|  |  |
| --- | --- |
| Сине-желтый щит со звездой и золотой звездой  Описание генерируется автоматически | **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 Информатики, искусственного интеллекта и систем управления

КАФЕДРА Теоретической информатики и компьютерных технологий

**РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

***К КУРСОВОЙ РАБОТЕ***

***НА ТЕМУ:***

***Компилятор диалекта Бейсика в ассемблер***

Студент ИУ9И-71Б **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Нгуен В.Т.**

(Группа) (Подпись, дата) (И.О.Фамилия)

Руководитель курсовой работы **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Коновалов А. В.**

(Подпись, дата) (И.О.Фамилия)

*Москва, 2025 г.*

ОГЛАВЛЕНИЕ

[Введение 3](#_Toc190313825)

[1. Теоретическая часть 5](#_Toc190313826)

[**1.1 Введение в язык BASIC и его характеристики** 5](#_Toc190313827)

[**1.2 Основные шаги по сборке компилятора** 9](#_Toc190313828)

[**1.2.1. Анализ лексики** 9](#_Toc190313829)

[**1.2.2. Парсинг (построение дерева разбора)** 10](#_Toc190313830)

[**1.2.3. Семантический анализ (построение абстрактного синтаксического дерева)** 12](#_Toc190313831)

[**1.2.4. Сборка транслятора в ассемблерный код** 14](#_Toc190313832)

[**1.2.5. Построение макета** 16](#_Toc190313833)

[**1.2.6. Заключение** 17](#_Toc190313834)

[2. Компоновка 18](#_Toc190313835)

[**2.1 Выбор языка программирования** 18](#_Toc190313836)

[**2.2 Грамматический анализ основного языка** 18](#_Toc190313837)

[**2.3 Синтаксический анализ и лексический анализ языка** 21](#_Toc190313838)

[**2.4 Семантический анализ** 23](#_Toc190313839)

[**2.5 Создание промежуточного кода** 26](#_Toc190313840)

[**2.6 Библиотека поддержки времени выполнения** 27](#_Toc190313841)

[**2.7 Оптимизация** 28](#_Toc190313842)

[3. Результаты 28](#_Toc190313843)

[Заключение 31](#_Toc190313844)

[Литература 32](#_Toc190313845)

[ПРИЛОЖЕНИЕ 33](#_Toc190313846)

# **Введение**

**Зачем создавать компилятор, его роль**

Компилятор является незаменимым инструментом в области информационных технологий, помогающим преобразовывать исходный код из языков программирования высокого уровня в машинный код или промежуточный код, который компьютеры могут понимать и выполнять. Создание компилятора направлено не только на эффективное использование языков программирования, но и на расширение возможностей оптимизации исходного кода, повышение производительности и минимизацию ошибок при разработке программного обеспечения.

Компилятор действует как мост между программистом и компьютерным оборудованием. Это помогает:

• Автоматизировать преобразование исходного кода, сводя к минимуму ручное вмешательство.

• Оптимизировать аппаратные ресурсы и производительность программ.

• Создать высокопортативное программное обеспечение, которое может легко работать на нескольких платформах.

**Обзор построения компилятора языка BASIC в LLVM-IR**

Язык BASIC известен своим простым синтаксисом, прост в изучении и подходит для создания базовых приложений. В сочетании с LLVM, мощной инфраструктурой компиляции, создание компилятора из BASIC в LLVM-IR обеспечивает глубокие возможности оптимизации, позволяя программам BASIC эффективно работать на различных платформах.

Этот процесс включает в себя:

• Анализ исходного кода BASIC для создания абстрактного синтаксического дерева (AST).

• Преобразовать AST в промежуточное представление LLVM-IR.

• Оптимизировать промежуточный код и сгенерировать  машинный код для эффективного выполнения.

Интеграция BASIC и LLVM не только использует простоту BASIC, но и обеспечивает вычислительную мощность и оптимизацию LLVM.

**Цели и основные этапы курса**

Курс направлен на предоставление фундаментальных знаний и практических навыков по созданию компиляторов. Основные цели включают в себя:

• Понимать основные концепции лексического, синтаксического и семантического анализа.

• Понимать процесс преобразования исходного кода в промежуточное представление (IR).

• Выполнять оптимизацию и генерацию целевого кода, обеспечивая эффективность и точность.

Основные этапы курса:

1. Анализ исходного кода: сломать исходный код на его основные компоненты и проверить их на валидность.

2. Построить абстрактное синтаксическое дерево (AST): представить программу с помощью логической древовидной структуры.

3. Генерация промежуточного кода (LLVM-IR): преобразование AST в промежуточный код, совместимый с LLVM.

4. Оптимизация кода: повышение производительности программы за счет удаления избыточных или неэффективных частей.

5. Генерация машинного кода: генерация исполняемого кода, подходящего для целевой платформы.

Курс не только дает теоретические знания, но и отрабатывает шаги по созданию полноценного компилятора, помогая студентам глубже понять структуру и принципы работы языков программирования и компьютерных систем.

# **1. Теоретическая часть**

## **1.1 Введение в язык BASIC и его характеристики**

**Развитие языка BASIC**

Язык BASIC (Beginner's All-purpose Symbolic Instruction Code) был разработан в 1964 году Джоном Г. Кемени и Томасом Э. Курцем в Дартмутском колледже с целью предоставления простого и легкого в освоении инструмента программирования для пользователей, не занимающихся компьютерными науками. BASIC прошел несколько этапов развития:

1. Начальный период (1964-1970):

BASIC родился как язык для преподавания и работы в Дартмутской системе разделения времени (DTSS). Это большой шаг вперед для того, чтобы приблизить программирование к областям, отличным от информатики.

1. Популярный период на персональных компьютерах (1970-1980):

BASIC стал самым популярным языком программирования на персональных компьютерах, таких как Altair 8800, Apple II, Commodore 64 и подобных системах. Microsoft BASIC, разработанный Биллом Гейтсом и Полом Алленом, сыграл ключевую роль в внедрении BASIC в популярные операционные системы, такие как CP/M и MS-DOS.

1. Период совершенствования и расширения (1980-1990 гг.):

В этот период BASIC был разработан со многими расширенными версиями, поддерживающими структурное программирование и графические пользовательские интерфейсы (GUI). Visual Basic, запущенный в 1991 году компанией Microsoft, является одним из заметных достижений, интегрируя поддержку объектно-ориентированного программирования и создания графических приложений.

1. Период рецессии и историческая роль (с 1990 года по настоящее время):

С развитием современных языков программирования, таких как Python, Java и C++, BASIC постепенно утратил свою популярность. Тем не менее, он до сих пор признан одним из исходных языков, вносящим важный вклад в универсализацию программирования во всем мире.

**Известные редакции BASIC**

1. Dartmouth BASIC (1964): Оригинальная версия, ориентированная на простоту, работающая по системе DTSS.
2. Microsoft BASIC (1975): Одна из самых популярных версий, разработанная для работы на системах CP/M, MS-DOS и ставшая стандартом на многих ранних персональных компьютерах.
3. GW-BASIC (1983): Вариант Microsoft BASIC, который поддерживает множество новых функций и интегрирует базовую среду разработки для персональных компьютеров.
4. QBASIC (1991): Улучшенная версия с более удобным интерфейсом, поддержкой структурного программирования, популярная на MS-DOS.
5. Visual Basic (1991): Одна из самых успешных версий, оптимизированная для создания графических приложений, поддержки объектно-ориентированного программирования и интеграции разработки графических пользовательских интерфейсов.
6. FreeBASIC (2004): Современная версия с открытым исходным кодом, совместимая с QBASIC и поддерживающая функции структурного программирования и объектно-ориентированного программирования.

**Значение и роль BASIC**

Язык BASIC является не только эффективным инструментом обучения, но и играет важную роль в популяризации программирования для всех классов пользователей, от студентов до исследователей. Разработка нескольких версий доказала его универсальность и способность адаптироваться к различным технологическим требованиям на протяжении всей истории.

Сегодня, несмотря на то, что он уже не так популярен, как раньше, изучение и применение BASIC, например, при компиляции в LLVM-IR, по-прежнему является способом сохранения наследия и изучения потенциальных ценностей этого языка в контексте современных технологий.

**Отличительные особенности BASIC:**

1. Простой синтаксис:

BASIC разработан с близким и простым для понимания синтаксисом, подходящим для тех, кто новичок в программировании. Например, в командах часто используются визуальные ключевые слова, такие как PRINT, INPUT, IF...THEN.

1. Портативные и популярные:

BASIC может быть развернут на различных платформах, от персональных компьютеров до встраиваемых систем. Различные версии BASIC, такие как QBASIC, Visual Basic, были разработаны для удовлетворения различных потребностей приложений.

1. Поддержка линейного программирования:

BASIC изначально был разработан для поддержки линейного программирования, что облегчает пользователям понимание хода программы. В более поздних версиях была добавлена поддержка структурного программирования и объектно-ориентированного программирования.

1. Удобство для пользователя:

Язык подчеркивает простоту изучения и удобство для пользователя, предоставляя множество инструментов для поддержки проверки ошибок и отладки.

1. Широкое применение:

BASIC широко используется в образовании и простых приложениях. Хотя он уже не так популярен, как раньше, он по-прежнему является основой для современных языков программирования, таких как Visual Basic.

Комментарии начинаются с апострофа ' и продолжаются до конца строки.

Идентификаторы и ключевые слова не чувствительны к регистру.

Имя каждой переменной и каждой функции указывает на ее тип (%—целое число, &—длинное целое число, ! — действительное число одинарной точности, # — реальность двойной точности, $ — строка).

Внутри функции неявно объявляется переменная с тем же именем, что и имя этой функции.

function – его значение является возвращаемым значением функции.

И индексация массивов, и вызов функции записываются при помощи круглых скобок — отличить одно от другого на этапе синтаксического анализа невозможно, поэтому в дереве они различаться не должны.

Цикл while может быть записан пятью способами:

Слова крупным планом

Описание генерируется автоматически

Первые две формы представляют собой циклы с предпосылками (положительными и отрицательными),

Два других являются постусловными, а последний — пятым, представляющим собой бесконечный цикл.

Цикл For можно прерывать операторами

Слова крупным планом

Описание генерируется автоматически

Первая форма разрывает текущий цикл, вторая форма (с переменными) позволяет разбивать сразу несколько вложенных циклов.

Цикл Do/Loop может быть прерван одним из следующих двух операторов (они являются синонимами):

Слова крупным планом

Описание генерируется автоматически

Процедуры и функции завершаются операторами соответственно.

Крупный план белого фона

Описание генерируется автоматически

Вы можете определить глобальную переменную:

Черный текст на белом фоне

Описание генерируется автоматически

Вы можете определить многомерный массив:



## **1.2 Основные шаги по сборке компилятора**

Сборка компилятора — это последовательный процесс, состоящий из основных этапов, каждый из которых играет решающую роль в преобразовании исходного кода с языка высокого уровня в целевой код. Первым этапом в этом процессе является лексический анализ, где исходный код обрабатывается для выявления основных компонентов языка.

### **1.2.1. Анализ лексики**

Лексический анализ (Lexical Analysis**)**  – это первый этап работы компилятора, перед которым ставится задача прочитать исходный код и преобразовать его в строку лексем – базовых единиц словарного запаса, определенных на основе синтаксиса языка программирования.

Задачи анализа словарного запаса:

Распознавание словаря: Разбить исходный код на основные компоненты, такие как ключевые слова, операторы, знаки препинания, имена переменных и числовые значения.

Удалить ненужные части: игнорировать символы, которые не влияют на работу программы, такие как пробелы, пустые строки и подписи.

Предоставление выходных данных: создание списка токенов с информацией о типе, стоимости и расположении токена в исходном коде, чтобы обеспечить следующий этап (синтаксический анализ).

Этапы лексический анализ

1. Чтение исходного кода:Исходный код обрабатывается в виде строки символов. Компилятор будет считывать каждый символ последовательно, чтобы определить словарные шаблоны.
2. Идентификация лексем:На основе правил и шаблонов, определенных в лексической спецификации языка (обычно представленных регулярными выражениями), строки символов идентифицируются как лексемы. Например, int, if, x, 123.
3. Преобразовать лексему в лексему**:** каждая лексема сопоставляется с типом лексемы. Токен — это структура данных, которая состоит из трех основных компонентов:
   * Тип токена: Например: KEYWORD, IDENTIFIER, NUMBER.
   * Значение токена: Значение лексемы, например, x или 123.
   * Позиция токена: расположение лексемы в исходном коде, что помогает при необходимости находить ошибки.
4. Обработка словарных ошибок:Если обнаружена лексема, которая не соответствует правилам языка, анализатор словарного запаса сообщит об ошибке и укажет местоположение ошибки в исходном коде, чтобы пользователь мог его исправить.
5. Распределение токенов: Окончательная цепочка токенов отправляется в Syntax Analyzer для дальнейшей обработки.

### **1.2.2. Парсинг (построение дерева разбора)**

Парсинг— это следующий этап в процессе сборки компилятора. На этом этапе изучается синтаксис входного исходного кода на основе формальных грамматических правил языка программирования. На выходе получается дерево синтаксического анализа, представляющее логическую структуру и отношения между компонентами исходного кода.

**Цели синтаксического анализа**

Проверить правильность синтаксиса:убедитесь, что исходный код соответствует синтаксическим правилам языка.

Построение дерева синтаксического анализа:создание структуры дерева, представляющей отношения между токенами в исходном коде.

Предоставление выходных данных для этапа семантического анализа:Выходные данные представляют собой дерево синтаксического анализа, которое используется для проверки семантики и семантических ограничений программы.

**Этапы парсинга**

1. Вводить  строку токена:  
    строка токена с этапа анализа словаря отправляется в парсер для обработки.
2. Ознакомьтесь с правилами грамматики:
   * Использовать правила контекстно-свободной грамматики (CFG), чтобы проверить, соответствует ли синтаксис строки токена языку.
   * Грамматические правила обычно представлены грамматикой формы Бэкуса-Наура (BNF) или расширенной формы BNF (EBNF).
3. Обработка синтаксиса и построение дерева разбора:
   * На основе грамматических правил парсер построит дерево синтаксического анализа.
   * Каждый узелв дереве представляет синтаксис, а конечные узлыпредставляют определенные лексемы (например, ключевые слова, операторы или имена переменных).
4. Обнаружение и обработка синтаксических ошибок:
   * Когда он обнаруживает, что исходный код не соответствует синтаксису, синтаксический анализатор сообщит об ошибке, указав местоположение и тип ошибки.
   * Некоторые компиляторы используют механизм Error Recoveryдля продолжения анализа, а не для его полной остановки.
5. Оптимизируем дерево парсера (опционально):
   * Дерево синтаксического анализа упрощается за счет удаления ненужных узлов или объединения промежуточных узлов, создавая абстрактное синтаксическое дерево (AST).
   * AST часто используется в качестве входных данных для этапа семантического анализа.

### **1.2.3. Семантический анализ (построение абстрактного синтаксического дерева)**

Семантический анализ– это следующий этап после парсинга. Этот этап гарантирует, что исходный код является не только синтаксически правильным, но и логически семантическим в соответствии с правилами языка программирования. Результатом этого этапа является абстрактное синтаксическое дерево (AST), которое дополняется семантической информацией.

**Задачи семантического анализа**

1. Проверить семантическую валидность исходного кода:убедитесь, что все утверждения, выражения и манипуляции имеют логическое значение и соответствуют правилам языка.
2. Построение абстрактного синтаксического дерева (AST): AST является более минималистичной и оптимизированной версией дерева синтаксического анализа, представляющей логику программы в более удобном для управления виде.
3. Подготовьтесь к следующим этапам: На этом этапе создаются внутренние посредники для помощи в переводе исходного кода в машинный или целевой код.

**Основные этапы семантического анализа**

1. Анализировать  типы данных:
   * Проверить тип данных выражений, переменных и значений, возвращаемых функцией.
   * Обнаружение и обработка ошибок несовместимости типов данных, например добавление строк к числам.
2. Проверить содержимое и область действия декларации:
   * Перед использованием убедитесь, что все переменные, константы, функции и процедуры объявлены.
   * Убедитесь, что переменные используются в правильной области, чтобы избежать конфликтов имен или недопустимых ссылок.
3. Разрешение названия:
   * Связывать  имена (имена переменных, функций, процедур) с соответствующими объектами в таблице символов.
   * Адресация ссылок, особенно в случае затемнения имен между разными диапазонами.
4. Анализ структуры управления:
   * Проверяем валидность условных операторов (IF, FOR, WHILE) и циклов.
   * Убедитесь, что выражения условий возвращают допустимое значение (обычно логическое значение).
5. Вызов процедуры:
   * Убедитесь, что вызовы функции имеют номер и тип параметра, которые соответствуют определению функции.
   * Убедитесь, что значение, возвращенное функцией, используется правильно.
6. Проверить семантические ограничения языка:
   * Проверка семантических ограничений, специфичных для языка, таких как запрет на постоянные изменения, разрешает только определенные типы данных в контексте.
7. Создавать  внутреннего представителя:
   * На этом этапе абстрактное синтаксическое дерево (AST) преобразуется в промежуточное представление (IR).
   * IR — это форма представления, которая легко преобразуется в машинный код или целевой код.

**Абстрактное синтаксическое дерево (AST)**

AST — это сокращенная версия дерева парсера, сохраняющая только ту информацию, которая необходима для представления логики программы. AST удаляет избыточные узлы и сосредотачивается на основных семантических компонентах, таких как:

* Условные операторы.
* Математические и логические выражения.
* Структура цикла.

### **1.2.4. Сборка транслятора в ассемблерный код**

Этапом сборки транслятора в ассемблерный код является переход от промежуточного представления (Intermediate Representation - IR)) к машинному коду или ассемблерному коду. На этом этапе основное внимание уделяется оптимизации производительности и обеспечению эффективного выполнения выходного кода на целевом оборудовании или платформе.

**Основные этапы сборки транслятора в ассемблерный код**

1. Выбор целевой структуры:
   * Определять  аппаратную архитектуру или среду выполнения (CPU, GPU, LLVM и т. д.).
   * Оптимизировать ассемблерный код в соответствии со спецификациями, специфичными для платформы (например, количеством регистров, скриптов или моделей памяти).
2. Переносить  внутреннее представление в Machine Code:
   * Использовать  промежуточное представление (IR)из предыдущего этапа для создания кода сборки.
   * Преобразовывать  управляющие структуры (IF, FOR, WHILE) и выражения в конкретные команды компьютера.
   * Присваивать  переменные и значения регистрам или расположениям в памяти.
   * Обрабатывать  математические, логические операции и операции доступа к памяти в соответствующих командах сборки.
3. Генерировать  дополнительные данные:
   * Таблица символов: Хранит информацию о расположении переменных, функций и меток в памяти.
   * Таблица меток**:** Хранит точки перехода в коде ассемблера для поддержки управления потоком (goto, loop).
   * Поддержка отладки:добавление сведений, помогающих отслеживать исходный код во время отладки, таких как сопоставление исходного кода с кодом сборки.
4. Поддержка среды выполнения:
   * Предоставляет необходимые компоненты для поддержки выполнения кода компиляции, в том числе:
     + Инициализация программы:настройка памяти, стека, кучи и глобальных переменных.
     + Библиотечные функции:подключение кода к библиотекам времени выполнения или операционным системам, предоставление таких служб, как ввод-вывод, управление памятью или вызов системных функций.
     + Обработка ошибок:Добавлены механизмы обнаружения и обработки исключений в ассемблерном коде.
   * Убедитесь, что код сборки совместим с операционной системой и компоновщиком в целевой системе.

### **1.2.5. Построение макета**

Этап создания макета — это заключительный процесс создания исполняемого файла или библиотеки из исходного кода. На этом этапе все компоненты программы организуются и повторно связываются в конечный продукт, который может быть запущен в целевой среде. Основными задачами на этом этапе являются выбор и загрузка модулей, создание таблиц символов, а также создание исполняемых файлов или библиотек.

**Основные этапы построения макета**

1. Выбирать  и Загружать  модули:
   * Модули**:** На этом этапе выбираются программные модули и загружаются в память. Каждый модуль обычно представляет определенный компонент программы (например, функции, процедуры, библиотеки).
   * Этот процесс может включать:
     + Динамическое связывание: Модули загружаются по мере выполнения программы и могут изменяться в зависимости от окружающей среды.
     + Статическая компоновка: Модули, которые подключаются непосредственно к исполняемому файлу во время компиляции.
   * В зависимости от того, как они построены, модули могут быть скомпонованы прямо во время компиляции или связаны при выполнении.
2. Создавать  таблицу символов:
   * Таблица символовсодержит информацию обо всех компонентах программы, включая переменные, константы, функции, процедуры и другие объекты.
   * В таблице символов хранится такая информация, как имя, тип данных, диапазон и расположение памяти этих компонентов. Это важный источник информации для связывания модулей и выполнения операций в программе.
   * На этапе компоновки доска символов используется для определения того, где и как связаны компоненты программы.
3. Создавать  исполняемый файл или библиотеку:
   * Генерация исполняемого файла: После загрузки всех модулей и создания таблицы символов компилятор генерирует исполняемый файл. Этот файл может быть .exe или **.out** (в зависимости от операционной системы). Этот исполняемый файл может быть запущен непосредственно в целевой среде.
   * Создание библиотеки:В дополнение к исполняемому файлу, в некоторых случаях может быть создана динамическая библиотека (DLL) или статическая библиотека. Эта библиотека содержит общие модули, которые могут быть использованы другими программами. Библиотеки помогают повторно использовать исходный код и свести к минимуму дублирование в приложениях.

### **1.2.6. Заключение**

Процесс сборки компилятора от исходного кода языка программирования до машинного кода или ассемблерного кода является сложным процессом и требует согласования различных этапов. Начиная с анализа словарного запаса, где исходный код разбивается на небольшие блоки для удобства обработки, через парсинг для построения структуры дерева парсинга, с последующим семантическим анализом для проверки валидности программы с точки зрения семантики и типов данных. Затем компилятор преобразует исходный код в промежуточное представление, оптимизирует его и, наконец, переходит к ассемблерному коду или машинному коду.

Процесс сборки транслятора в ассемблерный код оптимизирует производительность и генерирует машинный код, совместимый с целевым оборудованием. Наконец, этап построения организационной структуры и связывания модулей, создания таблицы символов и создания исполняемого файла или библиотеки помогает завершить скомпилированный продукт, готовый к развертыванию и выполнению.

Каждый этап процесса сборки компилятора играет решающую роль в обеспечении точности, эффективности и возможности повторного использования программного обеспечения. Оттуда программы могут быть преобразованы и выполнены в целевых системах эффективно и стабильно.

# **2. Компоновка**

## **2.1 Выбор языка программирования**

При разработке компилятора для языка Basic важную роль в обеспечении эффективности и реализуемости проекта играет выбор языка программирования и вспомогательных инструментов. Рассмотрев множество факторов, таких как гибкость, поддержка сообщества и способность к быстрому росту, Python был выбран в качестве основного языка для создания компилятора.

Python — это язык программирования высокого уровня с ясным, легко читаемым и простым в изучении синтаксисом, который минимизирует время разработки и повышает удобство сопровождения исходного кода. Python хорошо подходит для развертывания приложений, связанных с анализом языка, благодаря своей способности обрабатывать абстрактные данные и строить синтаксическое дерево (AST).

## **2.2 Грамматический анализ основного языка**

1. Основные понятия

* Константы :
  + Целое число (INTEGER):

INTEGER::= [0-9]+

* + Реальное число (REAL\_NUMBER):

REAL\_NUMBER ::= [0-9]+"." ([0-9]\*)? (e[-+]? [0-9]+)?

* + Строковый литерал(STRING\_CONST):

STRING ::= "\""[^\"]\*"\""

* Имя переменной или функции (IDENTIFIER):

IDENTIFIER::= [A-Za-z][A-Za-z0-9\_]\*

* Ключевые слова :
  + Procedure, Routine, Finish, Define, Allocate, Display, Condition, Clause, Repeat, Terminate.
* Оператор сравнения (Comparison\_operator):

Оператор сравнения ::= > | < | >= | <= | = | <>

* Оператор арифметики (Arithmetic\_operator):
  + Сложение/вычитание: **+**, **-**.
  + Умножение/деление: **\***, **/**.

2. Структура программы

* Структурапрограммы: (Program):

Program ::= Global\_declaration

Global\_declaration ::= Declaration Global\_declarations | eps

* Глобальные декларации (Declaration):

Declaration ::= Subroutine\_declaration | Subroutine\_definition | Function\_declaration | Function\_definition | Variable\_declaration

3. Декларация процедур и функций

* Объявление процедуры (Subroutine\_prototype):

Subroutine\_prototype ::= SUB IDENTIFIER "(" Parameters\_list ")"

* Объявление функции (Function\_prototype):

Function\_prototype ::= FUNCTION Varname "(" Parameters\_list ")"

* Определение процедуры(Subroutine\_definition) :

Subroutine\_definition ::= Subroutine\_prototype Statements END\_SUB

* Определение Функции(Function\_definition) :

Function\_definition ::= Function\_prototype Statements END\_FUNCTION

* Предварительное объявление процедур/функций (Subroutine\_declaration и Function\_declaration):

Subroutine\_declaration ::= DECLARE Subroutine\_prototype  
Function\_declaration ::= DECLARE Function\_prototype

4. Объявление переменных

* Объявление переменных (Variable\_declaration) :

Variable\_declaration ::= DIM Varname\_or\_array\_argument Variable\_initialization  
Variable\_initialization ::= ‘=’ Expression | ‘=’ InitialValueList | eps

* Инициализируем значение массива (InitialValueList):

InitialValueList ::= ‘{‘ Initializer\_list\_values ‘}’  
Initial\_list\_values ::= Initial\_list\_value "," Initiali\_list\_values | Initial\_list\_value  
Initial\_list\_value ::= Expression | Initial\_list

5. Списки параметров и аргументы

* Список параметров (Parameter\_list) :

Parameters\_list ::= Varname\_or\_array\_parameter "," Parameters\_list | Varname\_or\_array\_parameter | eps

* Список аргументов (Argument\_list) :

Arguments\_list ::= Expression "," Non\_empty\_arguments\_list | Expression | eps

6. Команды

* Заявление (Statement):

Statements ::= Statement Statements | Exit\_statement | Statement | eps

* Задание (Assignment\_statement):

Assignment\_statement ::= Varname "=" Expression | Func\_call\_or\_array\_index "=" Expression

* Условные (If\_Clause):

If\_Clause ::= IF Expression THEN Statements Else\_Clause END\_IF

Else\_Clause ::= ELSE Statements | eps

* Цикл (Loop):

Loop ::= For\_loop | While\_loop  
For\_loop ::= FOR Varname "=" Expression TO Expression Statements NEXT Varname  
While\_loop ::= DO Pre\_or\_post\_loop  
Pre\_or\_post\_loop ::= WHILE Pre\_loop | UNTIL Pre\_loop | Post\_loop

7. Выражения

* Выражение (Expression):

Expression ::= Arithmetic\_expression | Arithmetic\_expression Comparison\_operator Arithmetic\_expression

* Арифметическое выражение(Arithmetic\_expression):

Arithmetic\_expression ::= Term | Add\_operator Term | Arithmetic\_expression Add\_operator Term  
Add\_operator ::= "+" | "-"  
Term ::= Power | Term "\*" Power | Term "/" Power  
Power ::= Varname | Constant | "(" Expression ")" | Func\_call\_or\_array\_index

## **2.3 Синтаксический анализ и лексический анализ языка**

Для поддержки построения парсера была использована библиотека parser\_edsl.py . Эта библиотека поддерживает как лексический анализ, так и синтаксический анализ, что помогает эффективно преобразовывать исходный код в абстрактные синтаксические деревья (AST). Заметным преимуществом parser\_edsl.py является его способность автоматически проверять, принадлежит ли грамматика к классу LALR(1), гарантируя, что компилятор может обрабатывать сложные грамматические правила без ошибок. Кроме того, библиотека предоставляет инструменты для обработки синтаксических ошибок и создания подробных отчетов, что облегчает разработчикам отладку и оптимизацию компилятора.

В процессе парсинга важную роль играет вычисление свойств терминала и нетерминальных символов. parser\_edsl.py библиотека предоставляет механизм для эффективного выполнения этой задачи:

1. Рассчитаем свойства символа терминала

2. Оценивать  атрибуты неконечного символа

3. Обработка исключений и ошибок

4. Определять  структуру синтаксиса

Комбинируя грамматические правила и конструкторы синтаксиса, библиотека предоставляет мощный механизм для создания parser\_edsl.py эффективного парсера. Вычисление атрибутов терминала и бесконечных символов, наряду с возможностью обработки ошибок и создания подробных отчетов, делает процесс разработки компилятора плавным и надежным

Результат будет выглядеть следующим образом:

**A computer code with many small letters

AI-generated content may be incorrect.**

## **2.4 Семантический анализ**

После построения синтаксического дерева необходимо провести проверку каждой синтаксической структуры, чтобы убедиться, что они соответствуют семантическим правилам, определенным в описании языка. Этот процесс включает в себя проверку правильности синтаксических структур и устранение неоднозначности между вызовами функций и доступом к массиву.

**Семантическая проверка структуры синтаксиса**

1. FunctionProto:
   * Убедитесь, что имя параметра и имя функции уникальны.
2. FunctionDecl и FunctionDef:
   * Проверять  уникальность сигнатур функций или сопрограмм в глобальном масштабе.
3. VariableDecl:
   * Убедитесь, что имя переменной уникально в локальной области и что инициализация (по выражению или списку инициализации) выполняется правильно.
4. FuncCallOrArrayIndex:
   * Убедитесь, что вызовы функций или индексация массивов существуют и используются правильно.
5. FuncCall:
   * Проверять  допустимость вызовов функций, в том числе количество и тип аргументов.
6. ExitFor:
   * Убедитесь, что указанная переменная существует и находится вне блока цикла со счетчиком.
7. ExitWhile, ExitSubroutine, ExitFunction:
   * Убедитесь, что команды выхода используются вне блоков цикла без счетчиков или функций/сопрограмм.
8. AssignStatement:
   * Проверять  правильность присвоения значения переменной.
9. ForLoop:
   * Убедитесь, что имя счетчика уникально, начальное и конечное значения являются целыми числами, а шаг увеличения/уменьшения соответствует типу данных счетчика.
10. WhileLoop:
    * Проверять  целостность условия цикла, если оно предусмотрено.
11. IfElseStatemen:
    * Убедитесь, что условие является целочисленным значением.
12. UnaryOpExpr:
    * Проверка валидности операций с единицами измерения при применении к переменным.
13. BinOpExpr :
    * Убедитесь, что двоичные операции применяются к совместимым типам данных. Если типы данных несогласованы, следует проверить возможность преобразования в общий тип.

**Обработка семантических ошибок**

Во время семантического анализа может возникнуть множество ошибок из-за синтаксических или семантических проблем. Для обработки этих ошибок в системе реализованы определенные классы ошибок с понятным обменом сообщениями и создана таблица символов для управления областью и типом данных. Некоторые распространенные ошибки включают в себя:

* NegativeSizeError : Размер массива не может быть отрицательным.
* NonConstSizeInitializationError : размер массива не был определен во время инициализации списка.
* LengthMismatchError : несоответствие между размером массива и списком инициализации.
* LaunchizationTypeError : массив инициализируется выражением.
* InitializerListDimensionMismatchError : Размер списка инициализации не совпадает.
* InvalidInitializerListError : обнаруживает выражение и список инициализации в списке инициализации.
* ConversionError : не удается преобразовать тип данных.
* InvalidExitError : Недопустимый выход.
* UndefinedFunctionError : функция не определена.
* UndefinedSymbolError : Символ не был объявлен.
* RedefinedSymbolError : Символ, который был объявлен ранее.
* UnexpectedNextForError : счетчик не указан в цикле For.
* NotIntegerCounterError : счетчик цикла не является целочисленным типом.
* NonIntegerArraySizeInitializationError : размер массива не является целым числом во время инициализации.
* NonIntegerConditionError : условие цикла или оператор If не является целым числом.
* ArrayIndexingDimensionMismatchError : количество индексов превышает размер массива.
* NonIntegerArrayIndexingError : индекс массива не является целым числом.
* IncompatibleBinaryTypesError : несовместимые типы данных в двоичных операциях.
* InvalidUnaryTypeError : Операция юнита недопустима с текущим типом данных.

Комбинируя семантическое тестирование и правила обработки ошибок, система гарантирует, что скомпилированная программа BASIC будет соответствовать всем семантическим правилам и исключит потенциальные ошибки. Это помогает повысить надежность и эффективность работы компилятора.

## **2.5 Создание промежуточного кода**

На этапе генерации промежуточного кода абстрактное синтаксическое дерево (AST) преобразуется в код LLVM-IR, низкоуровневое промежуточное представление. Этот код может быть дополнительно оптимизирован и скомпилирован в машинный код для конкретной целевой платформы.

**Процесс реализации**

1. Выбирать  целевую архитектуру :
   * Определять  процессор, операционную систему и другие характеристики целевой платформы.
   * Это помогает гарантировать, что сгенерированный промежуточный код совместим с окончательной средой выполнения.
2. Преобразуем AST в промежуточный код :
   * На основе структуры AST каждый узел в дереве сопоставляется с соответствующими командами в LLVM-IR.
   * Пример:
     + Арифметические операции преобразуются в такие команды, как add, sub, mul, div.
     + Управляющие операторы, такие как циклы или условия, преобразуются в условные (br) блоки и блоки меток.
3. Генерировать  дополнительные данные :
   * Помимо кода, на этом этапе также создаются необходимые структуры данных, такие как таблицы данных, таблицы виртуальной памяти или таблицы переходов.
4. Поддержка среды выполнения :
   * Промежуточный код может включать в себя вызовы библиотечных функций, поддерживающих время выполнения, например, печать данных на экран или обработку строк.
5. Результаты :

Промежуточный код генерируется в виде файла .ll (LLVM-IR). Этот файл будет использоваться на последующих этапах, таких как оптимизация и линковка.

## **2.6 Библиотека поддержки времени выполнения**

Библиотека поддержки времени выполнения предоставляет основные функции для выполнения программ, такие как печать данных, обработка строк и управление памятью. Эта библиотека написана на языке C и скомпонована с промежуточным кодом LLVM-IR.

**Основные функции**

1. Данные экспорта :

Такие функции, как PrintI, PrintL, PrintD, PrintF и PrintS, реализованы для вывода различных типов данных (целых чисел, вещественных чисел, строк).

1. Работа со строками:

Функция StringConcat объединяет две строки символов.

Функция StringCopy копирует строку из источника в место назначения.

1. Точка входа:

Функция Main определена как точка входа в программу. Он отвечает за инициирование и координацию процесса выполнения.

1. Ссылка на промежуточный код :

С помощью инструмента llvm-link промежуточный код LLVM-IR связывается с библиотекой поддержки времени выполнения и другими модулями для создания полного модуля.

## **2.7 Оптимизация**

На этапе оптимизации повышается производительность и размер промежуточного кода LLVM-IR перед его компиляцией в машинный код.

# 

# **3. Результаты**

Преобразовать LLVM IR в ассемблерный код:

|  |  |
| --- | --- |
| 1 | llc --opaque-pointers test\_for\_basic/test1.ll -o test\_for\_basic/test1.s |

Компилировать ассемблерный код в объектный машинный код:

|  |  |
| --- | --- |
| 1 | clang -c test\_for\_basic/test1.s -o test\_for\_basic/test1.o |

Свяжить объектный машинный код (test1.o) со статической библиотекой (libruntimelib.a):

|  |  |
| --- | --- |
| 1 | llvm\_ir/build/libruntimelib.a -o final\_program |

Вход:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65 | ' 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 |

Результаты соответствуют ожиданиям:

A screen shot of a computer

AI-generated content may be incorrect.

# **Заключение**

В процессе реализации задачи «Компилятор языка BASIC» я выполнил основные задачи, включая анализ словаря, синтаксиса и семантической структуры языка BASIC, выбор подходящей технологии, построение этапа анализа (front-end), описание того, как компилировать структуры BASIC в LLVM IR, реализацию этапа синтеза (back-end) и библиотеки поддержки времени выполнения.

Во-первых, лексический анализ, синтаксиса и семантики BASIC помогает нам лучше понять, как работает язык. Язык BASIC имеет простой, но гибкий синтаксис, позволяющий пользователям легко писать программы для вычислений и обработки данных. Подробное описание этих элементов является важной основой для создания эффективных компиляторов.

С технологической точки зрения я решил использовать Python для реализации фазы анализа (фронтэнд) и LLVM для реализации фазы синтеза (бэкэнд). Python был выбран из-за его гибкости и мощных возможностей обработки строк, в то время как LLVM обеспечивает высокооптимизированную платформу компиляции и хорошую совместимость со многими различными аппаратными архитектурами. Переход от исходного кода BASIC к LLVM IR требует глубокого понимания обоих языков и требует высокого уровня навыков программирования и решения проблем.

Этап компиляции (бэкэнд) и библиотеки поддержки времени выполнения играют важную роль в обеспечении возможности корректного выполнения сгенерированного машинного кода на целевой системе. Библиотека поддержки времени выполнения предоставляет базовые функции, такие как печать данных, обработка строк и управление памятью, которые способствуют бесперебойной работе программ на языке BASIC.

# **Литература**

1. Библиотека Parser\_edsl (https://github.com/bmstu-iu9/parser\_edsl\_python)

2. Грамматика и история языка BASIC (https://en.wikipedia.org/wiki/BASIC)

3. Официальная документация LLVM подробно описывает синтаксис и семантику LLVM IR (https://llvm.org/docs/LangRef.html)

4. Как скомпилировать исходный код в LLVM IR (https://blog.llvm.org/2010/04/compiling-to-llvm-ir.html)

5. Оптимизация кода LLVM/IR (https://llvm.org/docs/Passes.html)

6. Связывание объектных файлов и библиотек поддержки времени выполнения (https://sourceware.org/binutils/docs/ld/)

# **ПРИЛОЖЕНИЕ**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46 | Program ::= Global\_declaration  Global\_declaration ::= Declaration Global\_declarations | eps  Declaration ::= Subroutine\_declaration | Subroutine\_definition | Function\_declaration | Function\_definition | Variable\_declaration  Subroutine\_prototype ::= SUB IDENTIFIER "(" Parameters\_list ")"  Function\_prototype ::= FUNCTION Varname "(" Parameters\_list ")"  Subroutine\_definition ::= Subroutine\_prototype Statements END\_SUB  Function\_definition ::= Function\_prototype Statements END\_FUNCTION  Subroutine\_declaration ::= DECLARE Subroutine\_prototype  Function\_declaration ::= DECLARE Function\_prototype  Variable\_declaration ::= DIM Varname\_or\_array\_argument Variable\_initialization  Variable\_initialization ::= "=" Expression | "=" InitialValueList | eps  InitialValueList ::= "{" Initializer\_list\_values "}"  Initializer\_list\_values ::= Initializer\_list\_value "," Initializer\_list\_values | Initializer\_list\_value  Initializer\_list\_value ::= Expression | Initializer\_list  Parameters\_list ::= Varname\_or\_array\_parameter "," Parameters\_list | Varname\_or\_array\_parameter | eps  Non\_empty\_parameters\_list ::= Varname\_or\_array\_parameter Non\_empty\_parameters\_list | Varname\_or\_array\_parameter  Varname\_or\_array\_parameter ::= Varname | Varname "(" Comma\_list ")"  Comma\_list ::= "," Comma\_list | eps  Arguments\_list ::= Expression "," Non\_empty\_arguments\_list | Expression | eps  Non\_empty\_arguments\_list ::= Expression "," Non\_empty\_arguments\_list | Expression  Varname\_or\_array\_argument ::= Varname | Varname "(" Non\_empty\_arguments\_list ")" | Varname "(" Comma\_list ")"  Func\_call\_or\_array\_index ::= Varname "(" Arguments\_list ")" | IDENTIFIER "(" Arguments\_list ")"  Func\_call ::= PRINT Arguments\_list  Varname ::= IDENTIFIER Type  Type ::= "%" | "&" | "!" | "#" | "$"  Statements ::= Statement Statements | Exit\_statement | Statement | eps  Non\_empty\_statements ::= Statement Non\_empty\_statements | Exit\_statement | Statement  Statement ::= Variable\_declaration | Assignment\_statement | Func\_call\_or\_array\_index | Func\_call | Loop | If\_Clause  Assignment\_statement ::= Varname "=" Expression | Func\_call\_or\_array\_index "=" Expression  If\_Clause ::= IF Expression THEN Statements Else\_Clause END\_IF  Exit\_statement ::= EXIT\_FOR | EXIT\_FOR Varname | EXIT\_DO | EXIT\_LOOP | EXIT\_SUB | EXIT\_FUNCTION  Else\_Clause ::= ELSE Statements | eps  Loop ::= For\_loop | While\_loop  For\_loop ::= FOR Varname "=" Expression TO Expression Statements NEXT Varname  While\_loop ::= DO Pre\_or\_post\_loop  Pre\_or\_post\_loop ::= WHILE Pre\_loop | UNTIL Pre\_loop | Post\_loop  Pre\_loop ::= Expression Statements LOOP  Post\_loop ::= Statements LOOP Post\_loop\_expression  Post\_loop\_expression ::= WHILE Expression | UNTIL Expression | eps  Expression ::= Arithmetic\_expression | Arithmetic\_expression Comparison\_operator Arithmetic\_expression  Comparison\_operator ::= ">" | "<" | ">=" | "<=" | "=" | "<>"  Arithmetic\_expression ::= Term | Add\_operator Term | Arithmetic\_expression Add\_operator Term  Add\_operator ::= "+" | "-"  Term ::= Power | Term "\*" Power | Term "/" Power  Power ::= Varname | Constant | "(" Expression ")" | Func\_call\_or\_array\_index  Constant ::= INTEGERANT | REAL\_NUMBERANT | STRINGANT |

**Грамматика языка BASIC**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53 | #include <cstdio>  #include <cwchar>  #include <clocale>  #include <cstring>  #include <cstdlib>  // Declare runtime helper functions **with** `extern "C"` to avoid name mangling  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(const wchar\_t\* val) { printf("%ls", val); }  extern "C" wchar\_t\* StringConcat(wchar\_t\* lhs, wchar\_t\* rhs) {  size\_t lhsLength = wcslen(lhs);  size\_t rhsLength = wcslen(rhs);  lhs = (wchar\_t\*)realloc(lhs, (lhsLength + rhsLength + **1**) \* sizeof(wchar\_t));  **return** wcscat(lhs, rhs);  }  extern "C" wchar\_t\* StringCopy(const wchar\_t\* str) {  size\_t length = wcslen(str);  wchar\_t\* result = (wchar\_t\*)calloc(length + **1**, sizeof(wchar\_t));  wcsncpy(result, str, length);  **return** result;  }  // Main function of BASIC program  extern "C" wchar\_t\* Main(wchar\_t\*\* argv, int len);  int main(int argc, char\*\* argv) {  setlocale(LC\_ALL, "C.UTF-8");  // Convert command line argument array **from** **char**\* to wchar\_t\*  wchar\_t\*\* args = (wchar\_t\*\*)calloc(argc, sizeof(wchar\_t\*));  **for** (int i = **0**; i < argc; i++) {  size\_t length = strlen(argv[i]);  args[i] = (wchar\_t\*)calloc(length + **1**, sizeof(wchar\_t));  **for** (size\_t j = **0**; j < length; j++) {  args[i][j] = argv[i][j];  }  }  Main(args, argc);  // Free memory  **for** (int i = **0**; i < argc; i++) {  free(args[i]);  }  free(args);  **return** **0**;  } |

**Библиотека поддержки времени**