprintf(buffer, "%s", $3); $$ = strdup(buffer); }
FUNC_CALL_ARGS:
        ARCS
        { $$ = $1; }
        |
        LARC ATOM
        { sprintf(buffer, "%s%s", $1, $2); $$ = strdup(buffer); }
        |
        FUNC_CALL_ARGS ENUMERATOR ATOM
        { sprintf(buffer, "%s%s%s", $1, $2, $3); $$ = strdup(buffer); }
        |
        FUNC_CALL_ARGS RARC
        { sprintf(buffer, "%s%s", $1, $2); $$ = strdup(buffer); }
```

```
//Объявление и присваивание переменной (FULL_PARTIAL_INITS и PARTIAL_INITS для цикла for!)
FULL_PARTIAL_INITS:
        FULL_INIT
        { $$ = $1; }
        |
        FULL_PARTIAL_INITS ENUMERATOR PARTIAL_INITS
        { sprintf(buffer, "%s%s%s", $1, $2, $3); $$ = strdup(buffer); }
PARTIAL_INITS:
        PARTIAL_INIT
        { $$ = $1; }
        |
        PARTIAL_INITS ENUMERATOR PARTIAL_INIT
        { sprintf(buffer, "%s%s%s", $1, $2, $3); $$ = strdup(buffer); }
FULL_INIT:
        VALUE_ASSIGN VAR_DECLARE
        { sprintf(buffer, "%s%s", $2, $1); $$ = strdup(buffer); }
PARTIAL_INIT:
        VALUE_ASSIGN VARIABLE
        { sprintf(buffer, "%s%s", $2, $1); $$ = strdup(buffer); }
VAR_DECLARE:
        VARIABLE TYPE
        { sprintf(buffer, "%s %s", $2, $1); $$ = strdup(buffer); }
VALUE_ASSIGN:
        EXPR ASSIGN
        { sprintf(buffer, "%s%s", $2, $1); $$ = strdup(buffer); }


//Объединение нетерминалов операций
EXPR:
```

```
        MATH_EXPR
        { $$ = $1; }
        |
        BOOL_EXPR
        { $$ = $1; }

//Обработка математических операций
MATH_EXPR:
        SUBADD
        { $$ = $1; }
SUBADD:
        MULDIV
        { $$ = $1; }
        |
        SUBADD SUB MULDIV
        { sprintf(buffer, "%s%s%s", $1, $2, $3); $$ = strdup(buffer); }
        |
        SUBADD ADD MULDIV
        { sprintf(buffer, "%s%s%s", $1, $2, $3); $$ = strdup(buffer); }
MULDIV:
        ATOM
        { $$ = $1; }
        |
        MULDIV MUL ATOM
        { sprintf(buffer, "%s%s%s", $1, $2, $3); $$ = strdup(buffer); }
        |
        MULDIV DIV ATOM
        { sprintf(buffer, "%s%s%s", $1, $2, $3); $$ = strdup(buffer); }

//Обработка условных операций
```

```
BOOL_EXPR:
        COMPARE_OPER LARC ATOM ENUMERATOR ATOM RARC
        { sprintf(buffer, "%s%s%s", $3, $1, $5); $$ = strdup(buffer); }

//Обобщение терминалов и схлопывание выражений
ATOM:
        VALUE
        { $$ = $1; }
        |
        VARIABLE
        { $$ = $1; }
        |
        LARC EXPR RARC
        { sprintf(buffer, "%s%s%s", $1,$2,$3); $$ = strdup(buffer); }
ARCS:
        LARC RARC
        { sprintf(buffer, "%s%s", $1, $2); $$ = strdup(buffer); }
// ТЕРМИНАЛЫ
VALUE:
        INTVAL
        { $$ = $1; }
        |
        FLOATVAL
        { $$ = $1; }
        |
        CHARVAL
        { $$ = $1; }
        |
        STRVAL
        { $$ = $1; }
```

```
        |
        BOOLVAL
        { $$ = $1; }
        ;
TYPE:
        INT_TYPE
        { $$ = $1; }
        |
        FLOAT_TYPE
        { $$ = $1; }
        |
        CHAR_TYPE
        { $$ = $1; }
        |
        STRING_TYPE
        { $$ = $1; }
        |
        BOOL_TYPE
        { $$ = $1; }
        |
        VOID_TYPE
        { $$ = $1; }
COMPARE_OPER:
        GE
        { $$ = $1; }
        |
        LE
        { $$ = $1; }
        |
        EQ
```

```
        { $$ = $1; }
        |
        NE
        { $$ = $1; }
        |
        GT
        { $$ = $1; }
        |
        LT
        { $$ = $1; }
        ;
%%

int main() {
        yyparse();
        return 0;
}

void yyerror(const char *s) {
        fprintf(stderr, "Error: %s\n", s);
}
```

ПРИЛОЖЕНИЕ Г

Код makefile

```
COMP = clang

all: compiler

compiler: BISON.tab.c BISON.tab.h lex.c
 $(COMP) BISON.tab.c lex.c -o myCompiler

lex.c: LEXER.l
 flex -o lex.c LEXER.l

BISON.tab.c: BISON.y
 bison -d BISON.y

clear:
 rm myCompiler BISON.tab.h BISON.tab.c lex.c
```

ПРИЛОЖЕНИЕ Д

Код транслированного файла программы

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <stdbool.h>
int returnPowered(int num,int ctr)
{
int result=1;
for(int counter=ctr;counter>=1;counter=counter-1)
{
result=result*num;
}
return result;
}
int main()
{
bool cat=true;
if(cat)
{
printf("%s\n", "Cat has been detected");
cat=false;
}
if(cat==false)
{
for(int ctr=0;ctr<=4;ctr=ctr+1)
{
printf("%s\n", "Meow");
```

```
}
}
int poweredNumber=returnPowered(2,3);
printf("%s\n", "Powered number is");
printf("%d\n", poweredNumber);
int ctrForWhile=5;
while(ctrForWhile>=1)
{
ctrForWhile=ctrForWhile-1;
printf("%d\n", ctrForWhile);
if(ctrForWhile==4)
{
printf("%s\n", "This is four");
}
else if(ctrForWhile==3)
{
printf("%s\n", "Number three");
}
else
{
printf("%s\n", "The number is too small Cant read");
}
}
return 0;
}
```

ПРИЛОЖЕНИЕ Е

Диаграммы синтаксического анализатора

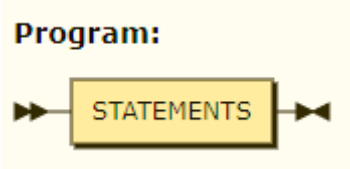

Рисунок 12 – корневой нетерминал синтаксического анализатора

Рисунок 13 – блок инструкций в фигурных скобочках

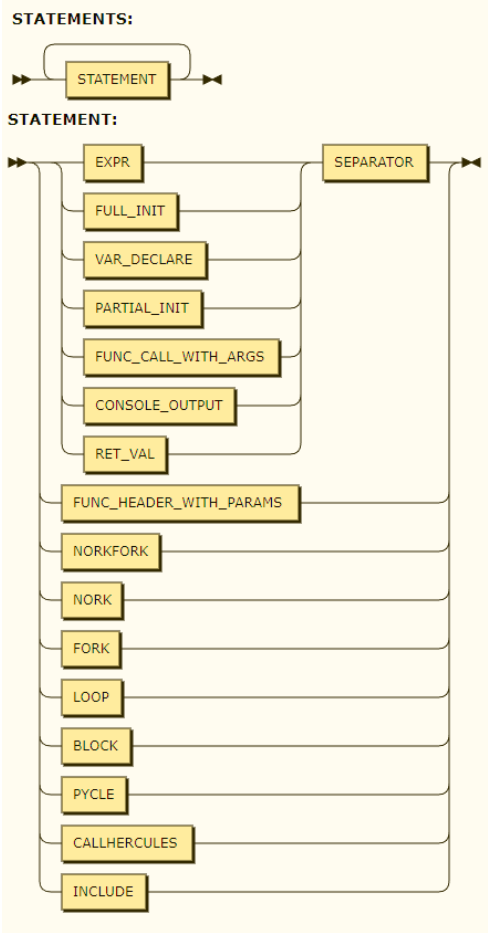

Рисунок 14 –инструкции

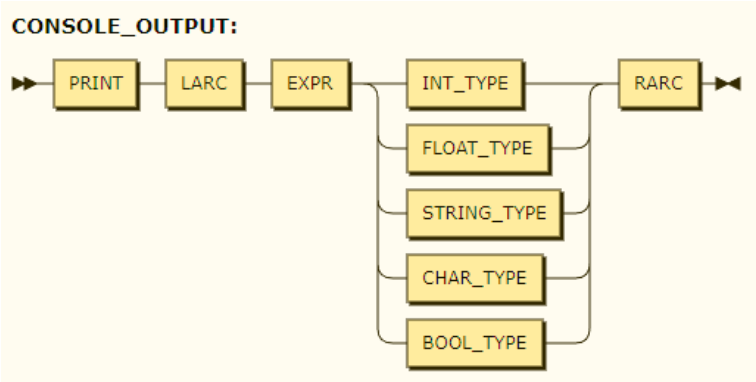

Рисунок 15 – консольный вывод

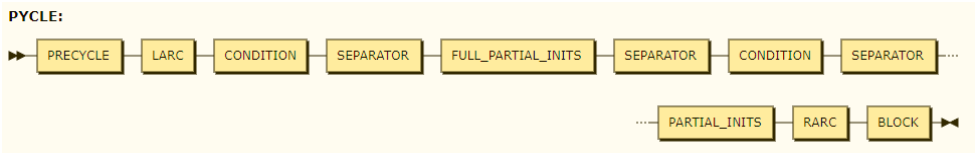

Рисунок 16 – цикл с предусловием

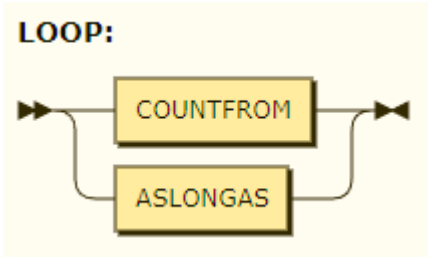

Рисунок 17 – обобщение стандартных циклов

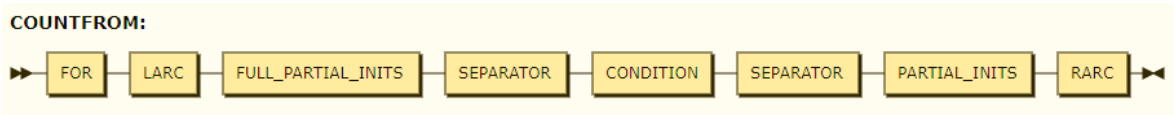

Рисунок 18 – цикл «for»

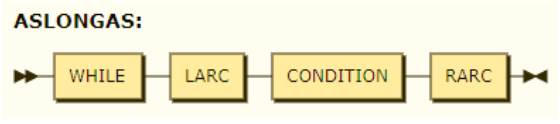

Рисунок 19 – цикл «while»

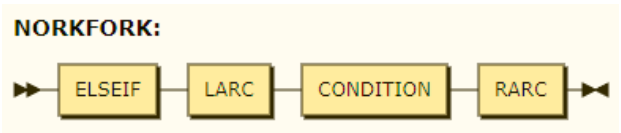

Рисунок 19 –условие «else if»

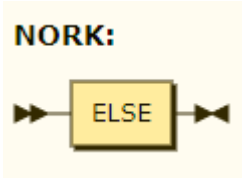

Рисунок 20 – условие «else»

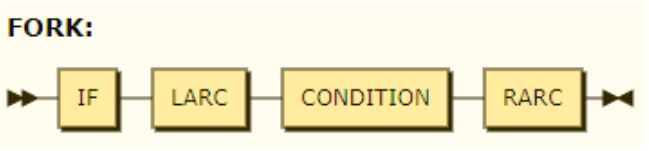

Рисунок 21 – условие «if»

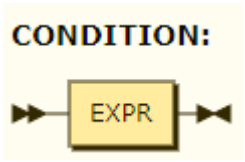

Рисунок 22 – семантическое обобщение «expr»

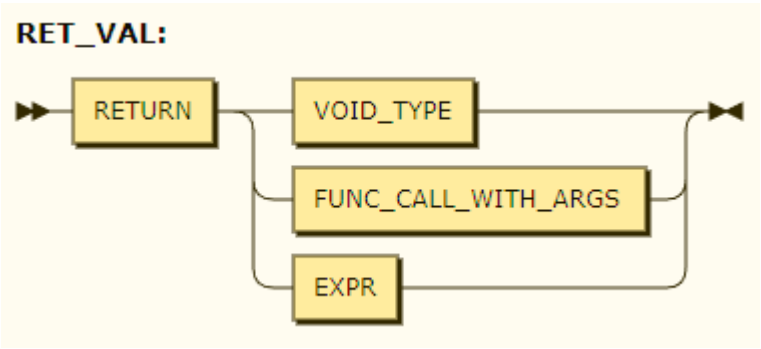

Рисунок 23 – возврат значения у функции

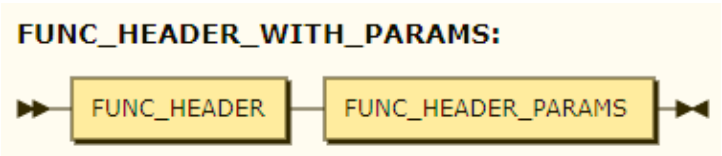

Рисунок 23 – заголовок функции с параметрами

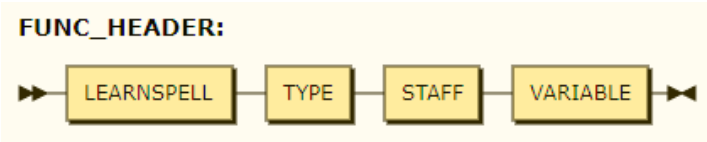

Рисунок 24 – заголовок функции

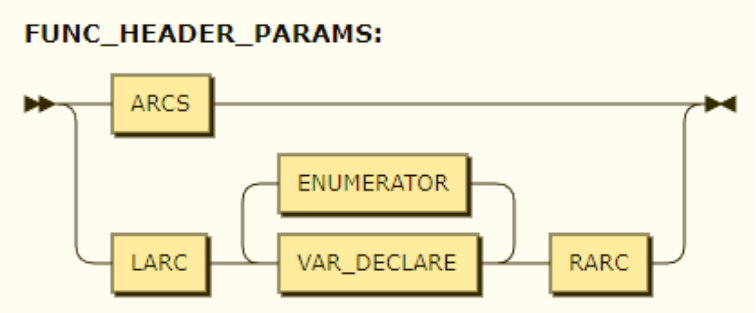

Рисунок 25 – параметры заголовка функции

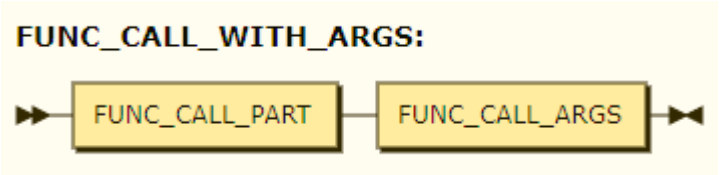

Рисунок 26 – вызов функции с аргументами

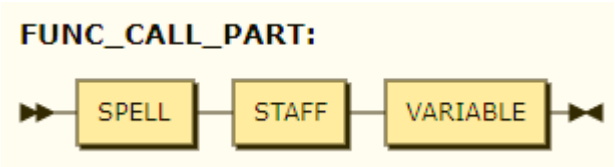

Рисунок 27 – комбинация, вызывающая функцию

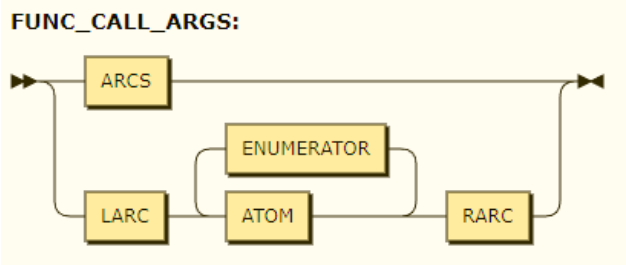

Рисунок 28 – аргументы вызова функции

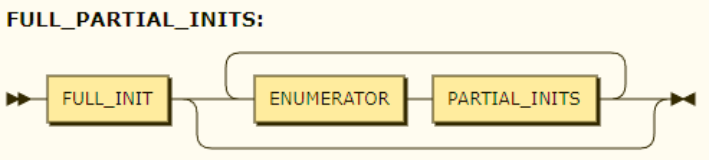

Рисунок 29 – объединение переменных в список для передачи циклам

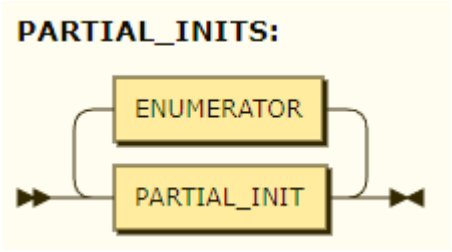

Рисунок 29 – список инициализаций переменных без объявлений

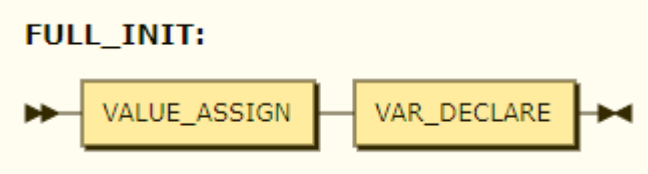

Рисунок 30 – объявление и инициализация переменной

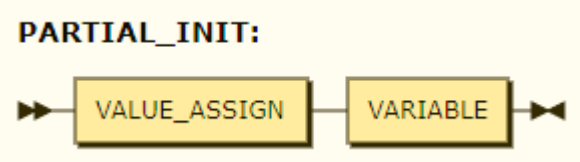

Рисунок 31 – присваивание значения переменной

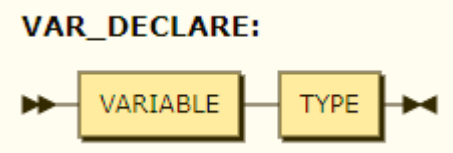

Рисунок 32 – объявление переменной

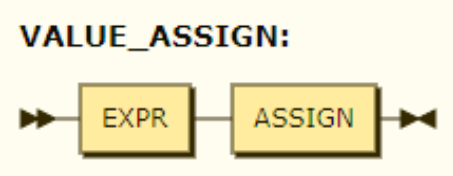

Рисунок 33 – присваивание значения переменной

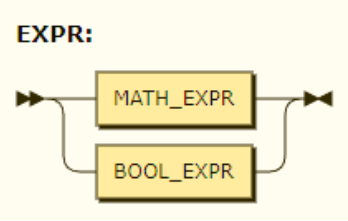

Рисунок 34 – обобщение выражений

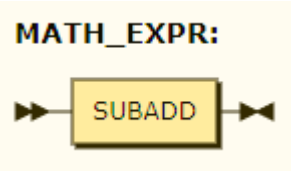

Рисунок 35 – математическое выражение

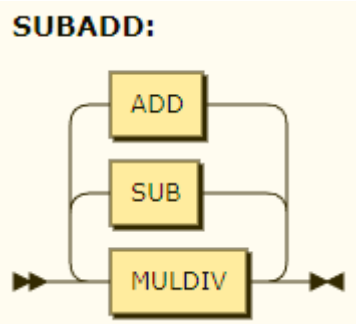

Рисунок 36 – блок обработки сложения и вычитания

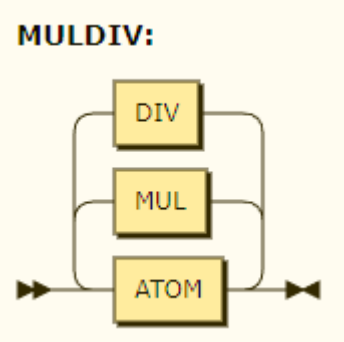

Рисунок 37 – блок обработки умножения и деления

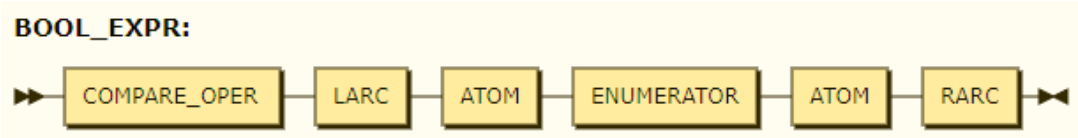

Рисунок 38 – логическое выражение

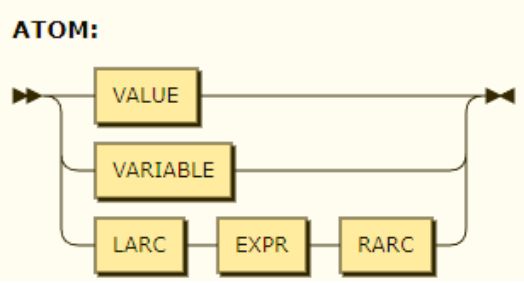

Рисунок 39 – обобщение в минимальную единицу

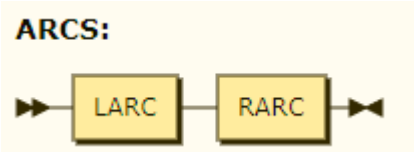

Рисунок 40 – обобщение круглых скобочек

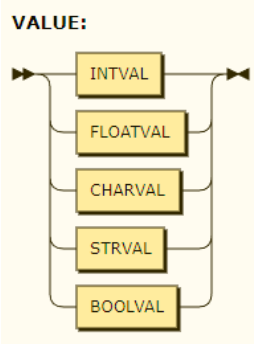

Рисунок 41 – обобщение значений

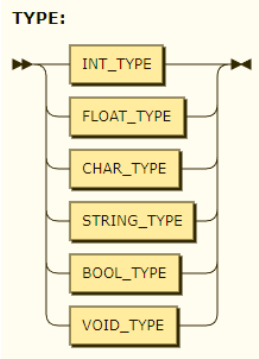

Рисунок 41 – обобщение типов данных

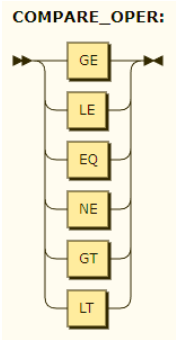

Рисунок 42 – обобщение логических операций