Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное образовательное учреждение
высшего образования
«Московский государственный технический университет
имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К КУРСОВОЙ РАБОТЕ

НА ТЕМУ:
«Разработка компилятора языка КуМир на языке Go»

Студент ИУ7-21М _____ **К. А. Аскарян**
(Группа) (Подпись, дата) (И.О.Фамилия)

Руководитель курсовой работы _____ **А. А. Ступников**
(Подпись, дата) (И.О.Фамилия)

2025 г.

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.

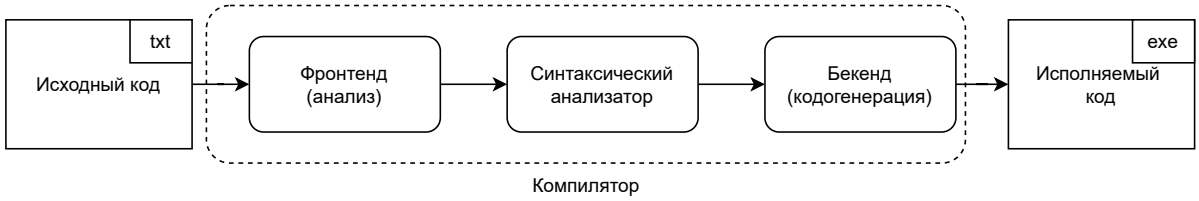

Рисунок 1.1 – Концептуальная структура компилятора

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

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.

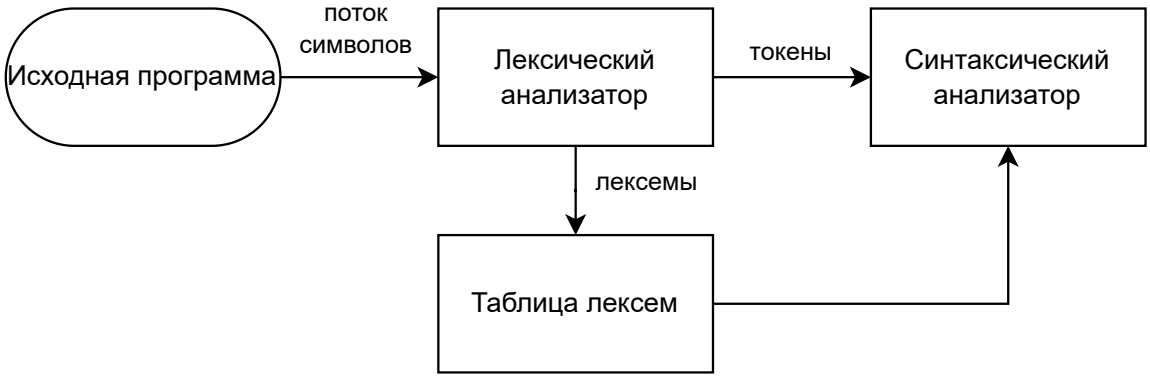

Рисунок 1.3 – Лексический анализатор

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].

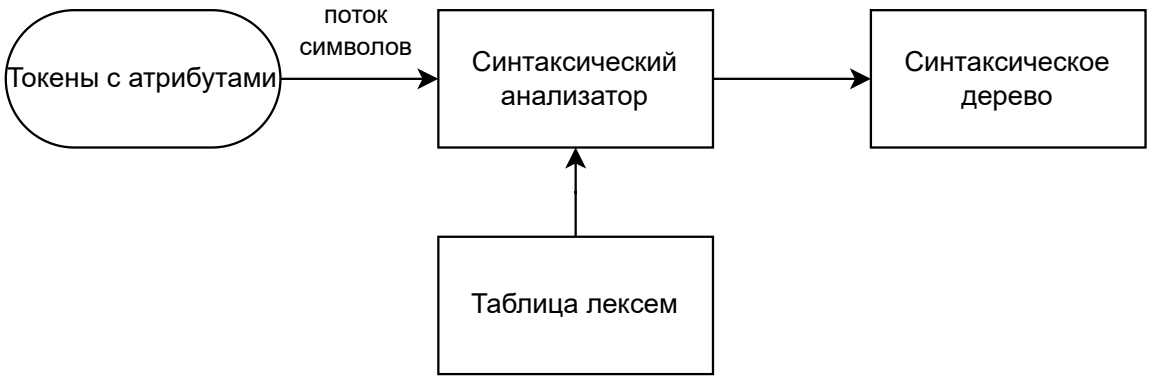

Рисунок 1.4 – Синтаксический анализатор

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].

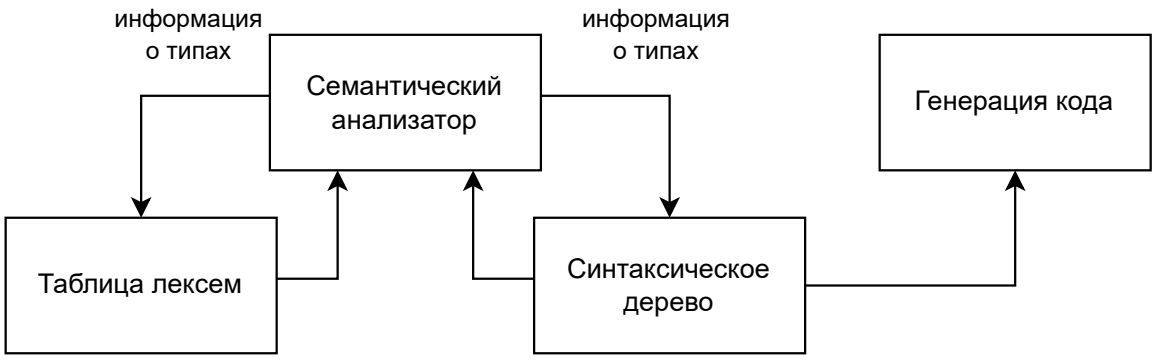

Рисунок 1.5 – Семантический анализатор

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].

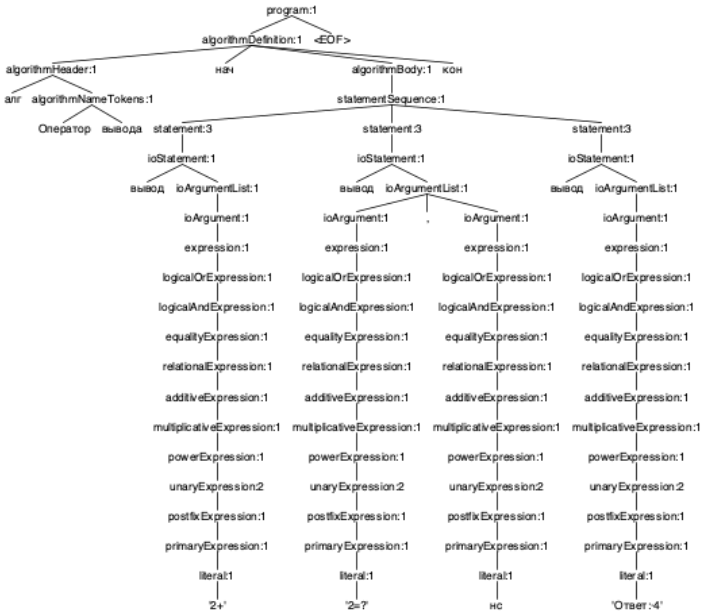

Рисунок 1.6 – Пример синтаксического дерева

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.

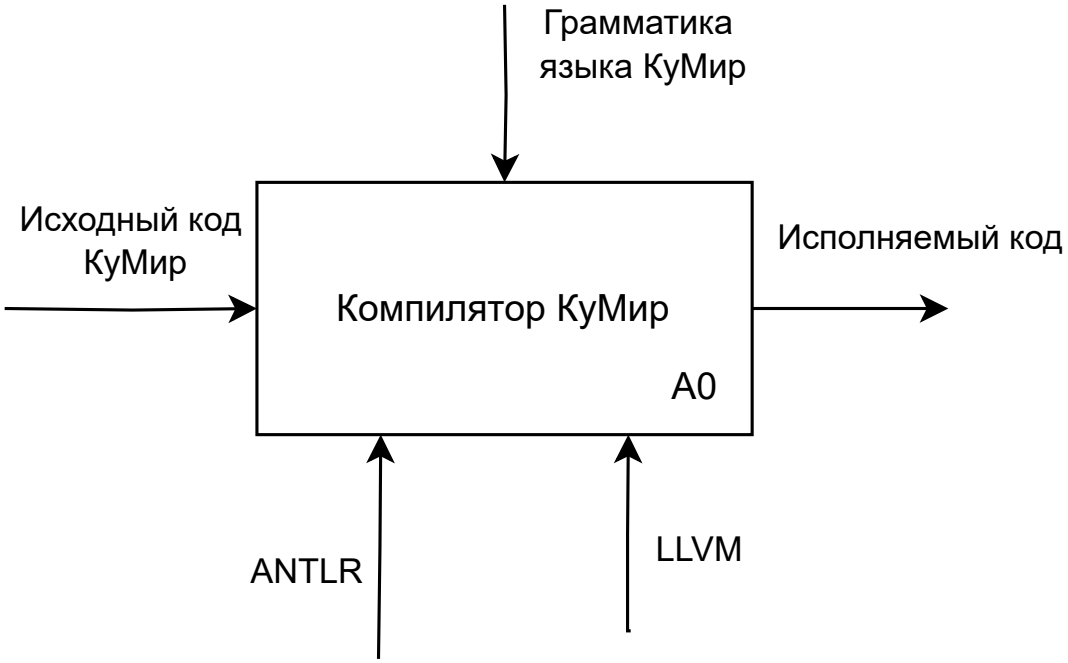

Рисунок 2.1 – Концептуальная модель разрабатываемого компилятора в нотации IDEF0

2.2 Язык КуМир

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

Система Кумир разработана в ФГУ ФНЦ НИИСИ РАН по заказу Российской Академии Наук и распространяется свободно на условиях лицензии GNU 2.0 [11].

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 VisitBasicType для определения типа скалярной переменной

```
func (v *IRVisitor) VisitBasicType(ctx *parser.BasicTypeContext)
    interface{} {
    if ctx.INTEGER_TYPE() != nil {
        return types.I32
    } else if ctx.REAL_TYPE() != nil {
        return types.Double
    } else if ctx.BOOLEAN_TYPE() != nil {
        return types.I1
    } else if ctx.CHAR_TYPE() != nil {
        return types.I8
    } else if ctx.STRING_TYPE() != nil {
        return types.NewPointer(types.I8)
    }
    v.Errors = append(v.Errors, fmt.Errorf("unknown basic type: %s",
        ctx.GetText()))
    return types.I32
}
```

Порядок функций

В КуМир нет четкой определенной функции `main`. В ее роли выступает первая встреченная функция. Функции, определенные за ней будут являться обычными функциями, которые можно будет вызвать из главной.

Однако, в `LLVM IR` выполнение должно начинать с функции `main`. Для этого используется код, представленный в листинге 3.2.

Листинг 3.2 – Определение main-функции

```
func (v *IRVisitor) declareAlgorithmPrototype(ctx
    *parser.AlgorithmHeaderContext) {
    ...
    if v.mainOriginalName == "" {
        v.mainOriginalName = llvmFuncName
        llvmFuncName = "main"
    }
    ...
}
```

В связи с тем, что объявление используемых функций узнаются позже, чем сгенерируется LLVM-представление происходит ситуация, что мы не знаем о других функциях в программе, а объявлять их раньше нельзя, т.к. они будут считаться главными. Для решения этой проблемы было принято решение делать предварительный обход по всем заголовкам функций и создавать их прототипы. Код обхода представлен в листинге 3.3

Листинг 3.3 – Обход по заголовкам функций

```
func (v *IRVisitor) VisitProgram(ctx *parser.ProgramContext) interface{} {
    ...
    for _, item := range ctx.AllAlgorithmDefinition() {
        v.declareAlgorithmPrototype(item.AlgorithmHeader().
            (*parser.AlgorithmHeaderContext))
    }
    for _, item := range ctx.AllAlgorithmDefinition() {
        v.Visit(item)
    }
    ...
}
```

Кириллические имена функций и переменных

Синтаксис языка КуМир представлен на русском языке, он позволяет в качестве идентификаторов использовать кириллические имена функций и пе-

ременных, но LLVM IR такого не допускает. Поэтому при каждом получении идентификатора он проходит обработку представленную в листинге 3.4.

Листинг 3.4 – Подготовка имени к использованию в дальнейшем коде

```
func containsNonASCII(s string) bool {
    for _, r := range s {
        if r > unicode.MaxASCII {
            return true
        }
    }
    return false
}

func NormalizeFunctionName(name string) string {
    transliterated := translit.EncodeToICAO(name)
    if containsNonASCII(name) {
        transliterated += "_rus"
    }

    var sb strings.Builder
    for i, r := range transliterated {
        if (r >= 'a' && r <= 'z') || (r >= 'A' && r <= 'Z') || (r >= '0'
           && r <= '9') || r == '_' {
            sb.WriteRune(r)
        } else if i == 0 && unicode.IsDigit(r) {
            sb.WriteRune('_')
            sb.WriteRune(r)
        } else {
            sb.WriteRune('_')
        }
    }
    return sb.String()
}
```

Код, представленный в этом листинге сначала транлитизирует кириллицу в латиницу, а после добавляет суффикс «_rus» для избежания конфликтов в ситуации наименования функций «Факт» и «Fact». Если идентификатор изначально был на латинице, то с ним ничего не происходит.

Статическая типизация

Язык КуМир строго типизирован так же, однако в нем так же, как и в *C* есть неявное приведение типов, а точнее расширение с типа `цел` до типа `вещ`, что равносильно расширению с `int` до `double` в *C*. Для реализации неявного расширения типов была реализована функция, представленная в

листинге 3.5.

Листинг 3.5 – Пример реализации метода VisitBasicType для определения типа скалярной переменной

```
func (v *IRVisitor) castToMatch(lhs, rhs value.Value) (value.Value,
    value.Value) {
    lt, rt := lhs.Type(), rhs.Type()
    if lt.Equal(rt) {
        return lhs, rhs
    }
    if _, ok := lt.(*types.IntType); ok {
        if _, isFloat := rt.(*types.FloatType); isFloat {
            lhs = v.currentBlock.NewSIToFP(lhs, rt)
            return lhs, rhs
        }
    }
    if _, ok := rt.(*types.IntType); ok {
        if _, isFloat := lt.(*types.FloatType); isFloat {
            rhs = v.currentBlock.NewSIToFP(rhs, lt)
            return lhs, rhs
        }
    }
    return lhs, rhs
}
```

Базовые функции языка

Были так же реализованы базовые функции языка, а именно:

— арифметические операции;
— логические операции;
— операции сравнения;
— условные конструкции;
— циклы;
— досрочный выход из цикла (break);
— функции и процедуры;
— функция вывода в стандартный поток ввода-вывода.

3.3 Тестирование

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

В эти примеры вошли:

— выводы различных величин;
— поиск максимума с помощью вложенных условных конструкций;
— 7 видов циклов, в том числе бесконечный и вложенный;
— программа с процедурой;
— программа с оператором выбора (switch);
— программа с выводом различных арифметических операций;
— рекурсивное вычисление факториала.

3.4 Пример работы программы

Для запуска работы программы достаточно выполнить одну из команд, представленных в листинге 3.6.

Листинг 3.6 – Пример запуска реализованного компилятора

```
Если запускать через go:
go run ./cmd/compiler/main.go -i <путь_до_файла>
Если программа скомпилирована в бинарник:
main -i <путь_до_файла> -o <путь_до_файла>
```

Кроме того, реализован следующий `CLI`-интерфейс:
— флаг `-i` указывает путь до файла с исходным кодом на языке КуМир, по умолчанию равен «./example/2+2.kum»;
— флаг `-o` указывает на путь скомпилированного исполняемого файла, по умолчанию равен «./example/out»;
— флаг `-d` позволяет сгенерировать AST-дерево для указанного файла с исходным кодом, по умолчанию равен `False`.

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

К этим примерам относятся:
— рекурсивное вычисление N-го числа Фибоначчи;
— циклическое вычисление N-го числа Фибоначчи;
— реверс массива.

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.

11. Система программирования КуМир [Электронный ресурс]. Режим доступа — URL: `https://www.niisi.ru/kumir/index.htm` (дата обращения: 10.05.2025).

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';
LOOP                : 'нц';
ENDLOOP_COND        : ('кц' WS 'при' | 'кц_при');
ENDLOOP             : 'кц';
IF                  : 'если';
THEN                : 'то';
ELSE                : 'иначе';
FI                  : 'все';
SWITCH              : 'выбор';
CASE                : 'при';
OUTPUT              : 'вывод';
ASSIGN              : ':=';
EXIT                : 'выход';
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';
REAL_TYPE           : 'вещ';
BOOLEAN_TYPE        : 'лог';
CHAR_TYPE           : 'сим';
STRING_TYPE         : 'лит';
TABLE_SUFFIX        : 'таб';
INTEGER_ARRAY_TYPE  : ('цел' WS? 'таб' | 'цел_таб');
```

```
REAL_ARRAY_TYPE     : ('вещ' WS? 'таб' | 'вещ_таб');
CHAR_ARRAY_TYPE     : ('сим' WS? 'таб' | 'сим_таб');
STRING_ARRAY_TYPE   : ('лит' WS? 'таб' | 'лит_таб');
BOOLEAN_ARRAY_TYPE  : ('лог' WS? 'таб' | 'лог_таб');

TRUE                : 'да';
FALSE               : 'нет';
POWER               : '**';
GE                  : '>=';
LE                  : '<=';
NE                  : '<>';
PLUS                : '+';
MINUS               : '-';
MUL                 : '*';
DIV                 : '/';
EQ                  : '=';
LT                  : '<';
GT                  : '>';
LPAREN              : '(';
RPAREN              : ')';
LBRACK              : '[';
RBRACK              : ']';
LBRACE              : '{';
RBRACE              : '}';
COMMA               : ',';
COLON               : ':';
SEMICOLON           : ';';
DIV_OP              : 'div';
MOD_OP              : 'mod';

CHAR_LITERAL        : '\'' (EscapeSequence | ~['\\\r\n] ) '\'' ;
STRING              : '"' (EscapeSequence | ~["\\\r\n] )*? '"'
                    | '\'' (EscapeSequence | ~['\\\r\n] )*? '\''
                    ;
REAL                : (DIGIT+ '.' DIGIT* | '.' DIGIT+) ExpFragment?
                    | DIGIT+ ExpFragment
                    ;
INTEGER             : DecInteger | HexInteger ;

ID                  : LETTER (LETTER | DIGIT | '_' | '@')* ;

LINE_COMMENT        : '|' ~[\r\n]* -> channel(HIDDEN);
DOC_COMMENT         : '#' ~[\r\n]* -> channel(HIDDEN);

WS                  : [ \t\r\n]+ -> skip;

fragment DIGIT      : [0-9];
fragment HEX_DIGIT  : [0-9a-fA-F];
```

```
fragment␣LETTER␣␣␣␣␣:␣[a-zA-Za-яA-ЯёЁ];
fragment␣DecInteger␣:␣DIGIT+;
fragment␣HexInteger␣:␣'$'␣HEX_DIGIT+;
fragment␣ExpFragment:␣[eE]␣[+-]?␣DIGIT+;
fragment␣EscapeSequence
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣:␣'\\'␣[btnfr"'\\]
                       ;
```

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

Листинг А.2 – Парсер грамматики в формате ANTRL

```
parser grammar KumirParser;

options { tokenVocab=KumirLexer; } // Use tokens from KumirLexer.g4


qualifiedIdentifier
    : ID // Currently simple ID, can be extended for module.member later
    ;

literal
    : INTEGER | REAL | STRING | CHAR_LITERAL | TRUE | FALSE  |
       NEWLINE_CONST
    ;

expressionList
    : expression (COMMA expression)*
    ;

arrayLiteral
    : LBRACE expressionList? RBRACE
    ;

primaryExpression
    : literal
    | qualifiedIdentifier
    | RETURN_VALUE
    | LPAREN expression RPAREN
    | arrayLiteral
    ;

argumentList
    : expression (COMMA expression)*
    ;

indexList
    : expression (COLON expression)?
    | expression COMMA expression
```

```
37      ;
38
39  postfixExpression
40      : primaryExpression ( LBRACK indexList RBRACK | LPAREN argumentList?
            RPAREN )*
41      ;
42
43  unaryExpression
44      : op=(PLUS | MINUS | NOT) unaryExpression | postfixExpression
45      ;
46
47  powerExpression
48      : unaryExpression (POWER powerExpression)?
49      ;
50
51  multiplicativeExpression
52      : powerExpression (op=(MUL | DIV | DIV_OP | MOD_OP) powerExpression)*
53      ;
54
55  additiveExpression
56      : multiplicativeExpression (op=(PLUS | MINUS)
            multiplicativeExpression)*
57      ;
58
59  relationalExpression
60      : additiveExpression (op=(LT | GT | LE | GE) additiveExpression)*
61      ;
62
63  equalityExpression
64      : relationalExpression (op=(EQ | NE) relationalExpression)*
65      ;
66
67  logicalAndExpression
68      : equalityExpression (AND equalityExpression)*
69      ;
70
71  logicalOrExpression
72      : logicalAndExpression (OR logicalAndExpression)*
73      ;
74
75  expression
76      : logicalOrExpression
77      ;
78
79  typeSpecifier
80      : arrayType
81      | basicType TABLE_SUFFIX?
82      ;
```

```
83
84 basicType
85     : INTEGER_TYPE | REAL_TYPE | BOOLEAN_TYPE | CHAR_TYPE | STRING_TYPE
86     ;
87
88 arrayType
89     : INTEGER_ARRAY_TYPE | REAL_ARRAY_TYPE | BOOLEAN_ARRAY_TYPE |
          CHAR_ARRAY_TYPE | STRING_ARRAY_TYPE
90     ;
91
92 arrayBounds
93     : expression COLON expression
94     ;
95
96 variableDeclarationItem
97     : ID (LBRACK arrayBounds (COMMA arrayBounds)* RBRACK)? ( EQ expression
          )?
98     ;
99
100 variableList
101     : variableDeclarationItem (COMMA variableDeclarationItem)*
102     ;
103
104 variableDeclaration
105     : typeSpecifier variableList
106     ;
107
108 globalDeclaration
109     : typeSpecifier variableList SEMICOLON?
110     ;
111
112 globalAssignment
113     : qualifiedIdentifier ASSIGN (literal | unaryExpression |
          arrayLiteral) SEMICOLON?
114     ;
115
116
117 parameterDeclaration
118     : (IN_PARAM | OUT_PARAM | INOUT_PARAM)? typeSpecifier variableList
119     ;
120
121 parameterList
122     : parameterDeclaration (COMMA parameterDeclaration)*
123     ;
124
125 algorithmNameTokens
126     : ~(LPAREN | ALG_BEGIN | SEMICOLON | EOF)+
127     ;
```

```
128
129 algorithmName: ID+ ;
130
131 algorithmHeader
132     : ALG_HEADER typeSpecifier? algorithmNameTokens (LPAREN parameterList?
        RPAREN)? SEMICOLON?
133     ;
134
135 algorithmBody
136     : statementSequence
137     ;
138
139 statementSequence
140     : statement*
141     ;
142
143 lvalue
144     : qualifiedIdentifier (LBRACK indexList RBRACK)?
145     | RETURN_VALUE
146     ;
147
148 assignmentStatement
149     : lvalue ASSIGN expression
150     | expression
151     ;
152
153 ioArgument
154     : expression (COLON expression (COLON expression)?)?
155     | NEWLINE_CONST
156     ;
157
158 ioArgumentList
159     : ioArgument (COMMA ioArgument)*
160     ;
161
162 ioStatement
163     : (OUTPUT) ioArgumentList
164     ;
165
166 ifStatement
167     : IF expression THEN statementSequence (ELSE statementSequence)? FI
168     ;
169
170 caseBlock
171     : CASE expression COLON statementSequence
172     ;
173
174 switchStatement
```

```
175     : SWITCH caseBlock+ (ELSE statementSequence)? FI
176     ;
177
178 endLoopCondition
179     : ENDLOOP_COND expression
180     ;
181
182 loopSpecifier
183     : FOR ID FROM expression TO expression (STEP expression)?
184     | WHILE expression
185     | expression TIMES
186     ;
187
188 loopStatement
189     : LOOP loopSpecifier? statementSequence (ENDLOOP | endLoopCondition)
190     ;
191
192 exitStatement
193     : EXIT
194     ;
195
196 statement
197     : variableDeclaration SEMICOLON?
198     | assignmentStatement SEMICOLON?
199     | ioStatement SEMICOLON?
200     | ifStatement SEMICOLON?
201     | switchStatement SEMICOLON?
202     | loopStatement SEMICOLON?
203     | exitStatement SEMICOLON?
204     | SEMICOLON // Allow empty statements
205     ;
206
207 algorithmDefinition
208     : algorithmHeader (variableDeclaration)*
209       ALG_BEGIN
210       algorithmBody
211       ALG_END algorithmName? SEMICOLON?
212     ;
213
214 programItem
215     : globalDeclaration
216     | globalAssignment
217     ;
218
219 program
220     : programItem* algorithmDefinition+ EOF
221     ;
```

ПРИЛОЖЕНИЕ Б

Тестовые программы

В листингах Б.1 — Б.14 представлены различные примеры кода и результаты его работы.

Листинг Б.1 – Пример программы с различными выводами

```
алг Оператор вывода
нач
    вывод '2+'
    вывод '2=?', нс
    вывод 'Ответ: 4', нс
    вывод 3.14, 10
кон
```

Результат программы:

Листинг Б.2 – Результат программы с различными выводами

```
2+2=?
Ответ: 4
3.140000 10
```

Листинг Б.3 – Пример программы условиями

```
алг Максимум
нач
    цел x, y, z, max
    x := 1
    y := 2
    z := 3
    если x > y то
        если x > z то
            max := x
        иначе
            max := z
        все
    иначе
        если y > z то
            max := y
        иначе
            max := z
        все
    все

    вывод "Максимальное число = ", max, нс
кон
```

Листинг Б.4 – Результат программы условиями

```
Максимальное число =  3
```

Листинг Б.5 – Пример программы с различными видами циклов

```
алг ПроверкаЦиклов
нач
    цел i
    вывод 'Цикл␣for␣от␣1␣до␣3', нс
    нц для i от 1 до 3
        вывод 'для:␣i␣=␣', i, нс
    кц

    вывод нс
    вывод 'Цикл␣for␣от␣1␣до␣5,␣шаг␣2', нс
    нц для i от 1 до 5 шаг 2
        вывод 'для:␣i␣=␣', i, нс
    кц

    вывод нс
    вывод 'Цикл␣while␣i␣<␣3', нс
    i := 0
    нц пока i < 3
        вывод 'пока:␣i␣=␣', i, нс
        i := i + 1
    кц

    вывод нс
    вывод 'цикл␣N␣раз', нс
    i := 0
    нц 3 раз
        вывод 'н-раз:␣i␣=␣', i, нс
        i := i + 1
    кц

    вывод нс
    вывод 'цикл␣до␣тех␣пор␣i␣<␣3', нс
    i := 0
    нц
        вывод 'н-раз:␣i␣=␣', i, нс
        i := i + 1
    кц_при i > 3
```

Листинг Б.5 – Пример программы с различными видами циклов

```
38      вывод нс
39      вывод 'Вложенные␣циклы␣for␣от␣1␣до␣3␣и␣от␣1␣до␣4', нс
40      нц для i от 1 до 3
41          нц для j от 1 до 3
42                  вывод 'i␣=␣', i, 'j␣=␣', j, нс
43          кц
44      кц
45
46      i := 0
47      вывод 'Бесконечный␣цикл␣с␣break', нс
48      нц
49          вывод 'i␣=', i, нс
50          если i > 5 то
51              выход
52          иначе
53              i := i + 1
54          все
55      кц
56  кон
```

Листинг Б.6 – Результат программы с различными видами циклов

```
1  Цикл for от 1 до 3
2  для: i =  1
3  для: i =  2
4  для: i =  3
5
6  Цикл for от 1 до 5, шаг 2
7  для: i =  1
8  для: i =  3
9  для: i =  5
10
11 Цикл while i < 3
12 пока: i =  0
13 пока: i =  1
14 пока: i =  2
15
16 цикл N раз
17 н-раз: i =  0
18 н-раз: i =  1
19 н-раз: i =  2
20
21 цикл до тех пор i < 3
22 н-раз: i =  0
23 н-раз: i =  1
24 н-раз: i =  2
25 н-раз: i =  3
```

Листинг Б.6 – Результат программы с различными видами циклов

```
Вложенные циклы for от 1 до 3 и от 1 до 4
i =  1 j =  1
i =  1 j =  2
i =  1 j =  3
i =  2 j =  1
i =  2 j =  2
i =  2 j =  3
i =  3 j =  1
i =  3 j =  2
i =  3 j =  3
Бесконечный цикл с break
i = 0
i = 1
i = 2
i = 3
i = 4
i = 5
i = 6
```

Листинг Б.7 – Пример программы с процедурой

```
алг Программа с процедурой
нач
    цел n
    n := -1
    если n < 0 то Error
    иначе вывод 'Нет␣ошибки', нс
    все
кон

алг Error
нач
    вывод 'Ошибка␣программы', нс
кон
```

Листинг Б.8 – Результат программы с процедурой

```
Ошибка программы
```

Листинг Б.9 – Пример программы с оператором выбора

```
алг Оператор выбора
нач
    вещ x
    цел sgn
    x := 1.5
    выбор
        при x < 0: sgn:= -1
        при x = 0: sgn:= 0
        при x > 0: sgn:= 1
    все
    вывод sgn, нс
кон
```

Листинг Б.10 – Результат программы с оператором выбора

```
1
```

Листинг Б.11 – Пример программы с выводом различных операций

```
алг Оператор вывода
нач
    цел a
    a := 1 + 2 - 3 + 4 * 5 * 7 * 9
    вывод a, a, 10+5, 10 / 5, 1 < 2, да, нет, нс
кон
```

Листинг Б.12 – Результат программы с выводом различных операций

```
1260 1260 15 2 да да нет
```

Листинг Б.13 – Пример программы с рекурсией

```
алг Факториал
нач
    цел N
    N := 3
    вывод Fact(N), нс
кон

алг цел Fact(цел N)
нач
    вывод '->␣N=', N, нс
    если N <= 1 то
        знач:= 1
    иначе знач:= N * Fact(N - 1)
    все
    вывод '<-␣N=', N, нс
кон
```

Листинг Б.14 – Результат программы с рекурсией

```
-> N= 3
-> N= 2
-> N= 1
<- N= 1
<- N= 2
<- N= 3
6
```

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 a
    a := фибоначчи(5)
    вывод a, нс
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 рекурсивного вычисления числа Фибоначчи

```
target triple = "arm64-apple-macosx14.0.0"

@.str.literal.0 = constant [2 x i8] c"\0A\00"
@.fmt.str.1 = constant [6 x i8] c"%d␣%s\00"

define i32 @main() {
entry:
    %0 = alloca i32
    %1 = alloca i32
    %2 = call i32 @fibonachchi_rus(i32 5)
    store i32 %2, i32* %1
    %3 = load i32, i32* %1
    %4 = getelementptr [2 x i8], [2 x i8]* @.str.literal.0, i32 0, i32 0
    %5 = getelementptr [6 x i8], [6 x i8]* @.fmt.str.1, i32 0, i32 0
    %6 = call i32 (i8*, ...) @printf(i8* %5, i32 %3, i8* %4)
    %7 = load i32, i32* %0
    ret i32 %7
}
```

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

```
define i32 @fibonachchi_rus(i32 %n) {
entry:
    %0 = alloca i32
    %1 = alloca i32
    store i32 %n, i32* %1
    %2 = load i32, i32* %1
    %3 = icmp sle i32 %2, 2
    br i1 %3, label %if.then.1, label %if.else.3

if.then.1:
    store i32 1, i32* %0
    br label %if.end.2

if.end.2:
    %4 = load i32, i32* %0
    ret i32 %4

if.else.3:
    %5 = load i32, i32* %1
    %6 = sub i32 %5, 1
    %7 = call i32 @fibonachchi_rus(i32 %6)
    %8 = load i32, i32* %1
    %9 = sub i32 %8, 2
    %10 = call i32 @fibonachchi_rus(i32 %9)
    %11 = add i32 %7, %10
    store i32 %11, i32* %0
    br label %if.end.2
}

declare i32 @printf(i8* %format, ...)
```

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 FibIter
нач
    цел n
    цел a, b, i
    n := 5
    вывод 'n␣=', n, нс
    a := 0
    b := 1
```

Листинг В.3 – Пример вычисления числа Фибоначчи через цикл

```
    если n = 0 то
        вывод 'F(', n, ')␣=', a, нс
        выход
    иначе
        если n = 1 то
            вывод 'F(', n, ')␣=', b, нс
            выход
        все
    все
    нц для i от 2 до n
        b := a + b
        a := b - a
    кц
    вывод 'F(', n, ')␣=', b, нс
кон
```

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

```
target triple = "arm64-apple-macosx14.0.0"

@.str.literal.0 = constant [4 x i8] c"n␣=\00"
@.str.literal.1 = constant [2 x i8] c"\0A\00"
@.fmt.str.2 = constant [9 x i8] c"%s␣%d␣%s\00"
@.str.literal.3 = constant [3 x i8] c"F(\00"
@.str.literal.4 = constant [4 x i8] c")␣=\00"
@.str.literal.5 = constant [2 x i8] c"\0A\00"
@.fmt.str.6 = constant [15 x i8] c"%s␣%d␣%s␣%d␣%s\00"
@.str.literal.7 = constant [3 x i8] c"F(\00"
@.str.literal.8 = constant [4 x i8] c")␣=\00"
@.str.literal.9 = constant [2 x i8] c"\0A\00"
@.fmt.str.10 = constant [15 x i8] c"%s␣%d␣%s␣%d␣%s\00"
@.str.literal.11 = constant [3 x i8] c"F(\00"
@.str.literal.12 = constant [4 x i8] c")␣=\00"
@.str.literal.13 = constant [2 x i8] c"\0A\00"
@.fmt.str.14 = constant [15 x i8] c"%s␣%d␣%s␣%d␣%s\00"

define i32 @main() {
entry:
    %0 = alloca i32
    %1 = alloca i32
    %2 = alloca i32
    %3 = alloca i32
    %4 = alloca i32
    store i32 5, i32* %1
    %5 = getelementptr [4 x i8], [4 x i8]* @.str.literal.0, i32 0, i32 0
    %6 = load i32, i32* %1
```

Листинг B.3 – Пример промежуточного представления LLVM IR для вычисления числа Фибоначчи через цикл

```
    %7 = getelementptr [2 x i8], [2 x i8]* @.str.literal.1, i32 0, i32 0
    %8 = getelementptr [9 x i8], [9 x i8]* @.fmt.str.2, i32 0, i32 0
    %9 = call i32 (i8*, ...) @printf(i8* %8, i8* %5, i32 %6, i8* %7)
    store i32 0, i32* %2
    store i32 1, i32* %3
    %10 = load i32, i32* %1
    %11 = icmp eq i32 %10, 0
    br i1 %11, label %if.then.1, label %if.else.3

if.then.1:
    %12 = getelementptr [3 x i8], [3 x i8]* @.str.literal.3, i32 0, i32 0
    %13 = load i32, i32* %1
    %14 = getelementptr [4 x i8], [4 x i8]* @.str.literal.4, i32 0, i32 0
    %15 = load i32, i32* %2
    %16 = getelementptr [2 x i8], [2 x i8]* @.str.literal.5, i32 0, i32 0
    %17 = getelementptr [15 x i8], [15 x i8]* @.fmt.str.6, i32 0, i32 0
    %18 = call i32 (i8*, ...) @printf(i8* %17, i8* %12, i32 %13, i8* %14,
        i32 %15, i8* %16)
    ret i32 0

if.end.2:
    %19 = load i32, i32* %1
    %20 = alloca i32
    store i32 2, i32* %20
    br label %loop.cond.6

if.else.3:
    %21 = load i32, i32* %1
    %22 = icmp eq i32 %21, 1
    br i1 %22, label %if.then.4, label %if.end.5

if.then.4:
    %23 = getelementptr [3 x i8], [3 x i8]* @.str.literal.7, i32 0, i32 0
    %24 = load i32, i32* %1
    %25 = getelementptr [4 x i8], [4 x i8]* @.str.literal.8, i32 0, i32 0
    %26 = load i32, i32* %3
    %27 = getelementptr [2 x i8], [2 x i8]* @.str.literal.9, i32 0, i32 0
    %28 = getelementptr [15 x i8], [15 x i8]* @.fmt.str.10, i32 0, i32 0
    %29 = call i32 (i8*, ...) @printf(i8* %28, i8* %23, i32 %24, i8* %25,
        i32 %26, i8* %27)
    ret i32 0

if.end.5:
    br label %if.end.2

loop.cond.6:
```

```
73      %30 = load i32, i32* %20
74      %31 = icmp sle i32 %30, %19
75      br i1 %31, label %loop.body.7, label %loop.end.9
76
77  loop.body.7:
78      %32 = load i32, i32* %2
79      %33 = load i32, i32* %3
80      %34 = add i32 %32, %33
81      store i32 %34, i32* %3
82      %35 = load i32, i32* %3
83      %36 = load i32, i32* %2
84      %37 = sub i32 %35, %36
85      store i32 %37, i32* %2
86      br label %loop.step.8
87
88  loop.step.8:
89      %38 = load i32, i32* %20
90      %39 = add i32 %38, 1
91      store i32 %39, i32* %20
92      br label %loop.cond.6
93
94  loop.end.9:
95      %40 = getelementptr [3 x i8], [3 x i8]* @.str.literal.11, i32 0, i32 0
96      %41 = load i32, i32* %1
97      %42 = getelementptr [4 x i8], [4 x i8]* @.str.literal.12, i32 0, i32 0
98      %43 = load i32, i32* %3
99      %44 = getelementptr [2 x i8], [2 x i8]* @.str.literal.13, i32 0, i32 0
100     %45 = getelementptr [15 x i8], [15 x i8]* @.fmt.str.14, i32 0, i32 0
101     %46 = call i32 (i8*, ...) @printf(i8* %45, i8* %40, i32 %41, i8* %42,
            i32 %43, i8* %44)
102     %47 = load i32, i32* %0
103     ret i32 %47
104 }
105
106 declare i32 @printf(i8* %format, ...)
```

В листинге В.4 — В.5 представлен пример разворота массива и промежуточное представление LLVM IR.

Листинг В.4 – Пример программы для развтора массива

```
1  алг Объявление массивов
2  нач
3      целтаб A[1:5]
4      цел N
5      N := 5
6      A[1] := 1
7      A[2] := 2
8      A[3] := 3
```

Листинг В.4 – Пример программы для развтора массива

```
    A[4] := 4
    A[5] := 5

    Реверс(A, N)
    вывод 'После реверса', нс
    ВыводМассива(A, N)
кон

алг ВыводМассива(целта6 A[1:5], цел длина)
нач
    нц для i от 1 до длина
        вывод A[i], нс
    кц
кон

алг Реверс(целта6 arr[1:5], цел N)
нач
    цел c
    нц для i от 1 до N div 2
        c:= arr[i]
        arr[i]:= arr[N+1-i]
        arr[N+1-i]:= c
    кц
кон
```

Листинг В.5 – Пример промежуточного представления LLVM IR для разворота массива

```
target triple = "arm64-apple-macosx14.0.0"

@.str.literal.0 = constant [26 x i8] c"\D0\9F\D0\BE\D1\81\D0\BB\D0\B5 \D1\80\D0\B5\D0\B2\D0\B5\D1\80\D1\81\D0\B0\00"
@.str.literal.1 = constant [2 x i8] c"\0A\00"
@.fmt.str.2 = constant [6 x i8] c"%s %s\00"
@.str.literal.3 = constant [2 x i8] c"\0A\00"
@.fmt.str.4 = constant [6 x i8] c"%d %s\00"

define i32 @main() {
entry:
    %0 = alloca i32
    %1 = alloca [5 x i32]
    %2 = alloca i32
    store i32 5, i32* %2
    %3 = sub i32 1, 1
    %4 = getelementptr [5 x i32], [5 x i32]* %1, i32 0, i32 %3
    store i32 1, i32* %4
    %5 = sub i32 2, 1
```

Листинг В.5 – Пример промежуточного представления LLVM IR для разворота массива

```
    %6 = getelementptr [5 x i32], [5 x i32]* %1, i32 0, i32 %5
    store i32 2, i32* %6
    %7 = sub i32 3, 1
    %8 = getelementptr [5 x i32], [5 x i32]* %1, i32 0, i32 %7
    store i32 3, i32* %8
    %9 = sub i32 4, 1
    %10 = getelementptr [5 x i32], [5 x i32]* %1, i32 0, i32 %9
    store i32 4, i32* %10
    %11 = sub i32 5, 1
    %12 = getelementptr [5 x i32], [5 x i32]* %1, i32 0, i32 %11
    store i32 5, i32* %12
    %13 = load i32, i32* %2
    call void @Revers_rus([5 x i32]* %1, i32 %13)
    %14 = getelementptr [26 x i8], [26 x i8]* @.str.literal.0, i32 0, i32 0
    %15 = getelementptr [2 x i8], [2 x i8]* @.str.literal.1, i32 0, i32 0
    %16 = getelementptr [6 x i8], [6 x i8]* @.fmt.str.2, i32 0, i32 0
    %17 = call i32 (i8*, ...) @printf(i8* %16, i8* %14, i8* %15)
    %18 = load i32, i32* %2
    call void @VyvodMassiva_rus([5 x i32]* %1, i32 %18)
    %19 = load i32, i32* %0
    ret i32 %19
}

define void @VyvodMassiva_rus([5 x i32]* %A, i32 %dlina_rus) {
entry:
    %0 = alloca i32
    store i32 %dlina_rus, i32* %0
    %1 = load i32, i32* %0
    %2 = alloca i32
    store i32 1, i32* %2
    br label %loop.cond.1

loop.cond.1:
    %3 = load i32, i32* %2
    %4 = icmp sle i32 %3, %1
    br i1 %4, label %loop.body.2, label %loop.end.4

loop.body.2:
    %5 = load i32, i32* %2
    %6 = sub i32 %5, 1
    %7 = getelementptr [5 x i32], [5 x i32]* %A, i32 0, i32 %6
    %8 = load i32, i32* %7
    %9 = getelementptr [2 x i8], [2 x i8]* @.str.literal.3, i32 0, i32 0
    %10 = getelementptr [6 x i8], [6 x i8]* @.fmt.str.4, i32 0, i32 0
    %11 = call i32 (i8*, ...) @printf(i8* %10, i32 %8, i8* %9)
```

Листинг В.5 – Пример промежуточного представления LLVM IR для разворота массива

```
64      br label %loop.step.3
65
66    loop.step.3:
67      %12 = load i32, i32* %2
68      %13 = add i32 %12, 1
69      store i32 %13, i32* %2
70      br label %loop.cond.1
71
72  loop.end.4:
73      ret void
74  }
75
76  define void @Revers_rus([5 x i32]* %arr, i32 %N) {
77  entry:
78      %0 = alloca i32
79      store i32 %N, i32* %0
80      %1 = alloca i32
81      %2 = load i32, i32* %0
82      %3 = sdiv i32 %2, 2
83      %4 = alloca i32
84      store i32 1, i32* %4
85      br label %loop.cond.1
86
87  loop.cond.1:
88      %5 = load i32, i32* %4
89      %6 = icmp sle i32 %5, %3
90      br i1 %6, label %loop.body.2, label %loop.end.4
91
92  loop.body.2:
93      %7 = load i32, i32* %4
94      %8 = sub i32 %7, 1
95      %9 = getelementptr [5 x i32], [5 x i32]* %arr, i32 0, i32 %8
96      %10 = load i32, i32* %9
97      store i32 %10, i32* %1
98      %11 = load i32, i32* %4
99      %12 = sub i32 %11, 1
100     %13 = getelementptr [5 x i32], [5 x i32]* %arr, i32 0, i32 %12
101     %14 = load i32, i32* %0
102     %15 = add i32 %14, 1
103     %16 = load i32, i32* %4
104     %17 = sub i32 %15, %16
105     %18 = sub i32 %17, 1
106     %19 = getelementptr [5 x i32], [5 x i32]* %arr, i32 0, i32 %18
107     %20 = load i32, i32* %19
108     store i32 %20, i32* %13
```

Листинг B.5 – Пример промежуточного представления LLVM IR для разворота массива

```
109     %21 = load i32, i32* %0
110     %22 = add i32 %21, 1
111     %23 = load i32, i32* %4
112   %24 = sub i32 %22, %23
113     %25 = sub i32 %24, 1
114     %26 = getelementptr [5 x i32], [5 x i32]* %arr, i32 0, i32 %25
115     %27 = load i32, i32* %1
116     store i32 %27, i32* %26
117     br label %loop.step.3
118
119 loop.step.3:
120     %28 = load i32, i32* %4
121     %29 = add i32 %28, 1
122     store i32 %29, i32* %4
123     br label %loop.cond.1
124
125 loop.end.4:
126     ret void
127 }
128
129 declare i32 @printf(i8* %format, ...)
```