



Московский государственный университет им. М. В. Ломоносова  
Факультет вычислительной математики и кибернетики

И. Г. Головин

# Практикум на ЭВМ. Модельный веб-сервер

Методическое пособие для студентов 2 курса

Москва  
2009

УДК 519.68  
ББК

*Печатается по решению Редакционно-издательского совета факультета  
вычислительной математики и кибернетики  
МГУ им. М. В. Ломоносова*

Рецензенты:

проф., д.ф.-м.н. И. В. Машечкин  
доцент, к.ф.-м.н. А. Н. Терехин

**И. Г. Головин**

Практикум на ЭВМ. Модельный веб-сервер: Методическое пособие для студентов II курса. — М.: Издательский отдел факультета ВМиК МГУ им. М.В.Ломоносова (лицензия ИД № 05899 от 24.09.2001), 2009 – 34 с.

ISBN-13 978-5-89407-408-5

Описывается задание практикума на ЭВМ для студентов II курса факультета ВМиК. Приведена постановка задачи, методические рекомендации по выполнению задания, варианты задания.

Для студентов факультета ВМК в поддержку основного лекционного курса «Системы программирования» и для преподавателей, ведущих практические занятия по этому курсу.

ISBN-13  
978-5-89407-439-9

© Издательский отдел факультета  
вычислительной математики  
и кибернетики  
МГУ им. М. В. Ломоносова, 2009  
© И. Г. Головин, 2009

Материалы для студентов ВМиК <http://www.cmcmsu.info/>

<b>1. Введение .....</b>	<b>5</b>
<b>2. Постановка задачи.....</b>	<b>6</b>
<b>3. Основные понятия.....</b>	<b>7</b>
3.1. Унифицированный идентификатор ресурса .....	8
3.2. HTTP-запрос .....	9
3.2.3 Метод GET .....	10
3.2.4 Метод HEAD .....	11
3.3. HTTP-ответ .....	11
3.3.3 Коды состояния, возвращаемые модельным сервером .....	12
3.4. HTTP-заголовок .....	13
3.4.3 Заголовок Date .....	14
3.4.4 Заголовок Host .....	14
3.4.5 Заголовок Referer .....	14
3.4.6 Заголовок User-agent .....	15
3.4.7 Заголовок Server .....	15
3.4.8 Заголовок Content-length .....	15
3.4.9 Заголовок Content-type .....	15
3.4.10 Заголовок Allow .....	15
3.4.11 Заголовок Last-modified .....	16
<b>4. Методические указания по выполнению первого этапа задания .....</b>	<b>17</b>
4.1. Внутренняя организация сервера .....	17
4.2. Общий интерфейс шлюза — основные понятия .....	19
4.2.3 Пример CGI-программы .....	20
<b>5. Методические указания по выполнению второго этапа задания .....</b>	<b>22</b>
<b>6. Методические указания по выполнению третьего этапа задания .....</b>	<b>25</b>
6.1. Способы встраивания программ на модельных языках в веб-сервер .....	26
6.2. Описание модельных языков .....	27
6.2.3 Модельный язык программирования .....	28
6.2.4 Модельный JavaScript (MJS).....	28
6.2.4.1. Переменные.....	28
6.2.4.2. Простые типы данных и операции .....	29
6.2.4.3. Составные типы данных.....	29
6.2.4.4. Встроенные объекты MJS .....	31
6.2.4.5. Универсальные значения и операции.....	33
6.2.4.6. Преобразования типов.....	34
6.2.4.7. Синтаксис MJS.....	35
<b>7. Список литературы .....</b>	<b>37</b>



# 1. Введение

Данное пособие содержит описание задания по практикуму на ЭВМ для студентов 2 курса факультета ВМиК МГУ. Задание поддерживает основной курс «Системы программирования», читаемый в четвертом семестре, и предназначено для выполнения в среде операционной системы семейства UNIX на языке Си++. Методическая цель задания — закрепить следующие знания и навыки:

- работа в системе программирования Си/Си++ в ОС семейства UNIX;
- разработка сетевых серверных приложений, использующих механизм сокетов (sockets);
- использование элементов теории формальных языков для реализации трансляторов с языков программирования.

Предполагается, что задание будет выполняться в течение четвертого семестра (в течение восьми недель), однако по усмотрению преподавателя выполнение задания можно начать и третьем семестре, поскольку задание состоит из трех частей. Первые две части не требуют знаний, преподаваемых в четвертом семестре (прежде всего — элементов теории формальных языков и основ построения трансляторов) и поэтому могут выполняться и в третьем семестре (на языке Си).

Конкретный вариант задания и сроки его выполнения определяются, разумеется, конкретным преподавателем. Можно варьировать следующие аспекты:

- набор реализуемых запросов и заголовков протокола HTTP;
- внутренняя организация сервера;
- способ встраивания программ на модельном языке в сервер;
- конкретный вариант синтаксиса и семантики модельного языка (набор типов данных и операций над ними, операторов, стандартных функций).

В пособии приведены минимальные требования, которые должны быть реализованы.

На официальном сайте кафедры алгоритмических языков <http://www.al.cs.msu.ru> можно найти дополнительные варианты задания.

## 2. Постановка задачи

Задание практикума разбивается на следующие подзадачи:

- Реализация модельного веб-сервера, поддерживающая описанное ниже подмножество протокола HTTP.
- Реализация поддержки общего шлюзового интерфейса (CGI) в разработанном веб-сервере.
- Реализация интерпретатора модельного языка описания сценариев для написания CGI-сценариев (варианты языков описаны ниже).  
Язык реализации — Си++ [1].

На каждом из шагов задания необходимо разработать тестовые программы, демонстрирующие работоспособность программы, реализованной на соответствующем шаге. Набор этих программ конкретизируется ниже в методических указаниях по реализации задания.

## 3. Основные понятия

Веб-сервер – это частный случай архитектуры «клиент-сервер». В этой архитектуре программа-сервер принимает и обрабатывает запросы от программ-клиентов. Веб-архитектура характеризуется двумя особенностями: во-первых, в качестве транспортного протокола используется семейство протоколов TCP/IP, и при этом между клиентом и сервером устанавливается потоковое (или надежное) соединение (подробнее об этом говорится в курсе «Операционные системы»[3]). Во-вторых (и это главное), в качестве прикладного протокола взаимодействия между клиентом и сервером используется протокол HTTP (hyper-text transport protocol) – протокол передачи гипертекста. Поэтому веб-серверы также часто называются HTTP-серверами. Ниже мы будем использовать название «веб-сервер».

HTTP-протокол использует простую модель «запрос-ответ». Веб-сервер принимает от веб-клиентов (обычно такими клиентами являются веб-браузеры, но могут быть и программы — поисковые роботы, клиенты веб-служб и т.д.) HTTP-запросы, обрабатывает их и посылает клиенту HTTP-ответ. Запрос клиента, как правило, содержит так называемый URI (uniform resource identifier) — «унифицированный идентификатор ресурса». Этот идентификатор содержит ссылку на местонахождение какого-либо ресурса на сервере. Примеры ресурсов — простой текст, гипертекст, изображение, звуковой файл и т.д. В общем случае, ресурсом может быть не только файл, но и любая информация, которая может быть программно сгенерирована и передана по протоколу TCP/IP. Ответ сервера обязательно содержит код возврата, а также, возможно, и запрошенную информацию.

Сайты, в которых информация (ресурсы) представляет собой набор файлов (обычно — в формате HTML — т.н. веб-страницы), называются статическими. Модельный веб-сервер, который требуется реализовать на первом этапе задания, поддерживает только статические сайты. Однако возможности веб-серверов по-настоящему раскрываются только на динамических сайтах, в которых ресурсы могут генерироваться и обновляться программно (т.е. динамически) во время обработки клиентского запроса. Самая простая и ранняя технология поддержки динамических сайтов — CGI — описывается ниже. Реализация поддержки такой технологии в модельном сервере представляет собой второй этап задания.

Особенностью HTTP-протокола, существенно упрощающей его реализацию, является т.н. отсутствие состояний. Это означает, что веб-сервер не обязан хранить историю запросов (хотя, конечно, может это делать в целях оптимизации и т.д.). Кроме того, вся служебная информация передается исключительно в текстовом виде (более того, в 7-битной кодировке ASCII).

Кроме поддержки HTTP-протокола реальные веб-серверы выполняют и множество других функций: аутентификацию доступа к ресурсам, сбор ста-

истики запросов, поддержка защищенных соединений по протоколу HTTPS и т.д. Однако из соображений простоты в модельном веб-сервере эти функции не поддерживаются.

Таким образом, основные понятия, связанные с веб-серверами, — это URI, HTTP-запрос и HTTP-ответ. Ниже мы рассмотрим эти понятия подробнее.

## 3.1. Унифицированный идентификатор ресурса

Унифицированный идентификатор ресурса (*URI*) — это строка символов (латиницы), которая определяет какой-либо ресурс (документ, изображение, службу и т.д.). Частным случаем *URI* является «Унифицированный локатор ресурса» (*URL*), который определяет не только сам ресурс, но и его местонахождение в сети и/или на компьютере.

Общий вид URI:

*⟨схема⟩ : ⟨ссылка URI⟩*

*⟨схема⟩* чаще всего обозначает сетевой протокол (*http*, *https*, *ftp*, *mailto* и т.д.), но не только, например, схема *file* означает локальную файловую систему, схема *ed2k* — файлообменную сеть eDonkey и т.д.

*⟨ссылка URI⟩* обозначает непосредственный идентификатор ресурса, вид которого зависит от схемы. Мы не будем рассматривать общий вид ссылки *URI* (равно как и полный список зарегистрированных схем — см. соответствующие стандарты [2]).

Для модельного сервера сделаем два упрощения. Во-первых, будем считать, что модельный сервер допускает только ресурсы в виде файлов данных или исполняемых файлов с CGI-программами (подробнее о CGI-программах и сценариях см. ниже). Во-вторых, будем считать, что *URI* имеет следующую (упрощенную) структуру:

*⟨схема⟩://⟨хост⟩⟨путь к файлу-ресурсу⟩ [?⟨параметры запроса⟩]*

Здесь *⟨хост⟩* указывает на адрес сайта в сети Интернет, возможно включающий в себя имя пользователя, пароль, порт (например, *www.w3c.org*, *127.0.0.1:8888*, *anonymous:goga1234@www.mysite.ru:8080*). Общий вид задания хоста нас интересовать не будет, т.к. задачу выделения адреса хоста из *URI* берет на себя веб-клиент (например, браузер).

Так, если пользователь вводит в адресной строке браузера *URI* следующего вида:

```
http://127.0.0.1:8888/cgi-bin/testcgi?name=igor&surname=
golovin&mail=golovin
```

то браузер выделяет схему (*http*), адрес веб-сервера (*127.0.0.1:8888*), устанавливает потоковое TCP/IP-соединение с сервером (в данном случае сервер работает на той же машине, что и браузер, т.к. адрес *127.0.0.1* — локальный)

и посылает серверу HTTP-запрос, который начинается со следующего заголовка (строки):

```
GET cgi-bin/testcgi?name=igor&surname=golovin&mail=igolovin
HTTP/1.0
```

Таким образом, веб-сервер видит в запросе только часть *URI*, а именно:

- *«путь к файлу-ресурсу»*;
- *«параметры запроса»*.

Для модельного веб-сервера *«путь к файлу-ресурсу»* — это корректный путь к запрашиваемому файлу-ресурсу, который начинается от «домашней» директории веб-сервера. Обычно такая директория задается в параметрах конфигурации веб-сервера, а для модельного сервера будем считать, что «домашняя» директория — это директория, откуда запущен веб-сервер.

*«параметры запроса»* — это последовательность пар *«имя=значение»*, разделенных знаком амперсанда &.

Параметры запроса нужны только в случае, когда запрашиваемый файл-ресурс является исполняемой CGI-программой. В случае обычных файлов (HTML и т.п.) параметры не нужны.

Сделаем важное замечание о кодировке *URI*. По стандарту *URI* должен записываться только латинскими символами (сейчас ведется работа по стандартизации международного идентификатора ресурса (IRI), но она ещё не закончена). Символы национальных алфавитов (русского и т.д.), а также специальные символы (пробелы, амперсанды, проценты, угловые скобки и т.п.) должны кодироваться латиницей (например, пробел кодируется последовательностью 3 символов %20). Для простоты анализа *URI* будем считать, что ни путь к файлу, ни строки в параметрах запроса не содержат русских букв и спецсимволов. Потребуем, чтобы имена и значения содержали только латинские буквы и цифры. Таким образом, мы игнорируем проблемы кодировки *URI*.

## 3.2. HTTP-запрос

HTTP-запрос отправляется веб-клиентом к веб-серверу и имеет вид:

```
<заголовок запроса> <конец строки>
<HTTP-заголовок> <конец строки>
...
<HTTP-заголовок> <конец строки>
<конец строки>
<тело запроса>
```

Здесь и далее *«конец строки»* — это символ с кодом 10 («\n»).

Структура заголовка запроса имеет вид:

*«HTTP-метод» «URI» HTTP/«версия протокола»*

«версия протокола» — это 2 арабские цифры, разделенные точкой (*версия.подверсия*). Сейчас используются версии HTTP-протокола 1.0 и 1.1.

Структура *URI* описана выше.

HTTP-метод определяет семантику запроса к серверу. Веб-сервер обязан поддерживать только два метода: *GET* и *HEAD*. Для простоты модельный сервер будет поддерживать только их.

### 3.2.3 Метод GET

Этот метод используется для запроса содержимого ресурса с сервера. Файл ресурса определяется *URI* из запроса. Если *URI* корректен, то сервер обязан вернуть содержимое запрашиваемого файла, если это файл данных. Если же файл — это CGI-программа, то она запускается, ей передаются параметры запроса из *URI* (они отделяются знаком вопроса), и сервер возвращает клиенту сгенерированный программой текст.

Тело запроса для метода *GET* пусто.

Недостатком метода GET является то, что параметры запроса явно передаются в адресной строке браузера и могут быть видны пользователю. Если параметры запросы включают приватную или конфиденциальную информацию (пароли, коды доступа и т.п.), то это может быть неприемлемо. Поэтому часто вместо метода *GET* используется метод *POST*, в заголовке которого *URI* не содержит параметров. Вместо этого параметры запроса передаются в теле сообщения. В модельном веб-сервере поддержка метода *POST* не обязательна.

Примеры заголовков запроса GET:

```
GET /index.html HTTP/1.1
```

По этому запросу сервер возвращает текст файла *index.html* из домашней директории сервера.

```
GET /cgi-bin/testcgi?name=igor&surname=golovin&mail=igolovin
HTTP/1.1
```

По этому запросу сервер запускает программу *cgi-bin/testcgi*, передав ей параметры запроса через переменную окружения. Сгенерированный программой текст включается сервером в ответ клиенту (подробнее об этом см. ниже в разделе «Общий шлюзовой интерфейс»).

Особым случаем запроса *GET* является случай, когда *URI* не содержит пути к ресурсу (а только адрес сайта). Например, пользователь может набрать в адресной строке браузера:

```
http://www.mysite.ru
```

(заметим, что это типичный случай начала веб-серфинга).

Браузер сгенерирует следующий заголовок запроса:

GET / HTTP/1.1

Веб-сервер по этому запросу должен показать т.н. главную страницу сайта — страницу по умолчанию. Будем считать, что модельный сервер покажет в этом случае страницу с именем *index.html* из домашней директории сервера.

### 3.2.4 Метод HEAD

Метод HEAD аналогичен методу GET с той разницей, что актуальное содержимое ресурса не передается клиенту. Сервер генерирует ответ, содержащий только заголовок ответа и HTTP-заголовки (тело ответа отсутствует). Метод HEAD используется веб-клиентами (браузерами) для проверки существования ресурса, извлечения метаданных (через HTTP-заголовки) и т. д. Кроме того метод позволяет выяснить, не изменился ли ресурс со времени последнего обращения к нему. Это очень важно, так как позволяет обеспечить кэширование ресурсов.

Дело в том, что по стандарту HTTP-протокола метод GET считается «идемпотентным», то есть выдающим одинаковые результаты на одинаковые запросы (при условии неизменности ресурса). Дата и время обновления ресурса указываются в ответе сервера на методы *GET/HEAD* (в HTTP-заголовке *Last-modified*). Поэтому браузер имеет возможность сохранить ресурс локально при первом обращении к нему и при последующих обращениях использовать эту копию без настоящей загрузки в случае, если ресурс не изменился. А выяснить неизменность ресурса и помогает метод *HEAD*.

## 3.3. HTTP-ответ

Ответ веб-сервера имеет следующую структуру:

```
<заголовок ответа> <конец строки>
<HTTP-заголовок> <конец строки>
...
<HTTP-заголовок> <конец строки>
<конец строки>
<тело ответа>
```

Заголовок ответа выглядит так:

HTTP/*<версия протокола>* *<код состояния>* *<пояснение>*

*<версия протокола>* — то же самое, что и в HTTP-запросе. Модельный сервер для простоты поддерживает версию 1.0.

*<код состояния>* — десятичное число (три арабских цифры), характеризующее дальнейшее сообщение и определяющее реакцию клиента.

*<пояснение>* — короткая необязательная строка, содержащая пояснение к коду состояния. Нужна только для облегчения анализа ответа человеком.

Веб-сервер генерирует ответ на любой запрос веб-клиента.

В случае, если метод в запросе не поддерживается сервером, то последний должен вернуть код состояния 501 (пояснение — *Not implemented*). При этом в ответ должен включаться HTTP-заголовок *Allow*, содержащий список допустимых методов (описание заголовков, поддерживаемых модельным сервером см. ниже).

Если же метод поддерживается сервером (т.е. это *GET* или *HEAD* для модельного сервера), и файл, идентифицируемый *URI*, допустим и доступен, то сервер выдает ответ с кодом состояния 200 (Ok). Текст файла-ресурса включается в ответ как тело ответа, при этом в ответ должны включаться заголовки *Content-type*, определяющий тип содержимого ресурса (см. ниже), и *Content-length*, содержащий длину тела в байтах. Вообще говоря эти заголовки должны включаться в ответ всегда, когда он содержит тело (а не только заголовки).

И, наконец, возможен случай (весьма частый!), когда указанный в запросе ресурс не найден. В этом случае выдается ответ с кодом статуса 404 (Not found). При этом сервер должен вернуть более развернутое, чем Not found, гипертекстовое пояснение в теле ответа (правда, в случае метода HEAD этого делать не надо).

Ниже приводится информация о всех кодах состояния, возвращаемых модельным веб-сервером, и соответствующий список заголовков (о самих заголовках — см. ниже).

### 3.3.3 Коды состояния, возвращаемые модельным сервером

Коды состояния разбиваются на пять групп (номер группы — первая цифра кода).

- 100–199 — Информация. Модельный сервер их не поддерживает.
- 200–299 — Успех. Сервер возвращает такие ответы в случае успешной и безошибочной обработки запроса.
- 300–399 — Перенаправление. Модельный сервер их не поддерживает.
- 400–499 — Ошибка клиента. Ответы информируют об ошибках в запросе. Для всех методов кроме HEAD сервер должен вернуть в теле ответа развернутое гипертекстовое сообщение об ошибке, которое клиент-браузер должен показать пользователю.
- 500–599 — Ошибка сервера. Ответы информируют об ошибках по вине сервера. Также, как и в случае четвертой группы, в теле ответа содержится гипертекстовое сообщение об ошибке.

В нижеследующей таблице перечислен минимальный набор кодов состояния, возвращаемых модельным веб-сервером.

Таблица 1. Некоторые коды состояния модельного веб-сервера

Код состояния и пояснение	Смысл	HTTP-заголовки
200 OK	Запрос ресурса успешен	Date, Server, Content-type-для GET, Content-length – для GET, Last-modified, тело – для GET
400 Bad request	Синтаксическая ошибка в запросе	Date, Server, Content-type, Content-length, тело
403 Forbidden	Запрос ресурса, который недоступен клиенту	Date, Server, Content-type, Content-length, тело
404 Not Found	Запрос несуществующего ресурса	Date, Server, Content-type, Content-length, тело
500 Internal Server Error	Любая ошибка сервера, которая не входит в список ошибок 5 класса	Date, Server, Content-type, Content-length, тело
501 Not Implemented	Сервер не имеет возможности обработать запрос (например, не поддерживает метод)	Date, Server, Allow, Content-type, Content-length, тело
503 Service Unavailable	Сервер временно не имеет возможности обработать запрос (например, из-за нехватки системных ресурсов, перегрузки и т.п.)	Date, Server, Content-type, Content-length, тело

## 3.4. HTTP-заголовок

Каждый HTTP-заголовок представляет собой строку вида:

*⟨имя заголовка⟩ : ⟨значение⟩*

Двоеточие должно следовать сразу за именем заголовка. Значение может содержать произвольные символы, кроме «\n» (перевод строки) и «\r» (возврат каретки).

Заголовки делятся на 4 группы:

- Основные заголовки — должны входить в любой запрос и ответ;
- Заголовки запроса — входят только в запрос от клиента;

- Заголовки ответа — входят только в ответ сервера;
- Заголовки сущности — сопровождают каждую сущность запроса или ответа

Модельный сервер поддерживает следующие заголовки:

- Основные — *Date*
- Запрос — *Host, Referer, User-agent*
- Ответ — *Server*
- Сущности — *Content-length, Content-type, Allow, Last-modified*

### 3.4.3 Заголовок Date

Заголовок *Date* содержит дату генерации сообщения. Формат даты, поддерживаемый модельным веб-сервером:

*Www, dd Mmm YYYY hh:mm:ss GMT*

Обратите внимание, что время указывается по Гринвичу (GMT). Здесь:

- *Www* — первые три буквы дня недели (по-английски), например, *Wed*;
- *dd* — день (две цифры);
- *Mmm* — первые три буквы названия месяца (по-английски), например, *Apr*;
- *YYYY* — год (четыре цифры);
- *hh:mm:ss* — часы, минуты, секунды, соответственно (две цифры).

Пример:

```
Date: wed, 01 Apr 2009 21:00:05 GMT
```

### 3.4.4 Заголовок Host

Заголовок *Host* содержит доменное имя хоста и порт сервера для запрашиваемого ресурса.

Пример:

```
Host: a1.cs.msu.ru:8080
```

### 3.4.5 Заголовок Referer

Содержит URI ресурса, с которого клиент сделал текущий запрос.

Пример:

```
Referer: http://127.0.0.1/testpage.html
```

### 3.4.6 Заголовок User-agent

Содержит название программы-клиента и его характеристики.

Пример:

```
User-Agent: SomeStrangeBrowser/0.2
```

### 3.4.7 Заголовок Server

Содержит название веб-сервера и его характеристики.

Пример:

```
Server: Model HTTP Server/0.1
```

### 3.4.8 Заголовок Content-length

Содержит (в десятичном виде) число байтов в теле сообщения.

Пример:

```
Content-Length: 95
```

### 3.4.9 Заголовок Content-type

Содержит т.н. MIME-формат возвращаемого ресурса. MIME-формат записывается в виде: *<тип>/<подтип>*.

Модельный сервер поддерживает следующие форматы: *text/plain*, *text/html*, *image/jpeg*.

Пример:

```
Content-Type: text/html
```

### 3.4.10 Заголовок Allow

Содержит список методов, поддерживаемых сервером.

Пример:

```
Allow: GET, HEAD
```

### 3.4.11 Заголовок Last-modified

Содержит дату последней модификации запрошенного ресурса. Формат аналогичен формату даты в заголовке *Date*.

Пример:

```
Date: wed, 01 Apr 2009 21:32:12 GMT
```

## 4. Методические указания по выполнению первого этапа задания

Целью первого этапа задания является реализация статического веб-сервера, поддерживающего подмножество HTTP-протокола, описанное выше.

Рекомендуется начать реализацию с двух простых программ, которые пригодятся при тестировании.

Первая программа — «псевдо-сервер», цель которого — запись реальных запросов, посылаемых веб-клиентами (например, различными веб-браузерами). Такой сервер должен принять запрос, записать его в лог-файл, выдать ответ с кодом 501 «*Not Implemented*» и немедленно закрыть соединение. Ответ можно заготовить заранее как текстовый файл и выдавать его в сокет по мере надобности. Сохраненные запросы можно использовать для отладки сервера.

Для того, чтобы посылать эти запросы к серверу, понадобится еще одно простое приложение — «псевдо-браузер». Это консольное приложение, которое устанавливает связь с сервером, посылает ему заранее заготовленный запрос (тут-то пригодится «псевдо-сервер», хотя тестовые запросы можно приготовить и «вручную») и записывает ответ сервера.

### 4.1. Внутренняя организация сервера

Рассмотрим вначале самую простую схему организации сервера, назовем ее «монопольной». В этой схеме сервер принимает запрос от клиента, обрабатывает его, формирует ответ, отправляет его клиенту и закрывает соединение. Схема монопольна, потому что сервер не может принимать других запросов, пока не обработает текущий запрос.

Псевдокод для такой схемы может выглядеть так:

```
// обычная подготовка TCP/IP сервера
int serverSocket = socket ( AF_INET, SOCK_STREAM, 0 );
// ...

struct sockaddr_in ServerAddress;
// заполнить ServerAddress
```

```

// ...
if ( bind ( serverSocket, &ServerAddress,
           sizeof ( ServerAddress ) ) < 0 )
{
    // фатальная ошибка
    // ...
}

if ( listen ( serverSocket, BACK_LOG ) < 0 )
{
    // фатальная ошибка
    // ...
}

// главный цикл
for (;;)
{
    struct sockaddr_in ClientAddress;
    size_t CAddrLen = sizeof ( ClientAddress );

    // ждем очередного клиента
    int cSocket = accept ( ( serverSocket, &ClientAddress,
                          &CAddrLen );

    if ( cSocket < 0 )
    {
        // ошибка – если будет повторяться, то фатальна
        // ...
    }

    /* собственно обработка запроса. Должна включать
       в себя корректный разрыв связи (shutdown – close) */
    ProcessClientRequest ( cSocket, &ClientAddress );
}
// ...

```

Приведенная схема имеет одно (и, пожалуй, единственное) достоинство — простоту. Например, псевдо-сервер (см. выше) целесообразно реализовать именно так. Однако никакой реальный сервер не использует эту схему по причине её крайней неэффективности. Проблема в том, что обработка корректного клиентского *GET*-запроса требует копирования файла (указанного в *URI* запроса) в клиентский сокет. Эта операция может занять много времени. Особенно нетерпимо то, что сервер в течение этого времени будет простаивать, ожидая завершения обмена с файлом и сокетом. Ведь в это же время можно обрабатывать запросы других клиентов! Поэтому более эффективная (и более сложная в реализации) схема использует асинхронный ввод/вывод. Главную роль в этой схеме играет системный вызов *select()*. Прочитать о том, как *select()* используется для асинхронного ввода/вывода можно в [4, гл.6]. Здесь отметим, что для достижения максимальной эффективности асинхронным должен быть не только обмен с сокетами, но и обмен с файлами-ресурсами. Для этого дескриптор открытого файла ресурса должен быть переведен в режим неблокирующего ввода с помощью системного вызова *ioctl()*. Заметим, что особенность реализации механизма сокетов в ОС UNIX такова, что открытые файловые дескрипторы не отличаются от открытых сокетов с точки зрения системного вызова *select()*, поэтому *select()* реагирует не только на «сокетные» события, но и на «файловые» события, что существенно упрощает реализацию асинхронного ввода/вывода.

## 4.2. Общий интерфейс шлюза — ОСНОВНЫЕ ПОНЯТИЯ

Общий интерфейс шлюза (или общий шлюзовой интерфейс — по-английски — Common Gateway Interface, CGI) — это стандартизованный протокол, позволяющий подключить к веб-серверу внешние программы для генерации содержимого запрашиваемых ресурсов. Такие программы называются «шлюзами», но мы будем использовать более распространенный термин «CGI-программа». Будем считать, что все CGI-программы модельного веб-сервера располагаются в директории `cgi-bin` из домашней директории сервера.

В случае если ресурс, запрошенный клиентом есть не файл данных, а исполняемая программа, то веб-сервер запускает эту программу и вывод этой программы направляет серверу. CGI определяет, каким образом информация о сервере и запросе передается в CGI-программу, и каким образом CGI-программа передает информацию обратно.

В отличие от обычных консольных приложений, получающих информацию при запуске через аргументы командной строки, CGI-программы получают ее через набор переменных окружения. В случае запросов *PUT* и *POST* информация от клиента передается через стандартный ввод (хотя и в этом случае переменные окружения также несут информацию о запросе и т.д.). Обратная информация передается через стандартный вывод CGI-программы (как совокупность HTTP-заголовков, обязательным является заголовок *Content-type*). Сервер должен перехватить стандартный вывод, сохранить информацию и передать её как ответ клиенту. Можно было бы просто перенаправить стандартный вывод в клиентский сокет без промежуточного сохранения вывода, но однако так не поступают по двум главным причинам: во-первых, заголовок *Content-length* можно сгенерировать только после завершения вывода, и во-вторых, клиента можно обезопасить от сбоя в CGI-программе.

Таким образом, CGI-программу можно написать на любом языке программирования, который поддерживает понятия стандартного ввода и вывода и имеет доступ к переменным окружения. А это — практически любой язык программирования, реализованный в ОС UNIX. Именно из-за такой простоты и гибкости CGI-интерфейс стал первым широко применяемым средством расширения функциональности веб-серверов и создания динамических веб-сайтов. Однако достоинства этого интерфейса практически сразу превратились в его недостатки: поскольку на каждый запрос CGI-ресурса веб-сервер запускает особый процесс, то нагрузка на реальные сервера резко возрастает, т.к. накладные расходы ОС на запуск процесса могут превысить расходы на собственно генерацию содержания. В настоящее время промышленные веб-сервера поддерживают CGI-интерфейс для совместимости, однако высоко нагруженные сайты создаются с помощью других технологий (серверные страницы, включаемые модули и т.п.). Рассмотрение таких технологий находится за пределами настоящего пособия.

Рассмотрим набор переменных окружения, которые должен поддерживать (и передавать в CGI-программу) модельный веб-сервер. Обратите внимание на то, что переменные делятся на 2 группы: свойства запроса (из-

влекаются либо из свойств TCP/IP соединения, либо из заголовков запроса) и свойства сервера.

Таблица 2. Переменные окружения модельного веб-сервера

Имя переменной	Пояснение	Пример
CONTENT_TYPE	MIME-тип содержимого запроса	text/plain
GATEWAY_INTERFACE	Версия протокола CGI	CGI/1.1
REMOTE_ADDR	IP-адрес клиента	127.0.0.1
REMOTE_PORT	TCP-порт клиента	8845
QUERY_STRING	Аргументы запроса (GET)	User=Igor&mail=igor@mail.ru
SERVER_ADDR	IP-адрес сервера	127.0.0.5
SERVER_NAME	Доменное имя сервера	www.mysoft.ru
SERVER_PORT	TCP-порт сервера	9005
SERVER_PROTOCOL	Версия протокола HTTP	HTTP/1.0
SERVER_SOFTWARE	Серверное программное обеспечение	Apache/1.3.12 (Unix) PHP/3.0.17
SCRIPT_NAME	Имя CGI-программы из запроса	/cgi-bin/cgittest1
SCRIPT_FILENAME	Полный путь к файлу CGI-программы	/usr/home/igor/wwwroot/ cgi-bin/cgittest1
DOCUMENT_ROOT	Полный путь к домашней директории сервера	/usr/home/igor/wwwroot
HTTP_USER_AGENT	Имя клиентской программы	Mozilla/4.0 (compatible; MSIE 5.0; Windows 98; DigExt)
HTTP_REFERER URI	URI ресурса, с которого клиент сделал текущий запрос	http://127.0.0.1/testpage.html

## 4.2.3 Пример CGI-программы

Приведем пример простой CGI-программы, написанной на языке Си. Си — не лучший язык программирования для CGI, однако в программе на Си хорошо видны особенности CGI-программ. Наш пример выдает текст простой HTML-страницы, сообщающей аргументы запроса и IP-адрес клиентского браузера. Будем считать, что запрос имеет следующий вид:

```
GET URI-нашей-программы?user=Igor&mail=igor@mail.ru HTTP/1.1
```

...заголовки запроса...

### Пример:

```
#include <stdio.h>
#include <stdlib.h>

int main ()
{
    /* выдаем обязательный заголовок Content-type
    и пустую строку перед телом сообщения */
    printf("Content-type: text/html\n\n");

    // выдаем текст собственно страницы
    printf ( "<html><body>" );
    printf ( "Приветствую! Вы ввели аргументы: '%s' с
адреса ' %s'", getenv ( QUERY_STRING ), getenv
( REMOTE_ADDR ) );
    printf ( "</body></html>" );
    return 0;
}
```

Как видим, структура CGI-программы очень проста, однако если мы попытаемся написать более содержательный пример (например, с разбором *QUERY\_STRING* и т.п.), то недостатки языка Си как языка программирования для CGI проявятся более ярко. Именно поэтому для программирования CGI обычно используются более подходящие языки (Perl, Python, PHP и т.д.)

## 5. Методические указания по выполнению второго этапа задания

Целью второго этапа задания является добавление поддержки CGI в ранее разработанный модельный веб-сервер.

Самая простая схема реализации CGI подразумевает, что сервер запускает отдельный процесс-потомок для обработки CGI-программы. Вывод программы перенаправляется во временный файл для последующего включения этого файла в ответ на запрос. Временный файл используется, т.к. программа может завершиться некорректно (и тогда её результаты должны быть проигнорированы без передачи клиенту), а также, чтобы узнать длину ответа и сформировать заголовок *Content-length*. После корректного завершения CGI-программы сервер генерирует ответ на основе выдачи программы.

При реализации CGI особенно проявляются недостатки первой схемы реализации сервера (когда тот не приступает к обработке следующего запроса, пока не выполнит до конца текущий). Действительно, в этом случае сервер простаивает в ожидании завершения CGI-программы, которая может работать произвольно долго. Заметим, что CGI-программы, будучи внешними по отношению к серверу, обязаны рассматриваться сервером как потенциально ненадежные (т.е. аварийно завершающиеся, зацикливающиеся и т.д. и т.п.). Поэтому единственно приемлемой схемой реализации является асинхронная. Однако здесь появляется своя проблема: как одновременно и асинхронно отслеживать как события ввода-вывода, так и момент завершения CGI-программ.

Дело в том, что после запуска процесса сервер должен перейти в ожидание событий двух видов: события ввода/вывода (от операций над сокетами и файловыми дескрипторами), а также события завершения процесса-потомка (CGI-программы). Системный вызов *select()* отслеживает события только первого рода, а завершение процесса надо обрабатывать особо. Один из вариантов, который можно использовать в этом случае, выглядит так:

- При запуске процессов сервер заносит идентификатор потомка в специальную таблицу активных CGI-процессов. Если эта таблица не пуста, то сервер выполняет вызов `select()` с ненулевым тайм-аутом (выбирается какое-либо относительно небольшое значение — например, полсекунды). В результате вызов `select()` завершается либо по приходу события от операций ввода/вывода, либо по истечению тайм-аута. После обработки событий ввода/вывода сервер не переходит сразу к ожиданию, а перед этим запрашивает ОС о завершении процессов-потомков. Для этого следует использовать системный вызов

```
// вызов неблокирующий – WNOHANG
waitpid(-1, &status, WNOHANG );
```

Возможно, понадобится несколько вызовов `waitpid()` в случае, если сразу несколько CGI-программ успели завершиться за время выполнения вызова `select()`. После того, как все завершившиеся к этому моменту процессы обработаны, сервер снова переходит в режим ожидания событий ввода/вывода (т.е. вызывает `select()`).

Отдельного разговора заслуживает вопрос о последовательности обработки событий: то ли выполнять вначале все операции `accept()`, затем ввода/вывода, затем обработку завершившихся потомков, то ли наоборот (вначале потомки, потом ввод-вывод, потом `accept()`), то ли использовать очередность запросов (если запрос обрабатывается дольше всех, то его события по его обработке выполнять в первую очередь) и т.д. В общем случае, ответ на этот вопрос и выбор стратегии «балансировки нагрузки на веб-сервер» весьма нетривиален, и его обсуждение выходит за рамки данного задания.

Итак, последовательность системных вызовов для одного шага цикла асинхронной обработки может быть такой:

```
// формирование масок для вызова select()
select()
// вышли из select() по тайм-ауту, либо по наступлению
// событий
// . . .

// обработка входящих соединений
accept()
// . . .

// обработка операций ввода-вывода
read()
write()
// . . .

// обработка завершившихся CGI-программ
waitpid()
// генерация ответа клиенту
// чистка ресурсов (удаление временных файлов и т.д.)
```

Последовательность системных вызовов для обработки CGI-запроса:

```
pid = fork ();
```

```
if ( 0 < pid )
{
/* родитель-сервер – продолжаем асинхронную обработку
событий */
}
else if ( pid < 0 )
{
// катастрофическая ошибка
//...
}
else
{
/* потомок – обработка CGI-программы:
– формирование массива переменных окружения env
– перенаправление стандартного вывода во временный файл
– перенаправление стандартного ввода (для метода POST)
– собственно запуск программы */
execvpe ( script_filename, argv, env );
// обработка ошибок запуска
}
}
```

## 6. Методические указания по выполнению третьего этапа задания

На третьем этапе требуется разработать интерпретатор модельного языка и встроить его в веб-сервер, реализованный на предыдущих этапах выполнения задания.

Как уже отмечалось выше, для написания CGI-программ пригоден практически любой язык программирования, поддерживающий стандартный ввод/вывод и считывание значений переменных окружения. Однако специфика CGI-программ, а именно — ориентация на генерацию текста (точнее, текста на языке HTML), приводит к тому, что использование традиционных языков системного программирования типа Си/Си++ оказывается неудобным: соответствующие CGI-программы получаются громоздкими, трудными для понимания и сопровождения. Традиционные преимущества таких языков — компилируемость, высокая эффективность разработанных программ, возможность управления любыми системными ресурсами — перестают быть таковыми для CGI-программ. Шлюзовым программам не обязательно быть максимально эффективными, так как основное время тратится не на выполнение программы генерации HTML-текста, а на передачу его по медленным (относительно процессора) сетевым каналам связи. Доступ к системным ресурсам для программ, запуск которых инициируется извне (по HTTP-запросу от веб-клиентов), должен быть не упрощен, а наоборот ограничен. Удобство разработки шлюзовых программ является более важным требованием к соответствующей системе программирования нежели эффективность получаемого кода.

Ниже описаны три варианта модельных языков, предлагаемых для встраивания в реализованный веб-сервер. Каждый из вариантов отвечает минимальным требованиям к языкам для написания CGI-программ, а именно:

- достаточно полный набор операторов, включающий в себя присваивание, ветвление на 2 варианта (*if-then-else*), циклы (*while-do*, *do-while*, *for*), составной оператор;
- как минимум, два типа данных: целый с базовыми арифметическими операциями и строковый с операцией конкатенации;
- набор стандартных функций, включающий в себя доступ к переменным окружения, базовые функции обработки строк и символьный ввод/вывод.

Кроме того, каждый из языков должен поддерживать хотя бы один из способов встраивания в веб-сервер.

## 6.1. Способы встраивания программ на модельных языках в веб-сервер

Мы рассмотрим три способа встраивания программ:

- базовый.
- внутривстраничный тег «<? ?>».
- внутривстраничный тег *SCRIPT*.

В базовом способе программа из GET-запроса, обрабатываемая интерпретатором модельного языка — это просто текст на этом языке, генерирующий в стандартный канал вывода HTML-страницу, которая и выдается как ответ веб-сервера. Именно такой способ мы применяли в примере для языка Си (правда, без интерпретатора). Этот способ — самый простой, и он применим к любой CGI-программе. Однако он является не самым удобным для разработчика.

При внутривстраничном способе встраивания программа из GET-запроса представляет собой HTML-текст, в который встроены фрагменты CGI-программы. Интерпретатор такой программы должен различать HTML-текст и код CGI-программы. HTML-текст просто выводится в стандартный канал, а фрагменты программы выполняются сразу после их считывания из текста файла. В этом случае надо писать фрагменты, генерирующие только изменяющиеся части страницы, а постоянные части можно сразу оформлять как HTML-текст. В результате общий объем CGI-программы существенно уменьшается (как и усилия по ее написанию и пониманию).

Разумеется, фрагменты CGI-программы должны четко выделяться синтаксически, а соответствующий интерпретатор должен уметь различать HTML-текст и собственно программу. Удобно включать фрагменты программы внутри специальных тегов. При втором способе встраивания используется тег «<? ?>», который не входит в HTML. Части программы размещаются между открывающим «<?» и закрывающим «?>» тегами.

Например, текст на языке PHP, выводящий тот же текст, что и в предыдущем примере на Си, выглядит так:

```
Content-type: text/html
<html><body>
Приветствую! Вы ввели аргументы:
  <? echo "$QUERY_STRING с адреса $REMOTE_ADDR" ?>
</body></html>
```

Здесь фрагмент программы на PHP содержит оператор вывода *echo*. Все остальное — это просто HTML-текст.

При третьем способе используется стандартный HTML-тег *SCRIPT*. Этот тег используется для встраивания исполняемых программ в любые HTML-страницы, причем выполняться эти программы могут как сервером (для генерации содержимого страницы), так и клиентом-браузером. Нас будет интересовать только первый вариант.

Тег *SCRIPT* имеет следующий вид:

```
<script language="lang_name" runat="server">
текст программы
</script>
```

Атрибут *LANGUAGE* содержит имя языка (в общем случае могут поддерживаться разные языки) — по умолчанию это JavaScript. Атрибут *RUNAT="server"* сообщает интерпретатору о том, что соответствующий фрагмент должен выполняться на сервере, а не на клиенте. Заметим, что по умолчанию этот атрибут соответствует клиенту, поэтому при отсутствии атрибута *RUNAT* интерпретатор обязан выводить весь тег *SCRIPT* в том виде, как он есть, в стандартный вывод (для последующей обработки клиентом).

Приведем пример программы на модельном JavaScript, который генерирует ту же самую страницу, что и в предыдущих примерах:

```
<html><body>
Приветствую! Вы ввели аргументы:
<script language="lang_name" runat="server">
  Response.write( "Приветствую! Вы ввели аргументы: " +
  Environment["QUERY_STRING"] + "с адреса " +
  Environment["REMOTE_ADDR"]);
</script>
<body></html>
```

Также как и в примере на PHP, фрагмент программы на JavaScript содержит обращение к функции вывода (только она называется *write* и является методом объекта *Response*). Весь остальной текст — это HTML.

Разумеется, внутривстраничные методы встраивания доставляют дополнительные «заботы» интерпретатору: ведь он должен анализировать не только текст на «своем» языке, но и HTML-текст. При этом «родной» текст должен быть проанализирован, переведен в промежуточное представление (например, разновидность ПОЛИЗ [5]) и выполнен (интерпретирован). В свою очередь, HTML-текст должен быть просто перенаправлен на стандартный вывод, поэтому требования к анализу и обработке HTML-текста существенно ниже. Интерпретатор должен вычленил текст программы и выполнить его, весь остальной текст — перенаправить. Обработка возможных ошибок в HTML, проверка корректности и возможности отображения HTML-текста — задача других инструментов.

## 6.2. Описание модельных языков

Для описания модельных языков используются следующие соглашения и обозначения (традиционные для расширенной БНФ):

- запись вида  $\{a\}$  означает итерацию цепочки  $a$ , т.е. цепочки вида:  $\varepsilon$ ,  $a$ ,  $aa$ ,  $aaa$  и т.д.;

- запись вида  $[\alpha]$  означает:  $\alpha$  или «пусто» ( $\epsilon$ );
- для отличия метасимволов БНФ (фигурных и квадратных скобок, а также вертикальной черты) от терминальных символов модельных языков они выделены **красным** шрифтом (например, **{**, **[**, **|**);
- служебные (ключевые) слова модельных языков выделены жирным шрифтом (*for*)

## 6.2.3 Модельный язык программирования

В качестве первого варианта языка предлагается использовать модельный язык программирования из задания практикума, разработанного Т. В. Руденко [6]. Единственное (и обязательное) расширение этого языка, необходимое для использования в CGI-технологии, это добавление специальных переменных для доступа к переменным окружения. Эти переменные имеют вид:  $\$Name$ , где идентификатор  $Name$  — это имя переменной окружения (например,  $\$QUERY\_STRING$ ). Все переменные окружения имеют тип *string*. Они могут появляться везде в выражениях строкового типа, но их значение не может быть изменено. Заметим, что в задании не определен синтаксис понятия идентификатор. Однако для наших целей имя (идентификатор) переменной окружения должно быть совместимо с требованиями ОС UNIX, поэтому будем требовать, чтобы в идентификатор (по крайней мере для имен переменных окружения) могли входить латинские буквы, цифры и символ подчеркивания « $\_$ ». Регистр букв имеет значение.

Модельный язык должен поддерживать единственный способ встраивания — внешний (внутристраничные не используются).

## 6.2.4 Модельный JavaScript (MJS)

Второй вариант модельного языка основан на языке JavaScript [7].

### 6.2.4.1. Переменные

Мы ссылаемся на переменные с помощью имен (идентификаторов). Имя — это последовательность латинских букв, цифр и знаков подчеркивания (« $\_$ »). Регистр букв имеет значение ( $A$  и  $a$  — это разные идентификаторы). Имя не может совпадать ни с одним из служебных слов (регистр букв в служебных словах также имеет значение — он всегда нижний).

Важнейшей особенностью языка MJS является динамическая типизация переменных, что означает, что переменная может иметь значение любого типа. Тип значения определяется при присваивании или инициализации переменной и может измениться при последующих присваиваниях. Поэтому необходимость в явном объявлении переменных отпадает, и в «родном» языке JavaScript переменные можно не объявлять. Не объявленная явно переменная начинает существовать с момента первого присвоения ей значения. Однако такая практика ухудшает как эффективность, так и надежность программного кода, поэтому в модельном варианте переменные необходимо явно объявить до первого использования с помощью служебного слова *var*, например:

```
var x = "string value";
var y;
```

Такое объявление (а точнее — определение) может появиться везде, где может появиться оператор (оно и является частным случаем оператора).

С понятием переменной связано понятие «область действия». Если переменная объявлена на верхнем уровне программы (т.е. непосредственно внутри какого-либо тега *SCRIPT*), то она является глобальной и ее область действия — вся программа (начиная с точки объявления). В противном случае переменная локальна и ее область действия — блок, где появилось ее объявление. После выхода из блока локальные переменные перестают существовать.

### 6.2.4.2. Простые типы данных и операции

В JJS есть 3 простых типа данных: строковый (*String*), числовой (*Number*) и логический (*Boolean*).

Константы строкового типа данных содержат любые символы, заключенные в двойные кавычки ("Пример строки"), либо в одинарные кавычки ('Еще один пример строки'). Если в тексте константы нужно указать кавычку, то она предваряется экранирующим символом «\» ("Пример строки с кавычками:\", \' "). Если в тексте нужно указать экранирующийся символ, то он дублируется («\|»).

Операции над строковым типом включают в себя конкатенацию (+) и шесть операций сравнения (==, !=, <, >, <=, >=). Используется обычное лексикографическое сравнение строк.

Числовой тип данных содержит как целые, так и вещественные значения. Для записи целочисленных констант используется десятичное представление (124, -13 и т.д.). Для записи вещественных констант используется либо представление с дробной частью, отделенной точкой (12.345), либо представление в виде «мантисса-порядок» (1.2E2, -0.5e-3). Используется десятичное представление, порядок отделяется латинской буквой E (e).

Операции над числовым типом включают в себя арифметические, операции сравнения, инкрементные и декрементные.

Арифметические операции традиционны: +, -, \*, /, а также остаток от деления (%). Также традиционны и операции сравнения (==, !=, <, >, <=, >=). Инкрементные и декрементные операции (++ и --) делятся на префиксные и постфиксные, как в Си. Их семантика совпадает с семантикой Си.

Логический тип данных содержит две константы (*true* и *false*). К нему применимы логические операции («и» — &&, «или» — ||, «отрицание» — !) и операции сравнения (==, !=, <, >, <=, >=).

### 6.2.4.3. Составные типы данных

JJS содержит два составных типа: массив (*Array*) и объект (*Object*).

Массив — это последовательность анонимных переменных. К каждой переменной из этой последовательности можно обратиться по номеру (индексу):

```
response.write ( arr[i] );
```

Индекс в массиве начинается с нуля.

Значения элементов массива могут быть заданы с помощью конструктора массива:

```
var a = [1,2,"string value", false];
```

Инициализатор может быть и пустым:  $b = []$ ;

Заметим, что элементы массива могут иметь разные типы (что вытекает из динамической типизации переменных).

Важнейшей особенностью массивов в MJS является их «динамичность»: размер массива может меняться во время работы программы. Добавить элемент в массив можно с помощью операции присваивания: если индекс для присваиваемой переменной больше числа элементов в массиве, то массив «расширяется», чтобы включить в себя новый элемент.

```
var a = [0,1,2,3];  
a[4] = 4;
```

При этом в массиве могут появиться дополнительные элементы (с неопределенным значением):

```
a[20] = 20;
```

Рассмотрим теперь объектный тип данных. Объект характеризуется набором именованных свойств. Доступ к свойству обеспечивает операция «.» (точка). У нее два аргумента — объект и идентификатор свойства: *obj.id*.

Свойства делятся на свойства-данные и свойства-методы. Последние являются функциями, и их можно вызывать как функции, например:

```
response.write ( "<tag/>" );
```

Заметим, что с точки зрения интерпретатора операция доступа к свойству имеет первым операндом ссылку на объект, а вторым операндом — идентификатор (фактически — строку). Поэтому нет ничего странного в том, что объект имеет еще одну форму операции доступа к свойству: операцию индексирования. Эта операция имеет первым аргументом ссылку на объект, а вторым — строку. Например, оператор:

```
x = obj.id;
```

полностью эквивалентен оператору:

```
x = obj["id"];
```

Заметим, что в отличие от операции «.» второй аргумент может быть вычисляемым, например, следующий фрагмент также присваивает переменной *x* значение свойства *id* объекта *obj*:

```
var id = "i";
x = obj[id + "d"];
```

Таким образом, объект становится похожим на массив, только объектная операция индексирования имеет аргументом не число, а строку. На самом деле, объект в MJS имеет и вторую форму операции индексирования: *obj[i]*, где *i* — целочисленный индекс (от 0 до *N* − 1, где *N* — свойств объекта). Правда, информация о том, какой именно индекс соответствует свойству, недоступна пользователю и зависит от реализации. Поэтому вторая форма операции индексирования используется крайне редко, тем не менее она существует.

Встает резонный вопрос: если объект фактически обобщает функциональность массива, а массив является динамически расширяемым объектом, то почему бы не позволить набору свойств объекта также быть расширяемым? На самом деле, в языке JavaScript принята именно такая интерпретация операции индексирования, но в модельном варианте для простоты это не реализовано. Таким образом, пользователь в MJS не может изменять набор свойств объекта.

Более того, пользователь в MJS не может создавать свои объекты (как это можно делать в JavaScript), и набор объектов ограничен только встроенными объектами.

#### 6.2.4.4. Встроенные объекты MJS

Объект *Response* представляет собой обертку над выводом CGI-программы. Он имеет единственный метод — *write(x)*, который выводит аргумент *x* в стандартный вывод.

**Пример:**

```
Response.write ( "<body>" );
```

Объект *Environment* — это набор переменных окружения. Каждое свойство объекта (доступное только для чтения) — это соответствующая переменная окружения, переданного процессу-интерпретатору.

**Пример:**

```
x = Environment [ "QUERY_STRING" ];
var y = Environment.REMOTE_ADDR;
```

Важными представителями встроенных объектов являются объекты-типы. Каждому типу MJS соответствует встроенный одноименный объект-тип.

Отметим, что значения в языке MJS являются копиями («клонами») соответствующих объектов-типов, поэтому свойства и методы этих объектов являются одновременно свойствами и методами значений.

#### 6.2.4.4.1. Boolean

Объект *Boolean* имеет единственный метод *toString()*, возвращающий «*true*» или «*false*».

#### 6.2.4.4.2. Array

Объект *Array* имеет свойство *length* (количество элементов) и три метода:

- *sort()* — сортирует элементы по возрастанию;
- *reverse()* — переставляет элементы в обратном порядке;
- *toString()* — возвращает строку, содержащую список значений элементов, разделенных запятыми.

#### 6.2.4.4.3. Number

Объект *Number* имеет три свойства:

- *MAX\_VALUE* — максимальное значение числа;
- *MIN\_VALUE* — минимальное значение числа;
- *NaN* — это специальное значение, которое вырабатывается при ошибках вычислений или преобразований.

Методы объекта *Number*:

- *toString([base])* — возвращает строку, представляющую значение числа в системе счисления с основанием *base* (по умолчанию — 10);
- *toFixed(n)* — возвращает строку, представляющую значение числа в форме "целая\_часть.дробная\_часть". Длина дробной части — *n* цифр;
- *toExponential(n)* — возвращает строку, представляющую значение числа в форме "мантисса E порядок". Мантисса нормализована, длина дробной части мантиссы — *n* цифр;

#### 6.2.4.4.4. String

Объект *String* имеет свойство *length* (количество символов в строке) и методы (позиции символов нумеруются с нуля):

- *charAt(позиция)* — возвращает строку, состоящую из одного символа, стоящего в указанной позиции в строке (или пустую строку, если позиция неверна).
- *indexOf(строка)* — возвращает номер позиции, с которой начинается первое слева вхождение подстроки строка (или  $-1$ , если вхождения нет)
- *lastIndexOf(строка)* — возвращает номер позиции, с которой начинается первое справа вхождение подстроки строка (или  $-1$ , если вхождения нет)

- *substr* (*смапм* [, *длина*]) — возвращает подстроку, начинающуюся с позиции *старт*, и содержащую *длина* символов. Если *длина* не задана или превышает возможную длину, то подстрока содержит символы до конца строки.

### 6.2.4.5. Универсальные значения и операции

В языке MJS существуют специальные значения, которые имеют универсальный характер. Это неопределенное значение *undefined* и пустое значение *null*. Неопределенное значение получает любая неинициализированная переменная. Пустое значение удобно для индикации отсутствия значения (объекта).

Также существуют универсальные операции, применимые к любым типам данных. Главная такая операция — это присваивание:

```
v = e
```

Левая часть операции — переменная или свойство объекта (доступное для записи), правая часть — выражение. Значение левой части присваивается правой части, а также служит результатом операции присваивания. Операция присваивания — правоассоциативна.

Кроме обычной операции присваивания в MJS по аналогии с языком Си++ существуют варианты операции присваивания для ряда бинарных инфиксных операций. В языке MJS это операции `+=`, `-=`, `*=`, `/=`, `*=`, `%=`. Их семантика совпадает с семантикой операций Си++.

Другая универсальная операция — это проверка на строгую тождественность (`===`). Она истинна, только если объекты имеют один и тот же тип и значение. Обычная операция сравнения сравнивает только значения, а если типы не совпадают, то происходит преобразование типов (см. ниже) к одному, а затем только проверка. Поэтому сравнение `1===1` ложно, а сравнение `1=="1"` истинно.

Операция проверки на нетождественность (`!==`) является отрицанием операции (`===`).

Последняя универсальная операция MJS — это операция определения типа *typeof*. Она возвращает строку, идентифицирующую тип аргумента. Конкретные значения определяются таблицей 3.

Таблица 3. Значения, возвращаемые операцией `typeof`

Тип	Значение
Неопределенный ( <i>undefined</i> )	<i>undefined</i>
Пустой ( <i>null</i> )	<i>Object</i>
Логический	<i>Boolean</i>
Числовой	<i>Number</i>
Строковый	<i>String</i>

Тип	Значение
Объектный	Object

### 6.2.4.6. Преобразования типов

Важной особенностью языка JavaScript является широкое использование неявных преобразований типов. Фактически интерпретатор JavaScript пытается выполнить операцию практически для любой комбинации типов операндов. Например, если операция требует числовых операндов (пример:  $a - b$ ), то оба операнда (если нужно) преобразуются к типу *Number*. Примечательным исключением является операция  $+$ . Она может применяться как к числовым, так и к строковым данным. При этом строковая операция имеет приоритет, поэтому если один из операндов — строковый, то другой преобразуется в строку. Поэтому оба оператора ниже дадут одно и то же строковое значение «1234»:

```
x = 12 + "34";
y = "12" + 34;
```

Следующие ниже таблицы описывают правила преобразования типов.

Таблица 4. Преобразования к логическому типу

Тип	Значение
Неопределенный (undefined)	false
Пустой (null)	false
Числовой	false, если значение = 0, иначе true
Строковый	false, если длина строки = 0, иначе true
Другие объекты	true

Таблица 5. Преобразования к числовому типу

Тип	Значение
Неопределенный (undefined)	Number.NaN
Пустой (null)	0
Логический	0, если значение = false, иначе 1
Строковый	Если строка представляет собой числовую константу, то соответствующее значение. Иначе — Number.NaN
Другие объекты	Number.NaN

Таблица 6. Преобразования к строковому типу

Тип	Значение
Неопределенный ( <b>undefined</b> )	<b>undefined</b>
Пустой ( <b>null</b> )	<b>null</b>
Логический	<b>true</b> или <b>false</b>
Строковый	<b>NaN</b> или строка, представляющая числовое значение
Другие объекты	значение, возвращаемое методом <b>toString()</b>

### 6.2.4.7. Синтаксис MJS

*⟨предложение⟩ ::= ⟨определение-функции⟩ | ⟨оператор⟩*  
*⟨определение-функции⟩ ::= **function** ⟨имя⟩ ([⟨имя⟩] {,⟨имя⟩}) ⟨блок⟩*  
*⟨оператор⟩ ::= ⟨объявление-переменной⟩ | ⟨пустой-оператор⟩ | ⟨блок⟩ |*  
*⟨условный-оператор⟩ | ⟨оператор-цикла⟩ |*  
*⟨оператор-перехода⟩ | ⟨оператор-выражение⟩*  
*⟨объявление-переменной⟩ ::=*  
***var** ⟨имя⟩ [ = ⟨выражение⟩ ] {, ⟨имя⟩ [ = ⟨выражение⟩ ] };*  
*⟨пустой-оператор⟩ ::= ;*  
*⟨блок⟩ ::= { ⟨оператор⟩ { ⟨оператор⟩ } }*  
*⟨условный-оператор⟩ ::= **if** (⟨выражение⟩) ⟨оператор⟩ [ **else** ⟨оператор⟩ ]*  
*⟨оператор-цикла⟩ ::= **while** (⟨выражение⟩) ⟨оператор⟩ |*  
***for** ([⟨выражение⟩];[⟨выражение⟩];[⟨выражение⟩]) ⟨оператор⟩ |*  
***do** ⟨оператор⟩ **while** (⟨выражение⟩); |*  
***for** ([**var**] ⟨имя⟩ **in** ⟨выражение⟩) ⟨оператор⟩*  
*⟨оператор-перехода⟩ ::= **break**; | **continue**; | **return** [⟨выражение⟩];*  
*⟨оператор-выражение⟩ ::= ⟨выражение⟩;*

Большинство операторов языка MJS соответствует операторам языка Си++. Исключение составляет цикл **for-in**. Его аналога в языке Си++ нет. Этот цикл служит для последовательного просмотра элементов в массиве или свойств в объекте. Переменная цикла получает на каждой итерации значение очередного элемента массива или свойства объекта (вспомним, что в MJS существует глубокая связь между массивами и объектами). Если переменная объявлена (через **var**) в цикл, то ее область действия — только оператор цикла, и после выхода из него переменная не существует.

Следующий оператор цикла суммирует все элементы массива *arr*:

```
var s = 0;  
for ( var e in arr ) s += e;
```

Операторы *break* и *continue* могут появляться только внутри тела цикла (произвольного). Оператор *return* может появляться только внутри тела функции. Если *return* не возвращает выражение, то функция является процедурой (в терминах языка Паскаль). Однако, если контекст вызова такой функции требует возврата значения (например,  $x = f()$ ), то это значение равно *Undefined*. Таким образом можно считать, что *return* всегда возвращает значение, по умолчанию равное *Undefined*.

## 7. Список литературы

1. Б. Страуструп. Язык программирования C++. Специальное издание. — М.; СПб.: «Издательство БИНОМ»–«Невский Диалект», 2001 г.
2. RFC3986. <http://www.iana.org/assignments/uri-schemes.html>
3. Н. В. Вдовикина, И.В.Машечкин, А.Н.Терехин, А.Н.Томилин. Операционные системы: взаимодействие процессов. — М.: МАКС Пресс, 2008 г.
4. У. Р. Стивенс. UNIX: разработка сетевых приложений. — СПб.: Питер, 2004.
5. И. А. Волкова, А. А. Вылиток, Т. В. Руденко. Формальные грамматики и языки. Элементы теории трансляции. — М.: МАКС Пресс, 2009 г.
6. Т. В. Руденко. Интерпретатор модельного языка. Задание практикума (<http://cmcmsu.no-ip.info/download/model.lang.practical.task.pdf>).
7. Т. Пауэлл, Ф. Шнайдер. Полный справочник по JavaScript. — М.: Издательский дом «Вильямс», 2006 г.