|  |  |
| --- | --- |
| Gerb-BMSTU_01 | 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 «Программное обеспечение ЭВМ и информационные технологии»  РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА  К КУРСОВОЙ РАБОТЕ  НА ТЕМУ:  «***Реализация компилятора языка Tenet***»  Студент ИУ7-82 А.Ю. Ермаков  (Подпись, дата)  Руководитель А.А. Ступников  (Подпись, дата) | |
| 2020 г. | |

Содержание

[Введение 3](#_Toc53650187)

[1 Аналитический раздел 4](#_Toc53650188)

[1.1 Способы исполнения программного кода 4](#_Toc53650189)

[1.1.1 Компилируемые языки программирования 4](#_Toc53650190)

[1.1.2 Интерпретируемые языка программирования 6](#_Toc53650191)

[1.1.3 Динамически-компилируемые языки программирования 7](#_Toc53650192)

[1.2 Компиляторы 10](#_Toc53650193)

[1.3 Построение лексического анализатора и синтаксического парсера 13](#_Toc53650194)

[1.4 Выводы 15](#_Toc53650195)

[2. Конструкторский раздел 16](#_Toc53650196)

[2.1 Лексический анализатор 16](#_Toc53650197)

[2.1.1 Построение НКА по регулярному выражению 16](#_Toc53650198)

[2.1.2 Построение Детерминированного Конечного Автомата 19](#_Toc53650199)

[2.1.3 Построение ДКА с наименьшим числом состояний 20](#_Toc53650200)

[2.2 Синтаксический Анализатор 21](#_Toc53650201)

[2.2.1 LL(1) анализатор 21](#_Toc53650202)

[2.2.2 LR(1) анализатор 23](#_Toc53650203)

[2.3 Разрабатываемый язык программирования 24](#_Toc53650204)

[2.4 Выводы 25](#_Toc53650205)

[3 Технологический раздел 26](#_Toc53650206)

[3.1 Построение АСД 27](#_Toc53650207)

[3.2 Генерация промежуточного представления IR 27](#_Toc53650208)

[3.3 Вывод 28](#_Toc53650209)

[Заключение 29](#_Toc53650210)

[Список литературы 30](#_Toc53650211)

[Приложение 1. Грамматика языка Tenet 31](#_Toc53650212)

# Введение

Целью курсовой работы по дисциплине «Конструирование компиляторов» является разработка собственного компилятора для выбранного языка программирования. Предполагается, что компилятор, разработанный в ходе данной курсовой работы, станет частью дипломной работы, целью которой является разработка виртуальной машины троичного процессора. Таким образом, в ходе данной работы стоит цель реализовать язык программирования и предназначенный для него компилятор, позволяющий преобразовывать исходный код на языке программирования, поддерживающим операции троичной логики, в исполняемый файл. Исполняемый файл предназначен для двоичных процессоров x86-64 и операционной системы Windows. Для реализации поставленной цели необходимо выполнить следующие задачи :

1. Сравнить существующие языки программирования по способу исполнения,
2. Проанализировать этапы преобразования программного кода,
3. Изучить существующие инструменты, позволяющие упростить процесс построения компилятора,
4. Выбрать наиболее подходящие инструменты для реализации компилятора и реализовать его для описанного языка программирования
5. Подготовить набор тестовых данных для демонстрации работоспособности полученного компилятора

# 1 Аналитический раздел

В данном разделе проведён сравнительный обзор языков программирования по способу их исполнения. На основе этого обзора выбран наиболее подходящий способ исполнения кода для реализуемого языка программирования. Рассмотрены основные этапы преобразования кода и инструменты, позволяющие упростить их реализацию.

## Способы исполнения программного кода

Любой процессор поддерживает набор команд, а также предоставляет наборе регистров для хранения данных и изменения поведения команд. Следовательно, любой программный код в конечном итоге должен быть преобразован к набору команд, поддерживаемых выбранным процессором. Набор таких команд называют машинными командами. Разработка программного продукта на машинном языке затруднительна в связи с тем, что машинные команды не являются читаемыми для человека. Поэтому существуют машинно-зависимые языки программирования (языки ассемблера) – языки, близкие к набору машинных команд, предоставляющие мнемонические команды, соответствующие командам процессора. Программный код, написанный на языке ассемблера, преобразуется в машинный код, а сам этот процесс называется компиляцией. Язык, преобразуемый в процессе компиляции в набор машинных команд, называется компилируемым языком.

### Компилируемые языки программирования

Процесс преобразования кода на языке ассемблера в исполняемый процессором машинный код можно разделить на следующие этапы:

* Препроцессинг
* Трансляция
* Компоновка (линковка)

В ходе этапа препроцессинга происходит предварительная подготовка кода на языке ассемблера – специальная утилита анализирует исходный код, удаляет все комментарии в файле, выполняет подстановку макросов и т.д. Получившийся в итоге программный код на языке ассемблера подвергается трансляции – переводу команд с языка ассемблера на набор машинных команд. Результатом трансляции служат объектные файлы – это файлы, содержащие наборы машинных команд, соответствующие исходным программным модулям на языке ассемблера. Эти модули затем подвергаются процессу компоновки или линковки – объединению разрозненных объектных файлов в единый исполняемый файл.

Очевидным недостатком языков ассемблера является их строгая привязанность к платформе – программа, написанная на языке ассемблера для процессора X не будет работать на процессоре Y, т.к. множество команд и доступных регистров этих процессоров может не совпадать. Для решения этой проблемы были разработаны языки программирования высокого уровня – языки, не оперирующие конкретными командами и регистрами процессоров, а преобразуемые в выбранный язык ассемблера. Таким образом становится возможна разработка программы для нескольких процессоров, при условии, что существует компилятор выбранного языка высокого уровня в машинно-зависимый язык выбранного процессора. Процесс компиляции более подробно рассмотрен ниже, сейчас же добавим, что при компиляции языка высокого уровня добавляется этап ассемблирования – преобразования программы на языке высокого уровня в код на языке ассемблера. Отметим также, что в процессе компиляции возможна оптимизация кода – вычисление, разворачивание циклов, определение и удаление недостигаемого кода (кода, который никогда не будет выполнен), замена вызова небольших функций телом этих функций, определение и замена одинаковых участков кода и т.д. Схематично процесс компиляции программы, написанной на языке С, представлен на рисунке 1.

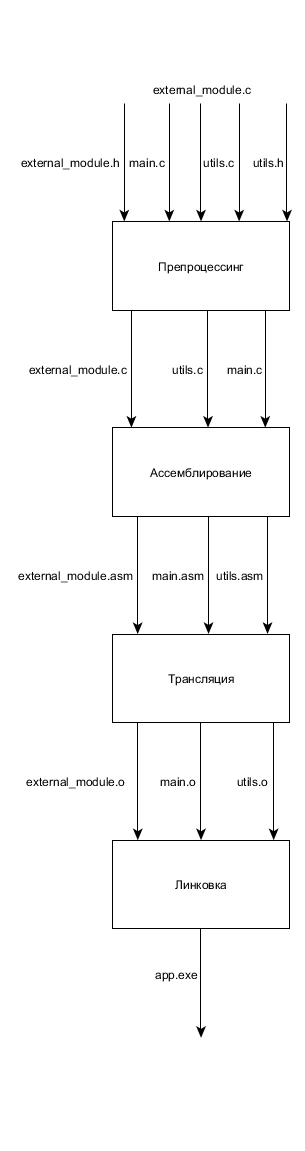


Рисунок 1. Процесс компиляции программы на языке С

### Интерпретируемые языка программирования

Альтернативой компиляции программного кода выступает интерпретация. Суть интерпретации заключается в том, что программа, написанная на интерпретируемом языке программирования, не претерпевает изменений – вместо этого на исполняемой машине устанавливается специальное программное обеспечение, называемое интерпретатором. Интерпретатор получает на вход программный код, преобразует его в набор машинных команд и сразу же выполняет их. Примером интерпретатора может служить интернет-браузер. В состав любого современного браузера входит интерпретатор JavaScript, который выполняет код, написанный на языке JavaScript. При этом браузеры широко распространены не только для различных процессоров, но также и для различных операционных систем. Стоит отметить, что при таком подходе невозможна оптимизация – выполнение кода происходит строчка за строчкой. Также любые семантические и синтаксические ошибки будут определены только в момент разбора конкретной строки, то есть при выполнении кода. Скорость интерпретируемых языков уступает скорости компилируемых: интерпретаторы можно рассматривать как средство виртуализации вычислительных ресурсов. Тем не менее, интерпретируемые языки получили широкое распространение и пользуются популярностью во всех сферах прикладного программирования – существуют интерпретаторы таких языков как Python и JavaScript даже для микропроцессоров.

1.1.3 Динамически-компилируемые языки программирования

JIT-компиляция (Just-In-Time компиляция, динамическая компиляция) – способ преобразования программного кода, использующий преимущества компиляции и интерпретации. В ходе компиляции программный код преобразуется в т.н. byte code. Byte code не является набором машинных команд поэтому не может быть исполнен на процессоре непосредственно, но вместе с тем он лишён привязки к конкретной платформе, что позволяет использовать его для различных процессоров и операционных систем. Для преобразования byte code в набор машинных команд используется т.н. среда исполнения (runtime) – набор утилит, преобразующих byte code в машинный код в момент исполнения последнего. Среду исполнения можно сравнить с интерпретатором, который вместо исходного кода программы преобразует byte code. Преобразование byte code занимает гораздо меньше времени, чем исходного кода, т.к. сложные операции по лексическому и синтаксическому разбору входной программы выполнены в процессе компиляции в byte code. Предварительная компиляция в byte code также позволяет провести процесс оптимизации кода, подобно тому, как это реализовано для компилируемых языков программирования. Синтаксические, семантические ошибки, а также ошибки типизации могут быть отловлены во время компиляции в byte code. Среда исполнения, в свою очередь, также предоставляет ряд методов оптимизации. Например, часто используемые функции могут быть скомпилированы из byte code в машинный код и сохранены на диске, что при следующем обращении к ним избавит от необходимости очередного преобразования из byte code в машинный код. Как правило, весь код приложения не компилируется сразу, т.к. это приведёт к увеличению времени запуска приложения – чем больше код приложения и чем больше операций для оптимизации этого кода необходимо выполнить, тем дольше запускается приложение т.к. большее количество времени тратится средой исполнения на выполнение этих оптимизаций. Схематично процесс компиляции и исполнения программы, написанной на языке C#, проиллюстрирован на рисунке 2. Intermediate Language – byte code, предназначенный для среды исполнения CLR – Common Language Runtime. Любое приложение, написанное для .Net Framework (.Net Core) может быть преобразовано в IL и выполняться при помощи среды исполнения CLR, либо же может быть скомпилировано для конкретной платформы. При этом исходный язык, преобразуемый в IL, не обязательно должен быть C#, более того – различные модули одного приложения могут быть написаны на разных языках программирования, для которых существует реализация .Net Framework.

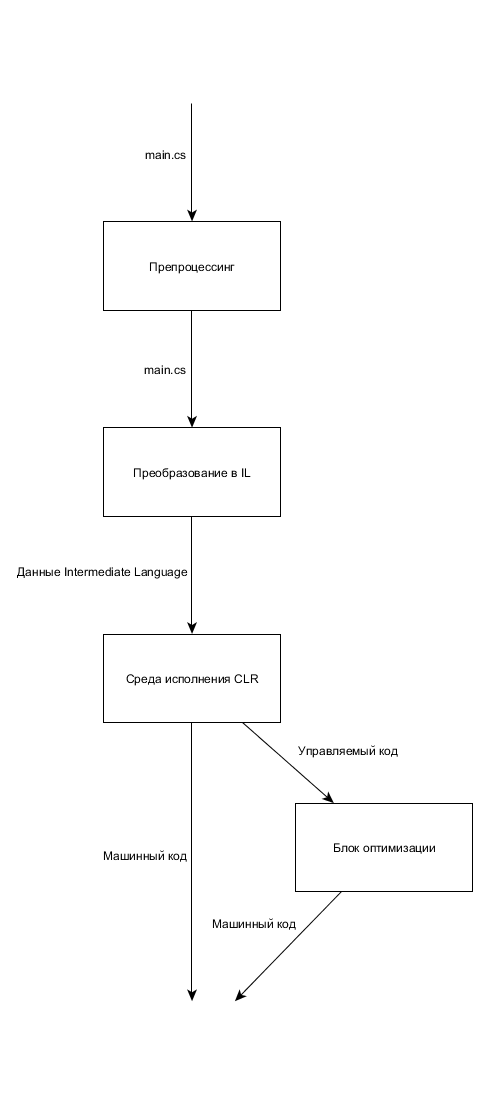


Рисунок 2. Процесс компиляции и исполнения программы, написанной на языке C#

## 1.2 Компиляторы

Компилятор – набор программ и утилит, преобразующих код на языке программирования в набор машинных команд. Машинный код, генерируемый компилятором, зависит от конкретного процессора, для которого он разработан. Также компиляторы должны генерировать исполняемый файл в соответствии с требованиями, предъявляемыми той операционной системой, на которой этот файл будет запускаться. В противном случае операционная система не сможет распознать файл как исполняемый даже в том случае, если набор машинных команд будет поддерживаться процессором. С учётом вышесказанного очевидна необходимость правильного выбора компилятора для конечной платформы.

Современные компиляторы способны выдавать высоко оптимизированный эффективный код. В состав компиляторов входят утилиты для анализа кода, способные найти и удалить неиспользуемые фрагменты кода, выполнять арифметические операции во время компиляции и многое другое. Компиляторы для архитектуры VLIW способны определить независимые наборы команд и объединить их в одновременно выполняемые инструкции.

Условно, компилятор можно разбить на три основные части:

* Frontend-компилятор
* Middleware
* Backend-компилятор

Frontend компилятор занимается обработкой исходного кода программы: преобразованием набора литерал в дерево разбора и составлением таблицы объектов (таблицы идентификаторов). Именно Frontend-компилятор выдаёт предупреждения о синтаксических ошибках: недопустимых идентификаторах, некорректных символах, ошибки, связанные со структурой программы и прочее. Удаление комментариев, разворачивание макросов и прочие манипуляции над текстом программы также выполняются Frontend-компилятором.

Middleware – промежуточный слой. Здесь происходит преобразование дерева разбора в промежуточную форму и последующая её оптимизация. Преобразование в промежуточную форму необходимо для того, чтобы отложить процесс генерации машинно-зависимого кода и выполнить машинно-независимые оптимизации. Это позволяет разрабатывать компиляторы для множества платформ путём замены backend-компилятора.

Backend-компилятор преобразует оптимизированную промежуточную форму в машинный код, а также выполняет машинно-зависимую оптимизацию. Результатом backend-компилятора является исполняемый файл.

Процесс компиляции, разбитый на фазы, изображён на рисунке 3.

* Лексический Анализ – преобразование входного потока литер в лексемы,
* Синтаксический Анализ – разбор структуры программы, преобразование набора лексем в Абстрактное Синтаксическое Дерево,
* Семантический Анализ – определение зависимостей между частями программы, определение типов объектов, областей видимости, соответствий параметров и т.д.,
* Генерация Промежуточного Представления – преобразование входного дерева во внутреннее представление, с целью оптимизации, удобства генерации машинного кода, откладывания процесса преобразования в машинный код (машинно-зависимый этап компиляции),
* Машинно-независимая Оптимизация – оптимизация промежуточного представления, включает в себя вычисление известных на момент компиляции констант, определение и удаление неиспользуемого кода, подстановка однострочных функций, разворачивание циклов и т.д.,
* Генерация Кода – преобразование внутреннего представления в машинно-зависимый набор команд,
* Машинно-зависимая Оптимизация – как правило является частью предыдущего этапа, заключается в распределении доступных регистров, выбора наиболее оптимального набора команд, использовании доступных расширений процессора и т.д.

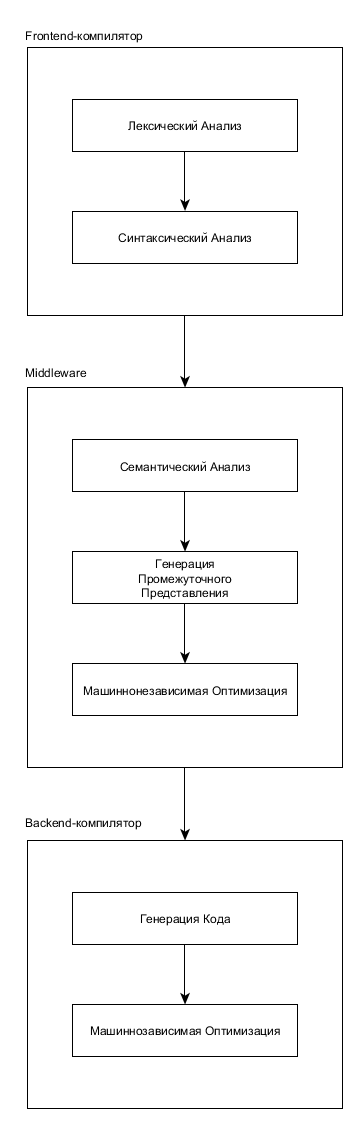


Рисунок 3. Фазы компиляции

В настоящий момент большой популярностью пользуется инфраструктура для разработки компиляторов – LLVM (Low-Level Virtual Machine). Это проект, который является разработкой студентов и преподавателей института Иллиноиса, призванный упростить процесс создания компиляторов для различных языков программирования. Основная идея заключается в том, что для разработки компилятора необходимо реализовать Frontend-компилятор и преобразование получившегося дерева в IR – Intermediate Representation – высокоуровневый ассемблер, содержащий набор RISC команд виртуальной машины с неограниченным набором регистров. Дальнейшие оптимизации выполняются непосредственно над IR. Преобразование IR-кода в машиннозависимый набор команд выполняется backend-компилятором, реализации которого существуют для множества различных платформ. В настоящий момент существует множество реализаций frontend-компиляторов, а также существует поддержка инструментов разработчика для множества существующих языков программирования. Наиболее популярным, тем не менее, является язык C++. Backend-компилятор LLVM не ограничен существующим набором двоичных x86-совместимых платформ, существуют в частности реализации преобразующие IR в код, поддерживаемый видеокартами nVidia, а также WebAssembly и JavaScript, что позволяет заниматься разработкой веб-интерфейсов на компилируемых языках программирования. Помимо статической компиляции, LLVM поддерживает JIT-компиляцию.

## 1.3 Построение лексического анализатора и синтаксического парсера

Основной задачей лексического анализатора, как было сказано выше, является преобразование входного потока символов в набор лексем. Для определения очередной лексемы удобно использовать регулярную грамматику или регулярное выражение, допускающее лексему. На практике для решения этой задачи строится конечный автомат, как правило минимальный детерминированный, допускающий только те цепочки символов, которые допускаются соответствующим регулярным выражением. Далее входной поток символов по очереди подаётся на все конечные автоматы. Если какой-либо конечный автомат распознал часть входной цепочки, то эта часть является соответствующей конечному автомату лексемой. Распознанная цепочка отбрасывается и лексический анализатор продолжает рассмотрение оставшейся части. В случае, если ни один конечный автомат не распознаёт цепочку, выдаётся сообщение об ошибке и анализ останавливается. В процессе работы лексического анализатора также может быть сформирована таблица объектов.

Основная задача синтаксического парсера заключается в построении дерева разбора программы. Это дерево описывает структуру программы в соответствии с контекстно-свободной грамматикой, описывающей выбранный язык программирования. Для построения этого дерева используют LL(1) или LR(1)-анализ или одну из их разновидностей. Более подробно строение LL(1) и LR(1) анализатора будет рассмотрено далее, сейчас же достаточно понимания того, что эти анализаторы состоят из нескольких блоков, а именно:

* Входной цепочки,
* Магазина,
* Таблицы анализатора,
* Алгоритма анализа.

Различия заключаются в структуре и способе построения Таблицы анализатора и самом Алгоритме анализа.

Лексический анализатор и Синтаксический парсер можно написать «вручную» на выбранном языке программирования – что зачастую и делают для простых грамматик. Однако для объёмных грамматик такой подход становится крайне затратным, а внесение изменений в исходную грамматику может привести к значительным изменениям написанного кода. Для упрощения решения задачи построения Лексического анализатора и Синтаксического парсера существуют различные инструменты, позволяющие генерировать указанные выше конструкции на выбранном языке программирования по заданной входной грамматике. Одним из таких инструментов является Antlr4 (Another Tool for Language Recognition v4) – это утилита, написанная на Java, позволяющая генерировать Лексический анализатор и Синтаксический парсер для множества языков программирования, включая Java, C++, C#, Swift, JavaScript и многие другие. В качестве входного параметра утилита получает файл в формате .g4 – описание грамматики выбранного языка. Для использования сгенерированных файлов необходимо установить antlr4-runtime для соответствующего языка программирования. Использование antlr4 позволяет в значительной степени упростить разработку Frontend-компилятора. Antlr4 также предоставляет плагины для популярных IDE, позволяющие тестировать грамматику без генерации кода.

## 1.4 Выводы

В ходе данного раздела были проанализированы основные способы исполнения программного кода: преобразование в набор машинных команд путём компиляции для конкретной платформы, интерпретация исходного кода и смешенный способ – JIT-компиляция. Для разрабатываемого языка программирования была выбрана статическая компиляция программного кода, т.к. при таком подходе удаётся добиться большей производительности, чем при интерпретации исходного кода. Использование JIT-компиляции затруднительно, т.к. в будущем планируется использование разрабатываемого языка для троичных процессоров, для которых не существует готовых сред выполнения, и разработка собственной среды выполнения на низкоуровневом языке программирования является крайней трудоёмкой задачей.

Также был проведён обзор инструментов, которые призваны упростить процесс разработки собственного языка, а именно инфраструктура для разработки собственного компилятора – LLVM, а также генератор Лексического анализатора и Синтаксического парсера Antlr4.

# 2. Конструкторский раздел

В данном разделе рассмотрены основные алгоритмы и подходы, используемые для реализации блоков компилятора, приведено описание характеристик разрабатываемого языка программирования и предъявляемых к нему требований.

## 2.1 Лексический анализатор

Центральную роль в лексическом анализаторе занимает Конечный Автомат (КА) – именно КА определяет, принадлежит некоторая цепочка α регулярному множеству R или нет. Регулярное множество R, или соответствующее ему регулярное выражение r, описывает допустимую входную лексему. Строгое математическое описание КА имеет вид *M = (Q, T, D, q0, F)*, где

* *Q* – конечное множество состояний,
* *T* – конечное множество допустимых входных символов,
* *D* – функция переходов,
* – начальное состояние,
* - множество конечных состояний.

Конечный Автомат является недетерминированным, т.к. находятся в некотором состоянии *q`* он может выполнить переход, в общем случае, в одно из нескольких состояний по одному и тому же символу. Более того, в общем случае в КА существуют переходы по пустому символу *ε*.

Конечный Автомат называют детерминированным (ДКА), если может совершить переход не более чем в одно состояние, а переходы по пустому символу *ε* отсутствуют.

### 2.1.1 Построение НКА по регулярному выражению

Идея алгоритма построения НКА по регулярному выражению заключается в следующем: любой КА может быть представлен в виде композиции КА, соответствующих подвыражениям. Разбиение КА продолжается до тех пор, пока все выражения не будут тривиальными. На каждом шаге разбиения очередной автомат имеет ровно одно заключительное состояние, в начальное состояние нет переходов из других состояний, из заключительного состояния нет перехода в другие состояния КА. Конечные автоматы, соответствующие выражению *ε* и выражению изображены на рисунке 4.

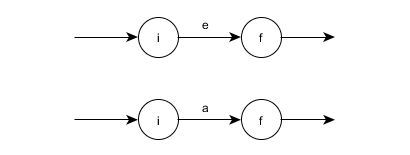


Рисунок 4. КА, соответствующие выражению *ε* и выражению

Для выражения *s|t* автомат M(*s|t*) строится согласно следующим принципам:

* *i* становится начальным состоянием автомата M(*s|t*),
* *f* становится конечным состоянием автомата M(*s|t*),
* из *i* существуют переходы по *ε* в начальные состояние автоматов M(*s*) и M(*t*), эти состояния не являются начальным для M(*s|t*),
* существуют переходы по *ε* из конечных состояний M(*s*) и M(*t*) в конечное состояние *f*, при этом конечные состояния M(*s*) и M(*t*) не являются конечными состояниями M(*s|t*).

Применение описанных выше принципов проиллюстрировано на рисунке 5.

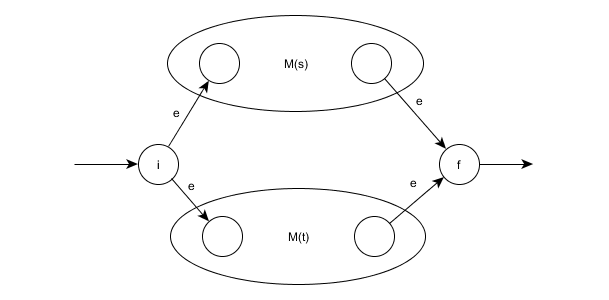


Рисунок 5. Построение конечного автомата для цепочки *s|t*

Автомат для выражения *st* строится следующим образом:

* Начальное состояние автомата M(*s*) становится начальным состоянием автомата M(*st*),
* Конечное состояние автомата M(*t*) становится конечным состоянием автомата M(*st*),
* Конечное состояние автомата M(*s*) и начальное состояние автомата M(*t*) объединяют в одно, причём оно не является ни начальным, ни конечным для автомата M(*st*).

Конечный автомат для выражения M(*st*), построенный таким образом, показан на рисунке 6.

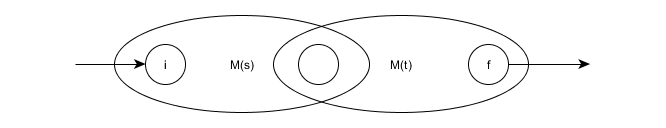


Рисунок 6. КА, построенный для выражения *st.*

Наконец, для выражения *s\** КА M(*s\**) строится следующим образом:

* Добавляется новое начальное состояние *i*,
* Добавляется новое конечное состояние *f*,
* Из состояния *i* существует два перехода по *ε* – в новое конечное состояние *f*, и в начальное состояние автомата M(*s*),
* Из конечного состояния автомата M(*s*) существует переход по *ε* в начальное состояние автомата M(*s*).

КА, построенный для выражения *s\** показан на рисунке 7.

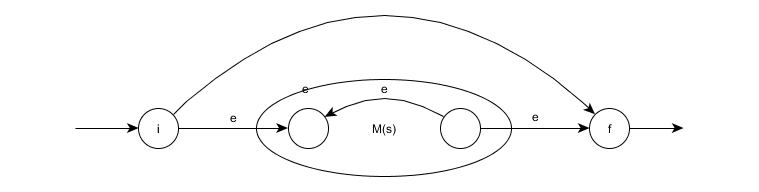


Рисунок 7. Конечный Автомат M(*s\**).

### 2.1.2 Построение Детерминированного Конечного Автомата

Для начала, рассмотрим алгоритм построения ДКА по НКА. Иными словами, необходимо и НКА M = (Q, T, D, q0, F) получить ДКА M`: L(M) = L(M`); Основная идея алгоритма основана на том, что каждое состояние результирующего ДКА M` представляет собой множество состояний исходного НКА M. В алгоритме используются две вспомогательные функции:

* E-closure(R), – множество состояний НКА, в которые можно перейти из R по символу *ε,*
* Move(R, *a*), – множество состояний НКА, в которые можно перейти из R по символу *a*.

Q` и D` в начальный момент пусты. Далее необходимо:

1. *q`0* = e-closure({*q0*}),
2. Добавить *q`0* в *Q`* как непомеченное состояние,
3. Пока в Q` есть непомеченное состояние R {

Пометить R;

Для каждого входного символа *a* {

S = e-closure(move(R, *a*));

Если S ≠ {

Добавить S в Q` как непомеченное состояние

D`(R, *a*) = S;

}}}

1. F` =

Отметим, что существует также алгоритм, позволяющий построить ДКА по регулярному выражению, однако в данной работе этот алгоритм рассмотрен не будет.

### 2.1.3 Построение ДКА с наименьшим числом состояний

Пусть дан ДКА M = (Q, T, D, q0, F). Будем называть такой ДКА всюду определённым, если для всех состояний существует переход по всем символам из множества допустимых символов.

Покажем, как построить всюду определённый ДКА M`: L(M) = L(M`). Для этого рассмотрим M` = (Q{q`}, T, D`, q0, F), где {q`} . Определим D` следующим образом:

Пусть M = (Q, T, D, q0, F) – всюду определённый ДКА. Покажем, как построить M` = (Q`, T, D`, q0`, F`): L(M) = L(M`), |Q`| = min.

Введём вспомогательную функцию split(G), которая разбивает G на подгруппы так, чтобы в одной подгруппе оказались только такие состояния s, t из G, для которых для каждого входного символа *a* существуют переходы по *a* в состояния из одной и той же группы в П. Тогда алгоритм приобретает следующий вид:

1. Построим начальное разбиение множество состояний П = {F, Q - F},
2. Для каждой группы G в П выполним split(G). В новом разбиение Пnew заменить G на множество всех полученных подгрупп,
3. Если Пnew = П => Пres = П, перейдём к шагу 4, иначе повторяем шаг 2 с П = Пnew,
4. Пусть П = {G1, …, Gn}. Тогда Q` = {G1, …, Gn}, q0` = G: ,
5. Если M` содержит состояние, которое не является допускающим и из которого нет путей в допускающие, оно удаляется из M`. Также удаляются все состояния, недостижимые из начального.

## 2.2 Синтаксический Анализатор

Синтаксический Анализатор, как правило, строится на основе LL(1) или LR(1) анализатора или одной из их разновидностей. Принципиальное отличие между LL и LR анализаторами заключается в способе обхода дерева – сверху вниз или снизу верх. В первом случае происходит замена самого левого нетерминала, а во втором – самого правого, что и отражено в название парсеров.

### 2.2.1 LL(1) анализатор

В основе LL анализатора лежит принцип разбора сверху-вниз. Задача такого разбора заключается в определении очередного правила вывода, которое нужно применить к нетерминалу на данном шаге. Графически такой разбор представляется в виде дерева. В начальный дерево состоит из одной вершины – аксиомы грамматики. В этот момент по символу входной цепочки анализатор должен определить, какое правило перехода S → XY должно быть применено к S. При этом X и Y становятся узлами дерева. Затем необходимо определить правило, применяемое к X и т.д. Это продолжается до тех пор, пока для сентенциальной формы, соответствующей левому выводу, не будет найдено правило Z → *a*. После чего процесс повторяется для следующего левого нетерминального символа сентенциальной формы.

Рассмотрим алгоритм табличного предсказателя, представленного на рисунке 8. Такой предсказатель определяет очередное правило с помощью таблицы, которая строится по грамматике.

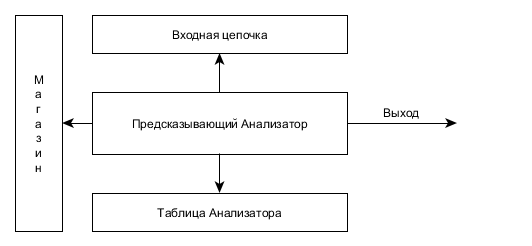


Рисунок 8. LL(1)-анализатор

Принцип работы LL-анализатора в следующем: входная цепочка содержит анализируемую строку, оканчивающуюся знаком $. На выход подаётся последовательность применяемых правил вывода. Таблица анализа представляет собой двухмерный массив M[A, a], где A – нетерминал, a – терминал или символ $. Значением M[A, a] служит некоторое правило грамматики, а его отсутствие обозначает ошибку вывода. Магазин может содержать последовательность символов грамматики с символом $ на дне. В начальный момент входная цепочка имеет вид *w*$, где *w* – анализируемая цепочка, магазин имеет вид S$, где S – аксиома грамматики. На каждом очередном такте анализатор просматривает символ X – символ на верхушке магазина и очередной символ входной цепочки a. Возможны следующие варианты:

* X = a = $ - анализ успешно завершён,
* X = a ≠ $ - X удаляется из массива, а указатель входной цепочки перемещается на следующий за a символ,
* Если X – терминал и X ≠ a, анализатор прекращает работу с сообщением о непринадлежности цепочки *w* входной грамматике,
* Если X – нетерминал, то, в зависимости от значения M[X, a] возможны два случая:
  + M[X, a] – правило для X. В таком случае X на вершине магазина заменяется на правую часть этого правила, правило дублируется на выход,
  + M[X, a] отсутствует – анализатор прекращает работу с сообщением о непринадлежности цепочки грамматике.

Важную роль в данном алгоритме играет таблица M[X, a]. Для построения таблицы необходимо определить две вспомогательные функции: first(α) и follow(X). Первая используется для определения множества первых нетерминальных символов в цепочке α, вторая – для определения всех нетерминальных символов, следующих за символом X в данной грамматике. Алгоритм построения таблицы следующий:

* Для каждого правила A → α
  + Для каждого терминала *a* из first(α) M[A, *a*] = A → α,
  + Если для каждого терминала b из follow(A) M[A,b] = A → α.

### 2.2.2 LR(1) анализатор

Разбор вида снизу-вверх (сдвиг-свёртка) является основным способом построения дерева в LR анализаторах. Суть его заключается в том, что каждый терминал входной цепочки рассматривается как лист дерева преобразований, и в ходе анализа происходит обратный процесс – терминальные символы сворачиваются в нетерминальные, пока те, в конечном итоге, не свернуться в аксиому грамматики. Свёртка достигается путём сопоставления текущей подцепочки правой части некоторого правила в грамматике G и замены этой подцепочки левой частью правила.

LR анализатор состоит из тех же блоков, что и LL анализатор, но их функционирование отличается. В частности, таблица анализатора состоит теперь из двух частей: функций действий (Action) и функций переходов (Goto). В магазине хранятся символы в порядке S0…XiSi… XmSm, где Xi – символ грамматики, Si – символ состояния. Функция действий Action[Sm, ai] может иметь одно из четырёх значений:

* Shift S – сдвиг состояния,
* Reduce A → γ- свёртка по правилу грамматики,
* Accept – допуск,
* Error – ошибка.

Функция переходов Goto[Sm, A], в свою очередь, принимает одно из двух значений:

* S – символ состояния,
* Error – ошибка.

На очередном этапе анализа поведение анализатора определяется функцией действий Action[Sm, ai]:

* Error – обнаружение ошибки,
* Accept – разбор успешен,
* Shift S – в магазин помещаются символ ai и символ состояния S. Входным символов становится ai + 1,
* Reduce A → γ – анализатор выполняет свёртку, из магазина удаляется 2r символов (где r – длина правила): r символов состояния и r символов грамматики, после чего в магазин помещается левая часть правила вывода A и S – символ состояния, определяемый Goto[Sm-r, A].

## 2.3 Разрабатываемый язык программирования

Разрабатываемый язык должен быть процедурным. Это значит, что в рамках данной курсовой работы поддержка таких парадигм программирования как Объектно-Ориентированное, Протоколо-Ориентированное, Функциональное программирование и т.д. не реализована. Разрабатываемый язык позволяет пользователю создавать собственные функции.

Язык должен быть типизирован. Предполагается использование таких типов данных как Real – для чисел с плавающей точкой, Int – для целочисленных данных и String – для хранения строк. Целочисленные данные могут быть представлены в двоичном, троичном, восьмеричном, десятичном и шестнадцатеричном формате. Для записи чисел с плавающей точкой используется запись с десятичной точкой.

В языке присутствуют переменные и константы. Переменные и константы обязательно должны иметь тип и начальное значение. Значением может быть вычислимое выражение.

Основной функциональной единицей языка является операция. Каждая операция заканчивается символом ‘;’. Операцией может быть определение или объявления функции, создание переменной, константы или какое-либо вычислимое выражение. Вычислимое выражение всегда представляет некоторое значение – это может быть результат вызова функции, применения математический операций, результат сравнения, идентификатор (переменная или константа) или значение, а также композиция выражений.

Программа разбивается на области видимости: глобальную и локальные. Глобальная область видимости может быть пустой, или состоять из набора операций. Локальная область видимости – это тело функции, условного оператора или цикла. Условный оператор и цикл допустимы только внутри локальной области видимости.

Объявление и определение функции обязательно должно содержать набор аргументов. В случае, если функция не принимает аргументов, она должна явно указать значение void. Void также может использоваться в качестве возвращаемого типа значения. Указание типа возвращаемого значения обязательно. Если некоторая функция f не принимает аргументов, то при вызове функции указывать void не нужно. Это позволяет проще визуально отделить определение функции от её вызова.

## 2.4 Выводы

В данном разделе рассмотрены основные алгоритмы и подходы, применяемые при разработке основных блоков Frontend-компилятора: Лексического анализатора и Синтаксического парсера. Составлен список требований, предъявляемых к разрабатываемому языку.

# 3 Технологический раздел

В данном разделе рассмотрены основные используемые технологии, библиотеки и языки программирования. Дано обоснование выбора используемых решений.

В качестве основного используемого языка был выбран C#. Выбор обусловлен тем, что утилита antlr в числе прочих языков поддерживает язык C#: для него разработан runtime, а также поддерживается генерация парсеров. Для языка C# также существует nuget пакет для работы с LLVM. Изначально планировалась вести разработку на языке С++, т.к. именно на этом языке написана библиотека LLVM. Однако, т.к. в качестве основной платформы для разработки выбрана Windows, использование C++ стало затруднительным, т.к. основной компилятор C++ под Windows – MinGW, в состав которого не входит библиотека codecvc, необходимая для правильно работы antlr. Указанная библиотека входит в число стандартных библиотек GNU GCC, однако этот компилятор доступен только на операционных системах Unix.

В качестве основного IDE был выбран Rider, т.к. этот IDE обладает удобной подсветкой синтаксиса, автогенерацией кода, встроенным отладчиком и средствами для работы с пакетным менеджером nuget. Rider основан на движке InteliJIdea, что позволяет ему использовать огромный набор доступных для этого движка плагинов и сторонних расширений. Одним из таких расширений является плагин ANTLR v4 Grammar Plugin, предназначенный для автоматизированной генерации парсеров, подсветки файлов грамматики, а также позволяющий прогонять входные цепочки на соответствие выбранной грамматике, что позволяет на ходу отлаживать файл грамматики без необходимости собирать готовое приложение.

Основываясь на требованиях, предъявляемых к разрабатываемому ЯП, была описана грамматика поддерживаемого языка. Листинг грамматики приведён в приложении 1.

## 3.1 Построение АСД

Генерацию Конкретного Синтаксического Дерева выполняют сгенерированные antlr лексеры и парсеры. Получившееся дерево реализует интерфейс IParseTree и позволяет получить доступ к его узлам и листьям. Для преобразования, получившегося КСД в АСД и дальнейшей генерации промежуточного представления, библиотека antlr предоставляет два подхода: использование паттерна Visitor и использование паттерна Listener. В первом случае, необходимо реализовать класс, который будет выполнять обход дерева. В момент анализа конкретного узла, antlr вызовет соответствующий метод объекта Visitor, передав в него текущий контекст. Listener, с другой стороны, позволяет подписать на события, и во время обхода дерева получать оповещения о том, что в данный момент начат обход некоторого узла или завершён.

В данной работе использовался подход, основанный на паттерне Visitor, т.к. этот подход является более компактным и читаемым, чем подход, основанный на асинхронных событиях. Более того, в случае с использованием паттерна Visitor нет необходимости хранить контекст, т.к. он передаётся в качестве параметра обхода.

В результате обхода получаем АСД, которое не содержит информации о конкретном языке, который был разобран, но вместо этого содержит всю необходимую информацию о структуре программы. Такое дерево удобно для дальнейшей обработки и с лёгкостью может быть использовано для преобразования в IR.

## 3.2 Генерация промежуточного представления IR

Для получившегося дерева можно выполнить преобразование в IR. К счастью, работать с этим языком напрямую нет необходимости, т.к. LLVM предоставляет набор методов, позволяющий использовать готовое API и абстрагироваться от низлежащего представления.

Основными инструментами для преобразования AST в IR служат классы LLVMContext, хранящий текущий контекст, IRBuilder, предназначенный для непосредственной генерации IR-кода, а также Module, хранящий в себе определения функций, переменных и констант для текущего модуля (определяет их зону видимости).

Для генерации промежуточного представления необходимо реализовать у каждого узла АСД метод преобразования этого узла в IR. В рамках этого метода вызывать соответствующие функции класса IBuilder для добавления инструкций, и класса Module для добавления определения функций и переменных. LLVM также предоставляет набор оптимизаторов, способных проанализировать текущее промежуточное представление и произвести необходимый набор оптимизаций. Пользователь может настроить, какие оптимизации необходимо произвести и в какой момент.

## 3.3 Вывод

В данном разделе рассмотрены варианты использования библиотеки Antlr и LLVM в контексте текущего курсового проекта. Выбраны среда разработки и язык программирования. Составлена грамматика поддерживаемого языка.

# Заключение

В ходе выполнения данной курсовой работы были изучены основные принципы построения компиляторов, а также инструменты, призванные упростить разработку компиляторов. В результате выполнения курсовой работы была сформулирована грамматика разрабатываемого языка программирования, при помощи инструмента Antlr были сгенерированы лексический анализатор и синтаксический парсер. Также были написаны классы, позволяющие выполнить обход получившегося дерева разбора с целью выполнения дальнейших преобразований.

# Список литературы

*My First Language Frontend with LLVM Tutorial*. (б.д.). Получено из LLVM Compiler Infrastructure: https://llvm.org/docs/tutorial/MyFirstLanguageFrontend/index.html

Parr, T. (б.д.). *ANTLR*. Получено из antlr: https://www.antlr.org/

Серебряков В.А. (2006). *Теория и реализация языков программирования.* Москва: МЗ Пресс.

Э.С., Т. (2003). *Архитектура Компьютера* (изд. 4). Москва, Россия: С^ППТЕР.

# Приложение 1. Грамматика языка Tenet

|  |
| --- |
| grammar Tenet;  program: global\_scope;  global\_scope: operation global\_scope | EOF; scope: operation scope | control\_flow |; operation: (declare\_function | define\_function | define\_variable | define\_constant | expression) EOO ;  *// definations* declare\_function: FUNCTION IDENTIFIER'('function\_arguments')' TYPE\_SPECIFIER (TYPE | VOID); define\_function: FUNCTION IDENTIFIER'('function\_arguments')' TYPE\_SPECIFIER (TYPE | VOID) function\_body; define\_variable: VAR IDENTIFIER TYPE\_SPECIFIER TYPE ASSIGN expression; define\_constant: CONST IDENTIFIER TYPE\_SPECIFIER TYPE ASSIGN expression; function\_arguments: function\_arguments',' function\_arguments | IDENTIFIER TYPE\_SPECIFIER TYPE | VOID; function\_body: '{' scope (scope | 'return' expression EOO | ) '}';  *// expression is everything that returns some value* expression: (call\_function | math\_operation | compare | IDENTIFIER | BASIC\_TYPES) (expression |);  call\_function: IDENTIFIER'('function\_params')'; function\_params: function\_params',' function\_params | IDENTIFIER | BASIC\_TYPES |;  *// math operations* math\_operation: MATH\_OPERATION '('expression ',' expression')'; compare: COMPARISON '(' expression ',' expression ')';  *// control flow* control\_flow: while | if; while: WHILE '(' expression ')' '{' scope '}'; if: IF '(' expression ')' '{' scope '}' (else | ); else: ELSE (if | '{' scope '}');  BASIC\_TYPES: INTEGER | REAL | STRING; INTEGER: ('-' | '+')?(BINARY | TERNARY | OCTAL | HEXADECIMAL | DECIMAL); REAL: ('-' | '+')?[0-9]+'.'[0-9]\*; STRING: '"' .\*? '"';  *// Integer Number Types* BINARY: [0-1]+':bin'; TERNARY: ('-1' | '0' | '1')+':ter'; OCTAL: [0-7]+':oct'; HEXADECIMAL: [0-9a-fA-F]+':hex'; DECIMAL: [0-9]+(':dec' | );  *// Keywords* FUNCTION: 'function'; VAR: 'var'; CONST: 'const'; WHILE: 'while'; IF: 'if'; ELSE: 'else';  *// Operators* ASSIGN: '='; TYPE\_SPECIFIER: ':'; TYPE: 'int' | 'real' | 'string'; VOID: 'void'; MATH\_OPERATION: 'mul' | 'div' | 'mod' | 'add' | 'sub' | 'pow'; COMPARISON: 'equal' | 'more' | 'less' | 'emore' | 'eless'; EOO: ';'; *// End of operation* IDENTIFIER: (ALPHA | '\_')(ALPHA | DIGIT | '\_')\*;  ALPHA: [a-zA-Z]; DIGIT: [0-9];  WHITESPACE: [ \n\r\t] -> skip; SINGLE\_LINE\_COMMENT: '//' ~[\n\r]\* ([\n\r] | EOF) -> skip; MULTI\_LINE\_COMMENT: '/\*' .\*? '\*/' -> skip; |