СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Компилятор — это программная система, которая преобразует код, написанный на языке программирования, в форму, пригодную для выполнения на компьютере [1].

Современный мир зависит от языков программирования, поскольку все программное обеспечение на компьютерах написано на том или ином языке, и компиляторы играют ключевую роль в этом процессе [1].

Целью данной работы является разработка компилятора для языка КуМир. Компилятор должен выполнять чтение текстового файла, содержащего код на языке КуМир и генерировать на выходе LLVM IR программы, пригодный для запуска.

Для достижения поставленной цели необходимо решить следующие **задачи**:

1. проанализировать грамматику языка КуМир;
2. изучить существующие средства для анализа исходного кода программы, системы генерации низкоуровневого кода;
3. реализовать прототип компиляторы;
4. провести тестирование компилятора.

1 Аналитический раздел

Компилятор — это программа, которая считывает текст программы, написанной на одном языке — исходном, и транслирует (переводит) его в эквивалентный текст на другом языке — целевом. Одна из важных ролей компилятора состоит в сообщении об ошибках в исходной программе, обнаруженных в процессе трансляции [1].

1.1 Структура компилятора

Конструктивно компилятор состоит из [2, 3]:

- фронтенда (compiler frontend), который занимается построением промежуточного представления из исходного кода и состоит из:
  - препроцессора;
  - лексического, синтаксического и семантического анализаторов;
  - генератора промежуточного представления;
- мидленд (middle-end), включающий в себя различные оптимизации;
- бэкенда (compiler backend), который занимается кодогенерацией.

На рисунке 1.1 представлена схема концептуальной структуры компилятора.

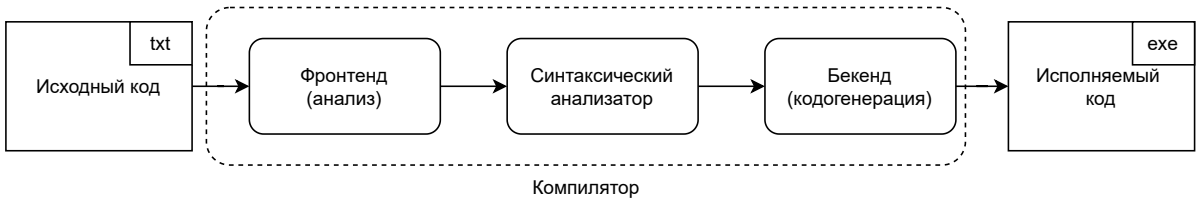

Рисунок 1.1 – Концептуальная структура компилятора

Рассмотрим работу компилятора по фазам [4]. Обобщенная структура компилятора и основные фазы компиляции показаны на рисунке 1.2.

Рисунок 1.2 – Обобщенная структура и фазы компиляции

1.1.1 Препроцессор

Иногда сборка поручается программе, который выполняет предварительную обработку перед фазой фронтенда компилятора.

Препроцессор может [1, 2]:

1. раскрывать макросы в инструкции исходного языка;

2. обрабатывать включение файлов;
3. обрабатывать языковые расширения.

1.2 Лексический анализ

На фазе лексического анализа входная программа, представляющая собой поток литер, разбивается на лексемы — слова в соответствии с определениями языка. Основными формализмами, лежащими в основе реализации лексических анализаторов, являются конечные автоматы и регулярные выражения [4].

Лексический анализатор может работать в двух основных режимах [4]:

1. как подпрограмма, вызываемая синтаксическим анализатором для получения очередной лексемы;
2. как полный проход, результатом которого является файл лексем.

В процессе выделения лексем лексический анализатор может [4]:

— самостоятельно строить таблицы объектов (идентификаторов, строк, чисел и т.д.);

— выдавать значения для каждой лексемы при обращении к ней, в этом случае таблицы объектов строятся на последующих фазах (например, при синтаксическом анализе).

На этапе лексического анализа обнаруживаются простейшие ошибки [4]:

— недопустимые символы;

— неправильная запись чисел;

— ошибки в идентификаторах.

На рисунке 1.3 представлен лексический анализатор.

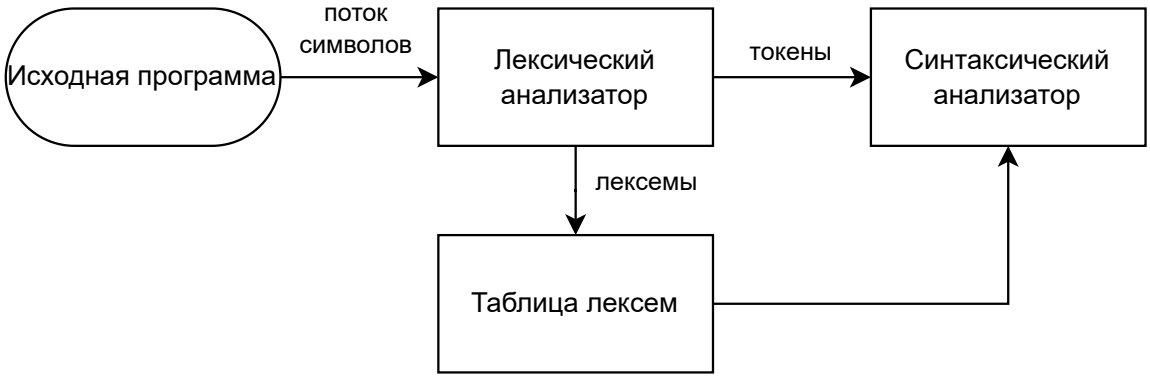

Рисунок 1.3 – Лексический анализатор

1.3 Синтаксический анализатор

Вторая фаза компилятора - синтаксический анализ или разбор (parsing) [1].

Основная задача синтаксического анализа — разбор структуры программы. Как правило, под структурой понимается дерево, соответствующее разбору в контекстно-свободной грамматике языка [4].

Результатом синтаксического анализа является синтаксическое дерево со ссылками на таблицы объектов. В процессе синтаксического анализа также обнаруживаются ошибки, связанные со структурой программы [4].

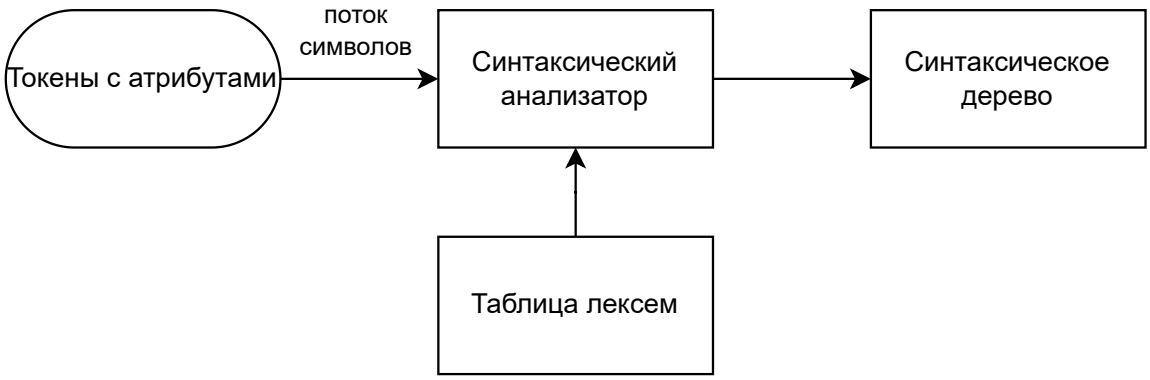

Рисунок 1.4 – Синтаксический анализатор

1.4 Семантический анализатор

Семантический анализатор использует синтаксическое дерево и информацию из таблицы символов для проверки исходной программы на семантическую согласованность с определением языка. Он также собирает информацию о типах и сохраняет ее в синтаксическом дереве или в таблице символов для последующего использования в процессе генерации промежуточного кода [1].

Важной частью семантического анализа является проверка типов, когда компилятор проверяет, имеет ли каждый оператор операнды соответствующего типа. Например, многие определения языков программирования требуют, чтобы индекс массива был целым числом; компилятор должен сообщить об ошибке, если в качестве индекса массива используется число с плавающей точкой [1].

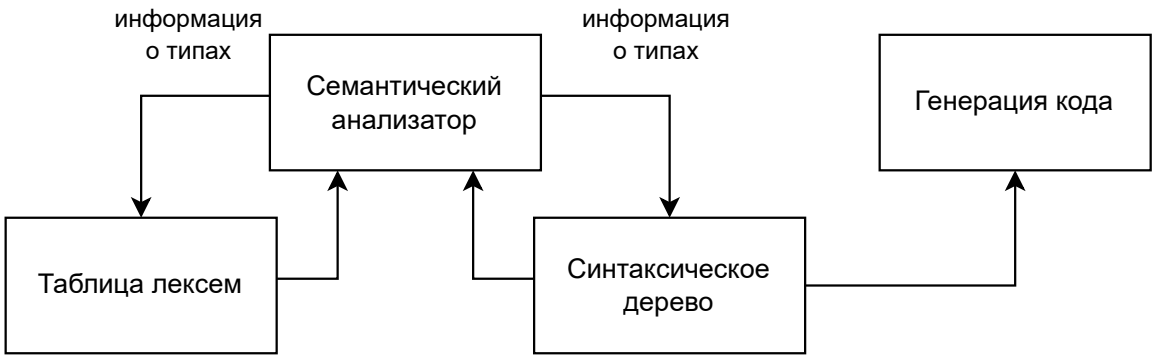

Рисунок 1.5 – Семантический анализатор

1.5 Генерация кода

В процессе трансляции исходной программы в целевой код компилятор может создавать одно или несколько промежуточных представлений различного вида. Синтаксические деревья являются видом промежуточного представления; обычно они используются в процессе синтаксического и семантического анализа [1].

После синтаксического и семантического анализа исходной программы многие компиляторы генерируют явное низкоуровневое или машинное промежуточное представление исходной программы, которое можно рассматривать как программу для абстрактной вычислительной машины. Такое промежуточное представление должно обладать двумя важными свойствами: оно должно легко генерироваться и легко транслироваться в целевой машинный язык [1].

Генератор кода получает в качестве входных данных промежуточное представление исходной программы и отображает его в целевой язык. Если целевой язык представляет собой машинный код, для каждой переменной, используемой программой, выбираются соответствующие регистры или ячейки памяти. Затем промежуточные команды транслируются в последовательности машинных команд, выполняющих те же действия. Ключевым моментом генерации кода является аккуратное распределение регистров для хранения переменных [1].

1.6 Таблица символов

Таблица символов представляет собой структуру данных, содержащую записи для каждого имени переменной, с полями для атрибутов имени. Струк-

тура данных должна быть разработана таким образом, чтобы позволять компилятору быстро находить запись для каждого имени, а также быстро сохранять данные в записи и получать их из нее [1].

Важная функция компилятора состоит в том, чтобы записывать имена переменных в исходной программе и накапливать информацию о разных атрибутах каждого имени. Эти атрибуты могут предоставлять информацию о выделенной памяти для данного имени, его типе, области видимости (где именно в программе может использоваться его значение) и, в случае имен процедур, такие сведения, как количество и типы их аргументов, метод передачи каждого аргумента (например, по значению или по ссылке), а также возвращаемый тип [1].

1.7 Синтаксическое дерево

Синтаксическое дерево — дерево, в котором каждый внутренний узел представляет операцию, а дочерние узлы — аргументы этой операции. Порядок операций в дереве согласуется с обычными правилами, например, умножение имеет более высокий приоритет, чем сложение, и должно быть выполнено до сложения [4].

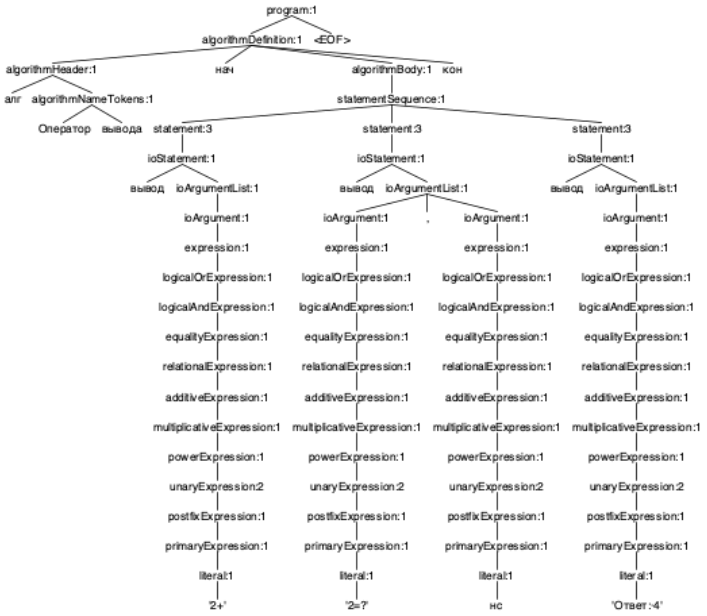

Рисунок 1.6 – Пример синтаксического дерева

1.8 Генераторы лексических анализаторов

Существует множество генераторов, наиболее популярные из них — Lex, Flex и ANTLR4. Lex — стандартный инструмент для получения лексических анализа- торов в операционных системах Unix [5]. В результате обработки входного потока получается исходный файл на языке C. Lex-файл разделяется на три блока: блок определений, правил и кода на C.

Flex заменяет Lex в системах на базе пакетов GNU и имеет аналогичную функциональность [6].

ANTLR (ANother Tool for Language Recognition) — генератор лексических и синтаксических анализаторов, позволяет создавать анализаторы на таких языках, как: Java, Go, C++ и других [7]. ANTLR генерирует классы нисходящего рекурсивного синтаксического анализатора, на основе правил, заданных грамматикой.

Он также позволяет строить и обходить деревья синтаксического ана- лиза с использованием паттернов посетитель или слушатель. Благодаря своей эффективности и простоте использования, ANTLR является одним из наиболее предпочтительных генераторов анализаторов при создании кода синтаксического анализатора. В текущей работе было решено использовать этот инструмент.

1.9 Генераторы синтаксических анализаторов

Для генерации синтаксических анализаторов применяются следующие инструменты:

- Yacc/Bison: Yacc — стандартный генератор парсеров для Unix-систем, а Bison представляет его GNU-аналог [8].
- Coco/R: Комбинированный генератор лексических и синтаксических анализаторов [9]. Лексеры реализуют конечные автоматы, а парсеры используют метод рекурсивного спуска. Поддерживает языки C, Java и другие.
- ANTLR: Универсальный инструмент (ранее упомянутый) для создания анализаторов.

Методы разбора

— Yacc/Bison: Принимают контекстно-свободную грамматику и используют LALR-разбор (LR с предпросмотром). Канонические LR-анализаторы обладают несколько большей мощностью, но требуют значительных ресурсов памяти для таблиц, что ограничивает их практическое применение.
— ANTLR: Использует расширенный LL(*)-подход с поддержкой левой рекурсии.
— Coco/R: Основан на классическом LL(1)-разборе.

Сравнение LL и LR подходов

LL(k)-анализаторы (k токенов предпросмотра) [1]:
— Строят левосторонний вывод
— Преимущества: Высокая скорость работы, простота реализации
— Недостатки: Задержки в обнаружении ошибок из-за откатов
LR-анализаторы:
— Производят правый вывод
— Преимущества: Шире охват языков, раннее обнаружение ошибок
— Недостатки: Сложность реализации, ресурсоёмкие таблицы
LR-анализ эффективнее обнаруживает синтаксические ошибки при первом несоответствии грамматике, тогда как LL(k) может задерживать диагностику в случаях с общими префиксами альтернатив.

1.10 LLVM

Проект LLVM (Low Level Virtual Machine) представляет собой программную инфраструктуру для построения компиляторов и вспомогательных утилит [10]. Ключевые компоненты:

1. LLVM IR: Платформонезависимое промежуточное представление (байт-код). Генерируется для множества архитектур (ARM, x86/x86-64, GPU AMD/Nvidia и др.).
2. Компиляция и исполнение: Преобразование IR в машинный код выполняется `clang`. Также доступен интерпретатор IR для непосредственного выполнения.

Особенности LLVM IR:

— Система типов: Поддерживает целые числа (arbitrary bitwidth), числа с плавающей точкой, массивы, структуры, функции, типизированные указатели.
— Инструкции: Преимущественно бинарные (два аргумента → один результат). Строгая статическая типизация: типы операндов и результата явно указаны и взаимосвязаны.
— Арифметика: Операнды должны совпадать по типу; операции перегружены для числовых типов и векторов.
— Преобразование типов: Требует явных инструкций приведения (включая int↔ptr и универсальную `bitcast`).
— Работа с памятью:
    — `load` (чтение), `store` (запись) — доступ по типизированному указателю.
    — `alloca` — выделение памяти на стеке (автоматическое освобождение при выходе из функции).
    — `getelementptr` — *вычисление адреса* (без доступа к памяти!) элементов структур/массивов с сохранением типизации, поддерживает произвольную вложенность и индексацию.

2 Конструкторский раздел

2.1 Концептуальная модель

Концептуальная модель разрабатываемого компилятора в нотации IDEF0 представлена на рисунке 2.1.

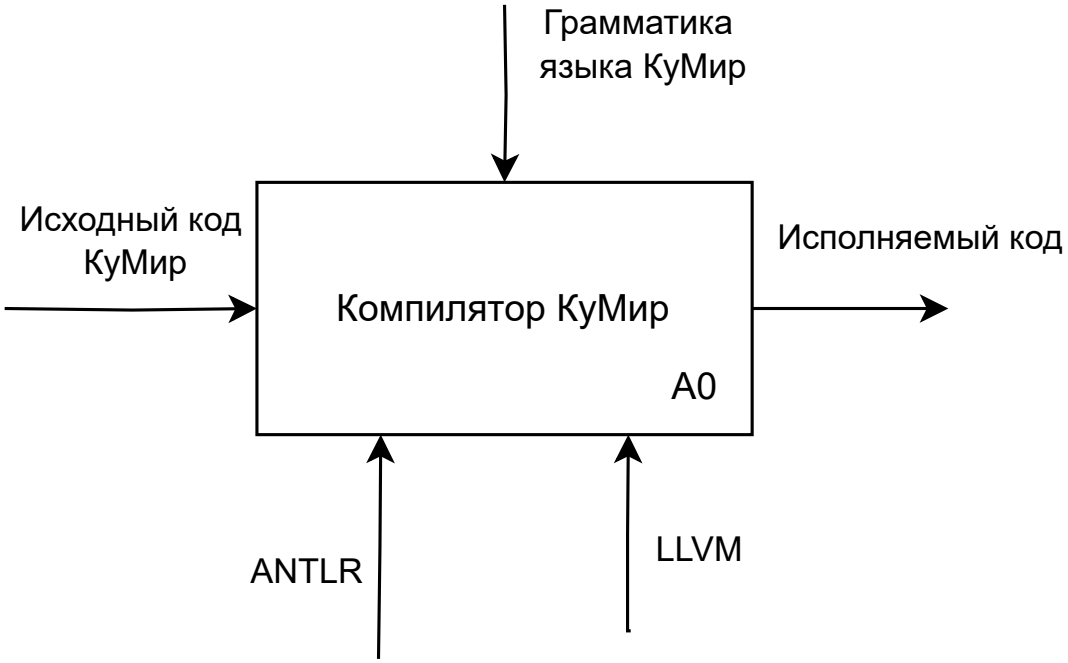

Рисунок 2.1 – Концептуальная модель разрабатываемого компилятора в нотации IDEF0

2.2 Язык КуМир

КуМир (Комплект Учебных МИРов) - система программирования, предназначенная для поддержки начальных курсов информатики и программирования в средней и высшей школе.

Грамматика языка представлена в приложении А.

2.3 Лексический и синтаксический анализ

В данной работе для генерации лексического и синтаксического анализаторов используется инструмент ANTLR4. На вход системы подаётся формальное описание грамматики языка в формате, поддерживаемом ANTLR4.

Процесс генерации включает создание:

— Классов лексера (Lexer) и парсера (Parser)
— Вспомогательных классов и файлов поддержки
— Шаблонов классов для обхода синтаксического дерева

Анализ выполняется последовательно:

1. Лексер преобразует входной поток символов (исходный код) в поток токенов
2. Парсер обрабатывает поток токенов, формируя дерево разбора (parse tree)

Ошибки, обнаруженные на этапах лексического и синтаксического анализа, выводятся в стандартный поток вывода.

2.4 Семантический анализ

Для обхода абстрактного синтаксического дерева (АСТ) доступны две стратегии:

— **Listener:** Реализует автоматический обход в глубину, активируя обработчики при входе в узел и выходе из него
— **Visitor:** Предоставляет контролируемый обход с явным указанием порядка посещения узлов через специализированные методы

В представленной реализации используется паттерн Visitor, обеспечивающий:

— Гибкое управление порядком обхода
— Возможность выборочной обработки узлов
— Реализацию специализированных методов посещения для каждого типа узла

Обход начинается с корневого узла, соответствующего точке входа программы.

3 Технологический раздел

3.1 Выбор средств программной реализации

В качестве языка реализации компилятора выбран Go. Это решение обусловлено следующими факторами.

— Кросс-платформенность: Скомпилированный компилятор может выполняться на различных операционных системах и архитектурах.
— Интеграция с LLVM: Существуют готовые библиотеки для генерации LLVM IR-кода из программ на Go.
— Поддержка инструментария: Генератор анализаторов ANTLR предоставляет возможность генерации кода на языке Go.

3.2 Основные компоненты программы

В результате работы ANTLR были сгенерированы интерфейсы BaseVisitor и BaseListener, файлы с данными для интерпретатора ANTLR и файлы с токенами и реализации анализаторов.

Был реализован интерфейс BaseVisitor, т.к. он предоставляет контролируемый обход с явным указанием порядка посещения узлов через специализированные методы вида `VisitXXX`. Пример реализации такого метода представлен в листинге 3.1.

Листинг 3.1 – Пример реализации метода BaseVisitor

```
1 TODO
```

Статическая типизация

TODO

Базовые функции языка

TODO

3.3 Тестирование

Было проведено тестирование работы компилятора для базовых конструкций КуМир в соответствии с грамматикой. Примеры программ для тестирования представлены в приложении Б.

3.4 Пример работы программы

Примеры программ на КуМир и соответствующий им результат работы компилятора на LLVM IR представлены в Приложении В.

ЗАКЛЮЧЕНИЕ

В ходе данной работы была достигнута цель: разработан компилятора язык КуМир, который выполняет чтение текстового файла, содержащего код на языке КуМир и генерирует на выходе LLVM IR программы, пригодный для запуска.

Были решены все задачи:

1. проанализирована грамматика языка КуМир;
2. изучены существующие средства для анализа исходного кода программы, системы генерации низкоуровневого кода;
3. реализован прототип компилятора;
4. проведено тестирование компилятора.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Компиляторы / *Альфред Ахо, Моника С Лам, Рави Сети [и др.]* // Принципы, технологии, инструментарий. 2003.

2. *Владимиров Константин.* Оптимизирующие компиляторы. Структура и алгоритмы. Litres, 2024.

3. Modern compiler design / *Dick Grune, Kees Van Reeuwijk, Henri E Bal [и др.].* Springer Science & Business Media, 2012.

4. *Серебряков ВА, Галочкин МП.* Основы конструирования компиляторов // М.: Эдиториал УРСС. 2001. Т. 221, № 1.

5. *Lesk Michael E, Schmidt Eric.* Lex: A lexical analyzer generator. Bell Laboratories Murray Hill, NJ, 1975. Т. 39.

6. How to test program generators? A case study using flex / *Prahladavaradan Sampath, AC Rajeev, KC Shashidhar [и др.]* // Fifth IEEE International Conference on Software Engineering and Formal Methods (SEFM 2007) / IEEE. 2007. С. 80–92.

7. *Parr Terence, Wells Peter, Klaren Ric [и др.].* What's ANTLR. 2004.

8. *Bhamidipaty Achyutram, Proebsting Todd A.* Very fast YACC-compatible parsers (for very little effort) // Software: Practice and Experience. 1998. Т. 28, № 2. С. 181–190.

9. *Mössenböck Hanspeter.* Coco/R: A generator for fast compiler front-ends // ETH, Eidgenössische Technische Hochschule Zürich, Departement Informatik, Institut für Computer Systeme. 1990. Т. 127.

10. *Sarda Suyog, Pandey Mayur.* LLVM essentials. Packt Publishing Ltd, 2015.

ПРИЛОЖЕНИЕ А

Грамматика языка КуМир

В листинге А.1 представлен лексер грамматики в формате ANTRL.

Листинг А.1 – Лексер грамматики в формате ANTRL

```
lexer grammar KumirLexer;

options { caseInsensitive = true; }

ALG_HEADER          : 'алг';
ALG_BEGIN           : 'нач';
ALG_END             : 'кон';
PRE_CONDITION       : 'дано';
POST_CONDITION      : 'надо';
ASSERTION           : 'утв';
LOOP                : 'нц';
ENDLOOP_COND        : ('кц' WS 'при' | 'кц_при');
ENDLOOP             : 'кц';
IF                  : 'если';
THEN                : 'то';
ELSE                : 'иначе';
FI                  : 'все';
SWITCH              : 'выбор';
CASE                : 'при';
INPUT               : 'ввод';
OUTPUT              : 'вывод';
ASSIGN              : ':=';
EXIT                : 'выход';
PAUSE               : 'пауза';
STOP                : 'стоп';
FOR                 : 'для';
WHILE               : 'пока';
TIMES               : 'раз';
FROM                : 'от';
TO                  : 'до';
STEP                : 'шаг';
NEWLINE_CONST       : 'нс';
NOT                 : 'не';
AND                 : 'и';
OR                  : 'или';
OUT_PARAM           : 'рез';
IN_PARAM            : 'арг';
INOUT_PARAM         : ('аргрез' | 'арг' WS 'рез' | 'арг_рез');
RETURN_VALUE        : 'знач';

INTEGER_TYPE        : 'цел';
```

```
42 REAL_TYPE            : 'вещ';
43 BOOLEAN_TYPE         : 'лог';
44 CHAR_TYPE            : 'сим';
45 STRING_TYPE          : 'лит';
46 TABLE_SUFFIX         : 'таб';
47
48 INTEGER_ARRAY_TYPE   : ('цел' WS? 'таб' | 'цел_таб');
49 REAL_ARRAY_TYPE      : ('вещ' WS? 'таб' | 'вещ_таб');
50 CHAR_ARRAY_TYPE      : ('сим' WS? 'таб' | 'сим_таб');
51 STRING_ARRAY_TYPE    : ('лит' WS? 'таб' | 'лит_таб');
52 BOOLEAN_ARRAY_TYPE   : ('лог' WS? 'таб' | 'лог_таб');
53
54 TRUE                  : 'да';
55 FALSE                 : 'нет';
56
57 POWER                 : '**';
58 GE                    : '>=';
59 LE                    : '<=';
60 NE                    : '<>';
61 PLUS                  : '+';
62 MINUS                 : '-';
63 MUL                   : '*';
64 DIV                   : '/';
65 EQ                    : '=';
66 LT                    : '<';
67 GT                    : '>';
68 LPAREN                : '(';
69 RPAREN                : ')';
70 LBRACK                : '[';
71 RBRACK                : ']';
72 LBRACE                : '{';
73 RBRACE                : '}';
74 COMMA                 : ',';
75 COLON                 : ':';
76 SEMICOLON             : ';';
77 ATAT                  : '@@';
78 AT                    : '@';
79 DIV_OP                : 'div';
80 MOD_OP                : 'mod';
81
82 CHAR_LITERAL          : '\'' (EscapeSequence | ~['\\\r\n] ) '\'';
83 STRING                : '"' (EscapeSequence | ~["\\\r\n] )*? '"'
84                       | '\'' (EscapeSequence | ~['\\\r\n] )*? '\''
85                       ;
86 REAL                  : (DIGIT+ '.' DIGIT* | '.' DIGIT+) ExpFragment?
87                       | DIGIT+ ExpFragment
88                       ;
89 INTEGER               : DecInteger | HexInteger;
```

```
90
91  ID␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣:␣LETTER␣(LETTER␣|␣DIGIT␣|␣'_'␣|␣'@')*␣;
92
93  LINE_COMMENT␣␣␣␣␣␣␣␣:␣'|'␣~[\r\n]*␣->␣channel(HIDDEN);
94  DOC_COMMENT␣␣␣␣␣␣␣␣␣:␣'#'␣~[\r\n]*␣->␣channel(HIDDEN);
95
96  WS␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣:␣[␣\t\r\n]+␣->␣skip;
97
98  fragment␣DIGIT␣␣␣␣␣␣␣:␣[0-9];
99  fragment␣HEX_DIGIT␣␣␣:␣[0-9a-fA-F];
100 fragment␣LETTER␣␣␣␣␣␣:␣[a-zA-Za-яA-ЯёЁ];
101 fragment␣DecInteger␣:␣DIGIT+;
102 fragment␣HexInteger␣:␣'$'␣HEX_DIGIT+;
103 fragment␣ExpFragment:␣[eE]␣[+-]?␣DIGIT+;
104 fragment␣EscapeSequence
105 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣:␣'\\'␣[btnfr"'\\]
106                         ;
```

В листинге А.2 представлен парсер грамматики в формате ANTRL.

Листинг А.2 – Парсер грамматики в формате ANTRL

```
1  parser grammar KumirParser;
2
3  options { tokenVocab=KumirLexer; }
4
5  qualifiedIdentifier
6      : ID
7      ;
8
9  literal
10     : INTEGER | REAL | STRING | CHAR_LITERAL | TRUE | FALSE  |
          NEWLINE_CONST
11     ;
12
13 expressionList
14     : expression (COMMA expression)*
15     ;
16
17 arrayLiteral
18     : LBRACE expressionList? RBRACE
19     ;
20
21 primaryExpression
22     : literal
23     | qualifiedIdentifier
24     | RETURN_VALUE
25     | LPAREN expression RPAREN
26     | arrayLiteral
```

```
27      ;
28
29  argumentList
30      : expression (COMMA expression)*
31      ;
32
33  indexList
34      : expression (COLON expression)? // Single index or slice
35      | expression COMMA expression    // 2D index
36      ;
37
38  postfixExpression
39      : primaryExpression ( LBRACK indexList RBRACK | LPAREN argumentList?
            RPAREN )*
40      ;
41
42  unaryExpression
43      : (PLUS | MINUS | NOT) unaryExpression | postfixExpression
44      ;
45
46  powerExpression
47      : unaryExpression (POWER powerExpression)?
48      ;
49
50  multiplicativeExpression
51      : powerExpression ((MUL | DIV | DIV_OP | MOD_OP) powerExpression)*
52      ;
53
54  additiveExpression
55      : multiplicativeExpression ((PLUS | MINUS) multiplicativeExpression)*
56      ;
57
58  relationalExpression
59      : additiveExpression ((LT | GT | LE | GE) additiveExpression)*
60      ;
61
62  equalityExpression
63      : relationalExpression ((EQ | NE) relationalExpression)*
64      ;
65
66  logicalAndExpression
67      : equalityExpression (AND equalityExpression)*
68      ;
69
70  logicalOrExpression
71      : logicalAndExpression (OR logicalAndExpression)*
72      ;
73
```

```
74 expression // Top-level expression rule
75     : logicalOrExpression
76     ;
77
78 typeSpecifier
79     : arrayType
80     | basicType TABLE_SUFFIX?
81     ;
82
83 basicType
84     : INTEGER_TYPE | REAL_TYPE | BOOLEAN_TYPE | CHAR_TYPE | STRING_TYPE
85     ;
86
87 arrayType
88     : INTEGER_ARRAY_TYPE | REAL_ARRAY_TYPE | BOOLEAN_ARRAY_TYPE |
        CHAR_ARRAY_TYPE | STRING_ARRAY_TYPE
89     ;
90
91 arrayBounds
92     : expression COLON expression
93     ;
94
95 variableDeclarationItem
96     : ID (LBRACK arrayBounds (COMMA arrayBounds)* RBRACK)? ( EQ expression
        )?
97     ;
98
99 variableList
100    : variableDeclarationItem (COMMA variableDeclarationItem)*
101    ;
102
103 variableDeclaration
104    : typeSpecifier variableList
105    ;
106
107 globalDeclaration
108    : typeSpecifier variableList SEMICOLON?
109    ;
110
111 globalAssignment
112    : qualifiedIdentifier ASSIGN (literal | unaryExpression |
        arrayLiteral) SEMICOLON?
113    ;
114
115 parameterDeclaration
116    : (IN_PARAM | OUT_PARAM | INOUT_PARAM)? typeSpecifier variableList
117    ;
118
```

```
119
120 parameterList
121     : parameterDeclaration (COMMA parameterDeclaration)*
122     ;
123
124
125 algorithmNameTokens
126     : ~(LPAREN | ALG_BEGIN | PRE_CONDITION | POST_CONDITION | SEMICOLON |
            EOF)+
127     ;
128
129 algorithmName: ID+ ;
130
131
132 algorithmHeader
133     : ALG_HEADER typeSpecifier? algorithmNameTokens (LPAREN parameterList?
            RPAREN)? SEMICOLON?
134     ;
135
136 preCondition
137     : PRE_CONDITION expression SEMICOLON?
138     ;
139
140 postCondition
141     : POST_CONDITION expression SEMICOLON?
142     ;
143
144 algorithmBody
145     : statementSequence
146     ;
147
148 statementSequence
149     : statement*
150     ;
151
152 lvalue
153     : qualifiedIdentifier (LBRACK indexList RBRACK)?
154     | RETURN_VALUE
155     ;
156
157 assignmentStatement
158     : lvalue ASSIGN expression
159     | expression
160     ;
161
162 ioArgument
163     : expression (COLON expression (COLON expression)?)? // Expression
            with optional formatting
```

```
164     | NEWLINE_CONST
165     ;
166
167 ioArgumentList
168     : ioArgument (COMMA ioArgument)*
169     ;
170
171 ioStatement
172     : (INPUT | OUTPUT) ioArgumentList
173     ;
174
175 ifStatement
176     : IF expression THEN statementSequence (ELSE statementSequence)? FI
177     ;
178
179 caseBlock
180     : CASE expression COLON statementSequence
181     ;
182
183 switchStatement
184     : SWITCH caseBlock+ (ELSE statementSequence)? FI
185     ;
186
187 endLoopCondition
188     : ENDLOOP_COND expression
189     ;
190
191 loopSpecifier
192     : FOR ID FROM expression TO expression (STEP expression)?
193     | WHILE expression
194     | expression TIMES
195     ;
196
197 loopStatement
198     : LOOP loopSpecifier? statementSequence (ENDLOOP | endLoopCondition)
199     ;
200
201 exitStatement
202     : EXIT
203     ;
204
205 pauseStatement
206     : PAUSE
207     ;
208
209 stopStatement
210     : STOP
211     ;
```

```
212
213 assertionStatement
214     : ASSERTION expression
215     ;
216
217 procedureCallStatement
218     : qualifiedIdentifier (LPAREN argumentList? RPAREN)?
219     ;
220
221 statement
222     : variableDeclaration SEMICOLON?
223     | assignmentStatement SEMICOLON?
224     | ioStatement SEMICOLON?
225     | ifStatement SEMICOLON?
226     | switchStatement SEMICOLON?
227     | loopStatement SEMICOLON?
228     | exitStatement SEMICOLON?
229     | pauseStatement SEMICOLON?
230     | stopStatement SEMICOLON?
231     | assertionStatement SEMICOLON?
232     | SEMICOLON // Allow empty statements
233     ;
234
235
236 algorithmDefinition
237     : algorithmHeader (preCondition | postCondition | variableDeclaration)*
238       ALG_BEGIN
239       algorithmBody
240       ALG_END algorithmName? SEMICOLON?
241     ;
242
243 programItem
244     : globalDeclaration
245     | globalAssignment
246     ;
247
248 program
249     : programItem* algorithmDefinition+ EOF
250     ;
```

ПРИЛОЖЕНИЕ Б

Тестовые программы

В листингах Б.2 — Б.3 представлены различные примеры кода и результаты его работы.

Листинг Б.1 – Пример программы с выводом и условными операторами

```
алг Сложные условия
нач
    цел v
    вывод 'Введите возраст: '
    ввод v
    если v >= 25 и v <= 40 то
        вывод 'подходит'
    иначе
        вывод 'не подходит'
    все
кон
```

Листинг Б.2 – Пример программы с рекурсивным вызовом функции и циклами

```
алг Простые числа с логической функцией
нач
    цел n
    вывод 'Введите число: '
    ввод n
    нц пока isPrime(n)
        вывод n,  ' - простое число', нс
        вывод 'Введите число: '
        ввод n
    кц
кон

алг лог isPrime(цел n)
нач
    цел count = 0, k
    k:= 2
    нц пока k*k <= n и count = 0
        если n mod k = 0 то
            count:=count+1
    все
    k:= k + 1
    кц
    знач:= (count = 0)
кон
```

Листинг Б.3 – Пример программы с массивом, break и циклом for

```
алг Поиск в массиве с break
нач
    цел i, N
    вывод 'Введите␣размер␣массива:␣'
    ввод N
    целтаб A[1:N]
    вывод 'Введите␣элементы␣массива:␣', нс
    нц для i от 1 до N
        ввод A[i]
    кц
    цел X, nX
    вывод 'Что␣ищем?␣'
    ввод X
    nX:= 0
    нц для i от 1 до N
        если A[i] = X то
            nX:= i
            выход
        все
    кц
    если nX > 0 то
        вывод "A[", nX, "]=", X
    иначе
        вывод "Не␣нашли!"
    все
кон
```

ПРИЛОЖЕНИЕ В

Тестовые программы с результатом на LLVM IR

В листинге В.1 — В.2 представлен пример рекурсивного вычисления числа Фибоначчи и промежуточное представление LLVM IR.

Листинг В.1 – Пример рекурсивного вычисления числа Фибоначчи

```
алг main
нач
  цел i
  нц для i от 1 до 10
    вывод фибоначчи(i), нс
  кц
кон

алг цел фибоначчи(цел n)
нач
  если n <= 2 то
    знач := 1
  иначе
    знач := фибоначчи(n - 1) + фибоначчи(n - 2)
  все
кон
```

Листинг В.2 – Пример промежуточного представления LLVM IR для вычисления числа Фибоначчи

В листинге В.3 — В.4 представлен пример вычисления числа Фибоначчи через цикл и промежуточное представление LLVM IR.

Листинг В.3 – Пример вычисления числа Фибоначчи через цикл

```
алг main
нач
  цел i
  нц для i от 1 до 10
    вывод фибоначчи(i), нс
  кц
кон

алг цел фибоначчи(цел n)
нач
  если n <= 2 то
    знач := 1
  иначе
    знач := фибоначчи(n - 1) + фибоначчи(n - 2)
```

```
15   все
16 кон
```

Листинг В.4 – Пример промежуточного представления LLVM IR для вычисления числа Фибоначчи через цикл

В листинге В.5 — В.6 представлен пример разворота массива и промежуточное представление LLVM IR.

Листинг В.5 – Пример развтора массива

```
1  алг main
2  нач
3    цел i
4    нц для i от 1 до 10
5      вывод фибоначчи(i), нс
6    кц
7  кон
8
9  алг цел фибоначчи(цел n)
10 нач
11   если n <= 2 то
12     знач := 1
13   иначе
14     знач := фибоначчи(n - 1) + фибоначчи(n - 2)
15   все
16 кон
```

Листинг В.6 – Пример промежуточного представления LLVM IR для разворота массива