Министерство образования и науки Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМЕНИ Н. Г. ЧЕРНЫШЕВСКОГО»

Кафедра «Математическая
кибернетика и компьютерные
науки»

ОБУЧАЮЩИЙ КУРС ПО ДИСЦИПЛИНЕ «РАЗРАБОТКА КОМПИЛЯТОРОВ»

АВТОРЕФЕРАТ БАКАЛАВРСКОЙ РАБОТЫ

студента 4 курса 451 группы
направления 09.03.04 — Программная инженерия
факультета КНиИТ
Филатова Михаила Дмитриевича

Научный руководитель
к. ф.-м. н. _____ С. В. Миронов

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

Саратов 2018

ВВЕДЕНИЕ

Программное обеспечение является неотъемлемой частью современного мира. Его разработка ведется для создания, обработки и отображения информации, относящейся к различным сферам человеческой деятельности. Все существующее на данный момент программное обеспечение написано на какомлибо языке программирования. Но для того, чтобы конечная программа могла быть исполнена на компьютере или другом устройстве, она должна быть преобразована в набор машинных команд. Эту функцию выполняет соответствующий транслятор или компилятор языка программирования.

Основой разработки компиляторов является теория формальных языков и трансляций. Этот раздел математики создавался для изучения естественных языков. Однако данная теория нашла широкое применение в области разработки и реализации языков программирования. Использование технологий, основанных на синтаксических методах, стало общепринятым. При этом эффективное и грамотное применение этих средств требует от разработчика знаний математических основ теории формальных языков и трансляций.

Несмотря на наличие большого числа языков программирования, на сегодняшний день продолжают разрабатываться новые языки и компиляторы для них. Задача построения компилятора языка, переводящего исходную программу в оптимальную последовательность команд машинного кода, не имеет однозначного решения. Каждый язык программирования имеет свои ключевые черты и особенности, что требует индивидуального подхода к разработке компилятора для каждого языка.

Все это говорит о необходимости понимания программистом принципов функционирования и устройства компиляторов. Целью настоящей работы является создание обучающего курса для изучения дисциплины «Разработка компиляторов». В данном курсе структурированы знания, необходимые для использования существующих и реализации новых компиляторов для языков программирования. Для достижения поставленной цели сформулированы следующие задачи:

- предоставить необходимые теоретические выкладки и методические рекомендации для изучения дисциплины;
- описать современные технические средства и способ работы с ними;
- сформировать наборы практических заданий для закрепления получен-

ных теоретических знаний;

— реализовать описанные практические задания и проверить их корректность.

В первой части работы представлены теоретические материалы, необходимые для понимания принципов и подходов к разработке компиляторов, описаны основные этапы компиляции программы и их теоретические основы. Во второй части описан практикум по дисциплине, содержащий наборы заданий для практической реализации, а также методические материалы по разработке конечного компилятора.

1 Теоретические материалы курса

Язык программирования — это формальная знаковая система, используемая для записи компьютерных программ. Для того чтобы программный код мог быть выполнен процессором его необходимо привести к набору машинных команд (инструкций процессора). Такой процесс преобразования программы на одном языке программирования в программу на другом языке программирования называется трансляцией, а программа, выполняющая это преобразование — транслятором [1]

Программы-трансляторы делятся на два класса: компиляторы и интерпретаторы [2]. Компиляторы предварительно переводят исходную программы в целевую (набор машинных команд). После преобразования полученные команды заносятся в память процессора и исполняются.

Процесс компиляции кода разделен на две основные части: анализ и синтез [3].

Во время фазы анализа выявляется структура входной программы, исходный код проверяется на соответствие правилам используемого языка программирования, а в случае обнаружения ошибок компилятор формирует информативное сообщение, чтобы пользователь мог исправить допущенные ошибки. Также при анализе кода формируется вспомогательная структура — таблица символов, которая используется на этапе синтеза. Фаза анализа кода состоит из трех последовательных этапов

— лексический анализ;
— синтаксический анализ;
— семантический анализ.

Фаза синтеза отвечает за формирование итоговой исполняемой программы на целевом языке программирования. После получения разбора исходного кода компилятор раскладывает на команды промежуточного представления программы на инструкции машинного кода.

1.1 Лексический анализ

Лексический анализ – первый этап компиляции. Объектом изучения лексического анализатора является сам язык, его лексика.

Минимальная единица языка, имеющая значение в рамках описанного языка, называется лексемой. Иначе говоря, лексема – это слово над алфавитом,

принадлежащее некоторому языку. Именно выделение лексем в программе на исходном языке является задачей лексического анализа.

Во время лексического анализа на вход принимается поток символов исходной программы. Анализатор разбирает этот поток на отдельные лексемы, принадлежащие исходному языку программирования. В случае невозможности определить очередную лексему, анализатор прекращает свою работу. Если же разбор завершился успешно, полученный набор лексем передается на вход синтаксическому анализатору.

Для определения лексем используются регулярные выражения. Регулярные выражения – это формальный язык поиска подстроки в исходной строке [4]. Для поиска используется строка-образец (паттерн, шаблон или маска), которая состоит из символов языка исходного текста, а также метасимволов, которые имеют особое значение.

Каждая лексема языка принадлежит некоторому классу лексем, обладающих общим смыслом. Так в языках программирования выделяют такие классы, как «идентификатор», «число», «ключевое слово», «пробельный символ» и другие. Несмотря на то, что лексемы одного класса могут иметь разное значение, они несут одинаковый смысл. Кроме значения самой лексемы, сохраняется и класс, которому она принадлежит. Такая пара называется токеном:

<класс_лексемы, значение_атрибута>

1.2 Синтаксический анализ

Второй этап компиляции — синтаксический анализ [5]. Синтаксический анализ – это процесс, который определяет, принадлежит ли некоторая последовательность лексем языку, порождаемому грамматикой.

Грамматикой называется четверка $G = (N, \sigma, P, S)$, где N, σ – алфавиты нетерминальных и терминальных символов соответственно, причем эти алфавиты не пересекаются ($N \cap \sigma = \emptyset$), P – конечный набора правил грамматики, а S – начальный нетерминал.

Терминал (терминальный символ) – элемент грамматики, представляющий из себя конкретное неделимое значение. Лексемы языка, выделенные на этапе лексического анализа, являются терминалами грамматики языка.

Нетерминал (нетерминальный символ) – объект, обозначающий сущность языка, но не имеющий конкретно определенной формы, символьного

отображения. Нетерминал представляет из себя комбинации терминалов и/или нетерминалов. Среди всех нетерминалов выделяется начальный нетерминал S, называемый начальным символом грамматики. Из него с помощью правил выводятся цепочки терминалов, представляющие предложения языка.

Множество P представляет из себя набор правил замены одного набора (левая часть) символов на другой (правая часть). Говорят, что «правая часть» выводится из «левой части» применением правила.

Совокупность правил и символов грамматики задает древовидную структуру элементов языка. Корнем является начальный элемент грамматики. Для каждого элемента, представляющего «левую часть» какого-либо правила отводится ветвь к потомку, представленному «правой частью» правила. Листьями такой древовидной структуры являются терминалы, так как для них вывод невозможен [6].

Последовательность узлов (цепочек), получаемых после применения очередного правила для разложения начального символа грамматики называется выводом, а последовательность правил, применяемых при выводе называется разбором.

Чтобы поток токенов, полученных при лексическом анализе удовлетворял грамматике языка, должна найтись такая последовательность правил грамматики, по которой начальный символ грамматики может быть разложен на последовательность терминалов, соответствующих этому потоку. Для этого синтаксический анализатор должен построить разбор и сопоставить со входным потоком, полученным на этапе лексического анализа кода.

1.3 Семантический анализ

Семантический анализ — третий этап компиляции кода. Лексический и синтаксический анализ имеют дело со структурными, то есть внешними, текстовыми конструкциями языка. Семантика же, ориентированная на содержательную интерпретацию, имеет дело с внутренним представлением «смысла» объектов, описанных в программе. Семантический анализатор использует синтаксическое дерево и информацию из таблицы символов для проверки соблюдения во входной программе семантических соглашений входного языка.

Для проведения семантического анализа необходимы таблица символов и синтаксическое дерево, полученные на предыдущих этапах компиляции, так как для проверки семантики языка важны как взаимное расположение

элементов, так и их конкретное значение.

При каждом завершении анализа очередной синтаксической конструкции входного языка выполняется ее семантическая проверка на основе имеющихся в таблице идентификаторов данных. Вторая часть семантического анализа проводится в начале этапа подготовки генерации кода. Здесь выполняется полный семантический анализ программы с заполненными таблицами идентификаторов.

Проверка элементарных смысловых норм языка программирования – сервисная функция, предоставляемая компилятором. В отличие от семантических требований языка, строго проверяемых семантическим анализатором, выполнение смысловых норм не является обязательным.

Также задачей семантического анализа является соблюдение области видимости переменных.

1.4 Генерация кода

Завершающим этапом компиляции является генерация кода. Генератор кода конвертирует синтаксически корректную программу в последовательность инструкций машинного кода.

После синтаксического и семантического анализа исходной программы многие компиляторы генерируют явное низкоуровневое или машинное промежуточное представление исходной программы, которое можно рассматривать как программу для абстрактной вычислительной машины.

Если целевым языком является машинный код, то для переменных отводятся соответствующие регистры, в промежуточные операции и команды транслируются в последовательности машинных команд, которые могут быть исполнены процессором компьютера.

Архитектура набора команд целевой машины существенно влияет на сложность разработки генератора кода, дающего высококачественный машинный код. Наиболее распространенными являются архитектуры RISC (reduced instruction set computer) и CISC (complex instruction set computer), а также стековая архитектура [7].

Хотя оптимизация кода не является обязательным этапом компиляции программы, в современных компиляторах ему уделяется большое внимание. Под оптимизацией чаще всего подразумевается ускорение работы целевой программы. Но также под это определение попадает уменьшение расходуемых

ресурсов, памяти и другие критерии.

После синтаксического анализа промежуточный представление может претерпеть некоторое число оптимизаций машинно-независимых оптимизаций. Данный вид оптимизации на оптимальное использование переменных и адресного пространства.

Непосредственно этап генерации кода сопряжен с машинно-зависимой оптимизацией. Такая оптимизация направлена на оптимальное использование возможностей конечной вычислительной машины. В основном все сводится к максимальному использованию имеющихся регистров для избежания излишних операций загрузки и выгрузки данных из регистров.

В итоге генератор кода выдает готовую исполняемую программу на целевом языке. Данная программа осуществляет все операции, которые были описаны во входной программе на исходном языке программирования.

2 Практикум

В практической части описаны задания, выполнение которых поможет закрепить полученные теоретические знания, а также описание необходимых программных средств и исходного языка программирования. В приведенных заданиях внимание уделяется «языковой» стороне теории компиляторов, а именно анализу исходного кода.

2.1 Исходный язык программирования

В качестве исходного языка программирования в практических заданиях используется COOL [8] (Classroom Object-Oriented Language). Cool является функциональным строго типизированным языком, содержащим элементы объектно-ориентированного программирования [9].

Каждый класс содержит в себе некоторый (возможно пустой) наборы полей и функций. Для облегчения задачи построения компилятора, все функции имеют глобальную область видимости. Центральным элементом языка является выражение (expression или expr). Выражение — это любое вычислимое значение языка Cool, состоящее в свою из комбинации других выражений и примитивов. Данный язык реализует только базовые концепции, присущие большинству существующих языков программирования. Несмотря на это, структура языка достаточно разнообразна для наглядной демонстрации работы компилятора.

Лексика языка состоит из следующих классов лексем:

— ключевые слова;
— константы и идентификаторы;
— комментарии;
— пробельные символы.

Грамматика языка Cool может иметь вид, изображенный на рис. 1

2.2 Рекомендованные технические средства

Для генерации анализаторов кода рекомендована связка Flex и Bison [10].

Flex используется для генерации лексических анализаторов (сканеров) [11]. Сканер — это программа, которая распознает лексические паттерны в тексте. Для получения анализатора нужно описать лексику заданного языка и подать ее на вход Flex. Лексика задается в виде пар регулярных выражений, задающих лексемы языка, и правил — команд на языке C/C++ [12], выполняемых

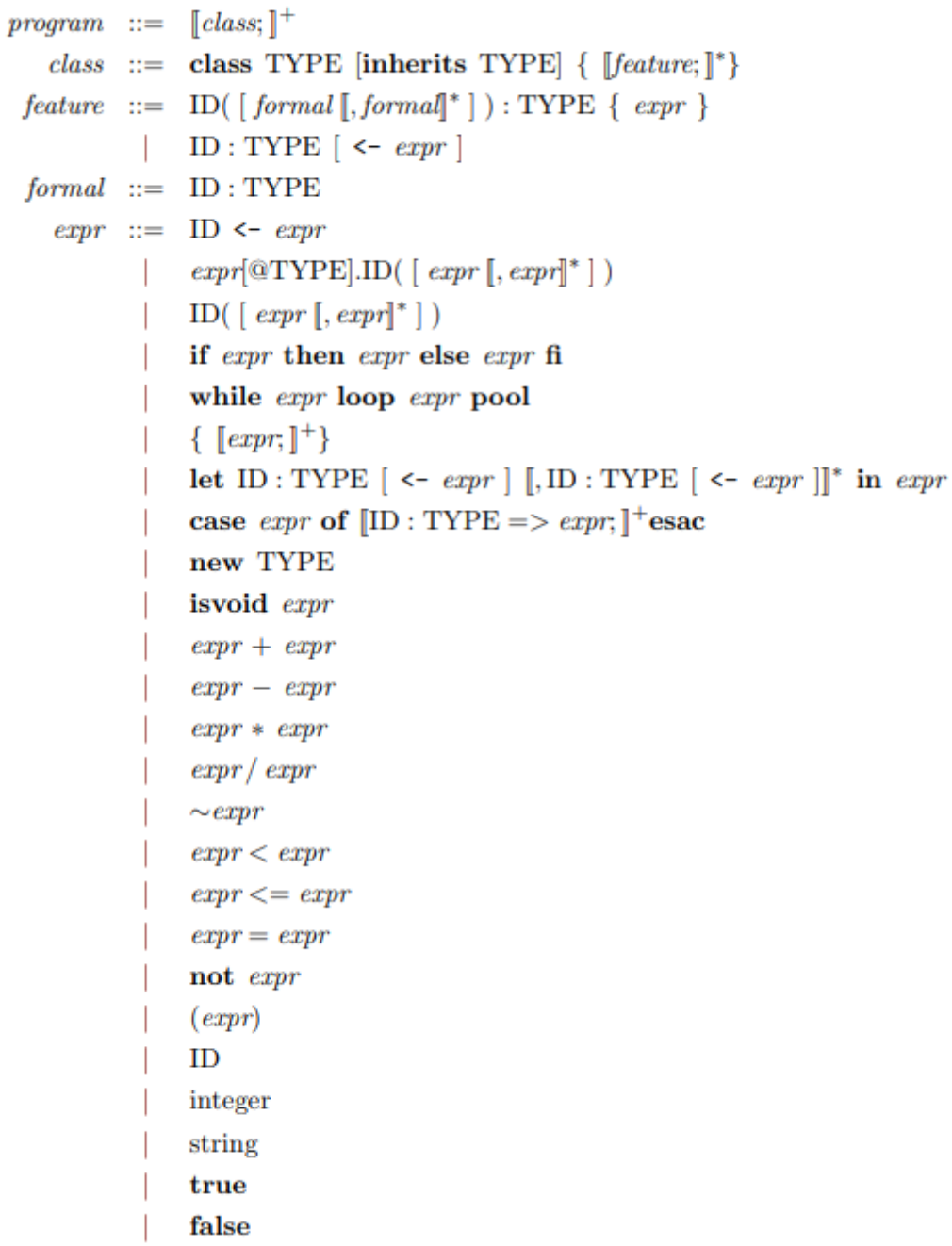

Рисунок 1 – Синтаксис языка Cool

в случае распознавания лексемы.

Bison - генератор синтаксических анализаторов [13]. Он конвертирует заданную контекстно-свободную грамматику в детерминированный LR анализатор. Bison принимает на вход файл с описанием формальной грамматики языка. В полученных файлах реализуется алгоритм синтаксического анализа по набору токенов, полученных на этапе лексического анализа. Для этого в ходе работы синтаксического анализатора происходит вызов лексического анализатора.

2.3 Лексический анализатор

В упражнении на генерацию лексического анализатора требуется реализовать анализаторы для выражений, предложенных в задании, с помощью генератора лексических анализаторов Flex.

В данном разделе описывается предварительная конфигурация проекта, настройка и подключение генератора Flex на стадии сборки проекта. Также здесь приведен пример выполнения задания по разработке и набор тестовых заданий для самостоятельной работы.

Составить регулярные выражения для языка над алфавитом $\sum = \{0, 1\}$, которые соответствуют следующим лексемам:

- (a) множество строк, начинающихся и заканчивающихся на одну и ту же цифру;
- (b) множество строк, представляющих двоичное число, в котором сумма его цифр четная;
- (c) множество строк, содержащих подстроку 10100.

Лексика языка:

```
/* Объявление вспомогательных структур анализатора */
digit        [01]
a                (1{digit}*1)|(0{digit}*0)
b                (0*10*1)*0*
c                {digit}*10100{digit}*
```

Таблица 1 – Тестовый набор упражнения 1

1	a
11	a
110	b
110100	c
101	a
10100	b
101001	a
10	a a

2.4 Синтаксический анализатор

В данном упражнении необходимо реализовать синтаксические анализаторы с использованием лексического анализатора, полученного на основе предыдущего упражнения.

В данном разделе практикума приведена инструкция по настройке проекта и добавлению генераторов Flex и Bison к проекту. Для закрепления полученных теоретических знаний предлагается набор практических заданий по работе с синтаксическими анализаторами, а также разобранный пример выполнения задания.

Пример сформулирован для следующего задания: язык представляет из себя набор строк, заданных на алфавитом $\sum = \{2, 7, +, (,)\}$, которые являются корректными арифметическими выражениями для вычисления нечетного результата в десятичной системе счисления.

Примеры строк, принадлежащих языку:

$$7 \ ; \ (72 + 72) + 27 \ ; \ (((777)))$$

Примеры строк, не принадлежащих языку:

$$22 + 22 \ ; \ (72 + 2)) \ ; \ 22 + 77722 + 22 + + 7$$

Грамматика, описывающая заданный язык:

```
// Блок правил грамматики
%%

//Начальный символ грамматики
input: line input
        | line
        ;

// Нетерминал для очередной строки входного потока
line: S '\n'        {cout << "=> " << $1 << endl; }
        ;

// Нетерминал S - итоговое значение выражения
S:    S '+' E               {$$ = $1 + $3; }
        | E '+' S               {$$ = $1 + $3; }
        | '(' S ')'               {$$ = $2; }
        | D '7'                     {$$ = $1 * 10 + 7; }
        ;

// Нетерминал E - промежуточное выражение
E:          E '+' E               {$$ = $1 + $3; }
        | S '+' S               {$$ = $1 + $3; }
```

```
23          | '(' E ')'                 {$$ = $2; }
24          | D '2'                       {$$ = $1 * 10 + 2; }
25
26  // Нетерминал D - Очередная цифра числа
27  D:                                        {$$ = 0; }
28          | D '7'                       {$$ = $1 * 10 + 7; }
29          | D '2'                       {$$ = $1 * 10 + 2; }
30          ;
31  %%
```

2.5 Семантический анализатор

В данном упражнении необходимо реализовать этап семантического анализа. Требуется проверить исходный код на наличие ошибок и при их наличии составить отчет о полученных ошибках.

В качестве исходного кода рассмотрены программы на языке Cool. Перед выполнением этого задания необходимо провести предварительную конфигурацию лексического и синтаксического анализатора на основе предыдущих упражнений.

Лексика языка:

```
ANY [a-zA-Z0-9.,_\-+(){}~]
SYMBOL [{}\.;:()<>\-]
INTEGER [0-9]+
TRUE true|TRUE
FALSE false|FALSE
KEYWORD class|inherits|new|if|then|else|fi|while|loop|pool
|let|in|case|of|esac|isvoid|of
IDEN [a-z][a-zA-Z0-9_]*
COMMENT (\-\-{SYMBOL}*)|(\*{SYMBOL}*\*)
TYPE [A-Z][a-zA-Z0-9_]*
STRING \"{ANY}*\"
SPACE [ \t\n]
```

Грамматика языка задается на основе схемы языка, приведенной в разделе описания языка.

В данном упражнении разобраны основные виды проверок семантического анализатора. Так как порядок объявления классов и функций не имеет

строгих ограничений, в ходе анализа применяется два вида проверок: одни проверки могут быть проведены непосредственно при разборе исходного кода (использование не объявленной переменной или несоответствие типов), другие должны быть отложены до окончания разбора, так как не могут быть осуществлены без заполненной семантической таблицы (использование функций и классов, так как они могут быть объявлены после вызова). Первый тип проверок описывается в блоке действия для элемента грамматики, описывающего вызов или объявление переменных и выражений, а второй тип вызывается при свертке корневого элемента грамматики языка (в общем случае самой программы).

В текущем разделе практикума приведены примеры для каждого типа проверок, а также настройка приложения для работы с токенами различных типов. В результате реализации приведенных проверок получен семантический анализатор для программ языка Cool. Запуск данного анализатора на некорректной входной программе приводит к получению отчета о полученных ошибках (рис. 2).

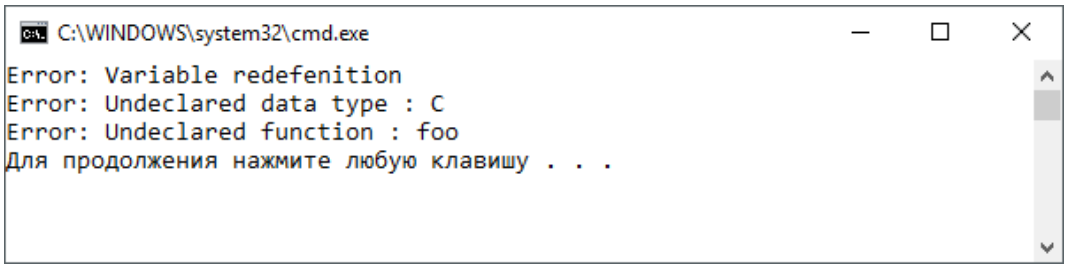

Рисунок 2 – Отчет об ошибках

После выполнения задания по реализации семантического анализатора возможно проведение полного анализа исходной программы на корректность по всем критериям. Отсутствие полученных в ходе анализа ошибок говорит о корректности поданной на вход программы и возможности ее дальнейшего перевода в программу на целевом языке.

ЗАКЛЮЧЕНИЕ

В настоящей работе реализован пакет заданий и методических рекомендаций для обучения по теме «Разработка компиляторов». В полученном методическом пособии структурированы знания, требуемые для понимания принципов действия компиляторов, основных методик и подходов к их разработке. На основе теоретических материалов был разработаны практические задания по дисциплине, предназначенные для отработки навыков реализации компилятора языка высокого уровня с использованием современных технологий.

Задания курса рассчитаны на реализацию с помощью средств генерации анализаторов Flex и Bison на языке C++.

Материалы разработанного курса могут быть использованы для проведения занятий по предмету «Теория формальных языков и трансляций» на 4 курсе обучения по направлениям, связанным с информационными технологиями.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1 *Карпов, Ю.* Теория и технология программирования. Основы построения трансляторов / Ю. Карпов. — Санкт-Петербург: БХВ-Петербург, 2005.

2 Основы компиляторов [Электронный ресурс]. — URL: https://www.intuit.ru/studies/courses/26/26/lecture/799 (Дата обращения 15.05.2018) Загл. с экр. Яз. рус.

3 *Хопкрафт, Д.* Введение в теорию автоматов, языков и вычислений / Д. Хопкрафт, Р. Мотвани, Д. Ульман. — Москва: Вильямс, 2008.

4 *Фридл, Д.* Регулярные выражения / Д. Фридл. — Москва: Символ-Плюс, 2000.

5 *Ахо, А.* Теория синтаксического анализа, перевода и компиляции / А. Ахо, Д. Ульман. — Москва: Мир, 1978.

6 *Волкова, И. А.* Формальные грамматики и языки. Элементы теории трансляции / И. А. Волкова, Т. В. Руденко. — Москва, 1999.

7 *Казакова, И. А.* История вычислительной техники / И. А. Казакова. — Пенза: ПГУ, 201.

8 The Cool Runtime System [Электронный ресурс]. — URL: http://web.stanford.edu/class/cs143/materials/cool-runtime.pdf (Дата обращения 14.05.2018) Загл. с экр. Яз. англ.

9 *Буч, Г.* Объектно-ориентированный анализ и проектирование с примерами приложений / Г. Буч, А. Максимчук, М. Энгл, Б. Янг, Д. Коналлен, К. Хьюстон. — Москва: Вильямс, 2010.

10 *Levine, R.* Flex and Bison / R. Levine. — Wiley, 2009.

11 Lexical Analysis With Flex, for Flex 2.6.2 [Электронный ресурс]. — URL: https:/http://westes.github.io/flex/manual/ (Дата обращения 09.05.2018) Загл. с экр. Яз. англ.

12 *Страуструп, Б.* Язык программирования C++ / Б. Страуструп. — Москва: Бином, 2010.

13 Bison 3.0.5 [Электронный ресурс]. — URL: https://www.gnu.org/software/bison/manual/html_node/index.html (Дата обращения 13.05.2018) Загл. с экр. Яз. англ.