МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМЕНИ Н. Г. ЧЕРНЫШЕВСКОГО»
Кафедра дискретной математики и информационных технологий

РЕАЛИЗАЦИЯ КОМПИЛЯТОРА ДЛЯ ЯЗЫКА ПРОГРАММИРОВАНИЯ LITEASM

АВТОРЕФЕРАТ БАКАЛАВРСКОЙ РАБОТЫ

студента 4 курса 421 группы
направления 09.03.01 — Информатика и вычислительная техника
факультета КНиИТ
Логинова Ильи Владиславовича

Научный руководитель
доцент, к. ф.-м. н. _____ О. В. Мещерякова

Заведующий кафедрой
доцент, к. ф.-м. н. _____ Л. Б. Тяпаев

Саратов 2021

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В настоящее время существует большое количество различный языков программирования [1] [2], каждый из которых предоставляет разный функциональные возможности для разных задач. При выборе языка программирования разработчик должен учитывать не только его возможности, но и особенности архитектуры процессора, действующую операционную систему и особенности аппаратно-программного комплекса. Все эти характеристики и представляют собой платформу для решения конкретных задач.

Компиляция – это процесс перевода текста, написанного на языке программирования, в набор машинных кодов. Компилятор – это программа, выполняющая процесс компиляции. Современные языки программирования по большей частью лишены зависимости от используемой платформы, тогда как компиляторы напрямую зависят от неё. И ещё меньше компиляторов, которые поддерживают несколько различных платформ [3]. Огромное разнообразие аппаратных средств, архитектур и широта решаемых задач инициирует проектирование, разработку и реализацию компиляторов, которые обеспечивают поддержку самой операционной системы, поддержку различных архитектур процессора и ориентируются на особенности аппаратного комплекса. Эти задачи возникают при разработке программного обеспечения микроконтроллеров, т.е. микросхем, предназначенных для управления электронными устройствами. Типичный микроконтроллер сочетает на одном кристалле функции процессора и периферийного устройства, оперативное запоминающее устройство, постоянное запоминающее устройство и способный выполнять относительно простые задачи. Широкий класс различных периферийных устройств требует широкого класса микроконтроллеров с различными физическими и логическими характеристикам.

Современные задачи предполагают использование различных вычислительных систем и комплексов, сильно отличающихся друг от друга, и поэтому интерес представляет компилятор, который поддерживает различные платформы. Сегодня известны несколько таких компиляторов:

— `GNU Compiler Collection` [4] (`GCC`) – набор компиляторов для различных языков: `Ada`, `C`, `C++` [5], `Fortran` и др. Достоинством этого компилятора является возможность поддержки различных архитектур процессора, например: `x86`, `x86-64`, `ARM` и др.

— `MinGW` [6] – программный порт компилятора `GCC` для операционной системы `Windows` [7], обеспечивающий её более полную поддержку.
— `Intel oneAPI Toolkits` [8] – компилятор `C++` для процессоров семейств `x86`, `x86-64` и `IA-64` [9], позволяющий использовать операции, которые характерны только для таких процессоров.
— `IAR C Compiler` [10] – компилятор для языков `C` и `C++`, поддерживающий различные микропроцессоры.

Разработчик может столкнуться с проблемой отсутствия компилятора под определённую используемую платформу. С такой проблемой чаще всего встречаются разработчики программного обеспечения для микроконтроллеров. Решение этой проблемы может происходить 3 способами:

1. ограничение числа поддерживаемых платформ. Но тогда пользователь лишается возможности выбора платформы;
2. использование языков низкого уровня или машинный код, что позволяет выполнять операции, присущие определённой платформе, которые недоступны на других платформах;
3. реализация компилятора для одной определённой платформы, способный поддерживать и другие платформы, но это связано с большими затратами человеко-машинных ресурсов.

Целью данной работы является реализация конфигурируемого компилятора для языка программирования `LiteASM`. Цель данного языка – упростить низкоуровневую разработку под различные платформы, а именно упростить написание программного кода и решить проблему с каталогизацией компонентов кода.

Задачами стали:

– проектирование языка программирования `LiteASM`;
– проектирование конфигурируемого компилятора;
– реализация компилятора.

1 Основная часть

В первом параграфе автор сформулировал и спроектировал основные конструкции языка `LiteASM`, целью которого стало упрощение низкоуровневого программирования, упрощения перехода с языка ассемблера. К свойствам этого языка относятся:

1. Особая интерпретация численных литералов. Язык позволяет указывать численные литералы в разных системах счисления. По умолчанию число интерпретируется в десятичной системе счисления. Для указания числа в двоичной системе счисления необходимо добавить в конец букву `b`. Для указания шестнадцатеричного числа необходимо указать перед числом `0x` и/или в конце букву `h`.
2. Переменные. Переменные объявляются с помощью слова `var`. Они могут быть не определены, тогда компилятор заполнит место, выделенное под эту переменную, нулевыми байтами. Язык не имеет возможности динамического выделения памяти, поэтому все глобальные переменные хранятся в исполняемом файле. Отсутствует приведение типов. Так же стоит упомянуть, что значения локальных переменных могут храниться в части памяти исполняемого файла, а также в регистрах процессора, это определяется статическим способом при анализе жизненного цикла переменной.
3. Функции. Функции объявляются с помощью слова `fun`, могут иметь аргументы и возвращаемое значение. В языке отсутствует возможность переопределения функций и объявления функций с одинаковым именем и различными аргументами. Так же при определении функции можно указать модификаторы `in` и `out`, которые разрешают или запрещают соответственно использование непосредственного кода функции вместо её вызова. Эти же модификаторы могут быть указаны и при вызове функции.
4. Выражения. В языке поддерживаются основные выражения из других высокоуровневых языков, такие как циклы (`for`, `while`, `do...while`), условные ветвления (`if...else`, `switch`) и переходы (`break`, `continue`, `goto`).
5. Возможность импорта файлов, функций и переменных. Функции и переменные в одном файле не имеют доступа к функциям и перемен-

ным из другого файла, пока они не импортированы. Для импорта файла необходимо указать имя пакета файла и через точку имя файла (`std.console`). Для импорта функции необходимо указать имя пакета файла, в котором она находится, через точку имя файла и имя функции (`std.console.print`). Для импорта переменной необходимо указать имя пакета файла, в котором она находится, через точку имя файла и через символ # имя переменной (`ru.loginov.another#a`).

6. Возможность работы с пакетами. Под пакетом здесь мы будем понимать расширяемый маркированный набор файлов. Пакеты решают задачу классификации и каталогизации файлов и исключают коллизию имён. Имя пакета состоит из строк латинского алфавита, разделяемых точками.

Программа, выводящая 100 раз фразу `"Hello, world"` на языке `LiteASM` представлена ниже:

```
package ru.loginov
import std.console.print
import std.console.printInt
import ru.loginov.another#a

var str: byte[1101B] = "Hello, world"

fun main(): int {
  for (var i: int = 0; i < 100; i++) {
    print(str)
  }
  printInt(a + 0xA); return 1
}
```

Очевидно, что код программы значительно проще кода аналогичной программы на языке ассемблер, и конструкции и спецификации языка легко узнаваемы для программиста.

Во втором параграфе работы автор строит компилятор с языка `LiteASM`. Разработанный компилятор реализован на языке `Kotlin` [12] и `Java` [13]. Выбор этих языков обусловлен возможностью работы в условиях различных

платформ. Так же использовался инструмент ANTLR4 [14] для генерации нисходящего лексического и синтаксического анализаторов для языка Java. Такой выбор объясняется следующими причинами:

— удобство работы с абстрактным синтаксическим деревом;
— использование единой нотации для описания лексических и синтаксических анализаторов;
— свободное программное обеспечение;
— предоставление сообщений об ошибках;
— наличие визуальных сред разработки, помогающие при проектировании и разработки грамматики.

Особенностью построенного компилятора стала возможность определять тип компиляции, который может определять способ генерации машинного кода. Например, эта особенность может использоваться алгоритмом генерации машинного кода для выбора формата генерации файла (например: COM или EXE).

При проектировании компилятора были определены основные входные параметры:

— список файлов для компиляции;
— имя архитектуры;
— имя операционной системы;
— тип компиляции (новый дополнительный строковый параметр, который может не использоваться);
— имя функции, которая представляет собой точку входа в приложение;
— настройки компилятора.

Наличие таких входных параметров подразумевают, что компилятор должен адаптироваться под определённую платформу с помощью настроек, и поэтому конструкции языка, такие как типы, операции и функции базовой библиотеки, будут определены самим пользователем. Базовая библиотека – это набор функций, которые, как правило, могут компилироваться напрямую в машинный код. Этот набор функций может легко расширяться. Так же операции языка ассемблер и операции, специфичные для определённой платформы, будут выступать в качестве функций базовой библиотеки. Если какая-либо платформа лишена той или иной функции, то компилятор предоставит возможность написания такой функции. После компиляции по-

лучается файл, содержащий бинарный код под определённую платформу, операционную систему и тип компиляции.

Первым этапом компиляции является лексический и синтаксический анализ с помощью анализаторов, сгенерированных посредством инструмента `ANTLR4`. Лексический анализ необходим для того, чтобы преобразовать набор символов в значащие последовательности, которые называются лексемами. Для каждой лексемы строится выходной токен. Токен – это пара объектов: вид лексемы и её конкретное значение. После проведения лексического анализа компилятор получает поток токенов. Анализ производится только над файлами, которые имеют расширение `.lasm`. Лексический анализатор реализуется `Java`-классом `LiteAsmLexer`.

Цель синтаксического анализа – это создание древовидного представления, которое описывает грамматическую структуру потока токенов. Синтаксическое дерево – это дерево в котором каждая вершина представляет операцию, а дочерние вершины – это аргументы этой операции. Синтаксический анализатор реализуется классом `LiteAsmParser`.

Данные классы используются следующим образом:

— `input` – входящий поток символов;

— `val lexer = LiteAsmLexer(CharStreams.fromStream(input))` – создание объекта класса `LiteAsmLexer` и проведение лексического анализа входящего потока символов;

— `val parser = LiteAsmParser(CommonTokenStream(lexer))` – создание объекта класса `LiteAsmParser` и проведение синтаксического анализа потока токенов полученного после лексического анализа;

— `val result = parser.text()` – получение синтаксического дерева для набора инструкций программы.

При возникновении синтаксической ошибки, пользователь будет проинформирован, где встретилась ошибка, но процесс компиляции будет остановлен. Пример синтаксического дерева для кода `var a: int = 5` (объявление переменной `a` и присвоение ей значения, равного 5) приведён на рисунке 1.

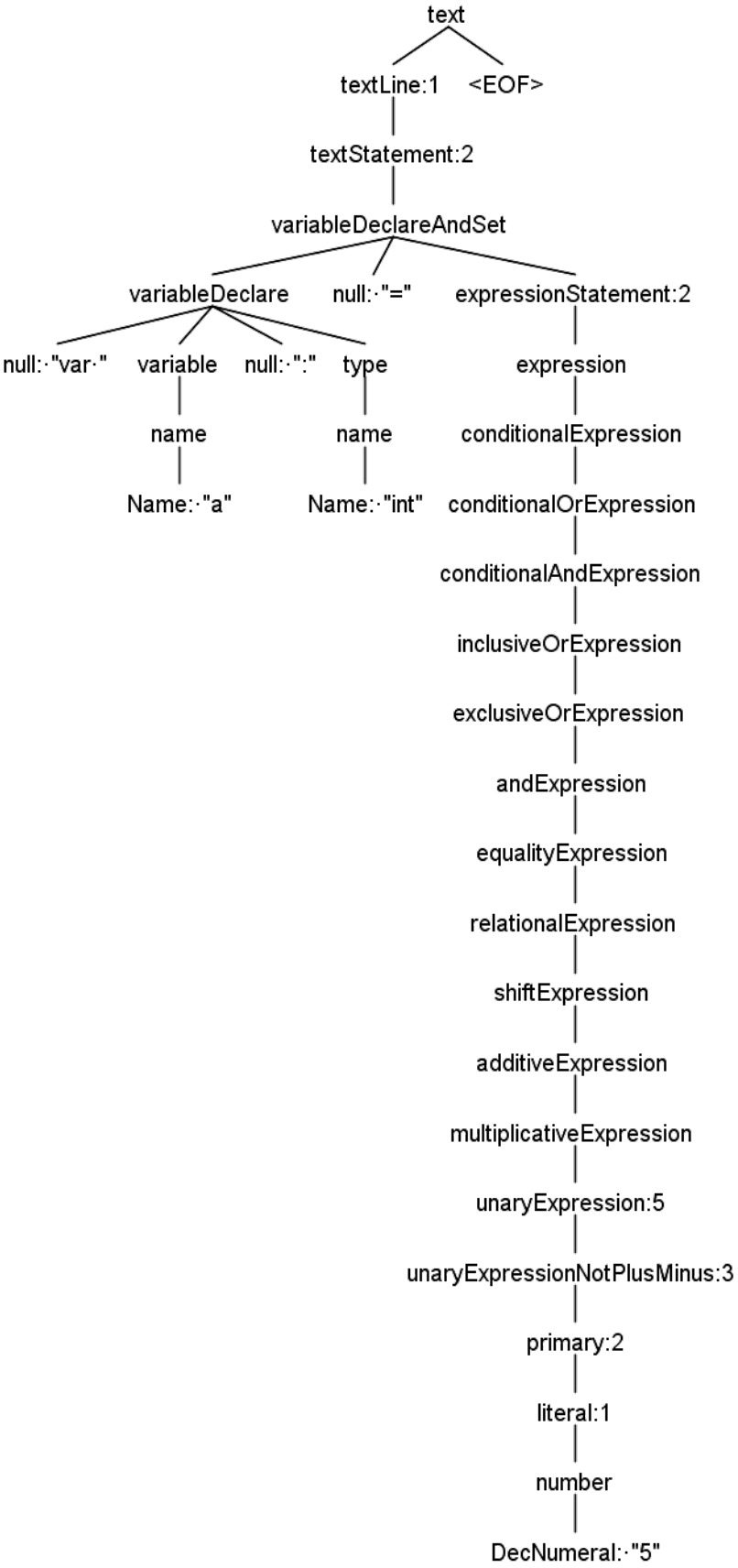

Рисунок 1 – Пример синтаксического дерева

где:

— `text` – текст программы;
— `textLine` – строка текста программы;
— `testStatement` – оператор в строке текста программы;
— `variableDeclareAndSet` – объявление и определение переменной;
— `variableDeclare` – объявление переменной;
— `null:"var "` – последовательность символов, с которой начинается объявление переменной;
— `variable` – имя переменной;
— `type` – имя типа переменной;
— `expressionStatement` – оператор выражения;
— `number` – число;
— `DecNumber` – десятичное число.

Следующим этапом компиляции является семантический анализ, который включает:

— проверку типов;
— проверку, что все используемые переменные существуют;
— проверку, что все используемые функции существуют;
— проверку, что все используемые операции существуют.

После лексического анализа формируется поток токенов, который подвергается синтаксическому анализу. Построенное синтаксическое дерево подаётся на вход семантического анализатора. После семантического анализа получается внутреннее представление.

Перед выполнением семантического анализа компилятор производит загрузку базовой библиотеки. Это происходит с помощью конфигурационных файлов, которые определяют имя функции, входящие и выходящие параметры, их типы, а также машинный код, который будет генерироваться, если эта функция используется в программе.

Далее происходит частичная проверка типов. Это предполагает, что проверяется только объявления функций и переменных на то, используют ли они типы, существующие в языке. Полную проверку типов возможно провести только после генерации внутреннего представления, которое хранит информацию о том, какие операции и функции, с какими параметрами были использованы.

Далее строится внутреннее представление. Внутреннее представление является деревом, вершинами которого будет объекты класса `CompileUnit`, вершины представляют собой операцию, а дочерние вершины – аргументы этих операций, имеющие строгий порядок. Объекты класса `CompileUnit` имеют определённую функциональность и классифицируются в соответствии с ней, а именно:

- `MULTI` - список операций;
- `OPERATION` - операция (арифметическая, логическая и т.п.);
- `FUNCTION_CALL` - вызов функции;
- `VARIABLE_DECLARE` - объявление переменной (используется для объявления локальных переменных);
- `VARIABLE_USE` - использование переменной (как аргумента);
- `NUMBER` - использование числа (как аргумента);
- `REGISTER` - использование регистра (как аргумента);
- `STRING` - использование строкового литерала (как аргумента);
- `NONE` - отсутствие каких-либо действий (используется как аргумент).

И возможны расширения:

- `BASE64` - использование формата `base64` [19] для непосредственной записи байтов;
- `BYTES` - использование массива байт для непосредственной записи;
- `CUSTOM` - тип для расширения.

После семантического анализа и создания внутреннего представления для текста программы компилятор выбирает алгоритм генерации машинного кода, основываясь на выбранной пользователем архитектуре, операционной системе и типе компиляции. На этом этапе происходит полная проверка типов, распределение регистров и проверка возможности использования тех или иных операций и функций. Предложенная версия компилятора предполагает только одну реализацию алгоритма генерации, использующую настройки компилятора, но пользователь имеет возможность добавлять свои алгоритмы генерации в зависимости от поставленной задачи. Алгоритм по умолчанию выглядит следующим образом:

Шаг 1. Анализ внутреннего представления для функции входа.

Шаг 2. Анализ использования локальных переменных и анализ их жизненного цикла.

Шаг 3. Выделение памяти под глобальные и локальные переменные (если требуется).

Шаг 4. Добавление безусловного перехода на адрес точки входа в программу (если требуется).

Шаг 5. Создание файла и генерация машинного кода в соответствии с вычисленными адресами переменных в памяти.

Главной особенностью построенного компилятора является его гибкость и адаптивность, которые реализованы с помощью настроек. Эти настройки обеспечивают независимость компилятора от используемой платформы. Это обеспечивает переносимость компилятора, легкость и прозрачность в программировании, а это в свою очередь позволяет широко использовать язык `LiteASM` и его компилятор для решения широкого спектра задач.

Первая группа настроек компилятора указывается через параметры командной строки:

— `--settings` – путь до файла с настройками;
— `--entry-point` – имя функции начала, т.е. функции с которой следует начать выполнять программу;
— `--input` – пути до файлов для компиляции;
— `--output` – путь до папки для сгенерированных файлов;
— `--architecture` – архитектура под которую будет генерироваться код;
— `--os` – операционная система под которую будет генерироваться код;
— `--type` – тип генератора для генерации кода.

Для гибкой компиляции машинного кода был создан класс `Replaceable`, который определяет в какой набор бит будет генерироваться определённая операция или регистр. Объект этого класс может задаваться с помощью нескольких форматов:

— `base64` – принимает строку в формате `base64` и преобразует её в набор байт;
— `bits` – принимает строку в виде двоичного числа и преобразует её в набор бит;
— `bytes` – принимает набор чисел, которые интерпретируются как набор байтов;
— `class` и `method` – указывает метод в определённом `Java`-классе, который принимает на вход определённый набор параметров (аргументы

операции и контекст компилятора) и возвращает набор бит.

Для определения входных и выходных параметров операции используется класс `Direction`, объекты которого задаются следующими форматами:

- `#M` – Входной аргумент является ссылкой на ячейку в памяти.
- `#R` – Входной аргумент является любым регистром.
- `#R<число>` – Входной аргумент является регистром определённого типа.
- `#L` – Входной аргумент является литералом.
- Остальные комбинации символов интерпретируются как имена регистров.

Вторая группа настроек компилятора, зависящая от платформы, указывается с помощью текстового файла с определённой структурой в формате `json` [20]:

- `charset` – кодировка строковых литералов;
- `header` – объект класса `Replaceable`, который будет генерировать последовательность бит в начале выходного файла;
- `trailer` – объект класса `Replaceable`, который будет генерировать последовательность бит в конце выходного файла;
- `architecture` – имя архитектуры процессора (`x86`);
- `os` – имя операционной системы (`win`);
- `type` – имя типа компиляции;
- `registers` – регистры которые существуют на данной платформе:
  - `name` – имя регистра;
  - `size` – размер регистра в байтах;
  - `types` – типы переменных, которые можно записать в регистр;
  - `kind` – тип регистра (представляет собой числовое значение, которое по-разному интерпретируется разными алгоритмами генерации), может использовать для классификации регистров (используемых в циклах, используемых для локальных переменных и т.д.);
  - `replace` – объект класса `Replaceable`.
- `operations` – операции и выражения, которые могут использоваться компилятором:
  - `format` – описание операции, которое задаёт, какие значение полей и в каком порядке будут генерировать саму операцию:
    - `field` – имя поля, в котором будет содержаться значение;

        - `replace` – объект класса `Replaceable`;
        - `required` – параметр который указывает обязательность данного поля.
    - `list` – список операций и выражений, который имеет определённую структуру:
        - `tag` – имя операции или выражения;
        - `lexeme` – нетерминальный символ для операции;
        - `contract` – определяет спецификацию (указание входных и выходных параметров) операции.
            - `input` – объект класса `Direction` определяющие входящие параметры;
            - `output` – объект класса `Direction` определяющие выходящие значения;
            - `replace` – объект класса `Replaceable`;
            - элементы формата, которые определяют значения полей в формате операции.
- `functions` – список функций, совокупность который будет загружена как базовая библиотека:
    - `package` – имя пакета;
    - `name` – имя функции;
    - `replace` – объект класса `Replaceable`;
    - `contract` – определяет спецификацию функции:
        - `input` – список типов для входящих параметров;
        - `output` – тип возвращаемого результата.

ЗАКЛЮЧЕНИЕ

Для программиста выбор языка и компилятора – непростая задача. Необходимо учитывать много особенностей не только языка, но и архитектуру процессора, особенности операционной системы, состав и архитектуру аппаратного комплекса. Наблюдаемое разнообразие архитектур, вариантов и модификаций операционных систем, огромный список внешних устройств определяет большое количество существующих платформ, что усложняет проектирование и разработку программных комплексов. Разработчики языков программирования по-разному решают эти проблемы. В работе предпринята попытка избежать зависимости от архитектуры, операционной системы и характеристик программного комплекса.

В работе спроектирован и реализован язык программирования целью которого, является упрощение и прозрачность низкоуровневого программирования, удобство программирования микроконтроллеров, возможность поддержки широкого спектра существующих платформ, возможность расширения функций. Этот язык назван `LiteASM`. И для этого языка реализован конфигурируемый компилятор для генерации кода под множество платформ, определяемых самим пользователем. Построенный компилятор позволяет, используя единый код, осуществлять программирование большого класса микроконтроллеров с разными архитектурами, характеристиками и возможностями. Цель работы достигнута и все поставленные задачи выполнены.

Дальнейшая динамика этого компилятора связана с оптимизацией внутреннего представления и машинного кода, поддержки статических и динамических библиотек, поддержки пользовательских библиотек, расширение функциональности за счёт поддержки процессоров с явным параллелизмом операций.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1 Теренс Пратт, Марвин Зелковиц. Языки программирования: разработка и реализация. — 4-е издание. — Питер, 2002. - 690 с. - Яз. Рус.

2 Гавриков М. М., Иванченко А. Н., Гринченков Д. В. Теоретические основы разработки и реализации языков программирования. — КноРус, 2013. — 178 с. - Яз. Рус.

3 Робин Хантер. Основные концепции компиляторов. — М.: Вильямс, 2002. — 256 с. — Яз. Рус.

4 GCC. [Электронный ресурс]. URL: https://gcc.gnu.org/ (дата обращения: 21.03.2021). - Яз. Англ.

5 Standart C++ [Электронный ресурс]. URL: https://isocpp.org/ (дата обращения: 5.04.2021). - Яз. Англ.

6 mingw-w64 [Электронный ресурс]. URL: http://mingw-w64.org (дата обращения: 21.05.2021). Загл. с экр. – Яз. Англ.

7 Windows [Электронный ресурс]. URL: https://www.microsoft.com/ru-ru/windows (дата обращения: 9.05.2021). Загл. с экр. – Яз. Англ.

8 Intel oneAPI Toolkits. [Электронный ресурс]. URL: https://software.intel.com/content/www/us/en/develop/tools/oneapi/all-toolkits.html (дата обращения: 22.05.2021). Загл. с экр. – Яз. Англ.

9 Intel 64 and IA-32 Architectures Software Developer Manuals [Электронный ресурс]. URL: https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html (дата обращения: 2.05.2021). - Яз. Англ.

10 IAR Systems. [Электронный ресурс]. URL: https://www.iar.com/products/free-trials/ (дата обращения: 24.05.2021). Загл. с экр. – Яз. Англ.

11 Альфред В. Ахо, Моника С. Лам, Рави Сети, Джеффри Д. Ульман. Компиляторы: принципы, технологии и инструментарий. - 2 изд. - М.: Вильямс, 2008. - 1184 с. - Яз. Рус.

12 Kotlin Programming Language. [Электронный ресурс]. URL: https://kotlinlang.org/ (дата обращения: 02.05.2021). - Яз. Англ.

13 Java. [Электронный ресурс]. URL: https://java.com (дата обращения: 20.03.2021). Загл. с экр. – Яз. Англ.

14 ANTLR. [Электронный ресурс]. URL: https://www.antlr.org/ (дата обращения: 15.05.2021). - Яз. Англ.

15 Белоусов А. И., Ткачев С. Б. Дискретная математика. - М.: МГТУ, 2006. - 743 с. - Яз. Рус.

16 А. Ахо, Дж. Ульман. Теория синтаксического анализа, перевода и компиляции. Т. 1. - Пер. с англ. В. Н. Агафонова под ред. В. М. Курочкина. - М.: Мир, 1978. - 309 с.

17 LLVM. [Электронный ресурс]. URL: https://llvm.org/ (дата обращения: 20.05.2021). - Яз. Англ.

18 Clang. [Электронный ресурс]. URL: https://clang.llvm.org/ (дата обращения: 22.05.2021). - Яз. Англ.

19 RFC4648 [Электронный ресурс]. URL: https://datatracker.ietf.org/doc/html/rfc4648 (дата обращения: 17.05.2021). - Яз. Англ.

20 JSON [Электронный ресурс]. URL: https://www.json.org/ (дата обращения: 6.05.2021). - Яз. Англ.