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 и системы управления

КАФЕДРА Теоретическая информатика и компьютерные технологии

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА К КУРСОВОЙ РАБОТЕ ПО КУРСУ БАЗЫ ДАННЫХ НА ТЕМУ:

«Компилятор диалекта Бейсика в ассемблер»

Студент _____ _____
подпись, дата *фамилия, и.о.*

Научный руководитель _____ _____
подпись, дата *фамилия, и.о.*

Москва 2024

О Г Л А В Л Е Н И Е

В В Е Д Е Н И Е

В мире программирования и разработки программного обеспечения компиляторы играют ключевую роль, обеспечивая перевод высокоуровневого исходного кода программы в машинный код, который может быть исполнен целевой вычислительной машиной. Исторически компиляторы были созданы для различных языков программирования, и каждый компилятор обычно имеет свои особенности, оптимизации и специфические подходы к генерации кода.

Одним из наиболее широко используемых языков программирования является BASIC (Beginner's All-purpose Symbolic Instruction Code) - язык, который был разработан в конце 1960-х годов и стал популярным благодаря своей простоте и доступности. В ходе его развития появилось множество диалектов и вариаций, а также специфические реализации, включая интерпретаторы и компиляторы.

Целью данной курсовой работы является исследование процесса создания компилятора для диалекта языка программирования BASIC, использующего ассемблер в качестве целевой платформы. Ассемблер, как наиболее низкоуровневый язык программирования, обладает прямым управлением аппаратурой компьютера и предоставляет разработчику возможность максимального контроля над исполняемым кодом.

В рамках данной курсовой работы будут рассмотрены основные этапы проектирования и реализации компилятора, а также принципы и техники, используемые при преобразовании исходного кода диалекта BASIC в код целевого ассемблера. Кроме того, будут изучены особенности архитектуры BASIC и возможности их адаптации и оптимизации при создании компилятора.

Данная курсовая работа расширяет понимание процесса трансляции высокоуровневых языков программирования в машинный код, а также открывает новые перспективы для применения ассемблера в разработке программного обеспечения.

1. Анализ поставленной задачи

В ходе выполнения курсовой работы необходимо разработать лексическую структуру и грамматику для диалекта BASIC по спецификаций, полученной от преподавателя, в дальнейшем реализовать стадию анализа для полученной лексической структуры. Также необходимо выбрать целевой ассемблер, разработать подход к генераций кода целевого ассемблера и реализовать генерацию кода. Разработать библиотеку поддержки времени выполнения, которая содержит реализацию стандартных функций и процедур, используемых в программе. Сгенерированный код должен быть совместим на стадий компановки с кодом на языке C.

1.1 Формулировка поставленной задачи

Разработать компилятор BASIC, который транслирует исходный код на диалекте BASIC в ассемблерный код для целевой платформы. Этот компилятор должен быть способен обрабатывать синтаксические конструкции BASIC, включая операторы, выражения, управляющие структуры и вызовы функций. Генерируемый ассемблерный код должен быть совместим с кодом на языке программирования C при компоновке. Кроме того, необходимо разработать библиотеку поддержки времени выполнения, которая предоставляет доступ к стандартным функциям и процедурам BASIC.

1.2 Исследование предметной области

Язык программирования BASIC (Beginner's All-purpose Symbolic Instruction Code) - это высокоуровневый язык программирования, который был разработан для упрощения процесса обучения программированию и создания программ для начинающих. Изначально созданный в 1960-х годах Джоном Кемени и Томом Куртом в Дартмутском колледже в США, BASIC был одним из первых языков, который стал доступным для широкого круга пользователей.

BASIC был разработан для использования студентами, не имеющими опыта в программировании. Первоначальная версия предоставляла набор инструкций для работы с числами, строками и массивами данных. С течением времени язык развивался, добавлялись новые функции и возможности, что сделало его популярным среди как начинающих, так и опытных программистов.

Основные особенности языка BASIC включают в себя:

1. **Простота**:

   BASIC изначально разрабатывался для того, чтобы быть простым и понятным для начинающих программистов. Он предоставляет простой и понятный синтаксис, что облегчает освоение основ программирования.

2. **Интерактивность**:

   Многие реализации BASIC предлагают интерактивный режим, позволяющий пользователю вводить и исполнять команды непосредственно в процессе работы с программой, что делает процесс отладки и тестирования более простым.

3. **Портативность**:

   BASIC имеет множество различных реализаций, которые могут работать на различных платформах, что делает его доступным для широкого круга пользователей.

4. **Использование в образовании**:

   BASIC широко используется в образовательных учреждениях для обучения основам программирования благодаря своей простоте и доступности.

5. **Расширяемость**: Хотя BASIC изначально был простым языком, в него были добавлены различные расширения и возможности, позволяющие программистам создавать более сложные программы.

Варианты BASIC

С течением времени было разработано множество различных вариантов BASIC. Некоторые из наиболее известных включают:

1. **Classic Basic:**
    - Одна из самых ранних реализаций Basic, включает простые команды и структуры управления, такие как PRINT, INPUT, IF...THEN...ELSE, FOR...NEXT.
    - Примеры реализаций: Dartmouth BASIC, Microsoft BASIC.
2. **Visual Basic (VB):**
    - Развитие Basic с упором на создание графических пользовательских интерфейсов (GUI).
    - Имеет интегрированную среду разработки (IDE), предоставляющую удобные инструменты для создания приложений.
    - Примеры: Visual Basic 6, Visual Basic .NET.
3. **QBASIC:**
    - Интерпретируемая версия Basic, включенная в операционную систему MS-DOS.
    - Предоставляет базовые средства разработки для начинающих программистов.
4. **FreeBASIC:**
    - Современная реализация Basic, ориентированная на совместимость с QuickBASIC и поддерживающая множество операционных систем, включая Windows, Linux и DOS.
    - Обладает мощными возможностями, включая поддержку объектно-ориентированного программирования и низкоуровневой работы с памятью.
    - Поддерживает как процедурное, так и модульное программирование.

5. **SmallBASIC:**

- Легковесная и простая в освоении версия Basic, предназначенная для обучения программированию.
- Имеет ограниченные возможности по сравнению с более полноценными версиями Basic, но при этом подходит для начинающих.

BASIC играл значительную роль в развитии компьютерного обучения и программирования, обеспечивая простой и доступный язык для начинающих. Несмотря на то что его популярность снизилась с появлением более мощных языков программирования, включая C++ и Java, BASIC остается важным языком, особенно в образовательных целях и для быстрого прототипирования программ.

1.3 Теоретическая часть

Разработока компилятора представляет из себя последовательную реализацию этапов, через которые проходит исходный код программы, преобразуясь из высокоуровневого представления в машинный код или исполняемый файл. Они включают в себя: Лексический анализ, Синтаксический анализ, Семантический анализ, Оптимизация кода, Генерация кода, Компоновка

Каждая фаза компиляции играет свою роль в процессе преобразования исходного кода в исполняемый файл, обеспечивая его корректность, эффективность и готовность к выполнению на целевой платформе.

1.3.1 Лексический анализ

Лексический анализ является первым этапом компиляции, отвечающим за преобразование исходного кода программы из последовательности символов в поток лексем, или токенов. Цель лексического анализа состоит в том, чтобы выделить значимые элементы программы и преобразовать их во внутреннее

представление, которое будет использоваться для последующего синтаксического анализа.

Этапы работы лексического анализа:

1. **Сканирование входного потока символов**:

   Лексический анализатор начинает работу с чтения символов из входного потока и формирования лексем на основе распознанных шаблонов. При этом символы могут агрегироваться в группы (например, для чисел или строк), идентифицироваться как ключевые слова или операторы, или просто считываться как отдельные символы.

2. **Идентификация лексем (токенов)**:

   В процессе сканирования входного потока символов лексический анализатор идентифицирует лексемы - минимальные значимые единицы исходного кода, такие как идентификаторы, ключевые слова, операторы, числа, строки и другие элементы. Каждой лексеме присваивается соответствующий тип.

3. **Формирование токенов**:

   После идентификации лексемы лексический анализатор создает соответствующий токен, который представляет собой структуру данных, содержащую информацию о типе лексемы, ее значении (если таковое есть), а также местонахождении в исходном коде (например, номер строки и позиция символов).

4. **Обработка ошибок и исключений**:

   В случае обнаружения некорректной последовательности символов, несоответствующей правилам лексического анализа, лексический анализатор должен обрабатывать ошибки и исключения. Это может включать в себя генерацию сообщений об ошибке или исключений, указывающих на неправильный формат или нераспознанные символы.

5. **Возвращение токенов синтаксическому анализатору**:

После сканирования и идентификации всех лексем входного кода лексический анализатор возвращает полученные токены синтаксическому анализатору для дальнейшей обработки и построения синтаксического дерева разбора.

Лексический анализатор зачастую реализуется с использованием объектно-ориентированной модели. Он сканирует входной поток символов и идентифицирует лексемы в соответствии с заранее заданными доменам, После этого он передает лексемы синтаксическому анализатору для дальнейшей обработки.

1.3.2 Синтаксический анализ

Следующим этапом разработки компилятора является этап синтаксического анализа и для этого необходим синтаксический анализатор.

Синтаксический анализатор принимает на вход поток лексем (токенов), полученных от лексического анализатора, и проверяет их на принадлежность грамматике языка. Основная задача синтаксического анализатора состоит в построении синтаксического дерева разбора, которое представляет собой структуру данных, отражающую иерархию синтаксических конструкций программы. Корень дерева обычно представляет программу в целом, а узлы дерева представляют собой операторы, выражения, управляющие конструкции и т. д.

В случае обнаружения синтаксической ошибки, такой как неправильный порядок лексем или неожиданный символ, синтаксический анализатор должен обработать ошибку и, по возможности, продолжить разбор программы. Это может включать восстановление после ошибки и продолжение разбора с теми лексемами, которые могут быть корректно идентифицированы.

Реализация синтаксического анализатора может быть выполнена с использованием различных алгоритмов, включая рекурсивный спуск, LR-парсеры, LL-парсеры и их комбинации. Как правило, синтаксический анализатор является одним из основных компонентов компилятора и может быть реализован как отдельный модуль или класс.

1.3.3 Семантический анализ

Семантический анализ - это важный этап компиляции, который следует за лексическим и синтаксическим анализом. Он отвечает за анализ и проверку смысла программы, обнаружение семантических ошибок и создание внутреннего представления программы, которое будет использоваться для дальнейшей оптимизации и генерации кода. Рассмотрим более подробно этот этап:

1. **Анализ типов данных**:

   Одной из ключевых задач семантического анализа является проверка совместимости типов данных в выражениях и операторах. Анализатор должен убедиться, что операции применяются к соответствующим типам данных, что операнды совместимы и что результаты операций имеют ожидаемые типы.

2. **Проверка объявлений и областей видимости**:

   Семантический анализатор анализирует объявления переменных, функций и других идентификаторов в программе, а также проверяет их области видимости. Это включает в себя проверку наличия объявлений перед использованием идентификаторов, уникальность имен в каждой области видимости и доступ к переменным внутри правильной области видимости.

3. **Разрешение имён**:

   При наличии нескольких объявлений с одинаковыми именами семантический анализатор должен разрешить, к какому именно объявлению обращается данное использование имени. Это может включать в себя учет областей видимости, а также правил приоритета и перегрузки имен.

4. **Анализ управляющих структур**:

   Семантический анализатор проверяет правильность использования управляющих структур, таких как циклы, условные операторы и ветвления. Это включает в себя проверку наличия условных выражений в условных операторах, правильность выходов из циклов и т. д.

5. **Вызовы функций и процедур**:

   Анализатор проверяет корректность вызовов функций и процедур, включая соответствие количества и типов аргументов ожидаемым параметрам, наличие объявлений вызываемых функций и процедур и другие аспекты, связанные с вызовами функций.

6. **Проверка семантических ограничений языка**:

   Кроме того, семантический анализатор выполняет проверку соблюдения специфических семантических ограничений, предусмотренных для данного языка программирования. Это могут быть ограничения, связанные с безопасностью типов, использованием указателей, работой с памятью и другие.

7. **Генерация внутреннего представления**:

   В конце семантического анализа создается внутреннее представление программы, которое может быть использовано для последующих этапов компиляции, таких как оптимизация и генерация кода. Это внутреннее представление часто представляется в виде абстрактного синтаксического дерева (AST) или других структур данных.

1.3.4 Генерация кода

Следующим этапом является этап генерации кода - это один из ключевых этапов компиляции, на котором абстрактное синтаксическое дерево, преобразуется в машинный код или другой формат исполняемого файла, который может быть непосредственно выполнен на целевой платформе.

Этапы генераций кода:

1. **Выбор целевой архитектуры**:

   Перед тем как начать генерацию кода, необходимо определить целевую архитектуру или платформу, на которой будет выполняться созданный код. Это включает в себя выбор целевого процессора, операционной системы и других характеристик, которые могут повлиять на генерацию кода.

2. **Преобразование внутреннего представления в машинный код**:

   На основе внутреннего представления программы, которое может быть абстрактным синтаксическим деревом или другой структурой данных, генератор кода создает соответствующий машинный код. Это включает в себя преобразование операций и структур данных из высокоуровневого представления в инструкции машинного кода, понятные процессору.

3. **Генерация дополнительных данных**:

   Помимо машинного кода, генератор кода также может создавать дополнительные данные, необходимые для выполнения программы, такие как таблицы данных, таблицы виртуальной памяти, таблицы переходов и другие.

4. **Поддержка окружения выполнения**:

   В зависимости от характеристик целевой платформы, генератор кода может включать в себя поддержку специфических функций или API, необходимых для взаимодействия с окружением выполнения, таким как операционная система, библиотеки или внешние устройства.

1.3.5 Компоновка

Этап компоновки - это процесс объединения различных модулей программы в единый исполняемый файл или библиотеку. Основные шаги этого этапа включают:

1. **Выбор и загрузка модулей**: Выбор необходимых модулей программы и их загрузка в компоновщик.
2. **Создание таблиц символов**: Формирование таблицы символов, которая содержит информацию о всех функциях, переменных и других символах в программе.
3. **Генерация исполняемого файла или библиотеки**: Создание окончательного исполняемого файла или библиотеки, который может быть загружен и выполнен операционной системой.

2. Разработка

2.1 Спецификация диалекта языка программирования BASIC

- Комментарии начинаются с апострофа ' и продолжаются до конца строки.
- Идентификаторы и ключевые слова не чувствительны к регистру.
- В имени каждой переменной и каждой функции указывается её тип (%— целое, &—длинное целое, ! — вещественное одинарной точности, # — вещественное двойной точности, $— строка).
- Внутри функции неявно объявляется переменная с тем же именем, что и имя самой функции — её значение является возвращаемым значением функции.
- И индексация массивов, и вызов функции записываются при помощи круглых скобок.
- Цикл с условием может быть записан пятью способами:

```
Do While ….     Do Until ….     Do              Do              Do
   ….              ….              ….              ….              ….
Loop            Loop            Loop While ….   Loop Until ….   Loop
```

Листинг 1.

Первые две формы — циклы с предусловием (положительным и отрицательным), две другие — с постусловием и, наконец, пятая — бесконечный цикл.

- Цикл For можно прерывать операторами:

```
Exit For
Exit For i%
```

Листинг 2.

Первая форма прерывает текущий цикл, вторая (с переменной) позволяет прервать сразу несколько вложенных циклов.

- Цикл Do/Loop можно прервать любым из двух операторов (они синонимы):

```
Exit Do
Exit While
```

Листинг 3.

- Процедуры и функции прерываются, соответственно, операторами

```
Exit Sub
Exit Function
```

Листинг 4.

- Можно определять глобальные переменные:

```
Dim some_global_var%
Dim some_global_array%(100)
```

Листинг 5.

- Можно определять многомерные массивы:

```
Dim matrix(100, 100)
```

Листинг 6.

Семантика диалекта языка программирования BASIC:

- В программе не может быть двух функций с одинаковыми именами и сигнатурами
- В программе не может быть двух переменных с одинаковыми именами в одной области видимости
- В программе нельзя обратиться к переменной/вызвать функцию до её объявления

- Операции +, -, /, * могут применяться к аргументам числовых типов. Если операнды имеют равный тип, то результатом операции будет больший из них. Типы по возрастанию: целое короткое целое, длинное целое, вещественное одинарной точности, вещественное двойной точности.
- Операция + применима также и к строкам, результатом является строка.
- Значение числового типа можно присваивать переменной или передавать в качестве параметра того же или большего типа
- Значение строкового типа можно присваивать переменной или передавать в качестве параметра только строкового типа.
- Индексами массивов могут быть только целочисленные значения.
- Индексирование массива начинается с 1
- Передача массивов и строк в функцию и сопрограмму осуществляется по ссылке, остальных типов по значению
- Переменной цикла For может быть только целочисленная.
- Массивы инициализируются оператором Dim. Размер массива может задаваться только целочисленным положительным константным значением.

Пример корректной программы на данном диалекте BASIC приведён в приложений А.

2.2 Грамматика диалекта BASIC

В ходе выполнения курсовой работы была разработана грамматика диалекта языка программирования BASIC с помощью спецификаций описанной в разделе 2.1. (в соответствий с приложением Б.)

Из описанной грамматики можно выделить следующие терминалы:

```
SUB; FUNCTION; END; DECLARE; DIM; PRINT
"{";  "}";  "("; ")"; ","
"%"; "&"; "!"; "#"; "$"
IF; THEN; ELSE
FOR; TO; NEXT; EXIT
DO; LOOP; WHILE; UNTIL
">"; "<"; ">="; "<="; "="; "<>"; "+"; "-"
```

Листинг 7.

Описание нетерминалов грамматики:

- Program — нетерминал содержащий все нетерминалы глобальных определений (символов)
- GlobalSymbols — список нетерминалов глобальных символов
- GlobalSymbol — нетерминал переписывающийся в один из возможных 5 глобальных определений
- SubroutineProto — Прототип сопрограммы, описывающий имя и параметры, которые принимает сопрограмма
- FunctionProto — Прототип функций, описывающий имя и параметры, которые принимает функция
- SubroutineDef — Определение сопрограммы, содержащее прототип сопрограммы и её тело
- FunctionDef — Определение функций, содержащее прототип функций и её тело

- SubroutineDecl — Объявление сопрограммы, содержащее прототип сопрограммы
- FunctionDecl — Объявление функций, содержащее прототип функций
- VariableDecl — Обьявление переменной, содержащее имя переменной, тип и возможно пустую инициализацию
- VariableInit — Инициализируемое значение переменной, для простых типов — некоторое выражение, для массивов — список инициализаций
- InitializerList — список инициализаций, содержащий либо выражения, либо списки инициализаций, необходимые для инициализаций массивов больших размерностей, чем 1
- InitializerListValues — список значений списка инициализаций
- InitializerListValue — значение списка инициализаций
- ParametersList — список параметров функций или сопрограммы
- NonEmptyParametersList — не пустой список параметров функций или сопрограммы
- VarnameOrArrayParam — параметр простого типа или массив
- CommaList — список запятых, содержащихся в параметре типа массив
- ArgumentsList — список аргументов, передаваемых в функцию или сопрограмму
- NonEmptyArgumentsList — не пустой список аргументов, передаваемых в функцию или сопрограмму
- VarnameOrArrayArg — аргумент просто типа или типа массив, передаваемых в функцию или сопрограмму
- FuncCallOrArrayIndex — Вызов функций или обращение по индексу(индексам) к массиву
- FuncCall — Вызов функций
- Varname — имя переменной и её тип

- Type — простой тип переменной
- Statements — список операторов, который может закончится оператором прерывания
- NonEmptyStatements — не пустой список операторов, который может закончится оператором прерывания
- Statement — один из возможных 6 операторов
- AssignStatement — оператор присваивания, который может быть также определён как опередение переменной
- IfStatement — оператор ветвления, содержащий тело положительного блока и возможно содержащий тело отрицательного блока
- ExitStatement — один из возможных 6 операторов прерывания
- Loop — нетерминал содержащий либо цикл со счётчиком, либо цикл без счётчика
- ForLoop — цикл со счетчиком, содержащий начальное значение и финальное значение счётчика, тело цикла и инкремент
- WhileLoop — один из 5 возможных циклов без счётчика
- PreLoop — цикл без счётчика с предусловием
- PostLoop — цикл без счётчика с постусловием
- Expr — нетерминал содержащий некоторое выражение
- CmpOp — операторы сравнения
- ArithmExpr — нетерминал содержащий некоторое арифмитическое выражение с операциями +, -
- AddOp — операторы сложения и вычитания
- Term — нетерминал содержащий некоторое арифмитическое выражение с операциями *, /
- Power — нетерминал, содержащий член некоторого выражения
- Const — один из возможных 3 типов констант

- INT_CONST — интегральная константа
- REAL_CONST — вещественная константа
- STRING_CONST — строкЛистинг 8.овая константа
- IDENTIFIER — идентификатор, являющий либо именем переменной, либо именнем функций

2.3 Лексический и Синтаксический анализ

Подробный и тщательный анализ существующих методов, инструментов и средств разработки для реализаций этапов лексического и семантического анализа привёл к выбору использования библиотеки parser_edsl.py[5]

Библиотека parser_edsl.py позволяет создавать LALR(1)-парсеры в наглядной форме: достаточно описать нетривиальные лексические домены с использованием регулярных выражений и грамматику в БНФ-подобной нотации.

В процессе разбора библиотека вычисляет атрибуты терминальных и нетерминальных символов:

- функция, вычисляющая атрибут терминального символа, передаётся в конструктор терминального символа — её аргументом является лексема,
- функция, вычисляющая атрибут нетерминального символа, записывается в правиле грамматики — её аргументами являются атрибуты символов правой части правила.

Результатом разбора является атрибут аксиомы

Библиотека не умеет восстанавливаться при ошибках при лексическом и синтаксическом анализе: исключение порождаются на первой же ошибке.

Также для абстрактного синтаксического дерева были определены синтаксические структуры (в соответствий с приложением В.)

Комбинируя правила грамматики и конструкторы синтаксический структур, с помощью данной библиотеки мы можем построить абстрактное синтаксическое дерево.

2.4 Семантический анализ

В последствий построения синтаксического дерева, проверим для каждой синтаксической структуры корректность по приведённым в разделе 2.1. правилам семантики

Проверки семантики для синтаксических структур:

1. FunctionProto — Уникальность имён параматеров и имени функций
2. FunctionDecl — Уникальность сигнатуры функций/сопрограммы в глобальной области видимости
3. FunctionDef - Уникальность сигнатуры функций/сопрограммы в глобальной области видимости
4. VariableDecl — Уникальность имени в локальной области видимости и корректность инициализаций выражением или списком инициализаций
5. FuncCallOrArrayIndex — Существование и корректность вызова функций или индексирования массива
6. FuncCall — Существование и корректность вызова функций
7. ExitFor — Существование переменной, если была указана, и внешнего блока оператора цикла со счётчиком
8. ExitWhile, ExitSubroutine, ExitFunction — Существование внешнего блока оператора цикла без счётчика\сопрограммы\функций
9. AssignStatement — Корректность присваивания переменной выражения
10. ForLoop — Уникальность имени счётчика, целочисленность счётчика, целочисленность начального и конечного значения, осуществление инкремент по объявленному счётчику, возможность привести целочисленные типы начального и конечного значения к типу счётчика

11. WhileLoop — Целочисленность условия, если оно было указано
12. IfElseStatement — Целочисленность условия
13. UnaryOpExpr — Коррекность использования унарной операций для переменной
14. BinOpExpr — Возможность приведения к общему типу, в случае неравенства типов левого и правого выражения, корректность бинарной операций

2.5 Генерация кода

В следствие необходимости в совместимости при компановке с кодом на языке программирования C и тщательного анализа существующих инструментов, библиотек и средств разработки, виртуальный ассемблер LLVM-IR[1] был выбран целевым ассемблером

2.5.1 LLVM-IR

LLVM-IR (Intermediate Representation) - это промежуточное представление, используемое в LLVM (Low Level Virtual Machine), которое представляет собой низкоуровневое представление программы. LLVM-IR является промежуточным представлением между исходным кодом на высокоуровневом языке программирования и машинным кодоЛистинг 8.м, который выполняется на конкретной аппаратной платформе.

LLVM-IR поддерживает статическую типизацию, что означает, что каждое значение имеет определенный тип данных. Это позволяет проводить проверки типов и оптимизации на этапе компиляции.

LLVM-IR может быть преобразован в машинный код для конкретной аппаратной платформы с помощью LLVM компилятора. Это позволяет выполнить программу на различных архитектурах без необходимости переписывать ее код.

LLVM-IR играет ключевую роль в экосистеме LLVM, обеспечивая единое промежуточное представление для компиляции программ на различных языках программирования и их оптимизации.

2.5.2 Библиотека поддержки времени выполнения

Библиотека поддержки времени выполнения должна реализовывать методы вывода в выходной поток простых типов и операций конкатенаций, присваивания и копирования строк.

В следствие выбора промежуточного представления LLVM, появляется возможность реализаций библиотеки поддержки времени выполнения на любом языке программирования для которого написан фронтенд LLVM и есть возможность прямого управления памятью, но наиболее простым и удобным вариантом является язык программирования C.

2.5.2 Библиотека поддержки времени выполнения

Компоновка с библиотекой поддержки времени выполнения и другими возможными модулями, написанными на C или других языках программирования, будет осуществляться с помощью утилиты llvm-link[4], входящей в состав LLVM (Low Level Virtual Machine), которая предназначена для объединения нескольких модулей LLVM-IR в один модуль.

2.7 Оптимизаций

В проект LLVM также входит утилита llvm-opt, позволяющая проводит многочисленные оптимизаций над промежуточным представлением, для генераций наиболее оптимального кода для конкретного целевого устройства

3. Программная реализация

Начальным этапом реализации задачи курсового проекта является выбор подходящего языка программирования. Посчитав наиболее удобным и практичным, был выбран Python и библеотека llvmlite[3] для генераций промежуточного представления LLVM-IR

3.1. Лексический и синтаксический анализ

Для реализаций данного этапа в первую очередь необходимо было определить список терминалов и нетерминалов (приведённых в приложений Б.) с помощью методов библиотеки parser_edsl.py и записать в БНФ-подобной форме грамматику языка, определить синтаксические структуры (приведённые в приложений В.) и их конструкторы.

```
    NExpr, NCmpOp, NArithmExpr, NAddOp, NTerm, NPower, NConst, NMulOp = \
       map(pe.NonTerminal, 'Expr CmpOp ArithmExpr AddOp Term Power Const MulOp'.split())
    NExpr |= NArithmExpr
    NExpr |= NArithmExpr, NCmpOp, NArithmExpr, BinOpExpr.create
    NArithmExpr |= NTerm
    NArithmExpr |= NAddOp, NTerm, UnaryOpExpr.create
    NArithmExpr |= NArithmExpr, NAddOp, NTerm, BinOpExpr.create
    NAddOp |= '+', lambda: '+'
    NAddOp |= '-', lambda: '-'
    NTerm |= NPower
    NTerm |= NTerm, NMulOp, NPower, BinOpExpr.create
    NMulOp |= '*', lambda: '*'
    NMulOp |= '/', lambda: '/'
    NPower |= NVarname, Variable.create
    NPower |= NConst
```

Листинг 8.

Впоследствие, результатом реализаций данного этапа является абстрактное синтаксическое дерево, следующего вида

```
    Program(decls=[SubroutineDecl(pos=Position(offset=0, line=1, col=1),
    proto=SubroutineProto(pos=Position(offset=0, line=1,col=1),
                          name=Varname(pos=Position(offset=0,line=1,col=1),
                                       name='PrintI',
                                       type=<basic_types.VoidT object at
0x7f67b0cc0790>),
                          args=[Variable(pos=Position(offset=0,line=1,col=1),
                                         name=Varname(pos=Position(offset=0,line=1,col=1),
                                                      name='val',
                                                      type=<basic_types.IntegerT object at
0x7f67b1157e20>),
                                         type=<basic_types.IntegerT object at
0x7f67b1157e20>)],
                          type=<basic_types.ProcedureT object at 0x7f67b0cc1030>),
external=True),
```

Листинг 9.

3.2. Семантический анализ

На данном этапе реализаций для каждой синтаксической структуры, был определён метод relax, производящий, в соответствий с разделом 2.4, семантическую проверку и устраняющий неопределённость между вызовом функций и обращением к массиву. Также для этого было реализованы классы ошибок с соответствующими сообщениями об ошибках и была реализована таблица символов.

- InitializationNegativeSize - Размер массива при обьявлений не может являться отрицательным
- InitializationNonConstSize - Размер не определён при инциализаций списком
- InitializationLengthMismatchError - Несоответствие размеров массива и списка инициализаций

- InitializationTypeError - Массив инициализируется выражением
- InitializerListDimensionMismatch - Не совпадение размерностей списков инициализаций
- InappropriateInitializerList — Встречены выражения и списки инициализаций в списке инициализаций
- ConversionError - Невозможно преобразовать типы
- InappropriateExit - Неопределённый exit
- UndefinedFunction - Функция не определена
- UndefinedSymbol - Символ не определён
- RedefinitionError - Повторное объявление символа
- UnexpectedNextFor - Ожидался счётчик у инкремента цикла for
- NotIntFor — Не целый тип счётчика
- ArrayNotIntInit - Не целый тип размерности массива при инициализаций
- WhileNotIntCondition — Условие цикла нецелочисленного типа
- IfNotIntCondition - Условие нецелочисленного типа
- ArrayIndexingDimensionMismatchError - Размерность массива меньше, чем список индексирования
- ArrayNotIntIndexing - Массив индексируется не целочисленным типом
- BinBadType - Несовместимые типы при бинарной операций
- UnaryBadType - Неопределённая унарная операция для данного типа

3.3. Библиотека поддержки времени выполнения

На этапе разработки библиотеки поддержки времени выполнения была написана программа на языке программирования C, реализующая базовые методы описанные в разделе 2.5.2, в качестве строкового литерала был выбран wchar_t для возможности вывода строк, содержащих не ASCII символы

```
#include <cstdio>
#include <cstring>
#include <cstdlib>
#include <cwchar>
#include <clocale>
extern "C" void PrintI(int val){ printf("%d", val); }
extern "C" void PrintL(long val){ printf("%ld", val); }
extern "C" void PrintD(double val){ printf("%lf", val); }
extern "C" void PrintF(float val){ printf("%f", val); }
extern "C" void PrintS(wchar_t* val){ printf("%ls", val); }
extern "C" wchar_t* StringConcat(wchar_t* lhs, wchar_t* rhs){
   unsigned long __sz_lhs = wcslen(lhs), __sz_rhs = wcslen(rhs);
   lhs = (wchar_t*)reallocarray(lhs, __sz_lhs + __sz_rhs + 1, sizeof(wchar_t));
   return wcscat(lhs, rhs);
}
extern "C" wchar_t* StringCopy(wchar_t* str){
   unsigned long __sz = wcslen(str);
   wchar_t* result = (wchar_t*)calloc(__sz + 1, sizeof(wchar_t));
   wcsncpy(result, str, __sz);
   return result;
}
```

Листинг 10.

Также определена функций Main являющаяся точкой входа программы написанной на разрабатываемом языке

```
extern "C" wchar_t* Main(wchar_t** argv, int len);
int main(int argc, char** argv){
   setlocale(LC_ALL, "C.UTF-8");
   wchar_t** args = (wchar_t**)calloc(argc, sizeof(wchar_t*));
   for(int idx = 0; idx < argc; idx++){
      unsigned long __sz = strlen(argv[idx]);
      args[idx] = (wchar_t*) calloc(__sz + 1,sizeof(wchar_t));
      for(int row_idx = 0; row_idx < __sz; row_idx++)
         args[idx][row_idx] = argv[idx][row_idx];
   }
   Main(args, argc);
   for(int idx = 0; idx < argc; idx++) free(args[idx]);
   free(args);
   return 0;
}
```

Листинг 11.

3.4. Генерация кода

На этапе генераций кода для каждой проверенной на предыдущем этапе синтаксической структуры был реализован метод codegen, выполняющий кодогенерацию в промежуточное представление с помощью библиотеки llvmlite

Примеры кодогенераций различных программ (графы были построены с помощью сайта godbolt[2]):

Объявление переменной

```
Function Join$()
    i% = 0
End Function
```

```
define i32* @"Join"(){
entry:
  %"Join" = alloca i32*, i32 1
  store i32* null, i32** %"Join"
  %"iI" = alloca i32, i32 1
  store i32 0, i32* %"iI"
  br label %"return"
return:
  %".11" = load i32*, i32** %"Join"
  ret i32* %".11"
}
```

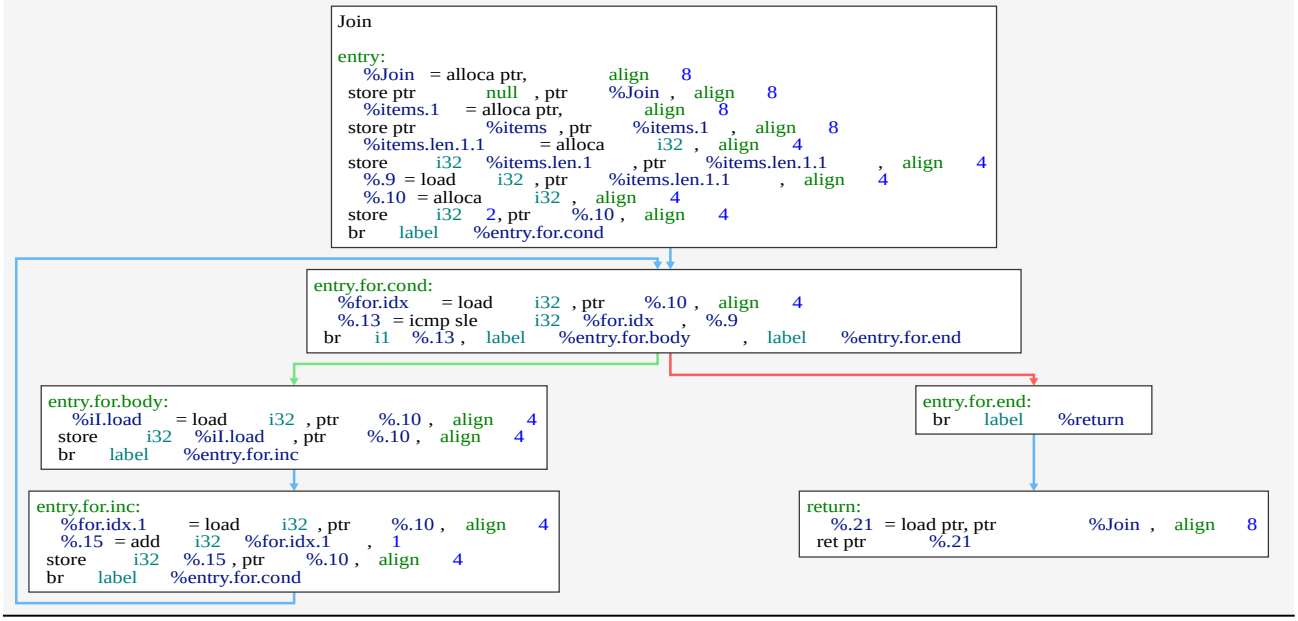
Объявление функций

```
Function Join$(sep$, items$())
End Function
```

```
define i32* @"Join"(i32* %"sep", i32** %"items", i32 %"items.len.1")
{
entry:
  %"Join" = alloca i32*, i32 1
  store i32* null, i32** %"Join"
  %"sep.1" = alloca i32*, i32 1
  store i32* %"sep", i32** %"sep.1"
  %"items.1" = alloca i32**, i32 1
  store i32** %"items", i32*** %"items.1"
  %"items.len.1.1" = alloca i32, i32 1
  store i32 %"items.len.1", i32* %"items.len.1.1"
  br label %"return"
return:
  %".10" = load i32*, i32** %"Join"
  ret i32* %".10"
}
```

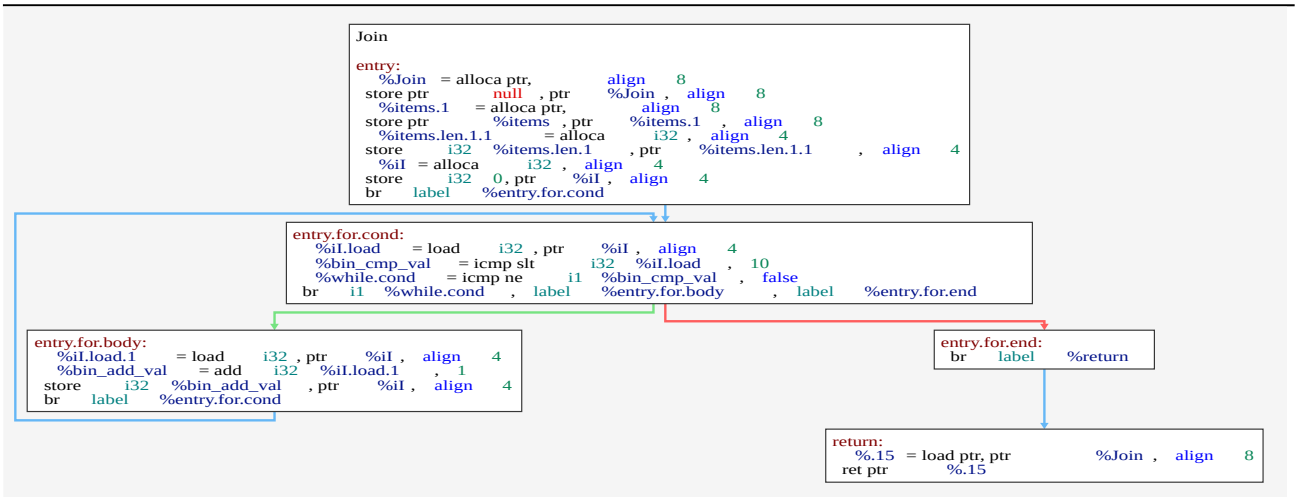
Цикл со счётчиком

```
Function Join$(items$())
    For i% = 2 To Len%(items$)
        i% = i%
    Next i%
End Function
```



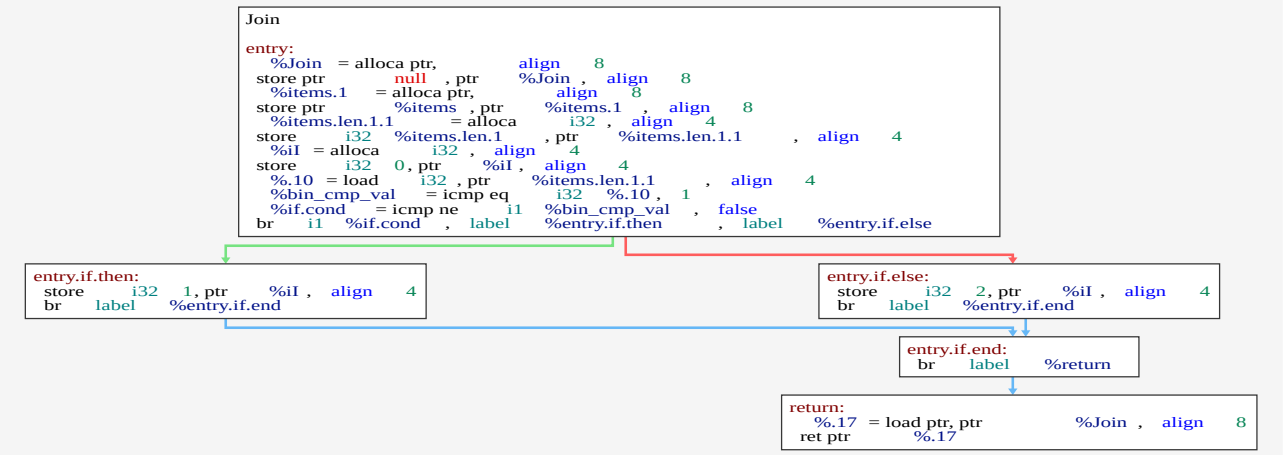
Цикл без счётчика

```
Function Join$(sep$, items$())
    i% = 0
    Do While i% < 10
        i% = i% + 1
    Loop
End Function
```



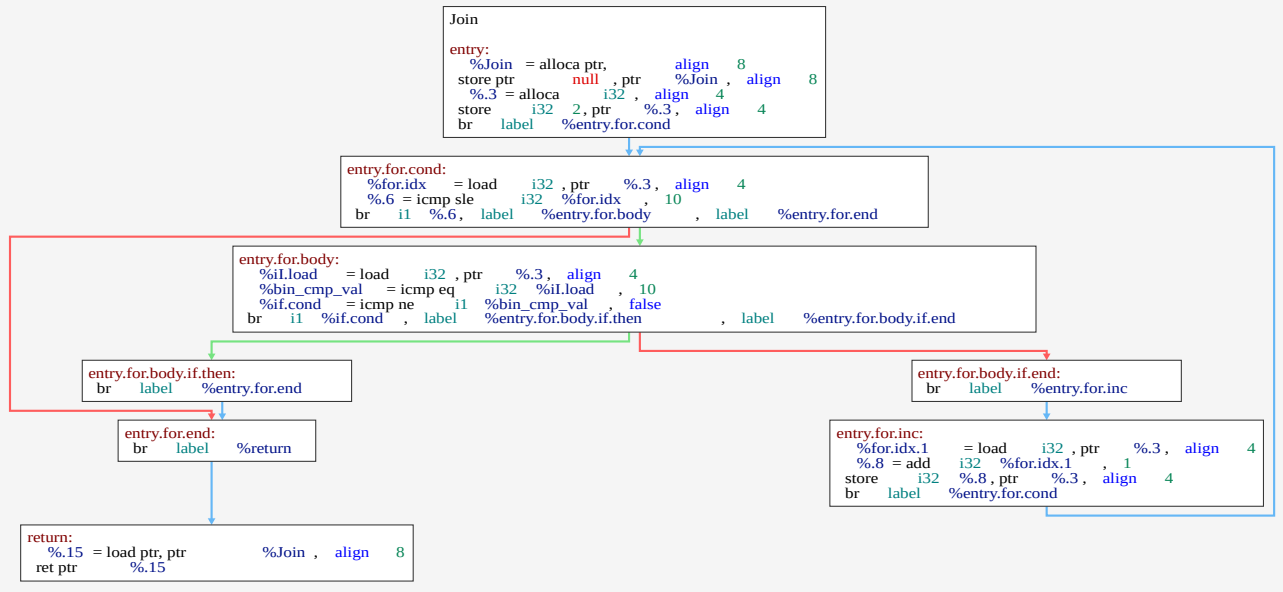
Оператор ветвления

```
Function Join$(sep$, items$())
   i% = 0
   If Len%(items$) = 1 Then  i% = 1
   Else i% = 2
   End If
End Function
```



Оператор возврата из цикла со счётчиком

```
Function Join$()
   For i% = 2 To 10
      If i% = 10 Then
         Exit for i%
      End If
   Next i%
End Function
```



Объявление функций без тела

```
Declare Function Join$(sep$, items$())
```

```
declare i32* @"Join"(i32* %"sep", i32** %"items", i32 %"items.len.1")
```

Использование функций библиотеки поддержки времени выполнения

```
Function Join$(items$())
   c$ = items$(1) + items$(2)
   Print c$
End Function
```

```
Join

entry:
   %Join = alloca ptr, align 8
  store ptr null, ptr %Join, align 8
   %items.1 = alloca ptr, align 8
  store ptr %items, ptr %items.1, align 8
   %items.len.1.1 = alloca i32, align 4
  store i32 %items.len.1, ptr %items.len.1.1, align 4
   %cS = alloca ptr, align 8
   %idx_sub = sub i32 1, 1
   %gep_load = load ptr, ptr %items.1, align 8
   %gep_idx = getelementptr inbounds ptr, ptr %gep_load, i32 %idx_sub
   %gep_idx_load = load ptr, ptr %gep_idx, align 8
   %idx_sub.1 = sub i32 2, 1
   %gep_load.1 = load ptr, ptr %items.1, align 8
   %gep_idx.1 = getelementptr inbounds ptr, ptr %gep_load.1, i32 %idx_sub.1
   %gep_idx_load.1 = load ptr, ptr %gep_idx.1, align 8
   %str_concat_val = call ptr @StringConcat(ptr %gep_idx_load, ptr %gep_idx_load.1)
  store ptr %str_concat_val, ptr %cS, align 8
   %cS.load = load ptr, ptr %cS, align 8
  call void @PrintS(ptr %cS.load)
  br label %return

return:
   %.10 = load ptr, ptr %Join, align 8
  ret ptr %.10
```

4. Тестирование

Исходными тестовыми данными для функционального тестирования является текст корректной программы приведённой в приложений А.

Результат работы скомпилированной и скомпанованной программы с аргументами {Hello, World} :

```
Аргументы программы: ./a.out, Hello, World
50-е число Фибоначчи — 12586269025
Таблица значений функции y = x³ + x² + x + 1:
x = -50, y = -122549.000000
x = -49, y = -115296.000000
x = -48, y = -108335.000000
x = -47, y = -101660.000000
x = -46, y = -95265.000000
x = -45, y = -89144.000000
....
x = 49, y = 120100.000000
x = 50, y = 127551.000000
Сумма перечисленных значений y: 85951.000000
```

Фактический результат работы совпадает программы совпадает с ожидаемым

З А К Л Ю Ч Е Н И Е

В ходе выполнения курсового проекта был разработан компилятор диалекта языка программирования BASIC в целевой ассемблер виртуальной машины LLVM.

Для проектирования и разработки компилятора были использованы утилиты llvm-link и opt входящие в состам LLVM, библиотека построения LALR(1) парсера parser_edsl.py, библиотека для написания промежуточного представления - LLVM-IR llvmlite

Выполненное тестирование компилятора, с использованием заготовленных тестовых данных, привело к выводу, что приложение не содержит ошибок и работает в соответствии с заданными функциональными требованиями.

Результаты выполненной работы свидетельствуют о успешной реализации поставленных задач и достижении поставленных целей. Созданный компилятор Бейсика в ассемблер представляет собой функциональный инструмент, способный эффективно транслировать программы на Бейсике в промежуточное представление — LLVM-IR, обеспечивая высокую производительность и надежность.

В ходе выполнения курсового проекта были систематизированы и закреплены теоретические и практические знания в области проектирования и разработки компиляторов.

Цель курсового проекта выполнена, все поставленные задачи для выполнения задания выполнены.

СПИСОК ЛИТЕРАТУРЫ

1 LLVM: LLVM Language Reference: [сайт]. – По всему миру, 2024. – URL: https://llvm.org/docs/LangRef.html (дата обращения 27.02.2024). – Текст. Изображение: электронные.

2 godbolt: Compiler Explorer: [сайт]. – По всему миру, 2024. – URL: https://godbolt.org/ (дата обращения 27.02.2024). – Текст. Изображение: электронные.

3 Llvmlite: llvmlite — llvmlite documentation: [сайт]. – Москва, 2024. – URL: https://llvmlite.readthedocs.io/en/latest/index.html (дата обращения 27.02.2024). – Текст. Изображение: электронные.

4 llvm-link: llvm-link - LLVM bitcode linker БД: [сайт]. – Москва, 2024. – URL: https://llvm.org/docs/CommandGuide/llvm-link.html (дата обращения 28.02.2024). – Текст. Изображение: электронные.

5 parser_edsl: Курсовая работа Даниэлы Обущаровой 2023: [сайт]. – Москва, 2024. – URL: https://github.com/bmstu-iu9/parser_edsl_python (дата обращения 23.02.2024). – Текст. Изображение: электронные.

6 Llvm-opt: opt - LLVM optimizer: [сайт]. – Москва, 2024. – URL: https://llvm.org/docs/CommandGuide/opt.html (дата обращения 29.02.2024). – Текст. Изображение: электронные.

ПРИЛОЖЕНИЕ А

Пример корректной програмы на диалекте BASIC

```
' Суммирование элементов массива
Function SumArray#(Values#())
   SumArray# = 0
   For i% = 1 To Len%(Values#)
      SumArray# = SumArray# + Values#(i%)
   Next i%
End Function
' Вычисление многочлена по схеме Горнера
Function Polynom!(x!, coefs!())
   Polynom! = 0
   For i% = 1 to Len%(coefs!)
      Polynom! = Polynom! * x! + coefs!(i%)
   Next i%
End Function
' Вычисление многочлена x³ + x² + x + 1
Function Polynom1111!(x!)
   Dim coefs!(4)
   For i% = 1 To 4
      coefs!(i%) = 1
   Next i%
   Polynom1111! = Polynom!(x!, coefs!)
End Function
' Инициализация массива числами Фибоначчи
Sub Fibonacci(res&())
   n% = Len%(res&)
   If n% >= 1 Then
      res&(1) = 1
   End If
   If n% >= 2 Then
      res&(2) = 1
   End If
   i% = 3
   Do While i% <= n%
      res&(i%) = res&(i% - 1) + res&(i% - 2)
      i% = i% + 1
   Loop
End Sub
```

```
' Склеивание элементов массива через разделитель: Join$(", ", words)
Function Join$(sep$, items$())
   If Len%(items$) >= 1 Then
      Join$ = items$(1)
   Else
      Join$ = ""
   End If
   For i% = 2 To Len%(items$)
      Join$ = Join$ + sep$ + items$(i%)
   Next i%
End Function

' Главная процедура
Sub Main(args$())
   Print "Аргументы программы: ", Join$(", ", args$), "\n"
   ' Объявление локального массива
   Dim fibs&(100)
   Fibonacci(fibs&)
   Print "50-е число Фибоначчи — ", fibs&(50), "\n"
   Dim ys#(101)
   Print "Таблица значений функции y = x³ + x² + x + 1:\n"
   For x% = -50 To 50
      y! = Polynom1111!(x%)
      Print "x = ", x%, ", y = ", y!, "\n"
      ys#(x% + 51) = y!
   Next x%
   Print "Сумма перечисленных значений y: ", SumArray#(ys#), "\n"
End Sub
```

ПРИЛОЖЕНИЕ Б

Грамматика диалекта языка программирования BASIC

INT_CONST ::= [0-9]+
REAL_CONST ::= [0-9]+"."([0-9]*)?(e[-+]?[0-9]+)?
STRING_CONST ::= "\""[^\"]*"\""
IDENTIFIER ::= [A-Za-z][A-Za-z0-9_]*
<Program> ::= <GlobalSymbols>
<GlobalSymbols> ::= <GlobalSymbol> <GlobalSymbols> | eps
<GlobalSymbol> ::= <SubroutineDecl> | <SubroutineDef> | <FunctionDecl> | <FunctionDef> | <VariableDecl>
<SubroutineProto> ::= SUB IDENTIFIER "(" <ParametersList> ")"
<FunctionProto> ::= FUNCTION <Varname> "(" <ParametersList> ")"
<SubroutineDef> ::= <SubroutineProto> <Statements> END SUB
<FunctionDef> ::= <FunctionProto> <Statements> END FUNCTION
<SubroutineDecl> ::= DECLARE <SubroutineProto>
<FunctionDecl> ::= DECLARE <FunctionProto>
<VariableDecl> ::= DIM <VarnameOrArrayArg> <VariableInit>
<VariableInit> ::= "=" <Expr> | "=" <InitializerList> | eps
<InitializerList> ::= "{" <InitializerListValues> "}"
<InitializerListValues> ::= <InitializerListValue>","<InitializerListValues> | <InitializerListValue>
<InitializerListValue> ::= <Expr> | <InitializerList>
<ParametersList> ::= <VarnameOrArrayParam>","<ParametersList> | <VarnameOrArrayParam> | eps
<NonEmptyParametersList>::=<VarnameOrArrayParam><NonEmptyParametersList> | <VarnameOrArrayParam>
<VarnameOrArrayParam> = <Varname> | <Varname>"("<CommaList>")"
<CommaList> ::= ","<CommaList> | eps
<ArgumentsList> ::= <Expr>","<NonEmptyArgumentsList> | <Expr> | eps
<NonEmptyArgumentsList> ::= <Expr>","<NonEmptyArgumentsList> | <Expr>
<VarnameOrArrayArg> = <Varname> | <Varname>"("<NonEmptyArgumentsList>")" | <Varname>"("<CommaList>")"
<FuncCallOrArrayIndex> ::= <Varname>"(" <ArgumentsList> ")" | IDENTIFIER"(" <ArgumentsList> ")"
<FuncCall> ::= PRINT <ArgumentsList>
<Varname> ::= IDENTIFIER <Type>
<Type> ::= "%" | "&" | "!" | "#" | "$"
<Statements> ::= <Statement> <Statements> | <ExitStatement> | <Statement> | eps
<NonEmptyStatements> ::= <Statement> <NonEmptyStatements> | <ExitStatement> | <Statement>
<Statement> ::= <VariableDecl> | <AssignStatement> | <FuncCallOrArrayIndex> | <FuncCall> |

<Loop> | <IfStatement>
<AssignStatement> ::= <Varname>"="<Expr> | <FuncCallOrArrayIndex>"="<Expr>
<IfStatement> ::= IF <Expr> Then <Statements> <ElseStatement> END IF
<ExitStatement> ::= EXIT FOR | EXIT FOR <Varname> | EXIT DO | EXIT LOOP | EXIT SUB | EXIT FUNCTION
<ElseStatement> ::= ELSE <Statements> | eps
<Loop> ::= <ForLoop> | <WhileLoop>
<ForLoop> ::= FOR <Varname> = <Expr> TO <Expr> <Statements> NEXT <Varname>
<WhileLoop> ::= DO <PreOrPostLoop>
<PreOrPostLoop> ::= WHILE <PreLoop> | UNTIL <PreLoop> | <PostLoop>
<PreLoop> ::= <Expr> <Statements> LOOP
<PostLoop> ::= <Statements> LOOP <PostLoopExpr>
<PostLoopExpr> ::= WHILE <Expr> | UNTIL <Expr> | eps
<Expr> ::= <ArithmExpr> | <ArithmExpr> <CmpOp> <ArithmExpr>
<CmpOp> ::= > | < | >= | <= | = | <>
<ArithmExpr> ::= <Term> | <AddOp> <Term> | <ArithmExpr> <AddOp> <Term>
<AddOp> ::= + | -
<Term> ::= <Power> | <Term> * <Power> | <Term> / <Power>
<Power> ::= <Varname> | <Const> | (<Expr>) | <FuncCallOrArrayIndex>
<Const> ::= INT_CONST | REAL_CONST | STRING_CONST

ПРИЛОЖЕНИЕ В

Синтаксические структуры

```
class Expr(abc.ABC):
    pass
class Statement(abc.ABC):
    pass
@dataclass
class Varname:
    pos: pe.Position
    name: str
    type: Type
@dataclass
class ConstExpr(Expr):
    pos: pe.Position
    value: typing.Any
    type: Type
@dataclass
class Variable:
    pos: pe.Position
    name: Varname
    type: Type
@dataclass
class Array(Variable):
    size: list[Expr]
@dataclass
class FunctionProto:
    name: Varname
    args: list[Variable]
    type: ProcedureT
@dataclass
class FunctionDecl:
    proto: FunctionProto
    external: bool
@dataclass
class FunctionDef:
    proto: FunctionProto
    body: list[Statement]
class InitializerList:
    values: list[Expr | list]
    type: Type
@dataclass
class VariableDecl:
    variable: Variable
    init_value: typing.Optional[Expr | InitializerList | None]
@dataclass
class FuncCallOrArrayIndex:
    name: Varname
    args: list[Expr]
@dataclass
class FuncCall:
    name: Varname
    args: list[Expr]
    type: Type
```

```
@dataclass
class ExitFor(ExitStatement):
    name: Variable | None
@dataclass
class ExitWhile(ExitStatement):
    pass
@dataclass
class ExitSubroutine(ExitStatement):
    pass
@dataclass
class ExitFunction(ExitStatement):
    pass
@dataclass
class AssignStatement(Statement):
    variable: Variable | FuncCallOrArrayIndex
    expr: Expr
@dataclass
class ForLoop(Statement):
    variable: Variable
    start: Expr
    end: Expr
    body: list[Statement]
    next: Variable
@dataclass
class WhileLoop(Statement):
    condition: Expr | None
    body: list[Statement]
    type: WhileType
@dataclass
class IfElseStatement(Statement):
    condition: Expr
    then_branch: list[Statement]
    else_branch: list[Statement]
@dataclass
class UnaryOpExpr(Expr):
    op: str
    unary_expr: Expr
@dataclass
class BinOpExpr(Expr):
    left: Expr
    op: str
    right: Expr
@dataclass
class Program:
    decls: list[SubroutineDecl | FunctionDecl | SubroutineDef | FunctionDef | VariableDecl]
    symbol_table: SymbolTable
```

ПРИЛОЖЕНИЕ Г

Программный код

Функция семантической проверки объявления переменной

```
   def relax(self, symbol_table: SymbolTable):
      lookup_result = symbol_table.lookup(self.variable.symbol(), local=True, by_type=False,
by_origin=False)
      if lookup_result:
         raise RedefinitionError(self.pos, self.variable.name, lookup_result.name.pos)
      if self.init_value:
         self.init_value = self.init_value.relax(symbol_table)
         if isinstance(self.variable, Array):
            if not self.init_value.type.castable_to(self.variable.type.type):
               raise ConversionError(self.pos, self.init_value.type, self.variable.type.type)
            if isinstance(self.init_value, Expr):
               raise InitializationTypeError(self.pos, self.variable.type, self.init_value.type)
            else:
               self.variable = self.variable.relax(symbol_table)
               self.variable.size = self.__check_sizes(self.variable.size, self.init_value.size())
               self.variable.type.size = self.variable.size
               symbol_table.add(self.variable.symbol())
         else:
            if not self.init_value.type.castable_to(self.variable.type):
               raise ConversionError(self.pos, self.init_value.type, self.variable.type)
            if isinstance(self.init_value, InitializerList):
               sz = self.init_value.size()
               self.variable = Array(self.variable.pos, self.variable.name, ArrayT(self.variable.type, sz), sz)
               symbol_table.add(self.variable.symbol())
            else:
               if isinstance(self.init_value, ConstExpr) and self.variable.type != self.init_value.type:
                  self.init_value = ConstExpr(self.init_value.pos, self.init_value.value, self.variable.type)
               symbol_table.add(self.variable.symbol())
      else:
         if isinstance(self.variable, Array):
            if isinstance(self.variable.size[0], int):
               raise InitializationUndefinedLengthError(self.pos, self.variable.size)
            for idx, sz in enumerate(self.variable.size):
               if not (isinstance(sz, ConstExpr) and isinstance(sz.type, IntegralT)):
                  raise InitializationUndefinedLengthError(self.pos, self.variable.size)
               self.variable.size[idx] = self.variable.size[idx].value
         symbol_table.add(self.variable.symbol())
         self.variable = self.variable.relax(symbol_table)
      return self
```

Функция кодогенераций бинарных операций

```
    def codegen(self, symbol_table: SymbolTable, lvalue: bool = True):
        lhs_val = self.left.codegen(symbol_table, False)
        rhs_val = self.right.codegen(symbol_table, False)
        result_val = None
        if self.type == StringT():
            str_concat_symbol = Program.string_concat_symbol()
            lookup_result = symbol_table.lookup(str_concat_symbol, by_type=False)
            if isinstance(symbol_table.llvm.builder, ir.IRBuilder):
                result_val = symbol_table.llvm.builder.call(lookup_result.llvm_obj, [lhs_val, rhs_val],
"str_concat_val")
        elif isinstance(self.left.type, NumericT) and isinstance(self.right.type, NumericT):
            if self.op in ('>', '<', '>=', '<=', '=', '<>'):
                result_val = self.type.cmp(self.op, symbol_table.llvm.builder, "bin_cmp_val")(lhs_val, rhs_val)
            else:
                if self.left.type != self.type:
                    lhs_val = self.left.type.cast_to(self.type, symbol_table.llvm.builder)(lhs_val)
                if self.right.type != self.type:
                    rhs_val = self.right.type.cast_to(self.type, symbol_table.llvm.builder)(rhs_val)
                if self.op == "+":
                    result_val = self.type.add(symbol_table.llvm.builder, "bin_add_val")(lhs_val, rhs_val)
                elif self.op == "-":
                    result_val = self.type.sub(symbol_table.llvm.builder, "bin_sub_val")(lhs_val, rhs_val)
                elif self.op == "*":
                    result_val = self.type.mul(symbol_table.llvm.builder, "bin_mul_val")(lhs_val, rhs_val)
                elif self.op == "/":
                    result_val = self.type.div(symbol_table.llvm.builder, "bin_div_val")(lhs_val, rhs_val)
        return result_val
```

Класс таблицы символов

```
class SymbolTable:

    def __init__(self, parent=None,
                 block_type: SymbolTableBlockType = SymbolTableBlockType.Undefined,
                 block_obj: object = None,
                 llvm_entry: SymbolTableLLVMEntry = None):
        self.parent = parent
        self.table = []
        self.children = []
        self.block_type = block_type
        self.block_obj = block_obj
        self.llvm = llvm_entry

    def add(self, symbol: Symbol):
```

```
        lookup_local_result = self.lookup(symbol, local=True)
        if lookup_local_result:
            raise RedefinitionError(symbol.name.pos, symbol.name, lookup_local_result.name.pos)
        self.table.append(symbol)

    def lookup(self, symbol: Symbol, local=False, by_name=True, by_type=True, by_origin=True,
accumulate=False) \
            -> list[Symbol] | Symbol | None:
        current = self
        lookup_result = []
        while current:
            for current_symbol in current.table:
                result = True
                if by_name:
                    result &= symbol.name == current_symbol.name
                if by_type:
                    result &= symbol.type == current_symbol.type
                if by_origin:
                    result &= symbol.external == current_symbol.external
                if result:
                    if accumulate:
                        lookup_result += [current_symbol]
                    else:
                        return current_symbol
            if local:
                if accumulate:
                    return None if len(lookup_result) == 0 else lookup_result
                else:
                    return None
            current = current.parent
        if accumulate:
            return None if len(lookup_result) == 0 else lookup_result
        else:
            return None
```