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

ФАКУЛЬТЕТ «Информатика, искусственный интеллект и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К КУРСОВОЙ РАБОТЕ

НА ТЕМУ:

«Компилятор подмножества языка BASIC»

Студент ИУ7-23М (Группа) _____ (Подпись, дата) Керимов А. Ш. (Фамилия И. О.)

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

2022 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Компилятор — это программа, переводящая написанный на языке программирования текст в набор машинных кодов. Целью данного курсового проекта является разработка компилятора для подмножества языка BASIC.

В ходе выполнения курсового проекта необходимо решить следующие основные задачи:

— провести анализ предметной области;

— разработать блок лексического и синтаксического анализа с явным построением дерева разбора для заданного исходного кода;

— разработать блок семантического анализа и генерации кода LLVM IR.

1 Аналитический раздел

В данном разделе приводится краткий обзор предметной области. Рассмотрены основные этапы компиляции, а также проанализированы методы их реализации.

1.1 Этапы компиляции

Компиляция состоит из следующих основных этапов [1]:

— лексический анализ;

— синтаксический анализ;

— семантический анализ;

— генерация кода.

1.1.1 Лексический анализ

Задачей лексического анализа является аналитический разбор входной цепочки символов, составляющих текст компилируемой программы, с целью получения на выходе последовательности токенов. Также данный процесс называют «токенизацией».

Лексический анализатор функционирует в соответствии с некоторыми правилами построения допустимых входных последовательностей. Данные правила могут быть определены, например, в виде детерминированного конечного автомата, регулярного выражения или праволинейной грамматики. С практической точки зрения выяснено, что наиболее удобным способом является формализация работы лексического анализатора с помощью грамматики.

Лексический анализ может быть представлен и как самостоятельная фаза трансляции, и как составная часть фазы синтаксического анализа. В первом случае лексический анализатор реализуется в виде отдельного модуля, который принимает последовательность символов, составляющих текст

компилируемой программы, и выдаёт список обнаруженных лексем. Во втором случае лексический анализатор фактически является подпрограммой, вызываемой синтаксическим анализатором для получения очередной лексемы [2].

В процессе лексического анализа обнаруживаются лексические ошибки — простейшие ошибки компиляции, связанные с наличием в тексте программы недопустимых символов, некорректной записью идентификаторов, числовых констант.

На этапе лексического анализа обычно также выполняются такие действия, как удаление комментариев и обработка директив условной компиляции.

1.1.2 Синтаксический анализ

Синтаксический анализатор использует поток токенов, полученный на этапе лексического анализа для создания древовидного промежуточного представления, которое описывает грамматическую структуру потока токенов. Типичным представлением является абстрактное синтаксическое дерево, в котором каждый внутренний узел представляет операцию, а дочерние узлы — аргументы этой операции [3].

Последующие фазы компиляции используют грамматическую структуру, которая помогает проанализировать исходную программу и сгенерировать целевую.

На этапе синтаксического анализа фиксируются синтаксические ошибки, то есть ошибки, связанные с нарушением принятой структуры программы.

1.1.3 Семантический анализ

Семантический анализатор используя, синтаксические дерево, проверяет исходную программу на семантическую согласованность с опреде-

лением языка. Он также собирает информацию о всех встреченных типах и сохраняет ее для последующего использования в процессе генерации промежуточного кода.

Таким образом, семантический анализатор предназначен для нахождения семантических ошибок и накопления данных о переменных, функциях и используемых типах, используемых при генерации кода.

1.1.4 Генерация кода

Генерация кода — это последний этап процесса компиляции. Зачастую данный этап компиляции включает в себя и машинно-зависимую оптимизацию кода. На этом этапе компилятор генерирует машинно-зависимый код. Генератор кода должен иметь представление о среде выполнения целевой машины и её наборе команд.

На этом этапе компилятор выполняет несколько основных задач:

— выбор инструкций — какую инструкцию использовать;

— создание расписания инструкций — в каком порядке должны быть упорядочены инструкции;

— распределение регистров — выделение переменных в регистры процессора;

— отладка данных — отладка кода с помощью отладочных данных.

Итоговый машинный код, сгенерированный генератором кода, может быть выполнен на целевой машине. Именно так высокоуровневый исходный код, преобразуется в формат, который можно запустить на любой целевой машине.

1.2 Методы реализации лексического и синтаксического анализаторов

Реализовать лексический и синтаксический анализаторы можно двумя способами:

— используя стандартные алгоритмы анализа;

— с помощью инструментов генерации анализаторов.

1.3 Алгоритмы лексического и синтаксического анализа

Существует два основных подхода к проведению синтаксического анализа: нисходящий анализ и восходящий.

Нисходящий синтаксический анализ решает задачу построения дерева разбора в прямом порядке обхода (обход в глубину). То есть реализует поиск левого порождения входной строки [1]. В общем виде нисходящий анализ представлен в анализе методом рекурсивного спуска, который может использовать откаты, т.е. производить повторный просмотр считанных символов.

Восходящий синтаксический анализ соответствует построению дерева разбора для входной строки, начиная с листьев (снизу) и идя по направлению к корню (вверх). Общий вид восходящего синтаксического анализа известен как анализ типа «перенос/свертка» [1]. На каждом шаге свертки некоторая подстрока, соответствующая правой части продукции, замещается левым символом данной продукции.

1.4 Генераторы анализаторов

Среди наиболее используемых генераторов анализаторов выделяются: Lex и Yacc, Accent, ANTLR.

Генератор синтаксических анализаторов YACC (Yet Another Compilers' Compiler) — это программа, которая строит LALR-анализаторы. На вход

YACC получает описание грамматики в форме, близкой к форме Бэкуса-Наура и некоторую дополнительную информацию. Выходом является программа на языке Си, реализующая восходящий разбор. Как правило, Yacc применяется в сочетании с Lex — стандартным генератором лексических анализаторов.

Accent не полагается на определенные подклассы контекстно-свободных грамматик. Язык спецификаций Accent аналогичен языку Yacc, но Accent использует символические имена для атрибутов вместо чисел. Это позволяет записывать вашу грамматику в расширенной форме Бэкуса-Наура, в которой можно указывать повторения, варианты выбора и необязательные части, не вводя специальных правил для этих целей. Accent также можно использовать в сочетании с Lex.

ANTLR (ANother Tool for Language Recognition) — мощный генератор синтаксического анализатора для чтения, обработки, исполнения или перевода структурированных текстовых или двоичных файлов. Он широко используется для создания языков, инструментов и фреймворков.

Из грамматики ANTLR генерирует синтаксический анализатор, который может строить и обходить деревья синтаксического анализа. На основе правил заданной в виде РБНФ грамматики ANTLR генерирует классы нисходящего рекурсивного синтаксического анализатора. При этом обход дерева разбора можно выполнить с использованием паттернов посетитель или слушатель [4].

Принимая во внимание эффективность и простоту использования ANTLR, для построения кода синтаксического анализатора было решено применить в работе данный генератор.

1.5 LLVM

LLVM [5] — проект программной инфраструктуры для создания компиляторов и сопутствующих им утилит. Состоит из набора компиляторов из языков высокого уровня (так называемых «фронтендов»), системы оптимизации, интерпретации и компиляции в машинный код.

В основе инфраструктуры используется RISC-подобная платформо-

независимая система кодирования машинных инструкций (байткод LLVM IR), которая представляет собой высокоуровневый ассемблер.

Написан на C++, обеспечивает оптимизации на этапах компиляции, компоновки и исполнения. Изначально в проекте были реализованы компиляторы для языков Си и C++ при помощи фронтенда Clang, позже появились фронтенды для множества языков, в том числе: ActionScript, Ада, C#, Common Lisp, Crystal, CUDA, D, Delphi, Dylan, Fortran, Graphical G Programming Language, Halide, Haskell, Java (байткод), JavaScript, Julia, Kotlin, Lua, Objective-C, OpenGL Shading Language, Ruby, Rust, Scala, Swift, Xojo.

LLVM может создавать машинный код для множества архитектур, в том числе ARM, x86, x86-64, PowerPC, MIPS, SPARC, RISC-V и других (включая GPU от Nvidia и AMD).

Некоторые проекты имеют собственные LLVM-компиляторы (например LLVM-версия GCC), другие используют инфраструктуру LLVM, как например Glasgow Haskell Compiler.

Разработка начата в 2000 году в Университете Иллинойса. К середине 2010-х годов LLVM получил широкое распространение в индустрии: использовался, в том числе, в компаниях Adobe, Apple и Google. В частности, на LLVM основана подсистема OpenGL в Mac OS X 10.5, а iPhone SDK использует препроцессор (фронтенд) GCC с бэкэндом на LLVM. Apple и Google являются одними из основных спонсоров проекта, а один из основных разработчиков — Крис Латтнер — 11 лет проработал в Apple (с 2017 года — в Tesla Motors, с 2020 года — в разработчике процессоров и микроконтроллеров на архитектуре RISC-V SiFive).

LLVM поддерживает следующие типы данных:

— целые числа произвольной разрядности;

— числа с плавающей точкой;

— пустое значение void;

— массивы;

— структуры;

— вектора для упрощения SIMD-операций;

— функции.

Большинство инструкций в LLVM принимает два аргумента (операнда) и возвращает одно значение (трёхадресный код).

Значения определяются текстовым идентификатором. Локальные значения обозначаются префиксом %, а глобальные — @.

Тип операндов всегда указывается явно и однозначно определяет тип результата. Операнды арифметических инструкций должны иметь одинаковый тип, но сами инструкции «перегружены» для любых числовых типов и векторов.

LLVM поддерживает полный набор арифметических операций, побитовых логических операций и операций сдвига, а также специальные инструкции для работы с векторами.

LLVM IR строго типизирован, поэтому существуют операции приведения типов, которые явно кодируются специальными инструкциями. Набор из 9 инструкций покрывает все возможные приведения между различными числовыми типами: целыми и с плавающей точкой, со знаком и без, различной разрядности и пр. Кроме этого, есть инструкции преобразования между целыми и указателями, а также универсальная инструкция для приведения типов bitcast (ответственность за корректность таких преобразований возлагается на программиста).

Помимо значений-регистров, в LLVM есть и работа с памятью. Значения в памяти адресуются типизированными указателями. Обратиться к памяти можно с помощью двух инструкций: load и store.

Инструкция alloca выделяет память на стеке. Память, выделенная alloca, автоматически освобождается при выходе из функции при помощи инструкций ret или unwind.

Для вычисления адресов элементов массивов, структур и т. д. с правильной типизацией используется инструкция getelementptr. getelementptr только вычисляет адрес, но не обращается к памяти. Инструкция принимает

произвольное количество индексов и может разыменовывать структуры любой вложенности.

Также существует инструкции extractvalue и insertvalue. Они отличаются от getelementptr тем, что принимают не указатель на агрегатный тип данных (массив или структуру), а само значение такого типа. extractvalue возвращает соответственное значение подэлемента, а insertvalue порождает новое значение агрегатного типа.

2 Конструкторский раздел

2.1 Структура компилятора

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

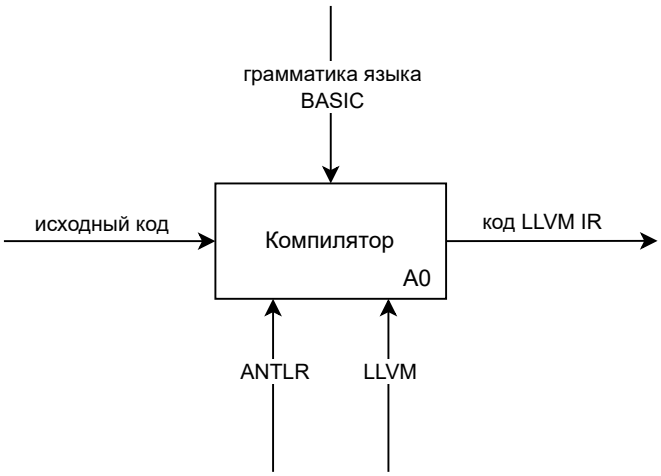

Рисунок 2.1 — Концептуальная модель в нотации IDEF0

На рисунке 2.2 представлена детализированная концептуальная модель в нотации IDEF0. ANTLR фронтенд получает на вход исходный код программы и генерирует конкретное синтаксическое дерево, на основе которого компилятором генерируется LLVM IR.

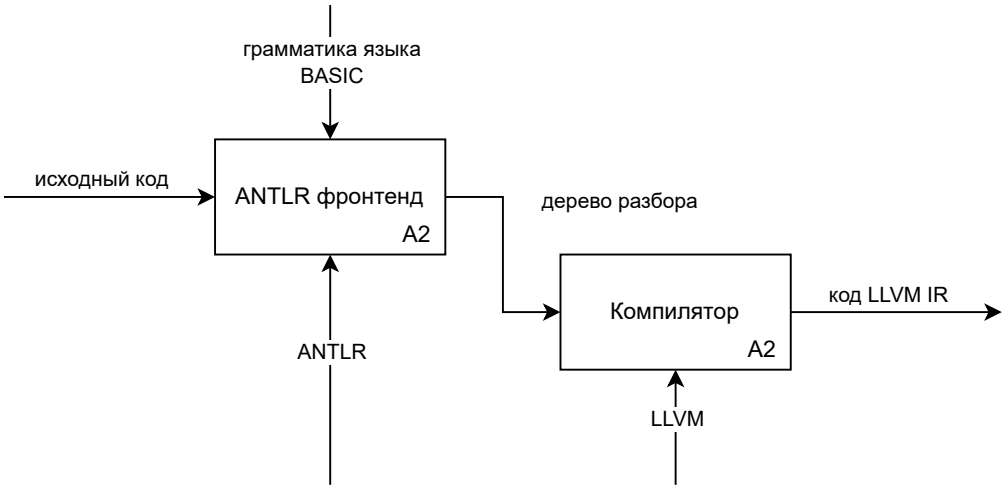

Рисунок 2.2 — Детализированная концептуальная модель в нотации IDEF0

2.2 Метод обхода AST

Для обхода абстрактного синтаксического дерева, которое строит ANTLR, использовался паттерн слушатель. ANTLR генерирует интерфейс слушателя, который реагирует на события, запускаемые встроенным классом для обхода дерева. Методы в слушателе — это просто обратные вызовы, которые вызываются при вхождении в некоторую вершину дерева при обходе (enter-методы) и при выходе (exit-методы) [4].

ANTLR генерирует интерфейс слушателя специфичный для заданной грамматики и содержащий exit- и enter-методы для каждого правила, которые нужно реализовать в зависимости от решаемой задачи. На рисунках 2.3 и 2.4 представлен порядок обхода дерева средствами ANTLR и порядок вызовов коллбэк-методов класса слушателя.

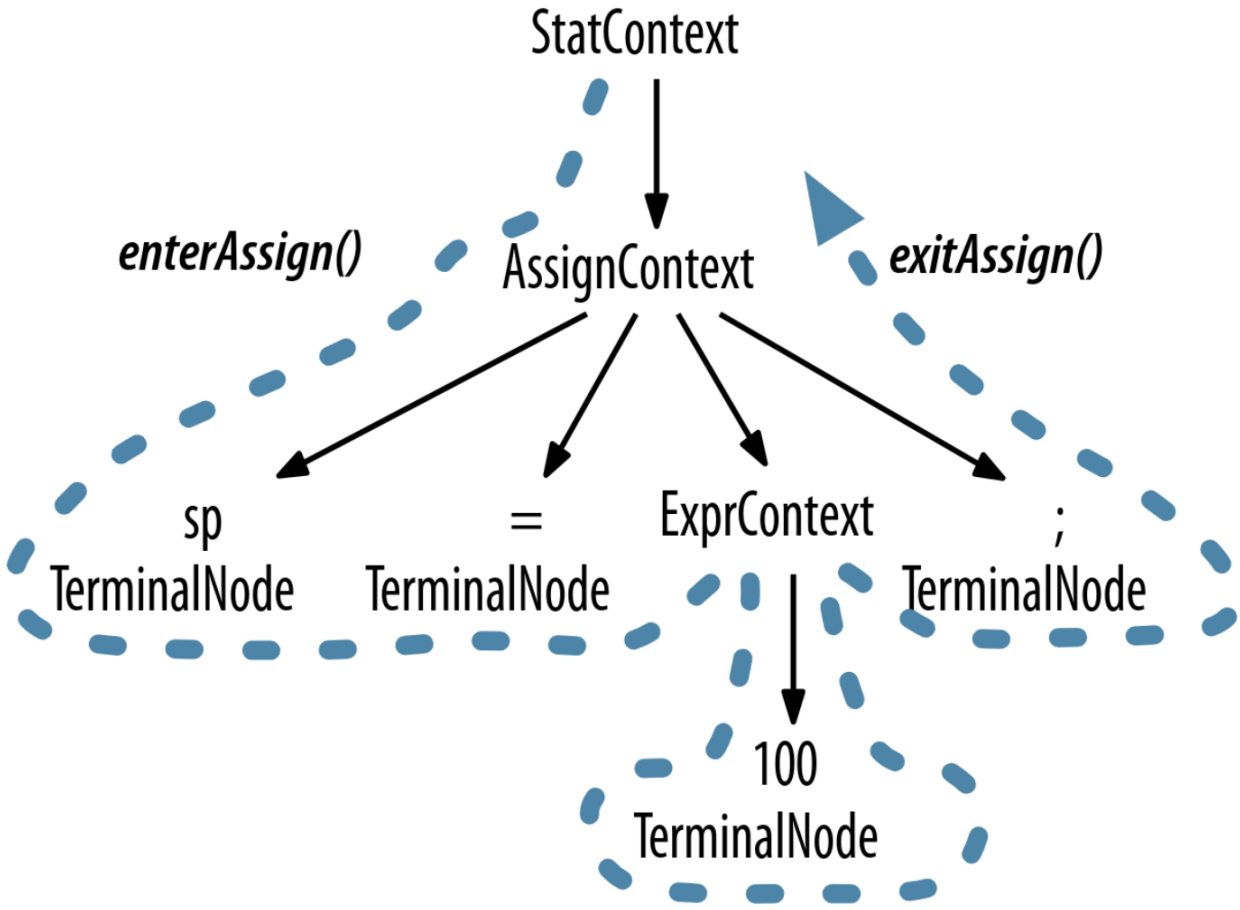

Рисунок 2.3 — Порядок обхода дерева разбора

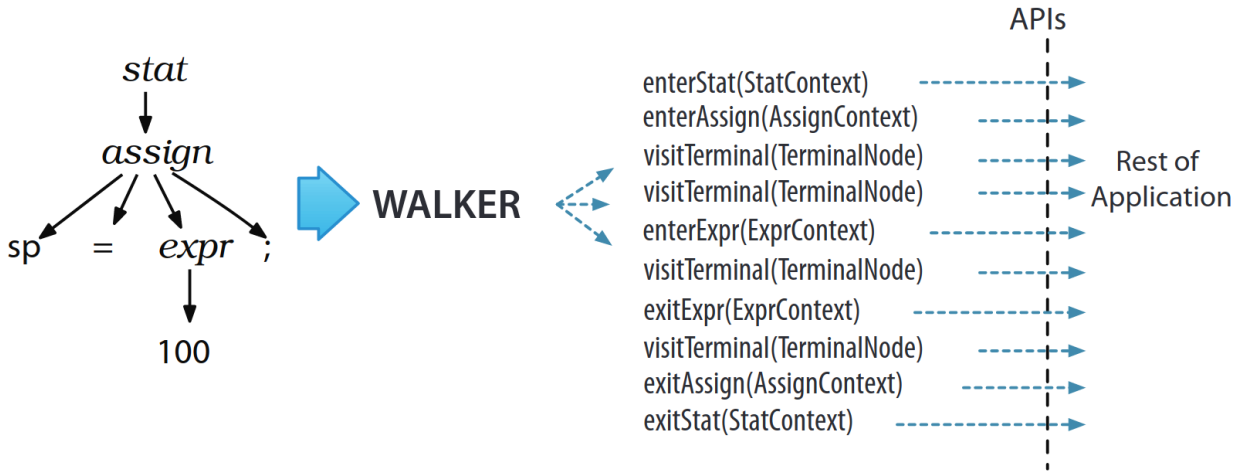

Рисунок 2.4 — Порядок вызова методов слушателя

2.3 Генерация LLVM IR

Для генерации кода LLVM IR необходимо произвести обход дерева разбора, полученного из парсера ANTLR, и в каждой вершине произвести генерацию соответствующую её типу.

Схема алгоритма для генерации кода LLVM IR для цикла for с выражениями, выполняющимися до входа в цикл и после выполнения тела цикла, приведена на рисунках 4 и 5.

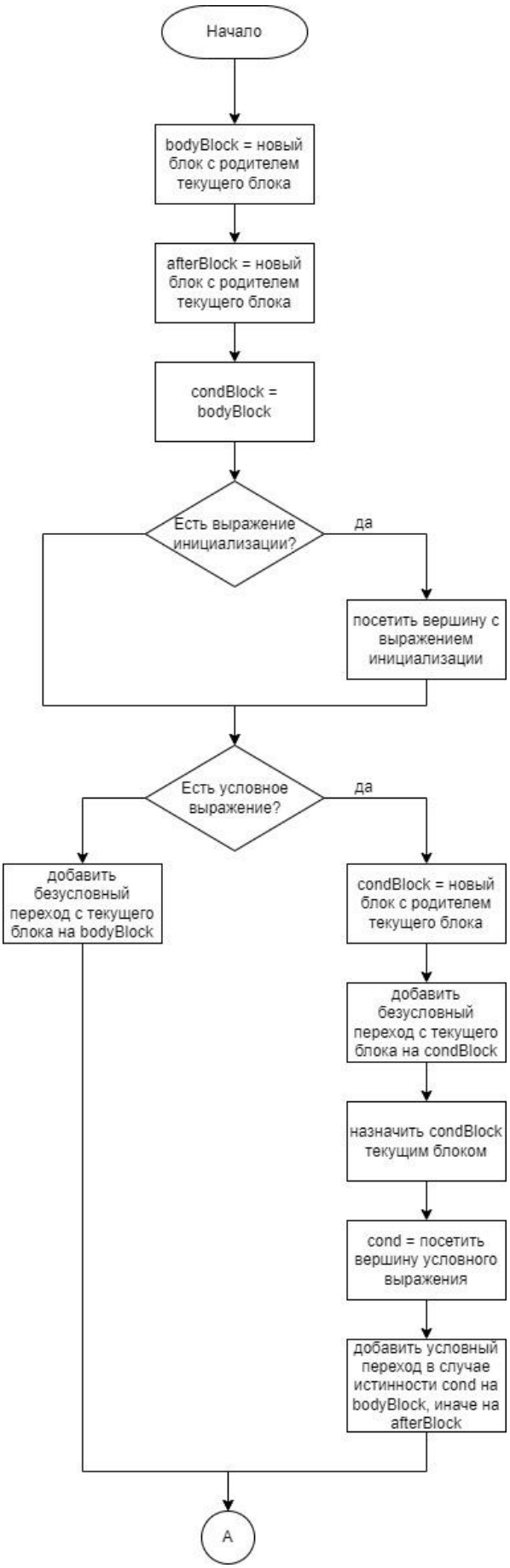

Рисунок 2.5 — Схема генерации кода для выражения инициализации и условного выражения цикла for

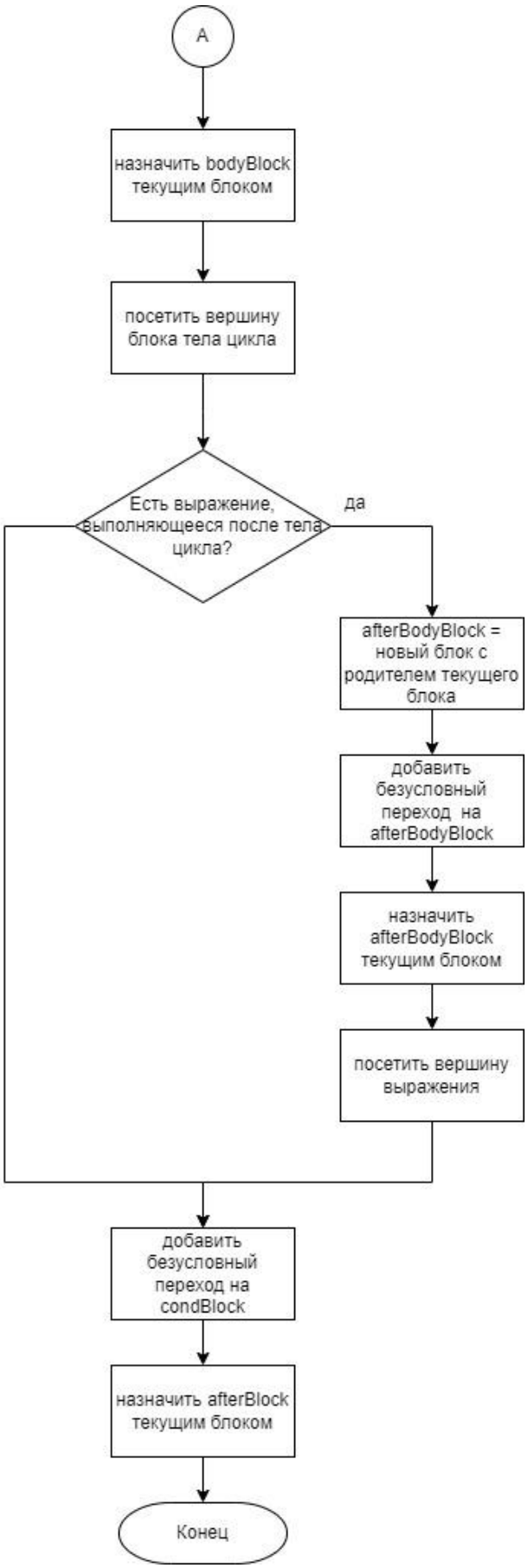

Рисунок 2.6 — Схема генерации кода для тела и выражения, выполняющегося после тела цикла for

3 Технологический раздел

3.1 Сгенерированные классы анализаторов

В результате работы ANTLR будут сгенерированы следующие классы:

— BasicLexer — лексический анализатор.

— BasicParser — синтаксический анализатор.

— BasicParserVisitor — абстрактный класс посетителя для обхода дерева.

— BasicParserListener — абстрактный класс слушателя, с пустыми методами.

Для обхода дерева, необходимо создать класс-наследник базового класса слушателя и переопределить enter- и exit-методы, соответствующие правилам исходной грамматики.

3.2 Обнаружение ошибок

Все ошибки, обнаруженные на этапах лексического и синтаксического анализов, обрабатываются средствами ANTLR. Возникающие при этом исключения обрабатываются и выводится текст соответствующей ошибки. Также при построении графа вызовов обрабатываются ошибки вызова функции до ее объявления.

3.3 Пример компиляции программы

Пример программы, написанный на рассматриваемом подмножестве языка BASIC представлен на листинге 3.1. Данная программа реализует сортировку пузырьком.

Листинг 3.1 — Программа на рассматриваемом подмножестве языка BASIC

```
SUB Main
        DIM A(100)
        INPUT "Array size: ", n
```

```
        PRINT "Input array"
        FOR i = 1 TO n + 1
                INPUT A(i)
        END FOR

        FOR i = 0 TO n
                FOR j = 2 TO n - i + 1
                        IF A(j) < A(j - 1) THEN
                                LET tmp = A(j)
                                LET A(j) = A(j - 1)
                                LET A(j - 1) = tmp
                        END IF
                END FOR
        END FOR

        FOR i = 1 TO n + 1
                PRINT A(i)
        END FOR
END SUB
```

В листинге 3.2 в приложении A приведен сгенерированный LLVM IR код для представленной на листинге 3.1 программы.

ЗАКЛЮЧЕНИЕ

В ходе работы над данным проектом был проведён анализ предметной области, разработан блок лексического и синтаксического анализа с явным построением дерева разбора для заданного исходного кода, разработан блок семантического анализа и генерации кода LLVM IR.

В результате чего был реализован компилятор подмножества языка BASIC с использованием ANTLR и LLVM.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Ахо А., Сети Р., Ульман Д. Компиляторы: принципы, технологии и инструментарий, 2-е изд.: Пер. с англ. — М.: ООО «И.Д. Вильямс», 2008. — 1184 с.

2. Серебряков В. А., Галочкин М. П. Основы конструирования компиляторов.

3. Ахо А., Ульман Дж. Теория синтаксического анализа, перевода и компиляции. Том 1.: Пер. с англ. — М.: «Мир», 1978.

4. Terence Parr. The Definitive ANTLR4 Reference. — M.: Pragmatic Bookshelf, 2013.

5. Chris Lattner. The Design of LLVM. Dr. Dobb's Journal, 2012.

ПРИЛОЖЕНИЕ А

Листинг 3.2 — Сгенерированный LLVM IR код

```
@g_str = private unnamed_addr constant [13 x i8] c"Array size: \00", align 1
@g_str.1 = private unnamed_addr constant [12 x i8] c"Input array\00", align 1
@g_str.2 = private unnamed_addr constant [2 x i8] c"?\00", align 1
@.str = private unnamed_addr constant [4 x i8] c"%s \00", align 1
@.str.1 = private unnamed_addr constant [4 x i8] c"%lf\00", align 1
@.str.2 = private unnamed_addr constant [5 x i8] c"%lf\0A\00", align 1
@stdin = external global %struct._IO_FILE*, align 8
@.str.3 = private unnamed_addr constant [4 x i8] c"%s\0A\00", align 1

define void @Main() {
        label_start:
        %A_addr = alloca double, i32 100, align 8
        %n_addr = alloca double, align 8
        %i_addr = alloca double, align 8
        %j_addr = alloca double, align 8
        %tmp_addr = alloca double, align 8
        %0 = call double @bsq_number_input(i8* getelementptr inbounds ([13 x i8], [13 x
            i8]* @g_str, i32 0, i32 0))
        store double %0, double* %n_addr, align 8
        call void @bsq_text_print(i8* getelementptr inbounds ([12 x i8], [12 x i8]*
            @g_str.1, i32 0, i32 0))
        store double 1.000000e+00, double* %i_addr, align 8
        %n = load double, double* %n_addr, align 8
        %add = fadd double %n, 1.000000e+00
        br label %1

        1:                                                ; preds = %4, %label_start
        %2 = load double, double* %i_addr, align 8
        %3 = fcmp olt double %2, %add
        br i1 %3, label %4, label %11

        4:                                                ; preds = %1
        %5 = call double @bsq_number_input(i8* getelementptr inbounds ([2 x i8], [2 x
            i8]* @g_str.2, i32 0, i32 0))
        %i = load double, double* %i_addr, align 8
        %6 = fptosi double %i to i32
        %7 = add i32 -1, %6
        %8 = getelementptr double, double* %A_addr, i32 %7
        store double %5, double* %8, align 8
        %9 = load double, double* %i_addr, align 8
        %10 = fadd double %9, 1.000000e+00
        store double %10, double* %i_addr, align 8
        br label %1

        11:                                               ; preds = %1
        store double 0.000000e+00, double* %i_addr, align 8
        %n1 = load double, double* %n_addr, align 8
        br label %12

        12:                                               ; preds = %48, %11
        %13 = load double, double* %i_addr, align 8
        %14 = fcmp olt double %13, %n1
        br i1 %14, label %15, label %51

        15:                                               ; preds = %12
        store double 2.000000e+00, double* %j_addr, align 8
        %n2 = load double, double* %n_addr, align 8
```

```
%i3 = load double, double* %i_addr, align 8
%sub = fsub double %n2, %i3
%add4 = fadd double %sub, 1.000000e+00
br label %16

16:                                               ; preds = %45, %15
%17 = load double, double* %j_addr, align 8
%18 = fcmp olt double %17, %add4
br i1 %18, label %19, label %48

19:                                               ; preds = %16
br label %20

20:                                               ; preds = %19
%j = load double, double* %j_addr, align 8
%21 = fptosi double %j to i32
%22 = add i32 -1, %21
%23 = getelementptr double, double* %A_addr, i32 %22
%24 = load double, double* %23, align 8
%j5 = load double, double* %j_addr, align 8
%sub6 = fsub double %j5, 1.000000e+00
%25 = fptosi double %sub6 to i32
%26 = add i32 -1, %25
%27 = getelementptr double, double* %A_addr, i32 %26
%28 = load double, double* %27, align 8
%lt = fcmp olt double %24, %28
br i1 %lt, label %29, label %44

29:                                               ; preds = %20
%j7 = load double, double* %j_addr, align 8
%30 = fptosi double %j7 to i32
%31 = add i32 -1, %30
%32 = getelementptr double, double* %A_addr, i32 %31
%33 = load double, double* %32, align 8
store double %33, double* %tmp_addr, align 8
%j8 = load double, double* %j_addr, align 8
%sub9 = fsub double %j8, 1.000000e+00
%34 = fptosi double %sub9 to i32
%35 = add i32 -1, %34
%36 = getelementptr double, double* %A_addr, i32 %35
%37 = load double, double* %36, align 8
%j10 = load double, double* %j_addr, align 8
%38 = fptosi double %j10 to i32
%39 = add i32 -1, %38
%40 = getelementptr double, double* %A_addr, i32 %39
store double %37, double* %40, align 8
%tmp = load double, double* %tmp_addr, align 8
%j11 = load double, double* %j_addr, align 8
%sub12 = fsub double %j11, 1.000000e+00
%41 = fptosi double %sub12 to i32
%42 = add i32 -1, %41
%43 = getelementptr double, double* %A_addr, i32 %42
store double %tmp, double* %43, align 8
br label %45

44:                                               ; preds = %20
br label %45

45:                                               ; preds = %44, %29
%46 = load double, double* %j_addr, align 8
%47 = fadd double %46, 1.000000e+00
```

```
        store double %47, double* %j_addr, align 8
        br label %16

        48:                                               ; preds = %16
        %49 = load double, double* %i_addr, align 8
        %50 = fadd double %49, 1.000000e+00
        store double %50, double* %i_addr, align 8
        br label %12

        51:                                               ; preds = %12
        store double 1.000000e+00, double* %i_addr, align 8
        %n13 = load double, double* %n_addr, align 8
        %add14 = fadd double %n13, 1.000000e+00
        br label %52

        52:                                               ; preds = %55, %51
        %53 = load double, double* %i_addr, align 8
        %54 = fcmp olt double %53, %add14
        br i1 %54, label %55, label %62

        55:                                               ; preds = %52
        %i15 = load double, double* %i_addr, align 8
        %56 = fptosi double %i15 to i32
        %57 = add i32 -1, %56
        %58 = getelementptr double, double* %A_addr, i32 %57
        %59 = load double, double* %58, align 8
        call void @bsq_number_print(double %59)
        %60 = load double, double* %i_addr, align 8
        %61 = fadd double %60, 1.000000e+00
        store double %61, double* %i_addr, align 8
        br label %52

        62:                                               ; preds = %52
        ret void
}

define i32 @main() {
        start:
        call void @Main()
        ret i32 0
}
```