МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
**«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИМЕНИ Н. Г. ЧЕРНЫШЕВСКОГО»**
Кафедра математической кибернетики и компьютерных наук

# БИБЛИОТЕКА ДЛЯ СОЗДАНИЯ КОМПИЛЯТОРОВ

АВТОРЕФЕРАТ БАКАЛАВРСКОЙ РАБОТЫ

студента 4 курса 451 группы
направления 09.03.04 — Программная инженерия
факультета КНиИТ
Дунаева Павла Дмитриевича

Научный руководитель
зав. к. техн. пр., к. ф.-м. н., доцент _____ И. А. Батраева

Заведующий кафедрой
к. ф.-м. н., доцент _____ С. В. Миронов

Саратов 2022

**СОДЕРЖАНИЕ**

# ВВЕДЕНИЕ

Разработка компилятора представляет собой сложную задачу, требующей знаний в различных областях научного знания, а также написания большого количества сложного кода. Сегодня разработчик может воспользоваться помощью различных инструментов, упрощающих разработку трансляторов, начиная с генераторов синтаксических анализаторов и заканчивая библиотеками кодогенерации. Эти инструменты имеют различное применение: одни представляют собой сложные фреймворки, обладающие большим функционалом и требующие зачастую тонкой настройки, что делает их привлекательными, когда речь идёт о серьёзной промышленной разработке, ориентированной на рынок и предусматривающей долгосрочную поддержку. Другие библиотеки предоставляют небольшой функционал, ориентированный прежде всего на реализацию трансляторов предметно-ориентированных языков, роль которых обычно не выходит за рамки одного проекта. Синтаксический анализ и кодогенерация реализуется всегда разными инструментами, что, с одной стороны, даёт разработчику возможность более гибкого проектирования архитектуры компилятора, но, с другой стороны, может отталкивать новичков, желающих попробовать свои силы в разработке собственного компилятора.

Таким образом, существует потребность в инструменте, подходящем для первых опытов в разработке компилятора. Подобные средства могут при этом пригодиться и профессионалам, если есть необходимость быстрого создания прототипа компилятора. Такой инструмент может не обладать большой гибкостью настройки, но должен быть простым в использовании и покрывать так или иначе все компоненты компилятора.

Целью данной работы является создание простой библиотеки для создания компилятора, включающей в себя средства для упрощения синтаксического анализа, семантического анализа и кодогенерации. В ходе работы были поставлены следующие задачи:

- изучить теорию генерации синтаксических анализаторов;
- исследовать возможности автоматизации семантического анализа;
- изучить принципы генерации целевого кода;
- реализовать на основе полученных знаний библиотеку для создания компилятора под язык программирования C#.

# 1 Теоретическая часть

## 1.1 Лексический анализ

Одним из способов задания лексики языка являются регулярные выражения. Регулярное выражение над алфавитом $\Sigma$ представляет собой формулу, атомарными частями которой служат $a \in \Sigma$ — символы алфавита, представляющие множества из одного символа $\{a\}$ — и $\epsilon$ — пустая строка — и в которой допустим ряд теоретико-множественных операций. Регулярное выражение $r$ задаёт язык $L(r)$, который может быть представлен этой формулой. [1]

Регулярное выражение может быть преобразовано в распознаватель связанной с ним лексемы — недетерминированный конечный автомат (НКА). [2] В ходе работы была разработана своя модификация НКА — НКА с промежуточным алфавитом.

НКА с промежуточным алфавитом называется семёрка

$$M = (Q, U, \Sigma, \gamma, \delta, q_0, F),$$

где

- $Q$ — конечное множество состояний автомата;
- $U$ — конечное множество, называемое промежуточным алфавитом;
- $\Sigma$ — конечное множество, входной алфавит;
- $\gamma$ — отображение $\Sigma \to \mathcal{P}(U)$, называемое функцией конвертации, оно **неоднозначно** переводит входной алфавит в промежуточный;
- $\delta$ — отображение $Q \times U \to Q \cup \{\emptyset\}$ (подразумевается, что $\emptyset \notin Q$), называемое функцией переходов, оно **однозначно** изменяет состояние автомата в зависимости от предыдущего состояния и символа промежуточного алфавита, переход в $\emptyset$ означает, что для данной комбинации аргументов функция не определена;
- $q_0$ — начальное состояние автомата;
- $F$ — множество конечных состояний автомата.

Автомат работает следующим образом: когда на вход поступает очередной символ $\sigma \in \Sigma$, он преобразуется в один из символов $u \in \gamma(\sigma)$, состояние автомата $q_i$ изменяется на $q_{i+1} = \delta(q_i, u)$. Автомат принимает строку $s$ тогда и только тогда, когда, будучи запущенным в состоянии $q_0$ и полностью прочитав строку $s$, автомат может перейти в одно из конечных состояний.

Каждому такому автомату соответствует обычный НКА $M' = (Q, \Sigma, \delta', q_0, F)$, где $\delta'(q, \sigma) = \bigcup\limits_{u \in \gamma(\sigma)} \{\delta(q, u)\})$.

Использование проиежуточного алфавита позволяет существенно сократить количество хранимых переходов.

Для преобразования регулярного выражения в обычный НКА существует рекурсивный алгоритм. [2] Он может быть легко адаптирован для задачи получения НКА с промежуточным алфавитом из регулярного выражения, что было сделано в ходе работы.

Разбиение входной последовательности символов на лексемы может осуществляться различными способами. В работе используется следующий: определяется наибольший префикс входной последовательности, который является корректной лексемой, определяется её тип (если возможно выявить несколько типов, берётся тип с наибольшим приоритетом), данная лексема отделяется от исходной строки, оставшаяся строка обрабатывается циклически таким же образом до тех пор, пока строка не станет пустой или не будет возможности выделить непустую корректную лексему.

Таким образом, в качестве способа задания лексики языков были выбраны регулярные выражения, а в качестве распознователя, в который они преобразуются, используется собственная модификация недетерминированного конечного автомата — НКА с промежуточным алфавитом.

### 1.2 Синтаксический анализ

Синтаксис языка программирования может быть задан с помощью формальной грамматики. Грамматикой называется четвёрка $G = (N, \Sigma, P, S)$, где $N$ — конечное множество нетерминальных символов, $\Sigma$ ($\Sigma \cap N = \emptyset$) — множество терминальных символов, P — конечное множество правил (продукций) — конструкций вида $\alpha \to \beta$, где $\alpha \in (N \cup \Sigma)^* N (N \cup \Sigma)^*$ и $\beta \in (N \cup \Sigma)^*$, $S \in N$ — начальный символ грамматики. Если все правила $P$ имеют вид $A \to \beta$, где $A \in N$, то грамматика называется контекстно-свободной. Грамматика рекурсивно задаёт язык следующим образом. Строка называется выводимой, если она удовлетворяет следующим правилам:

1. $S$ — выводимая цепочка;
2. Если $\alpha\beta\gamma$ — выводимая цепочка, и $(\beta \to \delta) \in P$, то $\alpha\delta\gamma$ — выводимая цепочка.

Грамматика $G$ задаёт язык $L(G)$, состоящий из множества цепочек $s \in \Sigma^*$, выводимых $G$. [2]

Пусть синтаксис языка задан таким способом. Одной из разновидностей синтаксического анализа является анализ «перенос/свёртка» (далее ПС-анализ). Он применяется для работы только с языками, заданными контекстно-свободными грамматиками. Анализаторы, построенные по данному принципу, имеют стек для хранения символов грамматики. Входная последовательность читается слева направо. Изначально стек пуст, ни один из символов входной последовательности не прочитан. В процессе сканирования анализатор ноль или более раз переносит символы из входной последовательности в стек, пока не будет готов выполнить свёртку — замену $n$ верхних символов стека $X_1, \ldots, X_n$ на один другой символ $Y$, если существует продукция $Y \to X_1, \ldots, X_n$. Кроме переноса и свёртки, анализатор может принять строку (успешно завершить синтаксический анализ) или остановить анализ с ошибкой, возможно запустив некоторую процедуру восстановления после ошибки. [1]

Главная проблема ПС-анализа заключается в том, чтобы определить, что необходимо делать в каждый конкретный момент времени: перенос, свёртку, завершение анализа или сигнал об ошибке. Одним из возможных решений является использование LR-анализаторов, принимающих решение с помощью автомата в качестве управляющей структуры данных. LR-анализаторы имеют множество разновидностей, в работе используется LR(1)-анализатор.

Таким образом, в качестве способа задания синтаксиса языков были выбраны грамматики, а в качестве анализатора используются LR(1)-автоматы.

### 1.3 Семантический анализ

Семантический анализатор использует построенные в ходе синтаксического анализа структуры данных для проверки исходной программы на семантическую (смысловую) согласованность с определением языка. Он также собирает информацию о типах и сохраняет её для последующего использования в процессе генерации промежуточного кода. [1]

Объекты, встречающиеся в коде, такие как переменные, функции, классы и т. д., может быть удобно организовать в семантическую сеть. Семантическая сеть — это граф, дуги которого представляют собой различного рода отношения между вершинами, представляющими объекты сети. [3] В качестве отношений можно выбрать, например, вложенность объектов (например, это отношение

между классом и его методами, отношение между локальной областью видимости и локальной переменной), наследование классов и т. п.. С помощью поиска по семантической сети могут быть решены, например, задачи связывания встреченного идентификатора с конкретным объектом (переменной, параметром, полем и т.д., тем более если в одной и той же области видимости могут быть доступны без учёта перекрытия друг друга объекты с одинаковыми именами).

Таким образом, в качестве вспомогательного инструмента для семантического анализа пользователю может быть предложена модель семантической сети с возможностью поиска по ней необходимых объектов.

### 1.4 Генерация целевого кода

Прежде, чем сгенерировать целевой код, компилятор обычно строит некоторое внутреннее представление программы в виде промежуточного кода, над которым перед трансляцией в целевой обычно производятся дополнительные действия, такие как, например, оптимизация. [4] Компилятор может поддерживать несколько различных промежуточных кодов, отличающихся уровнями абстракции, переводя код программы из одного промежуточного представления в другое. Одним из возможных промежуточных представлений является трёхадресный код, в котором каждая операция представляется в виде четвёрки `<операция, источник1, источник2, приёмник>` или тройки `<операция, источник, приёмник>`. [5]

В качестве целевого кода для модуля библиотеки, отвечающего за кодогенерацию, был выбран машинный код процессоров архитектуры AMD64. Архитектура поддерживает регистры — специальные ячейки сверхбыстрой памяти — и стек как отдельно выделенную часть оперативной памяти. [6]

В качестве формата исполняемого файла был выбран формат Portable Executable, который используется в ОС семейства Windows. Файл данного формата начинается с нескольких заголовков (старый MS-DOS EXE-заголовок, основные заголовки). [7] Далее идут заголовки секций и содержимое секций. [8]

Таким образом, в качестве целевого кода для клиентских компиляторов был выбран машинный код архитектуры AMD64, помещаемый в исполняемый файл формата Portable Executable.

# 2 Практическая часть

## 2.1 Обзор библиотеки CompileLib

Библиотека имеет три модуля, представленных следующими пространствами имён:

1. `CompileLib.Parsing` содержит классы и атрибуты, необходимые для синтаксического анализа;
2. `CompileLib.Semantics` даёт доступ к вспомогательным классам для семантического анализа;
3. `CompileLib.EmbeddedLanguage` содержит классы, с помощью которых образован встроенный язык EmbeddedLanguage, используемый в качестве промежуточного кода, в который может компилироваться исходный код и который затем компилируется библиотекой в целевой машинный код.

Все три пространства имён находятся в одной DLL, однако каждый из модулей может быть использован независимо от другого.

## 2.2 Синтаксический анализ

Описываемые классы принадлежат пространству имён `CompileLib.Parsing`.

Синтаксический анализ осуществляется с помощью класса `ParsingEngine`. Этот класс содержит два метода, которые анализируют входную последовательность символов в виде строки или текстового файла.

Объект `ParsingEngine` является представлением уже готового синтаксического анализатора. Для того чтобы сконструировать синтаксический анализатор, используется класс `ParsingEngineBuilder`.

Объект класса `ParsingEngineBuilder` создаётся конструктором без параметров, а затем достраивается методами, задающими лексику и синтаксис языка. Для этого класс имеет методы добавления описаний лексем и продукций.

Лексика языка задаётся с помощью метода `AddToken`, который принимает на вход имя лексемы и регулярное выражение, её описывающее.

Регулярные выражения практически полностью соответствуют стандарту POSIX. [9] В целях удобства была добавлена возможность определять собственные классы символов.

Метод `AddProductions` принимает класс, который содержит описание

продукций грамматики и методы, их обрабатывающие. Каждой продукции соответствует ровно один метод-обработчик. Из класса отбираются методы, которые определяются как обработчики продукций. Метод считается обработчиком только тогда, когда он удовлетворяет определённому ряду критериев.

Разметка обработчиков осуществляется с помощью атрибутов — специальной конструкцией языка C#, позволяющей снабдить элементы кода дополнительной информацией. [10] Атрибут `SetTag`, которым обязательно должен быть помечен обработчик, задаёт имя нетерминала, находящегося в левой части прикреплённой к методу продукции. Каждый параметр метода, кроме, возможно, последнего, соответствует символу из правой части продукции. Порядок параметров совпадает с порядком соответствующих символов правой части продукции. Каждый параметр, кроме, возможно, последнего, должен быть помечен ровно одним атрибутом, задающим символ грамматики, и не более чем одним атрибутом, задающим повторение символов. Символ грамматики может быть задан именем нетерминала или ключевым словом. В качестве атрибутов повторения допускаются атрибут, который делает включение символа в продукцию необязательным, и атрибут, который допускает повторение символа произвольное количество раз. Атрибуты повторения могут охватывать несколько символов одновременно, для этого существует атрибут продления предыдущего атрибута повторения.

Тип параметра метода выбирается в зависимости от символа, которому этот параметр соответствует. Если символ является терминальным, то тип параметра должен быть `string` или `Token` — класс для описания лексемы, который содержит свойства для имени типа, самой лексемы, её позиции — строки и столбца. Тип параметра, соответствующего нетерминальному символу, должен быть выбран таким образом, чтобы возвращаемое значение всех методов, описывающих продукции, в левой части которых находится данный символ, могло быть передано в качестве параметра с таким типом, здесь речь идёт в основном о простом совпадении этих типов и о наследовании классов.

Последний параметр метода обработчика может иметь отличное от остальных назначение. Если последний параметр помечен атрибутом `ErrorHandler`, то он может быть использован для восстановления анализатора после ошибок.

Собственно сборка синтаксического анализатора запускается вызовом метода `Create`, который принимает единственное значение — имя стартового

символа грамматики, метод возвращает экземпляр класса `ParsingEngine`.

В процессе построения синтаксического анализатора может выясниться, что по данной грамматике невозможно построить LR(1)-автомат. Тогда метод `Create` выбрасывает исключение типа `ParsingConflictException`, в котором подробно описывается неоднозначность.

Таким образом, синтаксис языка задаётся с помощью класса `ParsingEngineBuilder` и собственных классов пользователя, в которых он описывает синтаксис языка в виде грамматик с помощью атрибутов — специальной конструкции языка C#, которыми размечаются обработчики соответствующих продукций.

### 2.3 Семантическая сеть

Описываемые классы принадлежат пространству имён `CompileLib.Semantics`.

Объекты семантической сети представляются в виде экземпляров класса `CodeObject`. Объекты всегда имеют собственное имя и имя типа объекта (например, `"поле"` или `"класс"`). Объекты могут иметь атрибуты. Каждый атрибут характеризуется названием и, возможно, значением (строкового типа). Атрибут может не иметь значения, тогда становится важен факт его наличия или отсутствия. Объекты могут иметь отношения с другими объектами. Каждый объект владеет множеством пар `<имя отношения, ссылка на объект>`. В классе предусмотрены методы для чтения и записи атрибутов и отношений.

Класса `SemanticNetwork` используется для сложного поиска по семантической сети.

Основным инструментом поиска по сети является самостоятельно разработанный язык SearchLang. Основными объектами этого языка являются правила поиска. Каждое правило имеет имя, набор параметров и тело. Тело правила определяет принцип поиска по сети, параметры — что именно ищется (например, в качестве параметра может быть передано имя объекта или значение его атрибута). Правило можно рассматривать как функцию, возвращающую множество найденных объектов.

Поиск всегда имеет отправную точку (ОТ) — объект, из которого запускается поиск.

Тело правила состоит из одного выражения, которое описывает поиск. Атомарными выражениями являются множества связанных с ОТ объектов, ото-

бранных по некоторым критериям (в частности: имени, типу, атрибуту, имени отношения), и вызовы правил. Атомарные выражения связаны бинарными операциями, среди которых есть две теоретико-множественных операции — пересечение и объединение, а также некоторые дополнительные операции, среди которых, в частности, выделяется операция `X . Y`, которая для каждого элемента множества $X$ применяет правило $Y$ и объединяет результаты, что позволяет организовывать глубокий рекурсивный поиск.

Конструктор класса `SemanticNetwork` принимает на вход строку-скрипт, содержащую правила поиска. Метод `Search` принимает на вход ОТ, имя правила поиска и параметры, передаваемые в правило и возвращает список найденных объектов.

Фактически класс `SemanticNetwork` содержит код правил в виде абстрактного синтаксического дерева, который затем интерпретируется.

Таким образом, для облегчения семантического анализа пользователю предлагаются классы, обеспечивающие построение семантической сети и поиск по ней, который осуществляется с помощью встроенного языка SearchLang.

### 2.4 Backend компилятора: язык EmbeddedLanguage

Язык EmbeddedLanguage служит для генерации компилятором целевого кода. Это процедурный язык программирования. Язык реализован поверх C#, т. е. представлен в виде библиотечных средств, о которых пойдёт речь в этом разделе.

Описываемые классы принадлежат пространству имён `CompileLib.EmbeddedLanguage`.

Ключевым классом языка является `ELCompiler`. С его помощью можно определять функции, глобальные и локальные переменные, константы, инициализированные данные.

Язык EmbeddedLanguage имеет статическую типизацию. Все типы являются экземплярами класса `ELType`. Типы делятся на атомарные типы, типы указателей и структуры.

Среди атомарных типов есть типы для знаковых и беззнаковых целых чисел, для чисел с плавающей точкой и тип `Void`, играющий такую жу роль, как и его аналог в языке Си. Экземпляры типов сохранены в статических read-only полях класса `ELType`.

Структуры определяются с помощью класса `ELStructType` — наследника `ELType`. Класс имеет конструктор, в который передаются выравнивание полей (аналог прагмы `pack` в компиляторах Си) и типы полей.

Перед генерацией кода необходимо создать экземпляр класса `ELCompiler`. Далее будем считать, что данный экземпляр записан в переменную `compiler`.

Функция определяется вызовом метода `compiler.CreateFunction`. Первый параметр функции — возвращаемый тип, остальные (переменное количество) — типы параметров функции. Метод возвращает объект `ELFunction`, который в дальнейшем можно использовать для вызова функции.

Кроме определения собственных функций, можно импортировать функции из DLL. Для этого существует метод `compiler.ImportFunction`, который принимает на вход имя DLL, название функции, возвращаемый тип и типы параметров. Возвращаемый объект также имеет тип `ELFunction` и доступен для вызова.

Для того чтобы начать или продолжить записывать тело функции `f`, необходимо вызвать метод `f.Open()`. Помимо объявленных пользователем и импортированных функций всегда существует ещё одна неявно объявленная функция — точка входа, которая не возвращает значения и не принимает параметры. Для записи этой функции нужно воспользоваться методом `compiler.OpenEntryPoint()`.

Глобальные переменные можно объявить с помощью вызова метода `compiler.AddGlobalVariable`, указав тип переменной. Практически аналогично определяются локальные переменные для записываемой в настоящий момент функции — вызовом `compiler.AddLocalVariable`. Оба метода возвращают объект типа `ELVariable`.

Доступ к параметрам функции `f` можно получить с помощью вызова метода `f.GetParameter`, указав индекс параметра в 0-индексации. Параметр также имеет тип `ELVariable`.

Все выражения имеют тип `ELExpression`. В качестве выражений допускаются: глобальные и локальные переменные, целочисленные и вещественнозначные константы, предопределённые массивы, различные унарные и бинарные операции, приведение типов, разыменовывание и индексация указателей, вызовы функций.

Результат каждого выражения сохраняется в отдельную локальную пере-

менную, т. е. может быть использован многократно, будучи вычисляемым только один раз (на линейном участке кода).

Некоторые выражения являются экземплярами класса `ELMemoryCell`, представляя ячейки памяти. Над ними можно проводить дополнительные операции: чтение и запись значений, получение ссылки на ячейку и получение ячейки-поля структуры.

Помимо выражений, в языке предусмотрены конструкции управления потоком выражений. Для этого в языке предусмотрены операторы возврата из функции, а также операторы условного и безусловного перехода, для работы которых должны быть определены метки.

Собственно экземпляр класса `ELCompiler` может быть получен двумя способами. Первый — получение его с помощью конструктора класса без параметров — позволяет строить целевой код «с чистого листа». Второй — с помощью класса `ELCompilerBuilder` — позволяет определить ряд полезных функций, таких как функции для динамического выделения и освобождения памяти, чтения и записи в консоль, которые можно использовать в целевом коде.

Компиляция кода запускается вызовом `compiler.BuildAndSave`, единственный параметр метода — имя файла, в который будет записан целевой код.

Таким образом, в качестве инструмента кодогенерации пользователю предоставляется язык EmbeddedLanguage, встроенный в C# с помощью синтаксических средств языка-хоста.

### 2.5 Процесс компиляции кода на языке EmbeddedLanguage

Всякий раз, когда строится какое-то новое выражение, неважно с помощью какого метода или какой перегруженной операции, компилятор языка во внутреннем логе сохраняет запись о том, какое выражение с какими параметрами было построено, происходит проверка типов.

После вызова метода `BuildAndSave` лог компилятора преобразуется в промежуточный трёхадресный код `QuasiAsm`.

Промежуточный код `QuasiAsm` затем переводится непосредственно в целевой код — машинный код архитектуры AMD64. Прежде, чем приступить непосредственно к генерации целевого кода, оптимизируется размер пространства, выделяемого под локальные переменные. Таким образом, множество локальных переменных отображается на множество используемых программой

ячеек памяти.

После этого начинается непосредственная сборка целевого кода. Сборка происходит в два прохода. Во время первого прохода код генерируется частично незаполненным. Часть адресов, указываемых в командах, не заполняется, поскольку на этапе генерации кода адреса объектов, такие как адреса функций, инициализированных данных и место для адреса загрузки библиотечных функций ещё не определены. Вместо заполнения этих адресов составляется таблица, в которой указываются позиции для вставки и описание объектов, чьи адреса нужно вставить. На основе таблицы сами адреса генерируются уже во время второго прохода классом-сборщиком исполняемого файла, так как только на этом этапе становятся возможным вычислить адреса всех объектов.

Таким образом, сборка целевого кода проходит несколько этапов: сначала код на EmbeddedLanguage преобразуется во внутренний трёхадресный код, который затем преобразуется в целевой код, проходя стадию оптимизации, на которой для каждой функции уменьшается используемое стековое пространство за счёт отождествления локальных переменных, имеющих непересекающееся время жизни.

### 2.6 Пример компилятора

С помощью разработанной библиотеки за короткое время был создан компилятор Си-подобного объектно-ориентированного языка, использующего все её модули. С помощью средств модуля `CompileLib.Parsing` были описаны лексика, синтаксис языка, поведение компилятора при синтаксическом анализе исходного кода, в том числе при возникновении ошибок, при этом за счёт отсутствия необходимости реализовывать разбор исходного кода самостоятельно, было сэкономлено много времени, включая время на отладку алгоритма разбора. С помощью средств модуля `CompileLib.Semantics` была построена семантическая сеть из объектов исходного кода, по которой, в частности, применяется поиск объектов в локальной области видимости, реализованный на языке SearchLang, код на котором позволил лаконично описать параметры поиска. С помощью языка EmbeddedLanguage была реализована генерация целевого кода, которая за счёт высокоуровневых средств EmbeddedLanguage позволила полностью абстрагироваться от целевой архитектуры. Реализованный компилятор успешно транслирует исходный код в машинный.

# ЗАКЛЮЧЕНИЕ

Таким образом, в ходе работы были решены следующие задачи:

— изучены LR(1)-автоматы как средство синтаксического анализа и способ их генерации на основе описания грамматики языка;
— автоматизирован поиск в семантических сетях, элементами которых являются элементы исходного кода;
— исследована структура исполняемого файла формата Portable Executable и изучены возможности его генерации;
— реализована библиотека CompileLib для языка C#, которая существенно упрощает синтаксический анализ, семантический анализ и генерацию целевого кода, что было показано на примере разработанного в качестве примера компилятора.

## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ


1 *Ахо, А.* Компиляторы: принципы, технологии и инструментарий / А. Ахо, М. Лам, Р. Сети, Дж. Ульман. — Москва: Вильямс, 2008.

2 *Ахо, А.* Теория синтаксического анализа, перевода и компиляции / А. Ахо, Дж. Ульман. — Москва: МИР, 1978.

3 Intuit: курс «Интеллектуальные робототехнические системы» [Электронный ресурс]. — URL: https://intuit.ru/studies/courses/46/46/lecture/1370?page=4 (Дата обращения 01.05.2022). Загл. с экр. Яз. рус.

4 *Пратт, Т.* Языки программирования: разработка и реализация / Т. Пратт, М. Зелковец. — СПб: Питер, 2002.

5 *Хантер, Р.* Проектирование и конструирование компиляторов / Р. Хантер. — Москва: Финансы и статистика, 1984.

6 AMD64 Architecture Programmer's Manual: Volumes 1-5. — Santa Clara: Advanced Micro Devices, 2021.

7 *Касперски, К.* Техника отладки программ без исходных текстов / К. Касперски. — СПб: БХВ-Петербург, 2005.

8 Microsoft Portable Executable and Common Object File Format Specification. — Redmond: Microsoft Corporation, 2017.

9 The Open Group Base Specifications Issue 7, 2018 edition [Электронный ресурс]. — URL: https://pubs.opengroup.org/onlinepubs/9699919799/ (Дата обращения 01.05.2022). Загл. с экр. Яз. англ.

10 Metanit: Атрибуты в .NET [Электронный ресурс]. — URL: https://metanit.com/sharp/tutorial/14.4.php (Дата обращения 01.05.2022). Загл. с экр. Яз. рус.