Оглавление

Об этом сборнике

© Dmitry Ponyatov <dponyatov@gmail.com>

В этот сборник (блогбук) я пишу отдельные статьи и переводы, сортированные только по общей тематике, и добавляю их, когда у меняв очередной раз зачешется LaTeX.

Это сборник черновых материалов, которые мне лень компоновать в отдельные книги, и которые пишутся просто по желанию “чтобы было”. Заказчиков на подготовку учебных материалов подобного типа нет, бо̀льшая часть только на этапе освоения мной самим, просто хочется иметь некое слабоупорядоченное хранилище наработок, на которое можно дать кому-то ссылку.

Часть I

ML & функциональное программирование

Глава 1

Основы Standard ML © Michael P. Fourman

http://homepages.inf.ed.ac.uk/mfourman/teaching/mlCourse/notes/L01.pdf

1.1 Введение

ML обозначает “MetaLanguage”: МетаЯзык. У Robin Milner была идея создания языка программирования, специально адаптированного для написания приложений для обработки логических формул и доказательств. Этот язык должен быть **метаязыком** для манипуляции объектами, представляющими формулы на логическом **объектном языке**.

Первый *ML* был метаязыком вспомогательного пакета автоматических доказательств Edinburgh LCF. Оказалось что метаязык Милнера, с некоторыми дополнениями и уточнениями, стал инновационным и универсальным языком программирования общего назначения. Standard ML (SML) является наиболее близким потомком оригинала, другой — CAML, Haskell является более дальним родственником. В этой статье мы представляем язык SML, и рассмотрим, как он может быть использован для вычисления некоторых интересных результатов с очень небольшим усилием по программированию.

Для начала, вы считаете, что программа представляет собой последовательность команд, которые будут выполняться компьютером. Это неверно! Предоставление последовательности инструкций является лишь одним из способов программирования компьютера. Точнее сказать, что **программа — это текст *спецификации вычисления***. Степень, в которой этот текст можно рассматривать как последовательность инструкций, изменяется в разных языках программирования. В этих заметках мы будем писать программы на языке ML, который не является столь явно императивным, как такие языки, как Си и Паскаль, в описании мер, необходимых для выполнения требуемого вычисления. Во многих отношениях ML **проще** чем Паскаль и Си. Тем не менее, вам может потребоваться некоторое время, чтобы оценить это.

ML в первую очередь функциональный язык: большинство программ на ML лучше всего рассматривать как спецификацию **значения**, которое мы хотим вычислить, без явного описания примитивных шагов, необходимых для достижения этой цели. В частности, мы не будем описывать, и вообще беспокоиться о способе, каким значения, хранимые где-то в памяти, изменяются по мере выполнения программы. Это позволит нам сосредоточиться на **организации** данных и вычислений, не втягиваясь в детали внутренней работы самого вычислителя.

В этом программирование на ML коренным образом отличается от тех приемов, которыми вы привыкли пользоваться в привычном императивном языке. **Попытки транслировать ваши программисткие привычки на ML неплодотворны — сопротивляйтесь этому искушению!**

Мы начнем этот раздел с краткого введения в небольшой фрагмент на ML. Затем мы используем этот фрагмент, чтобы исследовать некоторые функции, которые будут полезны в дальнейшем. Наконец, мы сделаем обзор некоторых важных аспектов ML.

Крайне важно пробовать эти примеры на компьютере, когда вы читаете этот текст.[1]

---

[1] Пользовательский ввод завершается точкой с запятой ";". В большинстве систем, ";" должна завершаться нажатием [Enter]/[Return], чтобы сообщить системе, что надо послать строку в ML. Эти примеры тестировались на системе Abstract Hardware Limited's Poly/ML. В **Poly/ML** запрос ввода символ $>$ или, если ввод неполон — #.

Примечение переводчика Для целей обучения очень удобно использовать онлайн среды, не требующие установки программ, и доступные в большинстве браузеров на любых мобильных устройствах. В качестве рекомендуемых online реализаций Standrard ML можно привести следующие:

CloudML https://cloudml.blechschmidt.saarland/
описан в блогпосте B. Blechschmidt как онлайн-интерпретатор диалекта Moscow ML

TutorialsPoint SML/NJ http://www.tutorialspoint.com/execute_smlnj_online.php

Moscow ML (**offline**) http://mosml.org/ реализация Standart ML

- Сергей Романенко, Келдышевский институт прикладной математики, РАН, Москва
- Claudio Russo, Niels Kokholm, Ken Friis Larsen, Peter Sestoft
- используется движок и некоторые идеи из Caml Light © Xavier Leroy, Damien Doligez.
- порт на MacOS © Doug Currie.

Глава 2

Programming in Standard ML'97

http://homepages.inf.ed.ac.uk/stg/NOTES/

© Stephen Gilmore
Laboratory for Foundations of Computer Science
The University of Edinburgh

Часть II

Язык bI

Глава 3

DLR: Dynamic Language Runtime

DLR: Dynamic Language Runtime — может использоваться как runtime-ядро для реализации динамических языков, или только в качестве библиотеки хранилища данных

синтаксический парсер для разбора текстовых данных, файлов конфигурации, скриптов и т.п., необязателен. В результате разбора формируется синтаксическое дерево из динамических объектов DLR. По реализации может быть

конфигурируемым в runtime добавление/изменение/удаление правил правил грамматики в процессе работы программы

статическим неизменный синтаксис, реализация в виде внешнего модуля, в самом простом случае достаточно использования **flex**/**bison**

библиотека динамических типов данных выполняет функции хранения данных, может быть реализована

в *Lisp*-стиле базовый набор скаляров 4.2 (символы, строки и числа) и тип **cons-ячейка** позволяющий конструировать составные структуры данных

bI-**стиль** универсальный символьный тип 4.1, позволяющий хранить как скаляры, так и вложенные элементы; в базовый тип **AST** заложено хранение типа данных `tag`, его значения `value`, и два способа вложенных хранилищ: плоский упорядоченный список `nest` и именованный неупорядоченный со строковыми ключами `pars`.

От базового символьного типа наследуются

скаляры символ, строка, несколько вариантов чисел (целые, плавающие, машинные, комплексные)[1]

композиты структуры данных и объекты

функционалы объекты, для которых определен *оператор аппликации*

библиотека операций над данными для преобразования данных и символьных вычислений на списках, деревьях, комбинаторах и т.п.

Lisp стандартная библиотека функций языка *Lisp*

bI каждый тип данных имеет набор унарных и бинарных *операторов*, реализованых в виде виртуальных методов классов

подсистема ООП реализация механизмов ООП, наследования от класса и объекта-инстанса, вывод типов, преобразование объектных моделей

реализация механизмов функциональных языков хвостовая рекурсия, pattern matching, динамическая компиляция, автоматическое распараллеливание на map/reduce

менеджер памяти со сборщиком мусора

динамический компилятор функциональных типов — через библиотеку JIT LLVM

[1] критерием скалярности можно считать возможность распознавания элемента данных лексером

статический компилятор

в объектный код через LLVM
кодогенератор C_+^+

Расширенный функционал

подсистема облачных вычислений и кластеризации расширение DLR на кластера: распределение объектов и процессов между вычислительными узлами. Варианты кластера с высокой связностью[2], Beowulf[3] с постоянным составом, интернет-облака с переменным составом: узлы асинхронно подключаются/отключаются, гомо/гетерогенные: по аппаратной платформе узлов и ОС/среде на каждом узле. Распределедение вычислений на одно- и многопроцессорных SMP-системах[4]

прикладные библиотеки GUI, CAD/CAM/EDA, численные методы, цифровая обработка сигналов, сетевые сервера и протоколы,...

подсистема кросс-трансляции между ходовыми языками программирования (C++, JavaScript, Python, PHP, Паскакаль) через связку: парсер входного языка → система типов DLR → кодогенератор выходного языка

интерактивная объектная среда а-ля *SmallTalk* с виджетами и функционалом GUI, CAD, IDE и визуализациии данных

сервер приложений обслуживающий тонких браузерных клиентов по HTTP/JS

---

[2] аппаратная разделяемая память через сеть InfiniBand — "Сергей Королев"
[3] компьютеры общего назначени (офисные) с передачей сообщений по Gigabit Ethernet
[4] многопоточные вычисления на одном многоядерном узле

Глава 4

Система динамических типов

4.1 sym: символ = Абстрактный Символьный Тип /AST/

Использование класса **Sym** и виртуально наследованных от него классов, позволяет реализовать на C^{+}_{+} хранение и обработку **любых** данных в виде деревьев[1]. Прежде всего этот ***символьный тип*** применяется при разборе текстовых форматов данных, и текстов программ. **Язык** *bI* **построен как интерпретатор AST, примерно так же как язык** *Lisp* **использует списки**.

```
1 // ================================================== ABSTRACT SYMBOLIC TYPE (AST)
2 struct Sym {
```

тип (класс) и значение элемента данных

```
1 // ------------------------------------------------------------------------------
```

[1] в этом ACT близок к традиционной аббревиатуре AST: Abstract Syntax Tree

```
2     string tag;                          // data type / class
3     string val;                          // symbol value
```

конструкторы (токен используется в лексере)

```
1 // ------------------------------------------------------------ constructors
2     Sym(string,string);                  // <T:V>
3     Sym(string);                         // token
```

Хранение вложенных элементов реализовано через указатели на базовый тип **Sym**. Благодаря виртуальному наследованию и использованию RTTI, этими указателями можно пользоваться для работы с любыми другими наследованными типами данных[2]

AST может хранить (и обрабатывать) вложенные элементы

```
1 // ------------------------------------------------------------ nest[]ed elements
2     vector<Sym*> nest;
3     void push(Sym*);
4     void pop();
```

параметры (и поля класса)

```
1 // ------------------------------------------------------------ par{}ameters
2     map<string,Sym*> pars;
3     void par(Sym*);                      // add parameter
```

вывод дампа объекта в текстовом формате

```
1 // ------------------------------------------------------------ dumping
```

[2] числа, списки, высокоуровневые и скомпилированные функции, элементы GUI,..

```
2     virtual string dump(int depth=0);    // dump symbol object as text
3     virtual string tagval();             // <T:V> header string
4     string tagstr();                     // <T:'V'> Str-like header string
5     string pad(int);                     // padding with tree decorators
```

Операции над *символами* выполняются через использование набора *операторов*:

вычисление объекта

```
1 // ------------------------------------------------------ compute (evaluate)
2     virtual Sym* eval();
```

операторы

```
1  // ------------------------------------------------------------ operators
2      virtual Sym* str();                 // str(A)    string representation
3      virtual Sym* eq(Sym*);              // A = B     assignment
4      virtual Sym* inher(Sym*);           // A : B     inheritance
5      virtual Sym* member(Sym*);          // A % B,C   named member (class slot)
6      virtual Sym* at(Sym*);              // A @ B     apply
7      virtual Sym* add(Sym*);             // A + B     add
8      virtual Sym* div(Sym*);             // A / B     div
9      virtual Sym* ins(Sym*);             // A += B    insert
10 };
```

4.2 Скаляры

4.2.1 str: строка

4.2.2 int: целое число

4.2.3 hex: машинное hex

4.2.4 bin: бинарная строка

4.2.5 num: число с плавающей точкой

4.3 Композиты

4.3.1 list: плоский список

4.3.2 cons: cons-пара и списки в *Lisp*-стиле

4.4 Функционалы

4.4.1 op: оператор

4.4.2 fn: встроенная/скомпилированная функция

4.4.3 lambda: лямбда

Глава 5

Программирование в свободном синтаксисе: FSP

5.1 Типичная структура проекта FSP: *lexical skeleton*

Скелет файловой структуры FSP-проекта = lexical skeleton = skelex

Создаем проект **prog** из командной строки (*Windows*):

```
1 mkdir prog
2 cd prog
3 touch src.src log.log ypp.ypp lpp.lpp hpp.hpp cpp.cpp Makefile bat.bat .gitignore
4 echo gvim -p src.src log.log ... Makefile bat.bat .gitignore >> bat.bat
5 bat
```

Создали каталог проекта, сгенерили набор пустых файлов (см. далее), и запуститили батник-hepler который запустит **(g)Vim**.

Для пользователей GitHub `mkdir` надо заменить на

```
git clone -o gh git@github.com:yourname/prog.git
cd prog
git gui &
...
```

src.src		исходный текст программы на вашем скриптовом языке
log.log		лог работы ядра *bI*
ypp.ypp	**flex**	парсер **??**
lpp.lpp	**bison**	лексер **??**
hpp.hpp	C_+^+	заголовочные файлы **??**
cpp.cpp	C_+^+	код ядра **??**
Makefile	**make**	зависимости между файлами и команды сборки (для утилиты **make**)
bat.bat	*Windows*	запускалка **(g)Vim ??**
.gitignore	**git**	список масок временных и производных файлов **??**

5.1.1 Настройки (g)Vim

При использовании редактора/IDE **(g)Vim** удобно настроить сочетания клавиш и подсветку **синтаксиса вашего скриптовго языка** так, как вам удобно. Для этого нужно создать несколько файлов конфигурации .vim: по 2 файла[1] для каждого диалекта скрипт-языка[2], и привязать их к расширениям через

[1] (1) привязка расширения файла и (2) подсветка синтаксиса

[2] если вы пользуетесь сильно отличающимся синтаксисом, но скорее всего через какое-то время практики FSP у вас выработается один диалект для всех программ, соответсвующий именно вашим вкусам в синтаксисе, и в этом случае его нужно будет описать только в файлах /.vim/(ftdetect|syntax).vim

dot-файлы **(g)Vim** в вашем домашнем каталоге. Подробно конфигурирование **(g)Vim** см. 14.

filetype.vim	**(g)Vim**	привязка расширений файлов (.src .log) к настройкам **(g)Vi**
syntax.vim	**(g)Vim**	синтаксическая подсветка для скриптов
/.vimrc	*Linux*	настройки для пользователя
/vimrc	*Windows*	
/.vim/ftdetect/src.vim	*Linux*	привязка команд к расширению .src
/vimfiles/ftdetect/src.vim	*Windows*	
/.vim/syntax/src.vim	*Linux*	синтаксис к расширению .src
/vimfiles/syntax/src.vim	*Windows*	

5.1.2 Дополнительные файлы

README.md	github	описание проекта для репоитория github
logo.png	github	логотип
logo.ico	*Windows*	
rc.rc	*Windows*	описание ресурсов: логотип, иконки приложения, меню,..

logo.png: Логотип

5.1.3 Makefile

Для сборки проекта используем команду **make** или **ming32-make** для *Windows*/**MinGW**. Прописываем в **Makefile** зависимости:

универсальный Makefile для fsp-проекта

```
1  log.log: ./exe.exe src.src
2      ./exe.exe < src.src > $@ && tail $(TAIL) $@
3  C = cpp.cpp ypp.tab.cpp lex.yy.c
4  H = hpp.hpp ypp.tab.hpp
5  CXXFILES += -std=gnu++11
6  ./exe.exe: $(C) $(H) Makefile
7      $(CXX) $(CXXFILES) -o $@ $(C)
8  ypp.tab.cpp: ypp.ypp
9      bison $<
10 lex.yy.c: lpp.lpp
11     flex $<
```

./exe.exe

префикс ./ требуется для правильной работы **ming32-make**, поскольку в *Linux* исполняемый файл может иметь любое имя и расширение, можем использовать .exe.

Для запуска транслятора используем простейший вариант — перенаправление потоков stdin/stdut на файлы, в этом случае не потребуется разбор параметров командной строки, и получим подробную трассировку выполнения трансляции.

переменные `C` и `H` задают набор исходный файлов ядра транслятора на C_+^+:

cpp.cpp реализация системы динамических типов данных, наследованных от символьного типа AST 4.1. Библиотека динамических классов языка *bI* II компактна, предоставляет достаточных

набор типов данных, и операций над ними. При необходимости вы можете легко написать свое дерево классов, если вам достаточно только простого разбора.

hpp.hpp заголовочные файлы также используем из *bI* II: содержат декларации динамических типов и интерфейс лексического анализатора, которые подходят для всех проектов

ypp.tab.cpp ypp.tab.hpp C_{+}^{+} код синтаксического парсера, генерируемый утилитой **bison** 7.2

lex.yy.c код лексического анализатора, генерируемый утилитой **flex** 7.1

`CXXFLAGS += gnu++11` добавляем опцию диалекта C_{+}^{+}, необходимую для компиляции ядра *bI*

Глава 6

Синтаксический анализ текстовых данных

6.1 Универсальный Makefile

Универсальный Makefile сделан на базе 5.1.3, с добавлением переменной `APP` указывающий какой пример парсера следуует скомпилировать и выполнить.

Для хранения (и возможной обработки) отпарсенных данных используем ядро языка *bI* 4 — используем файлы **../bi/hpp.hpp** и **../bi/cpp.cpp**. Ядро **очень компактно**, но умеет работать со скалярными, составными и функциональными данными, и содержит минимальную реализацию *ядра динамического языка.*

Универсальный **Makefile**

```
1 APP = minimal
2 $(APP).log: ./$(APP).exe $(APP).src
3 	./$(APP).exe < $(APP).src > $@ && tail $(TAIL) $@
4 C = ../bi/cpp.cpp ypp.tab.cpp lex.yy.c
```

```
5  H = ../bi/hpp.hpp ypp.tab.hpp
6  CXXFILES += -I../bi -I. -std=gnu++11
7  ./$(APP).exe: $(C) $(H) minimal.mk
8      $(CXX) $(CXXFILES) -o $@ $(C)
9  ypp.tab.cpp: $(APP).ypp
10     bison -o $@ $<
11 lex.yy.c: $(APP).lpp
12     flex -o $@ $<
13
14 .PHONY: src
15 src: minimal.src comment.src string.src ops.src brackets.src
16
17 minimal.src: ../bi/cpp.cpp
18     head -n11 $< > $@
19 comment.src: ../bi/cpp.cpp
20     head -n11 $< > $@
21 string.src: ../bi/cpp.cpp
22     head -n11 $< > $@
23 ops.src: ../bi/cpp.cpp
24     head -n5 $< > $@
25 brackets.src: ../bi/cpp.cpp
26     head -n5 $< > $@
```

6.2 C_+^+ интерфейс синтаксического анализатора

```
extern int yylex();             // получить код следующиго токена, и yylval.o
extern int yylineno;            // номер текущей строки файла исходника
```

```
extern char* yytext;            // текст распознанного токена, asciiz
#define TOC(C,X) { yylval.o = new C(yytext); return X; }

extern int yyparse();           // отпарсить весь текущий входной поток токенов
extern void yyerror(string);    // callback вызывается при синтаксической ошибке
#include "ypp.tab.hpp"
```

6.3 Минимальный парсер

Рассмотрим минимальный парсер, который может анализировать файлы текстовых данных (например исходники программ), и вычленять из них последовательности символов, которые можно отнести к ***скалярам*** символ, строка и число. [1]

Лексер **minimal.lpp** /**flex**/

```
1 %{
2 #include "hpp.hpp"
3 %}
4 %option noyywrap
5 %option yylineno
6 %%
7 [a-zA-Z0-9_.]+         TOC(Sym,SYM)
8 %%
```

(../bi/)**hpp.hpp** содержит определения интерфейса лексера 6.2, и ядра языка *bI* 4 для хранения результатов разбора текстовых данных

[1] эти три типа можно назвать атомами computer science

`noyywrap` выключает использование функции `yywrap()`

`yylineno` включает отслеживание строки исходного файла, используется при выводе сообщений об ошибках. В минимальном парсере не используется, но требуется для сборки *bI*-ядра.

`%%..%%` набор правил группировки отдельных символов в элементы данных — ***токены***, правила задаются с помощью *регулярных выражений*

`TOC(Sym,SYM)` единственное правило, распознающее любые группы сиволов как класс **bi::sym**: латинские буквы, цифры и символы _ и . (точка)[2]

Парсер **minimal.ypp /bison/**

```
1  %{
2  #include "hpp.hpp"
3  %}
4  %defines %union { Sym*o; }          /* use universal bI abstract type */
5  %token <o> SYM STR NUM              /* symbol 'string' number */
6  %type <o> ex scalar                 /* expression scalar */
7  %%
8  REPL : | REPL ex { cout << $2->tagval(); } ;
9  scalar : SYM | STR | NUM ;
10 ex : scalar ;
11 %%
```

hpp.hpp заголовок аналогичен лексеру 6.3

[2] точка добавлена, так часто используется в именах файлов

`%defines %union` указывает какие типы данных могут храниться в узлах разобранного ***синтаксического дерева***. Поскольку мы используем *bI*-ядро, нам будет достаточно пользоваться только классами языка *bI*, прежде всего универсальным символьным типом AST 4.1 и его прозводными классами.

`%token` описывает токены, которые может возвращать лексер `??`, причем набор токенов должен быть согласованным между лексером и парсером[3]

`%type` описывает типы синтаксических выражений, которые может распознавать ***грамматика*** синтаксического анализатора,

`REPL` выражение, описывающее грамматику, аналогичную простейшему варианту цикла REPL: Read Eval Print Loop[4]. В нашем случае часть вычисления Eval не выполняется[5], а часть Print выполняется через метод `Sym.tagval()`, возвращающий котороткую строку вида `<класс:значение>` для найденного токена.

`ex` (expression) универсальное символьное выражение языка *bI*, в нашем случае оно должно представлять только `scalar`

`scalar` выражение, представляющиее только распознаваемые скаляры:

`SYM` символ,

`STR` строку **или**

`NUM` число[6]

[3] определение токенов генерируется в файл **ypp.tab.hpp**

[4] чтение/вычисление/вывод/повторить

[5] разобранное выражение не вычисляется, хотя используемое ядро *bI* и поддерживает такой функционал

[6] числа в грамматике языка *bI* по типам не делятся, токен соответствует как **int**, так и **num**

В качестве тестового исходника возьмем C_+^+ код ядра языка bI: **../bi/cpp.cpp**:

minimal.src: Тестовый исходник

```
#include "hpp.hpp"
#define YYERR "\n\n"<<yylineno<<":"<<msg<<"["<<yytext<<"]\n\n"
void yyerror(string msg) { cout<<YYERR; cerr<<YYERR; exit(-1); }
int main() { return yyparse(); }

Sym::Sym(string T, string V) { tag=T; val=V; }
Sym::Sym(string V):Sym("sym",V) {}

string Sym::tagval() { return "<"+tag+":"+val+">"; }
string Sym::tagstr() { return "<"+tag+":'"+val+"'>"; }
string Sym::pad(int n) { string S; for (int i=0;i<n;i++) S+='\t'; return S; }
```

minimal.log: Результат прогона

```
#<sym:include> "<sym:hpp.hpp>"
#<sym:define> <sym:YYERR> "\<sym:n>\<sym:n>"<<<sym:yylineno><<":"<<<sym:msg><<"["<<<sym:y
<sym:void> <sym:yyerror>(<sym:string> <sym:msg>) { <sym:cout><<<sym:YYERR>; <sym:cerr><<<s
<sym:int> <sym:main>() { <sym:return> <sym:yyparse>(); }

<sym:Sym>::<sym:Sym>(<sym:string> <sym:T>, <sym:string> <sym:V>) { <sym:tag>=<sym:T>; <sym
<sym:Sym>::<sym:Sym>(<sym:string> <sym:V>):<sym:Sym>("<sym:sym>",<sym:V>) {}

<sym:string> <sym:Sym>::<sym:tagval>() { <sym:return> "<"+<sym:tag>+":"+<sym:val>+">"; }
<sym:string> <sym:Sym>::<sym:tagstr>() { <sym:return> "<"+<sym:tag>+":'"+<sym:val>+"'>";
<sym:string> <sym:Sym>::<sym:pad>(<sym:int> <sym:n>) { <sym:string> <sym:S>; <sym:for> (<s
<sym:string> <sym:Sym>::<sym:dump>(<sym:int> <sym:depth>) { <sym:string> <sym:S> = "\<sym
```

```
13      <sym:return> <sym:S>; }
14
15 <sym:Sym>* <sym:Sym>::<sym:eval>() { <sym:return> <sym:this>; }
```

Как видно по логу **minimal.log**, все группы сиволов, соответствующих правилу лексера SYM6.3, распознались как объекты bI, остальные остались символами и попали в лог без изменений.

6.4 Добавляем обработку комментариев

В тестах программ и файлов конфигурации очень часто используются *комментарии*. В языке *Python*, bI и UNIX shell комментарием является все от символа # до конца строки.

Для обработки таких ***строчных комментариев*** достаточно добавить одно правило лексера, **обязательно первым правилом**:

Лексер со строчными комментариями

```
1 %{
2 #include "hpp.hpp"
3 %}
4 %option noyywrap
5 %option yylineno
6 %%
7  #[^\n]*                {}
8  [a-zA-Z0-9_.]+         TOC(Sym,SYM)
9 %%
```

Группа символов, начинающаяся с символа #, затем идет ноль или более `[]*` любых символов не равных `^` концу строки `\n`. Пустое тело правила: C_+^+ код в `{}` скобках — выполняется и ничего не делает.

Тело правила SYM — вызов макроса `TOC(C,X)`6.2, наоборот, при своем выполнении создает токен, и возвращает код токена `=SYM`.

comment.log: Результат прогона

```
1
2
3 <sym:void> <sym:yyerror>(<sym:string> <sym:msg>) { <sym:cout><<<sym:YYERR>; <sym:cerr><<<s
4 <sym:int> <sym:main>() { <sym:return> <sym:yyparse>(); }
5
6 <sym:Sym>::<sym:Sym>(<sym:string> <sym:T>, <sym:string> <sym:V>) { <sym:tag>=<sym:T>; <sym
7 <sym:Sym>::<sym:Sym>(<sym:string> <sym:V>):<sym:Sym>("<sym:sym>",<sym:V>) {}
8
9 <sym:string> <sym:Sym>::<sym:tagval>() { <sym:return> "<"+<sym:tag>+":"+<sym:val>+">"; }
10 <sym:string> <sym:Sym>::<sym:tagstr>() { <sym:return> "<"+<sym:tag>+":'"+<sym:val>+"'>"; }
11 <sym:string> <sym:Sym>::<sym:pad>(<sym:int> <sym:n>) { <sym:string> <sym:S>; <sym:for> (<s
```

Как видно из лога, из вывода исчезли первые 2 строки, начинающиеся на #, причем концы этих строк остались (но не были как-либо распознаны).

6.5 Разбор строк

Для разбора строк необходимо использовать лексер с применением ***состояний***. Строки имеют сильно отличающийся от основного кода синтаксис, и для его обработки нужно **переключать набор правил лексера**.

Лексер с состоянием для строк

```
1 %{
2 #include "hpp.hpp"
```

```
3  string LexString;    /* string parser buffer */
4  %}
5  %option noyywrap
6  %option yylineno
7  %x lexstring
8  %%
9  #[^\n]*              {}
10
11 \"                   {BEGIN(lexstring); LexString="";}
12 <lexstring>\"        {BEGIN(INITIAL); yylval.o = new Str(LexString); return STR;}
13 <lexstring>\n        {LexString+=yytext[0];}
14 <lexstring>.         {LexString+=yytext[0];}
15
16 [a-zA-Z0-9_.]+       TOC(Sym,SYM)
17 %%
```

`string LexString` строковая буферная переменная, накапливающая символы строки

`%x lexstring` создание отдельного состояния лексера `lexstring`

`INITIAL` основное состояние лексера

`<lexstring>\n` правило конца строки позволяет использовать многострочные строки[7]

`<lexstring>.` любой символ в состоянии `<lexstring>`

[7] символ конца строки не распознается метасимволом . (точка) в регулярном выражении, и требует явного указания

Лог разбора со строками

```
1
2
3 <sym:void> <sym:yyerror>(<sym:string> <sym:msg>) { <sym:cout><<<sym:YYERR>; <sym:cerr><<<s
4 <sym:int> <sym:main>() { <sym:return> <sym:yyparse>(); }
5
6 <sym:Sym>::<sym:Sym>(<sym:string> <sym:T>, <sym:string> <sym:V>) { <sym:tag>=<sym:T>; <syn
7 <sym:Sym>::<sym:Sym>(<sym:string> <sym:V>):<sym:Sym>(<str:'sym'>,<sym:V>) {}
8
9 <sym:string> <sym:Sym>::<sym:tagval>() { <sym:return> <str:'<'>+<sym:tag>+<str:':'>+<sym:
10 <sym:string> <sym:Sym>::<sym:tagstr>() { <sym:return> <str:'<'>+<sym:tag>+<str:':''>+<sym
11 <sym:string> <sym:Sym>::<sym:pad>(<sym:int> <sym:n>) { <sym:string> <sym:S>; <sym:for> (<s
```

Обратите внимание, что ранее попадавшие в лог строки в двойных кавычках, типа `"]\n\n"`, стали распознаваться как строковые токены `<str:']\n\n'>`. [8]

6.6 Добавляем операторы

Для разбора языков программирования необходима поддержка операторов, включим общепринятые одиночные операторы, операторы C_+^+ и bI. **Скобки различного вида тоже будет рассматривать как операторы.** Операторы реализованы в ядре bI как отдельный класс **op**, зададим пачку правил разбора операторов, создающих токены `TOC(Op,XXX)`:

Лексер с операторами

```
1 %{
2 #include "hpp.hpp"
```

[8] использованы 'одинарные кавычки' как в *Python*/*bI*

```
3  string LexString;    /* string parser buffer */
4  %}
5  %option noyywrap
6  %option yylineno
7  %x lexstring
8  %%
9  #[^\n]*              {}                /* # line comment */
10
11 \"                   {BEGIN(lexstring); LexString="";}
12 <lexstring>\"        {BEGIN(INITIAL); yylval.o = new Str(LexString); return STR;}
13 <lexstring>\n        {LexString+=yytext[0];}
14 <lexstring>.         {LexString+=yytext[0];}
15
16 [a-zA-Z0-9_.]+       TOC(Sym,SYM)      /* symbol */
17
18 \(                   TOC(Op,LB)        /* brackets */
19 \)                   TOC(Op,RB)
20 \[                   TOC(Op,LQ)
21 \]                   TOC(Op,RQ)
22 \{                   TOC(Op,LC)
23 \}                   TOC(Op,RC)
24
25 \+                   TOC(Op,ADD)       /* typical arithmetic operators */
26 \-                   TOC(Op,SUB)
27 \*                   TOC(Op,MUL)
28 \/                   TOC(Op,DIV)
29 \^                   TOC(Op,POW)
30                                        /* bI language specific */
```

```
\=                      TOC(Op,EQ)      /* assign */
\@                      TOC(Op,AT)      /* apply */
\~                      TOC(Op,TILD)    /* quote */
\:                      TOC(Op,COLON)   /* inheritance */

%%
```

Парсер с операторами

```
%{
#include "hpp.hpp"
%}
%defines %union { Sym*o; }          /* use universal bI abstract type */
%token <o> SYM STR NUM              /* symbol 'string' number */
%token <o> LB RB LQ RQ LC RC        /* brackets: () [] {} */
%token <o> ADD SUB MUL DIV POW      /* arithmetic operators: + - * / ^ */
%token <o> EQ AT TILD COLON         /* bi specific operators: = @ ~ : */
%type <o> ex scalar                 /* expression scalar */
%type <o> bracket operator
%%
REPL : | REPL ex { cout << $2->dump(); } ;
scalar : SYM | STR | NUM ;
ex : scalar | operator ;
bracket : LB | RB | LQ | RQ | LC | RC ;
operator :
    bracket
    | ADD | SUB | MUL | DIV | POW
    | EQ | AT | TILD | COLON
;
```

```
21 %%
```

Лог уже стал нечитаем, переключаемся на древовидный вывод через метод `Sym.dump()`.

Разбор с операторами

```
1
2
3
4 <sym:void>
5 <sym:yyerror>
6 <op:(>
7 <sym:string>
8 <sym:msg>
9 <op:)>
10 <op:{>
11 <sym:cout><<
12 <sym:YYERR>;
13 <sym:cerr><<
14 <sym:YYERR>;
15 <sym:exit>
16 <op:(>
17 <op:->
18 <sym:1>
19 <op:)>;
20 <op:}>
21
22 <sym:int>
23 <sym:main>
24 <op:(>
```

```
<op:)>
<op:{>
<sym:return>
<sym:yyparse>
<op:(>
<op:)>;
<op:}>
```

6.7 Обработка вложенных структур (скобок)

Обработка вложенных структур возможна только парсером, лексер оставляем без изменений. Хранение вложенных структур в виде дерева — главная фича типа *bI* AST4.1. Заменяем грамматическое выражение `bracket` на отдельные выражения для скобок:

Парсер со скобками

```
%{
#include "hpp.hpp"
%}
%defines %union { Sym*o; }      /* use universal bI abstract type */
%token <o> SYM STR NUM          /* symbol 'string' number */
%token <o> LB RB LQ RQ LC RC    /* brackets: () [] {} */
%token <o> ADD SUB MUL DIV POW  /* arithmetic operators: + - * / ^ */
%token <o> EQ AT TILD COLON     /* bi specific operators: = @ ~ : */
%token <o> SCOLON GR LS
%type <o> ex scalar             /* expression scalar */
%type <o> operator
%%
```

```
13 REPL : | REPL ex { cout << $2->dump(); } ;
14 scalar : SYM | STR | NUM ;
15 ex :
16       ex ex                    { $$=$1; $$->push($2); }
17     | scalar | operator
18     | LB ex RB                 { $$=new Sym("()"); $$->push($2); }
19     | LB  RB                   { $$=new Sym("()"); }
20     | LQ ex RQ                 { $$=new Sym("[]"); $$->push($2); }
21     | LC ex RC                 { $$=new Sym("{}"); $$->push($2); }
22 ;
23 operator :
24     ADD | SUB | MUL | DIV | POW
25     | EQ | AT | TILD | COLON
26     | SCOLON | GR | LS
27 ;
28 %%
```

Разбор со скобками

```
1
2
3
4
5
6
7 <sym:void>
8     <sym:yyerror>
9         <sym:()>
10            <sym:string>
```

```
                <sym:msg>
            <sym:{}>
                <sym:cout>
                    <op:<>
                        <op:<>
                            <sym:YYERR>
                                <op:;>
                                    <sym:cerr>
                                        <op:<>
                                            <op:<>
                                                <sym:YYERR>
                                                    <op:;>
                                                        <sym:exit>
                                                            <sym:()>
                                                                <op:->
                                                                    <sym:1>
                                                                <op:;>
                <sym:int>
                    <sym:main>
                        <sym:()>
                            <sym:{}>
                                <sym:return>
                                    <sym:yyparse>
                                        <sym:()>
                                            <op:;>
```

Глава 7

Синтаксический анализатор

Синтаксис языка *bI* был выбран алголо-подобным, более близким к современным императивным языкам типа C_+^+ и *Python*. Использование типовых утилит-генераторов позволяет легко описать синтаксис с инфиксными операторами и скобочной записью для композитных типов 4.3, и не заставлять пользователя закапываться в клубок *Lisp*овских скобок.

Инфиксный синтаксис **для файлов конфигурации** удобен неподготовленным пользователям, а возможность определения пользовательских функций и объектная система, встроенная в ядро *bI*, дает богатейшие возможности по настройке и кастомизации программ.

Единственной проблемой с точки зрения синтаксиса для начинающего пользователя *bI* может оказаться отказ от скобок при вызове функций, применение оператора явной аппликации `@`, и функциональные наклонности самого *bI*, претендующего на звание универсального **объектного мета-языка** и **языка шаблонов**.

7.1 lpp.lpp: лексер /flex/

lpp.lpp

```
1  %{
2  #include "hpp.hpp"
3  %}
4  %option noyywrap
5  %option yylineno
6  S [\-\+]?
7  N [0-9]+
8  %%
9  #[^\n]*              {}                 /* line comment */
10
11                                                              /* == numbers == */
12 {S}{N}               TOC(Sym,NUM)       /* integer */
13 {S}{N}\.{N}          TOC(Sym,NUM)       /* floating point */
14 {S}{N}[eE]{S}{N}     TOC(Sym,NUM)       /* exponential */
15
16 [a-zA-Z0-9_.]+       TOC(Sym,SYM)       /* symbol */
17
18 \(                   TOC(Sym,SYM)                                 /* == brackets == */
19 \)                   TOC(Sym,SYM)
20 \[                   TOC(Sym,SYM)       /* [list] */
21 \]                   TOC(Sym,SYM)
22 \{                   TOC(Sym,SYM)       /* {lambda} */
23 \}                   TOC(Sym,SYM)
24                                                              /* == operators == */
25 \=                   TOC(Sym,SYM)       /* assign */
```

```
\@                      TOC(Sym,SYM)      /* apply */
\:                      TOC(Sym,SYM)      /* inherit */
\~                      TOC(Sym,SYM)      /* quote */

\+                      TOC(Sym,SYM)      /* arithmetics */
\-                      TOC(Sym,SYM)
\*                      TOC(Sym,SYM)
\/                      TOC(Sym,SYM)
\^                      TOC(Sym,SYM)
                                                        /* == drop == */
[ \t\r\n]+              {}                /* spaces */
.                       {}                /* undetected chars */
%%
```

7.2 ypp.ypp: парсер /bison/

ypp.ypp

```
%{
#include "hpp.hpp"
%}
%defines %union { Sym*o; }          /* use universal bI abstract type */
%token <o> SYM STR NUM              /* symbol 'string' number */
%type <o> ex scalar                 /* expression scalar */
%%
REPL : | REPL ex { cout << $2->dump(); } ;
scalar : SYM | STR | NUM ;
```

```
10 ex : scalar ;
11 %%
```

В качестве типа-хранилища для узлов синтаксического дерева идеально подходит базовый символьный тип *bI* 4.1, причем его применение в этом качестве рассматривалось как основное: гибкое представление произвольных типов данных. Собственно его название намекает.

В качестве токенов-скаляров логично выбираются SYMвол, STRока и число NUM[1]. Надо отметить, что в принципе можно было бы обойтись единственным SYM, но для дополнительного контроля грамматики полезно выделить несколько токенов: это позволит гарантировать что в определении класса **??** вы сможете использовать в качестве суперкласса и имен полей только символы. По крайне мере до момента, когда в очередном форке *bI* не появится возможность наследовать любые объекты.

[1] их можно вообще рассматривать как элементарные частицы Computer Science, правда к ним еще придется добавить PTR: божественный указатель

Часть III

em*Linux* для встраиваемых систем

Структура встраиваемого микро*Linux*а

syslinux Загрузчик

em*Linux* поставляется в виде двух файлов:

1. ядро `$(HW)$(APP).kernel`
2. сжатый образ корневой файловой системы `$(HW)$(APP).rootfs`

Загрузчик считывает их с одного из носителей данных, который поддерживается загрузчиком[2], распаковывает в память, включив защищенный режим процессора, и передает управление ядру *Linux*.

Для работы em*Linux* не требуются какие-либо носители данных: вся (виртуальная) файловая система располагается в ОЗУ. При необходимости к любому из каталогов корневой ФС может быть *подмонтирована* любая существующая дисковая или сетевая файловая система (FAT,NTFS,Samba,NFS,..), причем можно явно запретить возможность записи на нее, защитив данные от разрушения.

Использование rootfs в ОЗУ позволяет гарантировать защиту базовой ОС и пользовательских исполняемых файлов от внезапных выключений питания и ошибочной записи на диск. Еще большую защиту даст отключение драйверов загрузочного носителя в ядре. Если отключить драйвера SATA/IDE и грузиться с USB флешки, практически невозможно испортить основную установку ОС и пользовательские файлы на чужом компьютере.

kernel Ядро *Linux* 3.13.xx

[2] IDE/SATA HDD, CDROM, USB флешка, сетевая загрузка с BOOTP-сервера по Ethernet

ulibc Базовая библиотека языка Си

busybox Ядро командной среды UNIX, базовые сетевые сервера

дополнительные библиотеки

zlib сжатие/распаковка gzip
??? библиотека помехозащищенного кодирования
png библиотека чтения/записи графического формата .png
freetype рендер векторных шрифтов (TTF)
SDL полноэкранная (игровая) графика, аудио, джойстик

кодеки аудио/видео форматов: ogg vorbis, mp3, mpeg, ffmpeg/gsm

К базовой системе добавляются пользовательские программы ***/usr/bin***
и динамические библиотеки ***/usr/lib***.

Процедура сборки

Глава 8

clock: коридорные электронные часы = контроллер умного дурдома

Глава 9

gambox: игровая приставка

Часть IV

GNU Toolchain и C_+^+ для встраиваемых систем

Глава 10

Программирование встраиваемых систем с использованием GNU Toolchain [19]

10.1 Введение

Пакет компиляторов GNU toolchain широко используется при разработке программного обеспечения для встраиваемых систем. Этот тип разработки ПО также называют ***низкоуровневым***, ***standalone*** или ***bare metal*** программированием (на Си и C_{+}^{+}). Написание низкоуровневого кода на Си добавляет программисту новых проблем, требующих глубокого понимания инструмента разработчика — **GNU Toolchain**. Руководства разработчика **GNU Toolchain** предоставляют отличную информацию по инструментарию, но с

[1] © http://bravegnu.org/gnu-eprog/
[2] © https://github.com/ponyatov/gnu-eprog/blob/ru/gnu-eprog.asciidoc

точки зрения самого **GNU Toolchain**, чем с точки зрения решаемой проблемы. Поэтому было написано это руководство, в котором будут описаны типичные проблемы, с которыми сталкивается начинающий разработчик.

Этот учебник стремится занять свое место, объясняя использование **GNU Toolchain** с точки зрения практического использования. Надеемся, что его будет достаточно для разработчиков, собирающихся освоить и использовать **GNU Toolchain** в их embedded проектах.

В иллюстративных целях была выбрана встроенная система на базе процессорного ядра ARM, которая эмулируется в пакете **Qemu**. С таким подходом вы сможете освоить **GNU Toolchain** с комфортом на вашем рабочем компьютере, без необходимости вкладываться в "физическое" железо, и бороться со сложностями с его запуском. Учебник не стремиться обучить работе с архитектурой ARM, для этого вам нужно будет воспользоваться дополнительными книгами или онлайн-учебниками типа:

- ARM Assembler http://www.heyrick.co.uk/assembler/
- ARM Assembly Language Programming http://www.arm.com/miscPDFs/9658.pdf

Но для удобства читателя, некоторое множество часто используемых ARM-инструкций описано в приложении 10.17.

10.2 Настройка тестового стенда

В этом разделе описано, как настроить на вашей рабочей станции простую среду разработки и тестирования ПО для платформы ARM, используя **Qemu** и **GNU Toolchain**. **Qemu** это программный[3] эмулятор нескольких распространенных аппаратных платформ. Вы можете написать программу на ассемблере и C_{+}^{+}, скомпилировать ее используя **GNU Toolchain** и отладить ее в эмуляторе **Qemu**.

[3] для i386 — программно-аппаратный, использует средства виртуализации хост-компьютера

10.2.1 Qemu ARM

Будем использовать **Qemu** для эмуляции отладочной платы **Gumstix connex** на базе процессора PXA255. Для работы с этим учебником у вас должен быть установлен **Qemu** версии не ниже 0.9.1.

Процессор[4] PXA255 имеет ядро ARM с набором инструкций ARMv5TE. PXA255 также имеет в своем составе несколько блоков периферии. Некоторая периферия будет описана в этом курсе далее.

10.2.2 Инсталляция Qemu на *Debian GNU/Linux*

Этот учебник требует **Qemu** версии не ниже 0.9.1. Пакет **Qemu** доступный для современных дистрибутивов *Debian GNU/Linux*, вполне удовлетворяет этим условиям, и собирать свежий **Qemu** из исходников совсем не требуется[5]. Установим пакет командой:

```
$ sudo apt install qemu
```

10.2.3 Установка кросс-компилятора GNU Toolchain для ARM

Если вы предпочитаете простые пути, установите пакет кросс-компилятора командной

```
sudo apt install gcc-arm-none-eabi
```

или

1. Годные чуваки из CodeSourcery[6] уже давно запилили несколько вариантов **GNU Toolchain**ов для разных ходовых архитектур. Скачайте готовую бинарную бесплатную lite-сборку **GNU Toolchain-ARM**

[4] Точнее SoC: система-на-кристалле

[5] хотя может быть и очень хочется

[6] подразделение Mentor Graphics

2. Распакуйте tar-архив в каталог */toolchains*:

```
$ mkdir ~/toolchains
$ cd ~/toolchains
$ tar -jxf ~/downloads/arm-2008q1-126-arm-none-eabi-i686-pc-linux-gnu.tar.bz2
```

3. Добавьте bin-каталог тулчейна в переменную среды `PATH`.

```
$ PATH=$HOME/toolchains/arm-2008q1/bin:$PATH
```

4. Чтобы каждый раз не выполнять предыдущую команду, вы можете прописать ее в дот-файл **.bashrc**.

Для совсем упертых подойдет рецепт сборки полного комплекта кросс-компилятора из исходных текстов, описанный в 12.

10.3 Hello ARM

В этом разделе вы научитесь пользоваться arm-ассемблером, и тестировать вашу программу на голом железе — эмуляторе платы **connex** в **Qemu**.

Файл исходника ассемблерной программы состоит из последовательности инструкций, по одной на каждую строку. Каждая инструкция имеет формат (каждый компонент не обязателен):

```
<метка>:    <инструкция>          @ <комментарий>
```

метка — типичный способ пометить адрес инструкции в памяти. Метка может быть использована там, где требуется указать адрес, например как операнд в команде перехода. Метка может состоять из латинских букв, цифр[7], символов `_` и `$`.

[7] не может быть первым символом метки

комментарий начинается с символа `@` — все последующие символы игнорируются до конца строки

инструкция может быть инструкцией процессора или директивой ассемблера, начинающейся с точки "."

Вот пример простой ассемблерной программы 1 для процессора ARM, складывающей два числа:

Листинг 1: Сложение двух чисел

```
        .text
start:                          @ метка необязательна
        mov   r0, #5            @ загрузить в регистр r0 значение 5
        mov   r1, #4            @ загрузить в регистр r1 значение 4
        add   r2, r1, r0        @ сложить r0+r1 и сохранить в r2 (справа налево)

stop:   b stop                  @ пустой бесконечный цикл для останова выполнения
```

`.text` ассемблерная директива, указывающая что последующий код должен быть *ассемблирован* в ***секцию кода*** *`.text`* а не в секцию `.data`. *Секции* будут подробно описаны далее.

10.3.1 Сборка бинарника

Сохраните программу в файл **add.s** [8]. Для ассемблирования файла вызовите ассемблер **as**:

```
$ arm-none-eabi-as -o add.o add.s
```

[8] .s или .S стандарное расширение в мире OpenSource, указывает что это файл с программной на ассемблере

Опция `-o` указывает выходной файл с ***объектным кодом***, имеющий стандартное расширение `.o`[9].

Команды кросс-тулчейна всегда имеют префикс целевой архитектуры (target triplet), для которой они были собраны, чтобы предотвратить конфликт имен с хост-тулчейном для вашего рабочего компьютера. Далее утилиты **GNU Toolchain** будут использоваться без префикса для лучшей читаемости. **не забывайте добавлять arm-none-eabi-, иначе получите множество странных ошибок типа "unexpected command"**.

```
$ (arm-none-eabi-)as -o add.o add.s
$ (arm-none-eabi-)objdump -x add.o
```

вывод команды **arm-none-eabi-objdump -x**: ELF-заголовки в файле объектного кода

```
1  add.o:     file format elf32-littlearm
2  add.o
3  architecture: armv4, flags 0x00000010:
4  HAS_SYMS
5  start address 0x00000000
6  private flags = 5000000: [Version5 EABI]
7
8  Sections:
9  Idx Name          Size      VMA       LMA       File off  Algn
10   0 .text         00000010  00000000  00000000  00000034  2**2
11                   CONTENTS, ALLOC, LOAD, READONLY, CODE
12   1 .data         00000000  00000000  00000000  00000044  2**0
```

[9] и внутренний формат ELF (как завещал великий *Linux*)

```
13                      CONTENTS, ALLOC, LOAD, DATA
14   2 .bss             00000000  00000000  00000000  00000044  2**0
15                      ALLOC
16   3 .ARM.attributes  00000014  00000000  00000000  00000044  2**0
17                      CONTENTS, READONLY
18 SYMBOL TABLE:
19 00000000 l    d  .text  00000000 .text
20 00000000 l    d  .data  00000000 .data
21 00000000 l    d  .bss   00000000 .bss
22 00000000 l       .text  00000000 start
23 0000000c l       .text  00000000 stop
24 00000000 l    d  .ARM.attributes      00000000 .ARM.attributes
```

Секция `.text` имеет размер `Size=0x0010` =16 байт, и содержит **машинный код**:

машинный код из секции `.text`: **objdump -d**

```
1 add.o:     file format elf32-littlearm
2
3 Disassembly of section .text:
4
5 00000000 <start>:
6    0:   e3a00005    mov r0, #5
7    4:   e3a01004    mov r1, #4
8    8:   e0812000    add r2, r1, r0
9
10 0000000c <stop>:
11   c:   eafffffe    b   c <stop>
```

Для генерации **исполняемого файла**[10] вызовем ***линкер*** **ld**:

```
$ arm-none-eabi-ld -Ttext=0x0 -o add.elf add.o
```

Опять, опция `-o` задает выходной файл. `-Ttext=0x0` явно указывает адрес, от которого будут отсчитываться все метки, т.е. секция инструкций начинается с адреса `0x0000`. Для просмотра адресов, назначенных меткам, можно использовать команду `(arm-none-eabi-)nm` [11]:

```
ponyatov@bs:/tmp$ arm-none-eabi-nm add.elf
...
00000000 t start
0000000c t stop
```

* если вы забудете опцию -T, вы получите этот вывод с адресами `00008xxx` — эти адреса были заданы при компиляции **GNU Toolchain**-ARM, и могут не совпадать с необходимыми вам. Проверяйте ваши .elfы с помощью **nm** или **objdump**, если программы не запускаются, или **Qemu** ругается на ошибки (защиты) памяти.

Обратите внимание на ***назначение адресов*** для меток `start` и `stop`: адреса начанаются с 0x0. Это адрес первой инструкции. Метка `stop` находится после третьей инструкции. Каждая инструкция занимает 4 байта[12], так что `stop` находится по адресу $0xC_{hex} = 12_{dec}$. ***Линковка*** с другим ***базовым адресом*** `-Ttext=nnnn` приведет к сдвигу адресов, назначенных меткам.

---

[10] обычно тот же формат ELF.o, слепленный из одного или нескольких объектных файлов, с некоторыми модификациями см. опцию -T далее

[11] NaMes

[12] в множестве команд ARM-32, если вы компилируете код для микроконтроллера Cortex-Mx в режиме команд Thumb или Thumb2, команды 16-битные, т.е. 2 байта

```
$ arm-none-eabi-ld -Ttext=0x20000000 -o add.elf add.o
$ arm-none-eabi-nm add.elf
... clip ...
20000000 t start
2000000c t stop
```

Выходной файл, созданный **ld** имеет формат, который называется ***ELF***. Существует множество форматов, предназначенных для хранения выполняемого и объектного кода[13]. Формат ELF применяется для хранения машинного кода, если вы запускаете его в базовой ОС[14], но поскольку мы собираемся запускать нашу программу на bare metal[15], мы должны сконвертировать полученный .elf файл в более простой ***бинарный формат***.

Файл в *бинарном формате* содержит последовательность байт, начинающуюся с определенного адреса памяти, поэтому бинарный файл еще называют *образом памяти*. Этот формат типичен для утилит программирования флеш-памяти микроконтроллеров, так как все что требуется сделать — последовательно скопировать каждый байт из файла в FlashROM-память микроконтроллера, начиная с определенного начального адреса.[16]

Команда **GNU Toolchain objcopy** используется для конвертирования машинного кода между разными объектными форматами. Типичное использование:

```
$ objcopy -O <выходной_формат> <входной_файл> <выходной_файл>
```

Конвертируем **add.elf** в бинарный формат:

[13] можно отдельно отметить Microsoft COFF (объектные файлы .obj) и PE (.exe)cutable

[14] прежде всего “большой” или встраиваемый *Linux*

[15] голом железе

[16] Та же операция выполняется и для SoC-систем с NAND-флешем: записать бинарный образ начиная с некоторого аппаратно фиксированного адреса.

```
$ objcopy -O binary add.elf add.bin
```

Проверим размер полученного бинарного файла, он должен быть равен тем же 16 байтам[17]:

```
$ ls -al add.bin
-rw-r--r-- 1 vijaykumar vijaykumar 16 2008-10-03 23:56 add.bin
```

Если вы не доверяете **ls**, можно дизассемблировать бинарный файл:

```
ponyatov@bs:/tmp$ arm-none-eabi-objdump -b binary -m arm -D add.bin

add.bin:     file format binary

Disassembly of section .data:

00000000 <.data>:
   0:   e3a00005        mov     r0, #5
   4:   e3a01004        mov     r1, #4
   8:   e0812000        add     r2, r1, r0
   c:   eafffffe        b       0xc
ponyatov@bs:/tmp$
```

Опция `-b` задает формат файла, опция `-m` (machine) архитектуру процессора, получить полный список сочетаний `-b/-m` можно командной `arm-none-eabi-objdump -i`.

[17] 4 инструкции по 4 байта каждая

10.3.2 Выполнение в Qemu

Когда ARM-процессор сбрасывается, он начинает выполнять команды с адрсе 0x0. На плате Commnex установлен флеш на 16 мегабайт, начинающийся с адрес 0x0. Таким образом, при сбросе будут выполняться инструкции с начала флеша.

Когда **Qemu** эмулирует плату connex, в командной строке должен быть указан файл, который будет считаться образом флеш-памяти. Формат флеша очень прост — это побайтный образ флеша без каких-либо полей или заголовков, т.е. это тот же самый *бинарный формат*, описанный выше.

Для тестирования программы в эмуляторе Gumstix connex, сначала мы создаем 16-мегабайтный файл флеша, копируя 16M нулей из файла **/dev/zero** с помощью команды **dd**. Данные копируются 4Кбайтными блоками[18] (4096 x 4K):

```
$ dd if=/dev/zero of=flash.bin bs=4K count=4K
4096+0 записей получено
4096+0 записей отправлено
 скопировано 16777216 байт (17 MB), 0,0153502 c, 1,1 GB/c

$ du -h flash.bin
16M     flash.bin
```

Затем переписываем начало **flash.bin** копируя в него содержимое **add.bin**:

```
$ dd if=add.bin of=flash.bin bs=4K conv=notrunc
0+1 записей получено
0+1 записей отправлено
 скопировано 16 байт (16 B), 0,000173038 c, 92,5 kB/c
```

[18] опция bs= (blocksize)

После сброса процессор выполняет код с адреса 0x0, и будут выполняться инструкции нашей программы. Команда запуска **Qemu**:

```
$ qemu-system-arm -M connex -pflash flash.bin -nographic -serial /dev/null

QEMU 2.1.2 monitor - type 'help' for more information
(qemu)
```

Опция `-M connex` выбирает режим эмуляции: **Qemu** поддерживает эмуляцию нескольких десятков вариантов железа на базе ARM процессоров. Опция `-pflash` указывает файл образа флеша, который должен иметь определенный размер (16M). `-nographic` отключает эмуляцию графического дисплея (в отдельном окне). Самая важная опция `-serial /dev/null` подключает последовательный порт платы на **/dev/null**, при этом в терминале после запуска **Qemu** вы получите **консоль монитора**.

Qemu выполняет инструкции, и останавливается в бесконечном цикле на `stop`, выполняя команду `stop: b stop`. Для просмотра содержимого регистров процессора воспользуемся ***монитором***. Монитор имеет интерфейс командной строки, который вы можете использовать для контроля работы эмулируемой системы. Если вы запустите **Qemu** как указано выше, монитор будет доступен через stdio.

Для просмотра регистров выполним команду `info registers`:

```
(qemu) info registers
R00=00000005 R01=00000004 R02=00000009 R03=00000000
R04=00000000 R05=00000000 R06=00000000 R07=00000000
R08=00000000 R09=00000000 R10=00000000 R11=00000000
R12=00000000 R13=00000000 R14=00000000 R15=0000000c
PSR=400001d3 -Z-- A svc32
FPSCR: 00000000
```

Обратите внимание на значения в регистрах r00..r02: 4, 5 и ожидаемый результат 9. Особое значение для ARM имеет регистр r15: он является указателем команд, и содержит адрес текущей выполняемой машинной команды, т.е. `0x000c: b stop`.

10.3.3 Другие команды монитора

Несколько полезных команд монитора:

help	список доступных команд
quit	выход из эмулятора
xp /fmt addr	вывод содержимого физической памяти с адреса addr
system_reset	перезапуск

Команда `xp` требует некоторых пояснений. Аргумент `/fmt` указывает как будет выводиться содержимое памяти, и имеет синтаксис `<счетчик><формат><размер>`:

счетчик число элементов данных

size размер одного элемента в битах: b=8 бит, h=16, w=32, g=64

format определяет формат вывода:

- **x** hex
- **d** десятичные целые со знаком
- **u** десятичные без знака
- **o** 8ричные
- **c** символ (char)
- **i** инструкции ассемблера

Команда `xp` в формате `i` будет дизассемблировать инструкции из памяти. Выведем дамп с адреса 0x0 указав fmt=4iw: 4 — 4 , i — инструкции размером w=32 бита:

```
(qemu) xp /4wi 0x0
0x00000000:  e3a00005      mov  r0, #5  ; 0x5
0x00000004:  e3a01004      mov  r1, #4  ; 0x4
0x00000008:  e0812000      add  r2, r1, r0
0x0000000c:  eafffffe      b    0xc
```

10.4 Директивы ассемблера

В этом разделе мы посмотрим несколько часто используемых директив ассемблера, используя в качестве примера пару простых программ.

10.4.1 Суммирование массива

Следующий код 2 вычисляет сумму массива байт и сохраняет результат в r3:

Листинг 2: Сумма массива

```
        .text
entry:  b start                 @ перепрыгиваем данные
arr:    .byte 10, 20, 25        @ read-only массив байт
eoa:                            @ адрес конца массива + 1

        .align
start:
        ldr   r0, =eoa          @ r0 = &eoa
        ldr   r1, =arr          @ r1 = &arr
        mov   r3, #0            @ r3 = 0
```

```
loop:   ldrb  r2, [r1], #1      @ r2 = *r1++
        add   r3, r2, r3        @ r3 += r2
        cmp   r1, r0            @ if (r1 != r2)
        bne   loop              @ goto loop
stop:   b stop
```

В коде используются две новых ассемблерных директивы, описанных далее: `.byte` и `.align`.

`.byte`

Аргументы директивы `.byte` ассемблируются в последовательность байт в памяти. Также существуют аналогичные директивы `.2byte` и `.4byte` для ассемблирования 16- и 32-битных констант:

```
.byte   exp1, exp2, ...
.2byte  exp1, exp2, ...
.4byte  exp1, exp2, ...
```

Аргументом может быть целый числовой литерал: двоичный с префиксом 0b, 8-ричный префикс 0, десятичный и hex 0x. Также может использоваться строковая константа в одиночных кавычках, ассемблируемая в ASCII значения.

Также аргументом может быть Си-выражение из литералов и других символов, примеры:

```
pattern:  .byte 0b01010101, 0b00110011, 0b00001111
npattern: .byte npattern - pattern
halpha:   .byte 'A', 'B', 'C', 'D', 'E', 'F'
dummy:    .4byte 0xDEADBEEF
nalpha:   .byte 'Z' - 'A' + 1
```

`.align`

Архитектура ARM требует чтобы инструкции находились в адресах памяти, выровненных по границам 32-битного слова, т.е. в адресах с нулями в 2х младших разрядах. Другими словами, адрес каждого первого байта из 4-байтной команды, должен быть кратен 4. Для обеспечения этого предназначена директива `.align`, которая вставляет выравнивающие байты до следующего выровненного адреса. Ее нужно использовать только если в код вставляются байты или неполные 32-битные слова.

10.4.2 Вычисление длины строки

Этот код вычисляет длину строки и помещает ее в `r1`:

Листинг 3: Длина строки

```
        .text
        b start

str:    .asciz "Hello World"

        .equ   nul, 0

        .align
start:  ldr   r0, =str          @ r0 = &str
        mov   r1, #0

loop:   ldrb  r2, [r0], #1      @ r2 = *(r0++)
        add   r1, r1, #1        @ r1 += 1
        cmp   r2, #nul          @ if (r1 != nul)
```

```
        bne   loop              @ goto loop

        sub   r1, r1, #1        @ r1 -= 1
stop:   b stop
```

Код иллюстрирует применение директив `.asciz` и `.equ`.

`.asciz`

Директива `.asciz` принимает аргумент: строковый литерал, последовательность символов в двойных кавычках. Строковые литералы ассемблируются в память последовательно, добавляя в конец нулевой символ `\0` (признак конца строки в языке Си и стандарте POSIX).

Директива `.ascii` аналогична `.asciz`, но конец строки не добавляется. Все символы — 8-битные, кириллица может не поддерживаться.

`.equ`

Ассемблер при свой работе использует ***таблицу символов***: она хранит соответствия меток и их адресов. Когда ассемблер встречает очередное определение метки, он добавляет в таблицу новую запись. Когда встречается упоминание метки, оно заменяется соответствующим адресом из таблицы.

Использование директивы `.equ` позволяет добавлять записи в таблицу символов вручную, для привязки к именам любых числовых значений, не обязательно адресов. Когда ассемблер встречает эти имена, они заменяются на их значения. Эти имена-константы, и имена меток, называются ***сиволами***, а таблицы записанные в объектные файлы, или в отдельные .sym файлы — ***таблицами символов***[19].

Синтаксис директивы .equ:

[19] также используются отладчиком, чтобы показывать адреса переходов в виде понятных программисту сиволов, а не мутных числовых констант

```
.equ <имя>, <выражение>
```

Имя символа имеет те же ограничения по используемым символам, что и метка. Выражение может быть одиночным литералом или выражением как и в директиве `.byte`.

> В отличие от .byte, директива .equ не выделяет никакой памяти под аргумент. Она только добавляет значение в таблицу символов.

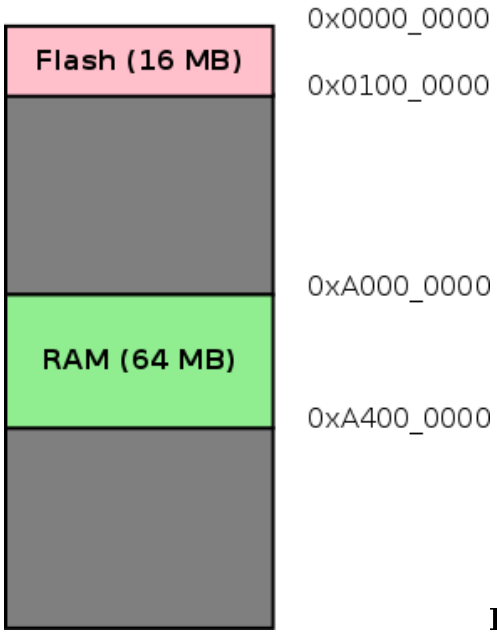
10.5 Использование ОЗУ (адресного пространства процессора)

Flash-память описанная ранее, в которой хранится машинный код программы, один из видов EEPROM[20]. Это вторичный носитель данных, как например жесткий диск, но в любом случае хранить данные и значения переменных во флеше неудобно как с точки зрения возможности перезаписи, так и прежде всего со скоростью чтения флеша, и кешированием.

В предыдущем примере мы использовали флеш как EEPROM для хранения константного массива байт, но вообще переменные должны храниться в максимально быстрой и неограниченно перезаписываемой RAM.

Плата connex имеет 64Mb ОЗУ начиная с адреса `0xA0000000`, для хранения данных программы. Карта памяти может быть представлена в виде диаграммы:

[20] Electrical Eraseable Programmable Read-Only Memory, электрически стираемая перепрограммируемая память только-для-чтения



Карта памяти Gumstix connex

[21]

Для настройки размещения переменных по нужным физическим адресам **нужна** некоторая **настройка адресного пространства**, особенно **если вы используете внешнюю память и переферийные устройства, подключемые к внешней шине**[22].

Для этого нужно понять, какую роль в распределении памяти играют ассемблер и линкер.

[21] здесь адреса считаются сверху вниз, что нетипично, обычно на диаграммах памяти адреса увеличиваются снизу вверх.

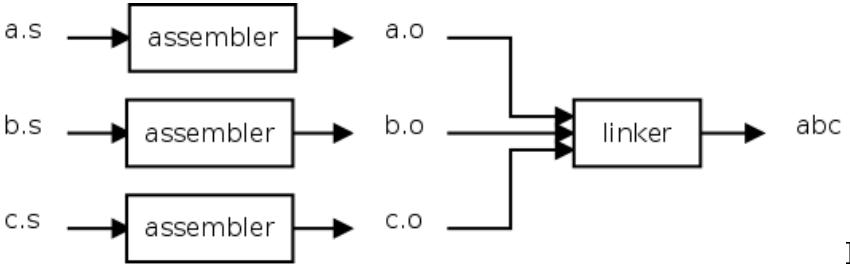
[22] или используете малораспространенные клоны ARM-процессоров, типа Миландровского 1986BE9x “чернобыль”

10.6 Линкер

Линкер позволяет ***скомпоновать*** исполняемый файл программы из нескольких объектных файлов, поделив ее на части. Чаще всего это нужно при использовании нескольких компиляторов для разных языков программирования: ассемблер, компиляторы C_+^+, Фортрана и Паскакаля.

Например, очень известная библиотека численных вычислений на базе матриц BLAS/LAPACK написана на Фортране, и для ее использования с сишной программой нужно слинковать program.o, blas.a и lapack.a[23] в один исполняемый файл.

При написании многофайловой программы (еще это называют ***инкрементной компоновкой***) каждый файл исходного кода ассемблируется в индивидуальный файл объектного кода. Линкер[24] собирает объектные файлы в финальный исполняемый бинарник.



Роль линкера

При сборке объектных файлов, линкер выполняет следующие операции:

- symbol resolution (разрешение символов)
- relocation (релокация)

В этой секции мы детально рассмотрим эти операции.

[23] .a — файлы архивов из пары сотен отдельных .o файлов каждый, по одному .o файлу на каждый возможный вариант функции линейной алгебры

[24] или компоновщик

10.6.1 Разрешение символов

В программе из одного файла при создании объектного файла все ссылки на метки заменяются их адресами непосредственно ассемблером. Но в программе из нескольких файлов существует множество ссылок на метки в других файлах, неизвестные на момент ассемблирования/компиляции, и ассемблер помечает их "unresolved" (неразреше́нные). Когда эти объектные файлы обрабатываются линкером, он определяет адреса этих меток по информации из других объектных файлов, и корректирует код. Этот процесс называется ***разрешением символов***.

Пример суммирования массива разделен на два файла для демонстрации разрешения символов, выполняемых линкером. Эти файлы ассемблируются отдельно, чтобы их таблицы символов содержали неразрешенные ссылки.

Файл **sumsub.s** содержит процедуру суммирования, а **summain.s** вызывает процедуру с требуемыми аргументами:

Листинг 4: summain.s вызов внешней процедуры

```
        .text
        b start                 @ пропустить данные
arr:    .byte 10, 20, 25        @ константный массив байт
eoa:                            @ адрес массива + 1
        .align
start:
        ldr   r0, =arr          @ r0 = &arr
        ldr   r1, =eoa          @ r1 = &eoa
        bl    sum               @ (вложенный) вызов процедуры
stop:   b stop
```

Листинг 5: sumsub.s код процедуры

```
        @ Аргументы (в регистрах)
        @ r0: начальный адрес массива
        @ r1: конечный адрес массива
        @
        @ Возврат результата
        @ r3: сумма массива

        .global sum

sum:    mov   r3, #0            @ r3 = 0
loop:   ldrb  r2, [r0], #1      @ r2 = *r0++    ; получить следующий элемент
        add   r3, r2, r3        @ r3 += r2      ; суммирование
        cmp   r0, r1            @ if (r0 != r1) ; проверка на конец массива
        bne   loop              @ goto loop     ; цикл
        mov   pc, lr            @ pc = lr       ; возврат результата по готовности
```

[25]

Применение директивы `.global` обязательно. В Си все функции и переменные, определенные вне функций, считаются видимыми из других объектных файлов, если они не определены с модификатором `static`. В ассемблере наоборот все метки считаются *статическими*[26], если с помощью директивы `.global` специально не указано, что они должны быть видимы извне.

Ассемблируйте файлы, и посмотрите дамп их таблицы символов используя комунду **nm**:

```
$ arm-none-eabi-as -o main.o main.s
```

[25] в архитектуре ARM нет специальной команды возврата ret, ее функцию выполняет прямое присваивание регистра указателя команд mov pc,lr

[26] или локальными на уровне файла

```
$ arm-none-eabi-as -o sum-sub.o sum-sub.s
$ arm-none-eabi-nm main.o
00000004 t arr
00000007 t eoa
00000008 t start
00000018 t stop
         U sum
$ arm-none-eabi-nm sum-sub.o
00000004 t loop
00000000 T sum
```

Теперь сфокусируемся на букве во втором столбце, который указывает тип символа: `t` указывает что символ определен в секции кода `.text`, `u` указывает что символ не определен. Буква в верхнем регистре указывает что символ глобальный и был указан в директиве `.global`.

Очевидно, что символ `sum` определ в **sum-sub.o** и еще не разрешен в **main.o**. Вызов линкера разрешает символьные ссылки, и создает исполняемый файл.

10.6.2 Релокация

Релокация — процесс изменения уже назначенных меткам адресов. Он также выполняет коррекцию всех ссылок, чтобы отразить заново назначенные адреса меток. В общем, релокация выполняется по двум основным причинам:

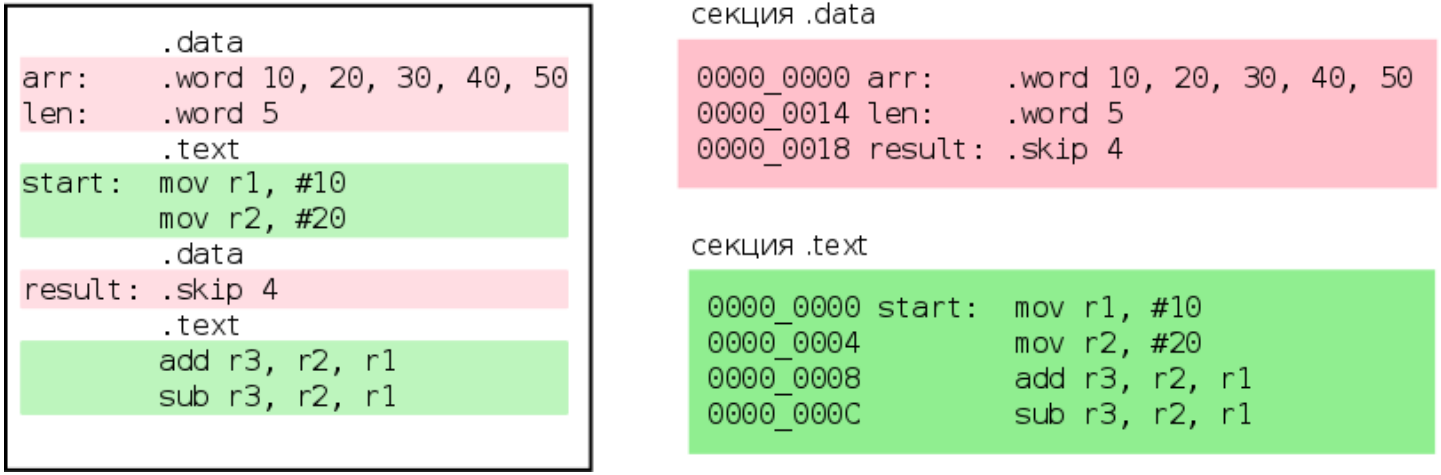
1. Объединение секций
2. Размещение секций

Для понимания процеса релокации, нужно понимание самой концепции секций.

Код и данные отличаются по требованиям при исполнении. Например код может размещаться в ROM-памяти, а данные требуют память открытую на запись. Очень хорошо, если **области кода и данных**

не пересекаются. Для этого программы делятся на секции. Большиство программ имеют как минимум две секции: `.text` для кода и `.data` для данных. Ассемблерные директивы `.text` и `.data` ожидаемо используются для переключения между этими секциями.

Хорошо представить каждую секцию как ведро. Когда ассемблер натыкается на директиву секции, он начинает сливать код/данные в соответствующее ведро, так что они размещаются в смежных адресах. Эта диаграмма показывает как ассемблер упорядочивает данные в секциях:



Секции

Теперь, когда у нас есть общее понимание ***секционирования*** кода и данных, давайте посмотрим по каким причинам выполняется релокация.

Объединение секций

Когда вы имеете дело с многофайловыми программами, секции в каждом объектном файле имеют одинаковые имена (‘.text‘,. . .), линкер отвечает за их объединение в выполняемом файле. По умолчанию секции

с одинаковыми именами из каждого **.o** файла объединяются последовательно, и метки корректируются на новые адреса.

Эффекты объединения секций можно посмотреть, анализируя таблицы символов отдельно в объектных и исполняемом файлах. Многофайловая программа суммирования может иллюстрировать объединение секций. Дампы таблиц символов:

```
$ arm-none-eabi-nm main.o
00000004 t arr
00000007 t eoa
00000008 t start
00000018 t stop
         U sum
$ arm-none-eabi-nm sum-sub.o
00000004 t loop <1>
00000000 T sum
$ arm-none-eabi-ld -Ttext=0x0 -o sum.elf main.o sum-sub.o
$ arm-none-eabi-nm sum.elf
...
00000004 t arr
00000007 t eoa
00000008 t start
00000018 t stop
00000028 t loop <1>
00000024 T sum
```

1. символ `loop` имеет адрес `0x4` в **sum-sub.o**, и `0x28` в **sum.elf**, так как секция `.text` из **sum-sub.o** размещена сразу за секцией `.text` из **main.o**.

Размещение секций

Когда программа ассемблируется, каждой секции назначается стартовый адрес `0x0`. Поэтому всем переменным назначаются адреса относительно начала секции. Когда создается финальный исполнямый файл, секция размещаются по некоторому адресу `X`, и все адреса меток, назначенные в секции, увеличиваются на `X`, так что они указывают на новые адреса.

Размещение каждой секции по определенному месту в памяти и коррекцию всех ссылок на метки в секции, выполняет линкер.

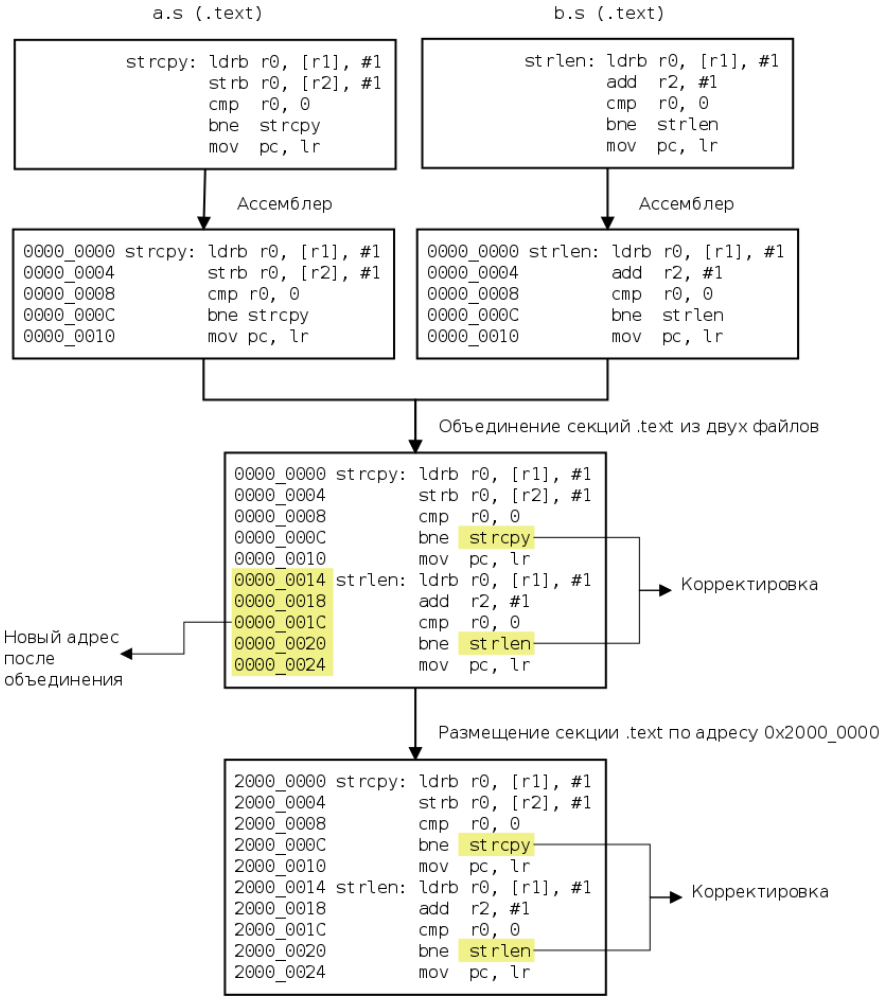
Эффект размещения секций можно увидеть по тому же дампу символов, описанному выше. Для простоы используем объектный файл однофайловой программы суммирования **sum.o**. Для увеличения заметности искуственно разместим секцию `.text` по адресу `0x100`:

```
$ arm-none-eabi-as -o sum.o sum.s
$ arm-none-eabi-nm -n sum.o
00000000 t entry <1>
00000004 t arr
00000007 t eoa
00000008 t start
00000014 t loop
00000024 t stop
$ arm-none-eabi-ld -Ttext=0x100 -o sum.elf sum.o <2>
$ arm-none-eabi-nm -n sum.elf
00000100 t entry <3>
00000104 t arr
00000107 t eoa
00000108 t start
00000114 t loop
00000124 t stop
```

...

1. Адреса меток назначаются с `0` от начала секции.
2. Когда создается выполняемый файл, линкеру указано разместить секцию кода с адреса `0x100`.
3. Адреса меток в `.text` переназначаются начиная с `0x100`, и все ссылки на метки корректируются.

Процесс объединения и размещения секций в общем показаны на диаграмме:



Объединение и размещение секций

10.7 Скрипт линкера

Как было описано в предыдущем разделе, объединение и размещение секций выполняется линкером. Программист может контролировать этот процесс через ***скрипт линкера***. Очень простой пример скрипта линкера:

Листинг 6: Простой скрипт линкера

```
SECTIONS { <1>
. = 0x00000000; <2>
.text : { <3>
abc.o (.text);
def.o (.text);
} <3>
}
```

1. Команда `SECTIONS` наиболее важная команда, она указывает как секции объединяются, и по каким адресам они размещаются.

2. Внутри блока `SECTIONS` команда `.` (точка) представляет ***указатель адреса размещения***. Указатель адреса всегда инициализируется `0x0`. Его можно модифицировать явно присваивая новое значение. Показанная явная установка на `0x0` на самом деле не нужна.

3. Этот блок скрипта определяет что секция `.text` выходного файла составляется из секций `.text` в файлах **abc.o** и **def.o**, причем именно в этом порядке.

Скрипт линкера может быть значительно упрощен и универсализирован с помощью использования символа шаблона `*` вместо индивидуального указания имен файлов:

Листинг 7: Шаблоны в скриптах линкера

```
SECTIONS {
. = 0x00000000;
.text : { * (.text); }
}
```

Если программа одновременно содержит секции ‘.text‘ и ‘.data‘, объединение и размещение секций можно прописать вот так:

Листинг 8: Несколько секций

```
SECTIONS {
 . = 0x00000000;
 .text : { * (.text); }

 . = 0x00000400;
 .data : { * (.data); }
}
```

Здесь секция `.text` размещается по адресу `0x0`, а секция `.data` — `0x400`. Отметим, что если указателю размещения не приваивать значения, секции `.text` и `.data` будут размещены в смежных областях памяти.

10.7.1 Пример скрипта линкера

Для демонстрации использования скриптов линкера рассмотрим применение скрипта **??** для размещения секций `.text` и `.data`. Для этого используем немного измененный пример программы суммирования массива, разделив код и данные в отдельные секции:

Листинг 9: Программа суммирования массива

```
        .data <1>
arr:    .byte 10, 20, 25    @ read-only массив байт
eoa:                        @ адрес конца массива + 1

        .text <2>
start:
        ldr   r0, =eoa      @ r0 = &eoa
        ldr   r1, =arr      @ r1 = &arr
        mov   r3, #0        @ r3 = 0
loop:   ldrb  r2, [r1], #1  @ r2 = *r1++
        add   r3, r2, r3    @ r3 += r2
        cmp   r1, r0        @ if (r1 != r2)
        bne   loop          @ goto loop
stop:   b stop
```

1. Изменения заключаются в выделении массива в секцию `.data` и удалении директивы выравнивания `.align`.

2. Также не требуется инструкция перехода на метку `start` для обхода данных, так как линкер разместит секции отдельно. В результате команды программы размещаются любым удобным способом, а скрипт линкера позаботится о правильном размещении сегментов в памяти.

При линковке программы в командной строке нужно указать использования скрипта:

```
$ arm-none-eabi-as -o sum-data.o sum-data.s
$ arm-none-eabi-ld -T sum-data.lds -o sum-data.elf sum-data.o
```

Опция `-T sum-data.lds` указывает что используется скрипт **sum-data.lds**. Выводим таблицу символов и видим размещение сегментов в памяти:

```
$ arm-none-eabi-nm -n sum-data.elf
00000000 t start
0000000c t loop
0000001c t stop
00000400 d arr
00000403 d eoa
```

Из таблицы символолов видно что секция `.text` размещена с адреса `0x0`, а секция `.data` с `0x400`.

10.7.2 Анализ объектного/исполняемого файла утилитой objdump

Более подробную информацию даст утилита **objdump**:

```
$ arm-none-eabi-as -o sum-data.o sum-data.s
$ arm-none-eabi-ld -T sum-data.lds -o sum-data.elf sum-data.o
$ arm-none-eabi-objdump sum-data.elf
```

Листинг 10: sum-data.objdump

```
sum-data.elf:     file format elf32-littlearm <1>
sum-data.elf
architecture: armv4 <2>, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x00000000 <3>
```

```
Program Header:
    LOAD off    0x010000 vaddr 0x000000 paddr 0x000000 align 2**16
         filesz 0x00000403 memsz 0x00000403 flags rwx
private flags = 5000200: [Version5 EABI <4>] [soft-float ABI <5>]

Sections: <6>
Idx Name          Size <10>     VMA <11>      LMA       File off  Algn <12>
<7>  0 .text      00000028  00000000  00000000  00010000  2**2
                  CONTENTS, ALLOC <13>, LOAD <14>, READONLY <15>, CODE <16>
<8>  1 .data      00000003  00000400  00000400  00010400  2**0
                  CONTENTS, ALLOC, LOAD, DATA <17>
<9>  2 .ARM.attributes 00000014  00000000  00000000  00010403  2**0
                  CONTENTS, READONLY
SYMBOL TABLE: <18>
00000000 l    d  .text 00000000 .text
00000400 l    d  .data 00000000 .data
00000000 l    d  .ARM.attributes 00000000 .ARM.attributes
00000000 l    df *ABS* 00000000 sum-data.o
00000400 l       .data 00000000 arr
00000403 l       .data 00000000 eoa
00000000 l       .text 00000000 start
0000000c l       .text 00000000 loop
0000001c l       .text 00000000 stop



Disassembly of section .text: <19>
```

```
00000000 <start>:
   0: e59f0018  ldr r0, [pc, #24] ; 20 <stop+0x4>
   4: e59f1018  ldr r1, [pc, #24] ; 24 <stop+0x8>
   8: e3a03000  mov r3, #0

0000000c <loop>:
   c: e4d12001  ldrb r2, [r1], #1
  10: e0823003  add r3, r2, r3
  14: e1510000  cmp r1, r0
  18: 1afffffb  bne c <loop>

0000001c <stop>:
  1c: eafffffe  b 1c <stop>
  20: 00000403  .word 0x00000403
  24: 00000400  .word 0x00000400
```

1. указание на архитектуру,
2. для которой предназначен исполняемый файл
3. стартовый адрес в секции `.text`, по умолчанию `0x0`[27]
4. ***ABI*** — соглашения о передаче
5. параметров в регистрах/стеке (для Си кода)
6. приведена подробная информация о секциях

[27] обязателен и фиксирован для прошивок микроконтроллеров, т.к. на него перескакивает аппаратный сброс

7. `.text` секция кода
8. `.data` секция данных
9. служебная информация
10. столбец `Size` указывает размер секции в байтах (hex)
11. `VMA`[28] указывает адрес размещения сегмента
12. `Algn` (Align) автоматическое выравнивание содержимого сегмента в памяти, в степени двойки `2**n`: код выравнивается кратно `2**2=4` байтам, данные не выравниваются `2**0=1`
13. Флаг `ALLOC` (Allocate) указывает что при загрузке программы под этот сегмент должна быть выделена память.
14. `LOAD` указывает что содержимое сегмента должно загружаться из исполняемого файла в память при использовании ОС, а для микроконтроллеров указывает программатору что сегмент нужно прошивать.
15. `READONLY` сегмент с константными неизменяемыми данными, которые могут быть размещены в ROM, а при использовании ОС область памяти должна быть помечена в таблице системы защиты памяти как R/O. Отсутствие флага `READONLY` + наличие `LOAD` указывает что данные должны загружаться **только в ОЗУ**.
16. сегмент кода
17. сегмент данных

[28] Virtual Memory Address

18. таблица символов

19. дизассемблированный код из секций, помеченных флагом `CODE`: `.text`

Глава 11

Embedded Systems Programming in C_+^+ [18]

[1]

[1] © http://www.bogotobogo.com/cplusplus/embeddedSystemsProgramming.php

Глава 12

Сборка кросс-компилятора GNU Toolchain из исходных текстов

12.1 Настройка целевой платформы

12.1.1 arm-none-eabi: процессоры ARM Cortex-Mx

12.1.2 i486-none-elf: ПК и промышленные PC104 без базовой ОС

12.1.3 arm-linux-uclibc: SoCи Cortex-A, PXA270,..

12.1.4 i686-linux-uclibc: микроLinux для платформы x86

12.2 dirs: создание структуры каталогов

12.3 gz: загрузка архивов исходных текстов компилятора

Глава 13

Оптимизация кода

13.1 PGO опитимизация

[1]

[1] © https://habrahabr.ru/post/138132/

Часть V

Микроконтроллеры Cortex-Mx

Часть VI

Технологии

Часть VII

Сетевое обучения

Часть VIII

Прочее

Ф.И.Атауллаханов об учебниках США и России

© Доктор биологических наук Фазли Иноятович Атауллаханов.
МГУ им. М. В. Ломоносова, Университет Пенсильвании, США

http://www.nkj.ru/archive/articles/19054/

...

У необходимости рекламировать науку есть важная обратная сторона: каждый американский учёный непрерывно, с первых шагов и всегда, учится излагать свои мысли внятно и популярно. В России традиции быть понятными у учёных нет. Как пример я люблю приводить двух великих физиков: русского Ландау и американца Фейнмана. Каждый написал многотомный учебник по физике. Первый — знаменитый "Ландау-Лифшиц", второй — "Лекции по физике". Так вот, "Ландау-Лифшиц" прекрасный справочник, но представляет собой полное издевательство над читателем. Это типичный памятник автору, который был, мягко говоря, малоприятным человеком. Он излагает то, что излагает, абсолютно пренебрегая своим читателем и даже издеваясь над ним. А у нас целые поколения выросли на этой книге, и считается, что всё нормально, кто справился, тот молодец. Когда я столкнулся с "Лекциями по физике" Фейнмана, я просто обалдел: оказывается, можно по-человечески разговаривать со своими коллегами, со студентами, с аспирантами. Учебник Ландау — пример того, как устроена у нас вся наука. Берёшь текст русской статьи, читаешь с самого начала и ничего не можешь понять, а иногда сомневаешься, понимает ли автор сам себя. Конечно, крупицы осмысленного и разумного и оттуда можно вынуть. Но автор явно считает, что это твоя работа — их оттуда извлечь. Не потому, что он не хочет быть понятым, а потому, что его не научили правильно писать. Не учат у нас человека ни писать, ни говорить понятно, это считается неважным.

...

Думаю, американская наука в целом устроена именно так: она продаёт не просто себя, а всю свою страну. Сегодня американцы дороги не метут, сапоги не тачают, даже телевизоры не собирают, за них это делает весь остальной мир. А что же делают американцы ? Самая богатая страна в мире ? Они объяснили,

в первую очередь самим себе, а заодно и всему миру, что они — мозг планеты. Они изобретают. “Мы придумываем продукты, а вы их делайте. В том числе и для нас”. Это прекрасно работает, поэтому они очень ценят науку.

...

Глава 14

Настройка редактора/IDE (g)Vim

При использовании редактора/IDE **(g)Vim** удобно настроить сочетания клавиш и подсветку синтаксиса языков, которые вы используете так, как вам удобно.

14.1 для вашего собственного скриптового языка

Через какое-то время практики FSP у вас выработается один диалект скриптов для всех программ, соответсвующий именно вашим вкусам в синтаксисе, и в этом случае его нужно будет описать только в файлах /.vim/(ftdetect|syntax).vim, и привязать их к расширениям через dot-файлы **(g)Vim** в вашем домашнем каталоге:

filetype.vim	**(g)Vim**	привязка расширений файлов (.src .log) к настройкам **(g)Vim**
syntax.vim	**(g)Vim**	синтаксическая подсветка для скриптов
/.vimrc	*Linux*	настройки для пользователя
/vimrc	*Windows*	
/.vim/ftdetect/src.vim	*Linux*	привязка команд к расширению .src
/vimfiles/ftdetect/src.vim	*Windows*	
/.vim/syntax/src.vim	*Linux*	синтаксис к расширению .src
/vimfiles/syntax/src.vim	*Windows*	

Книги must have любому техническому специалисту

Математика, физика, химия

- Бермант **Математический анализ** [22]
- Кремер **Теория вероятностей и матстатистика** [23]
- Смит **Цифровая обработка сигналов** [24]

Фейнмановские лекции по физике

1. Современная наука о природе. Законы механики. [27]
2. Пространство. Время. Движение. [28]
3. Излучение. Волны. Кванты. [29]
4. Кинетика. Теплота. Звук. [30]
5. Электричество и магнетизм [31]
6. Электродинамика. [32]
7. Физика сплошных сред. [33]
8. Квантовая механика 1. [34]

9. Квантовая механика 2. [35]

Обработка экспериментальных данных и метрология

- Князев, Черкасский **Начала обработки экспериментальных данных** [25]

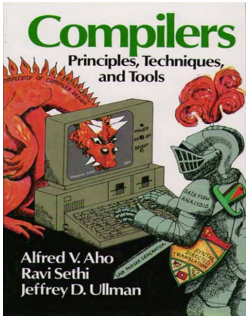
Программирование

- **Система контроля версий Git и git-хостинга GitHub**
  хранение наработок с полной историей редактирования, правок, релизов для разных заказчиков или вариантов использования
- **Язык Python** [20]
  написание простых скриптов обработки данных, автоматизации, графических оболочек и т.п. утилит
- **Язык C_+^+, утилиты GNU toolchain** [18, 19] (gcc/g++, make, ld)
  базовый Си, ООП очень кратко[1], без излишеств профессионального программирования[2], чисто вспомогательная роль для написания вычислительных блоков и критичных к скорости/памяти секций, использовать в связке с Python.
  Знание базового Си **критично при использовании микроконтроллеров**, из C_+^+ необходимо владение особенностями использования ООП и управления крайне ограниченной памятью: пользовательские менеджеры памяти, статические классы.
- Использование утилит **flex/bison**
  обработка текстовых форматов данных, часто необходимая вещь.

[1] наследование, полиморфизм, операторы для пользовательских типов, использование библиотеки STL
[2] мегабиблиотека Boost, написание своих библиотек шаблонов и т.п.

Литература

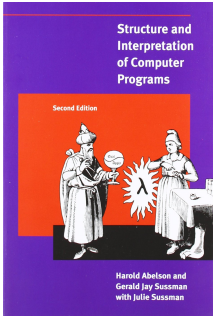
Разработка языков программирования и компиляторов

[1] 

Dragon Book
Компиляторы. Принципы, технологии, инструменты.
Альфред Ахо, Рави Сети, Джеффри Ульман.
Издательство Вильямс, 2003.
ISBN 5-8459-0189-8

[2] **Compilers: Principles, Techniques, and Tools**
Aho, Sethi, Ullman

Addison-Wesley, 1986.
ISBN 0-201-10088-6



[3] **SICP**
Структура и интерпретация компьютерных программ
Харольд Абельсон, Джеральд Сассман
ISBN 5-98227-191-8
EN: web.mit.edu/alexmv/6.037/sicp.pdf

[4] **Функциональное программирование**
Филд А., Харрисон П.

М.: Мир, 1993
ISBN 5-03-001870-0

[5] Функциональное программирование: применение и реализация
П.Хендерсон
М.: Мир, 1983

[6] **LLVM. Инфраструктура для разработки компиляторов**
Бруно Кардос Лопес, Рафаэль Аулер

Lisp/Sheme

Haskell

ML

[7] http://homepages.inf.ed.ac.uk/mfourman/teaching/mlCourse/notes/L01.pdf
Basics of Standard ML
© Michael P. Fourman
перевод 1

[8] http://www.soc.napier.ac.uk/course-notes/sml/manual.html
A Gentle Introduction to ML
© Andrew Cumming, Computer Studies, Napier University, Edinburgh

[9] http://www.cs.cmu.edu/~rwh/smlbook/book.pdf
Programming in Standard ML
© Robert Harper, Carnegie Mellon University

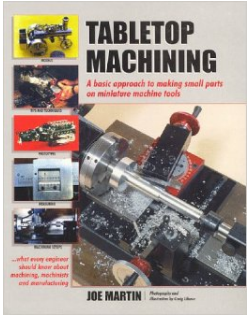
Электроника и цифровая техника

[10] **An Introduction to Practical Electronics, Microcontrollers and Software Design**
Bill Collis
2 edition, May 2014
http://www.techideas.co.nz/

Конструирование и технология

Приемы ручной обработки материалов

Механообработка

[11] 

Tabletop Machining
Martin, Joe and Libuse, Craig
Sherline Products, 2000

[12] **Home Machinists Handbook**
Briney, Doug, 2000

[13] **Маленькие станки**
Евгений Васильев
Псков, 2007
http://www.coilgun.ru/stanki/index.htm

Использование OpenSource программного обеспечения

LaTeX

[14] **Набор и вёрстка в системе LaTeX**
С.М. Львовский
3-е издание, исправленное и дополненное, 2003
http://www.mccme.ru/free-books/llang/newllang.pdf

[15] **e-Readers and LaTeX**
Alan Wetmore
https://www.tug.org/TUGboat/tb32-3/tb102wetmore.pdf

[16] **How to cite a standard (ISO, etc.) in BibLaTeX ?**
http://tex.stackexchange.com/questions/65637/

Математическое ПО: Maxima, Octave, GNUPLOT,..

[17] **Система аналитических вычислений Maxima для физиков-теоретиков**
В.А. Ильина, П.К.Силаев
http://tex.bog.msu.ru/numtask/max07.ps

САПР, электроника, проектирование печатных плат

Программирование

GNU Toolchain

[18] **Embedded Systems Programming in C_+^+**
© http://www.bogotobogo.com/
http://www.bogotobogo.com/cplusplus/embeddedSystemsProgramming.php

[19] **Embedded Programming with the GNU Toolchain**
Vijay Kumar B.
http://bravegnu.org/gnu-eprog/

Python

[20] **Язык программирования Python**
Россум, Г., Дрейк, Ф.Л.Дж., Откидач, Д.С., Задка, М., Левис, М., Монтаро, С., Реймонд, Э.С., Кучлинг, А.М., Лемburг, М.-А., Йи, К.-П., Ксиллаг, Д., Петрилли, Х.Г., Варсав, Б.А., Ахлстром, Дж.К., Роскинд, Дж., Шеменор, Н., Мулендер, С.
© Stichting Mathematisch Centrum, 1990–1995 and Corporation for National Research Initiatives, 1995–2000 and BeOpen.com, 2000 and Откидач, Д.С., 2001
http://rus-linux.net/MyLDP/BOOKS/python.pdf

Python является простым и, в то же время, мощным интерпретируемым объектно-ориентированным языком программирования. Он предоставляет структуры данных высокого уровня, имеет изящный синтаксис и использует динамический контроль типов, что делает его идеальным языком для быстрого написания различных приложений, работающих на большинстве распространенных платформ. Книга содержит вводное руководство, которое может служить учебником для начинающих, и справочный материал с подробным описанием грамматики языка, встроенных возможностей и возможностей, предоставляемых модулями стандартной библиотеки. Описание охватывает наиболее распространенные версии Python: от 1.5.2 до 2.0.

Разработка операционных систем и низкоуровневого ПО

[21] **OSDev Wiki**
http://wiki.osdev.org

Базовые науки

Математика

[22] **Краткий курс математического анализа для ВТУЗов**
Бермант А.Ф, Араманович И.Г.
М.: Наука, 1967
https://drive.google.com/file/d/0B0u4WeMjO894U1Y1dEJ6cncxU28/view?usp=sharing

Пятое издание известного учебника, охватывает большинство вопросов программы по высшей математике для инженерно-технических специальностей вузов, в том числе дифференциальное исчисление функций одной переменной и его применение к исследованию функций; дифференциальное исчисление функций нескольких переменных; интегральное исчисление; двойные, тройные и криволинейные интегралы; теорию поля; дифференциальные уравнения; степенные ряды и ряды Фурье. Разобрано много примеров и задач из различных разделов механики и физики. **Отличается крайней доходчивостью и отсутвием филонианов и “легко догадаться”.**

[23] **Теория вероятностей и математическая статистика**

Наум Кремер
М.: Юнити, 2010

[24] **Цифровая обработка сигналов. Практическое руководство для инженеров и научных работников**

Стивен Смит
Додэка XXI, 2008
ISBN 978-5-94120-145-7

В книге изложены основы теории цифровой обработки сигналов. Акцент сделан на доступности изложения материала и объяснении методов и алгоритмов так, как они понимаются при практическом

использовании. Цель книги - практический подход к цифровой обработке сигналов, позволяющий преодолеть барьер сложной математики и абстрактной теории, характерных для традиционных учебников. Изложение материала сопровождается большим количеством примеров, иллюстраций и текстов программ

[25] **Начала обработки экспериментальных данных**
Б.А.Князев, В.С.Черкасский
Новосибирский государственный университет, кафедра общей физики, Новосибирск, 1996
http://www.phys.nsu.ru/cherk/Metodizm_old.PDF

Учебное пособие предназначено для студентов естественно-научных специальностей, выполня- ющих лабораторные работы в учебных практикумах. Для его чтения достаточно знаний математики в объеме средней школы, но оно может быть полезно и тем, кто уже изучил математическую статистику, поскольку исходным моментом в нем является не математика, а эксперимент. Во второй части пособия подробно описан реальный эксперимент — от появления идеи и проблем постановки эксперимента до получения результатов и обработки данных, что позволяет получить менее формализованное представление о применении математической статистики. Посо- бие дополнено обучающей программой, которая позволяет как углубить и уточнить знания, полученные в методическом пособии, так и проводить собственно обработ- ку результатов лабораторных работ. Приведен список литературы для желающих углубить свои знания в области математической статистики и обработки данных.

Физика

[26] Савельев И.В.

Фейнмановские лекции по физике
Ричард Фейнман, Роберт Лейтон, Мэттью Сэндс

[27] **Современная наука о природе. Законы механики.**

[28] **Пространство. Время. Движение.**

[29] **Излучение. Волны. Кванты.**

[30] **Кинетика. Теплота. Звук.**

[31] **Электричество и магнетизм.**

[32] **Электродинамика.**

[33] **Физика сплошных сред.**

[34] **Квантовая механика 1.**

[35] **Квантовая механика 2.**

Химия

Стандарты и ГОСТы

[36] 2.701-2008 **Схемы. Виды и типы. Общие требования к выполнению**
http://rtu.samgtu.ru/sites/rtu.samgtu.ru/files/GOST_ESKD_2.701-2008.pdf