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

А. В. Столяров

Практикум на ЭВМ

Многопользовательский игровой сервер

Москва
2005

Автор будет признателен за конструктивную критику, в том числе за сообщения об обнаруженных в тексте пособия опечатках.

Адрес для связи: avst@cs.msu.su.

Черновая версия от 24 февраля 2005 г.

Введение

Задание практикума “многопользовательский игровой сервер” разработано для занятий практикума на ЭВМ, проводимых на втором курсе на факультете ВМиК МГУ в рамках основного учебного плана.

Задание предназначено для выполнения в операционной системе семейства Unix (например, FreeBSD или Linux) с использованием языков программирования C и C++.

Задание состоит из двух основных частей, каждая их которых выполняется в несколько этапов. В первой части задания предлагается реализовать программу-сервер, выполняющую роль ведущего в игре “Менеджер” [1] и позволяющую принимать участие в игре игрокам, находящимся на разных машинах, используя локальную сеть. Первая часть задания может выполняться на языке C, что позволяет её выполнить в осеннем семестре.

Вторая часть, предназначенная для выполнения на языке C++, состоит в создании программируемых роботов, способных принимать участие в игре “Менеджер”, имитируя действия игроков-людей. Такой робот представляет собой интерпретатор некоторого достаточно простого языка программирования, с помощью которого и задаётся его поведение в игре.

Задание практикума “многопользовательский игровой сервер” нацелено на выработку и закрепление следующих навыков:

- Первая часть (игровой сервер)
  - использование системы программирования (редакторов текстов, компилятора, отладчика, системы автоматической сборки) в ОС UNIX
  - программирование на языке C
  - создание сетевых приложений, использующих протокол TCP при помощи berkley sockets
  - использование мультиплексирования ввода-вывода для создания событийно-управляемых приложений

- Вторая часть (программируемые роботы)
  - объектно-ориентированное проектирование и программирование
  - программирование на языке C++
  - использование элементов теории формальных грамматик для разбора текстов на формальных языках (лексический анализ по регулярным грамматикам с помощью конечных автоматов, синтаксический анализ методом рекурсивного спуска, использование польской инверсной записи в качестве представления программы для осуществления интерпретации).

В настоящем пособии автор постарался ответить на наиболее типичные вопросы, возникающие у студентов по ходу работы над заданием.

1 Игра “Менеджмент”

Игра “Менеджмент” была предложена фирмой Avalon Hill Company для обучения основам управления предприятием. Ч. Уэзерелл отметил чрезвычайную привлекательность этой игры в качестве упражнения (этюда) для программистов.

В этом разделе излагаются сокращенные (упрощенные) правила игры “Менеджмент”. Полные правила игры желающие могут найти в книге [1].

1.1 Общие сведения

В игре участвуют N игроков. Каждый игрок имеет номер $1 \leq k \leq N$. Каждый игрок с номером k располагает некоторым количеством денег (условных долларов), S_k единицами сырья, P_k единицами продукции и F_k фабриками. В начале игры каждому игроку выдается 2 фабрики, 4 единицы сырья, 2 единицы готовой продукции и 10000 долларов.

Моделирование ведется пошагово, игровыми циклами. Цикл представляет собой условный игровой месяц.

1.2 Порядок игры

В каждом “месяце” игроки производят на своих фабриках продукцию из сырья. Одна фабрика может произвести одну единицу продукции, израсходовав при этом одну единицу сырья и \$2000.

В случае, если у игрока недостаточно сырья или денег, чтобы обеспечить работу всех фабрик, либо если в связи с неблагоприятной обстановкой на рынке у игрока скопилось слишком много готовой продукции, фабрика может ничего не производить (что не исключает ежемесячных издержек).

Каждый “месяц” банк проводит аукционы по продаже сырья и скупке продукции. Аукционы проводятся в соответствии с “обстановкой на рынке”, описываемой ниже. Заявки на аукционы подаются игроками “в темную”, т.е. игроки ничего не знают о заявках, подаваемых другими игроками. Однако по окончании аукциона банк сообщает всем игрокам полную информацию о результатах торгов (а именно, кому, сколько и по какой цене продано сырья, а также у кого, сколько и по какой цене куплено продукции).

В каждом “месяце” игрок может сделать заявку на строительство новых фабрик. Фабрика стоит $5000 и начинает давать продукцию на 5й “месяц” после начала строительства. Половина стоимости строительства или реконструкции списывается с игрока при подаче заявки, вторая половина – за месяц до окончания строительства или реконструкции.

Закончив подачу заявок на данный месяц, игрок заявляет об окончании хода. Когда все игроки заявили об окончании хода, игровой месяц завершается.

По итогам месяца с каждого игрока списываются ежемесячные издержки, а именно: $300 за оставшуюся на складе единицу сырья, $500 - за оставшуюся на складе единицу продукции, $1000 за фабрику (независимо от того, производила она в этом месяце продукцию или нет).

Игрок, которому не хватило денег на покрытие издержек, объявляется банкротом и выбывает из игры.

Каждый игрок в любой момент может узнать о количестве денег, фабрик, единиц сырья и продукции у остальных игроков.

1.3 Обстановка на рынке

Обстановка на рынке может находиться на одном из пяти уровней. В зависимости от уровня определяются предложение сырья (т.е. сколько единиц сырья банк продаст в этом “месяце”), спрос на продукцию (т.е. сколько единиц продукции банк купит в этом “месяце”), минимальную цену единицы сырья и максимальную цену единицы продукции. Значения этих величин определяются по таблице уровней состояния рынка (табл. 1).

Уровень	Сырье		Продукция	
	кол-во	min. цена	кол-во	max.цена
1	1.0*P	$800	3.0*P	$6500
2	1.5*P	650	2.5*P	6000
3	2.0*P	500	2.0*P	5500
4	2.5*P	400	1.5*P	5000
5	3.0*P	300	1.0*P	4500

P - общее количество необанкротившихся игроков.

Таблица 1: Уровни состояния рынка

Округление производится в сторону уменьшения, т.е., например, если в игре участвуют 3 “живых” игрока, а уровень рынка определен как 2й, то

количество продаваемого сырья будет 4 единицы, а покупаемой продукции - 7 единиц.

В начале игры уровень равен 3. Уровень для каждого следующего месяца определяется из предыдущего случайным образом в соответствии с таблицей вероятностей перехода (табл. 2).

Старый уровень	Новый уровень				
	1	2	3	4	5
1	1/3	1/3	1/6	1/12	1/12
2	1/4	1/3	1/4	1/12	1/12
3	1/12	1/4	1/3	1/4	1/12
4	1/12	1/12	1/4	1/3	1/4
5	1/12	1/12	1/6	1/3	1/3

Таблица 2: Вероятности смены уровня состояния рынка

1.4 Проведение аукционов

На каждом цикле игрок может в произвольном порядке дать заявку на участие в аукционе сырья, в аукционе продукции, заявку на производство, заявку на строительство новой фабрики и заявить об окончании своих действий на этот месяц. Все заявки подаются “в темную”, то есть они не видны другим игрокам. В заявке на участие в аукционе указывается число единиц для покупки или продажи и цена. Цена покупки сырья не должна быть ниже минимальной установленной для текущего месяца, цена продажи продукции - не выше максимальной. Заявка на производство не должна превышать количество имеющегося у игрока сырья, т.е. сырье, купленное на данном цикле, не может быть использовано при производстве продукции на этом же цикле.

Если сумма заявок превышает доступное количество единиц, выставленных на аукцион, банк в первую очередь удовлетворяет наиболее выгодные для него заявки, т.е. продает сырье игрокам, заявившим наибольшие цены, и покупает продукцию у игроков, установивших наименьшие цены. При прочих равных предпочтение отдается случайным образом (по жребию). Если размер очередной выбранной заявки превышает оставшееся количество доступных единиц, банк удовлетворяет заявку частично.

Например, если банк должен купить всего 6 единиц продукции, при этом игрок №1 выставил на продажу 3 единицы по цене 4500, игроки №2 и №3 выставили каждый по две единицы продукции по цене 5000, то банк

купит все 3 единицы у игрока №1, после чего жребий определит, какая из оставшихся заявок будет удовлетворена полностью. Соответственно, игрок, на которого падет жребий, продаст обе единицы продукции по цене 5000, а второй игрок продаст только одну из двух единиц.

Проведение аукциона можно начать в тот момент, когда получены заявки на данный аукцион от всех активных (необанкротившихся) игроков. Также можно проводить оба аукциона (продажи сырья и скупки продукции) в конце хода, т.е. непосредственно после того, как последний из игроков заявит об окончании действий а данном месяце. Практика показывает, что второй вариант проще реализовывать.

Если игрок заявил об окончании действий на данном цикле, не подав заявки на аукцион, его заявка считается нулевой.

Результаты аукционов (т.е. кому, сколько и по какой цене продано сырья, у кого, сколько и по какой цене куплено продукции) банк объявляет публично, то есть информация об этом доступна всем игрокам.

2 Реализация серверно-сетевой части

2.1 Постановка задачи

Программа-сервер выполняет функции ведущего игры. Игроки подключаются к серверу по сети с использованием протокола TCP/IP.

Возможны два подхода к организации протокола обмена прикладного уровня:

— Сервер ожидает от клиента команды в текстовом виде, предназначенном непосредственно для обработки человеком. В этом случае в качестве клиентской программы используется стандартная утилита telnet, входящая в базовую комплектацию практически любой Unix-системы.

— Сервер обрабатывает команды в определенном двоичном формате, удобном для обработки в программе. В этом случае необходимо реализовать клиентскую программу, ведущую диалог с пользователем и формирующую соответствующие данные для сервера, а также преобразующую получаемые от сервера ответы в приемлемую для пользователя форму.

Стартовыми параметрами сервера являются число игроков и номер TCP-порта, на котором программа должна ожидать запросы на соединение от клиентов. Стартовые параметры задаются в командной строке сервера. По согласованию с преподавателем возможны другие варианты получения стартовых параметров, например, из конфигурационного файла или из переменных окружения. Задавать стартовые параметры в тексте программы (т.е. так, что их изменение потребует перекомпиляции программы) *запрещается*.

После запуска программы-сервера она должна открыть сокет в режиме ожидания запросов на соединение (см. §2.2) на заданном порту, дождаться подключения заданного количества игроков, после чего перейти в режим игры, в котором и оставаться до момента, когда все игроки, кроме одного, по тем или иным причинам не выйдут из игры.

До тех пор, пока сервер не перешел в режим игры, на любую команду игрока он должен реагировать сообщением о том, что игра не началась; желательно также выдавать при этом информацию о том, сколько игроков в настоящее время уже вошли на сервер и сколько еще ожидается до начала игры.

После перехода в режим игры при попытке нового игрока подключиться к серверу он должен получить сообщение о том, что игра уже идет, после чего сервер должен разорвать соединение.

Требования к серверу:

— Сервер должен предоставлять игроку все возможности, предусмотренные правилами игры (см. §1), и проводить игру в соответствии с правилами.
— При проведении жеребьевок сервер не должен давать преимуществ никому из игроков.
— Сервер должен быть защищен от неправильных действий игроков, т.е. никакие действия одного игрока не должны нарушать ход игры.
— Сервер должен обеспечивать реальный многопользовательский режим работы, т.е. никакие действия игрока не должны приводить, даже кратковременно, к невозможности для остальных игроков вводить команды и получать результаты, если только это не предусмотрено правилами игры.
— Сервер не должен использовать активное ожидание. В частности, это означает, что в отсутствие активности игроков сервер не должен создавать никакой нагрузки на процессор. См. §2.3.
— Сервер обязан корректно обрабатывать потерю соединения с любым из игроков. Игрок, соединение с которым оборвалось, считается выбывшим из игры, о чем сообщается остальным игрокам.

Начать программирование следует с написания действующей модели сервера, в которой вместо игры фигурирует простейшая информационная сущность – глобальная целочисленная переменная, которую может изменить каждый из клиентов. В этой главе приведены все необходимые для этого сведения.

2.2 Организация TCP-сервера

Все сетевые взаимодействия в операционных системах семейства Unix организованы с помощью так называемых *сокетов* (sockets). С каждым сокетом связывается файловый дескриптор, позволяющий ссылаться на сокет при выполнении операций с ним.

При организации многопользовательского сервера понадобится два вида сокетов. Первый из них – *слушающий сокет* (listening socket) будет использоваться для ожидания и приема клиентских соединений. Сокеты другого вида представляют собой непосредственно “канал связи” с конкретным клиентом и используются для приема команд от клиентов и передачи клиентам сообщений сервера.

2.2.1 Создание сокета

Прежде всего, сокет (как объект ядра операционной системы) необходимо *создать* вызовом `socket()`:

```
#include <sys/types.h>
#include <sys/socket.h>

int socket(int domain, int type, int protocol);
```

где `domain` задает семейство адресации (address family), `type` задает тип коммуникации, `protocol` - конкретный протокол.

В рассматриваемой задаче необходим сокет, работающий в семействе IP-адресов[1] (задается константой `AF_INET`). Это означает, что адрес сокета будет состоять из *IP-адреса* и *номера порта*. IP-адрес состоит из четырех чисел от 0 до 255, обычно записываемых через точку, например: `192.168.15.131`. Номер порта - это целое число в диапазоне от 1 до 65535. **Следует учитывать, что в большинстве операционных систем порты с номерами от 1 до 1023 считаются привилегированными; это означает, что использовать их могут только процессы, обладающие правами суперпользователя.**

Среди существующих типов коммуникации для рассматриваемой задачи наиболее подходит так называемый *потоковый* (stream), задаваемый константой `SOCK_STREAM`. Сокеты этого типа коммуникации представляют собой двунаправленный канал, доступный на обоих концах как на запись, так и на чтение, в том числе и с помощью обычных вызовов `read()` и `write()`.

Наконец, в качестве параметра `protocol` можно указать 0, в результате чего система автоматически выберет единственный возможный для данной комбинации семейства адресов и типа сокета протокол TCP (иначе задаваемый константой `IPPROTO_TCP`).

Таким образом, окончательно вызов будет выглядеть так:

```
ls = socket(AF_INET, SOCK_STREAM, 0);
```

где ls - имя переменной типа int, которой будет присвоен номер файлового дескриптора, ассоциированного с вновьсозданным сокетом.

Полученный файловый дескриптор должен быть неотрицательным числом. Если вызов `socket()` вернул значение -1, это свидетельствует о происшедшей ошибке. Программа **обязательно** должна корректно обрабатывать такую ситуацию.

[1] Здесь и далее имеются в виду IP-адреса семейства протоколов IPv4.

2.2.2 Связывание сокета с адресом

Следующим шагом развертывания сервера является сопоставление созданному сокету конкретного адреса. Напомним, что в избранном нами семействе адресов (`AF_INET`) адресом является пара "IP-адрес + порт".

Компьютер, оснащенный стеком протоколов TCP/IP, может иметь произвольное количество ip-адресов. В частности, любой компьютер имеет адрес `127.0.0.1`, означающий сам этот компьютер и доступный только программам, работающим на этом же компьютере. При подключении к локальной сети компьютер также получает *интерфейсный* адрес для работы в этой сети.

Сервер может принимать соединения по заданному номеру порта на одном из IP-адресов, имеющихся в системе, либо на всех IP-адресах сразу. Последнее задается IP-адресом `0.0.0.0`, имеющим специальное значение и обозначающимся также константой `INADDR_ANY`.

Связывание сокета с конкретным адресом производится вызовом `bind()`:

```
#include <sys/types.h>
#include <sys/socket.h>

int bind(int sockfd, struct sockaddr *addr, int addrlen);
```

где `sockfd` – дескриптор сокета, полученный в результате выполнения вызова .socket().; `addr` – указатель на структуру, содержащую адрес; наконец, `addrlen` – размер структуры адреса в байтах.

Реально в качестве параметра `addr` используется не структура типа `sockaddr`, а структура другого типа, который зависит от используемого семейства адресации (см. §2.2.1). В избранном нами семействе `AF_INET` используется структура `struct sockaddr_in`, умеющая хранить пару "IP-адрес + порт". Эта структура имеет следующие поля:

- `sin_family` – обозначает семейство адресации (в данном случае значение этого поля должно быть установлено в `AF_INET`).
- `sin_port` – задает номер порта в *сетевом порядке байт*, который, вообще говоря, может отличаться от порядка байт, используемого на данной машине. Соответственно, значение для занесения в это поле должно быть получено из выбранного номера порта вызовом функции `htons()`[2]. Напомним, что номер порта задается как параметр командной строки программы-сервера.

[2]Название функции htons() получено как сокращение от *Host to Network Short*, т.е. преобразование из хостового в сетевой порядок байт для короткого целого. Более подробно понятие сетевого порядка байт будет рассмотрено в §2.6.1

— `sin_addr` – задает IP-адрес. Поле `sin_addr` само является структурой, имеющей лишь одно поле с именем `s_addr`, которое хранит IP-адрес в виде беззнакового четырехбайтного целого. Именно этому полю следует присвоить значение `INADDR_ANY`.

Вызов `bind()` возвращает 0 в случае успеха, -1 – в случае ошибки. Учтите, что существует множество ситуаций, в которых вызов `bind()` может не пройти; например, ошибка произойдет в случае попытки использования привилегированного номера порта (от 1 до 1023) или порта, который на данной машине уже кем-то занят (возможно, другой вашей программой). Поэтому обработка ошибок при вызове `bind()` особенно важна.

Итак, окончательно подготовка и вызов `bind()` могут выглядеть следующим образом:

```
struct sockaddr_in addr;
addr.sin_family = AF_INET;
addr.sin_port = htons(port);
addr.sin_addr.s_addr = INADDR_ANY;
if (0 != bind(ls, (struct sockaddr *) &addr, sizeof(addr)))
{
  /* Здесь следует поместить обработку ошибки */
}
```

где `ls` – переменная, хранящая дескриптор сокета, а `port` – переменная, в которую тем или иным способом занесен избранный номер порта.

2.2.3 Ожидание и прием клиентских соединений

После того, как сокет создан и с ним связан адрес, его необходимо перевести в состояние ожидания запросов на соединения, или, иначе говоря, в *слушающий режим* (listening state). Это достигается вызовом `listen()`:

```
#include <sys/socket.h>

int listen(int sockfd, int qlen);
```

Параметр `sockfd` задает дескриптор сокета. Параметр `qlen` означает максимальную длину очереди пришедших запросов на соединение, которые сервер еще не принял к обработке. Некоторые операционные системы не поддерживают значения `qlen`> 5, поэтому обычно вторым параметром вызова `listen()` задают просто число 5.

Вызов `listen()` возвращает 0 в случае успеха, -1 – в случае ошибки. С учетом этого вызов может выглядеть так:

```
if (-1 == listen(ls, 5)) {
  /* Здесь следует поместить обработку ошибки */
}
```

В результате выполнения вызова `listen()` в системе появится *слушающий сокет*, который можно увидеть с помощью команды

```
netstat -a -n | grep LISTEN
```

Клиент, находящийся на любой машине, с которой доступна наша сеть, может установить соединение с нашим сокетом. Чтобы получить возможность обмена информацией с этим клиентом одновременно с ожиданием запросов на соединение от других клиентов, нам необходимо *принять* запрос на соединение. Это делается с помощью вызова `accept()`:

```
#include <sys/types.h>
#include <sys/socket.h>

int accept(int sockfd, struct sockaddr *addr, int *addrlen);
```

Параметр `sockfd` задает дескриптор слушающего сокета, на котором следует принять соединение. Что касается параметров `addr` и `addrlen`, то они позволяют узнать, с какого адреса исходит соединение. Информация записывается в структуру, на которую указывает указатель `addr`. Переменная, на которую указывает `addrlen`, должна перед обращением к `accept()` содержать длину структуры `addr` в байтах; после обращения в ней будет содержаться реальная длина данных, записанных в эту структуру. Естественно, использовать следует структуру типа `struct sockaddr_in` (см. §2.2.2).

Если информация об источнике соединения не интересна, в качестве обоих параметров `addr` и `addrlen` можно передать нулевые указатели.

Если во входной очереди уже имеется запрос на соединение, вызов `accept()` возвращает управление немедленно; в противном случае управление будет возвращено после того, как такой запрос будет получен.

Вызов `accept()` возвращает файловый дескриптор, связанный с сокетом, через который будет осуществляться связь с клиентом, соединение которого только что было принято.

Полученный файловый дескриптор должен быть неотрицательным числом. Если вызов `socket()` вернул значение -1, это свидетельствует о происшедшей ошибке. Программа **обязательно** должна корректно обрабатывать такую ситуацию.

2.3 Мультиплексирование ввода-вывода

После того, как вызов `accept()` успешно отработает в первый раз, в вашей программе появятся два файловых дескриптора, требующих внимания. Это слушающий сокет и сокет, полученный в результате вызова `accept()` (сокет клиента). На слушающий сокет могут поступить новые запросы, которые необходимо принимать вызовом `accept()`; в то же время на сокет клиента могут поступить данные, переданные клиентом, которые необходимо прочитать с помощью вызова `read()`. Какое из этих событий произойдет раньше, неизвестно.

Более того, в программе могут быть и другие источники событий. Так, клиентов, скорее всего, будет больше одного. Кроме того, для реализации управления сервером нам, возможно, потребуется организовать диалог с пользователем, запустившим сервер, для чего необходима возможность обработки данных, поступающих со стандартного ввода программы-сервера.

2.3.1 Методы организации многопользовательского сервера

Существует несколько подходов к организации программ, работающих с несколькими источниками событий. Самый простой (и некорректный) из них состоит в организации циклического опроса всех имеющихся источников событий. Так, все имеющиеся сокеты можно перевести в так называемый *неблокирующий* режим, в котором все системные вызовы, относящиеся к таким сокетам, которые не могут быть исполнены без блокирования выполнения процесса, будут возвращать управление сразу же, сигнализируя об ошибке (в частности, вызов `accept()`, будучи вызванным в отсутствие необработанного запроса на соединение, немедленно вернет -1).

Этот способ имеет фундаментальный неустранимый недостаток, заключающийся в наличии так называемого *активного ожидания*. Действительно, даже если не происходит никаких событий, требующих обработки, программа-сервер будет вновь и вновь опрашивать имеющиеся дескрипторы, впустую занимая процессорное время. Естественно, пользоваться таким методом ни в коем случае не следует.

Второй вариант организации многопользовательского сервера, о котором, в частности, рассказывается в основном курсе лекций “Системное программное обеспечение” в III семестре, основан на создании отдельного процесса для работы с каждым источником событий. В этом случае после каждого вызова `accept()` немедленно выполняется вызов `fork()`, порождающий новый процесс, и родительский процесс возвращается к обработке входящих запросов на соединение, в то время как дочерний процесс

обслуживает клиента, используя полученный от `accept()` дескриптор. Подробно этот механизм рассмотрен в книге [2].

Такой вариант идеально подходит для случая, когда каждый клиент обслуживается отдельно от остальных и не имеет с ними никакой связи. Однако для случая многопользовательской игры, которая проходит в общем игровом пространстве, такой способ подходит заметно хуже, поскольку влечет активное использование разделяемой памяти и семафоров, что само по себе усложняет программу[3].

Третий способ называется *мультиплексированием ввода-вывода* и может быть осуществлен с помощью системных вызовов `select()` или `poll()`[4]. В дальнейшем мы ограничимся рассмотрением функции `select()`. При желании читатель может освоить функцию `poll()` самостоятельно, прибегнув к литературе [3] и команде `man`.

2.3.2 Вызов select()

Системный вызов `select()` предназначен для использования в ситуации, когда необходимо организовать работу с несколькими файловыми дескрипторами, не имея a priori информации о том, какой из дескрипторов первым потребует внимания программы. Кроме того, возможно, требуется отслеживание некоторых событий по времени (например, тайм-аутов на сетевых соединениях).

Прототип вызова `select()` выглядит следующим образом:

```
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

int select(int n, fd_set *readfds,
                  fd_set *writefds,
                  fd_set *exceptfds,
                  struct timeval *timeout);
```

Параметры `readfds`, `writefds` и `exceptfds` обозначают множества файловых дескрипторов, для которых нас интересует, соответственно, возмож-

[3]Тем не менее, построить сервер таким образом вполне возможно. Несмотря на сложности, связанные с использованием разделяемой памяти, это может быть интересно в качестве упражнения.

[4]Вообще говоря, select() и poll() предназначены для одних и тех же действий. select() несколько проще в работе, poll() несколько более универсален. В некоторых системах ядро реализует только один вариант интерфейса, при этом второй эмулируется через него в виде библиотечной функции. Так, в системе Solaris присутствует системный вызов poll(), а select() является библиотечной функцией. Кроме того, в некоторых современных системах присутствует также вызов kqueue(), реализующий альтернативный подход к выборке события.

ность немедленного чтения, возможность немедленной записи и наличие исключительной ситуации. Параметр `n` указывает, какое количество элементов в этих множествах является значащим. Этот параметр необходимо установить равным `max_d+1`, где `max_d` – максимальный номер дескриптора среди подлежищих обработке. Наконец, параметр `timeout` задает промежуток времени, спустя который следует вернуть управление, даже если никаких событий, связанных с дескрипторами, не произошло.

Объект "множество дескрипторов" задается переменной типа `fd_set`. Внутренняя реализация переменных этого типа нас, вообще говоря, не интересует[5]. Для работы с переменными этого типа система предоставляет в наше распоряжение следующие макросы:

```
FD_ZERO(fd_set *set);       /* очистить множество */
FD_CLR(int fd, fd_set *set);/* убрать дескриптор из мн-ва */
FD_SET(int fd, fd_set *set);/* добавить дескриптор к мн-ву */
FD_ISSET(int fd, fd_set *set);
                            /* входит ли дескр-р в мн-во? */
```

В рассматриваемой задаче объемы передаваемых по сети данных сравнительно незначительны, что позволяет предполагать, что системные вызовы, осуществляющие запись в сокеты, никогда не будут блокировать программу-сервер. Также можно считать, что на сокетах никогда не произойдут исключительные ситуации. Таким образом, аргументы `writefds` и `exceptfds` можно не использовать (вместо них передавать вызову `select()` нулевые указатели).

В простейшей версии программы-сервера также нет необходимости в использовании параметра `timeout`[6]. Поэтому можно передать нулевой указатель и в качестве пятого параметра вызова. На всякий случай отметим, что структура `timeval` имеет два поля типа long. Поле `tv_sec` задает количество секунд, поле `tv_usec` - количество микросекунд (миллионных долей секунды).

Вызов `select()` изменяет все переданные ему по указателю аргументы, так что перед каждым обращением к нему аргументы должны быть сформированы заново.

Вызов `select()` возвращает -1 в случае возникновения ошибки. Учтите, что, если ваша программа обрабатывает те или иные сигналы, вызов `select()` может вернуть -1 в случае, если его выполнение было прервано пришедшим сигналом. При этом значением переменной `errno` будет `EINTR`, что свидетельствует о нормальном ходе событий. Вы можете не учитывать данный комментарий, если ваша программа не

[5]Для различных систем она может оказаться разной.

[6]Необходимость использования этого параметра возникает при выполнении некоторых дополнительных задач.

перехватывает никаких сигналов. Вызов возвращает значение 0 в случае, если причиной выхода из вызова стало наступление заданного таймаута. Если вызов возвратил положительное число, оно означает количество дескрипторов, для которых произошло какое-то событие.

После возврата из вызова `select()` переданные ему переменные типа `fd_set` оказываются модифицированы. Если при входе в вызов эти множества содержали дескрипторы, относительно которых нас интересует информация о событиях, то по окончании вызова эти же множества содержат дескрипторы, на которых событие реально произошло (например, по сети прибыли данные, которые могут быть считаны).

Таким образом, работу с вызовом `select()` можно построить по следующей схеме (считаем, что номер слушающего сокета по-прежнему хранится в переменной `ls`; как хранить дескрипторы клиентских сокетов, читателю предлагается решить самостоятельно):

```
for(;;) { /* главный цикл */
  fd_set readfds;
  int max_d = ls;
   /* изначально полагаем, что максимальным является
      номер слушающего сокета */
  FD_ZERO(&readfds); /* очищаем множество */
  FD_SET(ls, &readfds);
     /* вводим в множество
        дескриптор слушающего сокета */
  int fd;
       /* организуем цикл по сокетам клиентов */
  for(fd=/*дескриптор первого клиента*/ ;
      /*клиенты еще не исчерпаны?*/     ;
      fd=/*дескриптор следующего клиента*/) {
     /* здесь fd - очередной клиентский дескриптор */
     /* вносим его в множество */
    FD_SET(fd, &readfds);
     /* проверяем, не больше ли он,
        нежели текущий максимум */
    if(fd > max_d) max_d = fd;
  }
  int res = select(max_d+1, &readfds, NULL, NULL, NULL);
  if(res < 1) {
    /* обработка ошибки, происшедшей в select()'е */
  }
  if(FD_ISSET(ls, &readfds)) {
```

```
    /* пришел новый запрос на соединение */
    /* здесь его необходимо принять
       вызовом accept() и запомнить
       дескриптор нового клиента */
  }
    /* теперь перебираем все клиентские дескрипторы */
  for(fd=/*дескриптор первого клиента*/ ;
      /*клиенты еще не исчерпаны?*/     ;
      fd=/*дескриптор следующего клиента*/)
    if(FD_ISSET(fd, &readfds)) {
      /* пришли данные от клиента с сокетом fd */
      /* читаем их вызовом \verb.read(). или
         \verb.recv(). и обрабатываем */
    }

  /* конец главного цикла, можно идти на
     следующую итерацию */
}
```

2.4 Прием и передача данных через сокеты

2.4.1 Чтение

Чтение данных из сокета можно произвести обычным вызовом `read()`, уже знакомым читателю из курса “Системное программное обеспечение”:

```
#include <unistd.h>

size_t read(int fd, void *buf, size_t len);
```

либо специально предназначенным для сокетов вызовом `recv()`:

```
#include <sys/types.h>
#include <sys/socket.h>

size_t recv(int fd, void *buf, size_t len,
                          unsigned int flags);
```

В обоих случаях `fd` задает файловый дескриптор (в случае `recv()` это обязательно должен быть дескриптор сокета); `buf` указывает на буфер, в который следует поместить прочитанные данные; `len` сообщает вызову размер буфера, чтобы избежать его переполнения. Параметр `flags`, имеющийся

только у вызова `recv()`, позволяет установить некоторые специфические режимы работы, которые в рассматриваемой задаче не нужны. В дальнейшем изложении мы будем использовать вызов `read()`, что позволит использовать одни и те же фрагменты кода с потоками различной природы (кроме сокетов, это могут быть стандартные ввод и вывод, каналы типа `pipe` и др.)

Вызов `read()` производит чтение из потока. Прочитанные данные располагаются в буфере, на который указывает параметр `buf`. Читается не более чем `len` байт данных, что позволяет избежать переполнения буфера. Если в указанном потоке отсутствуют данные, готовые к прочтению, вызов блокирует вызваавший процесс до тех пор, пока данные не появятся, и только после их прочтения вернет управление.

Вызов возвращает -1 в случае ошибки. В случае успешного чтения возвращается положительное число, означающее количество прочитанных байт. Естественно, это число не может быть больше `len`. Случай, когда вызов вернет 0, рассматривается в §2.4.3.

Необходимо учитывать, что сообщение, отправленное клиентской программой по сети через потоковый сокет, не обязательно будет прочитано на стороне сервера в один прием. Например, клиент отправил текстовую строку `"Just a string\n"`, имеющую длину 14 байт. Вполне возможно, что она будет прочитана за один вызов `read()`, однако механизм сетевых сокетов такой гарантии не дает. Это означает, что очередной вызов `read()` может, например, прочитать 4 байта `"Just"`, следующий за ним - 5 байт `" a str"`, и, наконец, еще один - оставшиеся `"ing\n"` В связи с этим сервер **обязательно** должен иметь накопительный буфер, в котором принятые данные будут храниться до тех пор, пока не будет получена команда целиком. Команды можно отделять друг от друга, например, символом перевода строки или пустой строкой (два перевода строки подряд).

2.4.2 Запись

Для записи в сокет можно пользоваться вызовом `write()`:

```
#include <unistd.h>

size_t write(int fd, const void *buf, size_t len);
```

либо специально предназначенным для сокетов вызовом `send()`:

```
#include <sys/types.h>
```

```
#include <sys/socket.h>

size_t send(int fd, const void *buf, size_t len,
                                unsigned int flags);
```

Параметр `fd` задает файловый дескриптор (в случае `send()` это обязательно должен быть дескриптор сокета); `buf` указывает на буфер, содержащий данные, которые необходимо передать через сокет; `len` задает количество этих данных. Параметр `flags`, имеющийся только у вызова `send()`, позволяет установить некоторые специфические режимы работы, которые в рассматриваемой задаче не нужны. Как и в случае с чтением, в дальнейшем изложении мы отдадим предпочтение вызову `write()`, как более универсальному.

Вызов возвращает -1 в случае ошибки. В случае успешного чтения возвращается положительное число, означающее количество переданных байт. Естественно, это число не может быть больше `len`.

Следует обратить особое внимание на передачу ограничительных символов. Рассмотрим пример. Допустим, в программе задана строковая константа:

```
const char juststring[50] = "This is just a string\n";
```

Тогда вызов

```
write(sd, juststring, 21);
```

передаст в сокет `fd` строку без символа перевода строки и без ограничительного нуля; вызов

```
write(sd, juststring, 22);
```

или, что то же самое,

```
write(sd, juststring, strlen(juststring));
```

передаст ту же строку уже с символом перевода строки, но по-прежнему без ограничивающего нуля, что может привести к ошибкам при обработке на другой стороне соединения, если только не предпринять специальных мер по восстановлению ограничителя. Грубой ошибкой будет вызов

```
write(sd, juststring, sizeof(juststring));
```

В рассматриваемом примере такой вызов передаст в сокет 24 байта полезной информации и 26 байт случайного мусора, который, будучи проинтерпретирован как очередная команда или ее часть, вызовет ошибки. А если бы мы описали `juststring` не как массив, а как указатель, передано было бы и вовсе ровно 4 байта (размер указателя на большинстве современных архитектур).

Более правильно было бы написать примерно так:

```
write(sd, juststring, strlen(juststring)+1);
```

если, конечно, вам не мешает перевод строки. Если же его передача нежелательна, вставьте на его место ограничительный 0 и уже после этого передавайте.

2.4.3 Разрыв соединения и обработка разрыва

Завершить работу с сокетом можно с помощью вызова `shutdown`:

```
#include <sys/socket.h>

int shutdown(int sd, int how);
```

Параметр `sd` задает дескриптор сокета, `how` – что именно следует прекратить. При `how == 0` сокет закрывается на чтение, при `how == 1` – на запись, при `how == 2` – полностью (в оба направления). В рассматриваемой задаче требуется только последний вариант.

Вызов `shutdown()` прекращает работу с сокетом, однако сам сокет вместе с файловым дескриптором продолжает существовать. Чтобы окончательно избавиться от ненужного сокета и освободить дескриптор, следует закрыть его вызовом `close()`:

```
#include <unistd.h>

int close(int fd);
```

Естественно, сокет будет также закрыт, если завершился процесс, в котором этот сокет использовался, и других процессов, использующих тот же сокет (например, дочерних процессов, созданных вызовом `fork()`, когда сокет уже существовал) не осталось.

Узнать о том, что ваш партнер по коммуникации разорвал соединение, можно по значению, возвращаемому вызовом `read()` или `recv()`. В случае, если все данные из сокета прочитаны и на противоположном конце соединение закрыто, вызов чтения из сокета вернет 0.

Следует учитывать, что, если сокет в таком состоянии включить в множество дескрипторов, обрабатываемых вызовом `select()` (см.§2.3), то вызов вернет управление немедленно, причем дескриптор рассматриваемого сокета будет помечен как готовый на чтение. Вызов же на чтение опять вернет 0. Поэтому, **если не предусмотреть в программе корректной обработки ситуации "конец файла" на дескрипторах сокетов, возможно ее зацикливание при разрыве соединения одним из клиентов**.

2.5 Организация программы-клиента

2.5.1 Установление соединения

Как и в случае программы-сервера, для случая программы-клиента потребуется сокет, однако на этот раз сокет требуется только один. Создать его необходимо точно так же, как это делается в сервере, вызовом `socket()` (см. §2.2.1).

В случае, если по каким-то причинам вам необходимо указать IP-адрес и/или порт, с которого должно исходить создаваемое клиентское соединение, можно применить к сокету вызов `bind()` (см. §2.2.2). Однако такие ситуации редки и при выполнении задания практикума возникнуть не должны.

Чтобы установить соединение с сервером, необходимо воспользоваться вызовом `connect()`:

```
#include <sys/types.h>
#include <sys/socket.h>

int connect(int sockfd, struct sockaddr *serv_addr,
                                    size_t addrlen);
```

Здесь `sockfd` – дескриптор сокета, полученный в результате выполнения вызова `socket()`; `serv_addr` – указатель на структуру, содержащую адрес сервера; наконец, `addrlen` – размер структуры адреса в байтах.

Как и в §2.2.2, для хранения адреса используем структуру типа `struct sockaddr_in`. Естественно, чтобы связаться с сервером, необходимо каким-либо образом узнать (например, запросить у пользователя) IP-адрес и порт сервера. Номер порта, как и раньше, необходимо перевести в сетевой порядок байт функцией `htons()`. Что касается IP-адреса, то, имея его текстовое представление (например, строку `"192.168.10.12"`), можно воспользоваться функцией `inet_aton()` для формирования структуры типа `struct in_addr` (напомним, что поле `sin_addr` структуры `sockaddr_in` имеет именно этот тип):

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int inet_aton(const char *cp, struct in_addr *inp);
```

Здесь `cp` – строка, содержащая текстовое представление IP-адреса, а `inp` указывает на структуру, подлежащую заполнению. Функция возвращает ненулевое значение, если заданная строка является записью валидного IP-адреса, и 0 в противном случае.

Допустим, дескриптор сокета хранится в переменной `sockfd`, нужный нам IP-адрес содержится в строке `char *serv_ip`, а порт – в переменной `port` в виде целого числа. Тогда подготовка и вызов `connect()` могут выглядеть так:

```
struct sockaddr_in addr;
addr.sin_family = AF_INET;
addr.sin_port = htons(port);
if(!inet_aton(serv_ip, &(addr.sin_addr))) {
   /* Ошибка - введен невалидный IP-адрес */
}
if (0 != connect(sockfd, (struct sockaddr *)&addr,
                                      sizeof(addr)))
{
  /* Здесь следует поместить обработку ошибки connect */
}
```

2.5.2 Встречные потоки данных и их обработка

Встречные потоки данных. Программа-клиент обычно имеет два основных потока данных. Первый из них образуют данные, введенные пользователем (то есть, например, прочитанные со стандартного ввода), которые после необходимых преобразований передаются серверу через сокет. Второй поток образуют данные, полученные от сервера в качестве отклика; после преобразований они тем или иным способом передаются пользователю (например, через поток стандартного вывода).

Это не создает никаких проблем, если мы считаем, что сервер способен что-либо передавать клиенту исключительно в рамках ответа на очередную команду. В этом случае мы можем действовать по следующей схеме:

1. Распечатываем приглашение;

2. Считываем команду, введенную пользователем;
3. Формируем и передаем команду серверу;
4. Ожидаем отклика сервера;
5. Преобразуем полученную информацию;
6. Выдаем ее пользователю;
7. Переходим на начало.

Схема работы оказывается строго последовательной благодаря тому, что, пока сервер не отозвался на нашу команду, пользователь лишается возможности что-либо вводить, а пока пользователь не завершил свой ввод, программа не обрабатывает данные, поступающие от сервера.

Однако такая схема работы лишает нас возможности принимать от сервера сообщения о событиях, никак не связанных с нашими командами. Например, сообщение о том, что завершен очередной аукцион, логично было бы разослать всем игрокам непосредственно по завершении аукциона, что может произойти после команды, выданной другим игроком. То же касается, например, сообщения о том, что кто-то из игроков потерял связь с сервером и выбыл из игры. Наконец, можно предусмотреть в программе-сервере разнообразные напоминания, связанные со слишком долгим отсутствием активности со стороны игрока.

Точно таким же образом может оказаться, что блокировать действия пользователя на время обработки команды сервером неудобно. Так, клиентская программа может некоторые команды пользователя обрабатывать самостоятельно, не обращаясь при этом к серверу (например, программа может запоминать статистику предыдущих циклов игры и выдавать ее по запросу пользователя).

Решение с помощью двух процессов. Одно из возможных решений – использовать два процесса, один из которых будет читать пользовательский ввод и передавать готовые команды серверу, а второй – принимать сообщения от сервера и выдавать информацию пользователю. При этом во избежание конфликтов желательно в первом процессе закрыть поток стандартного вывода, выполнив вызов `close(1)`, а во втором – поток стандартного ввода, выполнив `close(0)`.

Это решение подходит, например, в случае, если клиентская программа не поддерживает команд, выполняемых без обращения к серверу. В противном случае процессу, отвечающему за ввод и передачу, чтобы отреагировать на введенную «внутреннюю» команду, потребуется возможность

записи в поток стандартного вывода, что в некоторых системах приводит к возникновению ошибочной ситуации.

Решение на основе select(). Более корректное решение возможно с использованием уже знакомого читателю вызова `select()` (см. §2.3). В этом случае отпадает потребность во втором процессе, т.к. операции чтения не будут блокировать основной процесс. Как и при организации программы-сервера, можно ограничиться заданием только одного параметра – множества дескрипторов, для которых нас интересует готовность на чтение (`readfds`). Обработке в этом случае подлежат всего два дескриптора: дескриптор потока стандартного ввода (0) и дескриптор сокета.

2.6 Дополнительные сведения

2.6.1 Подробнее о порядке байт в целых числах

Порядок байт в представлении целых чисел в памяти может варьироваться от одной архитектуры к другой. Архитектуры, в которых старший байт числа имеет наименьший адрес, в англоязычной литературе обозначаются термином *big-endian*, а архитектуры, в которых наименьший адрес имеет младший байт – *little-endian*[7].

Чтобы сделать возможным взаимодействие по сети между машинами, имеющими разные архитектуры, принято соглашение, что передача целочисленной информации по сети всегда идет в прямом (*big-endian*) порядке байт, т.е. старший байт передается первым. Чтобы обеспечить переносимость программ на уровне исходного кода, в операционных системах семейства Unix введены стандартные библиотечных функции для преобразования целых чисел из формата данной машины (*host byte order*) в сетевой формат (*network byte order*). На машинах, порядок байт в архитектуре которых совпадает с сетевым, эти функции просто возвращают свой аргумент, в ином случае они производят необходимые преобразования. Вот эти функции:

```
#include <netinet/in.h>

unsigned long int htonl(unsigned long int hostlong);
unsigned short int htons(unsigned short int hostshort);
```

[7] «Термины» big-endians и little-endians введены Свифтом в книге “Путешествия Гулливера” и на русский язык обычно переводились как *тупоконечники и остроконечники*. Аргументы в пользу той или иной архитектуры действительно часто напоминают священную войну остроконечников с тупоконечниками.

```
unsigned long int ntohl(unsigned long int netlong);
unsigned short int ntohs(unsigned short int netshort);
```

Как можно догадаться, буква `n` в названиях функций означает network (т.е. сетевой порядок байт), буква `h` – host (порядок байт данной машины). Наконец, `s` обозначает короткие целые, а `l` – длинные целые числа. Таким образом, например, функция `ntohl()` используется для преобразования длинного целого из сетевого порядка байт в порядок байт, используемый на данной машине.

2.6.2 Как избежать "залипания" TCP-порта по завершении сервера

Часто при работе с сервером можно заметить, что после завершения программы-сервера ее некоторое время невозможно запустить с тем же значением номера порта. Это происходит обычно при некорректном завершении программы-сервера, либо если программа-сервер завершается при активных клиентских соединениях. В этих случаях ядро операционной системы некоторое время продолжает считать адрес занятым.

Избежать этих ситуаций при отладке программы-сервера очень сложно, однако система сокетов позволяет изменить поведение ядра в отношении адресов, "залипших" подобным образом. Для этого необходимо перед вызовом `bind()` выставить на будущем слушающем сокете опцию `SO_REUSEADDR`. Это делается с помощью системного вызова `setsockopt()`:

```
#include <sys/types.h>
#include <sys/socket.h>

int setsockopt(int sd, int level, int optname,
               const void *optval, int optlen);
```

Параметр `sd` задает дескриптор сокета, `level` обозначает уровень (слой) стека протоколов, к которому имеет отношение устанавливаемая опция (в данном случае это уровень сокетов, обозначаемый константой `SOL_SOCKET`) Параметр `optname` задаёт "имя" (на самом деле, конечно, это номер, или числовой идентификатор) устанавливаемой опции; в данном случае нам нужна опция `SO_REUSEADDR`.

Поскольку информация, связанная с нужной опцией, может иметь произвольную сложность, вызов принимает нетипизированный указатель на значение опции и длину опции (параметры `optval` и `optlen` соответственно). Значением опции в данном случае будет целое число 1, так что следует завести переменную типа int, присвоить ей значение 1 и передать в

качестве `optval` адрес этой переменной, а в качестве `optlen` – выражение `sizeof(int)`. Таким образом, наш вызов будет выгдядеть так:

```
int opt = 1;
setsockopt(ls, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));
```

2.7 Рекомендации по тестированию

Проверку программы следует произвести в несколько этапов (тестов). На каждом этапе вам потребуется несколько работающих командных интерпретаторов. Если вы используете систему XWindow, запустите несколько экземпляров программы `xterm`. Если вы предпочитаете работать с текстовой консолью, выполните вход в систему на нескольких виртуальных консолях одновременно[8].

1. Запустите программу-сервер. Убедитесь с помощью команды `netstat -an`, что выбранный вами TCP-порт находится в состоянии LISTEN. Если это так, в выдаче команды будет присутствовать примерно такая строка:

   ```
   tcp  0  0   0.0.0.0:7000  0.0.0.0:*  LISTEN
   ```

   (здесь 7000 – избранный вами номер порта).

2. Прервите выполнение программы-сервера и запустите ее с другим номером порта (не забудьте, что порты с номерами 1..1023 для пользовательских процессов не доступны). Убедитесь, что теперь в состоянии LISTEN находится новый TCP-порт.

3. Используя другой командный интерпретатор (в другом окне xterm'а или на другой консоли), запустите утилиту telnet. В команой строке необходимо указать адрес и TCP-порт сервера. Если тестирование проводится в рамках одной машины, в качестве IP-адреса можно использовать 127.0.0.1 (localhost) или 0. Например, если вы используете порт 7000, команда будет выглядеть так:

   ```
   telnet 127.0.0.1 7000
   ```

   или так:

[8] В системах Linux и FreeBSD переключение виртуальных консолей производится обычно комбинацией Alt-Fn, где n - номер консоли (1, 2, 3, ..., 12). Учтите, что не все консоли могут быть доступны.

```
telnet 0 7000
```

Убедитесь, что связь установлена. Для этого запустите команду `netstat -an`. Если связь действительно установлена, в ее выдаче будут присутствовать примерно такие строки:

```
tcp  0  0   127.0.0.1:7000  127.0.0.1:6537  ESTABLISHED
tcp  0  0   127.0.0.1:6537  127.0.0.1:7000  ESTABLISHED
```

здесь 7000 - номер порта вашего сервера, а 6537 - номер порта, используемого программой telnet (может оказаться любым числом, большим 1023).

4. Разорвите связь с сервером. Для этого нажмите комбинацию `Ctrl-]`; должно появиться приглашение `telnet>`. Введите команду close и нажмите Enter.

5. Убедитесь, что сервер по-прежнему слушает порт, как это описано на шаге 1.

6. Прервите выполнение программы-сервера и запустите ее вновь. Установите связь.

7. Дайте команду чтения параметров. В ответ сервер должен сообщить, что к нему подключен один клиент и значение глобального счетчика равно нулю.

8. Несколько раз (например, трижды) дайте команду на увеличение счетчика. Убедитесь, что команда чтения параметров выдает правильное значение глобального счетчика.

9. Разорвите связь, убедитесь, что сервер по-прежнему слушает порт. Вновь установите связь. Убедитесь, что глобальный счетчик не изменился, а число клиентов по-прежнему равно 1.

10. Не разрывая связь, подключите к серверу еще несколько клиентов, запустив несколько экземпляров программы telnet. Убедитесь, что сервер отвечает на команды всех клиентов. Убедитесь, что число клиентов, выдаваемое сервером, соответствует числу реально подключенных клиентов.

11. Дайте несколько раз команду на увеличение счетчика в одном клиенте, а команду на выдачу значений – в другом. Убедитесь, что значение счетчика соответствующим образом меняется.

12. Разорвите связь с каждым из клиентов вразброс, начав с тех, что были запущены в середине (т.е. не с первого и не с последнего). После отключения каждого клиента проверяйте корректность выдаваемого сервером количества клиентов.

13. После того, как все клиенты будут отключены, убедитесь, что сервер по-прежнему слушает порт. Попытайтесь подключиться к нему еще раз.

14. Не разрывая связь с клиентом, прервите выполнение программы-сервера (при этом telnet сообщит о потере связи). Попробуйте снова запустить программу-сервер с тем же номером порта. Если это не удастся, обратитесь к §2.6.2.

3 Программирование логики игры

Когда низкоуровневая часть кода сервера готова, можно приступать к программированию самой игры. В настоящей главе собраны рекомендации по выполнению этого этапа задания.

3.1 Общие сведения

Весь сеанс игры выглядит следующим образом. Один из игроков запускает игровой сервер (вашу программу). После этого все игроки запускают клиентскую программу (например, `telnet`) и устанавливают связь с сервером. В момент входа очередного игрока все игроки (в том числе и только что вошедший) должны получить информационное сообщение о том, сколько игроков к настоящему моменту соединилось с сервером и скольких еще сервер ожидает.

Сервер должен дождаться входа нужного числа игроков, после чего начать игру. В качестве альтернативы можно предусмотреть команду `start`, которую выдаёт один из игроков, сочтя, что участников уже достаточно.

До момента начала игры сервер должен на любую введенную команду отвечать сообщением о том, что игра еще не началась.

После начала игры в ответ на попытку соединиться с сервером новый клиент должен получить сообщение о том, что игра уже началась и новые участники не принимаются, после чего сервер должен разорвать соединение.

В случае, если с одним из игроков связь потеряна до начала игры, его место должно считаться свободным (все оставшиеся игроки получают сообщение о том, что один из игроков вышел, количество имеющихся игроков, количество ожидаемых игроков).

В случае, если с одним из игроков потеряна связь уже после начала игры, проще всего считать этого игрока “сошедшим с дистанции”, т.е. объявить его выбывшим из игры (банкротом).

3.2 Протокол взаимодействия с клиентом

Начать программирование логики игры можно с продумывания протокола взаимодействия с клиентом.

Для связи клиента с сервером необходимо разработать протокол прикладного уровня, т.е. некоторый набор соглашений между клиентом и сервером о том, какие данные будут передаваться и что они должны означать.

В простейшем случае в качестве клиентской программы может использоваться стандартная утилита telnet. Все, что эта программа может делать – это принимать от пользователя текстовые команды строка за строкой и передавать их в неизменном виде серверу[9].

3.2.1 Набор команд

Для упрощения задачи рекомендуется принять соглашение, что каждая команда занимает ровно одну строку. Учтите, что в зависимости от настройки утилиты telnet строка может заканчиваться одним символом `'\n'`, а может и двумя: `'\r' '\n'`. Проще всего игнорировать символ `'\r'` как пробельный.

Необходимо предусмотреть команды для следующих функций:

— **Получение информации о состоянии рынка**. В ответ на команду сервер должен выдать игроку количество и минимальную стоимость продаваемого банком на данном цикле сырья, а также количество и максимальную стоимость покупаемой продукции. Также разумно выдать количество активных (необанкротившихся) игроков.

— **Получение информации о состоянии дел других игроков**. Команда должна принимать один параметр (номер игрока) и в ответ на нее сервер должен выдать количество денег, единиц сырья, единиц продукции, строящихся, обыкновенных и автоматизированных фабрик у игрока с таким номером.

— **Производство продукции на фабриках**. В течение цикла (игрового месяца) команду можно дать несколько раз, добавляя новые заказы, однако собственно производство (списание единиц сырья и зачисление единиц продукции) должно произойти лишь в конце цикла. Это делается, чтобы исключить возможность продавать продукцию в том же месяце, в котором было закуплено для нее сырье. В качестве альтернативы можно запрашивать игроков о количестве производимой продукции один раз в начале цикла, не принимая никаких других команд, пока игрок не заявит о своих объемах производства. В этом случае производство выполняется немедленно.

— **Подача заявки на участие в аукционе сырья**. Команда имеет два параметра, количество закупаемого сырья и предлагаемая закупочная цена. Сервер должен сразу же проверить, достаточно ли у данного игрока единиц сырья и не задал ли игрок цену, ме́ньшую минимальной. Если игрок ошибся, ему необходимо об этом сообщить.

— **Подача заявки на участие в аукционе готовой продукции**.

[9]На самом деле, функциональность утилиты telnet гораздо более сложна, однако воспользоваться ею в рамках нашей задачи не представляется возможным.

Команда по своим характеристикам аналогична предыдущей.

— **Заявка на строительство новой фабрики**. С игрока тут же списывается половина стоимости фабрики и заводится строящаяся фабрика, которая в свой срок превратится в действующую (при этом произойдет списание второй половины суммы).

— **Окончание действий на данном ходе**. После подачи этой команды от данного игрока не принимаются команды, соответствующие активным действиям (заявки и строительство), но должны попрежнему отрабатываться команды получения информации. Это продолжается до тех пор, пока в игре есть игроки, еще не закончившие данный ход (желательно предусмотреть возможность узнать их список). В случае, если данный игрок не подал заявку на один из аукционов или на оба аукциона, соответствующие заявки считаются нулевыми. После того, как последний игрок заявит об окончании хода, сервер должен **сообщить всем игрокам результаты аукционов**, после чего начать следующий ход.

  В случае, если еще остались игроки, не закончившие ход, следует сообщить пользователю их количество.

— **Помощь**. По этой команде сервер должен выдать список существующих команд и их функции.

Команды должны быть достаточно мнемоничными, чтобы не запутаться в них. Например, в качесте команд можно использовать следующие английские слова и сокращения: `market`, `player`, `prod`, `buy`, `sell`, `build`, `turn`, `help`.

Весьма желательно сделать анализатор команд нечувствительным к количеству пробельных символов (пробелов, табуляций и возвратов каретки) между параметрами одной команды.

Также следует включить в сообщение о неопознанной команде информацию о том, как в вашей системе выглядит команда получения помощи.

3.2.2 Выдаваемые сообщения

При продумывании сообщений, выдаваемых сервером клиенту, следует учесть, что следующий этап задания предусматривает создание программ, имитирующих поведение игроков, так что выдаваемые вашим сервером сообщения необходимо будет анализировать программным образом.

С другой стороны, не следует забывать и о том, что обычно в игре участвуют обычные «живые» игроки.

Чрезмерно многословные сообщения, выдаваемые сервером, затруднят их машинный анализ; наоборот, чрезмерно формальные сообщения сдела-

ют игру неудобной с точки зрения игроков-людей. Важно найти оптимальный компромисс.

Одно из возможных решений состоит в том, чтобы информация, подлежащая машинному анализу, располагалась в строках, особым образом помеченных (например, начинающихся с символа %), а комментарии, предназначенные для человека, располагались в строках, такой пометки не имеющей. Вот пример ответа сервера на команду `market`:

```
Current month is 27th
Players still active:
%           3
Bank sells: items  min.price
%           6      500
Bank buys:  items  max.price
%           6      5500
```

Для человека такой ответ вполне читабелен. Что касается машинного анализа, то, выбрав только строки, первым символом которых является процент, удалив символ процента и разбив полученную информацию на отдельные слова, мы получим последовательность из пяти чисел. Помня, что выдана была команда `market`, мы знаем, что полученные числа означают количество участвующих в игре игроков, количество и минимальную цену сырья, количество и максимальную цену продукции соответственно. Программе, в отличие от человека, комментарии не нужны; отбрасывая их, мы облегчаем анализ текста, полученного от сервера.

3.3 Структуры данных

Известно, что от грамотного построения структур данных во многом зависит дальшейшая судьба любого более-менее сложного программистского проекта.

Попытаемся сформулировать некоторые рекомендации по проектированию структур данных для нашего игрового сервера.

Для начала отметим, что злоупотребление глобальными переменными – едва ли не самая серьёзная ошибка, какую только можно допустить в проектировании. Глобальные переменные затрудняют модификацию отдельных частей программы, делают невозможным повторное использование кода, порождают путаницу на этапе отладки.

Необходимо отметить, что обойтись без глобальных переменных можно *всегда*. Каждая функция может получить в качестве параметров все зна-

чения, которые ей нужны для работы, и адреса всех переменных, которые ей необходимо модифицировать.

Может показаться, что при этом каждой функции придётся передавать слишком большое количество параметров. Например, если в программе было описано сто глобальных переменных, то после избавления от них придётся перечислять сто параметров в каждом вызове функции.

Это, разумеется, не так. Во-первых, каждая отдельно взятая функция работает только с частью глобальных данных; скорее всего, если в программе было сто глобальных переменных, то большинство функций использует только пять-шесть, реже десять из них каждая. Во-вторых, логически связанные между собой данные можно группировать в переменные типа «структура» и передавать функциям вместо нескольких отдельных параметров адрес одной структуры.

Вернёмся к нашему игровому серверу. Можно заметить, что все данные, присутствующие в игре и определяющие состояние игры, делятся на глобальные данные (например, обстановка на рынке, номер текущего хода и т.п.) и данные, связанные с конкретным игроком (номер игрока, количество денег, сырья, готовой продукции, сведения о работающих и строящихся фабриках).

Это позволяет наметить общий подход к формированию данных, описывающих состояние игры. Следует, по-видимому, описать структуру “банкир”, содержащую глобальные данные и создать одну (и только одну) переменную этого типа. Кроме того, следует описать структуру “игрок”, которая будет содержать данные, связанные с одним игроком.

Переменные типа “игрок” можно объединить в массив либо в односвязный список. Скорее всего, более сложные структуры данных нам здесь не потребуются.

Можно сделать список игроков частью структуры “банкир”, в результате чего вся игровая информация окажется доступной через одну-единственную переменную.

Если теперь все функции, осуществляющие операции над игровыми данными, снабдить параметром “указатель на структуру банкир”, потребность в глобальных переменных полностью отпадет и можно будет описать единственный экземпляр структуры “банкир” как локальную переменную функции `main()`.

3.4 О таблице смены уровней рынка

При реализации смены состояния рынка в соответствии с таблицей 2 (см. стр. 6) студенты часто пользуются вложенными конструкциями

`switch/case`. Такая реализация требует существенных затрат времени и сил, а итоговый код получается громоздким, неудобным для восприятия и часто содержит ошибки.

Гораздо проще реализовать выбор нового уровня рынка через двумерную матрицу 5×5, в которой номер строки представляет текущее значение уровня рынка, номер столбца – новое значение, а соответствующая клетка таблицы определяет вероятность соответствующего перехода (именно таким образом организована информация в таблице 2).

Чтобы избежать использования приближенных вычислений, можно все значения вероятности, приведённые в таблице, домножить на 12. Получим следующую матрицу:

```
const int level_change[5][5] = {
   { 4, 4, 2, 1, 1 },
   { 3, 4, 3, 1, 1 },
   { 1, 3, 4, 3, 1 },
   { 1, 1, 3, 4, 3 },
   { 1, 1, 2, 4, 4 }
};
```

Получив с помощью функции `rand()` случайное число r от 1 до 12 включительно, делаем цикл по соответствующей строке матрицы, суммируя элементы, до тех пор, пока сумма не окажется больше либо равна r. Например, если текущий уровень – 2 и выпало число 8, то новый уровень будет 3, поскольку $3 + 4 = 7 < 8$, а $3 + 4 + 3 = 10 >= 8$. Будьте внимательны с индексацией: если уровни рынка нумеруются с 1 до 5, то соответствующие им индексы в матрице будут на единицу меньше.

Случайное число r от 1 до 12 проще всего получить с помощью выражения `rand()%12+1`, но так делать не рекомендуется, поскольку распределение младших бит псевдослучайных чисел может быть неравномерным.

В книге [6] рекомендуется использовать в подобных ситуациях следующий оператор:

```
r = 1 + (int)(12.0*rand()/(RAND_MAX+1.0));
```

3.5 О проведении аукционов

Реализация проведения аукционов является одним из наиболее алгоритмически сложных фрагментов программы. Наибольшее количество ошибок связано с обработкой ситуации, при которой в игре имеется несколько заявок с одинаковой установленной ценой, которые при этом не могут быть

удовлетворены одновременно (т.е. их сумма превышает доступное количество лотов). Необходимо помнить, что в этой ситуации победитель должен определяться по жребию.

Один из возможных алгоритмов реализации аукциона выглядит следующим образом[10]:

1. Скопировать данные о полученных заявках во временную структуру данных (например, список) и рассортировать их по убыванию предложенной цены. Структура данных должна содержать размер заявки, цену и указатель на оригинал заявки (либо просто указатель на объект игрока, подавшего заявку).

2. Если список заявок пуст, закончить.

3. Рассматривая максимальную цену (т.е. цену, указанную в первом элементе временного списка заявок), сосчитать общее количество единиц сырья, которые игроки готовы купить по этой цене. Это означает, что необходимо просуммировать размеры тех (нескольких первых в списке) заявок, в которых цена равна максимальной.

4. Если полученное количество не превышает количество доступных (продаваемых банком) единиц сырья, то удовлетворить все максимальные заявки, т.е. пометить все заявки, имеющие максимальную цену, как полностью удовлетворенные[11], уменьшить количество доступного сырья на сумму удовлетворенных заявок и убрать максимальные заявки из временного списка. После этого перейти к шагу 2.

5. В противном случае распределить оставшиеся единицы сырья по жребию между подателями максимальных заявок и на этом закончить.

3.6 Диалоги с пользователем

В некоторых случаях возникает потребность провести с пользователем (игроком) обмен несколькими последовательными репликами, т.е. диалог. Особенность здесь в том, что ответ пользователя на реплику сервера должен

[10]Здесь рассматривается аукцион продажи сырья; аукцион покупки готовой продукции реализуется аналогично с учетом того, что наиболее выгодной для банка является не максимальная, а минимальная предложенная цена.

[11]Важно понимать, что пометку об удовлетворении следует делать не во временной структуре данных, а в структурах, содержащих оригинальные данные; например, можно сделать соответствующую пометку в структуре соответствующего игрока.

быть воспринят именно как ответ на реплику, а не как очередная команда. Например, можно встроить в сервер запрос дополнительного подтверждения на исполнение некоторых "странных" команд; так, если пользователь заявил о готовности покупать сырье по цене выше 3000 (что заведомо не может окупиться), логично поинтересоваться, действительно ли это то, что он имел в виду[12].

Для реализации подобных возможностей студенты часто применяют самый очевидный (и заведомо недопустимый в условиях многопользовательского сервера) подход: выдать сообщение в сокет и после этого *дождаться ответа пользователя* (т.е. сразу за вызовом `write` ставится вызов `read`).

Проблема здесь состоит в том, что пользователь, разумеется, ответит на вопрос не сразу, и все это время сервер не будет обрабатывать команды других игроков, поскольку процесс сервера будет заблокирован на вызове `read` в ожидании ответа `одного` пользователя на заданный сервером вопрос. Игрок, знакомый с этой особенностью сервера, вполне может вызвать ситуацию диалога намеренно, чтобы приостановить игру.

С другой стороны, если после выдачи сообщения пользователю сервер продолжит обычное выполнение главного цикла с вызовом `select`, возникает проблема, как после очередного возврата из `select`'а проинтерпретировать данные, пришедшие от *одного конкретного клиента*, находящегося в режиме диалога, не как команду, а как ответ на заданный этому пользователю вопрос.

Очевидно, что информацию о том, что данному пользователю был задан некий вопрос, следует сохранить в структуре, соответствующей данному игроку. Получается, что при налиычии заданного вопроса, на который еще не получен ответ, ближайшая прочитанная с сокета данного клиента строка должна интерпретироваться как ответ на соответствующий вопрос, в противном же случае – как обычная команда. Заметим, что после того, как ответ на вопрос получен, необходимо вернуть соответствующего игрока в обычный режим, в котором его ввод будет восприниматься как команды.

Вышесказанное логично подводит нас к понятию *состояния*. Состояние связывается с каждым отдельным игроком (клиентом). В нормальном состоянии весь ввод, получаемый от клиента, воспринимается как команда. При задании игроку специфического вопроса необходимо изменить состояние на соответствующее данному конкретному вопросу, после чего продолжить выполнение главного цикла программы как обычно. Количество таких возможных состояний в точности равно количеству различных вопросов, которые в тех или иных ситуациях может задать игроку сервер, плюс еще одно состояние для "нормального режима". Более сложные диа-

[12]Этот параграф можно пропустить, если такой потребности у вас не возникло

логи, остоящие из более чем одной реплики, используют столько состояний, сколько в них имеется реплик, последовательно (при получении на очередной итерации главного цикла очередного ответа от пользователя) переходя от одного состояния к другому.

Программирование в таком стиле называется автомат-ориентированным.

Реализовать состояние рекомендуется в виде переменной **перечислимого типа**.

3.7 Рекомендации по тестированию

Тестирование игрового сервера следует проводит в микрогруппах по 3-4 человека. Участники группы играют несколько многопользовательских партий с использованием сервера очередного участника.

Предусмотрите в программе-сервере генерацию отчета по каждому ходу, включающего список поданных на аукционы заявок с индикацией об их удовлетворении/неудовлетворении, количество произведенной каждым игроком продукции и остаток средств на счету каждого игрока на момент конца хода. Информацию выдавайте в поток стандартного вывода сервера.

4 Программируемый робот

Вторая часть практикума предусматривает создание программы-робота, имитирующего поведение человека (игрока) в игре «Менеджмент». Это задание расчитано на выполнение в IV семестре и предполагает использование языка C++.

4.1 Постановка задачи

Стартовыми параметрами программы-робота являются ip-адрес и номер tcp-порта сервера, а также имя файла, содержащего программу на модельном языке (сценарий). Сценарий определяет дальнейшее поведение робота. Таким образом, программа-робот представляет собой комбинацию программы-клиента (см. § 2.5.1) и интерпретатора модельного языка.

Как и в случае с сервером, стартовые параметры задаются в командной строке. По согласованию с преподавателем возможны другие варианты получения стартовых параметров: например, из конфигурационного файла или из переменных окружения. Задавать стартовые параметры в тексте программы (т.е. так, что их изменение потребует перекомпиляции программы) *запрещается*.

Входной язык робота должен позволять использование всей информации, которая доступна обычному игроку, а также выдачу всех команд, которые мог бы выдать обычный игрок. Язык должен быть алгоритмически полным и иметь возможности, достаточные для задания нетривиальных стратегий (например, включающих в себя статистическую экстраполяцию). Для этого, в частности, необходимо предусмотреть в языке массивы (хотя бы одномерные). Вместе с тем, нет необходимости реализовывать в языке строчные переменные или переменные разных числовых типов; достаточно будет наличия переменных одного целочисленного типа (например, четырехбайтных целых); предпочтительнее, однако, было бы наличие переменных с плавающей точкой.

Для обеспечения в языке алгоритмической полноты необходимо предусмотреть конструкции, позволяющие задать ветвление и цикл. Минимальный набор конструкций для этого состоит из условного оператора и оператора безусловного перехода. Желательно предусмотреть в языке также составной оператор и оператор цикла с предусловием.

Обязательным требованием является равноправие всех пробельных символов (пробелов, табуляций, переводов строки и возвратов каретки) и допустимость любого их количества в любом месте программы, где допустим один пробел. Таким образом, нельзя

использовать конец строки в качестве разделителя операторов, как это делается в ранних версиях Бейсика и Фортрана. Для разделения операторов рекомендуется использовать символ “`;`” (точка с запятой).

4.2 Пример входного языка робота

Описываемый в этом параграфе язык является минимальным, т.е. упрощенным настолько, насколько это вообще возможно в рамках поставленной задачи. По требованию преподавателя на язык могут быть наложены дополнительные условия, усложняющие его. Однако даже в отсутствие дополнительных требований **рекомендуется предложить свой вариант языка, используя приведенный здесь язык лишь в качестве примера**.

4.2.1 Общее описание

Программа на модельном языке состоит из операторов, каждый из которых может быть помечен меткой. Для упрощения анализа имя метки начинается всегда с символа `@` и заканчивается двоеточием; имя метки может состоять из латинских букв, цифр и знаков подчеркивания. Если оператор начинается не с метки, считается, что этот оператор не имеет метки. Конец оператора обозначается точкой с запятой. В дальнейшем при описании синтаксиса операторов мы не упоминаем возможное наличие у оператора метки, чтобы не загромождать текст.

В языке присутствуют следующие операторы:

— оператор присваивания
— оператор безусловного перехода (`goto`)
— условный оператор (`if`)
— операторы игровых действий (`buy`, `sell`, `prod`, `build`, `upgrade` и `endturn`)
— оператор отладочной печати (`print`).

4.2.2 Арифметика. Выражения

В языке поддерживаются арифметические операции сложения (`+`), вычитания (`-`), умножения (`*`), целочисленного деления (`/`), вычисления остатка от деления (`%`), а также операции сравнения (`<`, `>`, `=`) и логические операции (`&`, `|`, `!`). Обязательна поддержка унарного минуса. Операции сравнения выдают значение `1` для обозначения истины и `0` для обозначения лжи. Логические операции интерпретируют число 0 как ложь, все остальные числа

– как истину. Результатом логической операции могут быть только числа `0` и `1`.

Наибольший приоритет имеют умножение, деление и остаток от деления, следующий уровень приоритета имеют сложение и вычитание, операции сравнения имеют низший приоритет.

В качестве операндов могут выступать константы, переменные и обращения к встроенным функциям.

В выражениях могут присутствовать круглые скобки любой вложенности.

4.2.3 Переменные. Массивы. Присваивания

Переменные в языке имеют имена, начинающиеся со знака "`$`". В имени переменной могут присутствовать латинские буквы, цифры и знак подчеркивания. После имени переменной может следовать указание индекса – произвольное арифметическое выражение, заключенное в квадратные скобки. Описывать переменные не требуется; переменная заносится в таблицу значений переменных в момент первого присваивания.

Оператор присваивания имеет следующий синтаксис:

```
<присваивание> ::= <переменная> = <выражение> ';'
<переменная> ::= <имя_переменной>          |
                 <имя_переменной> '[' <выражение> ']'
```

Можно заметить, что никакие другие операторы модельного языка не могут начинаться с переменной, поэтому переменная, встреченная при анализе в начале оператора, однозначно указывает на то, что анализируемый оператор является оператором присваивания.

Примеры операторов присваивания:

```
$a = 5;
$b[4] =15 ;
$b[$a]= $b[4]+3;
$c=($a+10) * $b[5];
```

Обратите внимание, что по обе стороны от знака присваивания допустимо любое количество пробельных символов.

Для упрощения реализации можно рассматривать элементы массива как самостоятельные переменные; при этом индекс становится частью имени переменной. С точки зрения интерпретатора элемент массива становится в таком случае своеобразной переменной, часть имени которой вычисляется в момент исполнения оператора. Если рассматривать только что

приведенный пример, то после выполнения таких операторов в таблице значений переменных должны появиться:

— переменная с именем “a” и значением 5;
— переменная с именем “b[4]” и значением 15;
— переменная с именем “b[5]” и значением 18;
— переменная с именем “c” и значением 270.

4.2.4 Условный оператор

Условный оператор имеет следующий синтаксис:

```
<условный_оператор> ::= 'if' <выражение> 'then' <оператор> ';'
```

При анализе оператора следует считывать выражение до тех пор, пока не встретится лексема, не являющаяся знаком операции, константой, переменной или обращением к функции. В данном случае следующей за выражением должна оказаться лексема `then`. Оператор, стоящий после `then`, должен быть проанализирован, однако его выполнение должно произойти только в случае, если вычисление выражения дало результат, отличный от нуля.

4.2.5 Встроенные функции для получения игровой информации

Для удобства анализа можно ввести соглашение, по которому все имена функций начинаются со знака “?”. Имя функции может состоять из латинских букв, цифр и знака подчеркивания. Если функция имеет аргументы, то вслед за именем функции должно идти заключенное в круглые скобки перечисление параметров функции через запятую. В минимальный набор функций, обеспечивающий доступ ко всей игровой информации, входят:

— `?my_id` (без параметров) – выдает номер игрока, присвоенный сервером нашему роботу;
— `?turn` (без параметров) – выдает текущий номер хода (условного месяца);
— `?players` (без параметров) – выдает общее число игроков;
— `?active_players` (без параметров) – выдает количество игроков, продолжающих игру (т.е. не обанкротившихся и не вышедших из игры к настоящему моменту);
— `?supply` (без параметров) – выдает количество сырья, выставленное банком на продажу на текущий ход;
— `?raw_price` (без параметров) – выдает минимальную стоимость сырья, определенную банком на текущий ход;

— `?demand` (без параметров) – выдает количество продукции, которую банк намерен купить на текущем ходу;
— `?production_price` (без параметров) – выдает максимальную цену продукции, определенную банком на текущий ход;
— `?money` (один параметр, номер игрока) – выдает количество денег у заданного игрока (соответственно, `?money(?my_id)` выдаст количество денег у игрока, управляемого нашим роботом);
— `?raw` (один параметр, номер игрока) – выдает количество сырья у заданного игрока;
— `?production` (один параметр, номер игрока) – выдает количество готовой продукции у заданного игрока;
— `?factories` (один параметр, номер игрока) – выдает общее количество (работающих) фабрик у заданного игрока;
— `?auto_factories` (один параметр, номер игрока) – какое количество фабрик данного игрока являются автоматизированными;
— `?manufactured` (один параметр, номер игрока) – сколько единиц продукции произведено на фабриках данного игрока на **предыдущем** ходу;
— `?result_raw_sold` (один параметр, номер игрока) – сколько единиц продукции произведено на фабриках данного игрока на **предыдущем** ходу;
— `?result_raw_price` (один параметр, номер игрока) – если данный игрок покупал сырье на предыдущем ходу, выдает цену, по которой совершена покупка, в противном случае - нуль;
— `?result_prod_bought` (один параметр, номер игрока) – сколько единиц продукции банк купил у данного игрока на предыдущем ходу;
— `?result_prod_price` (один параметр, номер игрока) – если данный игрок продавал продукцию на предыдущем ходу, выдает цену, по которой совершена продажа, в противном случае - нуль.

4.2.6 Встроенные операторы для совершения игровых действий

Операторы для совершения игровых действий имеют следующий синтаксис:

```
<игровой_оператор> ::= <имя0> ';'                          |
                       <имя1> <операнд> ';'                |
                       <имя2> <операнд1> ',' <операнд2> ';'
<имя0>             ::= 'endturn'
<имя1>             ::= 'prod' | 'build'
<имя2>             ::= 'buy' | 'sell'
```

Минимальный набор операторов включает:

— `buy <количество> , <цена>` – выставить заявку на покупку заданного количества сырья по заданой цене;

— `sell <количество> , <цена>` – выставить заявку на продажу заданного количества продукции по заданной цене;

— `prod <количество>` – запустить в производство заданное количество продукции;

— `build <количество>` – начать строительство заданного количества фабрик;

— `endturn` – сообщить серверу о завершении хода и дождаться сообщения от сервера о том, что начался новый ход.

Во всех случаях операнды представляют собой произвольные допустимые арифметические выражения. Для проверки правильности синтаксиса используется символ “;”, обозначающий конец оператора.

4.2.7 Оператор отладочной печати

Оператор отладочной печати имеет следующий синтаксис:

```
<оператор_печати> ::= 'print' <п_список> ';'
<п_список>        ::= <п_элемент> | <п_элемент> ',' <п_список>
<п_элемент>       ::= <выражение> | <строка>
```

Строка представляет собой произвольную строку символов, заключенных в двойные кавычки. Таким образом, список параметров оператора отладочной печати оказывается в рассматриваемом модельном языке единственным местом, где допустима текстовая константа.

Оператор отладочной печати обрабатывает свои параметры слева направо. Встреченные строки символов выдаются на стандартный вывод; встреченные выражения вычисляются и выдаются их результаты. Конец списка параметров определяется по встреченному символу “;”.

4.2.8 Пример программы-сценария

Следующая программа задает простейшую стратегию игры: на каждом ходу пытаемся купить 2 единицы сырья по минимальной цене, продать всю имеющуюся продукцию по максимальной цене, и произвести столько продукции, сколько имеется сырья, но не больше двух (поскольку мы никогда не строим фабрики, ясно, что больше двух единиц мы произвести не сможем).

```
@begin:
  print "Это ход номер" ?turn ;
  buy 2 ?raw_price ;
  sell ?production(?my_id) ?production_price ;
  $toprod = 2;
  if ?raw(?my_id) < $toprod then
          $toprod = ?raw(?my_id) ;
  prod $toprod ;
  endturn ;
  goto @begin ;
```

Учтите, что эта программа-сценарий приведена исключительно для примера. На практике вам следует самостоятельно разработать сценарии для своих роботов, причем сценарии гораздо более сложные, нежели приведенный выше. Заметим, что даже для случая совсем простой программы-сценария следует, по-видимому, избегать ситуаций банкротства, насколько это возможно, т.е. не подавать заявок на покупку сырья и на производство, если на выполнение заявки не хватает денег.

4.3 Клиентская часть программы

По-видимому, установить соединение следует сразу после того, как завершен анализ входного файла со сценарием, до начала выполнения сценария (поскольку прагматика языка сценария предполагает, что соединение уже установлено).

Адрес и порт игрового сервера ваша программа должна получить в качестве стартовых параметров. Системные вызовы, необходимые для организации TCP-клиента, описаны в §2.5.1.

Если протокол работы с сервером организован таким образом, что сервер присылает какие-либо данные клиенту только в ответ на запрос и никак иначе, при реализации робота можно обойтись без вызова `select()`. Достаточно при интерпретации скрипта, в случаях, когда необходимо обращение к серверу, передать данные серверу и сразу дать вызов `read()`, который заблокирует вашу программу до прихода данных с сервера. При этом следует иметь механизм, с помощью которого можно определить, закончил ли сервер передачу или пока передано не все. Можно предусмотреть в выдаче сервера какой-либо признак конца сообщения, например пустую строку.

В случае, если сервер может присылать клиенту какую-либо информацию по своей инициативе (например, сообщение о действиях других игроков), следует предусмотреть проверку готовности сокета к чтению. Это

можно сделать, например, с помощью вызова `select()` с нулевым значением таймаута. Если давать такой вызов, например, после выполнения каждого оператора сценария, можно отследить момент, когда сервер прислал какие-либо данные, прочитать их из сокета, проанализировать, сохранить полученную информацию в локальных переменных и продолжить интерпретацию сценария.

4.4 Лексический и синтаксический анализ

Необходимые в нашей задаче методы анализа и интерпретации изложены в пособии [5], поэтому в настоящем издании мы ограничимся рекомендациями, специфичными для конкретной задачи.

Анализ текста сценария рекомендуется вести в два этапа; на первом этапе провести лексический анализ текста, т.е. представить текст в виде списка лексем, на втором - провести синтаксический анализ и преобразовать список лексем во внутреннее представление, которое будет удобно интерпретировать (например, ПОЛИЗ).

Фаза лексического анализа может быть предельно упрощена благодаря особенностям входного языка. Например, для описанного в §4.2 языка существуют следующие лексемы:

— Разделители, а именно символы арифметических операций `+`, `-`, `*`, `/`, `%`, `<`, `>`, `=`, `&`, `|` и `!`; круглые и квадратные скобки; разделитель операторов – символ “`;`”; разделитель операндов – символ “`,`”;
— Целочисленные константы (непрерывная последовательность цифр);
— Строковые константы (произвольная строка символов, заключенная в двойные кавычки);
— Имена переменных (имена, начинающиеся с `$`);
— Имена меток (имена, начинающиеся с `@`);
— Имена функций (имена, начинающиеся с `?`);
— Ключевые слова `if`, `then`, `goto`, `print`, `buy`, `sell`, `prod`, `build` и `endturn`.

Во всех именах могут присутствовать только латинские буквы, цифры и знак подчеркивания, так что любой символ, не входящий в это множество, следует рассматривать как признак конца лексемы. Для разделения лексем могут использоваться пробельные символы; отделять лексемы-разделители от других лексем пробелами не обязательно (они являются разделителями сами по себе).

Поскольку имена разделены по типам еще на этапе лексического анализа, синтаксический анализ также оказывается достаточно простым; его можно провести методом рекурсивного спуска. О том, как это делается,

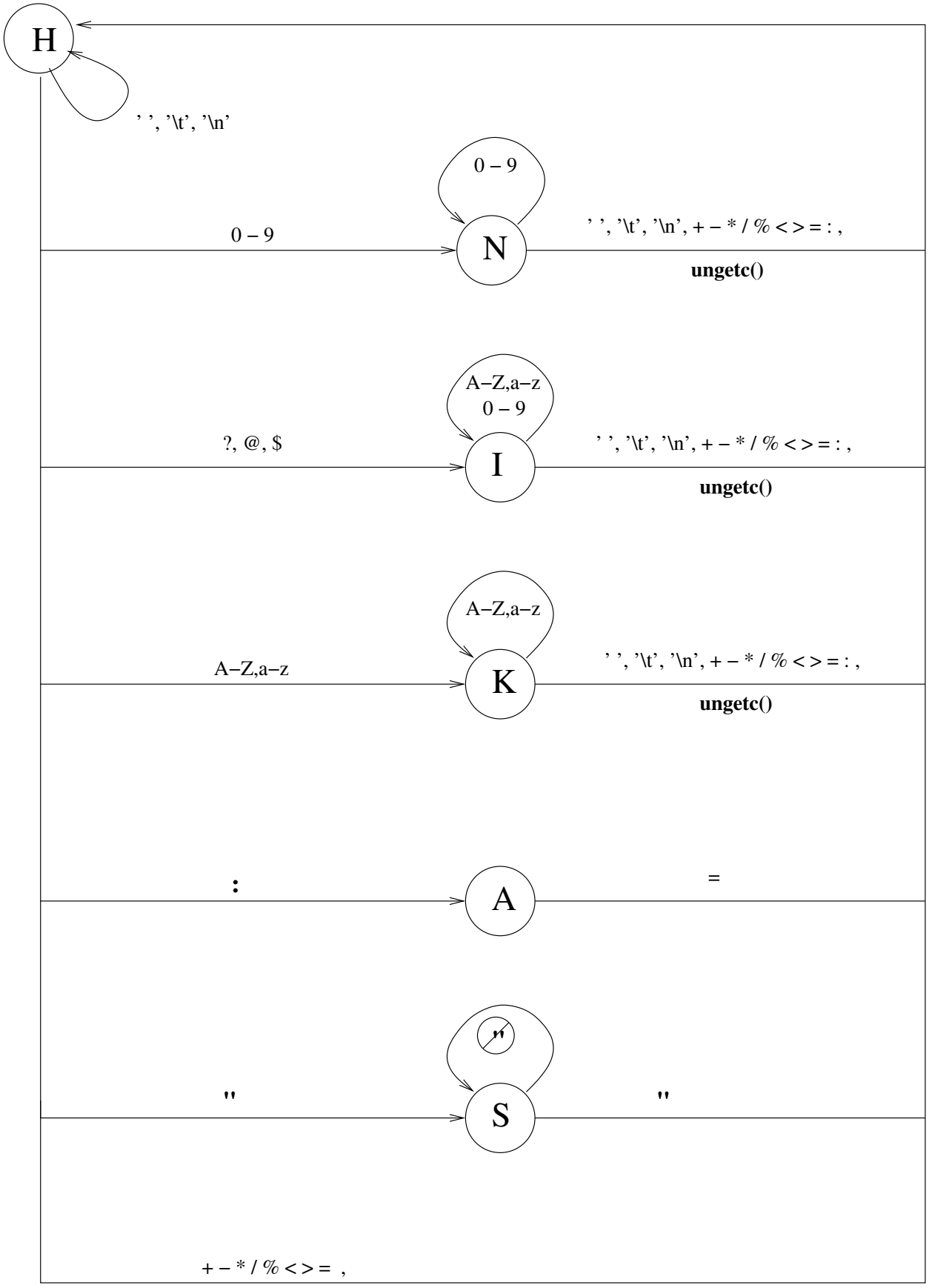

Рис. 1: Диаграмма состояний лексического анализатора

можно узнать из пособия [5].

Если реализуемый вами язык отличается от описанного выше, возможно, что его анализ окажется более сложной задачей. В любом случае, постарайтесь сохранить синтаксис языка достаточно простым, чтобы можно было применить метод рекурсивного спуска.

4.4.1 Реализация лексического анализатора

Поскольку задание выполняется на языке C++, следует, безусловно, попытаться использовать объектно-ориентированные средства этого языка. В частности, лексический анализатор представляет собой подзадачу, на примере которой можно освоить едва ли не самое широко используемое свойство объектно-ориентированного программирования – **инкапсуляцию**.

Прежде всего заметим, что лексический анализатор, построенный на основе регулярных грамматик, представляет собой конечный автомат, находящийся в одном из предопределённых состояний и имеющий, в дополнение к этому, некий буфер для накопления текущей лексемы. На каждом шаге работы автомат считывает очередной символ и в зависимости от его значения переходит в другие состояния, а также выполняет некоторые дополнительные действия.

На рис. 1 приведен пример диаграммы состояний лексического анализатора, соответствующего описанному в §4.2 языку. Начальное состояние автомата помечено буквой **H**. Состояние **N** соответствует анализу целого числа, состояние **I** – анализу идентификатора (имени переменной, функции или метки), состояние **K** – анализу ключевого слова. Состояние **A** используется для ввода знака присваивания `:=` (использовать отдельное состояние пришлось в силу того, что лексема `:=` состоит из двух символов). Состояние **S** используется для обработки текстовых констант.

Следует обратить внимание, что выход из состояний **N**, **I** и **K** происходит при обнаружении разделительного символа, который, вообще говоря, может быть частью следующей лексемы. Поэтому при выходе из этих трёх состояний следует перед возвратом в начальное состояние вернуть прочитанный символ во входной поток, чтобы он был снова проанализирован на следующем шаге. Соответствующие переходы на диаграмме помечены надписью `ungetc()`.

Действия по формированию лексем на диаграмме никак не отражены. Предполагается, что все анализируемые символы, кроме пробельных, сохраняются в накопительном буфере, содержимое которого после считывания последнего символа лексемы преобразуется в очередной объект, представляющий лексему. Легко заметить, что очередная лексема формируется

всякий раз при возврате в начальное состояние, если только накопительный буфер не пуст.

Ясно, что **реализовать лексический анализатор лучше всего в виде класса**, скрыв в его приватной части состояние автомата и накопительный буфер. Возможны, например, следующие подходы к организации этого класса:

— Конструктор класса не имеет параметров; его функции сводятся к инициализации накопительного буфера. Анализ выполняется при вызове метода `Run()`, имеющего один параметр – открытый на чтение файл (поток). Из этого потока автомат посимвольно считывает подлежащий анализу текст. Метод `Run()` возвращает адрес первого элемента сформированного списка лексем.

— Конструктор класса получает на вход открытый на чтение файл (поток). Запуск автомата осуществляется вызовом метода GetNextLexem() (без параметров). Автомат работает до момента формирования лексемы, после чего метод `GetNextLexem()` возвращает сформированную лексему либо некоторое специальное значение, обозначающее конец файла.

— Конструктор класса не имеет параметров. Метод класса `Step()` имеет один параметр – очередной символ в потоке. При вызове этого метода автомат производит один шаг[13]. В случае, если в результате очередного шага сформирована лексема, `Step()` возвращает эту лексему, в противном случае возвращается специальное значение (например, нулевой указатель, и т.п.), обозначающее, что лексема еще не готова. Обработка конца файла в этом случае возлагается на вызывающего; чтобы не потерять последнюю лексему, следует по достижении конца файла вызвать `Step()` еще раз, передав ему символ пробела.

Возможны, разумеется, и другие подходы к организации класса лексического анализатора.

Важно также уделить внимание представлению понятия «лексема». В пособии[5] предлагается использовать структуру из двух чисел, одно из которых представляет собой номер типа лексемы, второе – номер самой лексемы в соответствующей таблице. Такой подход имеет множество серьезных недостатков. Прежде всего, разумеется, ни в коем случае не следует в явном виде указывать в программе числа, соответствующие типам лексем; для этого следует ввести и использовать перечислимый тип. Программа, написанная с использованием перечислимого типа, гораздо лучше читается.

Далее, не следует, разумеется, формировать таблицы лексем в глобаль-

[13]В этом случае операция ungetc() отрабатывается рекурсивным вызовом Step() с тем же символом

ных переменных. Единственная таблица, которая может быть описана глобально – это таблица ключевых слов[14], поскольку эта таблица не подлежит изменению.

Один из возможных подходов состоит в том, чтобы хранить таблицы в объекте лексического анализатора. Также можно описать для таблиц отдельный класс, создать его объект и передать адрес этого объекта в конструктор лексического анализатора.

Наконец, можно работать и вообще без таблиц; например, объект "лексема" может представлять собой структуру, одно поле которой – это идентификатор типа лексемы, а второе – указатель на строку, хранящую саму лексему. Для этой структуры весьма желательно предусмотреть деструктор, уничтожающий соответствующую строку при уничтожении лексемы. Также могут существенно облегчить жизнь конструктор копирования и операторы сравнения и присваивания.

Учтите, что при выдаче сообщений об ошибке следует указывать номер строки в анализируемом тексте. Для этого в объекте "лексема" рекомендуется предусмотреть еще одно поле, хранящее номер строки анализируемого текста, в которой данная лексема была обнаружена.

4.4.2 Синтаксический анализатор

На этапе синтаксического анализа мы работаем с лексемами в качестве символов.

Производить синтаксический анализ следует наиболее простым из пригодных методов, а именно методом рекурсивного спуска. Для этого прежде всего необходимо выписать грамматику избранного языка и проверить ее на применимость рекурсивного спуска. При необходимости следует преобразовать грамматику.

Метод рекурсивного спуска предполагает описание значительного количества процедур (по одной на каждый нетерминальный символ грамматики). Кроме того, при описании метода рекурсивного спуска обычно упоминается глобальная переменная, хранящая текущий символ (т.е. текущую лексему).

Ясно, что при работе на языке C++ эти функции и переменную следует инкапсулировать в один объект.

Как и в случае лексического анализа, возможны разные подходы к проектирсованию класса синтаксического анализатора. Общим в них является

[14]Можно обратить внимание, что лексемы, соответствующие знакам операций +, -, *, / и т.п. также являются ключевыми словами; после фазы лексического анализа никакой разницы между обычными ключевыми словами и разделителями нет

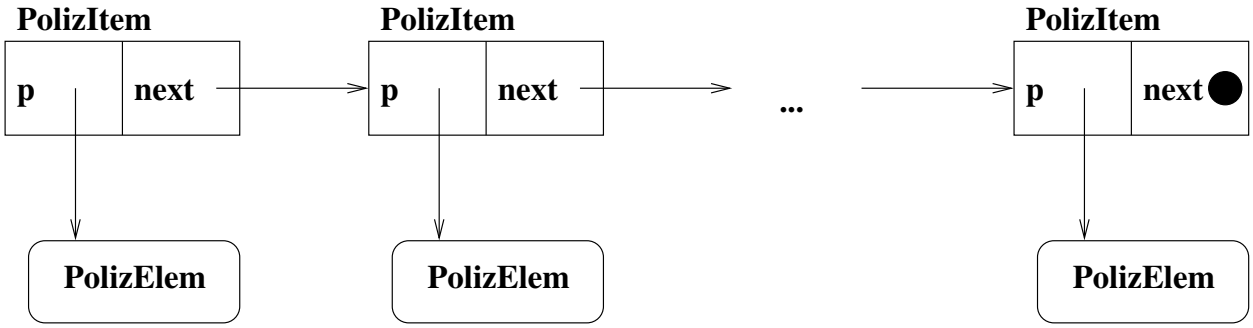

Рис. 2: Список элементов ПОЛИЗа

то, что функции, соответствующие символам грамматики, оформляются в виде методов в **приватной** части класса. Также в приватную часть необходимо поместить поле данных, хранящее текущую лексему. Приватной будет и функция "переход на следующую лексему".

При обнаружении ошибки для выхода из процедур рекурсивного спуска **следует использовать механизм исключений** языка C++.

Рекомендуется сначала создать синтаксический анализатор, способный проверить входную цепочку лексем на соответствие заданной грамматике и при ошибке выдающий осмысленную диагностику с указанием номера строки в анализируемом тексте.

После отладки в анализатор следует вставить код генерации внутреннего представления программы. Для этого необходимо преобразовать исходную грамматику в грамматику с действиями для синтаксически-управляемого перевода, после чего вставить соответствующие действия в нужные места кода синтаксического анализатора.

4.5 Интерпретация. ПОЛИЗ

Результатом работы анализатора должно стать внутреннее представление сценария, удобное для дальнейшей интерпретации. В качестве такого представления мы рекомендуем использовать ПОЛИЗ (см. [5]).

4.5.1 Примерная структура данных

Следует заметить, что элементы ПОЛИЗа относятся к разным типам, однако имеют и общие свойства; более того, они должны храниться в виде последовательной структуры данных (массива или списка), при этом обрабатываться схожими методами, работающими в зависимости от типа данного элемента ПОЛИЗа. Ясно, что понятие "*элемент полиза*" представляет собой пример предметной области, для моделирования которой прекрасно

подходит полиморфная иерархия классов.

Хранить элементы ПОЛИЗа можно, например, в односвязном списке, а метки, используемые операциями переходов, представлять указателями на соответствующее звено списка (в отличие от значения индекса в массиве элементов, как это предлагается в пособии [5]). Поскольку разные элементы ПОЛИЗа представляются объектами разных классов, звенья списка должны, по-видимому, хранить указатель на соответствующий объект, а не сам объект.

Далее в этом параграфе мы предполагаем, что базовый класс иерархии "элементы ПОЛИЗа" называется `PolizElem`, а структура для звена списка – `PolizItem`. Пример структуры данных, представляющей список элементов ПОЛИЗа, показан на рис. 2.

Заметим, что такая структура данных пригодна как для хранения самого ПОЛИЗа, так и для организации стека значений, необходимого в процессе интерпретации.

4.5.2 Типы элементов ПОЛИЗа

Рассмотрим теперь множество необходимых типов элементов ПОЛИЗа. Прежде всего, ПОЛИЗ может содержать целочисленные константы (или константы типа float, если в вашем языке предусмотрена арифметика с плавающей точкой). Кроме того, в языке фигурируют строковые литералы (строковые константы), для которых тоже необходимо предусмотреть возможность включения в ПОЛИЗ.

Вообще, константой называют любой элемент ПОЛИЗа, который, будучи встречен в процессе интерпретации, сразу же заносится в стек, после чего интерпретация продолжается со следующей позиции.

Можно заметить, что константами в этом смысле также являются метки, т.е. значения, задающие позицию в ПОЛИЗе для операций условного и безусловного перехода. В нашем примере таковые реализуются указателями на звено списка (т.е. на структуру типа `PolizItem`).

Наконец, напомним, что переменные могут входить в ПОЛИЗ двумя способами: как *обращение* к переменной (при интерпретации заменяется на значение переменной, которое и заносится в стек) и как *адрес* переменной, используемый обычно в левой части операторов присваивания[15]. В отличие от обращения, адрес не преобразуется к значению переменной, а заносится в стек как таковой, чтобы затем операция присваивания, используя сохраненный на стеке операнд, могла занести вычисленное значение в

[15]При изображении ПОЛИЗа на письме адреса переменных обычно подчеркивают, чтобы отличить их от обращений к переменным

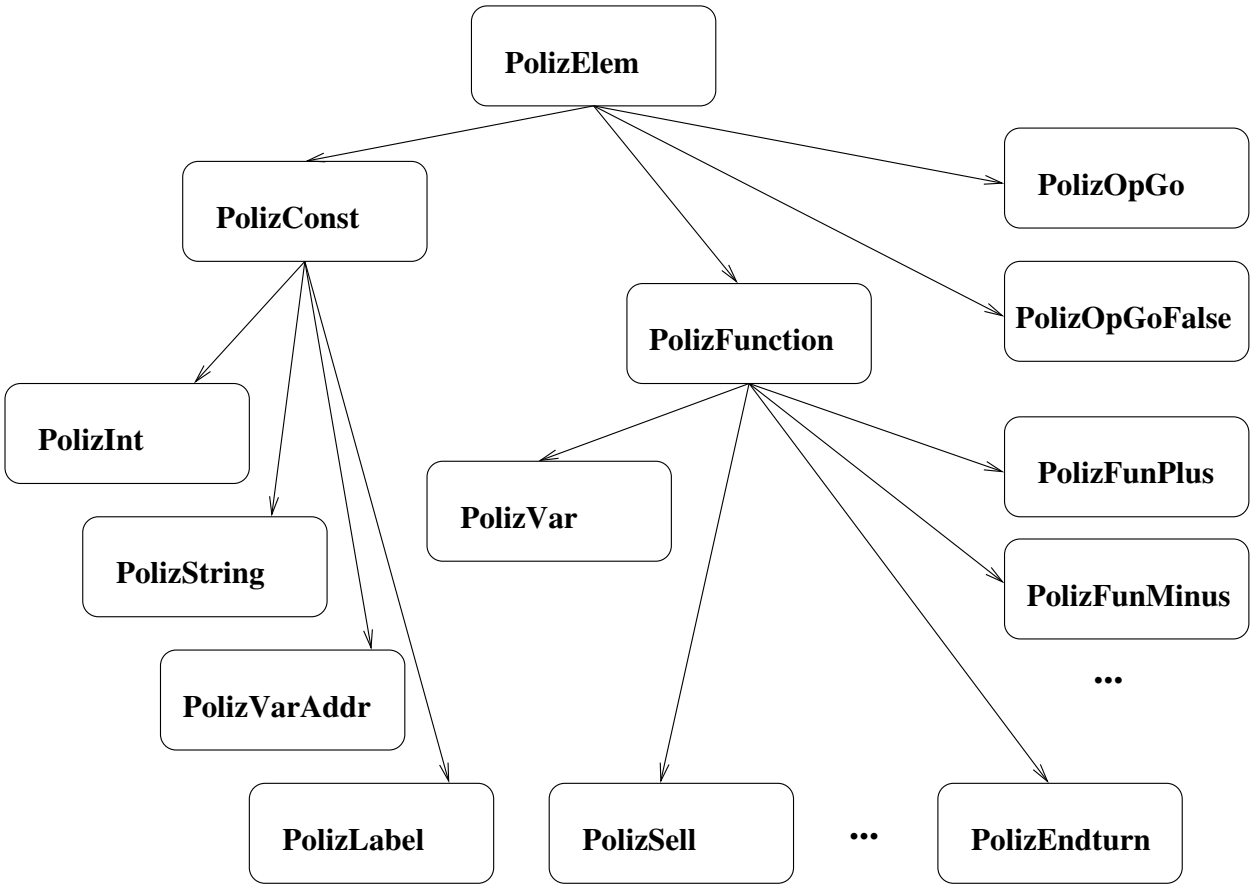

Рис. 3: Иерархия классов для представления ПОЛИЗа

нужное место. Таким образом, элемент ПОЛИЗа "адрес переменной" также является константой.

В простейшем случае перечисленными четырьмя типами исчерпывается список константных типов элементов ПОЛИЗа. Все остальные элементы представляют собой операции, т.е. их интерпретация подразумевает, в общем случае, извлечение из стека некоторого количества операндов, вычисление некоторого значения, которое потом заносится обратно в стек, и, возможно, выполнение некоторых иных действий. Количество операндов, подлежащих извлечению из стека при выполнении, может быть любым, в том числе и нулевым (например, для операции "обращение к переменной"). В нашем примере максимальная "арность" операции равна двум, т.е. больше двух аргументов ни одной операции не требуется.

Также существуют и операции, не вычисляющие значений, т.е. после их исполнения в стек ничего не заносится. Примеры таких операций – условный и безусловный переходы. Кроме того, для рассматриваемого языка необходимо предусмотреть операции, соответствующие операторам совершения игровых действий (см. §4.2.6) и оператору отладочной печати (см.

§4.2.7).

Как можно заметить, некоторые из операций должны иметь возможность влиять на процесс интерпретации путем явного изменения указателя на текущий интерпретируемый элемент. В нашем примере таких операций две: “безусловный переход” и “переход по лжи”.

Наконец, можно считать, что константы представляют собой частный случай операции, а именно, нуль-местную операцию, результат которой равен самому элементу, представляющему операцию.

Таким образом, понятие выполнения (вычисления) элемента ПОЛИЗа становится универсальным свойством всех объектов нашей иерархии.

Забегая вперед, договоримся избегать разделения структур данных, т.е. участия одних и тех же объектов в разных фрагментах структур данных. Это означает, например, что значения, заносимые в стек, всегда представляют собой специально порожденные для этого объекты; в некоторых случаях это будут копии объектов из ПОЛИЗа (при вычислении константы), в других случаях – только что созданные новые объекты (например, результат операции сложения).

Сразу же отметим, что копированию будут подвергаться только константные выражения, поскольку операции никогда в стек не заносятся. Таким образом, все константы, входящие в ПОЛИЗ, имеют как минимум одну общую операцию, отсутствующую для всех других классов иерархии. Имеется в виду операция создания копии. Это соображение сразу же наводит нас на мысль о создании общего предка для всех констант – абстрактного класса `PolizConst`, в котором вводится специальный метод для создания копии.

Как станет ясно из дальнейшего изложения, реализовывать операции ПОЛИЗа, не нуждающиеся в доступе к указателю на текущий интерпретируемый элемент (то есть все операции, кроме двух операций перехода) будет несколько проще, если ввести еще один абстрактный класс (мы назовем его `PolizFunction`), от которого будут наследоваться все операции, кроме операций перехода.

Окончательно наша иерархия классов примет вид, показанный на рис. 3.

4.5.3 Методы

Методы базового класса Сформулируем теперь требования к наиболее общему виду операции “вычисление элемента ПОЛИЗа” (далее предполагаем, что этот метод классов иерархии элементов ПОЛИЗа называется `Evaluate`). Операция должна иметь возможность:

— извлечения операндов из стека и занесения в стек нового значения, т.е. модификации стека;
— изменения указателя на текущий интерпретируемый элемент ПОЛИЗа (для выполнения операции перехода).

Как уже отмечалось в §3.3, использования глобальных переменных следует избегать, насколько это возможно. Таким образом, желательно передавать методу `Evaluate` всю необходимую информацию через параметры.

Поскольку как стек, так и указатель на текущий интерпретируемый элемент в нашем примере реализуются переменными типа `PolizItem*`, а сам элемент ПОЛИЗа никогда не изменяется в ходе интерпретации, профиль метода `Evaluate` в классе `PolizElem` может выглядеть так:

```
class PolizElem {
public:
    // ...
    virtual void Evaluate(PolizItem **stack,
                          PolizItem **cur_cmd) const = 0;
    // ...
};
```

Ясно, что этот интерфейс имеет слишком общий вид и может оказаться неудобен для реализации большинства объектов. Действительно, всего две операции (условный и безусловный переходы) нуждаются в прямом доступе к указателю текущей команды, тогда как все остальные типы элементов при вычислении производят над счетчиком одну и ту же операцию – сдвиг на следующий элемент. Что касается стека, то практически все команды после выполнения помещают в стек ровно одно значение либо не помещают ни одного. Более того, относительно всех констант можно сказать однозначно, что они вычисляются путём помещения в стек собственной копии.

Поскольку класс `PolizElem` имеет виртуальный метод, следует описать и виртуальный деструктор. В самом классе `PolizElem` он, естественно, будет пустым.

Наконец, для удобства работы со стеком можно добавить в класс `PolizElem` статические (т.е. не использующие текущий объект) методы `Push` и `Pop`. В итоге заголовок класса примет следующий вид:

```
class PolizElem {
public:
    virtual ~PolizElem() {}
    virtual void Evaluate(PolizItem **stack,
```

```
                              PolizItem **cur_cmd) const = 0;
protected:
    static void Push(PolizItem **stack, PolizElem *elem);
    static PolizElem* Pop(PolizItem **stack);
};
```

Забегая вперед, порекомендуем для методов `Push` и `Pop` следующие соглашения: при вызове `Push` в стек помещается непосредственно объект, указанный параметром `elem`. (т.е. предполагается, что соответствующая копия уже создана вызывающим), а при возврате из `Pop` возвращается указатель на объект, на который больше нигде указателей нет (можно заметить, что тут копию можно не создавать). Причины для принятия именно таких соглашений вскоре станут ясны.

Методы промежуточных классов Как уже говорилось, для классов, представляющих константы, необходима возможность создания копии объекта. Этого можно добиться, введя в классе `PolizConst` метод `Clone`:

```
virtual PolizElem* Clone() const = 0;
```

Метод следует объявить чисто виртуальным, т.к. мы не можем задать правило копирования, пригодное для всех констант одновременно. Реальное содержание этот метод получит в классах, описывающих конкретные виды констант.

Теперь вспомним, что вычисление константы заключается в помещении в стек её копии. Таким образом, метод `Evaluate` оказывается одинаковым для всех констант и может быть реализован в классе `PolizConst` с тем, чтобы в его потомках переписывать этот метод было не нужно. Реализация метода `PolizConst::Evaluate` может выглядеть так:

```
void PolizConst::Evaluate(PolizItem **stack,
                          PolizItem **cur_cmd) const
{
    Push(stack, Clone());
    *cur_cmd = (*cur_cmd)->next;
}
```

Те из операций ПОЛИЗа, которые не требуют вмешательства в указатель текущей команды (т.е. все, кроме двух операций перехода) объединим в класс `PolizFunction` и опишем для него метод

```
virtual PolizElem* EvaluateFun(PolizItem **stack) const = 0;
```

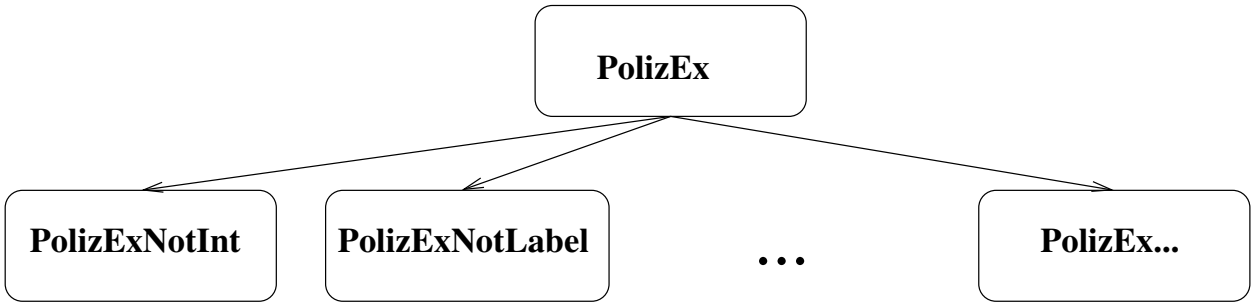

Рис. 4: Классы исключений интерпретатора ПОЛИЗа

От метода `Evaluate` этот новый метод отличается отсутствием параметра `cur_cmd` и наличием возвращаемого значения. Предполагаем, что этот метод осуществляет только изъятие из стека операндов, а операцию помещения в стек результата будет производить функция `PolizFunction::Evaluate`. Если операция не производит результата (это касается операции присваивания и операторов игровых действий), метод `EvaluateFun` должен будет возвращать нулевой указатель. С учетом сказанного реализация метода `PolizFunction::Evaluate` может выглядеть так:

```
void PolizConst::Evaluate(PolizItem **stack,
                          PolizItem **cur_cmd) const
{
    PolizElem *res = EvaluateFun(stack);
    if(res) Push(stack, res);
    *cur_cmd = (*cur_cmd)->next;
}
```

Для эстетов можно предложить еще один шаг – описать абстрактные классы `PolizFun0`, `PolizFun1` и `PolizFun2` для представления функций, получающих соответственно нуль, один и два аргумента[16]. Тогда при реализации конкретных функций операции со стеком не понадобятся.

4.5.4 Исключения

Для обработки исключительных ситуаций при интерпретации ПОЛИЗа вам понадобятся классы, описывающие конкретные случаи исключений.

В дальшейших примерах мы используем классы `PolizExNotLabel` и `PolizExNotNumber`, конструкторы которых получают на вход адрес объекта класса `PolizElem`; предполагается, что в конструктор будет передаваться

[16]Функции бóльшего числа аргументов в нашей задаче не встречаются

адрес объекта, ставшего причиной ошибки. Рекомендуется наследовать эти и другие классы исключений от некоего общего предка, например, `PolizEx` (см. рис. 4).

4.5.5 Примеры реализаций конкретных классов

В этом параграфе мы приведём реализацию нескольких классов иерархии. Реализацию остальных классов предоставим читателю произвести самостоятельно.

Прежде всего напомним, что узнать, действительно ли данный указатель, имеющий тип "указатель на класс-предок", указывает на объект потомок нужного типа, проще всего с помощью оператора `dynamic_cast`. Так, если переменная `p` имеет тип `PolizElem *`, выражение

```
dynamic_cast<PolizInt*>(p)
```

будет равно нулю, если `p` указывает на объект, не являющийся экземпляром `PolizInt`. Если же `p` действительно указывает на объект класса `PolizInt` или его потомка, выражение вернёт адрес типа `PolizInt *`, равный значению указателя `p`.

Константы Для начала опишем класс `PolizInt` как один из самых простых.

```
class PolizInt : public PolizConst {
    int value;
public:
    PolizInt(int a) { value = a; }
    virtual ~PolizInt() {}
    virtual PolizElem* Clone() const
        { return new PolizInt(value); }
    int Get() const { return value; }
};
```

Другие константы описываются совершенно аналогично. На всякий случай дадим также описание класса `PolizLabel`, т.к. этот класс вызывает у многих студентов непонимание.

```
class PolizLabel : public PolizConst {
    PolizItem* value;
public:
    PolizLabel(PolizItem* a) { value = a; }
```

```
    virtual ~PolizLabel() {}
    virtual PolizElem* Clone() const
        { return new PolizLabel(value); }
    PolizItem* Get() const { return value; }
};
```

Желающим можно предложить реализовать понятие константы в виде класса-шаблона `PolizGenericConst<class T>`, а приведенные выше классы оформить с помощью конструкции `typedef`.

Безусловный переход Описание класса операции "безусловный переход" может выглядеть примерно так:

```
class PolizOpGo : public PolizElem {
public:
    PolizOpGo() {}
    virtual ~PolizOpGo() {}
    void Evaluate(PolizItem **stack,
                  PolizItem **cur_cmd) const
    {
        PolizElem *operand1 = Pop(stack);
        PolizLabel *lab = dynamic_cast<PolizLabel*>(operand1);
        if(!lab) throw PolizExNotLabel(operand1);
        PolizItem *addr = lab->Get();
        *cur_cmd = addr;
        delete operand1;
    }
};
```

Обратите внимание на оператор `delete`. Нелишним будет напомнить, что в стеке хранятся копии констант; после извлечения с помощью функции `Pop` соответствующего объекта из стека он больше ни в каких структурах данных не фигурирует, единственный указатель на него – наш локальный `operand1`. Если не уничтожить этот объект, он окажется в "мусоре".

Операция сложения Реализацию класса `PolizFunPlus`, представляющего операцию ПОЛИЗа "сложение двух чисел", приведем в предположении, что этот класс наследуется непосредственно от `PolizFunction`.

```
class PolizFunPlus : public PolizFunction {
public:
```

```
    PolizFunPlus() {}
    virtual ~PolizFunPlus() {}
    PolizElem* EvaluateFun(PolizItem **stack) const
    {
        PolizElem *operand1 = Pop(stack);
        PolizLabel *i1 = dynamic_cast<PolizInt*>(operand1);
        if(!i1) throw PolizExNotInt(operand1);
        PolizElem *operand2 = Pop(stack);
        PolizLabel *i2 = dynamic_cast<PolizInt*>(operand2);
        if(!i2) throw PolizExNotInt(operand2);
        int res = i1->Get() + i2->Get();
        delete operand1;
        delete operand2;
        return new PolizInt(res);
    }
};
```

4.6 Еще о переводе в ПОЛИЗ

Перевод текста на вашем языке программирования в ПОЛИЗ лучше всего осуществлять на этапе синтаксического анализа с помощью действий, вставленных в грамматику.

У студентов часто возникают затруднения с пониманием того, каким образом происходит соответствующий перевод.

В этом параграфе мы опишем алгоритм перевода арифметических выражений в ПОЛИЗ с помощью стека операций.

Для перевода арифметического выражения в ПОЛИЗ будет просматривать выражение слева направо, выписывая по мере возможности элементы очередные ПОЛИЗа. В процессе этого будем использовать вспомогательный стек, в который будут в некоторых случаях помещаться символы операций и *открывающие* круглые скобки. Алгоритм устроен таким образом, что операнды (константы и переменные), а также закрывающие круглые скобки в стек никогда не попадают.

1. Итак, в начале алгоритма делаем текущей первую позицию исходного выражения. ПОЛИЗ считаем пустым. Стек очищаем и заносим в него открывающую круглую скобку.

2. Теперь рассматриваем текущий элемент выражения (пока таковые есть).

- Если этот элемент – операнд (переменная или константа), выписываем его в качестве очередного элемента ПОЛИЗа.
- Если этот элемент – открывающая скобка, заносим ее в стек.
- Если этот элемент – закрывающая фигурная скобка, извлекаем элементы из стека и выписываем их в качестве очередных элементов ПОЛИЗа до тех пор, пока на вершине стека не окажется открывающая скобка. Ее также извлекаем, но в ПОЛИЗ не записываем.
- Наконец, если этот элемент – символ операции, то:
  - если на вершине стека находится открывающая скобка, либо если приоритет операции на вершине стека *меньше* приоритета текущей операции, заносим текущую операцию в стек;
  - если, напротив, на вершине стека находится операция, имеющая *такой же или более высокий* приоритет, чем текущая, то извлекаем операцию из стека, выписываем извлеченную из стека операцию в качестве очередного элемента ПОЛИЗа, а текущую операцию снова сравниваем (возможно, что из стека будет в итоге извлечено больше одной операции). Когда операции такого же или более высокого приоритета на стеке кончились, заносим текущую операцию в стек.

3. Когда выражение полностью считано (больше нерассмотренных элементов нет), извлекаем элементы из стека и выписываем их в качестве очередных элементов ПОЛИЗа до тех пор, пока на вершине стека не окажется открывающая скобка. Ее также извлекаем, но в ПОЛИЗ не записываем.

4. Проверяем, что стек пуст. Если он не пуст, это свидетельствует о наличии в выражении незакрытой скобки. Заметим, что, если стек опустел раньше, чем кончилось выражение,– это, напротив, означает, что в выражении больше закрывающих скобок, чем открывающих.

Например, трансляция выражения $a - b + c * d$ будет происходить следующим образом:

— Помещаем в стек скобку
— выписываем операнд a
— помещаем в стек минус
— выписываем операнд b
— поскольку приоритет плюса не выше приоритета минуса, извлекаем из стека и выписываем минус (теперь ПОЛИЗ содержит $a\, b-$)

— поскольку на вершине стека опять скобка, заносим в стек плюс
— выписываем операнд c (теперь ПОЛИЗ содержит $a\,b - c$)
— поскольку на вершине стека сейчас плюс, а очередной элемент выражения – знак умножения, и поскольку умножение имеет более высокий приоритет, чем сложение, заносим умножение в стек (теперь стек содержит умножение, плюс и скобку)
— выписываем операнд d, получаем ПОЛИЗ $a\,b - c\,d$
— поскольку выражение закончилось, выбираем из стека по одному элементы, пока не встретим скобку; таким образом, выбираем и выписываем сначала умножение, потом сложение и получаем ПОЛИЗ $a\,b - c\,d\,* +$
— извлекаем из стека скобку; поскольку после этого стек оказался пуст и наше выражение тоже пусто, трансляция прошла успешно.

4.7 Рекомендации по объектно-ориентированному проектированию

Чтобы оценить, насколько удачно вы спроектировали иерархию классов, ответьте на несколько вопросов:

— Есть ли в ваших классах общее поле, обозначающее тип объекта?
— Есть ли хотя бы в одном из классов открытое (public) поле (не функция)?
— Есть ли в ваших классах поля с модификатором static?
— Пришлось ли при работе с вашими классами (как внутри методов, так и вне их) хотя бы один раз использовать оператор `switch`?

Если хотя бы на один вопрос вы ответили положительно, стоит еще подумать над проектированием иерархии.

Список литературы

[1] Ч. Уэзерелл. Этюды для программистов. *пер. с английского.* М.:Мир, 1982.

[2] А. М. Робачевский. Операционная система UNIX. Изд-во «BHV–Санкт-Петербург», Санкт-Петербург, 1997.

[3] У. Р. Стивенс. UNIX. Разработка сетевых приложений. М.:Изд-во «Питер», 2004.

[4] Г. Буч. Объектно-ориентированный анализ и проектирование с примерами приложений на C++. Второе издание. М.:Изд-во «Бином», 1999.

[5] И. А. Волкова, Т. В. Руденко. Формальные грамматики и языки. Элементы теории трансляции. Второе издание. М.: Издательский отдел ВМиК МГУ, 1999.

[6] William H. Press, Brian P. Flannery, Saul A. Teukolsky, William T. Vetterling. Numerical Recipes in C: The Art of Scientific Computing. New York: Cambridge University Press, 1992 (2nd ed.)

Содержание

Введение **3**

1 Игра “Менеджмент” **4**
1.1 Общие сведения 4
1.2 Порядок игры 4
1.3 Обстановка на рынке 5
1.4 Проведение аукционов 6

2 Реализация серверно-сетевой части **8**
2.1 Постановка задачи 8
2.2 Организация TCP-сервера 9
2.2.1 Создание сокета 10
2.2.2 Связывание сокета с адресом 11
2.2.3 Ожидание и прием клиентских соединений 12
2.3 Мультиплексирование ввода-вывода 14
2.3.1 Методы организации многопользовательского сервера 14
2.3.2 Вызов select() 15
2.4 Прием и передача данных через сокеты 18
2.4.1 Чтение 18
2.4.2 Запись 19
2.4.3 Разрыв соединения и обработка разрыва 21
2.5 Организация программы-клиента 22
2.5.1 Установление соединения 22
2.5.2 Встречные потоки данных и их обработка 23
2.6 Дополнительные сведения 25
2.6.1 Подробнее о порядке байт в целых числах 25
2.6.2 Как избежать “залипания” TCP-порта по завершении сервера 26
2.7 Рекомендации по тестированию 27

3 Программирование логики игры **30**
3.1 Общие сведения 30
3.2 Протокол взаимодействия с клиентом 30
3.2.1 Набор команд 31
3.2.2 Выдаваемые сообщения 32
3.3 Структуры данных 33
3.4 О таблице смены уровней рынка 34
3.5 О проведении аукционов 35
3.6 Диалоги с пользователем 36

3.7 Рекомендации по тестированию 38

4 Программируемый робот **39**
4.1 Постановка задачи 39
4.2 Пример входного языка робота 40
4.2.1 Общее описание 40
4.2.2 Арифметика. Выражения 40
4.2.3 Переменные. Массивы. Присваивания 41
4.2.4 Условный оператор 42
4.2.5 Встроенные функции для получения игровой информации 42
4.2.6 Встроенные операторы для совершения игровых действий 43
4.2.7 Оператор отладочной печати 44
4.2.8 Пример программы-сценария 44
4.3 Клиентская часть программы 45
4.4 Лексический и синтаксический анализ 46
4.4.1 Реализация лексического анализатора 48
4.4.2 Синтаксический анализатор 50
4.5 Интерпретация. ПОЛИЗ 51
4.5.1 Примерная структура данных 51
4.5.2 Типы элементов ПОЛИЗа 52
4.5.3 Методы 54
4.5.4 Исключения 57
4.5.5 Примеры реализаций конкретных классов 58
4.6 Еще о переводе в ПОЛИЗ 60
4.7 Рекомендации по объектно-ориентированному проектированию 62

Литература **63**