Федеральное государственное автономное образовательное учреждение высшего образования «Самарский национальный исследовательский университет имени академика С.П. Королева»

А. В. Гайдель, В. И. Проценко, А.В. Благов

ЛАБОРАТОРНЫЕ РАБОТЫ ПО КУРСУ «ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ»

Самара 2020

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ «САМАРСКИЙ НАЦИОНАЛЬНЫЙ
ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ имени академика С.П. КОРОЛЕВА»
А. В. Гайдель, В. И. Проценко, А. В. Благов

ЛАБОРАТОРНЫЕ РАБОТЫ ПО КУРСУ «ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ»

Рекомендовано редакционно-издательским советом федерального государственного автономного образовательного учреждения высшего образования «Самарский национальный исследовательский университет имени академика С.П. Королева» в качестве учебно-методического пособия по курсу «Технологии программирования».

Самара
Издательство Самарского университета
2020

УДК 004
ББК ??????

Рецензенты: ,

Гайдель А. В. Лабораторные работы по курсу «Технологии программирования»: учебно-методическое пособие / А.В. Гайдель, В.И. Проценко, А.В. Благов. – Самара: Изд-во Самарского университета, 2018. – 101 с.

УДК 004
ББК ??????

Оглавление

Введение

Требования к лабораторным работам

Лабораторная работа выполняется группой от одного до двух студентов. Все студенты в одной команде несут полную ответственность за лабораторную работу и получают за неё одинаковые оценки. Вариант задания назначается случайным образом. Результатом выполнения лабораторной работы является исходный код программы на одном из языков программирования, заранее выложенный в систему контроля версий в общедоступный репозиторий (например, GitHub или GitLab), и печатный отчёт о выполненной работе. Репозиторий должен оставаться доступным до конца семестра. Успешно сданная лабораторная работа может дифференцированно оцениваться по усмотрению преподавателя. Успешная сдача всех лабораторных работ является необходимым условием для допуска к экзамену. Сдавать лабораторную работу можно на лабораторных занятиях по расписанию. В случае, если личное присутствие невозможно, работу можно сдавать удалённо по видеосвязи с демонстрацией экрана в часы проведения занятий. За одно занятие нельзя дважды сдавать одну и ту же лабораторную работу.

В процессе сдачи лабораторной работы преподавателю предъявляется исходный код и запущенная программа для тестирования её работы. Преподаватель может задавать выполнившим работу студентам вопросы по исходному коду программы, по теории, связанной с выполненной работой, а также контрольные вопросы из соответствующего раздела. Работа считается успешно сданной, если:

– программа успешно работает и выполняет все необходимые действия;

– студенты, выполнявшие работу, правильно ответили на все вопросы преподавателя;

– отчёт по выполненной лабораторной работе оформлен в соответствии с требованиями и сдан в печатном виде.

Лабораторная работа может быть выполнена на любом языке программирования в любой среде разработки с использованием любых сторонних библиотек по выбору выполняющих поставленную задачу. При выполнении лабораторной работы не запрещается пользоваться внешними источниками данных, чужим исходным кодом, использованным на законных основаниях, готовыми сторонними библиотеками и любой функциональностью, встроенной в стандартную библиотеку языка программирования.

Отчёт о выполненной лабораторной работе должен быть оформлен в соответствии с требованиями ГОСТ, а также стандарта оформления учебных текстовых документов организации, в которой выполняется лабораторная работа. Он должен содержать следующие обязательные структурные элементы: титульный лист, задание, реферат, содержание, введение, основную часть и заключение. В основной части отчёта должны быть изложены теоретические сведения, положенные в основу лабораторной работы, материалы и методы исследования, а также полученные результаты.

1 Динамические библиотеки

1.1 Задание на лабораторную работу № 1

Требуется реализовать динамическую библиотеку, обладающую заданной функциональностью, а также отдельное приложение с пользовательским интерфейсом, использующее эту динамическую библиотеку и позволяющее пользователю воспользоваться её функциональностью.

Таблица 1 – Варианты задания на лабораторную работу № 1

№	**Функциональность динамической библиотеки**
0	Сложение двух целых чисел.
1	Перевод заданного целого числа в заданную позиционную систему счисления.
2	Получение заданного количества цифр после запятой в заданной обыкновенной дроби.
3	Определение, является ли заданный год високосным.
4	Вычисление суммы цифр заданного целого числа.
5	Подсчёт количества положительных делителей заданного целого числа.
6	Проверка, является ли заданное целое положительное число простым числом.
7	Вычисление площади треугольника на плоскости с заданными целочисленными координатами вершин.
8	Определение, лежат ли точки с заданными целочисленными координатами на плоскости на одной прямой.
9	Решение квадратного уравнения с заданными целочисленными коэффициентами.

10	Нахождение минимального и максимального числа в заданном наборе целых чисел.
11	Нахождение наибольшего общего делителя двух целых чисел.
12	Подсчёт количества различных чисел в заданном наборе целых чисел.
13	Определение наиболее часто встречающегося числа в заданном наборе целых чисел.
14	Перемножение двух заданных матриц.
15	Подсчёт количества символов в текстовом файле с заданным именем.
16	Замена заданной подстроки на другую заданную подстроку в текстовом файле с заданным именем.
17	Поиск наибольшей общей подстроки у двух заданных строк символов.
18	Проверка возможности получения одной заданной строки символов из другой заданной строки символов путём изменения порядка следования символов в первой строке.
19	Подсчёт количества арабских цифр, содержащихся в заданной строке символов.
20	Проверка вхождения заданной строки в другую заданную строку в качестве подстроки.
21	Вычисление суммы квадратов заданного количества первых положительных целых чисел.
22	Нахождение наибольшей цифры в заданном положительном целом числе.
23	Нахождение в заданном наборе целых чисел пары чисел с наименьшей суммой.
24	Определение, является ли заданная конечная последовательность целых чисел строго возрастающей.
25	Нахождение наиболее ранней даты среди дат из

	заданного набора.
26	Определение, являются ли два треугольника с заданными целочисленными координатами вершин подобными.
27	Поиск пары наиболее удалённых точек в заданном наборе точек на плоскости с целочисленными координатами.
28	Поиск пары наиболее близко расположенных точек в заданном наборе точек на плоскости с целочисленными координатами.
29	Вычисление значения функции Эйлера от заданного целочисленного аргумента.
30	Определение, являются ли два заданных целых положительных числа взаимно простыми.
31	Вычисление MurmurHash бинарной строки.
32	Нахождение расстояния Хэмминга между двумя бинарными строками.
33	Нахождение частичной суммы ряда Лейбница.
34	Конвертация одной валюты в другую.
35	Вычисление формулы Байеса.
36	Расчёт производной softmax функции.
37	Вычисление числа ПИ с заданной точностью.
38	Расчёт области восприятия нейрона n-ого слоя свёрточной нейронной сети с заданным размером ядра.
39	Нахождение степеней вершин случайного графа.
40	Расчёт передней и задней стенки глубины резко изображаемого пространства.
41	Функция-оракул предсказывающая исход соревновательной игры.
42	Нахождение расстояния между двумя точками земли, заданными широтой и долготой.

1.2 Теоретический материал

Компиляция и выполнение программы на C и C++ могут быть разделены на несколько этапов:

1. Лексический разбор исходного кода для получения токенов.
2. Построение на основе токенов абстрактного синтаксического дерева.
3. Построение графа исполнения.
4. Оптимизация графа исполнения.
5. Генерация объектного кода.
6. Линковка файлов с объектным кодом в один исполняемый файл.
7. Загрузка исполняемого файла и динамическая линковка средствами операционной системы.

В данном тексте мы обратим внимание на этапы 5-7, следующие за этапом оптимизации графа. Вы можете повторить выполнение команд в докер контейнере или виртуальной машине под управлением Ubuntu или CentOS. Пример запуска докер контейнеров:

```
$ docker run -it centos:latest /bin/bash
$ docker run -it ubuntu:latest /bin/bash
```

Пример конфигурации Ubuntu 20.04 и CentOS 8 под пользователем root:

```
apt update
apt install clang

yum update
yum install clang
```

Рассмотрим пример программы, часть функциональности которой — функцию helper — мы бы хотели выделить в библиотеку:

```
// main.c
#include <stdio.h>

void helper () {
  puts("helper");
}

int main () {
  helper();
}
```

Для компиляции будем использовать clang. Выполним первые шесть этапов компиляции для main.c файла:

```
$ clang main.c
```

Мы получили файл a.out.

```
$ ls
a.out main.c
```

В экосистеме linux существует набор утилит для анализа объектных и исполняемых файлов. Например, мы можем посмотреть заголовок файла с помощью утилиты readelf:

```
$ readelf -h a.out
```

В выводе мы можем увидеть информацию о типе файла, смещение адреса от начала файла, начиная с которого начнется выполнение инструкций и ряд других свойств.

Когда из файла с исходным кодом main.c выделяется функция helper, её реализация переносится в файл helper.c. Для файла с реализацией также создаётся заголовочный файл helper.h с сигнатурой этой функции.

```
// helper.h
void helper();

// helper.c
#include ”helper.h”
#include <stdio.h>

void helper () {
  puts("helper");
}

// main.c
#include “helper.h”

int main () {
  helper();
}
```

Примечание. Поиск заголовочных файлов, имена которых заключены в угловые скобки, начинается с системных директорий, а не с текущей директории, в отличие имён файлов в кавычках.

Теперь, на некоторое время, исключим последний этап — линковку из процесса построения проекта и сфокусируемся на объектных файлах:

```
$ clang -c helper.c # создаётся helper.o
$ clang -c main.c # создаётся main.o
```

К текущему моменту мы получим следующий набор файлов:

```
$ ls
helper.c helper.h helper.o main.c main.o
```

При просмотре заголовков файлов с расширением .o мы увидим другой тип — relocatable file:

```
$ readelf -h main.o
```

Объектные файлы — это практически исполняемые файлы. Они содержат машинный код, однако адреса памяти, которые фигурируют в нём не окончательные и требуют релокации при формировании окончательного исполняемого кода. Также объектные файлы содержат метаданные об адресах переменных и функций, называемых в контексте объектных файлов символами. Символы хранятся в ассоциативной структуре, называемой таблицей символов.

Если вы забудете указать во время линковки один из объектных файлов, линковщик не сможет найти символы функции, которые в них определены:

```
$ clang main.o
main.o: In function `main':
main.c:(.text+0x7): undefined reference to `helper'
clang-5.0: error: linker command failed with exit code 1
(use -v to see invocation)
```

Посмотрим таблицы символов с помощью утилиты nm:

```
$ nm helper.o
$ nm main.o
```

Что нам интересно из выдачи nm main.o это то, что символом помечен как U — неопределённый. Это означает, что к символу где-то обращаются в коде, но его определения в переданных

утилите файлах нет.

Примечание. Если вы не знаете, что делает linux утилита, одним из лучших источников информации о ней является man документация. Аналогичная документация поддерживается в Windows в powershell.

```
$ man nm
```

Для линковки проекта и создания исполняемого файла передайте в качестве аргументов все полученные ранее объектные файлы. После успешной линковки в исполняемом файле вы увидите объединённую таблицу символов.

Воспользуйтесь утилитой otool или objdump, чтобы посмотреть на машинный код объектных файлов.

Для статической линковки символы всех объектных файлов должны быть уникальными, так как по сути они являются уникальными адресами памяти по которым происходит чтение или запись в случае с переменными, и на которые перепрыгивает указатель исполнения в случае функций. В языках аналогичных C++, поддерживающих перегрузку функций, для удовлетворения требования уникальности символов символы приходится обогащать дополнительной информацией, взятой из сигнатуры функций. Посмотрите на таблицу символов объектного файла следующего кода:

```
 void doSomething(bool save) {}
void doSomething(int n) {}
int main() {}

$ clang class.cpp -std=c++11
$ nm a.out
```

В GNU у nm утилиты есть ключ --demangle, который вы

можете передать в качестве аргумента для раскодирования символов функций.

Такое искажение может создавать проблемы при выделении функциональности в библиотеку, которая будет использоваться из других языков. Функции, которые планируется использовать через FFI (Foreign Function Interface), в заголовочных файла C++ можно предварить словом extern “C” для отключения автоматического искажения данного символа. Но тогда ответственность за уникальность символа полностью лежит на вас.

Примечание. Слово extern позволяет обращаться к символу из других файлов, сделать его глобальным. Компилятор, работая с каждым файлом по отдельности, сможет получить информацию только о типе extern символа, но не о его местоположении, которое он доверит найти линковщику в таблицах символов объектных файлов.

Переведем взгляд непосредственно на подготовку статических и динамических библиотек. На этот раз объединим объектные файлы не в исполняемый файл, а в архив, то есть в статическую библиотеку. Архив файлов чаще всего ассоциируется со сжатием, например с zip архивом. Но в нашем случае архив будет несжатым.

Мы можем использовать утилиту ar для создания и манипулирования архивами. Для наглядности мы покажем её работу на двух функциях helper, объявленных в разных файлах.

```
$ clang -c helper1.c helper2.c
$ ar -rv libhello.a helper1.o helper2.o
```

Запуск ar с флагом -r создаёт архив с именем libhello.a и добавляет файлы helper1.o и helper2.o в его индекс. Флаг -v (verbose) указывает на подробную выдачу сообщений в

терминал.

Посмотрим на этот файл через утилиты file и nm:

```
$ file libhello.a
libhello.a: current ar archive

$ nm libhello.a
helper1.o:
0000000000000000 T helper1
                 U puts

helper2.o:
0000000000000000 T helper2
                 U puts
```

Теперь, когда у нас есть статическая библиотека (.a в Linux, .lib в Windows), мы можем прилинковать её к файлу, использующему функции библиотеки.

```
$ clang main.o libhello.a
```

Наш компилятор понимает, как использовать индекс архива, чтобы взять нужные функции и собрать конечный исполняемый файл. Линковщик во время построения исполняемого файла берёт из библиотек только необходимые части.

Статические библиотеки позволяют выделить общую для нескольких проектов функциональность и переиспользовать этот код. Протестированный, часто используемый код имеет ещё большую ценность. Таким образом абстрагирование функциональности снижает вероятность появления ошибок в новом проекте и снижает стоимость его разработки. Отметим, что использование библиотечной функциональности не требует доступа к её исходному коду, что может быть важно с правовой точки зрения.

Когда мы имеем в наличии статические библиотеки со всеми

необходимыми функциями на этапе линковки, то в результате мы получаем программу, которая содержит в себе всё необходимое. Это делает распространение и установку приложений лёгким процессом. Однако размер исполняемого файла может стать очень большим. Каждое обновление в этом случае потребует передачи большого объема информации. Также избыточно будет расходоваться оперативная память при запуске нескольких копий процесса в операционной системе, так как каждый процесс будет загружать весь исполняемый файл.

Решением этих проблем являются динамические библиотеки. Программа в этом случае распространяется в виде нескольких файлов: одного исполняемого файла и набора библиотек, которые могут доставляться из разных источников.

Динамические библиотеки позволяют перенести этап линковки с момента компиляции проекта на момент исполнения программы. В этом случае линковка возлагается на операционную систему. Динамические библиотеки как правило имеют разное расширение файла в разных ОС. В Linux это .so (shared library)[13], в Windows это .dll (dynamic linked library)[14], в OSX это .dylib. Вот пример того, как её можно создать:

```
$ clang -shared -fpic helper1.c helper2.c -o libhello.so
$ file libhello.so
libhello.so: ELF 64-bit LSB shared object, x86-64,
version 1 (SYSV), dynamically linked, not stripped
```

Первый флаг -shared говорит линковщику, что нужно создать специальный файл — динамическую библиотеку. Второй флаг -fpic преобразует абсолютные адреса в относительные, что даёт возможность разным процессам загружать библиотеку в разные адреса виртуальной памяти. Теперь попробуем скомпилировать проект с динамической библиотекой и посмотреть на работу

линковщика операционной системы (elf interpreter и libc).

```
$ clang main.c libhello.so
$ ./a.out
```

В Linux системе процесс линковки при запуске завершится неудачно, линковщик не найдёт библиотеку. “Вот же она!”, - скажете вы. Чтобы найти библиотеку загрузчик linux системы вначале полагается на список путей, разделенных двоеточием, хранимых в одном из атрибутов динамических секций ELF файла RPATH и RUNPATH, а также в переменной среды LD_LIBRARY_PATH. Затем поиск продолжается среди кэша путей к библиотекам /etc/ld.so.cache. Поиск завершается в доверенных системных каталогах /lib и /usr/lib [15]. Вы можете воспользоваться переменной среды LD_DEBUG, чтобы увидеть в каких директориях загрузчик пытается открыть файл библиотеки.

```
$ LD_DEBUG=libs ./a.out
```

Действуют следующие приоритеты для атрибутов и переменных среды:

1. Если установлена переменная RUNPATH, то RPATH игнорируется
2. Переменная среды LD_LIBRARY_PATH имеет приоритет в порядке поиска перед RUNPATH

RPATH крайне удобен для тех программ, которые распространяются со своими динамическими библиотеками, так как позволяет избежать их установки в систему. Секция RPATH считается устаревшей и сейчас под rpath аргументом в различных утилитах подразумевается RUNPATH.

Установим путь к файлу библиотеки в атрибуте RUNPATH,

проверим, что путь установлен, динамическая библиотека находится и программа работает.

```
$ clang main.c libhello.so -Wl,-rpath ‘./’
$ readelf -d ./a.out
$ ldd ./a.out
$ ./a.out
```

Если мы собираемся установить библиотеку в системную директорию, безопаснее в этом случае провести тестирование добавлением текущей директории к директориям поиска на время запуска приложения.

```
$ clang main.c libhello.so
$ LD_LIBRARY_PATH=. ./a.out
```

Во время сборки исполняемого файла компилятор также должен знать, где найти библиотеку. Вы можете указать полный путь к файлу, как было сделано выше. Но чаще всего для библиотек используются специальные флаги компилятора:

-l<имя библиотеки без префикса lib и расширения файла .so>

-L <список путей, в которых следует искать библиотеку>

-I <список путей, в которых следует искать заголовочные файлы>

Компилятор в linux также как и загрузчик полагается на соглашения, согласно которым системные библиотеки ищутся в /lib и /usr/lib, а их сопутствующие заголовочные файлы в /include и /usr/include. В случае, если наша библиотека установлена в систему, достаточно указать -lhello.

При построении динамической библиотеки в Windows для файла, который содержит список экспортированных символов функций и переменных, а также метаинформацию о DLL используется то же расширение, что и для статической

библиотеки .lib. Когда линковщик выполняет построение приложения, при нахождении первого обращения к функции или данным из DLL метаинформация о ней добавляется в секцию зависимостей исполняемого файла, заставляя Windows при его загрузке автоматически загрузить библиотеку. Кроме того увеличивается таблица импортов исполняемого файла, добавлением переадресации вызова функции заглушки на функцию из библиотеки.

Разработчик может пойти дальше в направлении динамичности, и, ценой отказа от информации о сигнатурах функций, загружать динамическую библиотеку в любой момент работы приложения.

Такая возможность есть как в Linux, так и в Windows. Посмотрим как это выглядит.

```
// main.c
#include <dlfcn.h>

int main() {
    void *hellolib = dlopen("libhello.so", RTLD_LAZY);
    void (*helper)();
    if (hellolib != 0) {
        *(void **)(&helper) = dlsym(hellolib, "helper1");
        helper();
    }
}
```

Соберём и запустим проект:

```
$ clang main.c -L./ -lhello -ldl
$ LD_LIBRARY_PATH=. ./a.out
```

Имена функций и соответствие аргументов типам функций в этом случае не могут быть статически проверены. Ответственность за указание правильных имён и аргументов

лежит на разработчике. Преимуществом такого подхода является возможность генерации имён функций на основе данных. А недостатком, увеличение вероятности появления сложно отлаживаемых ошибок и необходимость в дополнительном обвязочном коде.

Итак, осталось разобраться в наиболее явных различиях динамических библиотек в Windows и Linux. DLL и SO оба являются контейнерами для исполняемого кода и данных. Они могут быть загружены в область памяти других программ из которой можно запускать функции и к которой можно обращаться за данными. Загрузка библиотеки происходит во время исполнения: либо при старте приложения, либо когда в приложение явно запрашивает загрузку вызовом LoadLibrary в Windows или dlopen в Linux. Загрузка dll или so также вызывает загрузку тех библиотек от которых она зависит. А если зависимая библиотека уже загружена, используется она и повторной загрузки не происходит.

Функции из одной dll, shared object или исполняемого файла могут вызывать друг друга напрямую, так как все они всегда находятся на одном расстоянии относительно друг друга в памяти. Но dll и shared object могут быть загружены по любому адресу виртуальной памяти. Поэтому адреса вызовов функций пересекающих границы библиотек должны задействовать некоторый “релокационный” механизм для формирования актуального адреса. Тот же механизм работает и для символов переменных.

Как DLL так и shared objects обладают этим механизмом, но у shared objects он более мощный и позволяет обрабатывать транзитивные ссылки на данные, которые содержатся в структурах и классах, к примеру. Отсутствующей операцией в Windows является сдвиг адреса символа после релокации.

В ОС Windows символы явно не импортируются и не

экспортируются из DLL. Когда DLL создаётся, экспортируемые символы должны быть помечены аннотацией “__declspec(dllexport)” или приведены в файле определений. Когда используется DLL, к каждому импортируемому символу обычно обращаются через библиотеку импорта, которая содержит заглушки, формирующие слой между программой и DLL библиотекой. Именно здесь производятся релокации.

Эти дополнительные шаги делают процесс переноса Linux библиотек на Windows более сложным. Символы shared библиотек экспортируются по умолчанию и импортируются автоматически, если они нужны. Любой дополнительный код необходимый для релокаций генерируется автоматически линковщиком.

Другим важным отличием DLL и SO библиотек является отсутствие в Windows аналога RPATH. После того, как программа загружена директория может быть добавлена к директориям поиска библиотек вызовом SetDllDirectory или AddDllDirectory. Но, к сожалению, это помогает только для библиотек загружаемых вручную и не влияет на механизм загрузки библиотек при запуске программы.

Руководство создания проекта динамической библиотеки в Visual Studio вы можете найти в статье Пошаговое руководство. Создание и использование собственной библиотеки динамической компоновки (C++) на сайте https://docs.microsoft.com/.

1.3 Контрольные вопросы

1. Что такое библиотека в программировании?
2. Что такое динамическая библиотека?
3. Чем отличаются статические и динамические

библиотеки?

4. Каким образом организуется использование динамической библиотеки в исходном коде программы?
5. В чём заключаются преимущества и недостатки распространения программ в виде динамических библиотек?

1.4 Пример выполнения лабораторной работы

Ниже приведён пример исходного кода на языке C++ выполненного нулевого варианта данной лабораторной работы. Программы разработаны для операционных систем семейства Windows и могут быть собраны с использованием интегрированной среды разработки Microsoft Visual Studio. Заголовочный файл `APlusBLib.h` может быть использован как при сборке динамической библиотеки, так и при сборке основного приложения, использующего динамическую библиотеку. Файл `APlusBLib.cpp` содержит реализацию функциональности динамической библиотеки. Файл `Main.cpp` содержит исходный код основного приложения с консольным пользовательским интерфейсом.

Листинг 1. Содержимое файла `APlusBLib.h`

```
#pragma once

#ifdef LIB_EXPORT
#define LIB_API __declspec(dllexport)
#else
#define LIB_API __declspec(dllimport)
#endif

extern "C" LIB_API int APlusB(int a, int b);
```

Листинг 2. Содержимое файла APlusBLib.cpp

```
#define LIB_EXPORT

#include "APlusBLib.h"

int APlusB(int a, int b)
{
	return a + b;
}
```

Листинг 3. Содержимое файла Main.cpp

```
#include <iostream>
#include <string>
#include "APlusBLib.h"

const char EOLN = '\n';
const char YES_CHAR = 'Y';
const char NO_CHAR = 'N';
const int LEFT_BOUND = -1000000000;
const int RIGHT_BOUND = +1000000000;
const std::string ABOUT_MESSAGE = "A+B";
const std::string CONTINUE_MESSAGE =
"Continue? (Y/N)>";
const std::string INCORRECT_MESSAGE =
"Input is incorrect. Try again>";
const std::string INPUT_MESSAGE = "Input an integer>";
const std::string OUT_OF_BOUNDS_MESSAGE =
"This number is out of bounds";
const std::string OUTPUT_MESSAGE = "Result: ";
const std::string SKIP_CHARACTERS = " ";

void ClearInputStream(std::istream &in)
{
	in.clear();
	while (in.peek() != EOLN && in.peek() != EOF)
	{
		in.get();
	}
}

int Seek(std::istream &in)
```

```
{
	while (in.peek() != EOLN &&
SKIP_CHARACTERS.find((char)in.peek()) !=
std::string::npos)
	{
		in.get();
	}
	return in.peek();
}

bool CheckBounds(int n)
{
	bool ok = (LEFT_BOUND <= n && n <= RIGHT_BOUND);
	if (!ok)
	{
		std::cout << OUT_OF_BOUNDS_MESSAGE << " ["
<< LEFT_BOUND << ", " << RIGHT_BOUND << "]" <<
std::endl;
	}
	return ok;
}

int ReadInt(std::istream &in)
{
	std::cout << INPUT_MESSAGE;
	int ans;
	in >> ans;
	while (!in || Seek(in) != EOLN
|| !CheckBounds(ans))
	{
		ClearInputStream(in);
		std::cout << INCORRECT_MESSAGE;
		in >> ans;
	}
	return ans;
}

bool NeedContinue(std::istream &in)
{
	std::cout << CONTINUE_MESSAGE;
	char ans;
```

```
	in >> ans;
	while (!in || Seek(in) != EOLN || ans != YES_CHAR
&& ans != NO_CHAR)
	{
		ClearInputStream(in);
		std::cout << INCORRECT_MESSAGE;
		in >> ans;
	}
	return ans == YES_CHAR;
}

int main()
{
	std::cout << ABOUT_MESSAGE << std::endl;
	bool cont = true;
	while (cont)
	{
		int a = ReadInt(std::cin);
		int b = ReadInt(std::cin);
		std::cout << OUTPUT_MESSAGE << APlusB(a, b)
<< std::endl;
		cont = NeedContinue(std::cin);
	}
	return 0;
}
```

2 Межъязыковые форматы хранения и передачи данных

2.1 Задание на лабораторную работу № 2

Требуется разработать приложение или программный комплекс, обменивающийся данными по сети в формате JSON, XML или Protocol Buffers.

№	Вариант задания
0	Система мгновенного обмена сообщениями между двумя пользователями.
1	Сетевая игра «Данетки» для двух игроков. Первый игрок загадывает слово, а второй пытается угадать, задавая первому игроку вопросы, ответами на которые могут быть только «да» или «нет».
2	Система обмена файлами между двумя пользователями. Система должна позволять двум пользователям, физически находящимся за разными вычислительными устройствами, обмениваться файлами по сети.
3	Сетевая игра «Крестики-нолики» для двух игроков. Два игрока по очереди ставят в пустые клетки поля символы, причём первый игрок ставит крестики, а второй – нолики. Игрок выигрывает, если три клетки по горизонтали, по вертикали или по диагонали становятся заполненными его символами.
4	Игра “Который логический час?”. Игроки хранят целочисленное значение времени. Каждый ход игрок

	отвечает на входящие сообщения если они есть, а затем может спросить у любого другого игрока “Который час?”, либо пропустить ход. Каждое сообщение сопровождается текущим временем с добавлением единицы. Игрок при ответе выбирает максимальное значение между своим временем и временем входного сообщения + 1.
5	Сетевая игра «21» для двух игроков. Игроки ходят по очереди. Каждый ход игроку равновероятно показывается случайное число от 1 до 11. Он может добавить это число к сумме, либо закончить игру. Когда игрок набирает в сумме не менее 21, то игра для него заканчивается автоматически. Когда оба игрока закончили игру, игроки, у которых сумма больше 21, проигрывают, а среди остальных игроков выигрывает тот, у кого сумма больше.
6	Сетевая игра «Стишок» для четырёх игроков. Каждый игрок по очереди пишет строчку общего стиха на основе строчки предыдущего игрока. Остальные строки он не видит. Игроки проходят фиксированное количество итераций, и программа выводит общий стих, который в итоге получился.
7	Сетевая игра «Робот» для двух игроков. Каждый игрок управляет роботом, который обладает некоторым количеством здоровья, энергии и некоторым набором способностей, связанным с нанесением урона роботу другого игрока, с защитой от повреждений, наносимых другим игроком, и с восстановлением энергии. Роботы могут по очереди использовать свои способности, тратя на них энергию. Когда здоровье одного из роботов перестанет быть положительным целыми числом, другой робот выигрывает.
8	Кооперативная игра “Удалённая стыковка”. В игру играют два игрока: один на земле, другой на корабле,

	стыкующемся с космической станцией в точке лагранжа L1. Оператор на земле получает положение и вращение корабля, лётчик не знает положения, но может управлять кораблём и получать сообщения от оператора. Информация распространяется с задержкой. Игроки выигрывают при совершении успешной стыковки.
9	Сетевая игра «Команда» для двух игроков. Поле 8×8 клеток, на котором все объекты отображаются обоим игрокам. Каждый игрок управляет бойцом, который либо ходит в одном из четырёх направлений, либо стреляет в одном из четырёх направлений. При стрельбе уничтожается первый объект, находящийся в том направлении. Игроки ходят по очереди. На поле случайно генерируются враги. Если враг достигнет клетки с бойцом, то боец погибает. За ход все враги перемещаются на одну клетку в случайном направлении. Цель игроков уничтожить всех врагов. Игроки проигрывают, если оба погибают.
10	Сетевая игра «Поиск вслепую» для трёх игроков. Для каждого игрока генерируется случайное положение на двумерной плоскости так, чтобы они все находились на одинаковом расстоянии от награды. Каждому игроку выводится расстояние до награды на каждом ходу. Ход игрока заключается в выборе угла поворота, по которому игрок передвинется на одну единицу. Выигрывает игрок, который первый окажется от награды на расстоянии, не превышающем 1.
11	Сетевая игра со спичками для двух игроков. Имеется 100 спичек. Игроки ходят по очереди. Каждый игрок за свой ход может убрать от 10 до 20 спичек включительно. Проигрывает игрок, который не может сделать ход.
12	Удалённый интерпретатор арифметических

	выражений.
13	Сетевая игра «Числовой лабиринт» для двух игроков. Игра начинается с числа 1. Каждый игрок в начале игры получает 5 случайно сгенерированных арифметических операций с 5 случайно сгенерированными целыми числами, которые будут являться правыми операндами соответствующих операций. Игроки ходят по очереди. За свой ход игрок может применить одну из своих операций к текущему числу. Выигрывает игрок, первым получивший число 42.
14	Сетевая игра «Гомоку» для двух игроков. Игроки ходят по очереди. За свой ход игрок может поставить символ своего цвета на свободную клетку поля размером 8×8 клеток. Выигрывает игрок, который поставит непрерывный ряд из пяти своих символов по вертикали, по горизонтали или по диагонали.
15	Сетевая игра «Верю, не верю» для двух игроков. У каждого игрока изначально имеется три карты с надписью 1, две карты с надписью 2 и одна карта с надписью 3. Игроки ходят по очереди. Если ни одной карты не выложено, то игрок может выложить одну из своих карт в закрытую и назвать число от 1 до 3. Если на столе уже лежат некоторые карты, то игрок может поверить в то, что все карты имеют названную изначально надпись и выложить ещё одну карту в закрытую, либо не поверить и проверить последнюю карту, выложенную другим игроком. Если при этом надпись на карте совпадает с заявленной, то проверявший игрок забирает все лежащие на столе карты в руку, в противном случае, их забирает другой игрок. Выигрывает игрок, оставшийся без карт.
16	Сетевая игра «Провинции» для четырёх игроков. На клетчатом поле 8×8 в каждой клетке может находиться

	произвольное количество воинов противника. Изначально у каждого игрока есть по одному воину в одном из углов поля. Игроки ходят по очереди. В начале хода игрока в каждой клетке поля появляется столько воинов этого игрока, сколько их там уже есть. За свой ход игрок может переместить некоторое количество своих воинов из одной из клеток в соседнюю с ней клетку. Если в одной из клеток оказываются воины обоих игроков, то количество воинов каждого игрока в этой клетке уменьшается на наименьшее из количеств воинов каждого из игроков в этой клетке. Побеждает игрок, уничтоживший всех воинов противника.
17	Сетевая игра «Луноходы» для двух игроков. Каждый игрок управляет луноходом, перемещающимся по шахматной доске 8×8, на которой случайным образом расставлены препятствия. Игроки ходят по очереди. В начале хода игрок равновероятно получает от 1 до 6 очков действия. Перемещения лунохода в соседнюю клетку стоит 1 очко, а выстрел в некотором направлении – 3 очка. При выстреле, первый объект, находящийся в заданном направлении от лунохода, уничтожается. Побеждает игрок, уничтоживший луноход противника.
18	Сетевая игра «Камень, ножницы, бумага» для двух игроков. Каждый игрок в тайне от другого выбирает камень, ножницы либо бумагу, после чего определяется победитель. Камень бьёт ножницы, ножницы бьют бумагу, бумага бьёт камень.
19	Сетевая игра «Пушки» для двух игроков. Игроки имеют некоторую координату на числовой прямой. Они по очереди стреляют друг в друга, выбирая угол полёта снаряда. Снаряд летит обладает некоторой массой и притягивается к земле некоторой силой. Выигрывает игрок, попавший снарядом в противника.

20	Сетевая игра «Бойцы» для двух игроков. У каждого игрока есть 8 бойцов, различной силы, измеряющейся целыми числами от 1 до 8. Игроки в тайне друг от друга расставляют бойцов в желаемой последовательности. Когда бойцы разных игроков оказываются друг напротив друга, то боец с меньшей силой уничтожается, и соответствующий игрок зарабатывает 1 очко. Если в такой ситуации силы бойцов одинаковые, то уничтожаются оба бойца, и ни один из игроков не получает очков. Выигрывает игрок, который набрал больше очков, чем другой игрок.
21	Игра “Выбери лидера”. Игроки предлагаю кандидатуру из числа игроков. Голосование должно завершиться за определённое время. Вам мешают постоянные зависания клиентской программы и сбои в сети.
22	Сетевая игра «Звёздное ремесло» для двух игроков. Изначально у каждого игрока есть 0 кристаллов, 5 рабочих и 0 воинов. Игроки ходят одновременно, они не видят количество кристаллов, рабочих и воинов другого игрока. В свой ход игрок может либо потратить некоторое количество имеющихся кристаллов на строительство рабочих или воинов, либо напасть на другого игрока. Производство рабочего стоит 5 кристаллов, а воина – 10 кристаллов. В начале хода игрок получает столько кристаллов, сколько у него рабочих. Если один из игроков решил напасть на другого игрока, то он выигрывает, если у него строго больше воинов, чем у другого игрока, и проигрывает в противном случае.
23	Сетевая игра «Запомнить последовательность» для двух игроков. Один игрок выставляет последовательность пяти цветных шариков, а другой игрок должен запомнить последовательность и воспроизвести через пять секунд.

	Каждый ход игроки меняются ролями. Игрок проигрывает, если воспроизвёл последовательность неправильно.
24	Коллекционная карточная игра. У каждого игрока есть колода карт и несколько карт в руке. Каждый ход игрок набирает карты в руку из своей колоды и может разыгрывать некоторые из них, чтобы уменьшить количество карт в колоде противника. На каждой карте написано, какой эффект она производит. Игра заканчивается, когда один из игроков остался без карт в колоде.
25	Сетевая игра «Города» для двух игроков. Игроки по очереди, не повторяясь, называют города, начинающиеся на букву, на которую закончилось название города, названного другим игроком в прошлый раз. Игрок проигрывает, если не может назвать город, либо назвал город, ранее уже названный одним из игроков.
26	Сетевая игра «Угадай число» для 2-х игроков. Первый игрок загадывает целое число, а второй пытается угадать. Второй игрок может делать предположения о том, какое число загадал первый игрок, а тот отвечает ему, больше его число или меньше, чем предположенное вторым игроком.
27	Сетевая игра «Мафия» для восьми игроков. В начале игры два игрока скрыто от других игроков получают роль мафии. В начале каждого хода игроки голосуют, кто из игроков является мафией, и исключают из игры того игрока, за которого проголосовало больше игроков. В конце каждого хода игроки, получившие роль мафии, скрыто от других игроков голосуют, кого из игроков исключить из игры. Игра заканчивается победой мафии, если в игре остались только игроки, получившие роль мафии, либо победой честных жителей, если все игроки,

	получившие роль мафии, исключены из игры.
28	Сетевая игра «Пешки» для 2-х игроков. Игра представляет собой модифицированные шахматы, на доске 8×8 клеток расположены по 8 пешек у каждого игрока на втором ряду доски со стороны каждого игрока. Игроки ходят по очереди, перемещая одну из своих фигур по правилам, по которым пешки перемещаются в шахматах. Игрок выигрывает, если уничтожил все пешки противника, либо если своим ходом передвинул пешку к границе поля со стороны противника.
29	Сетевая игра «Кони» для двух игроков. Игра представляет собой модифицированные шахматы, на доске 8×8 клеток расположены по 8 коней у каждого игрока на черных клетках на первых двух рядах доски со стороны каждого игрока. Игроки ходят по очереди, перемещая одну из фигур на 2 клетки в одном направлении и 1 клетку в другом направлении, ортогональным первому. Когда конь перемещается на клетку, занятую конём другого цвета, тот конь уничтожается. Игрок выигрывает, если уничтожает половину фигур другого игрока.
30	Сетевая игра «Дуэль магов» для двух игроков. Каждый игрок изначально имеет 20 очков здоровья и 10 очков магии. Игроки ходят по очереди. В начале хода игрок получает несколько случайно сгенерированных действий, некоторым образом изменяющих очки здоровья и магии игроков. Игрок, очки здоровья которого становятся неположительными, проигрывает.
31	Сетевая игра «Быстрый счёт» для двух игроков. Сервер генерирует некоторое арифметическое выражение, которое показывается игрокам. Игроки должны вычислить значение выражения и ввести ответ. Игрок проигрывает,

	если ввёл неправильный ответ, либо если его противник ввёл правильный ответ раньше него.
32	Сетевая игра Гранди для двух игроков. Изначально имеется кучка из 10 камней. Игроки ходят по очереди. За свой ход игрок может разделить одну из кучек на две непустые кучки разных размеров. Проигрывает игрок, который не может сделать ход.
33	Игра проблема двух армий. Игроки должны договориться о времени нападения, отправляя гонцов с временем через вражескую территорию. С определённой вероятностью гонца могут перехватить.
34	Игра “Осцилляторы”. Клиентское приложение игрока отправляет сообщения случайному игроку с определённой частотой. Игрок может увеличить или уменьшить частоту на определённый шаг. Игроки выигрывают, когда все осцилляторы синхронизируются.
35	Многопользовательский эхо сервис. Пользователи записывают звук в клиентском приложении и слышат с задержкой звук сразу от всех участников.
36	Игра “Столовая философов”. Существует место в горах, куда съезжаются философы со всего мира поразмышлять. Для размышлений нужна энергия, поэтому философы охотно ходят в столовую с одним круглым столом. Каждый философ берёт с собой одну вилку и кладёт её справа. И каждый раз в меню столовой спагетти, которую можно есть только двумя вилками. Игроки выступают в роли философов и могут взять две вилки в любом порядке. Задача игроков договориться об алгоритме, по которому они будут брать вилки и не умереть с голоду.
37	Игра “На автобусе или пешком?”. Два игрока живут недалеко от работы и могут дойти пешком до неё за одинаковое время. Но на автобусе можно добраться ещё

	быстрее! Проблема в том, что расписание автобусов меняется каждый день. Выигрывает игрок, который приходил первым большее количество раз.
38	Игра “Крокодил”. Ведущий игрок получает возможность рисовать на видимым всем холсте. В начале игры ему загадывают слово. Другие игроки предлагают свои варианты через чат, с которыми игрок может согласиться или не согласиться. Раунд заканчивается, когда ведущий подтверждает, что слово отгадано.
39	Сетевая игра «Поединок» для двух игроков. У каждого игрока есть 5 героев, каждый из которых обладает тремя случайно сгенерированными положительными целыми числами: силой, ловкостью и интеллектом. Каждый ход игроки одновременно в тайне друг от друга выбирают одного из своих героев, после чего выбранные герои сражаются. Побеждает тот герой, у которого по крайней мере две характеристики превосходят соответствующие характеристики героя противника. При этом проигравший герой уничтожается. Побеждает игрок, уничтоживший всех героев противника.
40	Сетевая игра «Ним» для двух игроков. Вначале случайно генерируется несколько целых положительных чисел. Игроки ходят по очереди. За свой ход игрок может вычесть из одного из положительных чисел некоторое не превосходящее его число. Проигрывает игрок, который не может сделать ход.
41	Многопользовательский текстовый редактор. После подключения к серверу пользователи имеют доступ к одному и тому же тексту. Пользователи могут прочитать текст и внести в него изменения. Изменения произведённые одним пользователем распространяются всем остальным.

42	Сетевая игра «Дилемма заключённого». Двум подключившимся к серверу игрокам предоставляется выбор: сотрудничать или предать. После получения ответов игрокам начисляются очки в соответствии с матрицей выигрышей [[20,1],[1,10]].

2.2 Теоретическая справка

Обычно программы работают как минимум с двумя представлениями данных:

1. Представлением данных в памяти в виде структур, объектов, списков, массивов и других структур данных.
2. Представлением данных подготовленных для хранения или передачи по сети.

В первом случае представления оптимизированы с точки зрения эффективного чтения и модификации их центральным процессором. Во втором случае данные упаковываются в автономную последовательность байт, с адресацией отличной от адресации виртуальной памяти. Поэтому возникает необходимость в определённом способе преобразования между этими двумя представлениями.

Преобразование структур в памяти в автономную последовательность байт называют кодированием, сериализацией или маршаллингом. Обратный процесс часто называют парсингом, десериализацией или анмаршиллингом.

Во многих языка есть встроенная поддержка такого кодирования. Вероятно, вы уже сохраняли в Java объекты с помощью java.io.Serializable или в Python с помощью pickle. Существуют и сторонние библиотеки, например для Java широко известна библиотека Kryo.

Эти библиотеки очень удобны, так как позволяют сохранять состояние объектов в памяти с минимальными усилиями. Однако они содержат в себе один или несколько недостатков:

- формат кодирования привязан к одному языку,
- возможность создавать любые объекты языка в памяти по умолчанию является проблемой с точки зрения безопасности,
- библиотеки не отслеживают прямую и обратную совместимость меняющихся со временем описаний объектов,
- некоторые встроенные реализации предназначены для отладки программы и не оптимизированы.

По этим соображениями использовать в продуктовом коде для сохранения состояния программы встроенные в язык библиотеки это плохая идея.

Движение в сторону стандартизации кодировок, нацеленных на межсистемный обмен данными, привело к созданию форматов XML и JSON. Исходя из целей верификации и эффективности, параллельно разрабатывались бинарные форматы со схемами. Одним из наиболее известных является Protocol Buffers. Целью последующего текста является краткое знакомство с этими тремя форматами.

XML

Аббревиатура XML расшифровывается как расширяемый язык разметки. Под разметкой понимается описание логической структуры документа. Если вы ранее видели HTML, вы удивитесь на сколько похожи это два формата:

```
<?xml version="1.0" encoding="UTF-8"?>
<recipe>
   <title>
      Бутерброд
   </title>
   <ingredients>
      <ingredient qty="1">
         кусочек хлеба
      </ingredient>
      <ingredient qty=”1”>
         кусочек колбасы
      </ingredient>
      <ingredient qty="1">
         кусочек масла
      </ingredient>
   </ingredients>
   <instructions>
      Размажьте масло равномерно по хлебу и положите сверху
колбасу.
   </instructions>
</recipe>
```

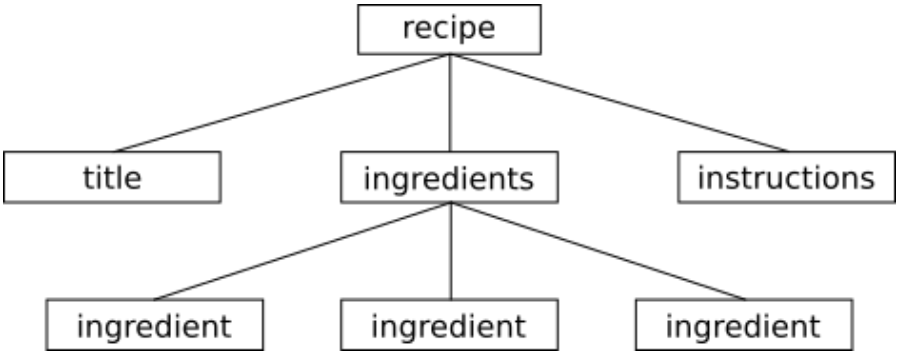

Рисунок 1. Структура рассматриваемого XML документа

Размер сообщения составляет 587 байт в кодировке UTF-8.

В XML документе в угловых скобках вы видите имена тегов (recipe, title, ingredients, ingredient), и их атрибуты (qty, name) со значениями. Теги, имеют открывающую и закрывающую часть, и могут быть вложенными друг в друга.

Главное отличие XML от HTML в том, что имена тегов и атрибутов XML проектируются разработчиком прикладного приложения, а имена тегов и атрибутов HTML проектируются Консорциумом Всемирной паутины (World Wide Web Consortium, W3C), разработчиками доминирующих на рынке браузеров и закрепляются в стандартах [link to w3c standard 1.0]. Указание стандарта как правило можно встретить в шапке XML. Наиболее широко поддерживаемым библиотеками стандартом является версия 1.0.

XML и HTML имеют общего родителя - SGML. HTML до версии 4 являлся приложением SGML (также как графический редактор является приложением языка программирования). А XML это упрощённый SGML (его подмножество).

В 1998 году XML вошёл в рекомендации W3C. С этого момента было создано множество форматов для описания документов, включая: RSS, Atom, SOAP, SVG, XHTML. Многие форматы офисных документов: Office Open XML (Microsoft Office), OpenDocument (OpenOffice, LibreOffice) были построены на его основе. Язык также стал широко применяться разработчиками как формат для обмена данными. Например, он был взят за основу для коммуникационного протокола XMPP.

Причиной активного использования формата в новых проектах и по сей день являются широкая доступность библиотек для разных языков программирования. Другим фактором можно считать и то, что XML является текстовым форматом, что позволяет анализировать и редактировать

данные не только с помощью библиотек, но и в любом текстовом редакторе.

Модели взаимодействия с XML документом: SAX, StAX и DOM.

Для того, чтобы ориентироваться в доступных способах чтения и записи в XML обратим внимание на два свойства. Во-первых, обработка документа может быть потоковой и блочной. Во-вторых, обработку элементов можно делегировать в callback-и (push стратегия) или проводить её в императивном стиле через абстракцию итератора (pull стратегия). В таблице 1 приводится классификация моделей взаимодействия с XML документом.

Таблица 1. Модели взаимодействия с XML документом

	поточная	блочная
push	SAX	-
pull	StAX	DOM

SAX

Ключевые слова: потоковая обработка, последовательный доступ, push-ориентированный парсер.

Примеры библиотек для различных языков программирования:

- C++: Expat (https://libexpat.github.io)
- Java: javax.xml.parsers.SAXParserFactory
- Scala: xs4s (https://github.com/ScalaWilliam/xs4s)

- Python: xml.sax.

SAX (Simple API for XML) — модель поточного парсера, которая рассматривает XML документ в виде последовательности событий, возникающих в начале и конце считывания infoset элементов (XML Information Set). Примерами таких событий являются начало/окончание документа, открытие и закрытие тега элемента, завершение считывания содержимого тега и другие.

События обработки документа делятся на классы и могут быть обработаны соответствующим обработчиком после его регистрации в парсере. Программисту доступны:

- ContentHandler
- ErrorHandler
- DTDHandler
- EntityResolver

Как правило перед парсингом устанавливают ContentHandler и ErrorHandler. В случае, если обработчик не установлен, все события в его сфере ответственности игнорируются.

Примечание. Все последующие примеры будут приводится на языке Scala с использованием Java API. Разница с использованием в Java будет заключаться в небольших синтаксических отличиях. Это позволит вам выполнить большую часть кода в онлайн компиляторе, например https://scastie.scala-lang.org/, по ходу чтения раздела. Также, с помощью REPL в IntelliJ Idea вы можете интерактивно взаимодействовать с примерами. Для этого скачайте IntelliJ IDEA Community Edition, при настройке установите scala плагин, создайте sbt проект и запустите Scala консоль в меню Tools.

Пример чтения документа:

```
val xml = """<?xml version="1.0" encoding="UTF-8"?>
<recipe>
   <title>
      Бутерброд
   </title>
   <ingredients>
      <ingredient qty="1">
         кусочек хлеба
      </ingredient>
      <ingredient qty="1">
         кусочек колбасы
      </ingredient>
      <ingredient qty="1">
         кусочек сливочного масла
      </ingredient>
   </ingredients>
   <instructions>
      Размажьте масло равномерно по хлебу и положите сверху
колбасу.
   </instructions>
</recipe>
"""

val spf = javax.xml.parsers.SAXParserFactory.newInstance();
spf.setNamespaceAware(true);
val sp = spf.newSAXParser();
val xmlr = sp.getXMLReader();

val handler = new org.xml.sax.helpers.DefaultHandler {
  override def startDocument(): Unit =
    print("Start document")

  override def startElement(x$1: String, x$2: String, x$3:
String, x$4: org.xml.sax.Attributes): Unit =
    println("Start element", x$1, x$2, x$3)

  override def endElement(x$1: String, x$2: String, x$3:
```

```
String): Unit =
    println("End element", x$1, x$2, x$3)

  override def characters(ch:Array[Char], start:Int,
length:Int):Unit =
    println(ch.toList, start, length)

  override def endDocument(): Unit =
    print("End document")
}
xmlr.setContentHandler(handler)

xmlr.parse(new org.xml.sax.InputSource(new
java.io.StringReader(xml)))
```

В приведённом коде создаётся новый SAX парсер, в котором регистрируется наследник класса DefaultHandler с переопределёнными методами. После запуска вы увидите для каких элементов XML срабатывают те или иные обработчики.

Обработка SAX парсером гораздо менее затратная по памяти по сравнению с блочной обработкой DOM парсером. Потоковая обработка потенциально не ограничивает размер обрабатываемого файла. Однако без доступа ко всему XML документу в памяти мы теряем возможность использовать XPath навигацию и преобразование документов с помощью языка XSLT.

StAX

Ключевые слова: потоковая обработка, последовательный доступ, pull-ориентированный парсер, создание XML.

Примеры библиотек для разных языков программирования:

- C++: Streaming XML parser in C++
- Java: javax.xml.stream
- Scala: XML SPaC
- Python: XMLPullParser

StAX (Streaming API for XML) — это API, для парсинга XML документа, являющийся чем-то средним между SAX и DOM. StAX API предоставляет методы для потокового разбора XML документов с pull стратегией. Pull стратегия в Java может быть реализована двумя способами: явным вызовом методов hasNext и next интерфейса XMLStreamReader или использованием интерфейса-итератора XMLEventReader в циклах. Итератор представляет XML документ в виде последовательности объектов XMLEvent. В отличие от SAX, StAX программный интерфейс позволяет не только считывать, но и записывать infoset элементы. В Java это делается также двумя способами: посредством XMLStreamWriter и XMLEventWriter.

Пример чтения документа на основе потока XML элементов:

```
import javax.xml.stream.XMLInputFactory
import javax.xml.stream.XMLStreamConstants

val xmlInputFactory = XMLInputFactory.newFactory()
val stringReader = new java.io.StringReader(xml)
val xmlStreamReader =
xmlInputFactory.createXMLStreamReader(stringReader)

while(xmlStreamReader.hasNext()) {
  xmlStreamReader.next() match {
    case XMLStreamConstants.START_DOCUMENT =>
      println("Start document")
    case XMLStreamConstants.END_DOCUMENT =>
      println("End document")
    case XMLStreamConstants.START_ELEMENT =>
```

```
      println("Start element", xmlStreamReader.getName())
    case XMLStreamConstants.END_ELEMENT =>
      println("End element", xmlStreamReader.getName())
    case XMLStreamConstants.CHARACTERS =>
      println("Contents",
xmlStreamReader.getTextCharacters().toList)
    case _ =>
      println("Other")
  }
}
```

Пример чтения документа на основе потока XML событий:

```
import javax.xml.stream.XMLInputFactory
import javax.xml.stream.XMLStreamConstants
import javax.xml.stream.events.{StartDocument, EndDocument,
StartElement, EndElement, Characters}

val xmlInputFactory = XMLInputFactory.newFactory()
val stringReader = new java.io.StringReader(xml)
val xmlEventReader =
xmlInputFactory.createXMLEventReader(stringReader)

while(xmlEventReader.hasNext()) {
  xmlEventReader.nextEvent() match {
    case e:StartDocument =>
      println("Start document")
    case e:EndDocument =>
      println("End document")
    case e:StartElement =>
      println("Start element", e.getName())
    case e:EndElement =>
      println("End element", e.getName())
    case e:Characters =>
      println("Contents", e.getData())
    case _ =>
      println("Other")
  }
```

```
}
```

Пример записи документа на основе потока XML элементов:

```
import javax.xml.stream.XMLOutputFactory

val xmlOutputFactory = XMLOutputFactory.newFactory()
val stringWriter = new java.io.StringWriter()
val xsw =
xmlOutputFactory.createXMLStreamWriter(stringWriter)

xsw.writeStartDocument()
xsw.writeStartElement("recipe")
xsw.writeStartElement("title")
xsw.writeCharacters("Бутерброд")
xsw.writeEndElement()
xsw.writeStartElement("ingredients")
xsw.writeStartElement("ingredient")
xsw.writeAttribute("qty", "1")
xsw.writeCharacters("кусочек хлеба")
xsw.writeEndElement()
xsw.writeStartElement("ingredient")
xsw.writeAttribute("name", "Докторская")
xsw.writeAttribute("qty", "1")
xsw.writeCharacters("кусочек колбасы")
xsw.writeEndElement()
xsw.writeStartElement("ingredient")
xsw.writeAttribute("qty", "1")
xsw.writeCharacters("кусочек сливочного масла")
xsw.writeEndElement()
xsw.writeEndElement()
xsw.writeStartElement("instructions")
xsw.writeCharacters("Размажьте масло равномерно по хлебу и
положите сверху колбасу.")
xsw.writeEndElement()
xsw.writeEndElement()
xsw.writeEndDocument()
xsw.flush()
```

```
xsw.close()

val xml = stringWriter.toString
print(xml)
```

Пример записи документа на основе потока XML событий:

```
import javax.xml.stream.XMLOutputFactory
import javax.xml.stream.XMLEventFactory
import javax.xml.stream.events.{StartDocument, EndDocument,
StartElement, EndElement, Characters}

val xmlOutputFactory = XMLOutputFactory.newFactory()
val stringWriter = new java.io.StringWriter()
val xew =
xmlOutputFactory.createXMLEventWriter(stringWriter)
val xef = XMLEventFactory.newFactory()
xew.add(xef.createStartDocument())
xew.add(xef.createStartElement("", "", "recipe"))
xew.add(xef.createStartElement("", "", "title"))
xew.add(xef.createCharacters("Бутерброд"))
xew.add(xef.createEndElement("", "", ""))
xew.add(xef.createStartElement("", "", "ingredients"))
xew.add(xef.createStartElement("", "", "ingredient"))
xew.add(xef.createAttribute("qty", "1"))
xew.add(xef.createCharacters("кусочек хлеба"))
xew.add(xef.createEndElement("", "", ""))
xew.add(xef.createStartElement("", "", "ingredient"))
xew.add(xef.createAttribute("name", "Докторская"))
xew.add(xef.createAttribute("qty", "1"))
xew.add(xef.createCharacters("кусочек колбасы"))
xew.add(xef.createEndElement("", "", ""))
xew.add(xef.createStartElement("", "", "ingredient"))
xew.add(xef.createAttribute("qty", "1"))
xew.add(xef.createCharacters("кусочек сливочного масла"))
xew.add(xef.createEndElement("", "", ""))
xew.add(xef.createEndElement("", "", ""))
xew.add(xef.createStartElement("", "", "instructions"))
```

```
xew.add(xef.createCharacters("Размажьте масло равномерно по
хлебу и положите сверху колбасу."))
xew.add(xef.createEndElement("", "", ""))
xew.add(xef.createEndElement("", "", ""))
xew.add(xef.createEndDocument())
xew.flush()
xew.close()

val xml = stringWriter.toString
print(xml)
```

Как XMLStreamReader, так и XMLEventReader позволяют приложению обходить содержимое XML независимо друг от друга. Разница заключается в том, как эти два подхода отдают части XML данных. XMLStreamReader ведёт себя как курсор, находящийся сразу за последним считанным XML токеном и предоставляет методы для получения более подробной информации о нём. XMLEventReader немного более интуитивный потому, что преобразовывает XML в последовательность объектов XMLEvent, способ обработки которой стандартен.

Подводя итог, можно сказать, что преимущества и недостатки StAX определяются потоковой природой обработки, и во многом аналогичны SAX. Но в отличие от SAX наличие явной абстракции потока позволяет сделать код более понятным и естественным образом дополнить API методами записи XML элементов в поток.

DOM

Ключевые слова: блочная обработка, случайный доступ, pull-ориентированный парсер, создание XML.

Примеры библиотек для разных языков программирования:

- C++: tinyxml2 (https://github.com/leethomason/tinyxml2)
- Java: java.xml.parsers.DocumentBuilderFactory
- Scala: scala-xml
- Python: xml.dom

Document Object Model (DOM) — программный интерфейс для преобразования XML документа в дерево узлов и обратно. После построения дерева приложение использует DOM API для навигации по нему, а также для извлечения из узлов infoset элементов. Важными отличиями DOM от SAX являются: необходимость загрузки всей информации XML в память, возможность произвольного доступа к элементам и наличие операций для изменения документов.

DOM рассматривает XML документ как дерево, состоящее из узлов различных типов: элемент (element), атрибут (attribute), текст (text), сущность (entity), ссылка на сущность (entity reference), документ (document), фрагмент документа (document fragment), тип документа (document type), секция CDATA, комментарий (comment), инструкция обработки (processing instruction).

Пример чтения документа DOM парсером:

```
import javax.xml.parsers.DocumentBuilderFactory
import org.w3c.dom.{Document, Node}

val dbf = DocumentBuilderFactory.newInstance()
val db = dbf.newDocumentBuilder()

val doc = db.parse(new org.xml.sax.InputSource(new
java.io.StringReader(xml)))
```

```
object DOMTraverser {

 def traverse(node:Node) :Unit = {
   if (node.hasChildNodes) {
     val childNodes = node.getChildNodes()
     for (i <- 0 to childNodes.getLength-1) {
       traverse__(childNodes.item(i))
     }
   } else {
     traverse__(node)
   }
 }

 def traverse__(node:Node) :Unit = {
   node.getNodeType match {
     case Node.ELEMENT_NODE =>
       println(s"Element node: ${node.getNodeName()}")
       val attrs = node.getAttributes()
       if (attrs != null) {
         for (i <- 0 to attrs.getLength-1) {
           println(s"${attrs.item(i)}")
         }
       }
       traverse(node)
     case _ =>
       println(s"Element contents: ${node.getNodeValue}")
   }
 }
}

if (doc.hasChildNodes) {
 val childNodes = doc.getChildNodes()
 for (i <- 0 to childNodes.getLength-1) {
   DOMTraverser.traverse(childNodes.item(i))
 }
}
```

В данном коде мы воспользовались классом DocumentBuilderFactory для создания DocumentBuilder. С

помощью экземпляра DocumentBuilder производится построение DOM дерева из строки xml, содержащей XML. Затем следует пример простейшего обхода этого дерева, включающий функции traverseDOM и traverseDOM__.

Пример записи документа DOM парсером

```
import javax.xml.parsers.DocumentBuilderFactory
import org.w3c.dom.{Document, Node}
import javax.xml.transform.TransformerFactory
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult

val dbf = DocumentBuilderFactory.newInstance()
val db = dbf.newDocumentBuilder()

val doc = db.newDocument()

val root = doc.createElement("recipe")
val title = doc.createElement("title")
val ingredients = doc.createElement("ingredients")
val instructions = doc.createElement("instructions")

val ingredient1 = doc.createElement("ingredient")
val ingredient2 = doc.createElement("ingredient")
val ingredient3 = doc.createElement("ingredient")

val titleContents = doc.createTextNode("Бутерброд")
val instructionsContents = doc.createTextNode("Размажьте
масло равномерно по хлебу и положите сверху колбасу.")

val ingredient1Contents = doc.createTextNode("кусочек
хлеба")
val ingredient2Contents = doc.createTextNode("кусочек
колбасы")
val ingredient3Contents = doc.createTextNode("кусочек
масла")
```

```
val ingredient1Attribute1 = doc.createAttribute("qty")
ingredient1Attribute1.setValue("1")

ingredient1.setAttribute("qty", "1")
ingredient2.setAttribute("qty", "1")
ingredient2.setAttribute("name", "Докторская")
ingredient3.setAttribute("qty", "1")

doc.appendChild(root)
root.appendChild(title)
root.appendChild(ingredients)
root.appendChild(instructions)
title.appendChild(titleContents)
ingredients.appendChild(ingredient1)
ingredients.appendChild(ingredient2)
ingredients.appendChild(ingredient3)
instructions.appendChild(instructionsContents)
ingredient1.appendChild(ingredient1Contents)
ingredient2.appendChild(ingredient2Contents)
ingredient3.appendChild(ingredient3Contents)

val tf = TransformerFactory.newInstance()
val transformer = tf.newTransformer()

val source = new DOMSource(doc)
val result = new StreamResult(System.out)

transformer.transform(source, result)
```

JSON

JSON (JavaScript Object Notation) — текстовый формат обмена данными, являющийся подмножеством языка JavaScript. Формат JSON представляет JSON объекты в виде неупорядоченного множества пар ключ:значение, разделённых

запятыми и заключённых в фигурные скобки. Формат описывается рекурсивно - в качестве значения могут выступать JSON объекты. Кроме JSON объектов значениями могут быть:

- одномерный массив — упорядоченное множество значений. Массив заключается в квадратные скобки *«[ ]»*. Значения разделяются запятыми. Массив может быть пустым, т.е. не содержать ни одного значения.
- число (целое или вещественное).
- литералы *true* (логическое значение «истина»), *false* (логическое значение «ложь») и *null*.
- строка — это упорядоченное множество из нуля или более символов юникода, заключённое в двойные кавычки. Символы могут быть указаны с использованием escape-последовательностей, начинающихся с обратной косой черты *«\»* (поддерживаются варианты \', \", \\, \/, \t, \n, \r, \f и \b), или записаны шестнадцатеричным кодом в кодировке Unicode в виде \uFFFF.

```
   {
      "title": "Бутерброд",
      "instructions": "Размажьте масло равномерно по хлебу и
положите сверху колбасу.",
      "ingredients": [
          {
          "name": “Кусочек хлеба”,
          "qty": 1
          },
          {
          "name": “Кусочек колбасы”,
          "qty": 2
          },
          {
          "name": “Кусочек масла”,
          "qty": 2
```

```
        }
    ]
}
```

Размер сообщения составляет 497 байт.

В приведённом выше примере адрес является JSON объектом.

Для JVM существует не меньшее количество библиотек для JSON, чем для XML, но в отличие от XML нет стандартной библиотеки. Приложения при обработке JSON опираются на сторонние библиотеки. Рассмотрим примеры сериализации/десериализации объектов в JSON с помощью библиотеки GSON.

Примечание. Для установки библиотеки в SBT проект, найдите её в https://mvnrepository.com/ и добавьте строку зависимости в build.sbt. После чего обновите проект и перезапустите Scala консоль.

Пример записи GSON библиотекой:

```
import com.google.gson.GsonBuilder

case class Cat(name:String, age:Int, color:java.awt.Color)
val cat = Cat(name="Василий", age=4,
color=java.awt.Color.BLACK)

val builder = new GsonBuilder()
val gson = builder.create()
val json = gson.toJson(cat)
```

Пример чтения GSON библиотекой:

```
import com.google.gson.GsonBuilder

val jsonText =
"""{"name":"Василий","age":4,"color":{"value":-
16777216,"falpha":0.0}}"""

val builder = new GsonBuilder()
val gson = builder.create()
val cat = gson.fromJson(jsonText, classOf[Cat])
println(s"Кот. Имя: ${cat.name}, Возраст: ${cat.age}, Цвет:
${cat.color}")
```

GSON поддерживает сериализацию примитивных типов и POJO классов. Если на практике требуется сериализовать неподдерживаемый тип, его сериализатор/десериализатор можно зарегистрировать при старте приложения.

```
import com.google.gson.GsonBuilder
import org.joda.time.Instant
import java.lang.reflect.Type
import com.google.gson.{JsonSerializer, JsonDeserializer,
JsonSerializationContext, JsonDeserializationContext,
JsonElement, JsonPrimitive, JsonParseException}

object InstantTypeConverter extends JsonSerializer[Instant]
with JsonDeserializer[Instant] {
   override def serialize(src: Instant, typeOfSrc: Type,
context: JsonSerializationContext): JsonElement = new
JsonPrimitive(src.getMillis())
   override def deserialize(json: JsonElement, typeOfT:
Type, context: JsonDeserializationContext): Instant = new
Instant(json.getAsLong())
 }
```

```
val builder = new GsonBuilder()
builder.registerTypeAdapter(classOf[Instant],
InstantTypeConverter)
val gson = builder.create()

println(gson.toJson(new Instant()))
println(gson.fromJson("""1592996747765""",
classOf[Instant]))
```

Более полное руководство доступно на странице библиотеки https://github.com/google/gson.

Альтернативные JSON библиотеки для Java:

Jackson

Jackson — это набор утилит для обработки данных на Java. Утилиты включают в себя библиотеку для потокового парсинга и потоковой генерации JSON, библиотеку связывания отображения классов на JSON. Работа с API потокового чтения и записи идёт в том же ключе, что в StAX, а связывание позволяет построить из JSON дерево объектов, аналогичное DOM. Дополнительные аннотации дают доступ к тонкой настройке связывания.

JSON-P

Несмотря на то, что стандартной библиотеки JSON из JDK для JVM нет, на её место есть кандидат. JSON-P (JSON Processing) API изначально планировалось включить в Java SE, однако затем её добавили только в Java EE.

JSON-P позволяет обрабатывать JSON файлы как потоково

(аналог StAX), так и блочно (аналог построения DOM).

JSON-P 1.0 состоит из 25 типов, расположенных в пакете javax.json, javax.json.spi и javax.json.stream. Пакет javax.json в основном содержит типы, являющиеся частью объектной модели. Потоковая часть API находится в javax.json.stream. В javax.json.spi определяется один класс --- сервис для обработки JSON.

Примеры библиотек для разных языков программирования:

- C++: JSON for Modern C++ (https://github.com/nlohmann/json)
- Scala: lift-json,
- Python: встроенный модуль json

JSON часто используется для асинхронного обмена сообщениями между браузером и сервером посредством AJAX https://en.wikipedia.org/wiki/Ajax_(programming) или WebSocket (https://ru.wikipedia.org/wiki/WebSocket). Формат также используется в некоторых NoSQL базах данных (MongoDB, CouchDB), в приложениях Twitter, Facebook, LinkedIn, Flickr, Google Maps. Относительно недавно в PostgreSQL появилась поддержка JSON в SQL запросах в виде функций и операторов (https://www.postgresql.org/docs/current/functions-json.html).

Данному формату посвящён сайт https://www.json.org, на котором можно узнать дополнительную информацию.

Protocol Buffers

Сообщение Protocol Buffer представляет из себя последовательность пар ключ:значение.

В бинарной версии сообщения в качестве ключа используется идентификатор поля, состоящий из номера поля и типа. Имя поля может быть получено только при наличии схемы, описывающей сообщение (.proto файл). Бинарный формат использует следующие типы полей:

Таблица 1. Типы полей бинарного представления Protocol Buffers сообщения

Код типа	Тип	Используется для следующих типов Protocol Buffers языка
0	Varint	int32, int64, uint32, uint64, sint32, sint64, bool, enum
1	64 бита	fixed64, sfixed64, double
2	последовательность байт фиксированной длины	string, bytes, embedded messages, packed repeated fields

3	Начало группы	группы запрещены в новых версиях
4	Окончание группы	группы запрещены в новых версиях
5	32 бита	fixed32, sfixed32, float

Наиболее необычный тип Varint, является важным для понимания бинарного представления. Varint это способ кодирования целого числа одним или несколькими байтами. Чем меньше число, тем меньше оно занимает места. Первый бит каждого октета является сигнальным флагом о том, что следующий октет в битовом потоке - часть текущего.

Рассмотрим два числа 1 и 300. Первое число умещается в байт, а второе потребует как минимум два байта.

```
1 - 0000 0001
300 - 1010 1100 0000 0010
```

Чтобы декодировать число 300, нужно откинуть первые биты октетов, а затем склеить полученные септеты в обратном порядке.

Ключ поля Protocol Buffer сообщения хранится в следующем формате. Последние 3 бита числа хранят один из кодов типа, указанных в таблице 3. Остальные являются целочисленным значением ключа, то есть номером тэга. Конкатенация значения ключа и кода типа кодируется с помощью varint.

Для того, чтобы начать использовать Protocol Buffer в первую очередь описывается структура данных, которую планируется

сохранять или передавать по сети. По этому описанию компилятор Protocol Buffer генерирует код целевого языка, состоящий из программного представления и функций сериализации/десериализации в бинарное представление.

В редакторе шестнадцатиричноего кода (https://hexed.it/) рассмотрим как кодируется пример с рецептом бутерброда, использованный ранее в тексте.

```
syntax = "proto2";

message Ingredient {
   required string name = 1;
   optional int32 qty = 2;
}

message Recipe {
   required string title = 1;
   required string instructions = 2;
   repeated Ingredient ingredients = 3;
}
```

Каждое поле может иметь один из следующих модификаторов:

- **required**: обязательное поле. Попытка создать или прочитать сообщение без указания значения этого поля приведёт к ошибке времени исполнения.
- **optional**: значение поля может быть не указано. В этом случае используется значение по-умолчанию из .proto файла или системное.
- **repeated**: поле может повторяться 0 или несколько раз (аналог квантификатора * в регулярных выражениях). Используется для хранения массивов.

Генерация программного представления

Теперь, когда у нас есть схема .proto, следующим шагом генерируется программное представление и функции сериализации/десериализации Protocol Buffer компилятором protoc, доступным на сайте https://github.com/protocolbuffers/protobuf/releases. В данном случае генерируются Java классы:

```
protoc -I=$SRC_DIR --java_out=$DST_DIR
$SRC_DIR/message.proto
```

Переменная среды SRC_DIR хранит путь к каталогу с .proto файлом, а DST_DIR путь, в котором появятся сгенерированные файлы. В результате выполнения команды сгенерируется файл Message.java.

Закодированное сообщение с помощью Protocol Buffers представленное в шестнадцатеричной системе счисления:

```
0A 12 D0 91 D1 83 D1 82 D0 B5 D1 80 D0 B1 D1 80 D0
BE D0 B4 12 73 D0 A0 D0 B0 D0 B7 D0 BC D0 B0 D0 B6
D1 8C D1 82 D0 B5 20 D0 BC D0 B0 D1 81 D0 BB D0 BE
20 D1 80 D0 B0 D0 B2 D0 BD D0 BE D0 BC D0 B5 D1 80
D0 BD D0 BE 20 D0 BF D0 BE 20 D1 85 D0 BB D0 B5 D0
B1 D1 83 20 D0 B8 20 D0 BF D0 BE D0 BB D0 BE D0 B6
D0 B8 D1 82 D0 B5 20 D1 81 D0 B2 D0 B5 D1 80 D1 85
D1 83 20 D0 BA D0 BE D0 BB D0 B1 D0 B0 D1 81 D1 83
2E 1A 1D 0A 19 D0 BA D1 83 D1 81 D0 BE D1 87 D0 B5
D0 BA 20 D1 85 D0 BB D0 B5 D0 B1 D0 B0 10 01 1A 21
0A 1D D0 BA D1 83 D1 81 D0 BE D1 87 D0 B5 D0 BA 20
D0 BA D0 BE D0 BB D0 B1 D0 B0 D1 81 D1 8B 10 01 1A
1D 0A 19 D0 BA D1 83 D1 81 D0 BE D1 87 D0 B5 D0 BA
20 D0 BC D0 B0 D1 81 D0 BB D0 B0 10 01
```

Размер сообщения составляет 234 байта.

Таблица 2. Расшифровка закодированного Protocol Buffer сообщения

HEX значение	**Расшифровка**
title	
0A	В двоичной системе: 0 0001 010 Тэг: 1 Тип: последовательность байт фиксированной длины
12	В десятичной системе: 18
D0 91 D1 83 D1 82 D0 B5 D1 80 D0 B1 D1 80 D0 BE D0 B4	“Бутерброд” В закодированном виде занимает 18 байт
instructions	
12	В двоичной системе: 0 0001 010 Тэг: 2 Тип: последовательность байт фиксированной длины

73	В десятичной системе: 115
D0 A0 D0 B0 D0 B7 D0 BC D0 B0 D0 B6 D1 8C D1 82 D0 B5 20 D0 BC D0 B0 D1 81 D0 BB D0 BE 20 D1 80 D0 B0 D0 B2 D0 BD D0 BE D0 BC D0 B5 D1 80 D0 BD D0 BE 20 D0 BF D0 BE 20 D1 85 D0 BB D0 B5 D0 B1 D1 83 20 D0 B8 20 D0 BF D0 BE D0 BB D0 BE D0 B6 D0 B8 D1 82 D0 B5 20 D1 81 D0 B2 D0 B5 D1 80 D1 85 D1 83 20 D0 BA D0 BE D0 BB D0 B1 D0 B0 D1 81 D1 83 2E	"Размажьте масло равномерно по хлебу и положите сверху колбасу." В закодированном виде занимает 115 байт
ingredients/ingredient	
1A	В двоичной системе: 0 0001 010 Тэг: 3 Тип: последовательность байт фиксированной длины

1D	В десятичной системе: 29
ingredients/ingredient/name	
0A	В двоичной системе: 0 0001 010 Тэг: 1 Тип: последовательность байт фиксированной длины
19	В десятичной системе: 25
D0 BA D1 83 D1 81 D0 BE D1 87 D0 B5 D0 BA 20 D1 85 D0 BB D0 B5 D0 B1 D0 B0	"кусочек хлеба" В закодированном виде занимает 25 байт
ingredients/ingredient/qty	
10	В двоичной системе: 0 0010 000 Тэг: 2 Тип: varint число
01	1
ingredients/ingredient	
1A	В двоичной системе: 0 0001 010

	Тэг: 1 Тип: последовательность байт фиксированной длины
21	В десятичной системе: 33
ingredients/ingredient/name	
0A	В двоичной системе: 00001 010 Тэг: 1 Тип: последовательность байт фиксированной длины
1D	В десятичной системе: 29
D0 BA D1 83 D1 81 D0 BE D1 87 D0 B5 D0 BA 20 D0 BA D0 BE D0 BB D0 B1 D0 B0 D1 81 D1 8B	"кусочек колбасы" В закодированном виде занимает 29 байт
ingredients/ingredient/qty	
10	В двоичной системе: 00010 000 Тэг: 2 Тип: varint число
01	1
ingredients/ingredient	

1A	В двоичной системе: 0 0001 010 Тэг: 1 Тип: последовательность байт фиксированной длины
1D	В десятичной системе: 29
ingredients/ingredient/name	
0A	В двоичной системе: 0 0001 010 Тэг: 1 Тип: последовательность байт фиксированной длины
19	В десятичной системе: 25
D0 BA D1 83 D1 81 D0 BE D1 87 D0 B5 D0 BA 20 D0 BC D0 B0 D1 81 D0 BB D0 B0 10 01	"кусочек масла" В закодированном виде занимает 25 байт
ingredients/ingredient/qty	
10	В двоичной системе: 0 0010 000 Тэг: 2 Тип: varint число

01	1

Пример записи сообщения

```
val recipeBuilder = Message.Recipe.newBuilder()
recipeBuilder.setTitle("Бутерброд")
recipeBuilder.setInstructions("Размажьте масло равномерно по
хлебу и положите сверху колбасу.")
recipeBuilder.addIngredientsBuilder().setName("кусочек
хлеба").setQty(1)
recipeBuilder.addIngredientsBuilder().setName("кусочек
колбасы").setQty(1)
recipeBuilder.addIngredientsBuilder().setName("кусочек
масла").setQty(1)
recipeBuilder.build().writeTo(new
java.io.FileOutputStream("recipe"))
```

Пример чтения сообщения

```
val recipe = Message.Recipe.parseFrom(new
java.io.FileInputStream("recipe"))
```

Protocol Buffers не разрабатывались для обработки больших сообщений. Негласное правило говорит, что если вы передаёте сообщения больше мегабайта, то имеет смысл выбрать другой формат.

Познакомившись с XML, JSON и Protocol Buffer отметим, что существуют и другие форматы, которые могут быть более подходящими для конкретной задачи. Для эффективного хранения табличных данных и многопоточного выполнения

аналитических запросов над ними лучше подходит колоночный формат parquet. Формат данных hdf5 позволяет хранить аннотированные многомерные массивы. Формат Avro подходит для хранения структурированных данных в распределённых файловых системах, наподобие HDFS.

2.3 Контрольные вопросы

1. Что такое XML/JSON/ProtocolBuffers?
2. В чём заключаются преимущества и недостатки хранения данных в виде документов XML/JSON?
3. Каким образом состояние программы может храниться в виде документа XML/JSON?
4. Какая технология позволяет привязывать типы данных в исходном коде программы к документам XML?
5. В чём заключаются преимущества и недостатки хранения состояния программы в виде документа XML/JSON?
6. Что представляет собой формат JSON и на чём он основан?
7. Как преобразовывать данные в формат JSON в компьютерной программе и для чего это может быть нужно?
8. Каким образом программы могут обмениваться информацией по сети?
9. Какие технологии могут использоваться в компьютерных программах для передачи данных по сети?

2.4 Пример выполнения лабораторной работы

Ниже приведён пример исходного кода на языке Python выполненного нулевого варианта данной лабораторной работы. Файл `server.py` содержит скрипт для серверной части приложения, который обеспечивает рассылку отправленных сообщений всем клиентам, таким образом любое количество пользователей может общаться в такой системе обмена мгновенными сообщениями. При запуске из консоли серверный скрипт принимает единственный параметр – номер порта для прослушивания. Остальные файлы содержат исходный код клиентского приложения. Модуль `application` представляет содержит основную логику работы клиентского приложения, включая приём и отправку сообщений. Модуль main представляет собой точку входа в клиентское приложение. В модуль messages вынесены строковые элементы графического интерфейса пользователя. Модуль model содержит объектную модель сообщений, которыми обмениваются клиенты и сервер. Этот модуль подключается как к клиентскому, так и к серверному приложению, так что такой файл входит одновременно в оба приложения. Модуль `view` определяет графический интерфейс пользователя клиентского приложения.

Листинг 1. Содержимое файла `server.py`

```
# -*- coding: utf-8 -*-
import json
import socket
import sys
import threading
import model


BUFFER_SIZE = 2 ** 10
```

```
CLOSING = "Application closing..."
CONNECTION_ABORTED = "Connection aborted"
CONNECTED_PATTERN = "Client connected: {}:{}"
ERROR_ARGUMENTS = "Provide port number as the first
command line argument"
ERROR_OCCURRED = "Error Occurred"
EXIT = "exit"
JOIN_PATTERN = "{username} has joined"
RUNNING = "Server is running..."
SERVER = "SERVER"
SHUTDOWN_MESSAGE = "shutdown"
TYPE_EXIT = "Type 'exit' to exit>"


class Server(object):

    def __init__(self, argv):
        self.clients = set()
        self.listen_thread = None
        self.port = None
        self.sock = None
        self.parse_args(argv)

    def listen(self):
        self.sock.listen(1)
        while True:
            try:
                client, address = self.sock.accept()
            except OSError:
                print(CONNECTION_ABORTED)
                return
            print(CONNECTED_PATTERN.format(*address))
            self.clients.add(client)
            threading.Thread(target=self.handle,
args=(client,)).start()

    def handle(self, client):
        while True:
            try:
                message =
model.Message(**json.loads(self.receive(client)))
```

```
            except (ConnectionAbortedError,
ConnectionResetError):
                print(CONNECTION_ABORTED)
                return
            if message.quit:
                client.close()
                self.clients.remove(client)
                return
            print(str(message))
            if SHUTDOWN_MESSAGE.lower() ==
message.message.lower():
                self.exit()
                return
            self.broadcast(message)

    def broadcast(self, message):
        for client in self.clients:
            client.sendall(message.marshal())

    def receive(self, client):
        buffer = ""
        while not buffer.endswith(model.END_CHARACTER):
            buffer +=
client.recv(BUFFER_SIZE).decode(model.TARGET_ENCODING)
        return buffer[:-1]

    def run(self):
        print(RUNNING)
        self.sock = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
        self.sock.bind(("", self.port))
        self.listen_thread =
threading.Thread(target=self.listen)
        self.listen_thread.start()

    def parse_args(self, argv):
        if len(argv) != 2:
            raise RuntimeError(ERROR_ARGUMENTS)
        try:
            self.port = int(argv[1])
        except ValueError:
```

```
            raise RuntimeError(ERROR_ARGUMENTS)

    def exit(self):
        self.sock.close()
        for client in self.clients:
            client.close()
        print(CLOSING)


if __name__ == "__main__":
    try:
        Server(sys.argv).run()
    except RuntimeError as error:
        print(ERROR_OCCURRED)
        print(str(error))
```

Листинг 2. Содержимое файла application.py

```
# -*- coding: utf-8 -*-
import json
import socket
import threading
import messages
import model
import view


BUFFER_SIZE = 2 ** 10


class Application(object):

    instance = None

    def __init__(self, args):
        self.args = args
        self.closing = False
        self.host = None
        self.port = None
        self.receive_worker = None
        self.sock = None
        self.username = None
```

```
        self.ui = view.EzChatUI(self)
        Application.instance = self

    def execute(self):
        if not self.ui.show():
            return
        self.sock = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
        try:
            self.sock.connect((self.host, self.port))
        except (socket.error, OverflowError):
            self.ui.alert(messages.ERROR,
messages.CONNECTION_ERROR)
            return
        self.receive_worker =
threading.Thread(target=self.receive)
        self.receive_worker.start()
        self.ui.loop()

    def receive(self):
        while True:
            try:
                message =
model.Message(**json.loads(self.receive_all()))
            except (ConnectionAbortedError,
ConnectionResetError):
                if not self.closing:
                    self.ui.alert(messages.ERROR,
messages.CONNECTION_ERROR)
                return
            self.ui.show_message(message)

    def receive_all(self):
        buffer = ""
        while not buffer.endswith(model.END_CHARACTER):
            buffer +=
self.sock.recv(BUFFER_SIZE).decode(model.TARGET_ENCODIN
G)
        return buffer[:-1]

    def send(self, event=None):
```

```
        message = self.ui.message.get()
        if len(message) == 0:
            return
        self.ui.message.set("")
        message = model.Message(username=self.username,
message=message, quit=False)
        try:
            self.sock.sendall(message.marshal())
        except (ConnectionAbortedError,
ConnectionResetError):
            if not self.closing:
                self.ui.alert(messages.ERROR,
messages.CONNECTION_ERROR)

    def exit(self):
        self.closing = True
        try:

self.sock.sendall(model.Message(username=self.username,
message="", quit=True).marshal())
        except (ConnectionResetError,
ConnectionAbortedError, OSError):
            print(messages.CONNECTION_ERROR)
        finally:
            self.sock.close()
```

Листинг 3. Содержимое файла `main.py`

```
# -*- coding: utf-8 -*-
import sys
import application


def main(args):
    app = application.Application(args)
    app.execute()


if __name__ == "__main__":
    main(sys.argv)
```

Листинг 4. Содержимое файла `messages.py`

```
# -*- coding: utf-8 -*-

CONNECTION_ERROR = "Could not connect to server"
ERROR = "Error"
INPUT_SERVER_HOST = "Input Server Host"
INPUT_SERVER_PORT = "Input Server Port"
INPUT_USERNAME = "Input your username"
SEND = "Send"
SERVER_HOST = "Server Host"
SERVER_PORT = "Server Port"
TITLE = "ezChat"
USERNAME = "Username"
```

Листинг 5. Содержимое файла `model.py`

```
# -*- coding: utf-8 -*-

import json

END_CHARACTER = "\0"
MESSAGE_PATTERN = "{username}>{message}"
TARGET_ENCODING = "utf-8"


class Message(object):

    def __init__(self, **kwargs):
        self.username = None
        self.message = None
        self.quit = False
        self.__dict__.update(kwargs)

    def __str__(self):
        return MESSAGE_PATTERN.format(**self.__dict__)

    def marshal(self):
        return (json.dumps(self.__dict__) +
END_CHARACTER).encode(TARGET_ENCODING)
```

Листинг 6. Содержимое файла `view.py`

```
# -*- coding: utf-8 -*-
import tkinter
```

```
import messages

from tkinter import messagebox, simpledialog


CLOSING_PROTOCOL = "WM_DELETE_WINDOW"
END_OF_LINE = "\n"
KEY_RETURN = "<Return>"
TEXT_STATE_DISABLED = "disabled"
TEXT_STATE_NORMAL = "normal"


class EzChatUI(object):

    def __init__(self, application):
        self.application = application
        self.gui = None
        self.frame = None
        self.input_field = None
        self.message = None
        self.message_list = None
        self.scrollbar = None
        self.send_button = None

    def show(self):
        self.gui = tkinter.Tk()
        self.gui.title(messages.TITLE)
        self.fill_frame()
        self.gui.protocol(CLOSING_PROTOCOL,
self.on_closing)
        return self.input_dialogs()

    def loop(self):
        self.gui.mainloop()

    def fill_frame(self):
        self.frame = tkinter.Frame(self.gui)
        self.scrollbar = tkinter.Scrollbar(self.frame)
        self.message_list = tkinter.Text(self.frame,
state=TEXT_STATE_DISABLED)
        self.scrollbar.pack(side=tkinter.RIGHT,
```

```
fill=tkinter.Y)
        self.message_list.pack(side=tkinter.LEFT,
fill=tkinter.BOTH)
        self.message = tkinter.StringVar()
        self.frame.pack()
        self.input_field = tkinter.Entry(self.gui,
textvariable=self.message)
        self.input_field.pack()
        self.input_field.bind(KEY_RETURN,
self.application.send)
        self.send_button = tkinter.Button(self.gui,
text=messages.SEND, command=self.application.send)
        self.send_button.pack()

    def input_dialogs(self):
        self.gui.lower()
        self.application.username =
simpledialog.askstring(messages.USERNAME,
messages.INPUT_USERNAME, parent=self.gui)
        if self.application.username is None:
            return False
        self.application.host =
simpledialog.askstring(messages.SERVER_HOST,
messages.INPUT_SERVER_HOST,
                parent=self.gui)
        if self.application.host is None:
            return False
        self.application.port =
simpledialog.askinteger(messages.SERVER_PORT,
messages.INPUT_SERVER_PORT,
                parent=self.gui)
        if self.application.port is None:
            return False
        return True

    def alert(self, title, message):
        messagebox.showerror(title, message)

    def show_message(self, message):

self.message_list.configure(state=TEXT_STATE_NORMAL)
```

```
        self.message_list.insert(tkinter.END,
str(message) + END_OF_LINE)

self.message_list.configure(state=TEXT_STATE_DISABLED)

    def on_closing(self):
        self.application.exit()
        self.gui.destroy()
```

3 Схемы форматов передачи данных

3.1 Задание на лабораторную работу № 3

Добавьте в свой предыдущий проект возможность сохранения состояния в виде периодического сохранения либо в виде функций импорта/экспорта. Выбранный формат для сериализации должен иметь схему. В проекте должен присутствовать код валидации данных, либо в самой программе при загрузке данных, либо в юнит-тестах, проверяющих корректность сохранения состояния.

3.2 Теоретическая справка

Схема — это описание структуры данных на формальном языке, содержащее допустимые имена, семантические связи между ними и ограничения, накладываемые на значения.

Существует несколько рациональных причин для явного описания схемы данных:

- схемы позволяют уменьшить количество передаваемых данных по сети за счёт исключения из процесса передачи имён полей,
- схемы позволяют валидировать данные созданные сторонними приложениями и проверять корректность сериализации данных во время автоматического тестирования,
- схемы помогают разработчику с генерацией кода представления данных в памяти и с генерацией документации.

Так валидация сериализованных объектов по схеме — отлично сочетается с разработкой через тестирование. Сначала договоритесь по схеме данных, затем согласуйте тесты со схемой и реализуйте удовлетворяющий им код.

Наиболее известно взаимодействие языков программирования со схемами баз данных через технологию ORM. Она позволяет отобразить классы и поля классов на таблицы и их колонки, что делает возможным работу с СУБД в терминах классов, а не таблиц. Аналогичное отображение возможно построить для схем форматов данных. В зависимости от ситуации может быть выгодным генерация классов по схеме, генерация схемы по классам или задание отображения между классами и схемой вручную.

Рассмотрим языки описания схем для форматов данных, с которыми мы познакомились в главе 2.2, посвящённой предыдущей лабораторной работе.

XML Schema

XML Schema — язык описания структуры XML-документа. Файл, содержащий XML Schema, обычно имеет расширение «.xsd» (XML Schema definition).

Приведём пример схемы для примера XML, используемого в предыдущем разделе. Описание элементов XML Schema можно найти на https://www.w3.org/TR/xmlschema-1/.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="recipe">
    <xs:complexType>
        <xs:all>
            <xs:element name="title" type="xs:string"/>
            <xs:element name="instructions"
```

```
type="xs:string"/>
            <xs:element name="ingredients" minOccurs="0">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element name="ingredient"
minOccurs="0" maxOccurs="unbounded" >
                            <xs:complexType>
                                <xs:simpleContent>
                                    <xs:extension
base="xs:string">
                                        <xs:attribute
name="qty" type="xs:nonNegativeInteger"/>
                                    </xs:extension>
                                </xs:simpleContent>
                            </xs:complexType>
                        </xs:element>
                    </xs:sequence>
                </xs:complexType>
            </xs:element>
        </xs:all>
    </xs:complexType>
    </xs:element>
</xs:schema>
```

Поэкспериментируйте с валидацией XML примера приведённой выше схемой в онлайн валидаторе https://www.freeformatter.com/xml-validator-xsd.html.

XML Schema является частью стека для создания веб сервисов, в дополнение состоящего из технологий: WSDL, SOAP и HTTP. WSDL (Web Services Description Language) — язык описания веб-сервисов и доступа к ним, основанный на языке XML. WSDL часто применяется в комбинации с технологией SOAP (протокол обмена сообщениями с веб-сервисом) и XML Schema для создания веб сервисов. Клиентская программа для

получения списка доступных операций веб сервиса может скачать его WSDL файл. Любой тип данных, который фигурирует в этом файле, описывается XML схемой. После чего приложение может запрашивать и отправлять данные в XML формате, вызывая доступные операции у веб-сервиса через SOAP и HTTP.

Пример валидации данных

Приведём пример чтения XML документа с одновременной валидацией. Данный пример можно запустить в онлайн компиляторе Scala.

```
import java.io.{ByteArrayInputStream, File, StringReader}
import javax.xml.parsers.DocumentBuilderFactory
import javax.xml.transform.dom.DOMSource
import javax.xml.validation.SchemaFactory
import org.xml.sax.{ErrorHandler, InputSource,
SAXParseException}

val xml_schema = """<xs:schema
xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:element name="recipe">
 <xs:complexType>
   <xs:all>
     <xs:element name="title" type="xs:string"/>
     <xs:element name="instructions" type="xs:string"/>
     <xs:element name="ingredients" minOccurs="0">
       <xs:complexType>
         <xs:sequence>
           <xs:element name="ingredient" minOccurs="0"
maxOccurs="unbounded" >
             <xs:complexType>
```

```
              <xs:simpleContent>
                <xs:extension base="xs:string">
                  <xs:attribute name="qty"
type="xs:nonNegativeInteger"/>
                </xs:extension>
              </xs:simpleContent>
            </xs:complexType>
          </xs:element>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:all>
 </xs:complexType>
 </xs:element>
</xs:schema>"""

val sf = SchemaFactory.newDefaultInstance()

val dbf = DocumentBuilderFactory.newInstance()
dbf.setNamespaceAware(true)
val db = dbf.newDocumentBuilder()
val schema_dom = db.parse(new InputSource(new
ByteArrayInputStream(xml_schema.getBytes("UTF-8"))))

//val s = sf.newSchema(new File("test-schema.xsd"))
val s = sf.newSchema(new DOMSource(schema_dom))
dbf.setSchema(s)
val validating_db = dbf.newDocumentBuilder()
validating_db.setErrorHandler(new ErrorHandler {
 override def warning(exception: SAXParseException): Unit =
{}
 override def error(exception: SAXParseException): Unit =
throw exception
 override def fatalError(exception: SAXParseException): Unit
= throw exception
```

```
})

try {
 val doc = validating_db.parse(new InputSource(new
StringReader(xml)))
} catch {
 case e:SAXParseException => println(e)
}
```

Так как схема хранится в формате XML, то считываем её уже известным способом. Стандартная XML библиотека в Java поддерживает валидацию, поэтому новых библиотек нам подключать не потребуется. После загрузки схемы, она задаётся в фабрике для DocumentBuilder, который с этого момента будут использовать её во время десериализации документа из текстового представления. По умолчанию все ошибки будут логироваться в поток вывода программы обработчиком ошибок по умолчанию. Для того, чтобы ошибки обрабатывать устанавливают свой обработчик, как это сделано в примере.

JSON Schema

```
{
    "$schema": "https://json-schema.org/draft/2019-
09/schema",
    "title": "recipe",
    "type": "object",
    "properties": {
        "title": {
            "type": "string"
        },
        "instructions": {
            "type": "string"
```

```
        },
        "ingredients" : {
            "type": "array",
            "items": {
                "type":"object",
                "properties": {
                    "name": {
                        "type": "string"
                    },
                    "qty": {
                        "type": "number"
                    }
                },
                "required" : ["name", "qty"]
            }

        }
    },
    "required": [
        "title",
        "instructions"
    ]
}
```

Поэкспериментируйте с валидацией JSON примера приведённой выше схемой в онлайн валидаторе http://json-schema-validator.herokuapp.com/.

Пример валидации данных

В данном примере мы будем использовать ту же библиотеку, что и на сайте для онлайн валидации. Библиотека валидации внутри использует Jackson, поэтому, во избежании

избыточности, для чтения JSON будет использоваться Jackson вместо GSON.

Как и множество других Java библиотек, библиотеку валидации json-schema-validator можно скачать с maven репозитория https://mvnrepository.com/. В примере используется версия 2.2.6. Также у проекта есть github сайт https://github.com/java-json-tools/json-schema-validator.

```
import com.github.fge.jackson.JsonLoader
import com.github.fge.jsonschema.main.JsonSchemaFactory

val json = """{
    "title": "Бутерброд",
    "instructions": "Размажьте масло равномерно по хлебу и
положите сверху колбасу.",
    "ingredients": [
        {
           "name": "Кусочек хлеба",
           "qty": 1
        },
        {
           "name": "Кусочек колбасы",
           "qty": "2"
        },
        {
           "name": "Кусочек масла",
           "qty": 2
        }
    ]
}
"""
val json_example = JsonLoader.fromString(json)

val json_schema = """{
```

```
   "$schema": "https://json-schema.org/draft/2019-
09/schema",
   "title": "recipe",
   "type": "object",
   "properties": {
       "title": {
           "type": "string"
       },
       "instructions": {
           "type": "string"
       },
       "ingredients" : {
           "type": "array",
           "items": {
               "type":"object",
               "properties": {
                   "name": {
                       "type": "string"
                   },
                   "qty": {
                       "type": "number"
                   }
               },
               "required" : ["name", "qty"]
           }

       }
   },
   "required": [
       "title",
       "instructions"
   ]
}
"""
```

```
val raw_schema = JsonLoader.fromString(json_schema)
val sf = JsonSchemaFactory.byDefault()
val schema = sf.getJsonSchema(raw_schema)

val report = schema.validate(json_example)
println(report)
```

Код этого примера довольно простой. Из строки json_schema со схемой инициализируется JSON объект, который используется для инициализации объекта типа JsonSchema. Затем загружается JSON с данными и запускается валидация по схеме вызовом у объекта JsonSchema метода validate. Все выявленные нарушения сохраняются в отчёт report.

Существуют и другие библиотеки для валидации, сравнение которых приводится на сайте https://github.com/ebdrup/json-schema-benchmark.

Protocol Buffer

Для Protocol Buffers схема является обязательной частью формата. Парсинг бинарного сообщения полагается на информацию, хранимую в .proto схеме, без которой извлечь полезную информацию крайне сложно. Приведём .proto описание сообщения ещё раз:

```
syntax = "proto2";

message Ingredient {
   required string name = 1;
   optional int32 qty = 2;
}

message Recipe {
```

```
   required string title = 1;
   required string instructions = 2;
   repeated Ingredient ingredients = 3;
}
```

Существует две причины, по которым парсинг Protobuf бинарного сообщения может абортироваться:

- во входном сообщении отсутствуют required поля,
- входная последовательность байт — это не protobuf сообщение.

Ошибки представлены классом InvalidProtocolBufferException, который содержит в себе несколько уточняющих вариантов:

- truncatedMessage — поток оборвался в середине чтения значения поля,
- negativeSize — указанная длина сообщения отрицательная,
- maformedVarint — неправильно закодирован тип varint,
- invalidTag — встретился нулевой тэг,
- invalidEndTag — неверный завершающий тэг группы,
- invalidWireType — неверный бинарный тип,
- recursionLimitExceeded — количество уровней вложенности больше порогового,
- sizeLimitExceeded — размер сообщения больше порогового,
- parseFailure — ошибка при разборе сообщения,
- invalidUtf8 — кодировка UTF-8 неверная.

Если же в сообщении больше полей, чем в используемой схеме, то приложение не будет останавливаться — новые поля пропускаются.

Protocol Buffers гарантирует соответствие типов и структуры сообщения, но в нём нет средств для описания семантических связей между значениями, аналогичных XML Schema. Этот дизайн был выбран разработчиками сознательно в угоду простоты формата.

Пример валидации данных

Рассмотрим пример чтения испорченного сообщения с обработкой ошибок.

```
import java.io.FileInputStream
import com.google.protobuf.InvalidProtocolBufferException

val recipeBuilder = Message.Recipe.newBuilder()
recipeBuilder.setTitle("Бутерброд")
recipeBuilder.setInstructions("Размажьте масло равномерно по
хлебу и положите сверху колбасу.")
recipeBuilder.addIngredientsBuilder().setName("кусочек
хлеба").setQty(1)
recipeBuilder.addIngredientsBuilder().setName("кусочек
колбасы").setQty(1)
recipeBuilder.addIngredientsBuilder().setName("кусочек
масла").setQty(1)
val msg = recipeBuilder.build()
print(msg)

val output_stream = new
java.io.FileOutputStream("corrupted_recipe")
output_stream.write("corruption".getBytes)
msg.writeTo(output_stream)
output_stream.close()

try {
```

```
 val read_msg = Message.Recipe.parseFrom(new
FileInputStream("corrupted_recipe"))
} catch {
 case e:
 case e:InvalidProtocolBufferException => print(e)
}
```

Мы рассмотрели в данной главе несколько языков описания схем для XML, JSON и Protocol Buffer. Наиболее сложный и развитый из них — это XML Schema. Схемы Protocol Buffer являются неотделимой частью формата. Конечно использование схем требует дополнительных усилий.

На практике по мере развития проекта при применении схем возникает проблема прямой и обратной совместимости. Например, такое происходит во время поэтапных обновлений. Такие обновления характерны для распределённых приложений, к которым принадлежит и класс клиент/серверных приложений. При обновлении подобных систем существует период времени в течении которого друг с другом взаимодействую старые и новые версии компонентов. Поэтому в рассмотренных языках в той или иной степени поддерживается эволюция схемы.

3.3 Контрольные вопросы

1. Что такое схема формата данных?
2. Какие языки описания схем можете привести в пример?
3. Для чего используются схемы форматов данных?
4. Какие преимущества и недостатки использования схем форматов данных?

5. В чём состоит проблема эволюции схемы?

Список литературы

1. Marrs, T. JSON at Work: Practical Data Integration for the Web / T. Marrs. — O'Reilly Media, Inc., 2017. – 376 p.

2. Mihalcea, V. High-Performance Java Persistence / V. Mihalcea. — VLAD MIHALCEA, 2016. — 486 p.

3. Sommerville, I. Software Engineering / I. Sommerville. – Pearson Education, 2004. — 781 p.

4. Urma, R.-G. Java 8 in Action: Lambdas, Streams, and functional-style programming / R. — G. Urma, M. Fusco, A. Mycroft. — Manning Publications, 2014. — 424 p.

5. Александреску, А. Современное проектирование на C++: Обобщенное программирование и прикладные шаблоны проектирования / А. Александреску. — СПб.: Вильямс, 2008. — 336 с.

6. Валиков, А.Н. Технология XSLT / А.Н. Валиков. — СПб.: БХВ-Петербург, 2002. — 544 с.

7. Кэгл, К. XML / К. Кэгл, Д. Хантер, Д. Гиббонс. — М.: Лори, 2006. — 638 с.

8. Майерс, С. Эффективное использование C++. 50 рекомендаций по улучшению ваших программ и проектов / С. Майерс. — СПб.: Питер, 2006. — 240 с.

9. Майерс, С. Эффективное использование C++. 35 новых способов улучшить стиль программирования / С. Майерс. — СПб.: Питер, 2006. — 224 с.

10. Мартин, Р. Чистая архитектура. Искусство разработки программного обеспечения / Р. Мартин. — СПб.: Питер, 2019. — 352 с.

11. Халперн, П. Стандартная библиотека C++ на примерах / П. Халперн. — М.: Вильямс, 2001. — 336 с.

12. Элджер, Дж. C++: Библиотека программиста / Дж. Элджер. — СПб.: Питер, 1999. — 320 с.

13. Shared libraries [Электронный ресурс]. — URL: https://tldp.org/HOWTO/Program-Library-HOWTO/shared-libraries.html (дата обращения 12.05.2020).

14. Dynamic-Link Libraries [Электронный ресурс]. — URL: https://docs.microsoft.com/en-us/windows/win32/dlls/dynamic-link-libraries (дата обращения 12.05.2020).

15. Kerrisk M. The Linux programming interface: a Linux and UNIX system programming handbook. – No Starch Press, 2010.

16. Friesen, Jeff. Java XML and JSON: Document Processing for Java SE. Apress, 2019.

17. Tim Bray. Extensible Markup Language (XML) 1.0 (Fifth Edition). [Электронный ресурс]. — URL:https://www.w3.org/TR/xml/ (дата обращения 12.05.2020).

Учебное издание

ЛАБОРАТОРНЫЕ РАБОТЫ ПО КУРСУ
«Технологии программирования»

Учебно-методическое пособие

Составители: Андрей Викторович Гайдель,
Владимир Игоревич Проценко,
Александр Владимирович Благов

Редактор ???
Компьютерная правка ???
Подписано в печать ???. Формат 60x84 1/16.
Бумага офсетная. Печать офсетная. Печ. л. ???.
Тираж ??? экз. Заказ ???. Арт. – ???/2020.

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО
ОБРАЗОВАНИЯ «САМАРСКИЙ НАЦИОНАЛЬНЫЙ
ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ имени академика
С.П. КОРОЛЕВА»
443086, САМАРА, МОСКОВСКОЕ ШОССЕ, 34

Изд-во Самарского университета.
443086, Самара, Московское шоссе, 34.