

# Оглавление

Об этом сборнике . . . . .	9
<b>I ML &amp; функциональное программирование</b>	<b>10</b>
<b>1 Основы Standard ML</b> © Michael P. Fourman	11
1.1 Введение . . . . .	11
<b>2 Programming in Standard ML'97</b>	<b>13</b>
An On-line Tutorial © Stephen Gilmore . . . . .	13
<b>II Язык Prolog: логическое программирование и искусственный интеллект</b>	<b>14</b>
<b>3 Учебник Фишера</b>	<b>15</b>
Введение . . . . .	15
3.1 Установка и запуск <i>Prolog</i> -системы . . . . .	17
3.2 Разбор примеров программ . . . . .	21
3.2.1 Раскраска карт . . . . .	21
3.2.2 Два определения факториала . . . . .	25
3.2.3 Классическая задача “Ханойские башни” . . . . .	29
3.2.4 Загрузка, редактирование, хранение программ . . . . .	32
3.2.5 2.5 Negation as failure . . . . .	34
3.2.6 2.6 Tree data and relations . . . . .	34
3.2.7 2.7 Prolog lists and sequences . . . . .	35
3.2.8 2.8 Change for a dollar . . . . .	35
3.2.9 2.9 Map coloring redux . . . . .	35
3.2.10 2.10 Simple I/O . . . . .	35
3.2.11 2.11 Chess queens challenge puzzle . . . . .	35
3.2.12 2.12 Finding all answers . . . . .	35
3.2.13 2.13 Truth table maker . . . . .	35
3.2.14 2.14 DFA parser . . . . .	35
3.2.15 2.15 Graph structures and paths . . . . .	36

3.2.16	2.16 Search . . . . .	36
3.2.17	2.17 Animal identification game . . . . .	36
3.2.18	2.18 Clauses as data . . . . .	36
3.2.19	2.19 Actions and plans . . . . .	36
3.3	Как работает <i>Prolog</i> . . . . .	36
3.3.1	Деривационные деревья, выборы и унификация . . . . .	36
3.3.2	3.2 Cut . . . . .	41
3.3.3	3.3 Meta-interpreters in <i>Prolog</i> . . . . .	41
3.4	4. Built-in Goals . . . . .	41
3.4.1	4.1 Utility goals . . . . .	41
3.4.2	4.2 Universals (true and fail) . . . . .	41
3.4.3	4.3 Loading <i>Prolog</i> programs . . . . .	41
3.4.4	4.4 Arithmetic goals . . . . .	41
3.4.5	4.5 Testing types . . . . .	41
3.4.6	4.6 Equality of <i>Prolog</i> terms, unification . . . . .	41
3.4.7	4.7 Control . . . . .	41
3.4.8	4.8 Testing for variables . . . . .	41
3.4.9	4.9 Assert and retract . . . . .	41
3.4.10	4.10 Binding a variable to a numerical value . . . . .	41
3.4.11	4.11 Procedural negation, negation as failure . . . . .	41
3.4.12	4.12 Input/output . . . . .	41
3.4.13	4.13 <i>Prolog</i> terms and clauses as data . . . . .	41
3.4.14	4.14 <i>Prolog</i> operators . . . . .	41
3.4.15	4.15 Finding all answers . . . . .	41
3.5	5. Search in <i>Prolog</i> . . . . .	41
3.5.1	5.1 The A* algorithm in <i>Prolog</i> . . . . .	41
3.5.2	5.2 The 8-puzzle . . . . .	41
3.5.3	5.3 $\alpha\beta$ search in <i>Prolog</i> . . . . .	41
3.6	6. Logic Topics . . . . .	41
3.6.1	6.1 Chapter 6 notes . . . . .	41
3.6.2	6.2 Positive logic . . . . .	41
3.6.3	6.3 Convert first-order logic to normal form . . . . .	41
3.6.4	6.4 A normal rulebase goal interpreter . . . . .	41
3.6.5	6.5 Evidentiary soundness and completeness . . . . .	41
3.6.6	6.6 Rule tree visualization using Java . . . . .	41
3.7	7. Introduction to Natural Language Processing . . . . .	41
3.7.1	7.1 <i>Prolog</i> grammar parser generator . . . . .	41
3.7.2	7.2 <i>Prolog</i> grammar for simple English phrase structures . . . . .	41
3.7.3	7.3 Idiomatic natural language command and question interfaces . . . . .	41
3.8	8. Prototyping with <i>Prolog</i> . . . . .	41
3.8.1	8.1 Action specification for a simple calculator . . . . .	41
3.8.2	8.2 Animating the 8-puzzle (\$5.2) using character graphics . . . . .	41
3.8.3	8.3 Animating the blocks mover (\$2.19) using character graphics . . . . .	41

3.8.4	8.4 Java Tic-Tac-Toe GUI plays against <i>Prolog</i> opponent (\$5.3)	41
3.8.5	8.5 Structure diagrams and <i>Prolog</i>	41
References		41
<b>4</b>	<b>ASTLOG: Язык для анализа синтаксических деревьев</b>	<b>42</b>
Abstract		42
4.1	Introduction	43
4.1.1	The <b>awk</b> Approach	43
4.1.2	The Logic Programming Approach	45
4.2	Elements of ASTLOG	46
Figure 1:	Complete Syntax of ASTLOG	46
4.2.1	Objects	47
4.2.2	The Current Object	47
Figure 2:	Some core ASTLOG primitives	49
Figure 3:	Some primitive node and symbol predicates	50
4.2.3	Examples	50
Figure 4:	Actual ASTLOG code for follow_stmt	53
Figure 5:	Definition of <b>flatten</b>	54
Figure 6:	Parameterized version, <b>flatten2</b>	54
4.3	Higher order features	55
4.3.1	3.1 Lambdas and Applications	55
Figure 7:	Parameterized version of sametree	56
Figure 8:	Embedded Query State Primitives	57
4.3.2	Queries as Objects	57
Figure 9:	Query Accumulators qcount and qlist	58
4.4	Implementation	59
4.5	Conclusions and Future Work	60
4.6	Acknowledgements	62
References		62
Appendix		63
Figure 10:	Outline of astlog Operational Semantics	64
<b>5</b>	<b>Warren's Abstract Machine</b>	<b>65</b>
<b>Абстрактная машина Варрена</b>		<b>65</b>
Предисловие к репринтному изданию		65
Предисловие		66
5.1	1 Введение 3	67
5.1.1	1.1 Существующая литература 3	67
5.1.2	1.2 Этот учебник 5	69
5.2	2 Унификация — ясно и просто 9	70
5.2.1	2.1 Term representation . . . . .	10
5.2.2	2.2 Compiling L queries . . . . .	11
5.2.3	2.3 Compiling L programs . . . . .	13
5.2.4	2.4 Argument registers . . . . .	19
		73

5.3	3 Flat Resolution 25 . . . . .	73
5.3.1	3.1 Facts . . . . .	73
5.3.2	3.2 Rules and queries . . . . .	73
5.4	4 Prolog 33 . . . . .	73
5.4.1	4.1 Environment protection . . . . .	73
5.4.2	4.2 What's in a choice point . . . . .	73
5.5	5 Optimizing the Design 45 . . . . .	73
5.5.1	5.1 Heap representation . . . . .	46
5.5.2	5.2 Constants, lists, and anonymous variables . . . . .	47
5.5.3	5.3 A note on set instructions . . . . .	52
5.5.4	5.4 Register allocation . . . . .	54
5.5.5	5.5 Last call optimization . . . . .	56
5.5.6	5.6 Chain rules . . . . .	57
5.5.7	5.7 Environment trimming . . . . .	58
5.5.8	5.8 Stack variables . . . . .	60
5.5.9	5.9 Variable classification revisited . . . . .	69
5.5.10	5.10 Indexing . . . . .	75
5.5.11	5.11 Cut . . . . .	83
5.6	6 Conclusion 89 . . . . .	73
5.7	A Prolog in a Nutshell 91 . . . . .	73
5.8	B The WAM at a glance 97 . . . . .	73
5.8.1	B.1 WAM instructions . . . . .	97
5.8.2	B.2 WAM ancillary operations . . . . .	112
5.8.3	B.3 WAM memory layout and registers . . . . .	117
<b>III Язык <i>bI</i></b>		<b>74</b>
<b>6</b>	<b>DLR: Dynamic Language Runtime</b>	<b>75</b>
<b>7</b>	<b>Система динамических типов</b>	<b>78</b>
7.1	sym: символ = Абстрактный Символьный Тип /AST/ . . . . .	78
7.2	Скаляры . . . . .	80
7.2.1	str: строка . . . . .	80
7.2.2	int: целое число . . . . .	80
7.2.3	hex: машинное hex . . . . .	80
7.2.4	bin: бинарная строка . . . . .	80
7.2.5	num: число с плавающей точкой . . . . .	80
7.3	Композиты . . . . .	80
7.3.1	list: плоский список . . . . .	80
7.3.2	cons: cons-пара и списки в <i>Lisp</i> -стиле . . . . .	80
7.4	Функционалы . . . . .	80
7.4.1	op: оператор . . . . .	80
7.4.2	fn: встроенная/скомпилированная функция . . . . .	80

7.4.3	<i>lambda</i> : лямбда	80
<b>8</b>	<b>Программирование в свободном синтаксисе: FSP</b>	<b>81</b>
8.1	Типичная структура проекта FSP: <i>lexical skeleton</i>	81
8.1.1	Настройки (g)Vim	82
8.1.2	Дополнительные файлы	82
8.1.3	Makefile	83
<b>9</b>	<b>Синтаксический анализ текстовых данных</b>	<b>84</b>
9.1	Универсальный Makefile	84
9.2	$C^+$ интерфейс синтаксического анализатора	85
9.3	Минимальный парсер	85
9.4	Добавляем обработку комментариев	87
9.5	Разбор строк	88
9.6	Добавляем операторы	89
9.7	Обработка вложенных структур (скобок)	92
<b>10</b>	<b>Синтаксический анализатор</b>	<b>94</b>
10.1	lpp.lpp: лексер /flex/	94
10.2	урр.урр: парсер /bison/	96
<b>IV</b>	<b>skelex: скелет программы в свободном синтаксисе</b>	<b>99</b>
Структура проекта	100	
<b>Makefile</b>	100	
урр.урр: синтаксический парсер	101	
lpp.lpp: лексер	103	
hpp.hpp: хедеры	104	
crr.crr: ядро интерпретатора	106	
Тестирование интерпретатора	108	
Комментарии	108	
Скаляры и базовые композиты	108	
Операторы	109	
<b>V</b>	<b>emLinux для встраиваемых систем</b>	<b>111</b>
Структура встраиваемого микроЛinuxа	112	
Процедура сборки	113	
<b>11</b>	<b>clock: коридорные электронные часы = контроллер умного дур-дома</b>	<b>114</b>
<b>12</b>	<b>gambox: игровая приставка</b>	<b>115</b>

<b>13 Программирование встраиваемых систем с использованием GNU Toolchain [23]</b>	<b>117</b>
13.1 Введение . . . . .	117
13.2 Настройка тестового стенда . . . . .	118
13.2.1 Qemu ARM . . . . .	118
13.2.2 Инсталляция Qemu на <i>Debian GNU/Linux</i> . . . . .	118
13.2.3 Установка кросс-компилятора GNU Toolchain для ARM . . . . .	118
13.3 Hello ARM . . . . .	119
13.3.1 Сборка бинарника . . . . .	120
13.3.2 Выполнение в Qemu . . . . .	123
13.3.3 Другие команды монитора . . . . .	125
13.4 Директивы ассемблера . . . . .	125
13.4.1 Суммирование массива . . . . .	125
13.4.2 Вычисление длины строки . . . . .	127
13.5 Использование ОЗУ (адресного пространства процессора) . . . . .	128
13.6 Линкер . . . . .	130
13.6.1 Разрешение символов . . . . .	130
13.6.2 Релокация . . . . .	132
13.7 Скрипт линкера . . . . .	137
13.7.1 Пример скрипта линкера . . . . .	138
13.7.2 Анализ объектного/исполняемого файла утилитой <b>objdump</b> . . . . .	139
13.8 Данные в RAM, пример . . . . .	140
13.8.1 RAM энергозависима ( <i>volatile</i> )! . . . . .	141
13.8.2 Спецификация адреса загрузки LMA . . . . .	142
13.8.3 Копирование ‘.data’ в ОЗУ . . . . .	143
13.9 Обработка аппаратных исключений . . . . .	145
13.10 Стартап-код на Си . . . . .	146
13.10.1 Стек . . . . .	147
13.10.2 Глобальные переменные . . . . .	149
13.10.3 Константные данные . . . . .	149
13.10.4 Секция <b>.eeprom</b> (AVR8) . . . . .	149
13.10.5 Стартовый код . . . . .	149
13.11 Использование библиотеки Си . . . . .	153
13.12 Inline-ассемблер . . . . .	154
13.13 Использование ‘make’ для автоматизации компиляции . . . . .	154
13.13.1 Выбор конкретной <i>цели</i> . . . . .	155
13.13.2 Переменные . . . . .	156
13.14 Contributing . . . . .	156
13.15 Credits . . . . .	156
13.15.1 People . . . . .	156
13.15.2 Tools . . . . .	156

13.16.15. Tutorial Copyright . . . . .	156
13.17A. ARM Programmer's Model . . . . .	156
13.18B. ARM Instruction Set . . . . .	156
13.19C. ARM Stacks . . . . .	156
<b>14 Embedded Systems Programming in C<sub>+</sub> [22]</b>	<b>157</b>
<b>15 Сборка кросс-компилятора GNU Toolchain из исходных текстов</b>	<b>158</b>
APP/HW: приложение/платформа . . . . .	159
Подготовка BUILD-системы: необходимое ПО . . . . .	159
dirs: создание структуры каталогов . . . . .	159
Сборка в ОЗУ на ramdiske . . . . .	160
Пакеты системы кросс-компиляции . . . . .	160
gz: загрузка исходного кода для пакетов . . . . .	161
Макро-правила для автоматической распаковки исходников . . . . .	162
Общие параметры для ./configure . . . . .	162
15.1 Сборка кросс-компилятора . . . . .	163
15.1.1 cclibs0: библиотеки поддержки gcc . . . . .	163
15.1.2 binutils0: ассемблер и линкер . . . . .	164
15.1.3 gcc00: сборка stand-alone компилятора Си . . . . .	166
15.1.4 newlib: сборка стандартной библиотеки libc . . . . .	167
15.1.5 gcc0: пересборка компилятора Си/C <sub>+</sub> . . . . .	167
15.2 Поддерживаемые платформы . . . . .	167
15.2.1 i386: ПК и промышленные PC104 . . . . .	167
15.2.2 x86_64: серверные системы . . . . .	167
15.2.3 AVR: Atmel AVR Mega . . . . .	167
15.2.4 arm: процессоры ARM Cortex-Mx . . . . .	167
15.2.5 armhf: SoCi Cortex-A, PXA270,. . . . .	167
15.3 Целевые аппаратные системы . . . . .	168
15.3.1 x86: типовой компьютер на процессоре i386+ . . . . .	168
<b>16 Porting The GNU Tools To Embedded Systems</b>	<b>169</b>
<b>17 Оптимизация кода</b>	<b>170</b>
17.1 PGO оптимизация . . . . .	170
<b>VII Микроконтроллеры Cortex-Mx</b>	<b>171</b>
<b>VIII os86: низкоуровневое программирование i386</b>	<b>172</b>
Специализированный GNU Toolchain для i386-pc-gnu . . . . .	173
MultiBoot-загрузчик . . . . .	173

<b>IX Спецификация MultiBoot</b>	<b>174</b>
<b>18 Introduction to Multiboot Specification</b>	<b>176</b>
18.1 The background of Multiboot Specification . . . . .	176
18.2 The target architecture . . . . .	176
18.3 The target operating systems . . . . .	177
18.4 Boot sources . . . . .	177
18.5 Configure an operating system at boot-time . . . . .	177
18.6 How to make OS development easier . . . . .	177
18.7 Boot modules . . . . .	178
The definitions of terms used through the specification . . . . .	178
<b>19 The exact definitions of Multiboot Specification</b>	<b>180</b>
19.1 OS image format . . . . .	180
19.1.1 The layout of Multiboot header . . . . .	180
19.1.2 The magic fields of Multiboot header . . . . .	181
19.1.3 The address fields of Multiboot header . . . . .	182
19.1.4 The graphics fields of Multiboot header . . . . .	183
19.2 Machine state . . . . .	183
19.3 Boot information format . . . . .	184
Examples . . . . .	189
History . . . . .	189
Index . . . . .	189
<b>X Технологии</b>	<b>190</b>
<b>XI Сетевое обучение</b>	<b>191</b>
<b>XII Базовая теоретическая подготовка</b>	<b>192</b>
<b>20 Математика</b>	<b>193</b>
20.1 Высшая математика в упражнениях и задачах [61] . . . . .	193
Запуск Maxima и Octave в пакетном режиме . . . . .	194
20.1.1 Аналитическая геометрия на плоскости . . . . .	194
<b>XIII Прочее</b>	<b>202</b>
Ф.И.Атауллаханов об учебниках США и России . . . . .	203
<b>21 Настройка редактора/IDE (g)Vim</b>	<b>204</b>
21.1 для вашего собственного скриптового языка . . . . .	204

<b>Книги</b>	<b>204</b>
Книги must have любому техническому специалисту . . . . .	204
Математика, физика, химия . . . . .	204
Обработка экспериментальных данных и метрология . . . . .	205
Программирование . . . . .	205
САПР, пакеты математики, моделирования, визуализации . . . . .	206
Разработка языков программирования и компиляторов . . . . .	207
Lisp/Sheme . . . . .	209
Haskell . . . . .	209
ML . . . . .	209
Электроника и цифровая техника . . . . .	210
Конструирование и технология . . . . .	211
Приемы ручной обработки материалов . . . . .	211
Механообработка . . . . .	211
Использование OpenSource программного обеспечения . . . . .	212
$\text{\LaTeX}$ . . . . .	212
Математическое ПО: Maxima, Octave, GNUPlot,..	213
САПР, электроника, проектирование печатных плат . . . . .	214
Программирование . . . . .	214
GNU Toolchain . . . . .	214
JavaScript, HTML, CSS, Web-технологии: . . . . .	214
Python . . . . .	214
Разработка операционных систем и низкоуровневого ПО . . . . .	215
Базовые науки . . . . .	215
Математика . . . . .	215
Символьная алгебра . . . . .	218
Численные методы . . . . .	220
Теория игр . . . . .	221
Физика . . . . .	221
Химия . . . . .	223
Задачники . . . . .	224
Математика . . . . .	224
Стандарты и ГОСТы . . . . .	225
<b>Индекс</b>	<b>225</b>

## Об этом сборнике

© Dmitry Ponyatov <[dponyatov@gmail.com](mailto:dponyatov@gmail.com)>

В этот сборник (блогбук) я пишу отдельные статьи и переводы, сортированные только по общей тематике, и добавляю их, когда у меня в очередной раз зачешется  $\text{\LaTeX}$ .

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, большая часть только на этапе освоения мной самим, просто хочется иметь некое слабоупорядоченное хранилище наработок, на которое можно дать кому-то ссылку.

Сборник сверстан в микроформат<sup>1</sup> для просмотра на телефонах и мобильных девайсах, проверялось на удобство чтения на Alcatel Onetouch 4007D Pixi: в горизонтальной ориентации вполне читается в транспорте.

---

<sup>1</sup> А6 и менее

## 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! Представление 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 программистские привычки на *ML* непролетарны — 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.<sup>1</sup>

**Примечание 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 Standart 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](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.

---

<sup>1</sup> 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 — #.

## 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 *Prolog*: логическое  
программирование  
и искусственный интеллект

# Глава 3

## Учебник Фишера

© J.R.Fisher 's *PrologTutorial* <sup>1</sup>

### Введение

*Prolog* — язык декларативного логического программирования. Детально рассматривая его имя, получаем что это сокращение от PROGramming in LOGic: логическое программирование. Наследие *Prologa* включает исследования в области *автоматического доказательства теорем* и других *дедуктивных систем*, разработанных в 1960-70х гг. *Механизм вывода Prologa* базируется на принципе разрешения Робинсона (1965) и механизмах вывода ответов, приложенных Грином (1968). Эти идеи используются вместе с процедурой линейного разрешения. Процедуры точного целевого линейное разрешения, такие как методы Kowalski / Kuehner (1971) и Kowalski (1974), дали толчок к разработке систем логического программирования общего назначения. “Первым” *Prologом* был “Марсельский *Prolog*”, реализация которого основана на работе Colmerauer (1970). Первым делательным описанием языка *Prolog* было руководство к интерпретатору Marseille Prolog (Roussel, 1975). Другим сильным влиянием на природу этого первого *Prologa* была адаптация этого интерпретатора к задачам *обработки естественных языков*.

*Prolog* является наиболее часто упоминаемым примеров языков программирования четвертого поколения, которые поддерживают парадигму **декларативного программирования**. Японский проект Fifth-Generation Computer Project<sup>2</sup>, анонсированный в 1981, применял *Prolog* как язык разработки, и сосредоточивал значительные усилия на языке и его возможностях. Программы в этом учебнике написаны на “стандартном” *Prologе* Эдинбургского университета<sup>3</sup>, как это сделано в классической книге по *Prologу* под авторством Clocksin и Mellish (1981,1992).

<sup>1</sup> © [https://www.cpp.edu/~jrfisher/www/prolog\\_tutorial/contents.html](https://www.cpp.edu/~jrfisher/www/prolog_tutorial/contents.html)

<sup>2</sup> компьютерный проект пятого поколения

<sup>3</sup> University of Edinburgh Prolog

Другой заметной версией *Prologa* является семейство реализаций *PrologII*, которые являются ответственными за Марсельского *Prologa*. Справочник Giannesini, et.al. (1986) использует версию *PrologII*. Есть некоторые различия между этими двумя вариантами *Prologa*; часть различий в синтаксисе, и часть в семантике. Тем не менее, студенты изучавшие одну из версий, впоследствии могут легко адаптировать к другой.

Цель этого учебника — помочь изучить самые необходимые, базовые концепции языка *Prolog*. Примеры программ были особенно аккуратно выбраны для иллюстрации программирования искусственного интеллекта на *Prolog*. *Lisp* и *Prolog* наиболее часто используемые языки символьного программирования для приложений искусственного интеллекта. Они часто упоминаются как великолепные языки для “исследовательского” и “прототипного программирования”.

Раздел 3.1 рассматривает среду программирования на *Prolog* для начинающих.

Раздел 3.2 объясняет синтаксис *Prologa* и многие аспекты программирования на нем через реализацию аккуратно выбранных программ-примеров. Эти примеры организованы так, чтобы студент обучался через реализацию *Prolog*-программ “сверху вниз” в декларативном стиле. Были приняты меры к рассмотрению техник программирования на *Prolog*, которые очень важны для курса искусственного интеллекта. Фактически, **этот учебник может служить удобным, маленьkim, кратким введением в применение Prologa в приложениях искусственного интеллекта**. Аспекты семантики языка *Prolog* рассматриваются с самого начала с точки зрения концепции дерева условий программы, которое используется для определения последовательностей спецификаций *Prolog*-программы в абстрактном виде. Автор надеется что этот подход позволит рассмотреть базовые принципы формальной верификации программ при программировании на *Prolog*. Последняя секция этого раздела приводит пример, который показывает что *Prolog* может быть эффективно использован для аккуратной, точной спецификации программных систем, несмотря на его репутацию трудно документируемого языка, так что *Prolog* легко использовать как средство прототипирования.

Раздел 3.3 рассматривает работу внутренних механизмов *Prolog*-движка. Раздел 3.3 рекомендуется просмотреть сразу после того, как студент изучил 2-3 примера программ из раздела 3.2. Последняя секция этого раздела рассматривает **мета-интерпретаторы Prologa**.

Раздел 3.4 дает краткий обзор основных встроенных предикатов, многие из которых используются в разделе 3.2..

Раздел 3.5 рассматривает разработку программ A\*-поиска на *Prolog*. Раздел 3.5.3 содержит программу  $\alpha\beta$ -поиска для игры tic tac toe.

Раздел 3.6 представляет уникальное и обширное описание логического мета-интерпретатора для нормальных логических баз правил.<sup>4</sup>

Раздел 3.7 представляет введение во встроенный в *Prolog* генератор парсеров

<sup>4</sup> Замечание от 9/4/2006: Я значительно отредактировал этот раздел, и обновил все ссылки на секции.

грамматики, и дает общий обзор приемов, с помощью которых *Prolog* может быть использован для разбора выражений натурального языка (английского). Также эта секция описывает построение программных интерфейсов, использующих идеоматически-простые натуральные языки.

Раздел 3.8 показывает приемы реализации различных *Prolog*-прототипов. Новый раздел 3.8.4 раскрывает интерактивную связку между машиной вывода *Prolog* и Java GUI для игры tic tac toe. Рассмотренная простая модель связки легко адаптируемая и применима.

Ранние версии частей этого учебника датируются 1988 годом. Вводный материал изначально использовался для объяснения работы интерпретатора *Prologa*, разработанного автором<sup>5</sup> для применения в учебном процессе. Автор надеется что вводный материал, собранный в форме этого учебника, может быть очень полезным для студентов, которые хотят быстрое, но при этом хорошо сбалансированное, введение в программирование на *Prolog*.

Для дальнейшего изучения *Prologa* можно посоветовать книги Clocksin и Melliss (1981,1992), O'Keefe (1990), Clocksin (1997, 2003), или Sterling и Shapiro (1986).

Подробные заметки по истории *Prolog* и обработке натуральных языков с его использованием содержатся в работе Pereira and Shieber (1987).

© Помона, Калифорния  
1988-2015

## 3.1 Установка и запуск *Prolog*-системы

Примеры этого учебника *Prologa* были подготовлены с использованием

- Quintus Prolog на компьютерах Digital Equipment Corporation MicroVAXes (далекая история)
- SWI Prolog на Sun Spark (давным давно)
- персональных компьютерах с Windows
- или OS X на Macах

Другие заметные *Prolog*-системы (Borland, XSB, LPA, Minerva . . . ) использовались для разработки и тестирования последние 25 лет. В этом учебнике запланирован новый раздел, в котором описано использование любых *Prolog*-систем в общем, но пока этот раздел недоступен.

Сайт SWI-Prolog содержит много информации, ссылки на загрузку, и документацию:

<http://www.swi-prolog.org/>

Особо следует отметить возможность попробовать SWI Prolog on-line без регистрации и SMS: <http://swish.swi-prolog.org/>. Этот вариант особенно удобен, так как не требует никакой установки ПО, административных прав, вы можете работать с этим учебником даже в интернет-кафе.

---

<sup>5</sup> сейчас недоступен

Примеры в этом учебнике используют упрощенную форму взаимодействия в типичным *Prolog*-интерпретатором, так что программы должны работать похоже в любой *Prolog*-системе эдинбургского типа или интерактивном компиляторе.

Если в вашей UNIX-системе уже установлен SWI-Prolog, запустите окно терминала, и начните интерактивную сессию командной:

```
user@computer$ swipl
```

Мы не будем использовать команду запуска именно в такой форме все время: при запуске могут быть указаны дополнительные параметры командной строки, которые можно использовать в определенных случаях. Читатель должен рассмотреть эту возможность после освоения базовых приемов работы, чтобы получить больше возможностей.

Если вы хотите установить SWI Prolog под Debian *Linux*, выполните команду:

```
sudo apt install swi-prolog
```

Под *Windows* инсталлятор SWI-Prolog добавляет иконку запуска интерпретатора, который вы можете запустить простым двойным щелчком по иконке. При запуске интерпретатор создает свое собственное командное окно.

После запуска интерпретатора обычно появляется сообщение о версии, лицензии и авторах, а затем выводится приглашение ввода *цели* типа

```
?- _
```

Интерактивные *цели* в *Prolog* вводятся пользователем за приглашением `?-.`

Многие *Prolog*-системы поддерживают предоставление документации по запросу из командной строки. В SWI Prolog встроена подробная система помощи. Документация индексирована, и помогает пользователю в процессе работы. Попробуйте ввести

```
?- help(help).
```

Обратите внимание что должна быть введены все символы, и ввод завершен возвратом каретки.

Для иллюстрации нескольких приемов взаимодействия с *Prolog* рассмотрим следующий пример сессии. Если приведена ссылка на файл, предполагается что это локальный файл в пользовательском каталоге, который был создан пользователем, получен копированием из другого публично доступного источника, или получен сохранением текстового файла из веб-браузера. Способ достижения последнего — следователь URL-ссылке на файл и сохранить его, или выбрать кусок текста из онлайн-учебника *Prologa*, скопировать его, вставить в текстовый редактор, и сохранить полученный файл из текстового редактора. Комментарии вида `/*...*/` после целей используются для описания этих целей.

## Листинг 1: Лог типичной Prolog-сессии

```
?- ['2_2.pl'].          /* 1. Загрузка программы из локального файла */
true.

?- listing(factorial/2). /* 2. Вывод листинга программы на экран */

factorial(0,1).

factorial(A,B) :-
    A > 0,
    C is A-1,
    factorial(C,D),
    B is A*D.

true.

?- factorial(10,What). /* 3. Вычислить 10! (в переменную) */
What=3628800 .          /* нажмите Enter */

?- ['2_7.pl'].           /* 4. Загрузить другую программу */

?- listing(takeout).

takeout(A,[A|B],B).
takeout(A,[B|C],[B|D]) :-
    takeout(A,C,D).

true.

?- takeout(X,[1,2,3,4],Y). /* 5. Взять X из списка [1,2,3,4] */
X=1  Y=[2,3,4] ;          Prolog ждет ... нужно ввести ';' и Enter
X=2  Y=[1,3,4] ;          следующий ...
X=3  Y=[1,2,4] ;          следующий ...
X=4  Y=[1,2,3] ;          следующий ...
false.                      Обозначает: больше нет ответов.

?- takeout(X,[1,2,3,4],_), X>3. /* 6. Конъюнкция целей */
X=4 ;
false.

?- halt.                  /* 7. Выход из интерпретатора в OS */
```

Комментарии в правой части были добавлены в текстовом редакторе. Они отмечают некоторые вещи, перечисленные ниже:

1. Определение *цели* Prologа завершается точкой . . В этом случае цель бы-

ла загружена в внешнего файла с исходным тестом программы. Этот скобочный стиль записи программы унаследован из самых первых реализаций *Prologa*. Можно загрузить несколько файлов сразу, указав их имена в одиночных кавычках, разделяя запятыми. В нашем случае имя файла **2\_2.pl**, программа содержит два программы на *Prolog* для вычисления факториала от положительного целого. Подробно эта программа описана в разделе [3.2.2](#). Файл программы ищется в текущем каталоге. Если поиск неуспешен, нужно явно указать полный путь обычным для вашей ОС способом.

2. Встроенный предикат *listing* выводит листинг программы из ОЗУ — в нашем случае программу вычисления факториала, загруженную ранее. Внешний вид этого литсинга несколько отличается от исходного кода в файле из [3.2.2](#). Заметим, что **Quintus Prolog** компилирует программу, если отдельно не указано что определенные предикаты являются динамическими. Скомпилированные предикаты не могут быть выведены через *listing*, поэтому если у вас он не срабатывает, возможно требуется дополнить исходник декларацией динамического предиката, чтобы пример сработал. В **SWI Prolog** этот пример работает без модификации.
3. Эта цель *factorial(10,What)* говорит “факториал 10ти что?”. Слово *What* начинается с большой буквы, указывающей что это **логическая переменная**. *Prolog* удовлетворяет цель находя все возможные значения переменной *What*.
4. Теперь в памяти находятся обе программы из файлов **2\_1.pl** и **2\_7.pl**. Файл **2\_7.pl** содержит несколько определений обработки списков (см. [3.2.7](#)).
5. Только что загруженная программа (**2\_7.pl**) содержит определение предиката *takeout*. Цель *takeout(X, [1,2,3,4], Y)* запрашивает поиск всех таких *X* что значение взятое из списка **[1,2,3,4]** оставляет остаток в переменной *Y*, для всех возможных случаев. Существует четыре способа сделать это, как показано в результате. Предикат *takeout* обсуждается в разделе [3.2.7](#). Таким образом, **в Prolog заложен поиск всех возможных ответов**: после того как будет выведен очередной ответ, *Prolog* ожидает реакции пользователя мигая курсором в конце строки с ответом. Если пользователь нажмет **;**, *Prolog* будет выполнять поиск следующего ответа. Если пользователь просто нажмет **Enter**, *Prolog* остановит поиск.
6. Составная, или **конъюнктивная цель**, определяет одновременное удовлетворение **двух** отдельных целей. Отметим что используется арифметическая цель (встроенное отношение) *X>3*. *Prolog* будет пытаться удовлетворить эти цели **слева направо**, в порядке чтения. В нашем случае существует единственный ответ. Отметим использование в цели **анонимной переменной \_**, **биндинг (привязка)** для которой не выводится (переменная “не важно”).

7. Цель `halt` всегда успешна и завершает работу интерпретатора.

## 3.2 Разбор примеров программ

В этом разделе мы рассмотрим несколько специально подобранных примеров программ на *Prolog*. Порядок примеров специально выбран от наиболее простых до более сложных. Ключевая цель — показать основные приемы *представления знаний* и методов декларативного программирования.

### 3.2.1 Раскраска карт

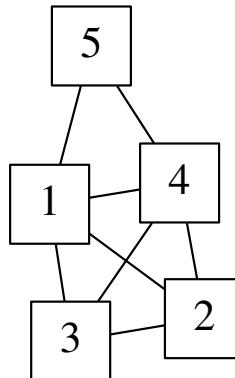
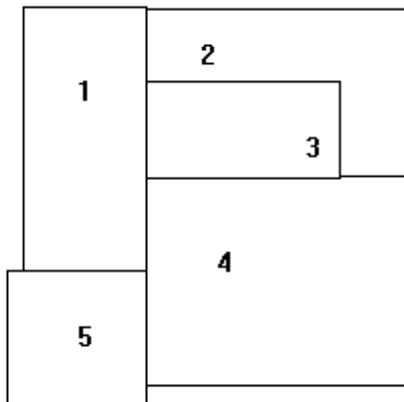
Этот раздел использует известную математическую проблему — **раскраска географических карт** — в качестве иллюстрации применения набора фактов и логических правил. Рассмотренная *Prolog*-программа показывает представление смежных регионов карты, ее раскраски, и определение конфликтов раскраски: когда **два смежных региона имеют одинаковый цвет**. Секция также показывает применение концепции **семантического дерева** и его применение в логическом программировании.

Согласно формулировке известной математической задачи по раскраске смежных плоских регионов<sup>6</sup>, необходимо подобрать минимум цветов раскраски, и цвета регионов, так что никакие два смежных региона не имеют один цвет. Два региона являются смежными, если они имеют некоторый общий сегмент границы, например<sup>7</sup>. По данным численным именам регионов строим представление в виде **графа смежности**:

---

<sup>6</sup> таких как географические карты

<sup>7</sup> упрощенно, только прямоугольные области



Мы удалили все границы, и нарисовали дугу между именами каждой двух смежных областей. Фактически граф смежности содержит полную оригинальную информацию о смежности областей. Для представления информации о смежности в синтаксисе *Prologa* запишем следующее:

---

adjacent (1 , 2).	adjacent (2 , 1).
adjacent (1 , 3).	adjacent (3 , 1).
adjacent (1 , 4).	adjacent (4 , 1).
adjacent (1 , 5).	adjacent (5 , 1).
adjacent (2 , 3).	adjacent (3 , 2).
adjacent (2 , 4).	adjacent (4 , 2).
adjacent (3 , 4).	adjacent (4 , 3).
adjacent (4 , 5).	adjacent (5 , 4).

это набор выражений устанавливает факт смежности  $A \rightarrow B$ : `adjacent(A,B)`.

Если загрузить этот файл в *Prolog*-систему, можно проверить работу целей:

```

?- adjacent(2,3).
true .
?- adjacent(5,3).
false .
?- adjacent(3,R).
R = 1 ;
R = 2 ;
R = 4 ;
false .

```

Аналогично можно задать два набора раскраски регионов используя единичные заключения: вариант **a** и вариант **b**:

```
color(1, red , a).      color(1 , red , b).
color(2 , blue , a).    color(2 , blue , b).
color(3 , green , a).   color(3 , green , b).
color(4 , yellow , a).  color(4 , blue , b).
color(5 , blue , a).    color(5 , green , b).
```

в форме

```
<имя отношения:color> (
  <номер зоны/узла графа>,
  <присвоенный цвет>,
  <имя раскраски>
).
```

Что обозначает **факт**: “имеется отношение `color` между номером узла, цветом и именем раскраски”<sup>8</sup>.

Теперь мы хотим написать *Prolog*-определение конфликта раскрасок, имея в виду совпадение цветов для двух регионов, например:

```
conflict(Coloring) :-  
  adjacent(X,Y),  
  color(X, Color , Coloring),  
  color(Y, Color , Coloring).
```

Например,

```
?- conflict(a).  
false .  
?- conflict(b).  
true .  
?- conflict(Which).  
Which = b .
```

Запрашивая отношение с неким значением-константой, или переменной<sup>9</sup> (последний случай), мы получаем от *Prolog*-системы заключение: выполняется ли запрошенное отношение-*целк* и при каких значениях переменных, имея в виду ранее

<sup>8</sup> причем не указывается какой элемент главный или подчиненный, все элементы отношения равноправны

<sup>9</sup> имя с большой буквы

определенный *набор фактов и отношений*<sup>10</sup>. В случае использования переменной *Prolog* выдаст нам **все** значения переменных, для которых запрос истинен.

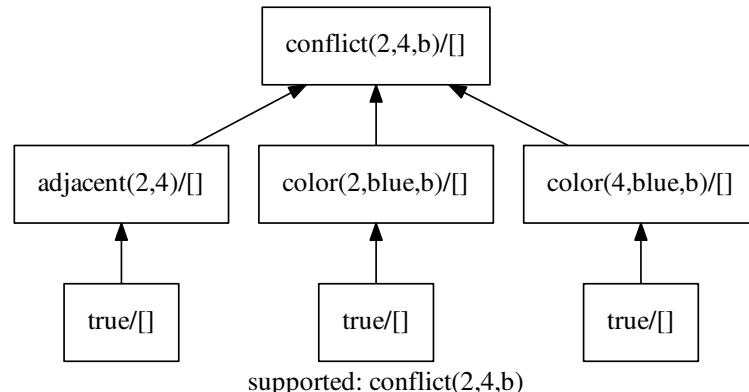
Можно определить другое отношение с тем же именем **conflict** но с другим количеством логических параметров:

```
conflict(R1,R2,Coloring) :-  
    adjacent(R1,R2),  
    color(R1,Color,Coloring),  
    color(R2,Color,Coloring).
```

*Prolog* позволяет отличать два отношения с одинаковым именем: одно имеет один параметр **conflict/1**, а другой — **conflict/3**.<sup>11</sup>

```
?- conflict(R1,R2,b).  
R1 = 2    R2 = 4  
?- conflict(R1,R2,b),color(R1,C,b).  
R1 = 2    R2 = 4    C = blue
```

Последняя *цель* значит что регионы 2 и 4 связаны (*adjacent*) и оба синие (*blue*). *Обоснованные* случаи, такие как **conflict(2,4,b)**, называются **консеквенцией** или **выводом** *Prolog*-программы. Один из способов демонстрации консеквенции — нарисовать **дерево заключений**, которое имеет консеквенцию в корне дерева, используя заключения программы для обхода дерева, получая в результате конечное дерево, в котором все листья имеют истинное значение. Например следующее дерево заключений может быть построено используя полностью обоснованные заключения программы без переменных:



Нижняя левая ветка дерева соответствует unit clause:

<sup>10</sup> которые являются *базой знаний*, или *экспертной системой*

<sup>11</sup> /цифра имеет название *аристотель*

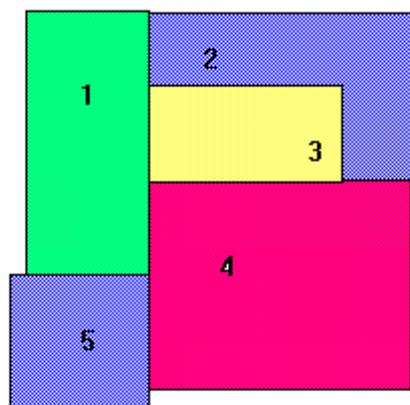
`adjacent(2,4).`

которая в *Prolog* эквивалента clause

`adjacent(2,4) :- true.`

С другой стороны `conflict(1,3,b)` не является consequence в *Prolog*-программе так как невозможно construct finite clause tree используя grounded clauses P содержащие все `true` листья. Аналогично `conflict(a)` не консеквенция, как можно ожидать. В последующих секциях clause деревья в subsequent sections описаны более подробно.

Мы повторно рассмотрим проблему раскраски карт в ??, где мы разработали *Prolog*-программу которая генерирует все возможные схемы раскраски<sup>12</sup>. Известная Гипотеза Четырех Цветов гласит что любая плоская карта требует для раскраски не более 4x цветов. Это было доказано в работе Appel и Haken (1976). Решение использовало компьютерную программу<sup>13</sup> для проверки всевозможных карт, с целью выявить возможные проблемные случаи. Следующая схема раскраски например требует не менее 4x цветов:



**Упражнение 2.1** Если карта имеет N регионов, определите сколько вычислений должно быть выполнено для определения есть ли конфликт раскраски. Аргументируйте используя program clause дерева.

### 3.2.2 Два определения факториала

Этот раздел вводит в вычисления математических функций используя *Prolog*. Обсуждаются различные встроенные арифметические операции. Также обсуждается концепция derivation дерева, и как derivation деревья связаны с трассировкой в *Prolog*.

<sup>12</sup> given colors to color with

<sup>13</sup> не на *Prolog*

В файле **2\_2.pl** находятся два определения предикатов, являющиеся определением функции вычисления факториала:

первый вариант

```
factorial(0,1).
```

```
factorial(N,F) :-  
    N>0,  
    N1 is N-1,  
    factorial(N1,F1),  
    F is N * F1.
```

Эта программа состоит из двух clauses. Первое заключение — формулировка **факта** (unit clause) **без тела**. Второе заключение — **правило**, так как **у него есть тело**. Тело второго заключения находится после `:-`, которое можно читать как “если”. Тело содержит литералы, разделенные запятыми, каждую запятую можно читать как “и”. **Заголовок правила** — весь текст **факта** или часть текста до `:-` в правиле. Рассматривая текст как декларативную программу, первое (фактическое) предложение читается как “факториал 0 есть 1”<sup>14</sup>, и второе предложение заявляет что “факториал  $N$  есть  $F$ <sup>15</sup> если  $N>0$  и  $N1$  есть  $N-1$ , и факториал  $N1$  есть  $F1$ , и  $F$  есть  $N*F1$ .

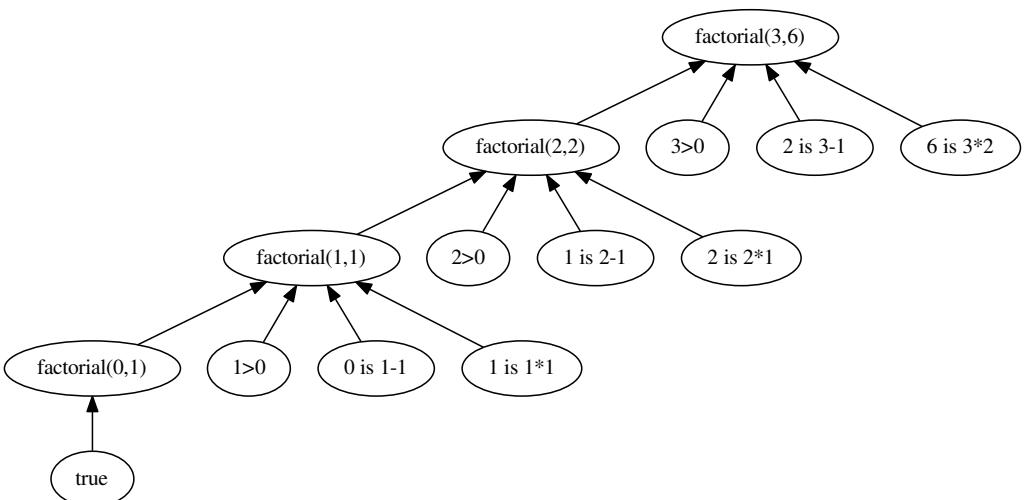
**Prolog-цель** (goal) для вычисления факториала от 3 дает ответ в W — **переменной цели**:

```
?- factorial(3,W).  
W=6 .
```

Рассмотрим следующее clause дерево сконструированное для литерала `factorial(3,W)`. Как описано в предыдущей секции, clause дерево не содержит никаких свободных переменных, вместо этого включает непосредственно их значения. Каждое ветвление под узлом определяется clause оригинальной программы, используя непосредственно вхождения значений переменных; узел задается заголовком правила, а литералы тела становятся дочерними узлами.

<sup>14</sup> или: 0 и 1 **связаны отношением** “факториал”, но у объектов одновременно могут быть и другие отношения, например биты(0,1) и целые(0,1)

<sup>15</sup> точнее: N и F связаны отношением “факториал”



**Все арифметические листья |true|** при исполнении<sup>16</sup>, и самая нижняя связь в дереве соответствует самому первому clause в программе вычисления факториала. Первый clause может быть записан как:

```
factorial(0,1) :- true.
```

и фактически `?- true.`. *Prolog*-цель которая всегда успешна<sup>17</sup>. Для краткости, мы не отрисовали `true` для всех листьев, являющихся арифметическими литералами.

Программное clause дерево показывает значение цели в корне дерева. Так, `factorial(3,6)` является консеквенцией *Prolog*-программы, так как существует clause дерево с корнем `factorial(3,6)`, все листья которого `true`. С другой стороны литерал `factorial(5,2)` не консеквенция, так как такого дерева для него нет, а значением программы для литерала `factorial(5,2)` является `false`:

```
?- factorial(3,6).
true .
?- factorial(5,2).
false .
```

как и следовало ожидать. Clause-деревья также называются AND-деревьями, так как чтобы корень был консеквенцией программы, все его поддеревья также должны быть консеквенциями. Позже clause деревья будут рассмотрены подробнее. Мы отметили что **clause дерево описывает семантику (значение) программы**. В разделе 3.6 мы рассмотрим другой подход к семантике программ. Clause-деревья представляют интуитивный и корректный подход к описанию семантики.

<sup>16</sup> в соответствии с предполагаемой интерпретацией

<sup>17</sup> `true` встроенный предикат

Нам нужно отличать clause деревья программы и **деревья вывода**. Clause-деревья статичны, и могут быть нарисованы для программы или цели через механизм удовлетворения частичных (под)целей, как описано выше. Грубо говоря, clause-деревья соответствуют декларативному чтению программы.

**Деревья вывода** наоборот, имеют в виду механизм привязки переменных *Prolog* и порядок в котором удовлетворяются вложенные частичные цели. Подробнее деревья вывода описаны в разделе [3.3.1](#), но тем не менее посмотрите анимацию, предоставляемую динамическим отладчиком, как описано ниже.

**Трассировка** исполнения *Prolog*-программы также показывает как переменные привязываются при удовлетворении целей. Следующий пример показывает включение/выключение трассировки в типичной *Prolog*-системе.

```
?- trace.  
% The debugger will first creep -- showing everything (trace).  
  
true .  
[trace]  
?- factorial(3,X).  
 (1) 0 Call: factorial(3,_8140) ? [Enter] creep  
 (1) 1 Head [2]: factorial(3,_8140) ? [Enter] creep  
 (2) 1 Call (built-in): 3>0 ? creep  
 (2) 1 Done (built-in): 3>0 ? creep  
 (3) 1 Call (built-in): _8256 is 3-1 ? creep  
 (3) 1 Done (built-in): 2 is 3-1 ? creep  
 (4) 1 Call: factorial(2, _8270) ? creep  
 ...  
 (1) 0 Exit: factorial(3,6) ?  
X=6 .  
[trace]  
?- notrace.  
% The debugger is switched off  
  
true .
```

The animated tree below gives another look at the derivation tree for the *Prolog* goal `factorial(3,X)`. To start (or to restart) the animation, simply click on the **Step** button.

Заголовок этого раздела говорит “**Два** определения факториала”, вот второй вариант, использующий три переменных:

второй вариант

```
factorial(0,F,F).
```

```
factorial(N,A,F) :-  
    N > 0,  
    A1 is N*A,  
    N1 is N -1,  
    factorial(N1,A1,F).
```

Для этой версии используйте следующую цель-запрос:

```
?- factorial(5,1,F).  
F=120 .
```

Второй параметр в определении называется *параметр-аккумулятор*, который также хорошо известен в функциональном программировании. Эта версия факториала определена с использованием *хвостовой рекурсии*. Важно чтобы вы выполнили следующие упражнения:

**Упражнение 3.2.2.1** Используя первый вариант программы факториала, четко покажите что не существует clause-дерева с корнем `factorial(5,2)`, имеющего все true листья.

**Упражнение 3.2.2.2** Нарисуйте clause-дерево для цели `factorial(3,1,6)` со всеми true-листьями, в виде аналогичном ранее описанному дереву для `factorial(1,1,1)`. Покажите, чем отличаются два варианта программы в процессе вычисления факториала? Также, протрассируйте цель `factorial(3,1,6)` используя Prolog-систему.

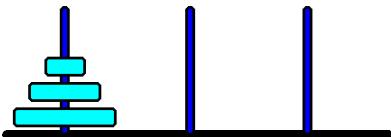
### 3.2.3 Классическая задача “Ханойские башни”

Показано формулирование и решение классической задача на *Prolog*. Рассмотрены декларативные и процедурные подходы к программированию. Решение задачи выводится на экран.

Цель известной головоломки — переместить  $N$  дисков с левого штыря на правый, используя центральный штырь как дополнительное хранилище. Требование: **нельзя класть больший диск на мénьший**. Следующая диаграмма показывает начальное положение для  $N=3$  дисков.

Рекурсивная программа на *Prolog*, решающая головоломку, состоит из двух утверждений:

## Ханойские башни



```
1 move(1 ,X,Y,_ ) :-  
2   write ( 'Move_top_disk_from_') ,  
3   write(X) ,  
4   write ( '_to_') ,  
5   write(Y) ,  
6   nl .  
7 move(N,X,Y,Z) :-  
8   N>1,  
9   M is N-1,  
10  move(M,X,Z,Y) ,  
11  move(1 ,X,Y,_ ) ,  
12  move(M,Z,Y,X) .
```

Переменная `_` (или любое другое имя начинающееся с подчеркивания) — переменные **don't-care** (не важно). *Prolog* позволяет использовать эти перемененные как обычные в любых структурах, но для них **не выполняется привязка**.

Вот что выводится при решении задачи при  $N=3$ :

```
?- move(3, left, right, center).  
Move top disk from left to right  
Move top disk from left to center  
Move top disk from right to center  
Move top disk from left to right  
Move top disk from center to left  
Move top disk from center to right  
Move top disk from left to right  
true .
```

Первое предложение программы описывает перемещение одного диска. Второе предложение описывает как можно получить решение рекурсивно. Например, декларативное чтение второго предложения для случая  $N=3$ ,  $X=left$ ,  $Y=right$ , и  $Z=center$  приводит к следующему:

```
move(3, left, right, center) если  
  move(2, left, center, right) и ] *  
  move(1, left, right, center) и  
  move(2, center, right, left). ] **
```

Это декларативное чтение очевидно правильно. Процедурное чтение тесно связано с декларативной интерпретацией рекурсивного утверждения, оно должно выглядеть как-то так:

удовлетворить цель `?-move(2, left, center, right)`, и потом

удовлетворить цель `?-move(1, left, right, center)`, и потом  
удовлетворить цель `?-move(2, center, right, left)`.

Аналогично мы можем записать декларативное прочтение для случая N=2:

```
move(2, left, center, right) если ] *
move(1, left, right, center) и
move(1, left, center, right) и
move(1, right, center, left).
move(2, center, right, left) если ] **
move(1, center, left, right) и
move(1, center, right, left) и
move(1, left, right, center).
```

Теперь подставим содержимое последних двух implications и увидим решение которое генерирует *Prolog*:

```
move(3, left, right, center) если
move(1, left, right, center) и
move(1, left, center, right) и *
move(1, right, center, left) и
-----
move(1, left, right, center) и
-----
move(1, center, left, right) и
move(1, center, right, left) и **
move(1, left, right, center).
```

Процедурное прочтение последних двух больших implication должно быть очевидно. Этот пример показывает при основных операции *Prologa*:

1. Цели сопоставляются с головой правила, и
2. тело правила (с соответствующе привязанными переменными) становится новой последовательностью целей; процесс повторяется
3. пока не будет удовлетворена основная цель или условие, или не будет выполнено простое действие, например выведен текст.

Процесс сопоставления переменных с образцом (variable matching)  
называется **унификацией**.

**Упражнение 3.2.3.1** Нарисуйте clause-дерево для цели `move(3, left, right, center)` и покажите что это консеквенция программы. Как полученное дерево связано с процессом подстановки, поисанным выше ?

**Exercise 3.2.3.2** Попробуйте *Prolog*-цель `?-move(3, left, right, left)`. Что не так? Предложите способ исправления, и проследите процесс работы исправления.

### 3.2.4 Загрузка, редактирование, хранение программ

Примеры показывают различные способы хранения и загрузки *Prolog*-программ, и пример вызова системного редактора. Читателю предлагается предварительно заглянуть в разделы 3.3.1, 3.3.2 чтобы иметь представление о том, как работает *Prolog*.

Стандартные предикаты для загрузки программ это `consult`, `reconsult`, и скобочная нотация загрузки `[ ... ]`. Например цель `?- consult('lists.pro')`. открывает файл `lists.pro` и загружает из него предложения в память.

Существует два способа, которыми *Prolog*-программа может быть неправильна:

1. исходный код имеет синтаксические ошибки, в этом случае при загрузке будут выводиться сообщения об ошибках, и
2. в программе есть какие-то логические ошибки, которые программист должен найти через тестирование программы.

Программа в ее текущей версии должна рассматриваться как прототип корректной версии в будущем, и принятая обычная практика редактирования текущей версии, и ее перезагрузка с повторным тестированием. Существуют хорошие приемы быстрого прототипирования, чтобы программист уделял все время и усилия на анализ проблемы. Интересно что если подход быстрого прототипирования кажется ошибочным, это отличный сигнал взять ручку и бумагу, еще раз проанализировать требования, и начать сначала!

Мы можем вызывать редактор непосредственно в *Prolog*:

```
?- edit('lists.pro'). %% редактор определенный пользователем, см. ниже ..
```

и после возврата из редактора<sup>18</sup> использовать цель

```
? reconsult('lists.pro').
```

для перезагрузки утверждений программы в память, автоматически замещая предыдущие определения. Если использовать `consult` вместо `reconsult`, старая<sup>19</sup> версия утверждений программы останется в памяти наряду с новыми определениями<sup>20</sup>.

Если в память было загружено несколько файлов, и требуется перезагрузить только один, используйте `reconsult`. Если перегружаемый файл определяет предикаты, которые не определяются в остальных файлах, перезагрузка не повлияет на кляузы, которые были загружены в остальных файлах.

Скобочная нотация очень удобна, например

<sup>18</sup> предполагается что новая версия файла была сохранена в том же файле

<sup>19</sup> и скорее всего неправильная

<sup>20</sup> это поведение зависит от конкретной версии *Prolog*-системы

```
?- ['file1.pro',file2.pro',file3.pro'] .
```

загрузит (точнее `reconsult`) все три файла в память *Prolog*-системы.

Многие *Prolog*-системы оставляют программисту определение любимого текстового редактора. Здесь описан пример программы, которая вызывает **TextEdit** на Mac(OSX)<sup>21</sup>.

```
edit(File) :-  
    name(File,FileString),  
    name('open -e ', TextEditString), %% укажите ваш любимый редактор  
    append(TextEditString,FileString,EDIT),  
    name(E,EDIT),  
    shell(E).
```

Для использования этого редактора, этот код должен быть загружен<sup>22</sup>

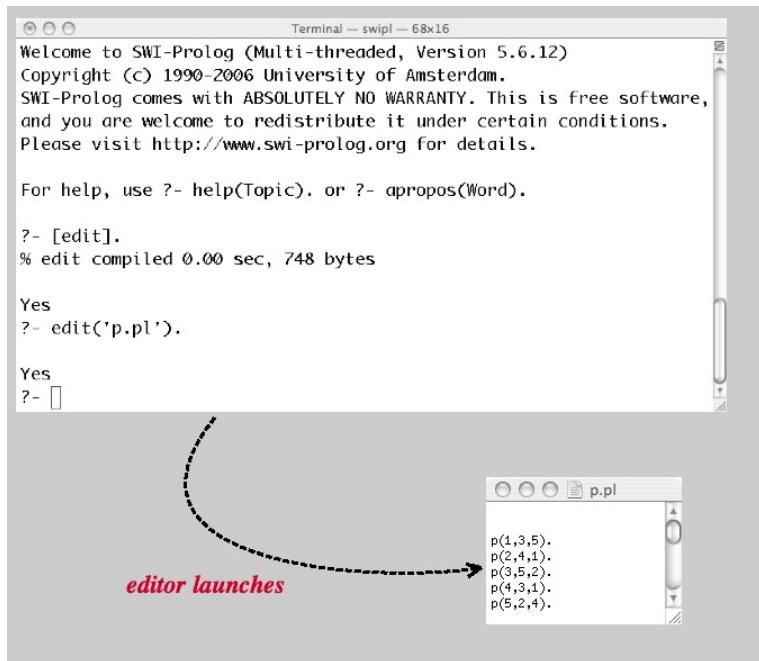
```
?- [edit].
```

```
yes
```

и цель `edit` может быть использована<sup>23</sup>

```
?- edit('p.pl').
```

```
{ TextEdit запускается с файлом, редактируйте его...}  
{ и сохраните измененную программу с тем же именем файла ... }
```



<sup>21</sup> это просто пример; мы не используем конкретно **TextEdit**

<sup>22</sup> предполагаем локальную *Prolog*-сессию

<sup>23</sup> опять же предполагаем что файл для редактирования локален для сессии

## Вызов внешнего редактора

После редактирования и сохранения мы можем перезагрузить новую версию

```
?- reconsult('p.pl').
```

{ наша prolog-сессия перезагружает программу для тестирования ...}

Для редактирования утверждений, введенных пользователем интерактивно, используйте цели

```
?-consult(user).  
?-reconsult(user).  
?- [user].
```

Пользователь вводит предложения интерактивно, используя символ останова . в конце набора утверждений, и сочетание клавиш **Ctrl**+**D** для окончания ввода.

**Упражнение 3.2.4** Проанализируйте как работает редактирование программы. Сначала попробуйте цели

```
?-name('name',NameString).
```

и

```
?- name(Name,"name").
```

`name/2` описана в разделе [3.4.13](#).

Теперь хороший момент для читателя немного заглянуть вперед и попробовать почитать первые две секции из раздела [3.3](#) “Как работает Prolog”, и затем вернуться к остальным примерам программ. Необходимо чтобы вы понимали как работают машина вывода *Prologa*, чтобы понять как конструируются следующие примеры программ.

### 3.2.5 2.5 Negation as failure

The section gives an introduction to *Prolog's* negation-as-failure feature, with some simple examples. Further examples show some of the difficulties that can be encountered for programs with negation as failure.

### 3.2.6 2.6 Tree data and relations

This section shows *Prolog* operator definitions for a simple tree structure. Tree processing relations are defined and corresponding goals are studied.

### **3.2.7 2.7 Prolog lists and sequences**

This section contains some of the most useful Prolog list accessing and processing relations. Prolog's primary dynamic structure is the list, and this structure will be used repeatedly in later sections.

### **3.2.8 2.8 Change for a dollar**

A simple change maker program is studied. The important observation here is how a *Prolog* predicate like 'member' can be used to generate choices, the choices are checked to see whether they solve the problem, and then backtracking on 'member' generates additional choices. This fundamental generate and test strategy is very natural in *Prolog*.

### **3.2.9 2.9 Map coloring redux**

We take another look at the map coloring problem introduced in Section 2.1. This time, the data representing region adjacency is stored in a list, colors are supplied in a list, and the program generates colorings which are then checked for correctness.

### **3.2.10 2.10 Simple I/O**

This section discusses opening and closing files, reading and writing of *Prolog* data.

### **3.2.11 2.11 Chess queens challenge puzzle**

This familiar puzzle is formulate in *Prolog* using a permutation generation program from Section 2.7. Backtracking on permutations produces all solutions.

### **3.2.12 2.12 Finding all answers**

*Prolog's* 'setof' and 'bagof' predicates are presented. An implementation of 'bagof' using 'assert' and 'retract' is given.

### **3.2.13 2.13 Truth table maker**

This section designs a recursive evaluator for infix Boolean expressions, and a program which prints a truth table for a Boolean expression. The variables are extracted from the expression and the truth assignments are automatically generated.

### **3.2.14 2.14 DFA parser**

A generic DFA parser is designed. Particular DFAs are represented as *Prolog* data.

### **3.2.15 2.15 Graph structures and paths**

This section designs a path generator for graphs represented using a static *Prolog* representation. This section serves as an introduction to and motivation for the next section, where dynamic search grows the search graph as it works.

### **3.2.16 2.16 Search**

The previous section discussed path generation in a static graph. This section develops a general *Prolog* framework for graph searching, where the search graph is constructed as the search proceeds. This can be the basis for some of the more sophisticated graph searching techniques in A.I.

### **3.2.17 2.17 Animal identification game**

This is a toy program for animal identification that has appeared in several references in some form or another. We take the opportunity to give a unique formulation using *Prolog* clauses as the rule database. The implementation of verification of askable goals (questions) is especially clean. This example is a good motivation for expert systems, which are studied in Chapter 6.

### **3.2.18 2.18 Clauses as data**

This section develops a *Prolog* program analysis tool. The program analyses a *Prolog* program to determine which procedures (predicates) use, or call, which other procedures in the program. The program to be analyzed is loaded dynamically and its clauses are processed as first-class data.

### **3.2.19 2.19 Actions and plans**

An interesting prototype for action specifications and plan generation is presented, using the toy blocks world. This important subject is continued and expanded in Chapter 7.

## **3.3 Как работает *Prolog***

### **3.3.1 Деривационные деревья, выборы и унификация**

Для иллюстрации того, как *Prolog*-программа создает ответы, рассмотрим следующую простую программу регистрации данных (это не функции):

Листинг:

```
/* program P                                cause # */  
p(a).  
p(X) :- q(X), r(X).  
                                /* #1 */  
                                /* #2 */
```

```

p(X) :- u(X).          /* #3 */

q(X) :- s(X).          /* #4 */

r(a).                  /* #5 */
r(b).                  /* #6 */

s(a).                  /* #7 */
s(b).                  /* #8 */
s(c).                  /* #9 */

u(d).                  /* #10 */

```

---

**Упражнение 3.3.1.1** Загрузите программу **P** в *Prolog* и посмотрите что случится при вводе цели `?-p(X)`. Когда будет выведен ответ, нажмайте  чтобы *Prolog* продолжил трассировку и нашел все ответы.

**Упражнение 3.3.1.2** Загрузите программы, включите трассировку, и посмотрите что происходит при вводе той же цели. Нажмайте **Enter** в каждой строке трассировки, и  в конце строки ответа, чтобы найти все ответы. Используйте `?-help(trace)` если необходимо.

## Листинг 2: Трассировка

```

?- trace.
true.

[trace] ?- p(X).
Call: (6) p(_G2873) ? [Enter] creep
Exit: (6) p(a) ? [Enter] creep
X = a ; []
Redo: (6) p(_G2873) ? creep
Call: (7) q(_G2873) ? creep
Call: (8) s(_G2873) ? creep
Exit: (8) s(a) ? creep
Exit: (7) q(a) ? creep
Call: (7) r(a) ? creep
Exit: (7) r(a) ? creep
Exit: (6) p(a) ? creep
X = a ;
Redo: (8) s(_G2873) ? creep
Exit: (8) s(b) ? creep

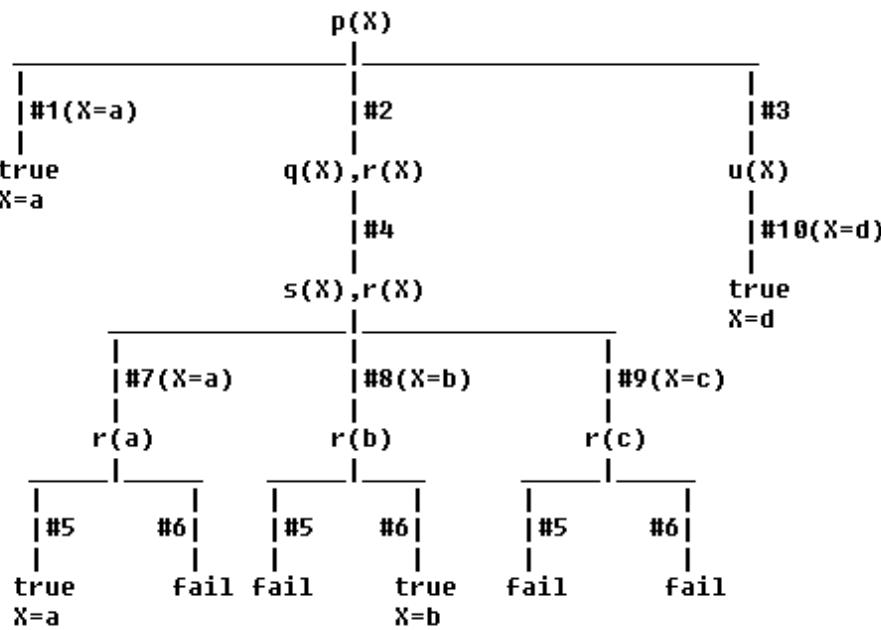
```

```

Exit: (7) q(b) ? creep
Call: (7) r(b) ? creep
Exit: (7) r(b) ? creep
Exit: (6) p(b) ? creep
X = b .

```

Следующая диаграмма показывает полное **дерево вывода** для цели  $?-p(X)$ . Ребра помечены номером утверждения в исходном файле программы **P**, которое было использовано для подмены цели подцелями. Прямые потомки под каждой (под)целью в дереве вывода соответствуют **вариантам выбора**. Например корневая цель  $p(X)$  **унифицируется** заголовками утверждений #1, #2, #3, порождая три выбора.

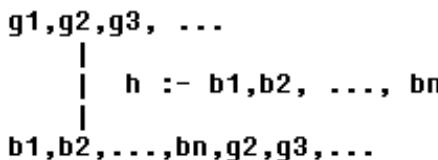


Трассировка упражнения 3.3.1.2 для цели  $?-p(X)$  соответствует обходу дерева вывода вглубь. Каждый узел дерева вывода *Prologa* в определенный момент времени становится текущей целью. Аналогично каждый узел — последовательность субцелей. Ребра сразу ниже узла соответствуют доступным выборам замены для текущего узла. Текущий side clause, номер которого отмечает дугу в дереве вывода<sup>24</sup>, тестируется следующим способом: если самая левая подцель текущего узла<sup>25</sup> унифицируется головой side clause<sup>25</sup>, затем самая левая подцель заменяется телом side clause<sup>26</sup>. Графически мы можем это показать вот так:

<sup>24</sup> отмечена как **g1** в небольшой диаграмме ниже

<sup>25</sup> отмечена как **h** в диаграмме

<sup>26</sup> **b1,b2,...,bn**



Одна важная вещь не показана в диаграмме — логические переменные в результирующей цели  $b_1, b_2, \dots, b_n, g_2, g_3, \dots$  были привязаны в результате унификации, и *Prolog* требует отслеживать эти унифицирующие подстановки, в процессе роста дерева вывода вниз, во всех ветках.

Итак, обход дерева вывода вглубь значат что альтернативные варианты выбора будут проверены тогда, когда поиск возвратиться в точку ветвления, содержащую этот выбор. Процесс называется **backtracking**.

Естественно, если хвост цели пуст, самая левая подцель эффективно удаляется. Если все подцели могут быть удалены по одному из путей дерева вывода, то находится ответ, и возвращается результат **true**. В этой точке привязки переменных могут быть использованы для дачи ответа на оригинальный запрос.



- 3.3.2 3.2 Cut
- 3.3.3 3.3 Meta-interpreters in *Prolog*
- 3.4 4. Built-in Goals
  - 3.4.1 4.1 Utility goals
  - 3.4.2 4.2 Universals (true and fail)
  - 3.4.3 4.3 Loading *Prolog* programs
  - 3.4.4 4.4 Arithmetic goals
  - 3.4.5 4.5 Testing types
  - 3.4.6 4.6 Equality of *Prolog* terms, unification
  - 3.4.7 4.7 Control
  - 3.4.8 4.8 Testing for variables
  - 3.4.9 4.9 Assert and retract
  - 3.4.10 4.10 Binding a variable to a numerical value
  - 3.4.11 4.11 Procedural negation, negation as failure
  - 3.4.12 4.12 Input/output
  - 3.4.13 4.13 *Prolog* terms and clauses as data
  - 3.4.14 4.14 *Prolog* operators
  - 3.4.15 4.15 Finding all answers
- 3.5 5. Search in *Prolog*
  - 3.5.1 5.1 The A\* algorithm in *Prolog*
  - 3.5.2 5.2 The 8-puzzle
  - 3.5.3 5.3  $\alpha\beta$  search in *Prolog*
- 3.6 6. Logic Topics
  - 3.6.1 6.1 Chapter 6 notes
  - 3.6.2 6.2 Positive logic
  - 3.6.3 6.3 Convert first-order logic to normal form
  - 3.6.4 6.4 A normal rulebase goal interpreter

# Глава 4

## ASTLOG: Язык для анализа синтаксических деревьев

<sup>1</sup> © Roger F. Crew <[rfc@microsoft.com](mailto:rfc@microsoft.com)>  
Microsoft Research Microsoft Corporation Redmond, WA 98052

### Abstract

We desired a facility for locating/analyzing syntactic artifacts in abstract syntax trees of Си/ $C_+^+$  programs, similar to the facility **grep** or **awk** provides for locating artifacts at the lexical level. *Prolog*, with its implicit pattern-matching and backtracking capability is a natural choice for such an application. We have developed a *Prolog* variant that avoids the overhead of translating the source syntactic structures into the form of a *Prolog* database; this is crucial to obtaining acceptable performance on large programs. An interpreter for this language has been implemented and used find various kinds of syntactic bugs and other questionable constructs in real programs like **Microsoft SQL server** (450Klines) and **Microsoft Word** (2Mlines) in time comparable to the runtime of the actual compiler.

The model in which terms are matched against an implicit current object, rather than simply proven against a database of facts, leads to a distinct “inside-out functional” programming style that is quite unlike typical *Prolog*, but one that is, in fact, well-suited to the examination of trees. Also, various second-order *Prolog* set-predicates may be implemented via manipulation of the current object, thus retaining an important feature without entailing that the database be dynamically extensible as the usual implementation does.

---

<sup>1</sup> © <http://www.cs.nyu.edu/~lharris/papers/crew.pdf>

## 4.1 Introduction

Tools like **grep** and **awk** are useful for finding and analyzing lexical artifacts; e.g., a one-line command locates all occurrences of a particular string. Unfortunately, many simple facts about programs are less accessible at the character/token level, such as the locations of assignments to a particular  $C_+^+$  class member. In general, reliably extracting such syntactic constructs requires writing a parser or some fragment thereof. And after writing one's twenty-seventh parser fragment, one might begin to yearn for a more general tool capable of operating at the syntax-tree level.

Even given a compiler front-end that exposes the abstract syntax tree (AST) representation for a given program, there remains the question of what exactly to do with it. To be sure, supplying a Си programmer with a sufficiently complete interface to this representation generally solves any problem one might care to pose about it. One may just as easily say that all problems at the lexical level may be solved via proper use of the UNIX standard IO library `<stdio.h>`, a true, but utterly trivial and unsatisfying statement. The question is rather that of building a simpler, more useful and flexible interface: one that is less error-prone, more concise than writing in Си, and more directly suited to the task of exploring ASTs. We first consider a couple of prior approaches.

### 4.1.1 The **awk** Approach

One of the more popular approaches is to extend the **awk** [?] paradigm. An **awk** script is a list of pairs, each being a regular-expression with an accompanying statement in a C-like imperative language. For each line in the input file, we consider each pair of the script in turn; if the regular-expression matches the line, the corresponding statement is executed.

Extending this to the AST domain is straightforward, though with numerous variations. One defines a regular-expression-like language in which to express tree patterns and an **awk**-like imperative language for statements. The tree nodes of the input program are traversed in some order (e.g., preorder), and for each node the various pairs of the script are considered as before.

We have two objections to this approach, the first having to do with the hardwired framework that usually implicit. In some cases (e. g., **TAWK** [?]), the traversal order for the AST nodes is essentially fixed; using a different order would be analogous to attempting to use plain **awk** to scan the lines of a text file in reverse order. In **A\*** [?], while the user may define a general traversal order, only one traversal method may be defined/active at any given time, making difficult any structure comparisons between subtrees or other applications that require multiple concurrent traversals. Since the imperative language is quite general in both cases, little is deffinitively impossible, however for some applications one may be little better off than when programming in straight Си.

The second objection has to do with the kinds of pattern-abstraction available. Inevitably there exist simply-described patterns that are a poor fit to a regular-

expression-like syntax. This tends to happen when said simple descriptions are in terms of the idioms of a particular programming language; most of the various tree-**awk** pattern languages tend to be designed with the intent of being language independent.

Suppose one wishes to find all consecutive occurrences of one statement immediately preceding another, e. g., places where a given system call `syscall()`; is followed immediately by an `assert()`; <sup>2</sup>. A tree-regular-expression pattern of the form

```
<syscall() pattern>; <assert() pattern>
```

(where ; is the regular-expression sequence operator) finds all instances of the two calls occurring consecutively within a single block, but it misses instances like

```
syscall();  
{  
    assert();  
    ...  
}
```

and

```
if (...) {  
    syscall();  
}  
else {  
    ...  
}  
assert();
```

While the tree-**awk** languages allow one to write patterns to match each of these cases, without a pattern-abstraction facility, we may be back at square one when it comes time to look for some **different** pair of consecutive function calls. We prefer to write a single consecutive-statement pattern constructor **once** and then be able to use it for a variety of cases where we need to find pairs of consecutive statements satisfying certain criteria, invoking it as

```
follow_stmt(<syscall() pattern>, <assert() pattern>)
```

for the above problem, or, if we instead want to be finding all of the places where a C switch-case falls through, as

```
follow_stmt(not(<unconditional-jump pattern>),  
            <case-labeled stmt pattern>)
```

---

<sup>2</sup> on the theory that testing of outcomes of system calls should be done in production code rather than just debugging code

One solution, used by **TAWK**, is to use **cpp**, the C preprocessor, to preprocess the script, allowing for pattern-abstractions to be expressed as `#define` macros whose invocations are then expanded as needed. This is unsatisfactory in a number of ways, whether one wants to consider the problem of recursively-defined patterns, macros with large bodies that result in a corresponding blow-up in the size of the script, or the difficulty of tracing script errors that resulted from complex macro-expansions.

Another way out is to fall back on the procedural abstraction available in the imperative language that the patterns invoke. One essentially uses a degenerate pattern that always matches and then allows the imperative code to test whether the given node is in fact the desired match, defining functions to test for particular patterns. Once again, it seems we are back to programming in straight C and not deriving as much benefit from having a pattern language available as we could be.

In general, the philosophical underpinning of the **awk** approach is that the designer has already determined the kinds of searches the user will want to do; the effort is put towards making those particular searches run efficiently. There is also an assumption that the underlying imperative language for the actions has all the abstraction facilities one will ever need, so that if the pattern language is lacking in various ways, this is not deemed a serious problem. While this is not an unreasonable approach, we have less confidence of having identified all of the reasonable search possibilities, and thus would prefer instead to make the pattern language more flexible and extensible, being willing to sacrifice some efficiency to do so.

#### 4.1.2 The Logic Programming Approach

Another common approach is to run an inference engine over a database of program syntactic structures [?, ?, ?]. *Prolog* [?] is a convenient language for this sort of application. Backtracking and a form of pattern matching are built in, the abstraction mechanisms to build up complex predicates exist at a fundamental level, and finally, *Prolog* allows for a more declarative programming style.

The problems with using *Prolog* are two-fold. First there is the issue of efficiency. Second, we must represent the AST for our source program in the *Prolog* database. Large programs ( $10^5..10^6$  lines) will result in correspondingly large *Prolog* databases, most likely with a significant performance penalty.

We finesse the second problem by not attempting to import the source program's AST at all, instead opting to modify the interpretation of the predicates and queries of *Prolog* so as to be applicable to external objects rather than just facts provable in the existing database. Removing reasons that require the database to grow beyond the initial script creates significant opportunities for optimization. This, however, requires removing primitives like `assert()` and `retract()` that allow for the dynamic (re)definition or removal of predicates, which in turn removes many higher-order logical features that are defined in terms of them. Fortunately, some of the more essential ones can be restored at relatively little cost.

## 4.2 Elements of ASTLOG

Section 4.2 gives the complete syntax for our language, ASTLOG. The ASTLOG interpreter reads a script of user-defined predicate operator definitions and then runs one or more queries.

As in *Prolog*, the definition of a user-defined predicate operator is composed of one or more *clauses*. A compound term `opname(term, ...)` appearing at top level in a clause body is interpreted as a predicate, whether `opname` be primitive or user-defined. In the latter case, the script is searched for a defining clause whose head terms successfully unify with the respective operand terms of the given compound term, variables are bound accordingly, and the terms of the clause body are likewise interpreted. The clause **succeeds** (i. e., is found to be true) if all of its body terms succeed. Whenever a clause head fails to unify, or a clause body term **fails** (i. e., is found to be false), or any primitive term fails by the rules of evaluation of that primitive, we backtrack to the last point where there was a choice (e. g., of clauses to try for a given compound term) and continue.

A *query* is a clause whose head terms are all variables. Ultimately, whenever all terms of a query body succeed, the bindings of any variables listed in the query head (*qhead*) are reported. Otherwise, we report failure. Thus far, this is all exactly like *Prolog*.

**Figure 1: Complete Syntax of ASTLOG**

script	::= named-clause*	script file syntax
query	::= imports? ( varname* ) clause-body ;	query syntax
imports	::= { varname+ }	
named-clause	::= opname anon-clause	
anon-clause	::= ( term* ) clause-body? ;	
clause-body	::= <- term+	
<hr/>		
Essential Term Syntax		
term	::= literal	reference to denotable object
	::= varname	
	::= opname ( term* )	compound term
	::= FN imports? ( anon-clause+ )	anonymous predicate-operator-valued (“lambda”) term
	::= ’ opname arity-spec?	named predicate-operator-valued (“function quote”) term
	::= ( term )( term* )	“application” term

## Gratuitous Term Syntax

<code>::= # constname</code>	named constant ( $\equiv$ corresponding literal number)
<code>::= [ term* ]</code>	<code>[ ]</code> $\equiv$ nil(), <code>[term]</code> $\equiv$ cons(term; nil()), etc..
<code>::= [ term+   term ]</code>	<code>[ term1   term2 ]</code> $\equiv$ cons(term1,term2), etc.
arity-spec	<code>::= / integer</code>

### 4.2.1 Objects

ASTLOG refers to external objects. Given a Cи/C<sub>+</sub><sup>+</sup> compiler front end that provides a (C<sub>+</sub><sup>+</sup>) interface to the syntactic/semantic data structures built during the parse of a given program, it is simple to graft this onto the core of ASTLOG so that it may recognize object references corresponding to

- whole C/C++ programs,
- single files,
- symbols,
- AST nodes (including statements, expressions, and declarations), and
- Cи/C<sub>+</sub><sup>+</sup> type descriptions.

For the purposes of ASTLOG, an *object* is simply some external entity that is significant for its identity and for the primitive predicates that it may satisfy. To simplify the language we regard the traditional constants (integers, floats, and strings) to be references to “external” objects as well, though one could just as easily take the converse view in which the universe of object references is just a (very large) pool of constants<sup>3</sup>.

In any case, object references are terms in ASTLOG. Only references to equal objects can unify, equality meaning numeric equality for numbers, same-sequence-of-characters for strings, and identity for all other classes of objects. Only objects that have denotations (numbers, strings and the unique `null object*`) can find their way into scripts.

### 4.2.2 The Current Object

The first significant departure from the *Prolog* model is that a query or predicate term always evaluates under an ambient *current object*. Every query and every term being evaluated as a predicate is not so much a standalone statement that may or may not be intrinsically true (i. e., provable from the “facts” in the script) as it is a specification that may or may not be satisfied by the current object, or, alternatively, a *pattern* that may or may not *match* the current object. For example, in *Prolog*

```
odd(3)
```

always succeeds by virtue of 3 being odd or because the “fact” `odd(3)` exists in the script somewhere. By contrast, in ASTLOG

---

<sup>3</sup> “atoms” in the usual *Prolog* terminology

`odd()`

succeeds if the current object happens to be the integer 3, fails if the current object is 4, and raises an error if the current object is the string "Hi mom". Another way to view this is that every predicate term takes an extra, hidden current-object operand.

While one normally only expects to see compound (and application) terms in predicate position, ASTLOG allows variables and object references there as well. The rules for matching are as follows:

- An object reference matches the current object, if it references an equal object.
- A bound variable matches according as whatever term it is bound to.
- An unbound variable gets bound to reference the current object (and thus automatically matches it).
- A compound term whose operator is defined via clauses matches if there exists a clause whose head operands unify with the term operands and whose body terms themselves all match the current object.

Section 4.3.1 describes the operator-valued and application terms.

The evaluation rules for compound terms having primitive operators are widely varied, however the operands are usually treated one of two ways:

1. (`foo-pred`) requiring the operand to be match some object<sup>4</sup>, not necessarily the same current object as that which the full term is being matched against. For example, the operand of `strlen` (see 4.2.2) and the second operand of `with` are treated this way.
2. (`foo`) requiring the operand be an object reference, whether this be a literal or an object-reference-bound variable. The operands of `re`, `gt`, and the first operand of `with` are treated this way.

Most primitives also expect a current object to be of a particular kind and raise an error if confronted with something different.

The use of an implicit current object is not by itself an increase in expressivity. If we had, in a *Prolog* database, terms representing the various AST nodes, there would be a fairly straightforward translation of ASTLOG terms into *Prolog* terms, one in which we simply modify all terms to make the current object an explicit operand.

Nevertheless, ASTLOG programs exhibit a distinct style of programming. Consider as an example that we might, in a typical functional language, write a function call

`strlen(string)`

---

<sup>4</sup> which becomes the current object for that evaluation

to find the length of the string returned by the expression `string`. Here the length result is implicitly returned to the context of the call. In *Prolog*, the natural style would be to express this as a relation

```
strlen(string, length)
```

which stipulates that `length` is in fact the length of `string`. In ASTLOG, we would write

```
strlen(length-pred)
```

where now it is the `string` argument that is implicitly supplied **as the current object** by the context while the length result is returned *to* the subterm `length-pred`, which in turn can be some arbitrary term expecting a numeric current object as its implicit argument. For example, given an `odd()` predicate as above, the term `strlen(odd())` would match any string consisting of an odd number of characters. It is this “inside-out functional” evaluation strategy that makes ASTLOG well-suited to constructing anchored patterns to match tree-like structures.

## Figure 2: Some core ASTLOG primitives

- `and(object-pred, ... )`  
The current object satisfies every `object-pred` operand.
- `or(object-pred, ... )`  
The current object satisfies some `object-pred` operand.
- `if(object-pred, then-pred, else-pred)`  
The current object satisfies `then-pred` or `else-pred` according as it satisfies or fails to satisfy `object-pred` (once; if `object-pred` matches but `then-pred` does not, we do not retry `object-pred`).
- `not(object-pred)`  
= `if(object-pred, or(), and())`
- `with(object, object-pred)`  
`object` satisfies `object-pred` (outer current object is ignored).
- `strlen(integer-pred)`  
The current string object has length satisfying `integer-pred`.
- `re(string)`  
The regular expression `string` matches the current string.
- `gt(integer)`  
The current integer is greater than `integer`.

- `minus(integer-pred, integer)`  
`integer-pred` matches the current integer + `integer`.
- `minus(integer, integer-pred)`  
`integer-pred` matches `integer` — the current integer.  
(An error is raised if neither operand of a minus term is an integer object reference.)
- `plus(integer-pred, integer)`  
`integer-pred` matches the current integer — `integer`

### Figure 3: Some primitive node and symbol predicates

- `parent(ast-pred)`  
This AST node is not a root node and its parent satisfies `ast-pred`.
- `kid(integer-pred; ast-pred)`  
This AST node has a child satisfying `ast-pred` whose (0-based) index satisfies `integer-pred`.
- `kidcount(integer-pred)`  
The number of children of this AST node satisfies `integer-pred`.
- `op(integer-pred)`  
The opcode of this AST node satisfies `integer-pred`.
- `atype(type-pred)`  
This AST node has a return type satisfying `type-pred`.
- `asym(symbol-pred)`  
This AST node is a symbol satisfying `symbol-pred`.
- `aconst(const-pred)`  
This AST node is a constant (integer, float or string) satisfying `const-pred`.
- `sname(string-pred)`  
This symbol's name satisfies `string-pred`.

There are named constants available for designating the opcodes of various kinds of nodes for use in `op()` terms, and the indices of particular children for use in `kid()`.

#### 4.2.3 Examples

Given the set of AST node primitives in Figure 3, we could write

```
and(op(#=), kid(#LEFT, asym(sname("foo"))))
```

which would be satisfied by any AST node that is an assignment (=) expression whose left-hand side is itself a symbol expression where the symbol name is "foo". Here, #= and #LEFT are numeric literals for the assignment node opcode and the assignment target's childindex, respectively.

To define a predicate `assignment/2` to match assignment nodes, a script could include the clause

```
assignment(target, value)
  <- op(#=),
      kid(#LEFT, target),
      kid(#RIGHT, value);
```

which would then allow writing the previous term as

```
assignment(asym(sname("foo")), _)
```

As in *Prolog*, the underscore (\_) is “wild-card” variable, i.e., one that is internally given a distinct identity so as not to be conflated with any other instances of \_. Such a variable, being guaranteed to be unbound, will match any object or unify with any term.

Defining a general purpose node-traversal predicate is also straightforward

```
somenode(pred)
  <- or(pred, kid(_, somenode(pred))));
```

Given this definition, an attempt to match `somenode(test)` to a given node will create an instance of the defining clause of `somenode/1` above with `pred` bound to `test`. Satisfying the clause body requires that either `pred` match the current node, or, if (when) that fails, that `kid(_, somenode(pred))` match the current node. The latter in turn will attempt to match the variable `_` with 0 (easy) and the term `somenode(pred)` with the first child, and, when that fails, `_` with 1 and `somenode(pred)` with the second child, etc... Making the interpreter fail and backtrack after each hit (in the usual manner of *Prolog*) eventually causes `test` to be matched with the original node and all of its descendants.

So, if we issue the query

```
(v) <- somenode(
  assignment(asym(sname("foo")), v)
);
```

on the root node of some function’s AST, we obtain, via the successive bindings reported for `v` on each hit, all of the expressions assigned to variables named "foo" within that function.

As an example that makes less trivial use of backtracking, consider the problem of whether two trees have the same structure (i.e., root nodes have the same opcode and all corresponding children have the same structure).

```
sametree(node)
<- op(nodeop),
  with(node, op(nodeop)),
  not(and(with(node, kid(n, nkid)),
    kid(n, not(sametree(nkid)))));
```

This defines a predicate `sametree(node)` that holds if `node` is a reference to an AST node with the same structure as the current object. The first line of the clause body binds the current node's opcode to `nodeop`, the second line compares that to the opcode of `node`, while the remaining lines search for children whose subtrees have distinct structure. The term `kid(n, nkid)` will match each child of `node`, since both variables are initially unbound. If `sametree(nkid)` happens to be true of the corresponding child of the current node, the inner `not` fails and we go back and try another child of `node`. If `sametree(nkid)` happens to be true of **every** corresponding child of the current node, then the enclosing `not` and thus the outer `sametree(node)` invocation succeeds.

The preceding version of `sametree/1` is a purely structural comparison; there is no attempt to take account of the commutativity/associativity of the various operators, e. g., `a + b` and `b + a` are not considered the same. If, say, we **did** want to consider commutativity, we could define

```
csametree(node)
<- op(nodeop),
  with(node, op(nodeop)),
  kidcount(if(with(nodeop, commutes()),
    any_perm(perm),
    id_perm(perm))),
  not(and(with(node, kid(corresp(perm, n),
    nkid)),
    kid(n, not(csametree(nkid)))));
```

along with suitable definitions of

### `commutes()`

the current integer is the opcode of a commutative operator,

### `any_perm(perm)`

`perm` is any permutation of the sequence  
`0, ... , (<current-object> - 1),`

### `id_perm(perm)`

`perm` is the identity permutation of the sequence  
`0, ... , (<current-object> - 1),`

### `corresp(perm, n)`

permutation `perm` takes the current integer to something matching `n`.

Here, permutations can be represented by list terms. Note that since all of the commutative  $C_+^+$  operators are, in fact, binary, this all simplifies significantly.

It should, incidentally, be clear that there is nothing about the core language that is specifically tailored for the examination of compiler-produced ASTs, let alone ASTs for a given language. The language in fact lends itself to the examination of a wide variety of external structures, e. g., hierarchical file systems, or collections of web pages. All that is needed is a suitable collection of primitive ASTLOG predicates for querying said structures.

## Figure 4: Actual ASTLOG code for follow\_stmt

Actual ASTLOG code for `follow_stmt` and how one uses it to find case statement fallthroughs. The `cond` operator is an if-then-elseif- construct, that is, `cond(p1, e1, p2)` is equivalent to `if(p1, e1, if(p2, e2, ..., e))`. `sfa(emit(string))` always succeeds and, as a side-effect, emits the source location of the current AST node in grep-output form.

```
follow_stmt.astlog
// FOLLOW_STMT(P1 P2)
//      <=> P1 and P2 are true of consecutive statements in this AST

follow_stmt(p1, p2)
<- if(op(#FUNCTION),
      kid(#FUNCTION/BODY, follow_stmt(p1, p2, *)),
      follow_stmt(p1, p2, *));

follow_stmt(p1, p2, after)
<- cond(op(#BLOCK), follow_block_stmt(p1, p2, after),
       op(#IF), kid(not(#IF/PRED), follow_stmt(p1, p2, after)),
       op(#SWITCH), kid(#SWITCH/BODY, follow_stmt(p1, p2, after)))

       op(#WHILE), follow_iter_stmt(#WHILE/BODY, p1, p2, after),
       op(#DO), follow_iter_stmt(#DO/BODY, p1, p2, after),
       op(#FOR), follow_iter_stmt(#FOR/BODY, p1, p2, after),

       or(op(#LABEL), op(#CASE), op(#DEFAULT)),
          kid(#LABELSTMT/STMT, follow_stmt(p1, p2, after)),

       follow_simple_stmt(p1, p2, after));

follow_simple_stmt(p1, p2, after)
<- with(after, not(*)), p1, with(after, first_stmt(p2));

follow_iter_stmt(nbody, p1, p2, after)
<- or(follow_simple_stmt(p1, p2, after),
      and(this, kid(nbody, follow_stmt(p1, p2, this))));
```

```

follow_block_stmt(p1, p2, after)
<- and(kid(minus(next,1), first),
       if(kid(next, second),
          with(first, follow_stmt(p1, p2, second)),
          with(first, follow_stmt(p1, p2, after))));

first_stmt(p)
<- if(op(#BLOCK),
      kid(0, first_stmt(p)),
      stmt);

// CASEFALL()
// emits all locations of switch-case fallthroughs in this AST tree
casefall()
<- follow_stmt(and(not(op(or(#BREAK,#CONTINUE,#GOTO,#RETURN))),
                   op(#CASE)),
               with(first, sfa(emit("Fall through to next case."))));


```

**Figure 5: Definition of flatten**

```

flatten(test, lst)
<- flatten(test, lst, []);

flatten(test, head, tail)
<- if(test,
      first(head, hrest),
      unify(head, hrest)),
      flattenkids(test, 0, hrest, tail);

flattenkids(test, n, head, tail)
<- if(kid(n, flatten(test, head, mid)),
      and(with(n, minus(nplus1,1)),
          flattenkids(test, nplus1,
                      mid, tail)),
      unify(head, tail));

first([o|rest],rest) <- o;
unify(x,x);

```

**Figure 6: Parameterized version, flatten2**

```

flatten2(test, lst)
<- flatten2(test, lst, []);

```

```

flatten2(test, head, tail)
<- if((test)(value),
      unify(head, [value|hrest]),
      unify(head, hrest)),
   flatten2kids(test, 0, hrest, tail);

flatten2kids(test, n, head, tail)
<- if(kid(n, flatten2(test, head, mid)),
      and(with(n, minus(nplus1,1)),
          flatten2kids(test, nplus1,
                      mid, tail)),
      unify(head, tail));

unify(x,x);

```

## 4.3 Higher order features

We have already included some of the non-1st-order features of *Prolog*, notably “cut” (in the form of `if()`) and the corresponding notion of negation, `not()`. There are others that turn out to be essential as well.

### 4.3.1 3.1 Lambdas and Applications

One may observe that, in `somenode(test)`, because this is an existential query, it does not matter that we are matching the same term `test` to every node of the tree. If variables in `test` get bound as a result of matching a given node, those bindings will be undone prior to advancing to the next node.

If one instead wants to write a conjunctive predicate over all tree nodes, say

```
flatten(test, list)
```

which holds if `list` is a list of **all** descendant nodes satisfying `test`, — we give a definition in Figure 5 — this will not work correctly if `test` contains any variables that are bound during the course of matching any node; said variables will **stay** bound for the duration of the `flatten` evaluation.

Even in an existential query, there is the possibility that the `test` being passed in will itself need to take a parameter. For example, one might imagine defining a version of `sametree` that also requires an additional user-specified `test` to hold at each corresponding pair of nodes. If `test` is a mere compound term, it can be matched against one of the nodes, but not both.

Thus we introduce **“application” terms** and operator-valued **“lambda” terms**. For an application `(fterm)(term;...)` to match the current object, the term `fterm` must be (or be a variable bound to) a predicate-operator-valued term, which will either be

- a reference, `'foo/3` to a named predicate operator, in which case the application evaluates exactly as the corresponding compound term would, or
- an anonymous predicate operator `FN{importvars ... } (anon-clauses ...)`, in which case the application evaluates **almost** exactly as if there were a named predicate-operator defined by the given clauses and this were a compound term on that operator. The difference is that any variables of those clauses that are also on the `{importvars... }` list are identified with the correspondingly-named variables in the clause where the `FN` term occurs lexically.

An `FN` term with imports can be thought of as a kind of ***closure***.

The parameterized version of flatten, namely

```
flatten2(test, list)
```

which holds iff list is a list of all `x` corresponding to descendants that `(test)(x)` matches, is defined in Figure ??.

## Figure 7: Parameterized version of sametree

```
sametree(node, equiv)
<- unify(same,
FN{same,equiv}
((node)
<- op(nodeop),
with(node,op(nodeop)),
(equiv)(node),
not(and(with(node,kid(n,nkid)),
kid(n,not((same)(nkid)))))),
(same)(node);
```

The parameterized version of sametree is invoked as

```
sametree(node, equiv)
```

which holds iff node is a reference to an AST node with the same tree structure as the current node and, for every descendant `n` of node, the corresponding node in the current tree satisfies `equiv(n)`; this predicate is defined in Figure 4.3.1. This definition demonstrates the use of import lists, both to define a recursive anonymous predicate, and to make `equiv` available at once to all evaluations of that predicate. Given that definition, the following

```
sametree(node,
FN((n) <- if(aconst(c),
with(n, aconst(c)),
and());))
```

would then test whether the current tree has the same structure as underneath node and such that all corresponding constants are the same.

## Figure 8: Embedded Query State Primitives

**query(fterm; query-pred)**

The embedded query state object created from fterm satisfies query-pred.

**qnext(pred; thisquery-pred; nextquery-pred)**

If the current embedded query state is a failure, pred is true, otherwise the current object satisfies this query-pred and, after the embedded query is advanced to the next hit or to failure, the resulting query state satisfies nextquery-pred.

**qget(object-pred;::: )**

Each object-pred matches the object bound to the corresponding variable in the head of the embedded query corresponding to the current query state object. An error will be raised if the embedded query has failed or if any head variable is not bound to an object.

### 4.3.2 Queries as Objects

Sometimes one wishes to build a collection or some other kind of aggregate of all objects found by a query. Unfortunately, when backtracking to get to the next hit, information about the previous hit will generally be lost. One solution is to rewrite the query into a conjunctive form, as we did in the previous section converting writing flatten as a conjunctive version of somenode (see Figure 4.2.3). We can already see that even in simple cases this process can be non-trivial and is not readily generalized.

It may also be the case for some conjunctive queries that they require memory proportional to the size of the data structure being searched, instead of merely memory proportional to the depth of the data structure. Judicious use of if() | astlog's moral equivalent of the cut operator | can avoid this, but this is sometimes cumbersome to get right.

As it happens, Prolog provides a number of setpredicates for accumulating query results. For example,

**bagof(x, term, list)**

binds list to a list of the bindings of x corresponding to each instance where term holds true. Unfortunately, this is usually implemented in terms of assert and retract, meaning we would have to abandon the idea of keeping our script small and fixed. Even just adding this as a new primitive is dubious if we have to add, say, another new primitive to merely count query hits, and yet more new primitives for each accumulation method anyone dreams up.

The key observation is that the execution model of astlog allows for the possibility of treating some subset of its own internal structures as "external" objects which can then serve as the current object of various kinds of queries. To be sure, some care needs to be exercised, since the internal structures of astlog are not static the way the program asts are. We can however, take a query whose hits we wish to accumulate, and

encapsulate its state after a given hit as an astlog object. Such an embedded query in a given state can now be the current object for the evaluation of some other predicate term. We thus only need to provide suitable primitive predicates applicable to query-state objects that may be used in such a term. Figure 4.3.1 lists these primitives.

## Figure 9: Query Accumulators qcount and qlist

```

qcount(n) <- qcount(0, n);
qcount(sofar, return)
<- qnext(unify(sofar, return),
with(sofar, minus(sofarp1,1)),
qcount(sofarp1, return));
qlist(lst)
<- qnext(unify(lst, []), 
qget(first(lst,rest)),
qlist(rest));
// utilities
first([o|rest],rest) <- o;
unify(x,x);

```

Using this mechanism, it is then possible to define a wide variety of accumulators of query results. Given an ast node, and a query to see if there exists a descendant satisfying `test(x)`

```
() <- somenode(test(x));
```

the corresponding query to count the number of descendants satisfying `test(x)` would be

```
(n) <- query(FN(() <- somenode(test(x)); ),
qcount(n));
```

where `qcount/1` is defined as in Figure 4.3.2. Evaluating the `query()` term starts an embedded query corresponding to the first operand and builds a query state object representing the resulting first state (first hit or failure). This object then becomes the current object to which we try to match `qcount(n)`. It is the `qnext()` term therein that does the actual work. If the query-state is a success state, we increment the count of hits thus far (`sofar`), advance the embedded query, and recursively try to match a `qcount` term to the new state. If the query-state is a failure, we unify the count of hits thus far with the `return` variable.

To build a list of bindings for `x` corresponding to the query hits, we can do

```
(list) <- query(FN((x) <- somenode(test(x)); ),
qlist(list));
```

which is essentially the same as before except that now `qlist(list)` uses `qget` to examine the query state. Since the embedded query has only one head variable `x`, the `qget` term must likewise have at most one operand.

Some care is required when using embedded queries to phrase them so that the head variables will always be bound to objects. `qget()` will in fact raise an error if a head variable is not bound to an object. This requirement is crucial since, with a non-object term, there is no guarantee that said term will remain intact when the embedded query backtracks to the next state. Better to keep terms constructed by an embedded query from polluting the outer world.

The mechanism is also somewhat impure in that evaluating a `qnext` on a given query state object essentially destroys that object. Subsequent attempts to match additional terms against that query state will raise an error since the state of a query is lost once we advance it.

## 4.4 Implementation

`astlog` has been implemented as an interpreter in roughly 11,000 lines of  $C_+^+$  for the core `astlog` interpreter and supporting utilities. Another 1100 lines define the roughly 60 primitives and supporting structures to invoke the various functions of the AST library. Coverage of the library API is in not entirely complete, but it is sufficient to perform various interesting tasks:

- Finding all instances of a simple assignment expression (`=`) occurring in any boolean context, for example,

```
if ((major == SORTM)
|| (major == MEMORYM)
|| ((major == BUFFERM)
&& (minor = B_NOIO)))
```

- Finding all instances of an equality-test (`==`) or dereference expression occurring in any void context (i. e., where results are discarded); the converse to the previous problem.
- Finding all case statement fall-throughs, i. e., where the preceding statement is not a `break`.
- Finding various patterns of irreducible control-flow in functions.
- Obtaining all static call-graph edges.
- Computing the McCabe cyclomatic complexity [?] of a function. Our code to do so looks like

```

mccabe(n) <- query(
FN(()<- somenode(
op(or(#IF,#FOR,#DO,
#WHILE,#CASE,
#?,#||,#&&)));
qcount(minus(n,1))
);

```

which might be compared with the 17-line version in Aria [?]. Admittedly, fairness would probably entail including the definitions of somenode and qcount as well.

- Finding gaps (unused space due to alignment rules) in structure definitions; this is a matter of traversing C<sub>II</sub> type structures rather than asts.

A typical running time (on a 200MHz Pentium P6 with 64meg of RAM) for a one-pass search that evaluates a simple predicate on every ast node in Microsoft **SQLserver** (roughly 450,000 lines, 4300 functions) is roughly 10 minutes, of which 7.5 minutes are taken up by the ast library building the actual trees. For Microsoft Word (roughly 2,000,000 lines) the corresponding times are 45-60 minutes of which about 30 minutes is taken up by the tree builder.

Though this dreadfully slow in comparison with grep, these times are arguably acceptable in comparison with the times taken by the actual compiler | what one might expect for a tool that requires the use of compiler's data structures. One is, of course, free to write arbitrarily non-linear programs in astlog, so there are no guarantees. In any case we would doubtless see a certain amount of speedup if we actually were to attempt some kind of compilation of the astlog code.

## 4.5 Conclusions and Future Work

We have described a language for doing syntax-level analysis for C/C++ programs, though the core language is, in fact, adaptable to many other kinds of structures. As with previous such tools, the utility to users who are thus no longer required to write their own parse/semantic-analysis phase is apparent. The contribution here is a pattern language sufficiently powerful to provide traversal possibilites beyond what is naturally available in prior awk-like frameworks while avoiding some of the inefficiencies of importing the entire program structure into a logical inference engine. The Pan work [?] stressed the need to partition code and data; this we have done in a rather straightforward way. The surprise is that the *Prolog* with-an-ambient-current-object model turns out to be so well suited to analyzing treelike structures.

To be sure, there are various rough edges:

1. As already noted, embedded queries are slightly unsafe; there may exist a more robust set of primitives to use. Some form of type inference to detect unsafe uses of qnext may also be worth considering. More generally, there is the issue

of typing of astlog expressions to reduce the incidence of unbound variables or objects of the wrong type appearing as operands where object-references of a particular type are required.

2. Occasionally, we run up against the generally cumbersome nature of arithmetic in Prolog, which is arguably worse in astlog. The “inside-out functional” nature of astlog may be good for ast patterns, but it can make arithmetic operations like

```
with(n; divide(minus(x; 1); 2))
```

downright unreadable. Algebraic syntax could help, e. g.,

```
with(n; (x - 1)=2)
```

but even so, one must stare at this pretty hard to realize that n is being multiplied by 2 and then incremented by 1.

One possibility is to complicate the language by introducing actual “forward” functional operator definitions. For example, with such forward operators for addition and multiplication, one could then write

```
with(2 n + 1; x)
```

where the appearance of the + (plus) term in a slot normally requiring an object reference invokes the forward return-value-to-context definition of the operator + to sum its operands rather than the usual “backward” return-value-to-operand definition (see Figure 2) in which one operand is treated as a predicate.

3. Though there is a surprising amount of mileage to be had via instantiating terms with unbound variables in them, there are those occasions when a genuinely mutable data structure is required. Fortunately, given the strong partition between the script/database and the objects, having mutable objects exist and primitives that side-effect them when they match would not disrupt astlog’s execution model.
4. Currently, new primitives need to be manually written. Given the current collection of macros available, this is not actually an arduous task. Still, while language-independence was not one of our priorities, given that the core language is rather language-independent anyway, one would hope for a more automatic means of adapting astlog to work with other language parsers, perhaps by adapting GENII [?] or some similar tool to generate code for the basic primitive predicate operators for a fresh language.

## 4.6 Acknowledgements

ASTLOG would not have been possible without the existence of an ast library for C/C++ implemented by the members of Program Analysis group at Microsoft Research particularly Linda O’Gara, David Gay, Erik Ruf and Bjarne Steensgaard. I would also like to thank Bruce Duba, Michael Ernst, Chris Ramming, and the conference reviewers for much useful commentary and discussion.

## References

- AKW86** A. V. Aho, B. W. Kernighan, and P. J. Weinberger. The AWK Programming Language. Addison Wesley, Reading, MA, 1986.
- BCD88** P. Borras, D. Clement, Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. Centaur: The system. In Proceedings of the SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Boston, MA, 1988.
- BGV90** Robert A. Ballance, Susan L. Graham, and Michael L. Van De Vanter. The pan language-based editing system for integrated development environments. In Proceedings of the 4th ACM SIGSOFT Symposium on Software Development Environments, pages 77..93, Irvine, CA, 1990.
- CMR92** Mariano Consens, Alberto Mendelzon, and Arthur Ryman. Visualizing and querying software structures. In Proceedings of the Fourteenth International ACM Conference on Software Engineering, pages 138..156, 1992.
- Dev92** Premkumar T. Devanbu. Genoa - a customizable, language-and-front-end independent code analyzer. In Proceedings of the Fourteenth International ACM Conference on Software Engineering, pages 307..319. ACM Press, 1992.
- DR96** Premkumar T. Devanbu and David S. Rosenblum. Generating testing and analysis tools with aria. ACM Transactions on Software Engineering and Methodology, 5(1):42..62, January 1996.
- GA96** William G. Griswold and Darren C. Atkinson. Fast, exible syntactic pattern matching and processing. In Proceedings of the IEEE Workshop on Program Comprehension. ACM Press, 1996.
- LR95** David A. Ladd and J. Christopher Ramming. A\*: A language for implementing language processors. IEEE Transactions on Software Engineering, 21(11):894..901, November 1995.
- McC76** T. McCabe. A complexity measure. IEEE Transactions on Software Engineering, 2(4):308..320, December 1976.

## Appendix

For those who would prefer to see a slightly more formal description, we include a brief outline of an operational semantics for astlog in Figure 10, one that bears some resemblance to the actual implementation.

For any given term that is not an object reference, one may imagine there being numerous instances of that term in existence at any given time. We differentiate the various instances by assigning each a unique frame identifier ( $f$ ) which is only significant for its identity. A variable  $v$  occurring within a given term  $t$  may, for a particular instance  $hf ; [[t]]_i$  of that term, be bound to some object  $o$  or other term instance  $hf_0 ; [[t_0]]_i$ , this being indicated by having a binding, i.e., one of  $hf ; [[v]]_i \text{ if } o$  or  $hf ; [[v]]_i \text{ if } hf_0 ; [[t_0]]_i$  present in the current binding stack, which in turn is nothing more than a list of bindings. The semantic function  $vlookup(B ; hf ; [[t]]_i)$  returns

- $hf ; [[t]]_i$  itself if  $t$  is not a variable.
- ? if the variable  $t$  is not bound in  $B$ .
- $o$  if  $hf ; [[t]]_i \text{ if } o$  is in  $B$
- $vlookup(B ; hf_0 ; [[t_0]]_i) \text{ if } hf ; [[t]]_i \text{ if } hf_0 ; [[t_0]]_i$  is in  $B$ .

At any given time, the full state of our abstract machine is described by a failure of the form  $B ' C :: F$  which consists of

- the current binding stack  $B$ ,
- the current continuation  $C = (o; f; g; C_0)$ , which in turn consists of a current object  $o$ , a current frame identifier  $f$ , a current goal, usually a term, but this can also be one of the auxiliary goals “apply(…)” or “cut(…),” and finally another continuation  $C_0$  to which we advance if the goal succeeds
- the next failure  $F$ , to which we advance if the current goal fails.

Note that unlike the case where the goal succeeds, failure may involve undoing one or more bindings; thus, a failure ( $F$ ) contains its own binding stack (a subset of  $B$ ) whereas the continuations ( $C, C_0$ ) do not.

The bottom half of Figure 10 (partially) defines a transition relation between states of the abstract machine. Given an initial current object  $o$  and a query  $[[query]]$  with  $n$  head variables, we take the initial state to be

`F0 = [] ' (o; f0; apply(f0; [[query]]; [[v1;:::; vn]]); yes) :: no` If there is a sequence of transitions `F0 ! B1 ' yes :: F\verb` then we have a hit and the various query head bindings are available as `vlookup(B1; hf0; [[vi ]]|i)` for  $i = 1 \dots n$ . Likewise, if `Fk ! Bk ' yes :: Fk+1` then we have a  $(k + 1)$ th hit.

When we have a  $(k + 1)$ th hit. The semantic function `mgu(B; f; [[t1;:::;tn]]; f 0 ; [[t 01;:::;t0n]])` returns an augmented binding stack that includes B together with those additional bindings that make up the most general unifier of the respective term instances `hf ; [[t1]]|i` with `hf 0 ; [[t 01]]|i`, etc... . If there is no most general unifier, `mgu()` returns `ufail`.

In the actual implementation, because the script is unified, we may precompute at load time mgus of all pairs of same-operator-and-arity compound terms occurring in the script, making clause invocation no more expensive than a function call in many cases. We also omit the “occurs check” [SS86] for the run-time portion of unification (i.e., where we’re transitively following variable bindings), with the usual increase in speed and infinite-loop risk. Thus far, unification has played a somewhat smaller role in astlog scripts than expected, so there’s some question whether we need to be doing even this much.

As noted above objects only unify with equal objects. The idea of allowing an object to unify with a compound predicate term that matches it has been considered, but rejected due to the significant complications it would introduce. Also, once one has subgoals being attempted during the course of unification, the user’s control over evaluation order is drastically reduced, something to be avoided if one is interested in having users being able to write efficient scripts.

**Figure 10: Outline of astlog Operational Semantics**

# Глава 5

# Warren's Abstract Machine Абстрактная машина Варрена

<sup>1</sup>

© Hassan Aït-Kaci <[hak@cs.sfu.ca](mailto:hak@cs.sfu.ca)>  
© David H. D. Warren

## Предисловие к репринтному изданию

Этот документ — репринтное издание книги имеющей то же название, которая была опубликована MIT Press, в 1991 году с кодом ISBN 0-262-51058-8 (мягкая обложка) and ISBN 0-262-01123-9 (тканый переплет). Редакция книги MIT Press сейчас не передается, и права на издание были переданы автору. Оригинальная версия<sup>2</sup> была бесплатно доступна всем, кто хочет ее использовать в некоммерческих целях, с веб-сайта автора:

<http://www.isg.sfu.ca/~hak/documents/wam.html>

*Сейчас ссылка недоступна, книга переехала на <http://wambook.sourceforge.net/>*

Если вы используете ее, пожалуйста дайте мне знать кто вы и для каких целей хотите ее использовать.

Thank you very much.

Hassan Aït-Kaci  
Burnaby, BC, Canada  
May 1997

---

<sup>1</sup> © <http://wambook.sourceforge.net/>

<sup>2</sup> английская <http://wambook.sourceforge.net/>

# Предисловие

Язык *Prolog* был задуман в начале 1970х Alain Colmerauer и его коллегами из Марсельского университета. Его реализация языка была первым практическим воплощением концепции *логического программирования*, предложенной Robert Kowalski. Ключевая идея логического программирования — вычисления могут быть выражены в виде контролируемого вывода (дедукции) из набора декларативных утверждений. Несмотря на то что эта область значительно развилась за последнее время, *Prolog* остается наиболее фундаментальным и широко известным языком логического программирования.

Первой реализацией *Prologa* был интерпретатор, написанный на Фортране членами группы Colmerauer. Несмотря на очень ущербную в некотором смысле реализацию, эта версия считается в некотором смысле первым камнем: она доказала жизнеспособность *Prologa*, помогла распространению языка, и заложила основные принципы реализаций *Prologa*. Следующим шагом возможно была *Prolog*-система для PDP-10, разработанная в Университете Эдинбурга мной и коллегами. Эта система построена на базе техник Марсельской реализации, с добавлением понятия компиляции *Prologa* в низкоуровневый язык (в случае PDP-10 это машинный код), а также различные техники экономии памяти. Позже я уточнил и абстрагировал принципы реализации *Prolog DEC-10* в то, что я называю **WAM** (Warren Abstract Machine).

**WAM** — абстрактная (виртуальная) машина с архитектурой памяти и набором команд, заточенных под язык *Prolog*. Она может быть эффективно реализована на широком наборе аппаратных архитектур, и служить целевой платформой для переносимых компиляторов *Prologa*. Сейчас она принимается как стандартный базис при реализации *Prologa*. Это конечно лично приятно, но неудобно в том, что WAM слишком легко принимается как стандарт. Несмотря на то что WAM явилась результатом длительной работы и большого опыта в реализации *Prologa*, это отнюдь не единственно возможный подход. Например, в то время как WAM применяет *копирование структуры*<sup>3</sup> для представления *термов Prologa*, метод *общих структур*<sup>4</sup>, использованный в Марсельской и DEC-10 реализациях, все еще можно рекомендовать к применению. Как бы то ни было, я считаю WAM хорошей отправной точкой для изучения технологий реализации *Prolog*-машины.

К сожалению до сих пор не было хорошей книги для ознакомления с внутренним устройством WAM. Мой оригинальный технический отчет слишком сложен, содержит только скелетное описание *Prolog*-машины, и написан для опытного читателя. Другие работы обсуждают WAM с различных точек зрения, но все же не могут быть использованы в качестве хорошего вводного руководства.

Поэтому очень приятно видеть появление этого прекрасного учебника, написанного Hassan Aït-Kaci. Эту книгу приятно читать. Она объясняет WAM с большой ясностью и элегантностью. Я думаю что читатели, интересующиеся информа-

<sup>3</sup> structure copying

<sup>4</sup> structure sharing

тикой, найдут эту книгу очень стимулирующим введением в увлекательную тему — реализацию *Prologa*. Я очень благодарен Хассану за донесение моей работы до широкой аудитории.

© David H. D. Warren  
Бристоль, UK  
Февраль 1991

## 5.1 1 Введение 3

В 1983 году Дэвид Варрэн разработал абстрактную машину для реализации языка *Prolog*, содержащую специальную архитектуру памяти и набор инструкций [?]. Эта разработка стала известна как Warren Abstract Machine (WAM) и стала стандартом де-факто для реализаций компиляторов *Prologa*. В [?] Варрэн описан WAM в минималистичном стиле, который слишком сложен для понимания неподготовленным читателем, даже заранее знакомым в операциями *Prologa*. Слишком многое было несказаным, и very little is justified in clear terms<sup>5</sup>. Это привело к очень скучному количеству поклонников WAM, которые могли был похвастаться пониманием деталей ее работы. Обычно это были реализаторы *Prologa*, которые решили уделить необходимое время для обучения через делание и кропотливого достижения просветления.

### 5.1.1 1.1 Существующая литература 3

Свидетельством недостатка понимания может служить тот факт, что за первые шесть лет было крайне мало публикаций о WAM, не говоря о том чтобы формально доказать ее корректность. Кроме оригинального герметического доклада Варрэна [?], практически не было никаких официальных публикаций о WAM. Несколько лет спустя группой Аргонской Национальной Лаборатории был выпущен единственный черновой стандарт [?]. Но следует отметить что этот манускрипт был еще менее понятен, чем оригиналный отчет Варрэна. Его недостатком была цель описать готовую WAM как есть, а не как пошагово трансформируемый и оптимизируемый проект.

Стиль пошагового улучшения фактически был использован в публикации David Maier и David S. Warren<sup>6</sup> [?]. В этой работе можно найти описание техник компиляции *Prologa* похожие на принципы WAM<sup>7</sup>. Тем не менее мы считаем что

<sup>5</sup> David H. D. Warren поделился в частной беседе что он “чувствовал что WAM важна, но к деталям ее реализации вряд ли будет широкий интерес, поэтому он использовал стиль личных заметок”

<sup>6</sup> Это другой человек, а не разработчик WAM, работа которого вдохновила S.Warren на исследования. В свою очередь достаточно интересно что David H. D. Warren позже работал над параллельной архитектурой реализации *Prologa*, поддерживая некоторые идеи, независимо предложенные David S. Warren.

<sup>7</sup> chap.9

эта похвальная попытка все еще страдает от нескольких недостатков, если его рассматривать как окончательный учебник. Прежде всего эта работа описывает собственный достаточно близкий вариант WAM, но строго говоря не ее саму. Так что описаны не все особенности WAM. Более того, объяснения ограничены иллюстративными примерами, и редко четко и исчерпывающие очерчивают контекст, в котором применяются некоторые оптимизации. Во-вторых, часть посвященная компиляции *Prologa*, идет очень поздно — в предпоследней главе, полагаясь в деталях реализации на свердetaлизированные процедуры на Паскаль, и структуры данных, последовательно улучшаемые в течение предыдущих разделов. Мы чувствуем что это уводит и запутывает читателя, интересующегося абстрактной машиной. Наконец, несмотря на то что публикация содержит серию последовательно улучшаемых вариантов реализации, этот учебник не отделяет независимые части *Prologa* в процессе. Все представленные версии — полные *Prolog*-машины. В результате, читатель интересующийся выбором и сравнением отдельных техник, которые он хочет применить, не может различить отдельные техники в тексте. По всей справедливости, книга Майера и С.Варрена имеет амбиции быть первой книгой по логическому программирования. Так что они совершили подвиг, охватывая так много материала, как теоретического так и практического, и даже включили техники компиляции *Prologa*. Более важно, что их книга была первой доступной официальной публикацией, содержащей реальный учебник по техникам WAM.

After the preliminary version of this book had been completed, another recent publication containing a tutorial on the WAM was brought to this author's attention. It is a book due to Patrice Boizumault [?] whose Chapter 9 is devoted to explaining the WAM. There again, its author does not use a gradual presentation of partial *Prolog* machines. Besides, it is written in French — a somewhat restrictive trait as far as its readership is concerned. Still, Boizumault's book is very well conceived, and contains a detailed discussion describing an explicit implementation technique for the `freeze` meta-predicate<sup>8</sup>.

Even more recently, a formal verification of the correctness of a slight simplification of the WAM was carried out by David Russinoff [?]. That work deserves justified praise as it methodically certifies correctness of most of the WAM with respect to *Prolog*'s SLD resolution semantics. However, it is definitely not a tutorial, although Russinoff defines most of the notions he uses in order to keep his work self-contained. In spite of this effort, understanding the details is considerably impeded without working familiarity with the WAM as a prerequisite. At any rate, Russinoff's contribution is nevertheless a **première** as he is the first to establish rigorously something that had been taken for granted thus far. Needless to say, that report is not for the fainthearted.

---

<sup>8</sup> chap.10

## 5.1.2 1.2 Этот учебник 5

### 1.2.1 Disclaimer and motivation 5

The length of this monography has been kept deliberately short. Indeed, this author feels that the typical expected reader of a tutorial on the WAM would wish to get to the heart of the matter quickly and obtain complete but short answers to questions. Also, for reasons pertaining to the specificity of the topic covered, it was purposefully decided not to structure it as a real textbook, with abundant exercises and lengthy comments. Our point is to make the WAM explicit as it was conceived by David H. D. Warren and to justify its workings to the reader with convincing, albeit informal, explanations. The few proposed exercises are meant more as an aid for understanding than as food for further thoughts.

The reader may find, at points, that some design decisions, clearly correct as they may be, appear arbitrarily chosen among potentially many other alternatives, some of which he or she might favor over what is described. Also, one may feel that this or that detail could be “simplified” in some local or global way. Regarding this, we wish to underscore two points: (1) we chose to follow Warren’s original design and terminology, describing what he did as faithfully as possible; and, (2) we warn against the casual thinking up of alterations that, although that may appear to be “smarter” from a local standpoint, will generally bear subtle global consequences interfering with other decisions or optimizations made elsewhere in the design. This being said, we did depart in some marginal way from a few original WAM details. However, where our deviations from the original conception are proposed, an explicit mention will be made and a justification given.

Our motivation to be so conservative is simple: our goal is not to teach the world how to implement Prolog optimally, nor is it to provide a guide to the state of the art on the subject. Indeed, having contributed little to the craft of Prolog implementation, this author claims glaring incompetence for carrying out such a task. Rather, this work’s intention is to explain in simpler terms, and justify with informal discussions, David H. D. Warren’s abstract machine **specifically** and **exclusively**. Our source is what he describes in [?, ?]. The expected achievement is merely the long overdue filling of a gap so far existing for whoever may be curious to acquire **basic** knowledge of Prolog implementation techniques, as well as to serve as a spring board for the expert eager to contribute further to this field for which the WAM is, in fact, just the tip of an iceberg. As such, it is hoped that this monograph would constitute an interesting and self-contained complement to basic textbooks for general courses on logic programming, as well as to those on compiler design for more conventional programming languages. As a stand-alone work, it could be a quick reference for the computer professional in need of direct access to WAM concepts.

### 1.2.2 Organization of presentation 6

Our style of teaching the WAM makes a special effort to consider carefully each feature of the WAM design in isolation by introducing separately and incrementally distinct

aspects of Prolog. This allows us to explain as limpidly as possible specific principles proper to each. We then stitch and merge the different patches into larger pieces, introducing independent optimizations one at a time, converging eventually to the complete WAM design as described in [?] or as overviewed in [?]. Thus, in 5.2, we consider unification alone. Then, we look at flat resolution (that is, Prolog without backtracking) in 5.3. Following that, we turn to disjunctive definitions and backtracking in 5.4. At that point, we will have a complete, albeit naïve, design for pure Prolog. In 5.5, this first-cut design will be subjected to a series of transformations aiming at optimizing its performance, the end product of which is the full WAM. We have also prepared an index for quick reference to most critical concepts used in the WAM, something without which no (real) tutorial could possibly be complete.

It is expected that the reader already has a basic understanding of the operational semantics of *Prolog* — in particular, of unification and backtracking. Nevertheless, to make this work also profitable to readers lacking this background, we have provided a quick summary of the necessary *Prolog* notions in 5.7. As for notation, we implicitly use the syntax of so-called Edinburgh Prolog (see, for instance, [?]), which we also recall in that appendix. Finally, 5.8 contains a recapitulation of all explicit definitions implementing the full WAM instruction set and its architecture so as to serve as a complete and concise summary.

## 5.2 2 Унификация — ясно и просто 9

Recall that a (first-order) term is either a *variable* (denoted by a capitalized identifier), a *constant* (denoted by an identifier starting with a lower-case letter) or a *structure* of the form  $f(t_1, \dots, t_n)$  where  $f$  is a symbol called a *functor* (denoted as a constant), and the  $t_i$ 's are first-order terms — the term's *subterms*. The number of subterms for a given functor symbol is predetermined and called its *arity*. In order to allow a symbol to be used with possibly different arities, we shall use the explicit notation  $f/n$  when referring to the functor consisting of the symbol  $f$  and arity  $n$ . Hence, two functors are equal if and only if they have the same symbol *and* arity. Letting  $n = 0$ , a constant is seen as a special case of a structure. Thus, a constant  $c$  will be designated as the functor  $c/0$ .

We consider here  $\mathcal{L}_0$ , a very simple language indeed. In this language, one can specify only two sorts of entities: a *program term* and a *query term*. Both program and query are first-order terms but not variables. The semantics of  $\mathcal{L}_0$  is simply tantamount to computing the most general unifier of the program and the query. As for syntax,  $\mathcal{L}_0$  will denote a program as  $t$  and a query as  $?-t$  where  $t$  is a term. The scope of variables is limited to a program (resp., a query) term. Thus, the meaning of a program (resp., a query) is independent of its variables' names. An interpreter for  $\mathcal{L}_0$  will dispose of some data representation for terms and use a unification algorithm for its operational semantics. We next describe  $\mathcal{M}_0 = (\mathcal{D}_0, \mathcal{I}_0)$ , an abstract machine design for  $\mathcal{L}_0$  consisting of a data representation  $\mathcal{D}_0$  acted upon by a set  $\mathcal{I}_0$  of machine instructions.

The idea is quite simple: having defined a program term  $p$ , one can submit any query  $?-q$  and execution either fails if  $p$  and  $q$  do not unify, or succeeds with a binding of the variables in  $q$  obtained by unifying it with  $p$ .



5.2.1	2.1 Term representation . . . . .	10
5.2.2	2.2 Compiling L queries . . . . .	11
5.2.3	2.3 Compiling L programs . . . . .	13
5.2.4	2.4 Argument registers . . . . .	19
5.3	3 Flat Resolution 25	
5.3.1	3.1 Facts . . . . .	26
5.3.2	3.2 Rules and queries . . . . .	27
5.4	4 Prolog 33	
5.4.1	4.1 Environment protection . . . . .	34
5.4.2	4.2 What's in a choice point . . . . .	36
5.5	5 Optimizing the Design 45	
5.5.1	5.1 Heap representation . . . . .	46
5.5.2	5.2 Constants, lists, and anonymous variables . . . . .	47
5.5.3	5.3 A note on set instructions . . . . .	52
5.5.4	5.4 Register allocation . . . . .	54
5.5.5	5.5 Last call optimization . . . . .	56
5.5.6	5.6 Chain rules . . . . .	57
5.5.7	5.7 Environment trimming . . . . .	58
5.5.8	5.8 Stack variables . . . . .	60
5.8.1	Variable binding and memory layout . . . . .	62
5.8.2	Unsafe variables . . . . .	64
5.8.3	Nested stack references . . . . .	67
5.5.9	5.9 Variable classification revisited . . . . .	69
5.5.10	5.10 Indexing . . . . .	75
5.5.11	5.11 Cut . . . . .	83
5.6	6 Conclusion 89	

## Часть III

### Язык $bI$

# Глава 6

## 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 привил 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 **7.2** (символы, строки и числа) и тип **cons-ячейка** позволяющий конструировать составные структуры данных

*bI-стиль* универсальный символьный тип **7.1**, 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 **parse**.

От базового символьного типа наследуются

**скаляры** символ, строка, несколько вариантов чисел (целые, плавающие, машинные, комплексные)<sup>1</sup>

<sup>1</sup> критерием скалярности можно считать возможность распознавания элемента данных лексером

**композиты** структуры данных и объекты  
**функционалы** объекты, для которых определен *оператор аппликации*

*или*

**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*, реализованных в виде виртуальных методов классов

**подсистема ОП** реализация механизмов ОП, наследования от класса и объекта, 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

**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<sup>2</sup>, Beowulf<sup>3</sup> с постоянным составом, интернет-облака с переменным составом: узлы асинхронно подключаются/отключаются, гомо/гетерогенные: по аппаратной платформе узлов и ОС/среде на каждом узле. Распределение вычислений на одно- и многопроцессорных SMP-системах<sup>4</sup>

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

**подсистема крос-трансляции** между ходовыми языками программирования ( $C_+^+$ , *JavaScript*, *Python*, PHP, Паскаль) через связку: парсер входного языка → система типов DLR → кодогенератор выходного языка

<sup>2</sup> аппаратная разделяемая память через сеть InfiniBand — “Сергей Королев”

<sup>3</sup> компьютеры общего назначения (офисные) с передачей сообщений по Gigabit Ethernet

<sup>4</sup> многопоточные вычисления на одном многоядерном узле

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

**сервер приложений** обслуживающий тонких браузерных клиентов по HTTP/JS

# Глава 7

## Система динамических типов

### 7.1 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<sup>1</sup>. 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.**

```
// _____ = ABSTRACT SYMBOLIC
struct Sym {
    // _____ тип (класс) и значение элемента данных
    string tag;           // data type / class
    string val;           // symbol value
    // _____ конструкторы (токен используется в лексере)
    // _____ с о
    Sym(string ,string ); // <T:V>
    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<sup>2</sup>

```
AST может хранить (и обрабатывать) вложенные элементы
// _____ nest [] e
```

<sup>1</sup> в этом АСТ близок к традиционной аббревиатуре AST: Abstract Syntax Tree

<sup>2</sup> числа, списки, высокоровневые и скомпилированные функции, элементы GUI,..

```
vector<Sym*> nest;
void push(Sym*);
void pop();
```

параметры (и поля класса)

```
// _____ pa
map<string ,Sym*> pars;
void par(Sym*); // add parameter
```

вывод дампа объекта в текстовом формате

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

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

вычисление объекта

```
// _____ compute
virtual Sym* eval();
```

операторы

```
// _____
virtual Sym* str(); // str(A) string represent
virtual Sym* eq(Sym*); // A = B assignment
virtual Sym* inher(Sym*); // A : B inheritance
virtual Sym* member(Sym*); // A % B,C named member (cl
virtual Sym* at(Sym*); // A @ B apply
virtual Sym* add(Sym*); // A + B add
virtual Sym* div(Sym*); // A / B div
virtual Sym* ins(Sym*); // A += B insert
};
```

## 7.2 Скаляры

- 7.2.1 str: строка
- 7.2.2 int: целое число
- 7.2.3 hex: машинное hex
- 7.2.4 bin: бинарная строка
- 7.2.5 num: число с плавающей точкой

## 7.3 Композиты

- 7.3.1 list: плоский список
- 7.3.2 cons: cons-пара и списки в *Lisp*-стиле

## 7.4 Функционалы

- 7.4.1 op: оператор
- 7.4.2 fn: встроенная/скомпилированная функция
- 7.4.3 lambda: лямбда

# Глава 8

## Программирование в свободном синтаксисе: FSP

### 8.1 Типичная структура проекта FSP: *lexical skeleton*

Скелет файловой структуры FSP-проекта = lexical skeleton = skelex

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

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

Создали каталог проекта, сгенерили набор пустых файлов (см. далее), и запустили 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 &
...

```

<b>src.src</b>		исходный текст программы на вашем скриптовом языке
<b>log.log</b>		лог работы ядра <i>bI</i>
<b>ypp.ypp</b>	<b>flex</b>	парсер ??
<b>lpp.lpp</b>	<b>bison</b>	лексер ??
<b>hpp.hpp</b>	<i>C<sub>+</sub><sup>+</sup></i>	заголовочные файлы ??
<b>cpp.cpp</b>	<i>C<sub>+</sub><sup>+</sup></i>	код ядра ??
<b>Makefile</b>	<b>make</b>	зависимости между файлами и команды сборки (для <i>Linux</i> )
<b>bat.bat</b>	<i>Windows</i>	запускалка <b>(g)Vim</b> ??
<b>.gitignore</b>	<b>git</b>	список масок временных и производных файлов ??

## 8.1.1 Настройки (g)Vim

При использовании редактора/IDE (g)Vim удобно настроить сочетания клавиш и подсветку **синтаксиса вашего скриптового языка** так, как вам удобно. Для этого нужно создать несколько файлов конфигурации .vim: по 2 файла<sup>1</sup> для каждого диалекта скриптового языка<sup>2</sup>, и привязать их к расширениям через dot-файлы (g)Vim в вашем домашнем каталоге. Подробно конфигурирование (g)Vim см. 21.

filetype.vim	(g)Vim	привязка расширений файлов (.sr
syntax.vim	(g)Vim	синтаксическая подсветка для скр
/vimrc	Linux	настройки для пользователя
/vimrc	Windows	
/.vim/ftdetect/src.vim	Linux	привязка команд к расширению .s
/vimfiles/ftdetect/src.vim	Windows	
/.vim/syntax/src.vim	Linux	синтаксис к расширению .src
/vimfiles/syntax/src.vim	Windows	

## 8.1.2 Дополнительные файлы

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



logo.png: Логотип

<sup>1</sup> (1) привязка расширения файла и (2) подсветка синтаксиса

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

## 8.1.3 Makefile

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

универсальный Makefile для fp-sp-проекта

```
log.log: ./exe.exe src.src
        ./exe.exe < src.src > $@ && tail $(TAIL) $@
C = cpp.cpp ypp.tab.cpp lex.yy.c
H =.hpp.hpp ypp.tab.hpp
CXXFILES += -std=gnu++11
./exe.exe: $(C) $(H) Makefile
        $(CXX) $(CXXFILES) -o $@ $(C)
ypp.tab.cpp: ypp.ypp
        bison $<
lex.yy.c: lpp.lpp
        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/stdout` 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 [7.1](#). Библиотека динамических классов языка *bI III* 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 III*: 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 10.2**

**lex.yy.c** код лексического анализатора, генерируемый утилитой **flex 10.1**  
`CXXFLAGS += gnu++11` добавляем опцию диалекта  $C_+$ , необходимую для компиляции ядра *bI*

# Глава 9

## Синтаксический анализ текстовых данных

### 9.1 Универсальный Makefile

Универсальный Makefile сделан на базе [8.1.3](#), с добавлением переменной APP указывающей какой пример парсера следует скомпилировать и выполнить.

Для хранения (и возможной обработки) отпарсенных данных используем ядро языка *bi* [7](#) — 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

```
APP = minimal
$(APP).log: ./$(APP).exe $(APP).src
    ./$(APP).exe < $(APP).src > $@ && tail $(TAIL) $@
C = .. / bi / cpp . cpp ypp . tab . cpp lex . yy . c
H = .. / bi / hpp . hpp ypp . tab . hpp
CXXFILES += -I .. / bi -I . -std=gnu++11
./$(APP).exe: $(C) $(H) minimal.mk
    $(CXX) $(CXXFILES) -o $@ $(C)
ypp . tab . cpp: $(APP).ypp
    bison -o $@ $<
lex . yy . c: $(APP).lpp
    flex -o $@ $<

.PHONY: src
src: minimal . src comment . src string . src ops . src brackets . src

minimal . src: .. / bi / cpp . cpp
    head -n11 $< > $@
comment . src: .. / bi / cpp . cpp
    head -n11 $< > $@
string . src: .. / bi / cpp . cpp
```

```
head -n11 $< > $@  
ops.src: .. / bi/cpp.cpp  
head -n5 $< > $@  
brackets.src: .. / bi/cpp.cpp  
head -n5 $< > $@
```

## 9.2 $C_+$ интерфейс синтаксического анализатора

```
extern int yylex(); // получить код следующего токена, и увл  
extern int yylineno; // номер текущей строки файла исходника  
extern char* yytext; // текст распознанного токена, asciz  
#define TOC(C,X) { yyval.o = new C(yytext); return X; }  
  
extern int yyparse(); // отпарсить весь текущий входной поток  
extern void yyerror(string); // callback вызывается при синтаксической  
#include "ypp.tab.hpp"
```

## 9.3 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.

<sup>1</sup>

Лексер **minimal.lpp** /flex/

```
%{  
#include "hpp.hpp"  
%}  
%option noyywrap  
%option yylineno  
%%  
[a-zA-Z0-9_.]+ TOC(Sym,SYM)  
%%
```

(.. / bi / ) **hpp.hpp** содержит определения интерфейса лексера 9.2, и ядра языка *bI*

<sup>7</sup> для хранения результатов разбора текстовых данных

**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.

<sup>1</sup> эти три типа можно назвать атомами computer science

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

`TOC(Sym, SYM)` единственное правило, распознающее любые группы символов как класс **bi::sym**: латинские буквы, цифры и символы `_` и `.` (точка)<sup>2</sup>

## Парсер `minimal.ypp /bison/`

```
%{  
#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->tagval(); } ;  
scalar : SYM | STR | NUM ;  
ex : scalar ;  
%%
```

`hpp.hpp` заголовок аналогичен лексеру 9.3

`%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 7.1 и его производными классами.

`%token` описывает токены, которые может возвращать лексер `??`, причем набор токенов должен быть согласованным между лексером и парсером<sup>3</sup>

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

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

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

<sup>2</sup> точка добавлена, так часто используется в именах файлов

<sup>3</sup> определение токенов генерируется в файл `ypp.tab.hpp`

<sup>4</sup> чтение/вычисление/вывод/повторить

<sup>5</sup> разобранное выражение не вычисляется, хотя используемое ядро *bI* и поддерживает такой функционал

`scalar` выражение, представляющее только распознаваемые скаляры:

`SYM` символ,

`STR` строку [или](#)

`NUM` число<sup>6</sup>

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

**minimal.src:** Тестовый исходник

**minimal.log:** Результат прогона

```
#<sym: include> "<sym: hpp . hpp>"  
#<sym: define> <sym: YYERR> "\<sym: n>\<sym: n><<<sym: yylineno><<"<<<  
<sym: void> <sym: yyerror>(<sym: string> <sym: msg>) { <sym: cout><<<sym: Y  
<sym: int> <sym: main>() { <sym: return> <sym: yyparse>(); }  
  
<sym: Sym>::<sym: Sym>(<sym: string> <sym: T>, <sym: string> <sym: V>) { <sym:  
<sym: Sym>::<sym: Sym>(<sym: string> <sym: V>);<sym: Sym>(" <sym: sym> ", <sym:  
  
<sym: string> <sym: Sym>::<sym: tagval>() { <sym: return> "<" + <sym: tag> +  
<sym: string> <sym: Sym>::<sym: tagstr>() { <sym: return> "<" + <sym: tag> +  
<sym: string> <sym: Sym>::<sym: pad>(<sym: int> <sym: n>) { <sym: string> <sym:  
<sym: string> <sym: Sym>::<sym: dump>(<sym: int> <sym: depth>) { <sym: str  
    <sym: return> <sym: S>; }  
  
<sym: Sym>*<sym: Sym>::<sym: eval>() { <sym: return> <sym: this>; }
```

Как видно по логу **minimal.log**, все группы символов, соответствующих правилу лексера **SYM**<sup>9.3</sup>, распознались как объекты  $bI$ , остальные остались символами и попали в лог без изменений.

## 9.4 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

```
%{  
#include "hpp . hpp"  
%}
```

<sup>6</sup> числа в грамматике языка  $bI$  по типам не делятся, токен соответствует как `int`, так и `num`

```
%option noyywrap
%option yylineno
%%
#[^\n]*          {}
[a-zA-Z0-9_.]+    TOC(Sym,SYM)
%%
```

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)**9.2**, наоборот, при своем выполнении создает токен, и возвращает код токена =SYM.

**comment.log:** Результат прогона

```
<sym: void> <sym: yyerror>(<sym: string> <sym: msg>) { <sym: cout><<<sym:>
<sym: int> <sym: main>() { <sym: return> <sym: yyparse>(); }

<sym: Sym>::<sym: Sym>(<sym: string> <sym: T>, <sym: string> <sym: V>) { <sym:>
<sym: Sym>::<sym: Sym>(<sym: string> <sym: V>):<sym: Sym>("<sym: sym>", <sym:>

<sym: string> <sym: Sym>::<sym: tagval>() { <sym: return> "<" + <sym: tag> +
<sym: string> <sym: Sym>::<sym: tagstr>() { <sym: return> "<" + <sym: tag> +
<sym: string> <sym: Sym>::<sym: pad>(<sym: int> <sym: n>) { <sym: string> <
```

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

## 9.5 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

```
%{
#include "hpp.hpp"
string LexString; /* string parser buffer */
%}
%option noyywrap
%option yylineno
%lex lexstring
%%
#[^\n]*          {}
\"
                {BEGIN(lexstring); LexString=""};
```

```

<lexstring>\"          {BEGIN(INITIAL); yyval.o = new Str(LexString);
<lexstring>\n          {LexString+=yytext[0];}
<lexstring>.           {LexString+=yytext[0];}

[a-zA-Z0-9_.]+          TOC(Sym,SYM)
%%
```

`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<sup>7</sup>

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

### Лог разбора со строками

```

<sym: void> <sym: yyerror>(<sym: string> <sym: msg>) { <sym: cout><<sym:>
<sym: int> <sym: main>() { <sym: return> <sym: yyparse>(); }

<sym: Sym>::<sym: Sym>(<sym: string> <sym: T>, <sym: string> <sym: V>) { <sym:>
<sym: Sym>::<sym: Sym>(<sym: string> <sym: V>):<sym: Sym>(<str: 'sym'>,<sym:>

<sym: string> <sym: Sym>::<sym: tagval>() { <sym: return> <str:'>+<sym:>
<sym: string> <sym: Sym>::<sym: tagstr>() { <sym: return> <str:'>+<sym:>
<sym: string> <sym: Sym>::<sym: pad>(<sym: int> <sym: n>) { <sym: string> <sym:>
```

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

## 9.6 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)`:

<sup>7</sup> символ конца строки не распознается метасимволом `.` (точка) в регулярном выражении, и требует явного указания

<sup>8</sup> использованы 'одинарные кавычки' как в *Python/bI*

## Лексер с операторами

```
%{  
#include "hpp.hpp"  
string LexString; /* string parser buffer */  
%}  
%option noyywrap  
%option yylineno  
%x lexstring  
%%  
#[^\\n]* { /* # line comment */  
  
\" {BEGIN(lexstring); LexString=""};  
<lexstring>\" {BEGIN(INITIAL); yyval.o = new Str(LexString);  
<lexstring>\\n {LexString+=yytext[0];}  
<lexstring>. {LexString+=yytext[0];}  
  
[a-zA-Z0-9_.]+ TOC(Sym,SYM) /* symbol */  
  
\( TOC(Op,LB) /* brackets */  
\) TOC(Op,RB)  
\[ TOC(Op,LQ)  
\] TOC(Op,RQ)  
\{ TOC(Op,LC)  
\} TOC(Op,RC)  
  
\+ TOC(Op,ADD) /* typical arithmetic operators */  
\- TOC(Op,SUB)  
\* TOC(Op,MUL)  
\/ TOC(Op,DIV)  
\^ TOC(Op,POW)  
  
\= TOC(Op,EQ) /* bi language specific */  
\@ TOC(Op,AT) /* assign */  
\~ TOC(Op,TILD) /* apply */  
\: TOC(Op,COLON) /* quote */  
/* 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
;
%%
```

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

## Разбор с операторами

```
<sym: void>
<sym: yyerror>
<op:(>
<sym: string>
<sym: msg>
<op:)>
<op:{>
<sym: cout><<
<sym: YYERR>;
<sym: cerr><<
<sym: YYERR>;
<sym: exit>
<op:(>
<op:->
<sym:1>
<op:)>;
<op:{>

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

## 9.7 Обработка вложенных структур (скобок)

Обработка вложенных структур возможна только парсером, лексер оставляем без изменений. Хранение вложенных структур в виде дерева — главная фича типа *bI AST* 7.1. 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
%%
REPL : | REPL ex { cout << $2->dump(); } ;
scalar : SYM | STR | NUM ;
ex :
    ex ex                  { $$=$1; $$->push($2); }
    | scalar | operator
    | LB ex RB              { $$=new Sym("(")); $$->push($2); }
    | LB RB                 { $$=new Sym("()"); }
    | LQ ex RQ              { $$=new Sym("["); $$->push($2); }
    | LC ex RC              { $$=new Sym("]"); $$->push($2); }
;
operator :
    ADD | SUB | MUL | DIV | POW
    | EQ | AT | TILD | COLON
    | SCOLON | GR | LS
;
%%
```

### Разбор со скобками

```
<sym: void>
<sym: yyerror>
<sym: ()>
<sym: string>
```

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

# Глава 10

## 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 7.3, и не заставлять пользователя закапываться в клубок *Lispovских 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  $\mathfrak{C}$ , и функциональные наклонности самого *bI*, претендующего на звание универсального **объектного метаязыка** и **языка шаблонов**.

### 10.1 lpp.lpp: лексер /flex/

lpp.lpp

```
%{
#include "hpp.hpp"
string LexString;                                     // string pa
void incLude(Sym*inc) {                                // .include
    if (!(yyin = fopen((inc->val).c_str(),"r")) ) yyerror("");
    yypush_buffer_state(yy_create_buffer(yyin,YY_BUF_SIZE));
}
%}
%option noyywrap
%option yylineno
%x lexstring docstring
S [-\+]?
N [0-9]+
```

```

%%
#[^\\n]*
{ }                                /* == line comment == */

^\\.end                               /* == . directive */
^\\.inc [ \\t]+[^\\n]+                  /* .end */
^\\.\\.[a-z]+[^\\n]*                   /* .include */
TOC(Directive,DIR)                   /* .directive */

/* 'string' */
<lexstring>'                         /* BEGIN( lexstring ); LexString="" */
<lexstring>\\t                        /* BEGIN(INITIAL); yyval.o=new Str(LexString); ret */
{LexString+=\\t;}                      /* LexString+='t'; */
{LexString+=\\n;}                      /* LexString+='n'; */
{LexString+=yytext[0];}                /* LexString+=yytext[0]; */
{LexString+=yytext[0];}                /* LexString+=yytext[0]; */

/* "docstring" */
<docstring>\"                         /* BEGIN( docstring ); LexString="" */
<docstring>\\t                        /* BEGIN(INITIAL); yyval.o=new Str(LexString); ret */
{LexString+=\\t;}                      /* LexString+='t'; */
{LexString+=\\n;}                      /* LexString+='n'; */
{LexString+=yytext[0];}                /* LexString+=yytext[0]; */
{LexString+=yytext[0];}                /* LexString+=yytext[0]; */

/* == numbers == */
TOC(Num,NUM)                          /* floating point */
TOC(Num,NUM)                          /* exponential */
TOC(Int,NUM)                          /* integer */
TOC(Hex,NUM)                          /* machine hex */
TOC(Bin,NUM)                          /* bin string */

/* == symbol == */
TOC(Sym,SYM)                          /* symbol */

/* == brackets == */
TOC(Op,LP)                            /* [ */
TOC(Op,RP)                            /* ] */
TOC(Op,LQ)                            /* { */
TOC(Op,RQ)                            /* } */
TOC(Op,LC)                            /* < */
TOC(Op,RC)                            /* > */
TOC(Op,LV)                            /* <vector> */
TOC(Op,RV)                            /* > */

/* == operators == */
TOC(Op,INS)                           /* + */
TOC(Op,DEL)                           /* - */

/* == operators == */
TOC(Op,EQ)                            /* = */
TOC(Op,AT)                            /* @ */
TOC(Op,TILD)                           /* ~ */
TOC(Op,COLON)                          /* : */

```

```

\%          TOC(Op,PERC)
\.
\,          TOC(Op,DOT)
\|          TOC(Op,COMMA)

\+          TOC(Op,ADD)
\-
\*          TOC(Op,SUB)
\/
\^          TOC(Op,MUL)
\/
\^          TOC(Op,DIV)
\^          TOC(Op,POW)

[ \t\r\n]+    {}                      /* == drop spaces == */

<<EOF>>    { yydrop_buffer_state(); }           /* end of .include */
if (!YY_CURRENT_BUFFER)
    yyterminate();
%%
```

## 10.2 yacc.ypp: парсер /bison/

ypp.ypp

```

%{
#include "hpp.hpp"
%}
%defines %union { Sym*o; }          /* universal bI abstract symbolic
%token <o> SYM STR NUM DIR DOC   /* symbol 'string' number .direc
%token <o> LP RP LQ RQ LC RC LV RV /* () [] {} ◇
%token <o> EQ AT TILD COLON      /* = @ ~ :
%token <o> DOT COMMA PERC         /* . , %
%token <o> ADD SUB MUL DIV POW    /* + - * / ^
%token <o> INS DEL               /* += insert -= delete
%token <o> MAP                  /* |
%type <o> ex scalar list lambda  /* expression scalar [ list ] {lam
%type <o> vector cons op bracket /* <vector> co,ns operator brack

%left INS
%left DOC
%left EQ
%left ADD SUB
%left MUL DIV
%left POW
%right AT
%right COMMA
%left PFX
%left TILD
```

```

%left PERC
%left COLON
%left DOT
%%
REPL : | REPL ex
;
ex      : scalar | DIR
| ex DOC
| LP ex RP
| LQ list RQ
| LC lambda RC
| LV vector RV
| TILD ex
| TILD op
| cons
| ADD ex %prec PFX
| SUB ex %prec PFX
| ex EQ ex
| ex AT ex
| ex COLON ex
| ex DOT ex
| ex PERC ex
| ex ADD ex
| ex SUB ex
| ex MUL ex
| ex DIV ex
| ex POW ex
| ex INS ex
| ex DEL ex
| ex MAP ex
;
op      : bracket |EQ |AT |TILD |COLON |DOT |COMMA |ADD |SUB |MUL |D
bracket : LP |RP |LQ |RQ |LC |RC |LV |RV ;
scalar  : SYM | STR | NUM ;
;
cons    : ex COMMA ex      { $$=new Cons($1,$3); } ;
list    :          | list ex { $$=new List(); }
|           { $$=$1; $$->push($2); }
;
lambda  :          | lambda SYM COLON { $$=$1; $$->par($2); }
| lambda ex      { $$=$1; $$->push($2); }
;
vector   :          | vector ex { $$=new Vector(); }
| vector ex     { $$=$1; $$->push($2); }
;
%%

/* REPL with full parse/eval logging */
{ cout << $2->dump();
cout << "\n-----";
cout << $2->eval()->dump();
cout << "\n-----\n"; } ;

```

В качестве типа-хранилища для узлов синтаксического дерева идеально подходит базовый символьный тип *bI* 7.1, 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<sup>1</sup>. 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.

---

<sup>1</sup> их можно вообще рассматривать как элементарные частицы Computer Science, правда к ним еще придется добавить PTR: божественный указатель

## Часть IV

**skelex:** скелет программы в  
свободном синтаксисе

В этом разделе описана общая структура любого проекта, использующего принципы *программирования в свободном синтаксисе*, в виде примера определения синтаксиса и семантики языка *bI*.

Материал дублирует другие разделы, но может быть использован как вариант **минимизированного** языкового ядра FSP-проекта: нет комментариев, лишних классов, подробного описания работы ядра и т.п., **только краткие пояснения и минимальный код**.

## Структура проекта

### Создание проекта

```
git clone -o gh git@github.com:user/lexprogram.git
cd lexprogram
touch src.src log.log \
      ypp.ypp lpp.lpp.hpp.hpp.cpp.cpp Makefile .gitignore
gvim -p src.src log.log ... Makefile .gitignore >> bat.bat
bat.bat
```

src.src	<i>bI</i>	текст программы в свободном синтаксисе
log.log	<i>bI</i>	лог интерпретатора
ypp.ypp	<b>bison</b>	парсер синтаксиса
lpp.lpp	<b>flex</b>	лексер
hpp.hpp	<i>C<sub>+</sub><sup>+</sup></i>	хедеры
cpp.cpp	<i>C<sub>+</sub><sup>+</sup></i>	ядро интерпретатора
Makefile	<b>make</b>	скрипты сборки проекта
.gitignore	<b>git</b>	маски файлов, не попадающие в git-проект
bat.bat	<i>Windows</i>	helper запуска ( <b>g)Vim</b>

### .gitignore

```
*~  
*.swp  
exe.exe  
log.log  
ypp.tab.?pp  
lex.yy.c
```

### bat.bat

```
@start .
@gvim -p src.src log.log ypp.ypp lpp.lpp.hpp.hpp.cpp.cpp Makefile
```

## Makefile

## Makefile

```
MODULE = $(notdir $(CURDIR))
log.log: ./exe.exe src.src
    ./exe.exe < src.src > log.log && tail $(TAIL) log.log
C = cpp.cpp ypp.tab.cpp lex.yy.c
H = hpp.hpp ypp.tab.hpp
CXXFLAGS = -std=gnu++11 -DMODULE=\\"$(MODULE)\\"
./exe.exe: $(C) $(H)
$(CXX) $(CXXFLAGS) -o $@ $(C)
ypp.tab.cpp: ypp.ypp
bison $<
lex.yy.c: lpp.lpp
flex $<
```

MODULE имя программного модуля, в примере получается автоматически из имени каталога проекта; при компиляции интерпретатора добавляется как глобальная константа, и может быть использована в скриптах.

TAIL = -n7|-n17|<none> при успешном выполнении интерпретатора выводятся последние \$(TAIL) строк лога, при отладке скриптов удобно добавлять **в конец программы** вывод отладочной информации. Конкретное значение параметра команды **tail** выбирается в зависимости от настроек вашей IDE, для **eclipse** на старом 15" мониторе мне удобен TAIL=-n7, для **(g)Vim** и командной строки можно увеличить до TAIL=-n17.

CURDIR полный путь для текущего каталога

\$(notdir ...) функция выделяет из полного пути последний /элемент

## ypp.ypp: синтаксический парсер

Весь код между %{...%} будет скопирован в выходной сгенерированный файл ypp.tab.cpp

Заголовочная часть с  $C_+^+$  кодом

```
%{
#include "hpp.hpp"
%}
```

используем универсальный тип для хранения дерева разбора

```
%defines %union { Sym*; }
```

токены для скалярных типов

```
%token <o> SYM NUM STR /* symbol number 'string' */
```

правило для скалярных типов

scalar : SYM | NUM | STR ;

## символ, число и строка — атомы информатики

токены для скобок

%token <o> LP RP LQ RQ LC RC /\* ( ) [ ] { } \*/

[L]eft/[R]ight [P]arens, [Q]uad, [C]url

пачка операторов IV

%token <o> EQ AT TILD PERC PIPE /\* = @ ~ % | \*/  
%token <o> COLON DOT COMMA /\* : . , \*/  
%token <o> ADD SUB MUL DIV POW /\* + - \* / ^ \*/  
%token <o> LL GG /\* < > \*/

типы выражений

%type <o> ex scalar /\* expression scalar \*/  
%type <o> list lambda /\* [ list ] { la:mbda } \*/

правила парсера помещаются между

%%

REPL-цикл интерпретатора

REPL : | REPL ex { cout << \$2->eval()->dump(); } ;

скаляры

scalar : SYM | NUM | STR ;

выражения

ex : scalar  
| LP ex RP { \$\$=\$2; }  
| LQ list RQ { \$\$=\$2; }  
| LC lambda RC { \$\$=\$2; }  
| ex COMMA ex { \$\$=new Cons(\$1,\$3); }  
| TILD ex { \$\$=\$1; \$\$->push(\$2); }  
;

списки

list : { \$\$= new List(); }  
| list ex { \$\$=\$1; \$\$->push(\$2); }  
;

## лямбда-определения

```
lambda : { $$= new List(); }
| lambda SYM COLON { $$=$1; $$->par($2); }
| lambda ex { $$=$1; $$->push($2); }
;
```

## lpp.lpp: лексер

Весь код между `%{...%}` будет скопирован в выходной сгенерированный файл `lex.yy.c`

Заголовочная часть с  $C_+^+$  кодом

```
%{
#include "hpp.hpp"
string LexString;
%}
```

определенна дополнительная переменная `LexString`: буфер используемый при разборе строк.

опция

```
%option noyywrap
```

подавляет вывод сообщений об отсутствии функции `yywrap`

опция включения счетчика нумерации строк

```
%option yylineno
```

сохраняет в переменной `yylineno` номер текущей строки

правила лексера помещаются между

```
%%
```

строчные комментарии

```
#[^\n]* { }
```

разбор строк через специальное состояние лексера

```
%x lexstring
```

```
,
```

```
<lexstring> {BEGIN(lexstring); LexString="";}
```

```
{BEGIN(INITIAL);
```

```
yylval.o = new Str(LexString); return STR; }
```

```
<lexstring>\t {LexString+='t';}
```

```
<lexstring>\\n      {LexString+='\n';}
<lexstring>\n      {LexString+=yytext[0];}
<lexstring>.      {LexString+=yytext[0];}
```

### распознавание чисел

```
S [\+\-]?
N [0-9]+
```

{S}{N}[eE]{S}{N}	TOC(Num,NUM)
{S}{N}\.{N}	TOC(Num,NUM)
{S}{N}	TOC(Int ,NUM)
0x[0-9A-F]+	TOC(Hex ,NUM)
0b[01]+	TOC(Bin ,NUM)

### односимвольные операторы

\=	TOC(Op ,EQ)
\@	TOC(Op ,AT)
\~	TOC(Op ,TILD )
\%	TOC(Op ,PERC)
\	TOC(Op ,PIPE )
\:	TOC(Op ,COLON)
\.	TOC(Op ,DOT)
\,	TOC(Op ,COMMA)
\+	TOC(Op ,ADD)
\-	TOC(Op ,SUB)
\*	TOC(Op ,MUL)
\	TOC(Op ,DIV)
\^	TOC(Op ,POW)
\<	TOC(Op ,LL)
\!	TOC(Op ,EX)
\>	TOC(Op ,GG)

## hpp.hpp: хедеры

```
#ifndef _H_SKELEX
#define _H_SKELEX
```

все остальное находится между препроцессорными “скобками”, блокирующими многократное включение кода

```
#endif // _H_SKELEX
```

```
#include
```

```
#include <iostream>
#include <sstream>
#include <cstdlib>
#include <vector>
#include <map>
using namespace std;
```

универсальный тип: Abstract Symbolic Type

```
struct Sym {
    string tag, val;
    Sym(string, string); Sym(string);
    vector<Sym*> nest; void push(Sym*);
    map<string, Sym*> pars; void par(Sym*);
    virtual string tagval(); string tagstr();
    virtual string dump(int=0); string pad(int);
    virtual Sym* eval();
    virtual Sym* eq(Sym*);
    virtual Sym* at(Sym*);
};
```

глобальная среда (таблица символов)

```
extern map<string, Sym*> env;
extern void env_init();
```

скаляры: строки

```
struct Str: Sym { Str(string); string tagval(); };
```

скаляры: числа

```
struct Int: Sym { Int(string); long val; string tagval(); };
struct Num: Sym { Num(string); double val; string tagval(); };
struct Hex: Sym { Hex(string); };
struct Bin: Sym { Bin(string); };
```

КОМПОЗИТЫ

```
struct List: Sym { List(); };
struct Cons: Sym { Cons(Sym*, Sym*); };
```

функционалы: оператор

```
struct Op: Sym { Op(string); };
```

## встроенные функции

```
typedef Sym*(*FN)(Sym*);  
struct Fn: Sym { Fn(string ,FN); FN fn; };
```

## лямбда-функции

```
struct Lambda: Sym { Lambda(); };
```

## интерфейс к лексеру/парсеру

```
extern int yylex();  
extern int yylineno;  
extern char* yytext;  
#define TOC(C,X) { yyval.o = new C(yytext); return X; }  
extern int yyparse();  
extern void yyerror(string);  
#include "ypp.tab.hpp"
```

## cpp.cpp: ядро интерпретатора

```
#include "hpp.hpp"
```

## обработка ошибок синтаксического анализатора

```
#define YYERR "\n\n<<yylineno<<: "<<msg<< [ "<<yytext<<"]\n\n"  
void yyerror(string msg) { cout<<YYERR; cerr<<YYERR; exit(-1); }
```

## функция main()

```
int main() { env_init(); return yyparse(); }
```

## конструкторы AST

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

```
void Sym::push(Sym*o) { nest.push_back(o); }  
void Sym::par(Sym*o) { pars[o->val]=o; }
```

## дамп AST

```
string Sym::tagval() { return "<" + tag + ":" + val + ">"; }  
string Sym::pad(int n) { string S; for (int i=0;i<n; i++) S+='\t'; ret  
string Sym::dump(int depth) { string S = "\n" + pad(depth)+tagval();  
for (auto it=nest.begin(), e=nest.end(); it!=e; it++)  
    S += (*it)->dump(depth+1);  
return S; }
```

```
Sym* Sym:: eval () {
    Sym*E = env[ val ]; if (E) return E;
    for (auto it=nest.begin(), e=nest.end(); it!=e; it++)
        (*it) = (*it)->eval ();
    return this; }
```

```
Sym* Sym:: eq (Sym*o) { env[ val ]=o; return o; }
Sym* Sym:: at (Sym*o) { push(o); return this; }
```

## строки и Sym::tagstr()

```
Str:: Str (string V):Sym("str",V) {}
string Str:: tagval () { return tagstr (); }
string Sym:: tagstr () { string S = '"';
    for (int i=0,n=val.length(); i<n; i++) {
        char c=val[ i ]; switch (c) {
            case '\t': S+="\\t"; break;
            case '\n': S+="\\n"; break;
            default: S+=c;
        }
    }
return S+""; }
```

## числа

```
Int:: Int (string V):Sym("int","",0) { val=atoi(V.c_str()); }
string Int:: tagval () { ostringstream os;
    os << "<" << tag << ":" << val << ">" ; return os.str(); }

Num:: Num (string V):Sym("num","",0) { val=atof(V.c_str()); }
string Num:: tagval () { ostringstream os;
    os << "<" << tag << ":" << val << ">" ; return os.str(); }
```

```
Hex:: Hex (string V):Sym("hex",V) {}
Bin:: Bin (string V):Sym("bin",V) {}
```

## КОМПОЗИТЫ

```
List:: List ():Sym("[","]") {}
```

## функционалы: оператор

```
Op:: Op (string V):Sym("op",V) {}
```

## встроенная функция

```
Fn:: Fn (string V, FN F):Sym("fn",V) { fn=F; }
```

```
Lambda :: Lambda () : Sym( "^", "^" ) { }
```

глобальная таблица символов

```
map<string ,Sym*> env;
void env_init() {
    env[ "MODULE" ] = new Sym(MODULE);
}
```

## Тестирование интерпретатора

### Комментарии

**test/comment.src**

```
# this is line comment from # till end of line
```

**test/comment.log**

## Скаляры и базовые композиты

**test/coretypes.src**

```
# core scalar and composite types
```

```
[                      # numbers / nested list /
 [                      # integers / list /
 -01 , 00 , +002      # int's / linked cons/
 0x12AF                # machine hex
 0b1101                # binary string
 ]
                      # floating numbers / cons/
 -01.230 ,             # point
 -04e+05                # exponential
 ]
symbol 'string
can\tbe
multilined ,
```

**test/coretypes.log**

```
<[:]>
  <:>
    <int:-1>
    <:>
      <int:0>
      <int:2>
    <hex:0x12AF>
    <bin:0b1101>
  <:>
    <num:-1.23>
    <num:-400000>
<:symbol>
'string\ncan\tbe\n\tmultilined'
```

---

## Операторы

A+B	add	сложение
A-B	sub	вычитание
A*B	mul	умножение
A/B	div	деление
A^B	pow	возведение в степень
A>>B	rsh	правый сдвиг
A<<B	lsh	левый сдвиг
<hr/>		
A>B	great	больше
A=>B	greateq	больше или равно
A<B	less	меньше
A<=B	lesseq	меньше или равно
A==B	eq	равно
A!=B	noteq	неравно
A&B	and	и
A B	or	или
A^B	xor	исключающее или
!A	not	не

---

A=B	assign	назначение/присвоение переменной <i>A предварительно вычисляется</i> ,
A@B	apply	результат является указателем на переменную применение (функции) <i>A</i> к (параметру) <i>B</i> применимо не только к функциям: в общем случае <i>A</i> может быть любым типом в том числе классом: в роли конструктора объекта
~A	quote	<b>блокировка вычисления</b> выражения <i>A</i>
A    B	map	применить распределенно <i>A</i> к членам <i>B</i> функция <i>map</i> : <i>A</i> функция, вычислить список → список параллельное вычисление: <i>A</i> constant-функция $f(x) = x$ <i>A@B</i> вычисляются параллельно при наличии поддержки в ядре интерпретатора
A%>B	member	вложить <i>B</i> как член <i>A</i> чаще всего используется в определении (добавлении) членов класса
A:B	inherit	наследовать <i>B</i> от <i>A</i> если <i>A</i> составное, выполняется множественное наследование в порядке итерации если <i>A</i> <b>не класс</b> , выполняется наследование копированием
A.B	index	доступ по индексу: <i>B</i> -ый член <i>A</i> <i>B</i> может быть именем или числовым индексом вложенного элемента из <i>A</i>
A<>B	symm	симметричное правило замены $A \leftrightarrow B$
A>>B	is	одностороннее правило замены $A \rightarrow B$
A<!>B	notsym	симметричный запрет замены $A \nleftrightarrow B$
A!>B	notis	односторонний запрет замены $A \not\rightarrow B$

## **Часть V**

# **emLinux для встраиваемых систем**

# Структура встраиваемого микро*Linux*

## **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<sup>2</sup>, 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

## **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

<sup>2</sup> IDE/SATA HDD, CDROM, USB флешка, сетевая загрузка с BOOTP-сервера по Ethernet

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

## Процедура сборки

## Глава 11

clock: коридорные электронные  
часы = контроллер умного  
дурдома

## Глава 12

gambox: игровая приставка

## Часть VI

# GNU Toolchain и $C_+^+$ для встраиваемых систем

# Глава 13

# Программирование встраиваемых систем с использованием GNU Toolchain [23]

© Vijay Kumar B.<sup>1</sup> перевод <sup>2</sup>

## 13.1 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 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

<sup>1</sup> © <http://bravegnu.org/gnu-eprog/>

<sup>2</sup> © <https://github.com/ponyatov/gnu-eprog/blob/ru/gnu-eprog.asciidoc>

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 [13.18](#).

## 13.2 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<sup>3</sup> эмулятор нескольких распространенных аппаратных платформ. Вы можете написать программу на ассемблере и  $C_+$ , скомпилировать ее используя **GNU Toolchain** и отладить ее в эмуляторе **Qemu**.

### 13.2.1 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<sup>4</sup> 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.

### 13.2.2 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<sup>5</sup>. Установим пакет командой:

```
$ sudo apt install qemu
```

### 13.2.3 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

<sup>3</sup> для i386 — программно-аппаратный, использует средства виртуализации хост-компьютера

<sup>4</sup> Точнее SoC: система-на-кристалле

<sup>5</sup> хотя может быть и очень хочется

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

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

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

3. Добавьте bin-каталог тулчайна 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-компиляции из исходных текстов, описанный в 15.

### 13.3 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<sup>7</sup>, символов \_ и \$.

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

**инструкция** может быть инструкцией процессора или директивой ассемблера, начинаящейся с точки “.”

Вот пример простой ассемблерной программы 3 для процессора ARM, складывающей два числа:

<sup>6</sup> подразделение Mentor Graphics

<sup>7</sup> не может быть первым символом метки

### Листинг 3: Сложение двух чисел

```
.text
start:
    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.

#### 13.3.1 Сборка бинарника

Сохраните программу в файл **add.s**<sup>8</sup>. Для ассемблирования файла вызовите ассемблер **as**:

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

Опция -o указывает выходной файл с *объектным кодом*, имеющий стандартное расширение .o<sup>9</sup>.

Команды кросс-тулчайна всегда имеют префикс целевой архитектуры (target triplet), для которой они были собраны, чтобы предотвратить конфликт имен с хост-тулчайном 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

```
add.o:      file format elf32-littlearm
add.o
architecture: armv4, flags 0x00000010:
HAS_SYMS
```

<sup>8</sup> .S или .S стандартное расширение в мире OpenSource, указывает что это файл с программной на ассемблере

<sup>9</sup> и внутренний формат ELF (как завещал великий *Linux*)

```
start address 0x00000000
private flags = 5000000: [ Version5 EABI]
```

### Sections:

Idx	Name	Size	VMA	LMA	File off	Algn
0	.text	00000010	00000000	00000000	00000034	2**2
		CONTENTS, ALLOC, LOAD, READONLY, CODE				
1	.data	00000000	00000000	00000000	00000044	2**0
		CONTENTS, ALLOC, LOAD, DATA				
2	.bss	00000000	00000000	00000000	00000044	2**0
		ALLOC				
3	.ARM.attributes	00000014	00000000	00000000	00000044	2**0
		CONTENTS, READONLY				

### SYMBOL TABLE:

00000000	l	d	.text	00000000	.text
00000000	l	d	.data	00000000	.data
00000000	l	d	.bss	00000000	.bss
00000000	l		.text	00000000	start
0000000c	l		.text	00000000	stop
00000000	l	d	.ARM.attributes	00000000	.ARM.attributes

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

машичный код из секции .text: **objdump -d**

```
add.o:      file format elf32-littlearm
```

Disassembly of section .text:

```
00000000 <start>:
```

```
 0:   e3a00005      mov r0, #5
  4:   e3a01004      mov r1, #4
  8:   e0812000      add r2, r1, r0
```

```
0000000c <stop>:
```

```
 c:   eaffffff      b    c <stop>
```

Для генерации **исполняемого файла**<sup>10</sup> 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 <sup>11</sup>:

<sup>10</sup> обычно тот же формат ELF.о, сплленный из одного или нескольких объектных файлов, с некоторыми модификациями см. опцию -T далее

<sup>11</sup> NaMes

```
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 начинаются с `0x0`. Это адрес первой инструкции. Метка `stop` находится после третьей инструкции. Каждая инструкция занимает 4 байта<sup>12</sup>, так что `stop` находится по адресу  $0xC_{hex} = 12_{dec}$ . **Линковка** с другим **базовым адресом** `-Ttext=nnnn` приведет к сдвигу адресов, назначенных меткам.

```
$ 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<sup>13</sup>. Формат ELF применяется для хранения машинного кода, если вы запускаете его в базовой ОС<sup>14</sup>, но поскольку мы собираемся запускать нашу программу на bare metal<sup>15</sup>, 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.<sup>16</sup>

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

<sup>12</sup> 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

<sup>13</sup> можно отдельно отметить Microsoft COFF (объектные файлы .obj) и PE (.exe)cutable

<sup>14</sup> прежде всего “большой” или встраиваемый *Linux*

<sup>15</sup> голом железе

<sup>16</sup> Та же операция выполняется и для SoC-систем с NAND-флешем: записать бинарный образ начиная с некоторого аппаратно фиксированного адреса.

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

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

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

Проверим размер полученного бинарного файла, он должен быть равен тем же 16 байтам<sup>17</sup>:

```
$ 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:   ea\xff\xfe      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**

### 13.3.2 Выполнение в **Qemu**

Когда ARM-процессор сбрасывается, он начинает выполнять команды с адресе 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 16М нулей из файла **/dev/zero** с помощью команды **dd**. Данные копируются 4Кбайтными блоками<sup>18</sup> (4096 x 4K):

<sup>17</sup> 4 инструкции по 4 байта каждая

<sup>18</sup> опция **bs=** (blocksize)

```
$ 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 с, 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 с, 92,5 kB/c
```

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.

### 13.3.3 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: ea\xff\fe    b    0xc
```

## 13.4 Директивы ассемблера

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

### 13.4.1 Суммирование массива

Следующий код 4 вычисляет сумму массива байт и сохраняет результат в r3:

## Листинг 4: Сумма массива

```
.text
entry: b start
arr:    .byte 10, 20, 25
eoa:           .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 асSEMBлируются в последовательность байт в памяти. Также существуют аналогичные директивы .2byte и .4byte для асSEMBлирования 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, асSEMBлируемая 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.

### 13.4.2 Вычисление длины строки

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

#### Листинг 5: 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 своей 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 **символами**, а таблицы записанные в объектные файлы, или в отдельные `.sym` файлы — **таблицами символов**<sup>19</sup>.

Синтаксис директивы `.equ`:

```
.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.

## 13.5 Использование ОЗУ (адресного пространства процессора)

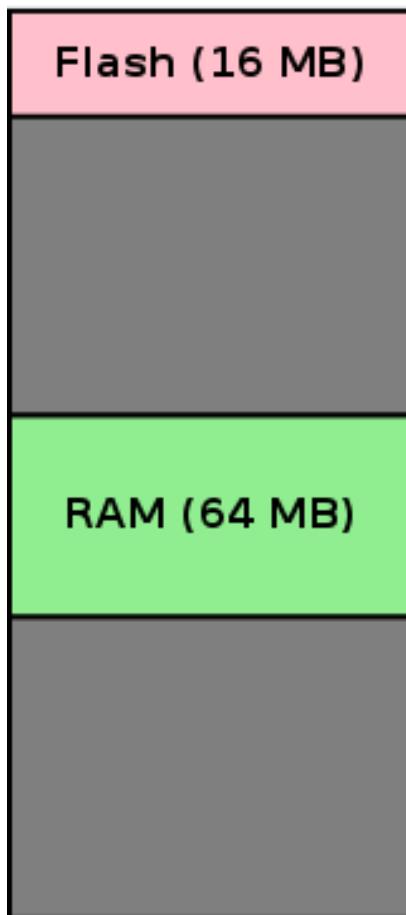
Flash-память описанная ранее, в которой хранится машинный код программы, один из видов EEPROM<sup>20</sup>. Это вторичный носитель данных, как например жесткий диск, но в любом случае хранить данные и значения переменных во флашке неудобно как с точки зрения возможности перезаписи, так и прежде всего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

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

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

<sup>19</sup> также используются отладчиком, чтобы показывать адреса переходов в виде понятных программисту символов, а не мутных числовых констант

<sup>20</sup> Electrical Eraseable Programmable Read-Only Memory, электрически стираемая перепрограммируемая память только-для-чтения



Карта памяти Gumstix

21

Для настройки размещения переменных по нужным физическим адресам **нужна** некоторая **настройка адресного пространства**, особенно **если вы используете внешнюю память и переферийные устройства, подключаемые к внешнейшине**<sup>22</sup>.

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

<sup>21</sup> здесь адреса считаются сверху вниз, что нетипично, обычно на диаграммах памяти адреса увеличиваются снизу вверх.

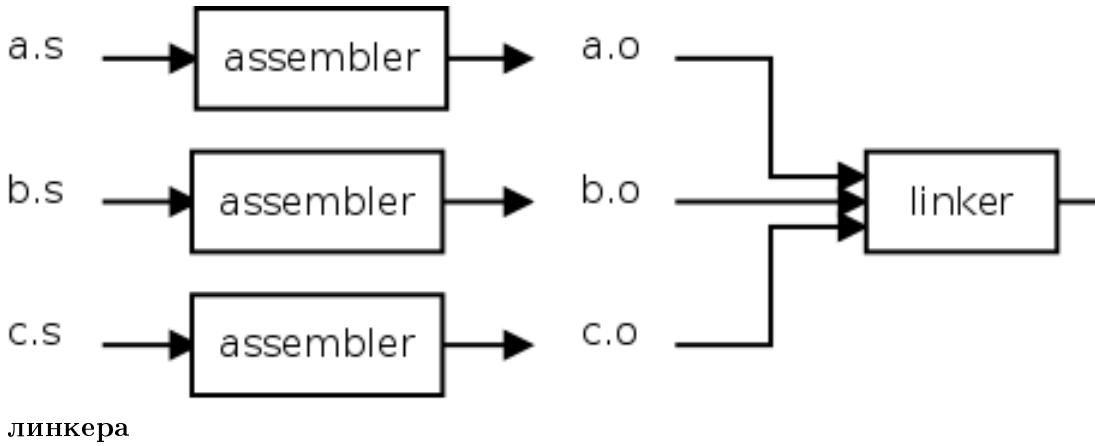
<sup>22</sup> или используете малораспространенные клоны ARM-процессоров, типа Миландровского 1986BE9x “чернобыль”

## 13.6 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 Паскаля.

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<sup>23</sup> в один исполняемый файл.

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



При сборке объектных файлов, линкер выполняет следующие операции:

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

В этой секции мы детально рассмотрим эти операции.

### 13.6.1 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

<sup>23</sup> .a — файлы архивов из пары сотен отдельных .o файлов каждый, по одному .o файлу на каждый возможный вариант функции линейной алгебры

<sup>24</sup> или компоновщик

их “unresolved” (неразрешённые). 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 6: 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 7: 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 элем
      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 lr
```

<sup>25</sup> в архитектуре ARM нет специальной команды возврата ret, ее функцию выполняет прямое присваивание регистра указателя команд mov pc,lr

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*<sup>26</sup>, если с помощью директивы `.global` специально не указано, что они должны быть видимы извне.

Ассемблируйте файлы, и посмотрите дамп их таблицы символов используя комманду `nm`:

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

### 13.6.2 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:

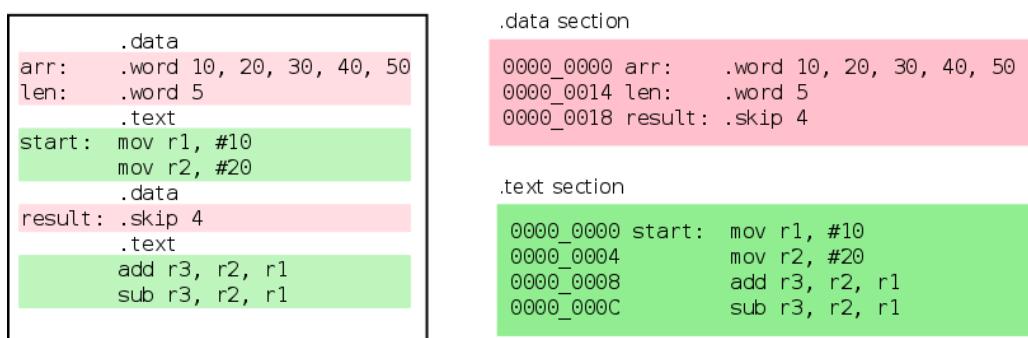
1. Объединение секций
2. Размещение секций

Для понимания процесса 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. Большинство программ имеют как минимум две секции: `.text` для кода и `.data` для данных. Ассемблерные директивы `.text` и `.data` ожидаются использовать для переключения между этими секциями.

<sup>26</sup> или локальными на уровне файла

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 на 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 исполняемых 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 исполняемый 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 простоты используем объектный файл однофайловой программы суммирования `sum.o`. Для увеличения заметности искусственно 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:

a.s (.text)

```
strcpy: ldrb r0, [r1], #1  
strb r0, [r2], #1  
cmp r0, 0  
bne strcpy  
mov pc, lr
```

b.s (.text)

```
strlen: ldrb r0, [r1]  
add r2, #1  
cmp r0, 0  
bne strlen  
mov pc, lr
```

Assembler

```
0000_0000 strcpy: ldrb r0, [r1], #1  
0000_0004 strb r0, [r2], #1  
0000_0008 cmp r0, 0  
0000_000C bne strcpy  
0000_0010 mov pc, lr
```

```
0000_0000 strlen: ldrb r0, [r1]  
0000_0004 add r2, #1  
0000_0008 cmp r0, 0  
0000_000C bne strlen  
0000_0010 mov pc, lr
```

Assembler

Merging .text sections from two files

```
0000_0000 strcpy: ldrb r0, [r1], #1  
0000_0004 strb r0, [r2], #1  
0000_0008 cmp r0, 0  
0000_000C bne strcpy  
0000_0010 mov pc, lr  
0000_0014 strlen: ldrb r0, [r1], #1  
0000_0018 add r2, #1  
0000_001C cmp r0, 0  
0000_0020 bne strlen  
0000_0024 mov pc, lr
```

Patch

New address  
after merge

Placing .text section at 0x2000\_0000

```
2000_0000 strcpy: ldrb r0, [r1], #1  
2000_0004 strb r0, [r2], #1  
2000_0008 cmp r0, 0  
2000_000C bne strcpy  
2000_0010 mov pc, lr  
2000_0014 strlen: ldrb r0, [r1], #1  
2000_0018 add r2, #1  
2000_001C cmp r0, 0  
2000_0020 bne strlen  
2000_0024 mov pc, lr
```

Patch

Объединение и размещение секций

## 13.7 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 8: 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 9: 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 10: 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 привавивать значения, секции `.text` и `.data` будут размещены в смежных областях памяти.

### 13.7.1 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 11: Программа суммирования массива

```
.data
arr: .byte 10, 20, 25 @ Read-only array of bytes
eoa: @ Address of end of array + 1

.text
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
```

- Изменения заключаются в выделении массива в секцию `.data` и удалении директивы выравнивания `.align`.
- 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 символов видно что секция `.text` размещена с адреса 0x0, а секция `.data` с 0x400.

### 13.7.2 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 12: `sum-data.objdump`

1. указание на архитектуру,
2. для которой предназначен исполняемый файл
3. стартовый адрес в секции `.text`, по умолчанию 0x0<sup>27</sup>
4. ***ABI*** — соглашения о передаче
5. параметров в регистрах/стеке (для Си кода)
6. приведена подробная информация о секциях
7. `.text` секция кода
8. `.data` секция данных
9. служебная информация
10. столбец `Size` указывает размер секции в байтах (hex)
11. `VMA`<sup>28</sup> указывает адрес размещения сегмента

---

<sup>27</sup> обязателен и фиксирован для прошивок микроконтроллеров, т.к. на него перескакивает аппаратный сброс

<sup>28</sup> Virtual Memory Address

12. **Align** (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
18. таблица символов
19. дизассемблированный код из секций, помеченных флагом **CODE**: `.text`

### 13.8 Данные в RAM, пример

Теперь мы знаем как писать скрипты линкера, и можем попытаться написать программу, разместив данные в секции `.data` в ОЗУ.

Программа сложения модифицирована для загрузки значений из ОЗУ, и записи результата обратно в ОЗУ: память для операндов и результат размещена в секции `.data`.

#### Листинг 13: Данные в ОЗУ

```
.data
val1: .4byte 10 @ First number
val2: .4byte 30 @ Second number
result: .4byte 0 @ 4 byte space for result
```

```
.text
.align
start:
ldr r0, =val1 @ r0 = &val1
ldr r1, =val2 @ r1 = &val2
```

```
ldr    r2, [r0] @ r2 = *r0
ldr    r3, [r1] @ r3 = *r1

add    r4, r2, r3 @ r4 = r2 + r3

ldr    r0, =result @ r0 = &result
str    r4, [r0] @ *r0 = r4

stop: b stop
```

#### Листинг 14: Скрипт для линковки

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

. = 0xA0000000;
.data : { * (.data); }
}
```

Дамп таблицы символов:

```
$ arm-none-eabi-nm -n add-mem.elf
00000000 t start
00000001c t stop
a0000000 d val1
a0000001 d val2
a0000002 d result
```

Скрипт линкера решил проблему с размещением данных, но **проблема с использованием ОЗУ еще не решена !**

#### 13.8.1 RAM энергозависима (volatile)!

ОЗУ стирается при отключении питания, поэтому для использования ОЗУ недостаточно разместить сегменты.

**Во флеше должен храниться** не только код, но **и данные**, чтобы при подаче питания специальный **startup код** выполнил **инициализацию ОЗУ**, копируя данные из флеша. Затем управление передается основной программе.

Поэтому секция `.data` имеет **два адреса размещения**: **адрес хранения** во флеше **LMA** и **адрес размещения** в ОЗУ **VMA**.

TIP: как видно из раздела ??, в терминах **ld** адрес хранения (загрузки) называется **LMA** (Load Memory Address), а адрес размещения (времени выполнения) **VMA** (Virtual Memory Address).

Нужно сделать следующие две модификации, чтобы программа работала корректно:

1. модифицировать .lds чтобы для секции .data в нем учитывались оба адреса: LMA и VMA.
2. написать небольшой кусочек кода, который будет **инициализировать память данных**, копируя образ секции .data из флеша (из адреса хранения LMA) в ОЗУ (по адресу исполнения, VMA).

### 13.8.2 Спецификация адреса загрузки LMA

VMA это адрес, который должен быть использован для вычисления адресов всех меток при исполнении программы. В предыдущем линк-скрипте мы задали VMA секции .data. LDA не указан, и по умолчанию равен VMA. Это нормально для сегментов, размещаемых в ROM. Но если используются инициализируемые сегменты в ОЗУ, нужно задать отдельно VMA и LMA.

Адрес загрузки LMA, отличающийся от адреса выполнения VMA, задается с помощью команды AT. Модифицированный скрипт показан ниже:

```
SECTIONS {
. = 0x00000000;
.text : { * (.text); }
etext = .; <1>

. = 0xA0000000;
.data : AT (etext) { * (.data); } <2>
}
```

1. В блоках описания секций можно создавать символы, назначая им значения: числовой адрес или текущую позицию с помощью точки ". ". Символу **etext** назначается адрес флеша, следующий сразу за концом кода. Отметим что **etext** сам по себе не выделяет никакой памяти, а только помечает адрес LMA сегмента .data в таблице символов.
2. При настройке сегмента .data с помощью ключевого слова AT (**etext**) назначается LMA для хранения содержимого сегмента данных. Команде AT может быть передан любой адрес или символ<sup>29</sup>. Так что в результате мы настроили адрес хранения .data на область флеша, помеченную символом **etext**.

<sup>29</sup> значением которого является валидный адрес

### 13.8.3 Копирование ‘.data’ в ОЗУ

Для копирования данных инициализации из флеши в ОЗУ требуется следующая информация:

1. Адрес данных во флеше (`flash_sdata`)
2. Адрес данных в ОЗУ (`ram_sdata`)
3. Размер секции `.data` (`data_size`)

Имея эту информацию, сегмент `.data` может быть инициализирован может быть скопирован следующим стартовым кодом:

```
ldr r0, =flash_sdata
ldr r1, =ram_sdata
ldr r2, =data_size
copy:
ldrb r4, [r0], #1
strb r4, [r1], #1
subs r2, r2, #1
bne copy
```

Для получения такой информации скрипт линкера нужно немного модифицировать:

Листинг 15: Скрипт линкера с символами для копирования секции `.data`

```
SECTIONS {
. = 0x00000000;
.text : { * (.text); }
flash_sdata = .; <1>

. = 0xA0000000;
ram_sdata = .; <2>
.data : AT(flash_sdata) { * (.data); }
ram_edata = .; <3>
data_size = ram_edata - ram_sdata; <3>
}
```

1. Начало данных во флеше сразу за секцией кода.
2. Начало данных — базовый адрес ОЗУ в адресном пространстве процессора.
3. Получение размера непросто: размер вычисляется вычитанием адресов метод начала и конца данных. Да, простые выражения тоже можно использовать в скрипте линкера.

Полный листинг программы с добавленной инициализацией данных:

## Листинг 16: Инициализация ОЗУ

```
.data
val1: .4byte 10 @ First number
val2: .4byte 30 @ Second number
result: .space 4 @ 1 byte space for result

.text

;; Copy data to RAM.
start:
ldr r0, =flash_sdata
ldr r1, =ram_sdata
ldr r2, =data_size

copy:
ldrb r4, [r0], #1
strb r4, [r1], #1
subs r2, r2, #1
bne copy

;; Add and store result.
ldr r0, =val1 @ r0 = &val1
ldr r1, =val2 @ r1 = &val2

ldr r2, [r0] @ r2 = *r0
ldr r3, [r1] @ r3 = *r1

add r4, r2, r3 @ r4 = r2 + r3

ldr r0, =result @ r0 = &result
str r4, [r0] @ *r0 = r4

stop: b stop
```

**Листинг 17: add-ram.objdump** Программа была ассемблирована и скомпилирована используя .lds в ??.

Запуск и тестирование программы в Qemu:

```
qemu-system-arm -M connex -pflash flash.bin -nographic -serial /dev/null
(qemu) xp /4dw 0xA0000000
a0000000:          10              30              40              0
```

На реальной физической системе с SDRAM, память не может использована сразу. Сначала нужно инициализировать контроллер памяти, и только затем обращаться к ОЗУ. Наш код работает потому, что симулятор не требует инициализации контроллера.

## 13.9 Обработка аппаратных исключений

Все примеры программ, приведенные выше, содержат гигантский баг: **первые 8 машинных слов в адресном пространстве зарезервированы для векторов исключений**. Когда возникает исключение, выполняется аппаратный переход на один из этих жестко заданных меток. Исключения и их адреса приведены в следующей таблице:

### Адреса векторов исключений

Исключение		Адрес
Сброс	Reset	0x00
Неопределенная инструкция	Undefined Instruction	0x04
Программное прерывание (SWI)	Software Interrupt (SWI)	0x08
Ошибка предвыборки	Prefetch Abort	0x0C
Ошибка данных	Data Abort	0x10
Резерв, не используется	Reserved, not used	0x14
Аппаратное прерывание	IRQ	0x18
Быстрое прерывание	FIQ	0x1C

Предполагается что по этим адресам находятся команды перехода, которые передадут управление на соответствующий произвольный адрес обработчика исключения. Во всех примерах ранее бы не вставляли таблицу обработчиков исключений, так как мы предполагали что эти исключения не случатся. Все эти программы можно скорректировать, слинковав их со следующим кодом:

```
.section "vectors"
reset: b      start
undef: b      undef
swi: b      swi
pabt: b     pabt
dabt: b     dabt
nop
irq: b      irq
fiq: b      fiq
```

Только обработчик `reset` векторизован на отдельный адрес `start`. Все остальные исключения векторизованы сами на себя. Таким образом если случится любое исключение, процессор зациклится на соответствующем векторе. В этом случае

возникшее исключение может быть идентифицировано в отладчике (мониторе Qemu, в нашем случае) по адресу указателя команд `pc=r15`.

В ассемблерном коде видно применение директивы `.section` которая позволяет создавать секции с произвольными именами, чтобы прописать для них отдельную обработку в скрипте линкера.

Чтобы обеспечить правильное размещение таблицы обработчиков, нужно скорректировать скрипт линкера:

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

Обратите внимание что секция `vectors` размещена сразу за инициализацией указателя размещения на первом месте, до всего остального кода, что гарантирует что таблица векторов будет находиться по жесткому адресу `0x0`.

## 13.10 Стартап-код на Си

Если процесс только что был сброшен, невозможно напрямую выполнить Си-код, так как в отличие от ассемблера, программы на Си требуют для себя некоторой предварительной настройки. В этом разделе описаны эти предварительные требования, и как их выполнить.

Мы возьмем пример Си-программы которая вычисляет сумму массива. И к концу раздела мы уже будем способны, сделав некоторые низкоуровневые настройки, передать управление и выполнить ее.

### Листинг 18: Сумма массива на Си

```
static int arr[] = { 1, 10, 4, 5, 6, 7 };
static int sum;
static const int n = sizeof(arr) / sizeof(arr[0]);

int main()
{
int i;

for (i = 0; i < n; i++)
sum += arr[i];
```

}

Перед передачей управления Си-коду, нужно выполнить следующие настройки:

1. Стек
2. Глобальные переменные
  - (а) Инициализированные
  - (б) Неинициализированные
3. Константные данные

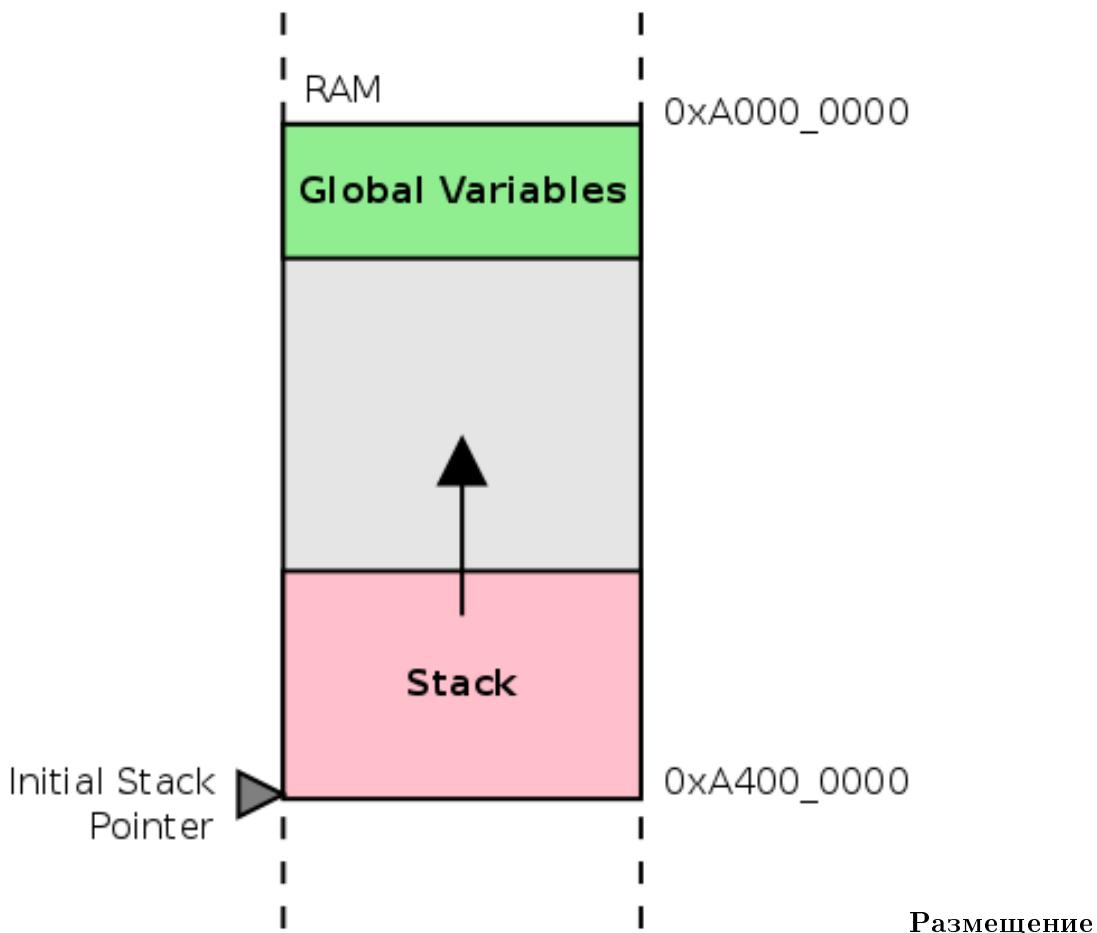
### 13.10.1 Стек

Си использует стек для хранения локальных (авто) переменных, передачи аргументов и результата функций, хранения адресов возврата из функций и т.д. Так что необходимо чтобы стек был настроен корректно перед передачей управление Си-коду.

На архитектуре ARM стеки очень гибкие, поэтому их реализация полностью ложиться на программное обеспечение. Для людей не знакомых с ARM, некоторый обзор приведен в ??.

Чтобы быть уверенным, что разные части кода, сгенерированного **разными** компиляторами, работали друг с другом, ARM создал [Стандарт вызова процедур для архитектуры ARM \(AAPCS\)](#). В нем описаны регистры которые должны быть использованы для работы со стеком и направление в котором растет стек. Согласно AAPCS, **регистр r13** должен быть использован для указателя стека. Также стек должен быть для указателя стека. Также стек должен быть **full-descending** (нисходящим).

Один из способов размещения глобальных переменных на стеке показан в диаграмме:



стека

Так что все, что нужно сделать в стартовом коде для стека — выставить **r13** на старший адрес ОЗУ, так что стек может расти вниз (в сторону младших адресов). Для платы **connex** это можно сделать командой

```
ldr sp, =0xA4000000
```

Обратите внимание что ассемблер предоставляет алиас **sp** для регистра **r13**.

Адрес 0xA4000000 сам по себе не указывает на ОЗУ. ОЗУ кончается адресом 0xA3FFFFFF. Но это нормально, так как стек **full-descending**, т.е. во время первого **push** в стек указатель **сначала уменьшится**, и только потом значение будет записано уже в ОЗУ.

## 13.10.2 Глобальные переменные

Когда компилируется Си-код, компилятор размещает инициализированные глобальные переменные в секцию `.data`. Как и для ассемблера, сегмент `.data` должен быть скопирован стартовым кодом в ОЗУ из флеша.

Язык Си гарантирует что все неинициализированные глобальные переменные будут инициализированы нулем<sup>30</sup>. Когда Си-программа компилируется, создается отдельный сегмент `.bss` для неинициализированных переменных. Так как для всего сегмента должно быть выполнено обнуление, его не нужно хранить во флеше. Перед передачей управления на Си-код, содержимое `.bss` должно быть зачищено startup-кодом.

## 13.10.3 Константные данные

GCC размещает переменные, помеченные модификатором `const`, в отдельный сегмент `.rodata`. Также `.rodata` используется для хранения всех "строковых констант".

Так как содержимое `.rodata` не модифицируется, оно может быть размещено в Flash/ROM. Для этого нужно модифицировать `.lds`.

## 13.10.4 Секция `.eeprom` (AVR8)

При написании прошивок для Atmel AVR8, существует модификатор `EEMEM` определенный в `avr/eeprom.h`:

```
#define EEMEM __attribute__((section(".eeprom")))
```

который использует модификатор GCC `__attribute__((section("...")))`, который приписывает объект данных к любой указанной секции. В частности, секция `.eeprom` выделяется из финального объектного файла, и программируется в Atmel ATmega отдельным вызовом `avrdude` (ПО программатора).

## 13.10.5 Стартовый код

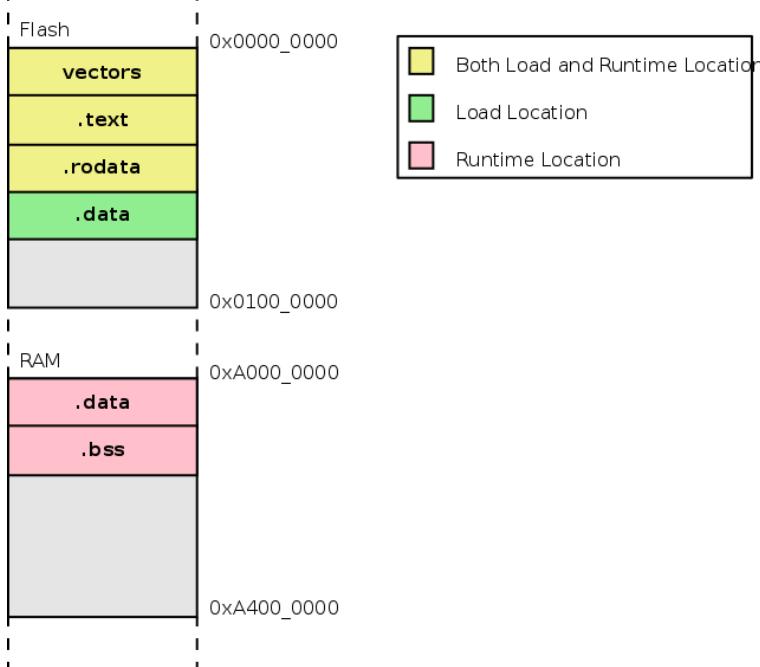
Теперь все готово к написанию скрипта линкера и стартового кода. Скрипт ?? модифицируется с добавлением размещения секций:

1. `.bss`
2. `vectors`
3. `.rodata`

Секция `.bss` размещается сразу за секцией `.data` в ОЗУ. Также создаются символы маркирующие начало и конец секции `.bss`, которые будут использованы в startup-коде при ее очистке. `.rodata` размещается сразу за `.text` во флеше:

---

<sup>30</sup> старый стандарт Си не гарантировал



## Размещение секций

Листинг 19: Скрипт линкера для Си кода

```

SECTIONS {
    . = 0x00000000;
    .text : {
        * (vectors);
        * (.text);
    }
    .rodata : {
        * (.rodata);
    }
    flash_sdata = .;

    . = 0xA0000000;
    ram_sdata = .;
    .data : AT (flash_sdata) {
        * (.data);
    }
    ram_edata = .;
    data_size = ram_edata - ram_sdata;
}

```

```
sbss = .;
.bss : {
    * (.bss);
}
ebss = .;
bss_size = ebss - sbss;
}
```

Startup-код включает следующие части:

1. вектора исключений
2. код копирования `.data` из Flash в RAM
3. код обнуления `.bss`
4. код установки указателя стека
5. переход на `_main`

#### Листинг 20: Стартовый код для Си программы на ассемблере

```
.section "vectors"
reset: b      start
undef: b      undef
swi: b       swi
pabt: b      pabt
dabt: b      dabt
nop
irq: b      irq
fiq: b      fiq

.text
start:
@ Copy data to RAM.
ldr   r0, =flash_sdata
ldr   r1, =ram_sdata
ldr   r2, =data_size

@ Handle data_size == 0
cmp   r2, #0
beq   init_bss
copy:
ldrb  r4, [r0], #1
strb r4, [r1], #1
subs r2, r2, #1
bne  copy

init_bss:
```

```
@@ Initialize .bss
ldr    r0, =sbss
ldr    r1, =ebss
ldr    r2, =bss_size

@@ Handle bss_size == 0
cmp    r2, #0
beq    init_stack

mov    r4, #0
zero:
strb   r4, [r0], #1
subs   r2, r2, #1
bne    zero

init_stack:
@@ Initialize the stack pointer
ldr    sp, =0xA4000000

bl    main

stop: b    stop
```

Для компиляции кода не требуется отдельно вызывать ассемблер, линкер и компилятор Си: программа **gcc** является оберткой, которая умеет делать это сама, автоматически вызывая ассемблер, компилятор и линкер в зависимости от типов файлов. Поэтому мы можем скомпилировать весь наш код одной командой:

```
$ arm-none-eabi-gcc -nostdlib -o csum.elf -T csum.lds csum.c startup.s
```

Опция **-nostdlib** используется для указания, что нам при компиляции не нужно подключать стандартную библиотеку Си (**newlib**). Эта библиотека крайне полезна, но для ее использования нужно выполнить некоторые дополнительные действия, описанные в разделе **??**.

Дамп таблицы символов даст больше информации о расположении объектов в памяти:

```
$ arm-none-eabi-nm -n csum.elf
00000000 t reset <1>
00000004 A bss_size
00000004 t undef
00000008 t swi
0000000c t pabt
00000010 t dabt
```

```
000000018 A data_size
000000018 t irq
00000001c t fiq
000000020 T main
000000090 t start <2>
000000a0 t copy
000000b0 t init_bss
000000c4 t zero
000000d0 t init_stack
000000d8 t stop
000000f4 r n <3>
000000f8 A flash_sdata
a0000000 d arr <4>
a0000000 A ram_sdata
a0000018 A ram_edata
a0000018 A sbss
a0000018 b sum <5>
a000001c A ebss
```

1. `reset` и остальные вектора исключений размещаются с `0x0`.
2. ассемблерный код находится сразу после 8 векторов исключений ( $8 * 4 = 32$ )
3. константные данные `n`, размещены во флеше после кода.
4. инициализированные данные `arr`, массив из 6 целых, размещен с начала ОЗУ `0xA0000000`.
5. неинициализированные данные `sum` размещен после массива из 6 целых. ( $6 * 4 = 24 = 0x18$ )

Для выполнения программы преобразуем ее в `.bin` формат, запустим в **Qemu**, и выведем дамп переменной `sum` по адресу `0xA0000018`:

```
$ arm-none-eabi-objcopy -O binary csum.elf csum.bin
$ dd if=/dev/zero of=flash.bin bs=4K count=4K
$ dd if=csum.bin of=flash.bin bs=4096 conv=notrunc
$ qemu-system-arm -M connex -pflash flash.bin -nographic -serial /dev/null
(qemu) xp /6dw 0xa0000000
a0000000:      1          10          4          5
a0000010:      6          7
(qemu) xp /1dw 0xa0000018
a0000018:     33
```

## 13.11 Использование библиотеки Си

**FIXME:** Эту секцию еще нужно написать.

## 13.12 Inline-ассемблер

FIXME: Эту секцию еще нужно написать.

## 13.13 Использование ‘make’ для автоматизации компиляции

Если вам надоело каждый раз вводить длинные команды, компилируя примеры из этого учебника, пришло время научиться пользоваться утилитой **make**. **make** это утилита, отслеживающая зависимости файлов, описанные в файле **Makefile**.

Умение читать и писать makeфайлы **must have** навык для программиста, особенно для больших многофайловых проектов, содержащих сотни и тысячи файлов, которые должны быть ассемблированы, откомпилированы или оттрансформированы в различные форматы.

Каждая зависимости между двумя или более файлами прописывается в **make-правиле**:

```
<цель> : <источник>
<tab><команда компиляции 1>
<tab><команда компиляции 2>
...
...
```

**цель** одно имя файла, или несколько имен, разделенных пробелами. Этот файл(ы) будут созданы или обновлены этим правилом.

**источник** 0+ имен файлов разделенных пробелами. Эти файл(ы) будут ‘проводиться на изменения’ используя метку времени последней модификации.

**tab** символ табуляции с ascii кодом 0x09, вы должны использовать текстовый редактор, который умеет работать с табуляциями, не заменяя их последовательностями пробелов.

**команда компиляции** любая команда, такая как вызов ассемблера или линкера, которая обновляет **цель**, выполняя некоторую полезную работу. **make-правило может не иметь команд компиляции**, если вам нужно прописать только зависимость файлов.

основной принцип каждого make-правила: если один из файлов-источников **новее** чем один из целевых файлов, будет выполнено тело правила, которое обновит **цели**.

Давайте напишем простой **Makefile** для простейшей программы, описанной в разделе ??:

## Листинг 21: Makefile

```
emulation: add.flash
    qemu-system-arm -M connex -pflash add.flash \
        -nographic -serial /dev/null
flash.bin: add.bin
    dd if=/dev/zero of=flash.bin bs=4K count=4K
    dd if=add.bin of=flash.bin bs=4K conv=notrunc
add.bin: add.elf
    arm-none-eabi-objcopy -O binary add.elf add.bin
add.elf: add.o
    arm-none-eabi-ld -o add.elf add.o
add.o: add.s
    arm-none-eabi-as -o add.o add.s
```

- обратите внимание на обратный слэш и следующую табулированную строку: вы можете делить длинные команды на несколько строк; каждая строка должна быть табулирована для следования синтаксису make-правила.

Введине в командной строке команду **make** без параметров, находясь в каталоге проета, в котором находится **Makefile** и исходные тексты программы, и вы сразу получите автоматически скомпилированные бинарные файлы и запущенный **Qemu**:

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

Если вы запускаете **make** без параметров, **первое правило** в **Makefile** будет обработано как **главная цель**, с обходом всех зависимостей в других правилах.

### 13.13.1 Выбор конкретной *цели*

Если вам нужно обновить только определенный файл-*цель*, поместите необходимое имя файла после команды **make**:

```
$ make add.o
make: 'add.o' is up to date.
```

Эта команда будет перекомпилировать только файл **add.o**, в том и только в том случае, если вы перед запуском команды изменили **add.s**. Если вы видите

сообщение типа **make**: **add.o is up to date.**, **исходные файлы не менялись**, и **make не будет запускать правило ассемблирования**.

Это очень полезно если у вас очень много файлов исходников<sup>31</sup>, и вы изменили несколько символов в одном файле. Без **make**<sup>32</sup> каждое микроскопическое изменение потребует перекомпиляции всего проекта, которое может длиться **несколько часов (!)**. Использование **make** позволяет выполнить всего несколько вызовов компилятора и линкера, что будет намного намного быстрее.

Возарашаясь к нашему **add.o**, вы можете заставить ассемблер выполниться не изменяя файл **add.s**, через команду **touch**:

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

Команда **touch** изменяет только дату модификации исходного файла **add.s**, не меняя его содержимое, так что **make** увидит что этот файл обновился, и запустит ассемблер для указанной цели **add.o**.

По умолчанию **make** выводит каждую команду и ее вывод. Если у вас есть какие-то причины для "тихой" работы **make**, вы можете добавить префикс "-" (минус) к командам компиляции.

### 13.13.2 Переменные

## 13.14 13. Contributing

## 13.15 14. Credits

### 13.15.1 14.1. People

### 13.15.2 14.2. Tools

## 13.16 15. Tutorial Copyright

## 13.17 A. ARM Programmer's Model

## 13.18 B. ARM Instruction Set

## 13.19 C. ARM Stacks

<sup>31</sup> например тысячи файлов, как у ядра *Linux*

<sup>32</sup> используя простой .rc shell-скрипт или .batник

## Глава 14

# Embedded Systems Programming in $C_+^+$ [22]

1

# Глава 15

## Сборка кросс-компилятора **GNU Toolchain** из исходных текстов

Если вам по каким-то причинам не подходит одна из типовых сборок кросс-компиляторов, поставляемых в виде готовых бинарных пакетов из репозитория вашего дистрибутива *Linux*, **GNU Toolchain** можно легко скопилировать **из исходных текстов** и установить в систему, даже имея только пользовательские права доступа.

Сборка **GNU Toolchain** из исходников может понадобиться, если вы хотите:

- самую свежую или какую-то конкретную версию **GNU Toolchain**
- опции компиляции: малораспространенный **target**-процессор, **нетиповой формат файлов объектного кода**<sup>1</sup> или экспериментальные оптимизаторы, не включенные в бинарные пакеты из дистрибутива *Linux*
- полпроцента ускорения работы компилятора благодаря жесткой оптимизации его машинного кода точно под ваш рабочий компьютер (**-march=native** -

При сборке используется утилита **make 13.13**, которой можно передать набор переменных конфигурирования. В таблице перечислен набор переменных конфигурирования сборки с указанием их значения по умолчанию<sup>2</sup> и имя mk-файла, где оно задано:

<sup>1</sup> например для i386 может понадобится сборка кросс-компилятора с **-target=i486-none-elf VIII** или **i686-linux-uclibc** вместо типовой компиляции для *Linux* типа **i486-linux-gnu**

<sup>2</sup> также приведены часто используемые варианты значения

APP	cross	Makefile	приложение: условное имя проекта (только латиница, буквы a-z)
HW	x86	Makefile	qemu vmware virtualpc x86 pc686 amd64 cortexM avr8
CPU	i386	hw/\$(HW).mk	
ARCH	i386	cpu/\$(CPU).mk	
TARGET	\$(CPU)-pc-elf	hw/\$(HW).mk	i686-linux-uclibc x86_64-linux i386-pc-elf arm-none-eabi avr-

## APP/HW: приложение/платформа

Для сборки необходимо выбрать имя проекта<sup>3</sup> и аппаратную платформу, для которой будет настраиваться пакет кросс-компилятора.

Особенно это важно для варианта сборки, когда собирается не только кросс-компилятор, но и базовая ОС — минимальная *Linux*-система из ядра, libc и дополнительных прикладных библиотек. В этом случае **APP/HW** используются для формирования имен файлов ядра **\$(APP)\$(HW).kernel**, названия и состава загрузочного образа **\$(APP)\$(HW).rootfs**, и внутренних настроек.

## Подготовка BUILD-системы: необходимое ПО

Для сборки необходимо установить следующие пакеты:

```
sudo apt install gcc g++ make flex bison m4 bc bzip2 xz-utils libncurses-
```

## dirs: создание структуры каталогов

```
user@bs:~/boox/cross$ make dirs
mkdir -p
/home/user/boox/cross/gz /home/user/boox/cross/src /home/user/boox/cross/toolchain /home/user/boox/cross/root
```

Командой `make dirs` создается набор вспомогательных каталогов:

TC	\$(CWD)/\$(APP)\$(ROOT).cross	каталог установки кросс-компилятора
ROOT	\$(CWD)/\$(APP)\$(ROOT)	каталог файловой системы для целевого
CWD	\$(CURDIR)	текущий каталог
GZ	\$(CWD)/gz	архивы исходных текстов GNU Toolchain, загрузчика, и библиотек
SRC	\$(CWD)/src	каталог для распаковки исходников
TMP	\$(CWD)/tmp	каталог для out-of-tree сборки GNU toolchain

<sup>3</sup> только латиница, буквы a-z

---

```
CWD = $(CURDIR)
```

```
GZ = $(CWD) / gz
```

```
SRC = $(CWD) / src
```

```
TMP = $(CWD) / tmp
```

```
ROOT = $(CWD) / $(APP) $(HW)
```

```
TC = $(CWD) / $(APP) $(HW).cross
```

```
DIRS = $(GZ) $(SRC) $(TMP) $(TC) $(ROOT)
```

```
.PHONY: dirs
```

```
dirs:
```

```
    mkdir -p $(DIRS)
```

---

## Сборка в ОЗУ на ramdiske

Если у вас есть админские права и достаточный объем RAM, после выполнения `make dirs` рекомендуется примонтировать на каталоги `SRC` и `TMP` файловую систему `tmpfs` — это значительно ускорит компиляцию, т.к. все временные файлы будут храниться только в ОЗУ:

```
/etc/fstab
```

---

```
tmpfs /home/user/src tmpfs auto,uid=yourlogin,gid=yourgroup 0 0
tmpfs /home/user/tmp tmpfs auto,uid=yourlogin,gid=yourgroup 0 0
```

Если вы прописали монтирование `ramdisk`ов в `/etc/fstab`, или сделали `mount -t` вручную, может оказаться нужным запускать `make` с явным указанием значений переменных `SRC/TMP`:

```
make blablabla SRC=/home/user/src TMP=/home/user/tmp
```

## Пакеты системы кросс-компиляции

### GNU Toolchain

---

```
1 # bintools: assembler, linker, objfile tools
2 BINUTILS_VER= 2.24
3 # 2.25 build error
4
5 # gcc: C/C++ cross-compiler
6 GCC_VER      = 4.9.2
7 # 4.9.2 used: bug arm/62098 fixed
```

```

8
9 # gcc support libraries
10 ## required for GCC build
11 GMP_VER      = 5.1.3
12 MPFR_VER     = 3.1.3
13 MPC_VER      = 1.0.2
14 ## loop optimisation
15 ISL_VER       = 0.11.1
16 # 0.11 need for binutils build
17 CLOOG_VER     = 0.18.1
18
19 # standard C/POSIX library libc (newlib)
20 NEWLIB_VER    = 2.3.0.20160226
21
22 # loader for i386 target
23 SYSLINUX_VER  = 6.03
24
25 # packages
26 BINUTILS      = binutils-$(BINUTILS_VER)
27 GCC            = gcc-$(GCC_VER)
28 GMP            = gmp-$(GMP_VER)
29 MPFR           = mpfr-$(MPFR_VER)
30 MPC            = mpc-$(MPC_VER)
31 ISL            = isl-$(ISL_VER)
32 CLOOG          = cloog-$(CLOOG_VER)
33 NEWLIB         = newlib-$(NEWLIB_VER)
34 SYSLINUX       = syslinux-$(SYSLINUX_VER)

```

make

**newlib** стандартная библиотека **libc**

**gz:** загрузка исходного кода для пакетов

```
user@bs$ make APP=cross HW=x86 GZ=/home/user/gz gz
```

В примере команды показано два обязательных параметра **APP/HW<sup>4</sup>** и необязательный **GZ**: поскольку я собираю кросс-компиляторы для нескольких целевых платформ, я создал каталог **\$(HOME)/gz** и загружаю туда архивы исходников **для всех проектов сразу<sup>5</sup>**. Более простой способ – просто сделать симлинк **ln -s ~/gz project/gz** и не переопределять переменную **GZ** явно.

---

<sup>4</sup> по ним могут закачиваться дополнительные файлы исходников, зависящие от платформы — например исходник загрузчика или бинарные файлы (блöбы) драйверов от производителя железки

<sup>5</sup> а не в **/gz** каждого проекта, нет смысла дублировать исходники **GNU Toolchain** одной и той же версии

mk/gz.mk

---

```
WGET = -wget -N -P $(GZ) -t2 -T2
```

```
.PHONY: gz
```

```
gz: gz_cross gz_libs gz_$(ARCH)
```

```
.PHONY: gz_cross
```

```
gz_cross:
```

```
$(WGET) ftp://ftp.gnu.org/pub/gmp/$(GMP).tar.bz2
```

```
$(WGET) http://www.mpfr.org/mpfr-current/$(MPFR).tar.bz2
```

```
$(WGET) http://www.multiprecision.org/mpc/download/$(MPC).tar.gz
```

```
$(WGET) ftp://gcc.gnu.org/pub/gcc/infrastructure/$(ISL).tar.bz2
```

```
$(WGET) ftp://gcc.gnu.org/pub/gcc/infrastructure/$(CLOOG).tar.gz
```

```
$(WGET) http://ftp.gnu.org/gnu/binutils/$(BINUTILS).tar.bz2
```

```
$(WGET) http://gcc.skazkaforyou.com/releases/$(GCC)/$(GCC).tar.bz2
```

```
.PHONY: gz_libs
```

```
gz_libs:
```

```
$(WGET) ftp://sourceware.org/pub/newlib/$(NEWLIB).tar.gz
```

```
.PHONY: gz_i386
```

```
gz_i386:
```

```
$(WGET) https://www.kernel.org/pub/linux/utils/boot/syslinux/$(SY
```

## Макро-правила для автоматической распаковки исходников

mk/src.mk

```
$(SRC)/%/README: $(GZ)%.tar.gz
```

```
    cd $(SRC) && zcat $< | tar x && touch $@
```

```
$(SRC)/%/README: $(GZ)%.tar.bz2
```

```
    cd $(SRC) && bzcat $< | tar x && touch $@
```

```
$(SRC)/%/README: $(GZ)%.tar.xz
```

```
    cd $(SRC) && xzcat $< | tar x && touch $@
```

## Общие параметры для .configure

mk/cfg.mk

---

```
# configure parameters for all packages
```

```
CFG_ALL = --disable-nls --disable-werror \
```

```
    --docdir=$(TMP)/doc --mandir=$(TMP)/man --infodir=$(TMP)/info
```

```
# [B]uild host configure
```

```
BCFG = configure $(CFG_ALL) --prefix=$(TC)
XPATH = PATH=$(TC)/bin:$PATH
# [T]arget configure
TCFG = configure $(CFG_ALL) --prefix=$(ROOT) CC=$(TARGET)-gcc
# get cpu cores
CPU_CORES ?= $(shell grep processor /proc/cpuinfo | wc -l)
# run make with -j flag or make CPU_CORES=<none> for one thread build
MAKE = make -j$(CPU_CORES)
INSTALL = make install
```

## 15.1 Сборка кросс-компилятора

Для пакетов кросс-компилятора существуют два варианта сборки пакетов:

**Пакеты с 0 в конце имени** задают сборку программ, которые будут выполняться на BUILD-компьютере, и компилировать код для TARGET-системы, т.е. это простейший вариант кросс-компиляции.

**Пакеты без 0**, которые могут появиться в будущем — **относятся только к сборке emLinux**, собирают кросс-компилятор **канадским крестом**:

- пакет собирается на BUILD-системе — ваш рабочий компьютер,
- выполняется на HOST-системе — например PC104 или роутер с emLinux,
- и компилирует код для TARGET-микропроцессора — модуль ввода/вывода на USB, подключенный к PC104)

### 15.1.1 cclibs0: библиотеки поддержки gcc

Для сборки **GNU Toolchain** необходим набор нескольких библиотек, причем **успешность сборки сильно зависит от их версий**, поэтому библиотеки **нужно собрать из исходников**, а не использовать девелоперские пакеты из дистрибутива BUILD-Linux.

Библиотеки чисел произвольной точности:

**gmp0** целых

**gmfr0** с плавающей точкой

**gmc0** комплексных

Библиотеки для работы с графами (нужны для компилятора оптимизатора **Graphite**)

**cloog0** polyhedral оптимизации

**isl0** манипуляция наборами целочисленных точек

```

WITH_CCLIBS0 = --with-gmp=$(TC) --with-mpfr=$(TC) --with-mpc=$(TC) \
    --without-ppl --without-cloog
# --with-isl=$(TC) --with-cloog=$(TC)

CFG_CCLIBS0 = $(WITH_CCLIBS0) --disable-shared
.PHONY: cclibs0
cclibs0: gmp0 mpfr0 mpc0
# cloog0 isl0

CFG_GMP0 = $(CFG_CCLIBS0)
.PHONY: gmp0
gmp0: $(SRC) $(GMP) / README
    rm -rf $(TMP) $(GMP) && mkdir -p $(TMP) $(GMP) && cd $(TMP) $(GMP)
        $(SRC) $(BCFG) $(CFG_GMP0) && $(MAKE) && $(INSTALL)-strip

CFG_MPFR0 = $(CFG_CCLIBS0)
.PHONY: mpfr0
mpfr0: $(SRC) $(MPFR) / README
    rm -rf $(TMP) $(MPFR) && mkdir -p $(TMP) $(MPFR) && cd $(TMP) $(MPFR)
        $(SRC) $(BCFG) $(CFG_MPFR0) && $(MAKE) && $(INSTALL)-strip

CFG_MPC0 = $(CFG_CCLIBS0)
.PHONY: mpc0
mpc0: $(SRC) $(MPC) / README
    rm -rf $(TMP) $(MPC) && mkdir -p $(TMP) $(MPC) && cd $(TMP) $(MPC)
        $(SRC) $(BCFG) $(CFG_MPC0) && $(MAKE) && $(INSTALL)-strip

CFG_CLOOG0 = --with-gmp-prefix=$(TC) $(CFG_CCLIBS0)
.PHONY: cloog0
cloog0: $(SRC) $(CLOOG) / README
    rm -rf $(TMP) $(CLOOG) && mkdir $(TMP) $(CLOOG) && cd $(TMP) $(CLOOG)
        $(SRC) $(BCFG) $(CFG_CLOOG0) && $(MAKE) && $(INSTALL)-strip

CFG_ISL0 = --with-gmp-prefix=$(TC) $(CFG_CCLIBS0)
.PHONY: isl0
isl0: $(SRC) $(ISL) / README
    rm -rf $(TMP) $(ISL) && mkdir $(TMP) $(ISL) && cd $(TMP) $(ISL)
        $(SRC) $(BCFG) $(CFG_ISL0) && $(MAKE) && $(INSTALL)-strip

```

### 15.1.2 binutils0: ассемблер и линкер

Чтобы побыстрее получить результат, который можно сразу потестировать, соберем сначала кросс-**binutils**, а потом все что относится к Сициальному компилятору<sup>6</sup>.

**-target** триплет целевой системы, например **i386-pc-elf**

---

<sup>6</sup> на самом деле **binutils0** надо собирать после **cclibs0**, так как есть зависимость от библиотек **isl0** и **cloog0**

CFG\_ARCH CFG\_CPU задаются в файлах `arch/$(ARCH).mk` и `cpu/$(CPU).mk`,  
и определяют опции сборки `binutils/gcc` для конкретного процессора<sup>7</sup>

-`with-sysroot` каталог где должны храниться файлы для целевой системы: откомпилированные библиотеки и каталог `include`

-`with-native-system-header-dir` имя каталога с `include`-файлами, относительно `sysroot`

arch/i386.mk

CFG\_ARCH =

cpu/i386.mk

ARCH = i386

CFG\_CPU = --with-cpu=i386 --with-tune=i386

mk/bintools.mk

CFG\_BINUTILS0 = --target=\$(TARGET) \$(CFG\_ARCH) \$(CFG\_CPU) \  
--with-sysroot=\$(ROOT) --with-native-system-header-dir=/include \  
--enable-lto --disable-multilib \$(WITH\_CCLIBS0) \  
--disable-target-libiberty --disable-target-zlib \  
--disable-bootstrap --disable-decimal-float \  
--disable-libmudflap --disable-libssp \  
--disable-libgomp --disable-libquadmath

.PHONY: binutils0

binutils0: \$(SRC)/\$(BINUTILS)/README

rm -rf \$(TMP)/\$(BINUTILS) && mkdir -p \$(TMP)/\$(BINUTILS) && cd \$(SRC)/\$(BINUTILS)/\$(BCFG) \$(CFG\_BINUTILS0) && \$(MAKE) && \$(INSTA

Файлы `binutils0` с TARGET- префиксами и типовые скрипты линкера

crossx86 . cross/bin/i386-pc-elf-readelf  
crossx86 . cross/bin/i386-pc-elf-addr2line  
crossx86 . cross/bin/i386-pc-elf-size  
crossx86 . cross/bin/i386-pc-elf-objdump  
crossx86 . cross/bin/i386-pc-elf-objcopy  
crossx86 . cross/bin/i386-pc-elf-nm  
crossx86 . cross/bin/i386-pc-elf-ld.bfd  
crossx86 . cross/bin/i386-pc-elf-elfedit  
crossx86 . cross/bin/i386-pc-elf-as  
crossx86 . cross/bin/i386-pc-elf-ranlib  
crossx86 . cross/bin/i386-pc-elf-c++filt  
crossx86 . cross/bin/i386-pc-elf-gprof

<sup>7</sup> например `-without-fpu` для `cpu/i486sx.mk`

```
crossx86.cross/bin/i386-pc-elf-ar
crossx86.cross/bin/i386-pc-elf-strip
crossx86.cross/bin/i386-pc-elf-strings

crossx86.cross/i386-pc-elf/lib/ldscripts/elf_i386.xr
crossx86.cross/i386-pc-elf/lib/ldscripts/elf_i386.xsc
crossx86.cross/i386-pc-elf/lib/ldscripts/elf_i386.xdc
crossx86.cross/i386-pc-elf/lib/ldscripts/elf_i386.xu
crossx86.cross/i386-pc-elf/lib/ldscripts/elf_i386.xc
crossx86.cross/i386-pc-elf/lib/ldscripts/elf_i386.x
crossx86.cross/i386-pc-elf/lib/ldscripts/elf_i386.xbn
crossx86.cross/i386-pc-elf/lib/ldscripts/elf_i386.xsw
crossx86.cross/i386-pc-elf/lib/ldscripts/elf_i386.xs
crossx86.cross/i386-pc-elf/lib/ldscripts/elf_i386.xw
crossx86.cross/i386-pc-elf/lib/ldscripts/elf_i386.xn
crossx86.cross/i386-pc-elf/lib/ldscripts/elf_i386.xdw
crossx86.cross/i386-pc-elf/lib/ldscripts/elf_i386.xd
```

### 15.1.3 **gcc00**: сборка stand-alone компилятора Си

Сборка кросс-компилятора Си выполняется в два этапа

**gcc00** минимальный **gcc** необходимый для сборки libc ??

**newlib** сборка стандартной библиотеки Си

**gcc0** пересборка полного кросс-компилятора Си/ $C_+^+$

mk/gcc.mk

CFG\_GCC\_DISABLE =

```
CFG_GCC00 = $(CFG_BINUTILS0) $(CFG_GCC_DISABLE) \
--disable-threads --disable-shared --without-headers --with-newl
--enable-languages="c"
```

```
CFG_GCC0 = $(CFG_BINUTILS0) $(CFG_GCC_DISABLE) \
--with-newlib \
--enable-languages="c,c++"
```

.PHONY: gcc00

gcc00: \$(SRC)/\$(GCC)/README

```
rm -rf $(TMP)/$(GCC) && mkdir -p $(TMP)/$(GCC) && cd $(TMP)/$(GCC)
$(SRC)/$(GCC)/$(BCFG) $(CFG_GCC00)
cd $(TMP)/$(GCC) && $(MAKE) all-gcc && $(INSTALL)-gcc
cd $(TMP)/$(GCC) && $(MAKE) all-target-libgcc && $(INSTALL)-target-libgcc
```

## 15.1.4 newlib: сборка стандартной библиотеки libc

Стандартная библиотека **libc**<sup>8</sup> обеспечивает слой совместимости со стандартом POSIX для ваших программ. Это удобно при адаптации чужих программ под вашу ОС, и при написании собственного **мультиплатформенного** кода.

mk/libc.mk

```
CFG_NEolib = --host=$(TARGET)  
.PHONY: newlib  
newlib: $(SRC) / $(NEWLIB) / README  
    rm -rf $(TMP) / $(NEWLIB) && mkdir -p $(TMP) / $(NEWLIB) && cd $(TMP)  
    $(XPATH) $(SRC) / $(NEWLIB) / $(TCFG) $(CFG_NEolib)  
#   && $(MAKE) && $(INSTALL)-strip
```

## 15.1.5 gcc0: пересборка компилятора Си/ $C_+$

# 15.2 Поддерживаемые платформы

## 15.2.1 i386: ПК и промышленные PC104

arch/i386.mk

```
CFG_ARCH =
```

## 15.2.2 x86\_64: серверные системы

arch/x86\_64.mk

## 15.2.3 AVR: Atmel AVR Mega

arch/avr.mk

## 15.2.4 arm: процессоры ARM Cortex-Mx

arch/arm.mk

## 15.2.5 armhf: SoCи Cortex-A, PXA270,..

arch/armhf.mk

<sup>8</sup> для микроконтроллерных систем — обрезанная версия, **newlib**

## 15.3 Целевые аппаратные системы

### 15.3.1 **x86**: типовой компьютер на процессоре i386+

hw/x86.mk

CPU = i386

TARGET = \$(CPU)-pc-elf

---

# Глава 16

## Porting The GNU Tools To Embedded Systems

Embed With GNU

Porting The GNU Tools To Embedded Systems

Spring 1995

Very \*Rough\* Draft

Rob Savoye - Cygnus Support

[http://ieee.uwaterloo.ca/coldfire/gcc-doc/docs/porting\\_toc.html](http://ieee.uwaterloo.ca/coldfire/gcc-doc/docs/porting_toc.html)

# Глава 17

## Оптимизация кода

### 17.1 PGO оптимизация

<sup>1</sup>

## Часть VII

# Микроконтроллеры Cortex-Mx

## Часть VIII

**os86: низкоуровневое  
программирование i386**

Если вам по каким-то причинам не подходит одна из типовых распространенных ОС, например требуется сделать систему управления жесткого реального времени<sup>2</sup>, информация в этом разделе поможет сделать ОС-поделку для типового WinInt ПК.

## Специализированный GNU Toolchain для i386-pc-gnu

Для компиляции кода вам потребуется специально собранный из исходников кросс-**GNU Toolchain** для целевой архитектуры i386 — *triplet* TARGET=i386-pc-elf. Процесс сборки подробно описан в отдельном разделе [15](#).

Для упрощения не будем завязываться на особенности конкретного ПК или эмулятора **Qemu**<sup>3</sup>, все они вполне аппаратно совместимы с любым i386 компьютером в базовой конфигурации, для которого мы и будем рассматривать примеры кода:

- APP=bare metal программирование, без базовой ОС
- HW=x86 типовой минимальный i386 компьютер

os86/Makefile

```
APP = bare
HW = x86
TARGET = i386-pc-elf

TODO = gz dirs cclibs0 binutils0 gcc00 newlib
.PHONY: toolchain
toolchain: $(APP) $(HW).cross /bin /$(TARGET)-g++
$(APP) $(HW).cross /bin /$(TARGET)-g++:
    cd .. / cross; $(MAKE) $(TODO) \
        CWD=$(CURDIR) GZ=$(HOME) /L/gz SRC=$(HOME) /L/src TMP=$(HOME) /L/tmp
    APP=$(APP) HW=$(HW)
```

## MultiBoot-загрузчик

Благодаря усилиям сообщества разработчиков OpenSource была успешно решена одна из проблем начинающего системного программиста — было создано несколько универсальных **загрузчиков**, берущих на себя заботу о чтении ядра ОС или bare metal программы, начальную инициализацию оборудования, включении защищенного режима, и передачу управления вашей ОС.

Чтобы ваша bare metal программа была успешно загружена, она должна удовлетворять требованиям **спецификации MultiBoot IX** быть слинкована в формат ELF и включать заголовок multiboot.

<sup>2</sup> или вы любитель гадить из прикладного ПО в аппаратные порты в обход всех соглашений и средств защиты ОС

<sup>3</sup> VMWare, VirtualPC

## Часть IX

# Спецификация MultiBoot

Этот файл документирует *Спецификацию Multiboot*, проект стандарта на последовательность загрузки. Этот документ имеет редакцию 0.6.96.

Copyright © 1995,96 Bryan Ford <[baford@cs.utah.edu](mailto:baford@cs.utah.edu)>

Copyright © 1995,96 Erich Stefan Boleyn <[erich@uruk.org](mailto:erich@uruk.org)>

Copyright © 1999,2000,2001,2002,2005,2006,2009 Free Software Foundation, Inc.

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided also that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions.

# Глава 18

## Introduction to Multiboot Specification

This chapter describes some rough information on the Multiboot Specification. Note that this is not a part of the specification itself.

### 18.1 The background of Multiboot Specification

Every operating system ever created tends to have its own boot loader. Installing a new operating system on a machine generally involves installing a whole new set of boot mechanisms, each with completely different install-time and boot-time user interfaces. Getting multiple operating systems to coexist reliably on one machine through typical chaining mechanisms can be a nightmare. There is little or no choice of boot loaders for a particular operating system — if the one that comes with the operating system doesn't do exactly what you want, or doesn't work on your machine, you're screwed.

While we may not be able to fix this problem in existing proprietary operating systems, it shouldn't be too difficult for a few people in the free operating system communities to put their heads together and solve this problem for the popular free operating systems. That's what this specification aims for. Basically, it specifies an interface between a boot loader and a operating system, such that any complying boot loader should be able to load any complying operating system. This specification does not specify how boot loaders should work — only how they must interface with the operating system being loaded.

### 18.2 The target architecture

This specification is primarily targeted at i386 PC, since they are the most common and have the largest variety of operating systems and boot loaders. However, to the extent that certain other architectures may need a boot specification and do not have one already, a variation of this specification, stripped of the x86-specific details, could

be adopted for them as well.

## 18.3 The target operating systems

This specification is targeted toward free 32-bit operating systems that can be fairly easily modified to support the specification without going through lots of bureaucratic rigmarole. The particular free operating systems that this specification is being primarily designed for are Linux, the kernels of FreeBSD and NetBSD, Mach, and VSTa. It is hoped that other emerging free operating systems will adopt it from the start, and thus immediately be able to take advantage of existing boot loaders. It would be nice if proprietary operating system vendors eventually adopted this specification as well, but that's probably a pipe dream.

## 18.4 Boot sources

It should be possible to write compliant boot loaders that load the OS image from a variety of sources, including floppy disk, hard disk, and across a network.

Disk-based boot loaders may use a variety of techniques to find the relevant OS image and boot module data on disk, such as by interpretation of specific file systems<sup>1</sup>, using precalculated *blocklists*<sup>2</sup>, loading from a special *boot partition*<sup>3</sup>, or even loading from within another operating system<sup>4</sup>. Similarly, network-based boot loaders could use a variety of network hardware and protocols.

It is hoped that boot loaders will be created that support multiple loading mechanism increasing their portability, robustness, and user-friendliness.

## 18.5 Configure an operating system at boot-time

It is often necessary for one reason or another for the user to be able to provide some configuration information to an operating system dynamically at boot time. While this specification should not dictate how this configuration information is obtained by the boot loader, it should provide a standard means for the boot loader to pass such information to the operating system.

## 18.6 How to make OS development easier

OS images should be easy to generate. Ideally, an OS image should simply be an ordinary 32-bit executable file in whatever file format the operating system normally uses. It should be possible to **nm** or disassemble OS images just like normal executables.

---

<sup>1</sup> e.g. the BSD/Mach boot loader

<sup>2</sup> e.g. LILO

<sup>3</sup> e.g. OS/2

<sup>4</sup> e.g. the VSTa boot code, which loads from DOS

Specialized tools should not be required to create OS images in a **special** file format. If this means shifting some work from the operating system to a boot loader, that is probably appropriate, because all the memory consumed by the boot loader will typically be made available again after the boot process is created, whereas every bit of code in the OS image typically has to remain in memory forever. The operating system should not have to worry about getting into 32-bit mode initially, because mode switching code generally needs to be in the boot loader anyway in order to load operating system data above the 1MB boundary, and forcing the operating system to do this makes creation of OS images much more difficult.

Unfortunately, there is a horrendous variety of executable file formats even among free Unix-like pc-based operating systems — generally a different format for each operating system. Most of the relevant free operating systems use some variant of a.out format, but some are moving to elf. It is highly desirable for boot loaders not to have to be able to interpret all the different types of executable file formats in existence in order to load the OS image — otherwise the boot loader effectively becomes operating system specific again.

This specification adopts a compromise solution to this problem. Multiboot-compliant OS images always contain a magic *Multiboot header* (see OS image format ??), which allows the boot loader to load the image without having to understand numerous a.out variants or other executable formats. This magic header does not need to be at the very beginning of the executable file, so kernel images can still conform to the local a.out format variant in addition to being Multiboot-compliant.

## 18.7 Boot modules

Many modern operating system kernels, such as Mach and the microkernel in VSta, do not by themselves contain enough mechanism to get the system fully operational: they require the presence of additional software modules at boot time in order to access devices, mount file systems, etc. While these additional modules could be embedded in the main OS image along with the kernel itself, and the resulting image be split apart manually by the operating system when it receives control, it is often more flexible, more space-efficient, and more convenient to the operating system and user if the boot loader can load these additional modules independently in the first place.

Thus, this specification should provide a standard method for a boot loader to indicate to the operating system what auxiliary boot modules were loaded, and where they can be found. Boot loaders don't have to support multiple boot modules, but they are strongly encouraged to, because some operating systems will be unable to boot without them.

## The definitions of terms used through the specification

**must** We use the term must, when any boot loader or OS image needs to follow a rule — otherwise, the boot loader or OS image is not Multiboot-compliant.

**should** We use the term should, when any boot loader or OS image is recommended to follow a rule, but it doesn't need to follow the rule.

**may** We use the term may, when any boot loader or OS image is allowed to follow a rule.

**boot loader** Whatever program or set of programs loads the image of the final operating system to be run on the machine. The boot loader may itself consist of several stages, but that is an implementation detail not relevant to this specification. Only the final stage of the boot loader — the stage that eventually transfers control to an operating system — must follow the rules specified in this document in order to be Multiboot-compliant; earlier boot loader stages may be designed in whatever way is most convenient.

**OS image** The initial binary image that a boot loader loads into memory and transfers control to start an operating system. The OS image is typically an executable containing the operating system kernel.

**boot module** Other auxiliary files that a boot loader loads into memory along with an OS image, but does not interpret in any way other than passing their locations to the operating system when it is invoked.

**Multiboot-compliant** A boot loader or an OS image which follows the rules defined as must is Multiboot-compliant. When this specification specifies a rule as should or may, a Multiboot-compliant boot loader/OS image doesn't need to follow the rule.

**u8** The type of unsigned 8-bit data.

**u16** The type of unsigned 16-bit data. Because the target architecture is little-endian, **u16** is coded in **little-endian**.

**u32** The type of unsigned 32-bit data. Because the target architecture is little-endian, **u32** is coded in **little-endian**.

**u64** The type of unsigned 64-bit data. Because the target architecture is little-endian, **u64** is coded in little-endian.

# Глава 19

# The exact definitions of Multiboot Specification

There are three main aspects of a boot loader/OS image interface:

1. The format of an OS image as seen by a boot loader.
2. The state of a machine when a boot loader starts an operating system.
3. The format of information passed by a boot loader to an operating system.

## 19.1 OS image format

An OS image may be an ordinary 32-bit executable file in the standard format for that particular operating system, except that it may be linked at a non-default load address to avoid loading on top of the pc's I/O region or other reserved areas, and of course it should not use shared libraries or other fancy features.

An OS image must contain an additional header called *Multiboot header*, besides the headers of the format used by the OS image. The Multiboot header must be contained completely within the first 8192 bytes of the OS image, and must be longword (32-bit) aligned. In general, it should come **as early as possible**, and may be embedded in the beginning of the text segment after the real executable header.

### 19.1.1 The layout of Multiboot header

The layout of the Multiboot header must be as follows:

Offset	Type	Field Name	Note
0	u32	magic	required
4	u32	flags	required
8	u32	checksum	required
12	u32	header_addr	if flags[16] is set
16	u32	load_addr	if flags[16] is set
20	u32	load_end_addr	if flags[16] is set
24	u32	bss_end_addr	if flags[16] is set
28	u32	entry_addr	if flags[16] is set
32	u32	mode_type	if flags[2] is set
36	u32	width	if flags[2] is set
40	u32	height	if flags[2] is set
44	u32	depth	if flags[2] is set

The fields ‘magic’, ‘flags’ and ‘checksum’ are defined in Header magic fields 19.1.2, the fields ‘header\_addr’, ‘load\_addr’, ‘load\_end\_addr’, ‘bss\_end\_addr’ and ‘entry\_addr’ are defined in Header address fields 19.1.1, and the fields ‘mode\_type’, ‘width’, ‘height’ and ‘depth’ are defined in Header graphics fields 19.1.4.

### 19.1.2 The magic fields of Multiboot header

**‘magic’** The field ‘magic’ is the magic number identifying the header, which must be the hexadecimal value 0x1BADB002.

**‘flags’** The field ‘flags’ specifies features that the OS image requests or requires of an boot loader. Bits 0-15 indicate requirements; if the boot loader sees any of these bits set but doesn’t understand the flag or can’t fulfill the requirements it indicates for some reason, it must notify the user and fail to load the OS image. Bits 16-31 indicate optional features; if any bits in this range are set but the boot loader doesn’t understand them, it may simply ignore them and proceed as usual. Naturally, all as-yet-undefined bits in the ‘flags’ word must be set to zero in OS images. This way, the ‘flags’ fields serves for version control as well as simple feature selection.

If bit 0 in the ‘flags’ word is set, then all boot modules loaded along with the operating system must be aligned on page (4KB) boundaries. Some operating systems expect to be able to map the pages containing boot modules directly into a paged address space during startup, and thus need the boot modules to be page-aligned.

If bit 1 in the ‘flags’ word is set, then information on available memory via at least the ‘mem\_\*’ fields of the Multiboot information structure (see Boot information format ??) must be included. If the boot loader is capable of passing a memory map (the ‘mmap\_\*’ fields) and one exists, then it may be included as well.

If bit 2 in the ‘flags’ word is set, information about the video mode table (see Boot information format ??) must be available to the kernel.

If bit 16 in the ‘flags’ word is set, then the fields at offsets 12-28 in the Multiboot header are valid, and the boot loader should use them instead of the fields in the actual executable header to calculate where to load the OS image. This information does not need to be provided if the kernel image is in elf format, but it must be provided if the images is in a.out format or in some other format. Compliant boot loaders must be able to load images that either are in elf format or contain the load address information embedded in the Multiboot header; they may also directly support other executable formats, such as particular a.out variants, but are not required to.

**‘checksum’** The field ‘checksum’ is a 32-bit unsigned value which, when added to the other magic fields (i.e. ‘magic’ and ‘flags’), must have a 32-bit unsigned sum of zero.

### 19.1.3 The address fields of Multiboot header

All of the address fields enabled by flag bit 16 are physical addresses. The meaning of each is as follows:

**header\_addr** Contains the address corresponding to the beginning of the Multiboot header — the physical memory location at which the magic value is supposed to be loaded. This field serves to **synchronize** the mapping between OS image offsets and physical memory addresses.

**load\_addr** Contains the physical address of the beginning of the text segment. The offset in the OS image file at which to start loading is defined by the offset at which the header was found, minus (header\_addr - load\_addr). load\_addr must be less than or equal to header\_addr.

**load\_end\_addr** Contains the physical address of the end of the data segment. (load\_end\_addr - load\_addr) specifies how much data to load. This implies that the text and data segments must be consecutive in the OS image; this is true for existing a.out executable formats. If this field is zero, the boot loader assumes that the text and data segments occupy the whole OS image file.

**bss\_end\_addr** Contains the physical address of the end of the bss segment. The boot loader initializes this area to zero, and reserves the memory it occupies to avoid placing boot modules and other data relevant to the operating system in that area. If this field is zero, the boot loader assumes that no bss segment is present.

**entry\_addr** The physical address to which the boot loader should jump in order to start running the operating system.

## 19.1.4 The graphics fields of Multiboot header

All of the graphics fields are enabled by flag bit 2. They specify the preferred graphics mode. Note that that is only a recommended mode by the OS image. If the mode exists, the boot loader should set it, when the user doesn't specify a mode explicitly. Otherwise, the boot loader should fall back to a similar mode, if available.

The meaning of each is as follows:

**mode\_type** Contains ‘0’ for linear graphics mode or ‘1’ for EGA-standard text mode.

Everything else is reserved for future expansion. Note that the boot loader may set a text mode, even if this field contains ‘0’.

**width** Contains the number of the columns. This is specified in pixels in a graphics mode, and in characters in a text mode. The value zero indicates that the OS image has no preference.

**height** Contains the number of the lines. This is specified in pixels in a graphics mode, and in characters in a text mode. The value zero indicates that the OS image has no preference.

**depth** Contains the number of bits per pixel in a graphics mode, and zero in a text mode. The value zero indicates that the OS image has no preference.

## 19.2 Machine state

When the boot loader invokes the 32-bit operating system, the machine must have the following state:

**‘EAX’** Must contain the magic value ‘0x2BADB002’; the presence of this value indicate to the operating system that it was loaded by a Multiboot-compliant boot loader (e.g. as opposed to another type of boot loader that the operating system can also be loaded from).

**‘EBX’** Must contain the 32-bit physical address of the Multiboot information structure provided by the boot loader (see Boot information format).

**‘CS’** Must be a 32-bit read/execute code segment with an offset of ‘0’ and a limit of ‘0xFFFFFFFF’. The exact value is undefined.

**‘DS’**

**‘ES’**

**‘FS’**

**‘GS’**

**'SS'** Must be a 32-bit read/write data segment with an offset of '0' and a limit of '0xFFFFFFFF'. The exact values are all undefined.

**'A20 gate'** Must be enabled.

**'CR0'** Bit 31 (PG) must be cleared. Bit 0 (PE) must be set. Other bits are all undefined.

**'EFLAGS'** Bit 17 (VM) must be cleared. Bit 9 (IF) must be cleared. Other bits are all undefined.

All other processor registers and flag bits are undefined. This includes, in particular:

**'ESP'** The OS image must create its own stack as soon as it needs one.

**'GDTR'** Even though the segment registers are set up as described above, the 'GDTR' may be invalid, so the OS image must not load any segment registers (even just reloading the same values!) until it sets up its own 'GDT'.

**'IDTR'** The OS image must leave interrupts disabled until it sets up its own IDT.

However, other machine state should be left by the boot loader in normal working order, i.e. as initialized by the bios (or DOS, if that's what the boot loader runs from). In other words, the operating system should be able to make bios calls and such after being loaded, as long as it does not overwrite the bios data structures before doing so. Also, the boot loader must leave the pic programmed with the normal bios/DOS values, even if it changed them during the switch to 32-bit mode.

## 19.3 Boot information format

FIXME: Split this chapter like the chapter "OS image format".

Upon entry to the operating system, the EBX register contains the physical address of a Multiboot information data structure, through which the boot loader communicates vital information to the operating system. The operating system can use or ignore any parts of the structure as it chooses; all information passed by the boot loader is advisory only.

The Multiboot information structure and its related substructures may be placed anywhere in memory by the boot loader (with the exception of the memory reserved for the kernel and boot modules, of course). It is the operating system's responsibility to avoid overwriting this memory until it is done using it.

The format of the Multiboot information structure (as defined so far) follows:

0	+-----+   flags   (required) +-----+
---	--

4	mem_lower	(present if flags[0] is set)
8	mem_upper	(present if flags[0] is set)
	+-----+	
12	boot_device	(present if flags[1] is set)
	+-----+	
16	cmdline	(present if flags[2] is set)
	+-----+	
20	mods_count	(present if flags[3] is set)
24	mods_addr	(present if flags[3] is set)
	+-----+	
28 - 40	syms	(present if flags[4] or     flags[5] is set)
	+-----+	
44	mmap_length	(present if flags[6] is set)
48	mmap_addr	(present if flags[6] is set)
	+-----+	
52	drives_length	(present if flags[7] is set)
56	drives_addr	(present if flags[7] is set)
	+-----+	
60	config_table	(present if flags[8] is set)
	+-----+	
64	boot_loader_name	(present if flags[9] is set)
	+-----+	
68	apm_table	(present if flags[10] is set)
	+-----+	
72	vbe_control_info	(present if flags[11] is set)
76	vbe_mode_info	
80	vbe_mode	
82	vbe_interface_seg	
84	vbe_interface_off	
86	vbe_interface_len	
	+-----+	

The first longword indicates the presence and validity of other fields in the Multiboot information structure. All as-yet-undefined bits must be set to zero by the boot loader. Any set bits that the operating system does not understand should be ignored. Thus, the ‘flags’ field also functions as a version indicator, allowing the Multiboot information structure to be expanded in the future without breaking anything.

If bit 0 in the ‘flags’ word is set, then the ‘mem\_\*’ fields are valid. ‘mem\_lower’ and ‘mem\_upper’ indicate the amount of lower and upper memory, respectively, in kilobytes. Lower memory starts at address 0, and upper memory starts at address 1 megabyte. The maximum possible value for lower memory is 640 kilobytes. The value returned for upper memory is maximally the address of the first upper memory hole minus 1 megabyte. It is not guaranteed to be this value.

If bit 1 in the ‘flags’ word is set, then the ‘boot\_device’ field is valid, and indicates which bios disk device the boot loader loaded the OS image from. If the OS image was not loaded from a bios disk, then this field must not be present (bit 3 must be clear). The operating system may use this field as a hint for determining its own root device, but is not required to. The ‘boot\_device’ field is laid out in four one-byte subfields as follows:

+	-	-	-	-	-	+					
	part3		part2		part1		drive				
+	-	-	-	-	-	+	-	-	+	-	+

The first byte contains the bios drive number as understood by the bios INT 0x13 low-level disk interface: e.g. 0x00 for the first floppy disk or 0x80 for the first hard disk.

The three remaining bytes specify the boot partition. ‘part1’ specifies the top-level partition number, ‘part2’ specifies a sub-partition in the top-level partition, etc. Partition numbers always start from zero. Unused partition bytes must be set to 0xFF. For example, if the disk is partitioned using a simple one-level DOS partitioning scheme, then ‘part1’ contains the DOS partition number, and ‘part2’ and ‘part3’ are both 0xFF. As another example, if a disk is partitioned first into DOS partitions, and then one of those DOS partitions is subdivided into several BSD partitions using BSD’s disklabel strategy, then ‘part1’ contains the DOS partition number, ‘part2’ contains the BSD sub-partition within that DOS partition, and ‘part3’ is 0xFF.

DOS extended partitions are indicated as partition numbers starting from 4 and increasing, rather than as nested sub-partitions, even though the underlying disk layout of extended partitions is hierarchical in nature. For example, if the boot loader boots from the second extended partition on a disk partitioned in conventional DOS style, then ‘part1’ will be 5, and ‘part2’ and ‘part3’ will both be 0xFF.

If bit 2 of the ‘flags’ longword is set, the ‘cmdline’ field is valid, and contains the physical address of the command line to be passed to the kernel. The command line is a normal C-style zero-terminated string.

If bit 3 of the ‘flags’ is set, then the ‘mods’ fields indicate to the kernel what boot modules were loaded along with the kernel image, and where they can be found. ‘mods\_count’ contains the number of modules loaded; ‘mods\_addr’ contains the physical address of the first module structure. ‘mods\_count’ may be zero, indicating no boot modules were loaded, even if bit 1 of ‘flags’ is set. Each module structure is formatted as follows:

0		mod_start				
4		mod_end				
8		string				
12		reserved (0)				
	+	-	-	-	-	+

The first two fields contain the start and end addresses of the boot module itself. The ‘string’ field provides an arbitrary string to be associated with that particular boot module; it is a zero-terminated ASCII string, just like the kernel command line. The ‘string’ field may be 0 if there is no string associated with the module. Typically the string might be a command line (e.g. if the operating system treats boot modules as executable programs), or a pathname (e.g. if the operating system treats boot modules as files in a file system), but its exact use is specific to the operating system. The ‘reserved’ field must be set to 0 by the boot loader and ignored by the operating system.

### **Caution: Bits 4 & 5 are mutually exclusive.**

If bit 4 in the ‘flags’ word is set, then the following fields in the Multiboot information structure starting at byte 28 are valid:

	+-----+
28	tabsz
32	strsz
36	addr
40	reserved (0)
	+-----+

These indicate where the symbol table from an a.out kernel image can be found. ‘addr’ is the physical address of the size (4-byte unsigned long) of an array of a.out format nlist structures, followed immediately by the array itself, then the size (4-byte unsigned long) of a set of zero-terminated ascii strings (plus sizeof(unsigned long) in this case), and finally the set of strings itself. ‘tabsz’ is equal to its size parameter (found at the beginning of the symbol section), and ‘strsz’ is equal to its size parameter (found at the beginning of the string section) of the following string table to which the symbol table refers. Note that ‘tabsz’ may be 0, indicating no symbols, even if bit 4 in the ‘flags’ word is set.

If bit 5 in the ‘flags’ word is set, then the following fields in the Multiboot information structure starting at byte 28 are valid:

	+-----+
28	num
32	size
36	addr
40	shndx
	+-----+

These indicate where the section header table from an ELF kernel is, the size of each entry, number of entries, and the string table used as the index of names. They correspond to the ‘shdr\_\*’ entries (‘shdr\_num’, etc.) in the Executable and Linkable Format (elf) specification in the program header. All sections are loaded, and the physical address fields of the elf section header then refer to where the sections are in

memory (refer to the i386 elf documentation for details as to how to read the section header(s)). Note that ‘shdr\_num’ may be 0, indicating no symbols, even if bit 5 in the ‘flags’ word is set.

If bit 6 in the ‘flags’ word is set, then the ‘mmap\_\*’ fields are valid, and indicate the address and length of a buffer containing a memory map of the machine provided by the bios. ‘mmap\_addr’ is the address, and ‘mmap\_length’ is the total size of the buffer. The buffer consists of one or more of the following size/structure pairs (‘size’ is really used for skipping to the next pair):

-4	size	
	+-----+	
0	base_addr	
8	length	
16	type	
	+-----+	

where ‘size’ is the size of the associated structure in bytes, which can be greater than the minimum of 20 bytes. ‘base\_addr’ is the starting address. ‘length’ is the size of the memory region in bytes. ‘type’ is the variety of address range represented, where a value of 1 indicates available ram, and all other values currently indicated a reserved area.

The map provided is guaranteed to list all standard ram that should be available for normal use.

If bit 7 in the ‘flags’ is set, then the ‘drives\_\*’ fields are valid, and indicate the address of the physical address of the first drive structure and the size of drive structures. ‘drives\_addr’ is the address, and ‘drives\_length’ is the total size of drive structures. Note that ‘drives\_length’ may be zero. Each drive structure is formatted as follows:

0	size	
	+-----+	
4	drive_number	
	+-----+	
5	drive_mode	
	+-----+	
6	drive_cylinders	
	+-----+	
8	drive_heads	
	+-----+	
9	drive_sectors	
	+-----+	
10 - xx	drive_ports	
	+-----+	

The ‘size’ field specifies the size of this structure. The size varies, depending on the number of ports. Note that the size may not be equal to  $(10 + 2 * \text{the number of ports})$ , because of an alignment.

The ‘drive\_number’ field contains the BIOS drive number. The ‘drive\_mode’ field represents the access mode used by the boot loader. Currently, the following modes are defined:

‘0’ CHS mode (traditional cylinder/head/sector addressing mode).

‘1’ LBA mode (Logical Block Addressing mode).

The three fields, ‘drive\_cylinders’, ‘drive\_heads’ and ‘drive\_sectors’, indicate the geometry of the drive detected by the bios. ‘drive\_cylinders’ contains the number of the cylinders. ‘drive\_heads’ contains the number of the heads. ‘drive\_sectors’ contains the number of the sectors per track.

The ‘drive\_ports’ field contains the array of the I/O ports used for the drive in the bios code. The array consists of zero or more unsigned two-bytes integers, and is terminated with zero. Note that the array may contain any number of I/O ports that are not related to the drive actually (such as dma controller’s ports).

If bit 8 in the ‘flags’ is set, then the ‘config\_table’ field is valid, and indicates the address of the rom configuration table returned by the GET CONFIGURATION bios call. If the bios call fails, then the size of the table must be zero.

If bit 9 in the ‘flags’ is set, the ‘boot\_loader\_name’ field is valid, and contains the physical address of the name of a boot loader booting the kernel. The name is a normal C-style zero-terminated string.

If bit 10 in the ‘flags’ is set, the ‘apm\_table’ field is valid, and contains the physical address of an apm table defined as below:

## Examples

## History

## Index

Часть X

Технологии

## Часть XI

# Сетевое обучение

## Часть XII

# Базовая теоретическая подготовка

# Глава 20

## Математика

### 20.1 Высшая математика в упражнениях и задачах [61]

В этом разделе будут размещены решения некоторых задач из [61] в “техническом” стиле: главное быстрый результат, а не точное аналитическое решение, поэтому будем использовать системы компьютерной математики. Будут рассмотрены приемы применения OpenSource пакетов:

**Maxima** [19] символьная математика, аналог **MathCAD**, on-line <http://maxima.org/>

**Octave** [21] численная математика, аналог **MATLAB**, on-line <http://octave-online.net/>

**GNUPLOT** [?] простейшее средство построения 3D/3D графиков

**WolframAlpha** <http://www.wolframalpha.com/> бесплатная on-line система символьной математики и база знаний, функционал и интерфейс очень ограничены, но вполне полезна в качестве **символьного калькулятора**

**Python** скриптовый язык программирования, в последнее время получил широкое применение в области численных методов, анализа данных и автоматизации, чаще всего применяется в комплекте с библиотеками:

**NumPy** поддержка многомерных массивов (включая матрицы) и высокочувственных математических функций, предназначенных для работы с ними

**SciPy** библиотека предназначенная для выполнения научных и инженерных расчётов: поиск минимумов и максимумов функций, вычисление интегралов функций, поддержка специальных функций, обработка сигналов, обработка изображений, работа с генетическими алгоритмами, решение обыкновенных дифференциальных уравнений,...

**Sympy** библиотека символьной математики <https://en.wikipedia.org/wiki/Sympy>

**Matplotlib** библиотека на языке программирования Python для 2D/3D визуализации данных. Получаемые изображения могут быть использованы в качестве иллюстраций в публикациях.

Подробно с применением *Python* при обработке данных можно ознакомиться в <http://scipy-cookbook.readthedocs.org/>

Также этот раздел можно использовать как пример использования системы верстки L<sup>A</sup>T<sub>E</sub>X для научных публикаций —смотрите **исходные тексты** файла <https://github.com/ponyatov/boox/tree/master/math/danko/danko.tex>.

## Запуск **Maxima** и **Octave** в пакетном режиме

При запуске **Maxima**/**Octave** выводится информация о программе и license disclaim. При их использовании в автоматическом режиме<sup>1</sup> требуется блокировать лишний вывод опцией -q. Как пример можно привести набор правил для **make**:

```
% .pdf: %.plot
    gnuplot $<
%.pdf: %.mac
    maxima -q < $<
%.log: %.mac
    maxima -q < $< > $@
%.pdf: %.m Makefile
    octave -q $< && pdfcrop o$@ $@
%.log: %.m Makefile
    octave -q $< > $@
```

\$@ **левая** часть make-правила

\$< **первый элемент** правой части правила

&& выполнить следующую команду только если предыдущая вернула код успешного выполнения `exit(0)`

**pdfcrop** <in> <out> **octave** выводит графики в полный лист А4, **pdfcrop** выполняет обрезку

### 20.1.1 Аналитическая геометрия на плоскости

#### Прямоугольные и полярные координаты

**1. Координаты на прямой. Деление отрезка в данном отношении.** Точку  $M$  координатной оси  $Ox$ , имеющую **абсциссу**  $x$ , обозначают через  $M(x)$ .

Расстояние  $d$  между точками  $M_1(x_1)$  и  $M_2(x_2)$  оси при любом расположении точек на оси находятся по формуле:

$$d = |x_2 - x_1| \quad (20.1)$$

---

<sup>1</sup> например в файлах Makefile 13.13

Пусть на произвольной прямой задан отрезок  $AB$  ( $A$  — начало отрезка,  $B$  — конец), тогда всякая третья точка  $C$  этой прямой делить отрезок  $AB$  в некотором отношении  $\lambda$ , где  $\lambda = \frac{AC}{CB}$ . Если отрезки  $AC$  и  $CB$  направлены в одну сторону, то  $\lambda$  приписывают знак “плюс”; если же отрезки  $AC$  и  $CB$  направлены в противоположные стороны, то  $\lambda$  приписывают знак “минус”. Иными словами,  $\lambda > 0$  если точка  $C$  лежит между точками  $A$  и  $B$ ;  $\lambda < 0$  если точка  $C$  лежит вне отрезка  $AB$ .

Пусть точки  $A$  и  $B$  лежат на оси  $Ox$ , тогда **координата точки  $C(\bar{x})$** , делящей отрезок между точками  $A(x_1)$  и  $B(x_2)$  в отношении  $\lambda$ , находится по формуле:

$$\bar{x} = \frac{x_1 + \lambda x_2}{1 + \lambda} \quad (20.2)$$

В частности, при  $\lambda = 1$  получается формула для координаты середины отрезка:

$$\bar{x} = \frac{x_1 + x_2}{2} \quad (20.3)$$

Формула 20.2 легко выводится из системы

$$\begin{cases} |A(x_1)C(\bar{x})| = \bar{x} - x_1 = a > 0 \Leftrightarrow \bar{x} > x_1 \\ |C(\bar{x})B(x_2)| = x_2 - \bar{x} = b > 0 \Leftrightarrow x_2 > \bar{x} \\ |A(x_1)B(x_2)| = x_2 - x_1 = a + b; \\ \lambda = a/b; \end{cases}$$

### WolframAlpha

```
solve x-x1=a ; x2-x=b ; x2-x1=a+b ; lambda=a/b for x
Reduce[{ x-x1==a, x2-x==b, x2-x1==a+b, lambda==a/b },{x}]
```

- Построить на прямой точки  $A(3)$ ,  $B(-2)$ ,  $C(0)$ ,  $D(\sqrt{2})$ ,  $E(-3.5)$ .

### WolframAlpha

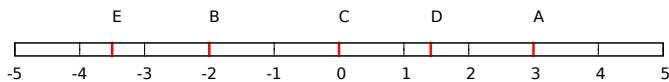


```
number line 3,-2,0,sqrt(2),-3.5 • 3 | • -2 | • 0 | • √2 | • -3.5
```

Листинг  
**GNUPLOT**

22:

```
set terminal pdf
set output 'g_1_1_1.pdf'
set size ratio .02
unset key
unset ytics
set xtics 1
set label "A" at 3,3
set label "B" at -2,3
set label "C" at 0,3
set label "D" at sqrt(2),3
set label "E" at -3.5,3
plot [-5:+5][0:1] '-' u 1:2 w i lw 5
3 1
-2 1
0 1
1.4142 1
-3.5 1
e
```



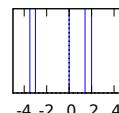
2

---

<sup>2</sup>  $\sqrt{2}$  пришлось указать численно, значение функции не подставилось

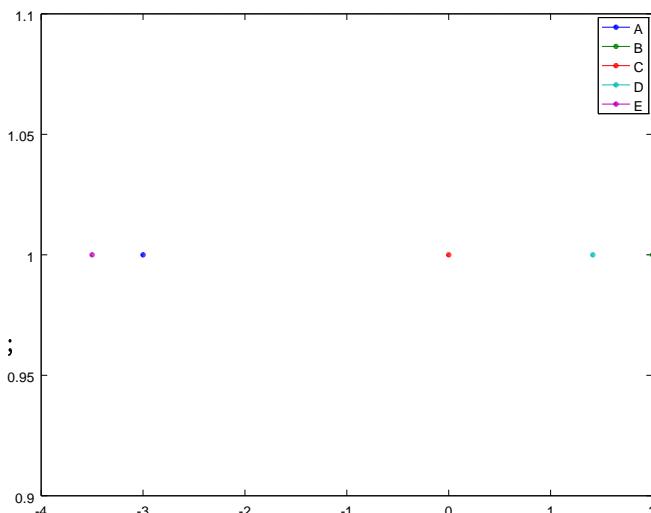
### Листинг 23: Maxima

```
A:-3;  
B:2;  
C:0;  
D:sqrt(2);  
E:-3.5;  
  
dat: [[A,1],[B,1],[C,1],[D,1],[E,1]];  
  
plot2d([discrete,dat],\  
 [x,-5,+5],[y,0,1],\  
 [style,impulses],[yticks,false],\  
 [xlabel,false],[ylabel,false],\  
 [gnuplot_term,"pdf size 5,1"],\  
 [gnuplot_out_file,"./m_1_1_1.pdf"]);
```



### Листинг 24: Octave

```
A=-3;  
B=2;  
C=0;  
D=sqrt(2);  
E=-3.5;  
  
plot(A,1,B,1,C,1,D,1,E,1)  
legend('A','B','C','D','E');  
print o_1_1_1.pdf
```



2. Отрезок  $AB$  четырьмя точками разделен на пять равных частей. Найти координату ближайшей к  $A$  точки деления, если  $A(-3)$ ,  $B(7)$ .

Пусть  $C(\bar{x})$  — искомая точка, тогда  $\lambda = \frac{AC}{CB} = \frac{1}{4}$ . Следовательно, по формуле 20.2 находим

$$C(\bar{x}) = \frac{x_1 + \lambda x_2}{1 + \lambda} = \frac{-3 + \frac{1}{4} \cdot 7}{1 + \frac{1}{4}} = C(-1)$$

## Maxima

```
m_1_1_2 (x1 ,x2 ,lambda) := (x1+lambda*x2)/(1+lambda);  
A : -3 ;  
B : 7 ;  
lambda : 1/4 ;  
  
C = m_1_1_2(A,B,lambda);
```

Определяем функцию `m(maxima)` <глава> <параграф> <задача> (по нумерации задач в [61]), и вычисляем функцию с подстановкой числовых значений.

```
(%i1)  
(%o1)      m_1_1_2(x1 , x2 , lambda) := 
$$\frac{x1 + \text{lambda} \cdot x2}{1 + \text{lambda}}$$
  
(%i2) (%o2)  
(%i3) (%o3)  
(%i4)      - 3  
(%o4)      7  
(%i5) (%o5)      1  
(%i6)      -  
                         4  
                                         C = - 1
```

В **Octave** **файлы с расширением .m** могут содержать не только последовательность команд, но и **выполнять роль определения библиотечной функции**. В этом случае имя функции должно совпадать с именем файла, где прописано ее определение.

### Листинг 25: шаблон определения функции

```
function [<результат_1>, <результат_2>, ...] = <имя> [<параметр_1>, <параметр_2> ...]  
    оператор_1;  
    ...  
    оператор_N;  
end;
```

### Octave – o\_1\_1\_2.m

```
function [xn] = o_1_1_2 (x1 ,x2 ,lambda)  
    xn = (x1+lambda*x2)/(1+lambda);  
end  
A = -3  
B = 7  
lambda = 1/4  
  
o_1_1_2(A,B,lambda)
```

Определяем функцию `o(ctave)_<глава>_<параграф>_<задача>` (по нумерации задач в [61]), и вычисляем функцию с подстановкой числовых значений.

```
A = -3
B = 7
lambda = 0.25000
ans = -1
```

3. Известны точки  $A(1)$ ,  $B(5)$  — концы отрезка  $AB$ ; вне этого отрезка расположена точка  $C$ , причем ее расстояние от точки  $A$  в 3 раза больше расстояния от точки  $B$ . Найти координату точки  $C$ .

Нетрудно установить что  $\lambda = -\frac{AC}{BC} = -3$ , таким образом

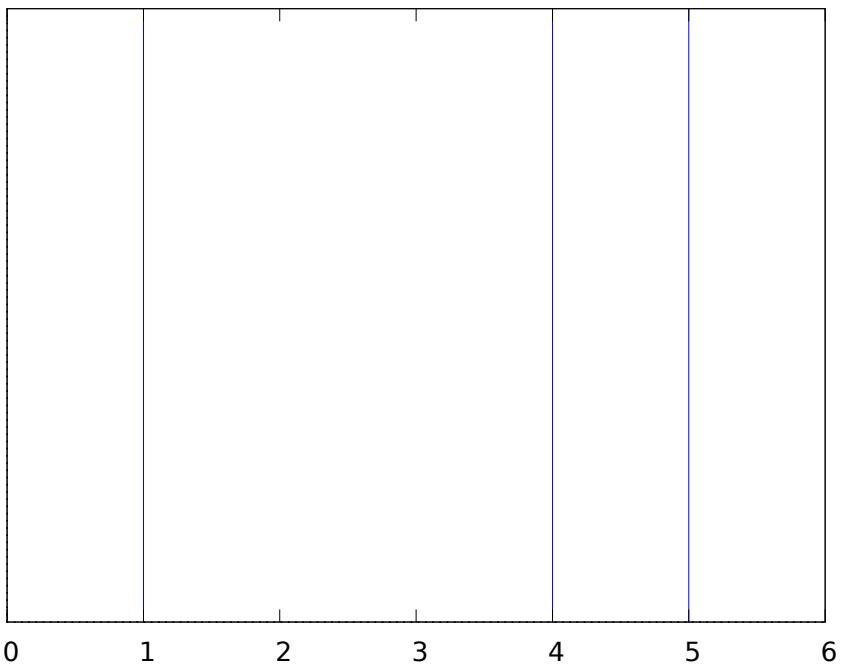
$$C(\bar{x}) = \frac{1 - 3 \cdot 5}{1 - 3} = C(7) \quad (20.4)$$

### Maxima

```
A:1;
B:5;
lambda:3;
C:(A+lambda*B)/(A+lambda);

dat: [[A,1],[B,1],[C,1]];

plot2d ([ discrete ,dat ], \
[x,A-1,B+1],[y,0,1], \
[ style , impulses ] , \
[ xlabel , false ] , [ ylabel , false ] , [ ytics , false ] , \
[ gnuplot _term , pdf ] , \
[ gnuplot _out _file , "./m_1_1_3.pdf "]);
```



4. Найти расстояние между точками

1.  $M(3) N(-5)$

*Python*

```
abs( (-5) - (3) )
8
```

2.  $P(-5.5) Q(-2.5)$

*Python*

```
def distance(a,b)
    return abs(a-b)
```

```
distance( -5.5 , -2.5 )
3.0
```

5. Найти координаты середины отрезка, если известны его концы<sup>3</sup>:

1.  $A(-6) B(7)$

2.  $C(-5) D(0.5)$

---

<sup>3</sup> используем формулу 20.3

```
% [danko3] equation:
function midpoint = danko3 (x1 , x2)
    midpoint = (x1+x2)/2;
end

danko3( -6 , 7 )
danko3( -5 , 0.5 )
```

**Листинг 26:**

```
ans = 0.50000
ans = -2.2500
```

**6.** Найти точку  $M$ , симметричную точке  $N(-3)$  относительно точки  $P(2)$ .

$$N(x_1)P(\bar{x}) = P(\bar{x})M(x_2)$$

Из 20.3:

$$2 = \frac{(-3) + x_2}{2}$$

**WolframAlpha** solve N=-3;P=2;P=(N+M)/2 for M  $\Rightarrow$  7

**Maxima**

```
N: -3;
P: 2;
solve (P=(N+M)/2 ,M);
```

**m\_1\_1\_6.log**

(%i1) (%o1)	- 3
(%i2) (%o2)	2
(%i3) (%o3)	[M = 7]
(%i4)	

**Часть XIII**

**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 издательство над читателем. Это типичный памятник автору, который был, мягко говоря, малоприятным человеком. Он излагает то, что излагает, абсолютно пре-небрегая своим читателем и даже издеваясь над ним. А у нас целые поколения выросли на этой книге, и считается, что всё нормально, кто справился, тот младец. 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 внятно, 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 21

## 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.

### 21.1 для вашего собственного скриптового языка

Через какое-то время практики FSP у вас выработается один диалект скриптов для всех программ, соответствующий 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:

<code>filetype.vim</code>	(g)Vim	привязка расширений файлов (.src .lo
<code>syntax.vim</code>	(g)Vim	синтаксическая подсветка для скрипт
<code>/.vimrc</code>	<i>Linux</i>	настройки для пользователя
<code>/vimrc</code>	<i>Windows</i>	
<code>/.vim/ftdetect/src.vim</code>	<i>Linux</i>	привязка команд к расширению .src
<code>/vimfiles/ftdetect/src.vim</code>	<i>Windows</i>	
<code>/.vim/syntax/src.vim</code>	<i>Linux</i>	синтаксис к расширению .src
<code>/vimfiles/syntax/src.vim</code>	<i>Windows</i>	

### Книги must have любому техническому специалисту

#### Математика, физика, химия

- Бермант **Математический анализ** [28]
- Тихонов, Самарский **Математическая физика** [37, 62]
- Демидович, Марон **Численные методы** [42, 43]
- Кремер **Теория вероятностей и матстатистика** [33]
- Ван дер Варден **Математическая статистика** [29]
- Кострикин **Введение в алгебру** [31, 32]

- Ван дер Варден **Алгебра** [30]

- Демидович **Сборник задач по математике для втузов. В 4 частях** [63, ?, ?, ?]
- Будак, Самарский, Тихонов **Сборник задач по математической физике** [62]

## Фейнмановские лекции по физике

1. Современная наука о природе. Законы механики. [48]
2. Пространство. Время. Движение. [49]
3. Излучение. Волны. Кванты. [50]
4. Кинетика. Теплота. Звук. [51]
5. Электричество и магнетизм [52]
6. Электродинамика. [53]
7. Физика сплошных сред. [54]
8. Квантовая механика 1. [55]
9. Квантовая механика 2. [56]

- Цирельсон **Квантовая химия** [58]
- Розенброк **Вычислительные методы для инженеров-химиков** [59]
- Шрайвер Эткинс **Неорганическая химия** [60]

## Обработка экспериментальных данных и метрология

- Смит **Цифровая обработка сигналов** [34]
- Князев, Черкасский **Начала обработки экспериментальных данных** [35]

## 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** [26]  
написание скриптов обработки данных, автоматизации, графических оболочек и т.п. утилит
- **JavaScript** [24] + **HTML**  
**генерация отчетов** и ввод исходных данных, интерфейс к сетевым расчетным серверам на *Python*, простые браузерные граф.интерфейсы и расчетки
- **Реляционные (и объектные) базы данных** /MySQL, Postgres (,ZODB,GC)  
хранение и простая черновая обработка табличных (объектных) данных экспериментов, справочников, настроек, пользователей.

- Язык  $C_+$ , утилиты GNU toolchain [22, 23] (gcc/g++, make, ld)  
базовый Си, ООП очень кратко<sup>1</sup>, без излишеств профессионального программирования<sup>2</sup>, 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.

## САПР, пакеты математики, моделирования, визуализации

- **Maxima** символьная математика [19]
- **Octave** численные методы [21]
- **GNUPLOT** простой вывод графиков
- **ParaView/VTK** навороченнейший пакет/библиотека визуализации всех видов
- **LATEX** верстка научных публикаций и генерация отчетов
- **KiCAD + ng-spice** электроника: расчет схем и проектирование печатных плат
- **FreeCAD** САПР общего назначения
- **Elmer, OpenFOAM** расчетные пакеты метода конечных элементов (мультифизика, сопротивление материалов, конструкционная устойчивость, газовые и жидкостные потоки, теплопроводность)
- **CodeAster + Salome** пакет МКЭ, особо заточенный под сопромат и расчет конструкций
- **OpenModelica** симуляция моделей со средоточенными параметрами<sup>3</sup> (электроника, электротехника, механика, гидропневмоавтоматика и системы управления)
- **V-REP** робототехнический симулятор
- **SimChemistry**<sup>4</sup> интересный демонстрационный симулятор химической кинетики молекул на микроуровне (обсчитывается движение и столкновение отдельных молекул)
- **Avogadro** 3D редактор молекул

<sup>1</sup> наследование, полиморфизм, операторы для пользовательских типов, использование библиотеки STL

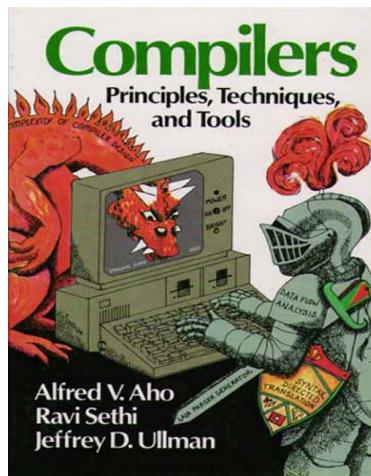
<sup>2</sup> мегабиблиотека Boost, написание своих библиотек шаблонов и т.п.

<sup>3</sup> для описания моделей элементов использует ООП-язык Modelica

<sup>4</sup> Windows

# Литература

## Разработка языков программирования и компиляторов



[1] **Dragon Book**

Компиляторы. Принципы, технологии, инструменты.

Альфред Ахо, Рави Сети, Джейфри 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

**Structure and  
Interpretation  
of Computer  
Programs**

Second Edition



Harold Abelson and  
Gerald Jay Sussman  
with Julie Sussman

**SICP**

[3]

**Структура и интерпретация компьютерных программ**

Харольд Абельсон, Джеральд Сассман

ISBN 5-98227-191-8

EN: [web.mit.edu/alexmv/6.037/sicp.pdf](http://web.mit.edu/alexmv/6.037/sicp.pdf)



[4]

**Функциональное программирование**

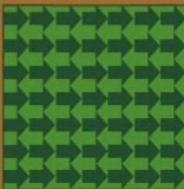
Филд А., Харрисон П.

М.: Мир, 1993

ISBN 5-03-001870-0

МАТЕМАТИЧЕСКОЕ  
ОБЕСПЕЧЕНИЕ  
ЭВМ

П.Хендерсон  
ФУНКЦИОНАЛЬНОЕ  
ПРОГРАММИРОВАНИЕ  
Применение  
и реализация



[5]

Функциональное программирование: применение и реализация

П.Хендерсон

М.: Мир, 1983



**LLVM:** инфраструктура  
для разработки компиляторов

Бруно Кардос Лопес

Рафаэль Аулер



[6]

LLVM. Инфраструктура для разра-

ботки 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

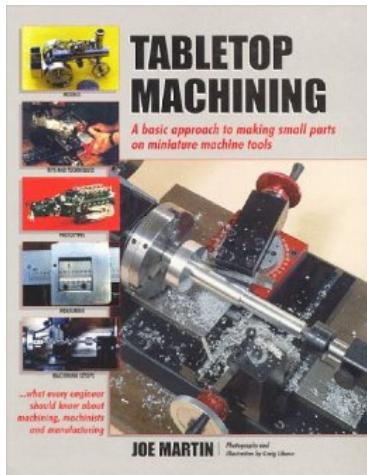
2 edition, May 2014

<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]

LATEX 2ε по-русски И. Котельников, П. Чеботаев

ISBN: 5-87550-195-2

[16] e-Readers and LATEX

Alan Wetmore

<https://www.tug.org/TUGboat/tb32-3/tb102wetmore.pdf>

[17] How to cite a standard (ISO, etc.) in BibLATEX ?  
<http://tex.stackexchange.com/questions/65637/>

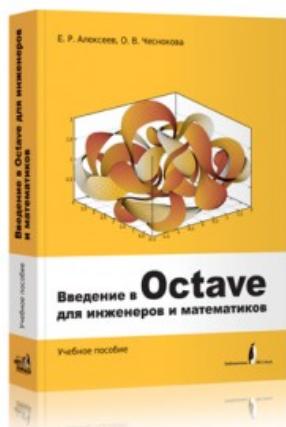
## Математическое ПО: Maxima, Octave, GNUPLOT,..

[18] Система аналитических вычислений Maxima для физиков-теоретиков  
Б.А. Ильина, П.К.Силаев  
<http://tex.bog.msu.ru/numtask/max07.ps>



[19] Компьютерная математика с Maxima  
Евгений Чичкарев

[20] Graphics with Maxima  
Wilhelm Haager



[21] Введение в Octave для инженеров и математиков

# САПР, электроника, проектирование печатных плат

## Программирование

### GNU Toolchain

- [22] **Embedded Systems Programming in C<sub>+</sub><sup>+</sup>**  
© <http://www.bogotobogo.com/>  
<http://www.bogotobogo.com/cplusplus/embeddedSystemsProgramming.php>
- [23] **Embedded Programming with the GNU Toolchain**  
Vijay Kumar B.  
<http://bravegnu.org/gnu-eprog/>

## JavaScript, HTML, CSS, Web-технологии:

- [24] **On-line пошаговый учебник JavaScript** на английском, поддерживает множество языков и ИТ-технологий, курс очень удобен и прост для совсем начинающих <https://www.codecademy.com>
- [25] On-line учебник *JavaScript* на русском <http://learn.javascript.ru/>

## Python

- [26] **Язык программирования Python**  
Россум, Г., Дрейк, Ф.Л.Дж., Откидач, Д.С., Задка, М., Левис, М., Монтаро, С., Реймонд, Э.С., Кучлинг, А.М., Лембург, М.-А., Йи, К.-П., Ксиллаг, Д., Петрилли, Х.Г., Варсав, Б.А., Ахлстром, Дж.К., Роскинд, Дж., Шеменор, Н., Муландер, 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

- [27] OSDev Wiki  
<http://wiki.osdev.org>

## Базовые науки

### Математика

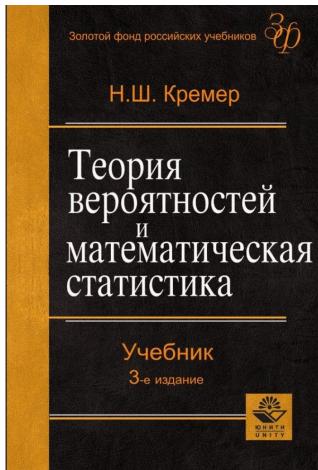


- [28] Краткий курс математического анализа для ВТУЗов  
Бермант А.Ф., Араманович И.Г.  
М.: Наука, 1967  
<https://drive.google.com/file/d/0B0u4WeMj0894U1Y1dEJ6cnCxU28/view?usp=sharing>

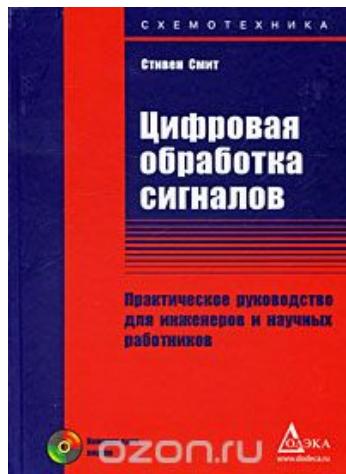
Пятое издание известного учебника, охватывает большинство вопросов программы по высшей математике для инженерно-технических специальностей вузов, в том числе дифференциальное исчисление функций одной переменной

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”.**

- [29] **Математическая статистика** Б.Л. Ван дер Варден
- [30] **Алгебра** Б.Л. Ван дер Варден
- [31] **Введение в алгебру. В 3 частях. Часть 1. Основы алгебры** А.И. Ко стрикин
- [32] **Введение в алгебру. В 3 частях. Линейная алгебра. Часть 2** А.И. Ко стрикин



- [33] **Теория вероятностей и математическая статистика**  
Наум Кремер  
М.: Юнити, 2010



[34]

## 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

[35] 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](http://www.phys.nsu.ru/cherk/Metodizm_old.PDF)

Учебное пособие предназначено для студентов естественно-научных специальностей, выполняющих 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. Пособие дополнено обучающей программой, которая позволяет как углубить и уточнить знания, полученные в методическом пособии, так и проводить собственно обработку результатов лабораторных работ. Приведен список литературы для желаю-

щих углубить свои знания в области математической статистики и обработки данных.



ozon.ru

[36]

Принципы современной математической физики Р. Рихтмайер

[37] Уравнения математической физики А.Н. Тихонов, А.А. Самарский

## Символьная алгебра

[38] Компьютерная алгебра

Панкратьев Евгений Васильевич

МГУ, 2007

Настоящее пособие составлено на основе спецкурсов, читавшихся автором на механико-математическом факультете в течение более 10 лет. Выбор материала в значительной мере определялся пристрастиями автора. Наряду с классическими результатами компьютерной алгебры в этих спецкурсах (и в настоящем пособии) нашли отражение исследования нашего коллектива. Прежде всего, это относится к теории дифференциальной размерности.

Е. В. ПАНКРАТЬЕВ

ЭЛЕМЕНТЫ  
КОМПЬЮТЕРНОЙ  
АЛГЕБРЫ



ozon.ru

[39]

**Элементы компьютерной алгебры**

Евгений Панкратьев

Год выпуска 2007

ISBN 978-5-94774-655-6, 978-5-9556-0099-4

Учебник посвящен описанию основных структур данных и алгоритмов, применяемых в символьных вычислениях на ЭВМ. В книге затрагивается широкий круг вопросов, связанных с вычислениями в кольцах целых чисел, многочленов и дифференциальных многочленов.



[40]

**Элементы абстрактной и компьютерной алгебры**

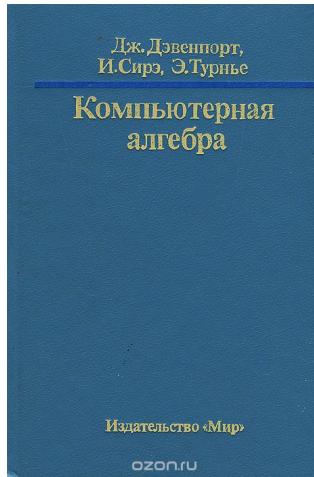
Дмитрий Матрос, Галина Поднебесова

2004

ISBN 5-7695-1601-1

В книгу включены следующие главы: алгебры, введение в системы компьютерной алгебры, кольцо целых чисел, полиномы от одной переменной, полиномы от нескольких переменных, формальное интегрирование, кодирование. Разбор доказательств утверждений и выполнение упражнений, приведенных

в учебном пособии, позволяют студентам овладеть методами решения практических задач, навыками конструирования алгоритмов.



- [41] **Компьютерная алгебра**

Дж.Дэвенпорт, И.Сирэ, Э.Турнье

Книга французских специалистов, охватывающая различные вопросы компьютерной алгебры: проблему представления данных, полиномиальное упрощение, современные алгоритмы вычисления НОД полиномов и разложения полиномов на множители, формальное интегрирование, применение систем компьютерной алгебры. Первый автор знаком читателю по переводу его книги "Интегрирование алгебраических функций"

(М.: Мир, 1985).

## Численные методы

- [42] **Основы вычислительной математики**

Борис Демидович, Исаак Марон

Книга посвящена изложению важнейших методов и приемов вычислительной математики на базе общего вузовского курса высшей математики. Основная часть книги является учебным пособием по курсу приближенных вычислений для вузов.

- [43] **Численные методы анализа. Приближение функций, дифференциальные и интегральные уравнения**

Б. П. Демидович, И. А. Марон, Э. З. Шувалова

В книге излагаются избранные вопросы вычислительной математики, и по содержанию она является продолжением учебного пособия [42]. Настоящее, третье издание отличается от предыдущего более доходчивым изложением. Добавлены новые примеры.

# Теория игр

## [44] Теория игр

Петросян Л. А. Зенкевич Н.А., Семина Е.А.

Учеб. пособие для ун-тов. — М.: Высш. шк., Книжный дом «Университет», 1998.

ISBN 5-06-001005-8, 5-8013-0007-4.

## [45] Математическая теория игр и приложения

Мазалов В.В.

Санкт-Петербург - Москва - Краснодар: Лань, 2010.

ISBN 978-5-8114-1025-5.

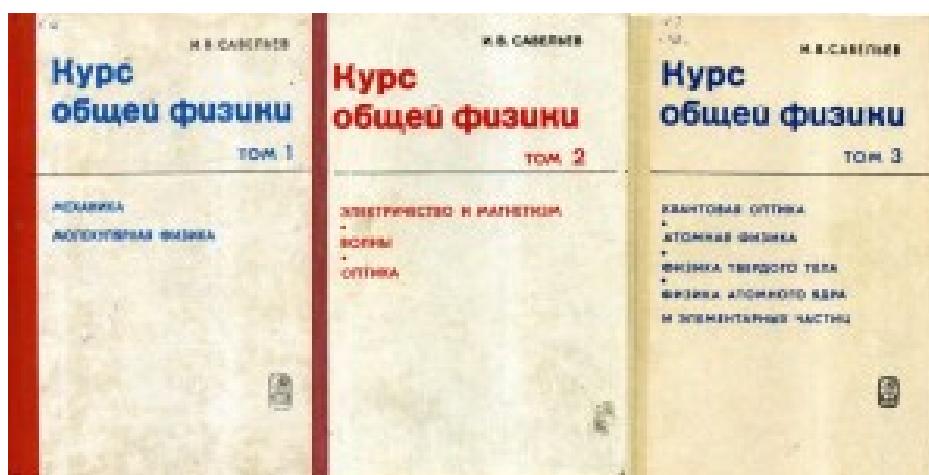
## [46] Теория игр

Оуэн Г.

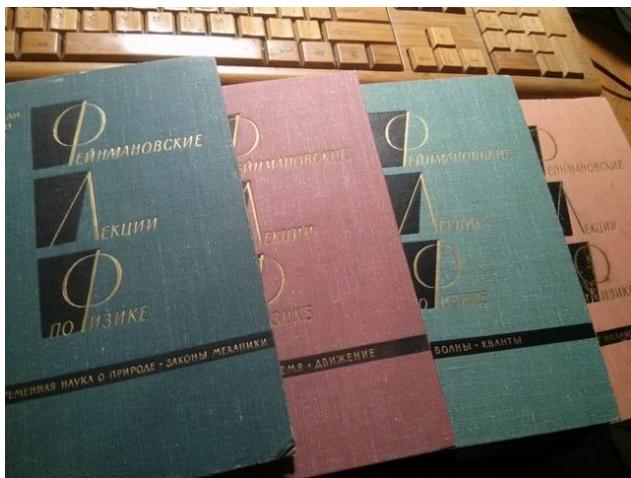
Книга представляет собой краткое и сравнительно элементарное учебное пособие, пригодное как для первоначального, так и для углубленного изучения теории игр. Для ее чтения достаточно знания элементов математического анализа и теории вероятностей.

Книга естественно делится на две части, первая из которых посвящена играм двух лиц, а вторая — играм  $N$  лиц. Она охватывает большинство направлений теории игр, включая наиболее современные. В частности, рассмотрены антагонистические игры, игры двух лиц с ненулевой суммой и основы классической кооперативной теории. Часть материала в монографическом изложении появляется впервые. Каждая глава снабжена задачами разной степени сложности.

# Физика



Савельев И.В.



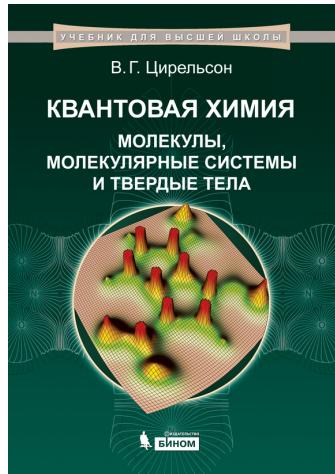
Фейнмановские лекции

## по физике

Ричард Фейнман, Роберт Лейтон, Мэттью Сэндс

- [48] Современная наука о природе. Законы механики.
- [49] Пространство. Время. Движение.
- [50] Излучение. Волны. Кванты.
- [51] Кинетика. Теплота. Звук.
- [52] Электричество и магнетизм.
- [53] Электродинамика.
- [54] Физика сплошных сред.
- [55] Квантовая механика 1.
- [56] Квантовая механика 2.
- [57] Основы квантовой механики Д.И. Блохинцев

# Химия



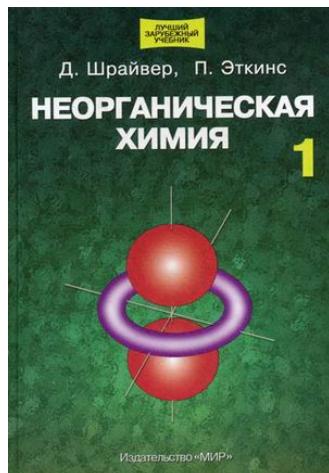
[58]

Квантовая химия. Молекулы, молекулярные системы и твердые тела. Учебное пособие Владимир Цирельсон



[59]

Вычислительные методы для инженеров-химиков X. Розенброк, С. Стори



[60]

**Неорганическая химия** В 2 томах  
Д. Шрайвер, П. Эткинс

## Задачники

### Математика



[61]

**Высшая математика в упражнениях и задачах**  
П.Е. Данко, А.Г.Попов, Т.Я. Кожевникова, С.П. Данко

[62] **Сборник задач по математической физике** Будак Б.М., Самарский А.А., Тихонов А.Н.

[63] **Сборник задач по математике для втузов. В 4 частях. Часть 1. Линейная алгебра и основы математического анализа**  
Демидович

## Стандарты и ГОСТы

- [64] 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](http://rtu.samgtu.ru/sites/rtu.samgtu.ru/files/GOST_ESKD_2.701-2008.pdf)

# Предметный указатель

- „ 120
- ::, 85
- абсцисса, 194
- адрес хранения, 141
- адрес размещения, 141
- анонимная переменная, 20
- базовый адрес, 122
- бинарный формат, 122
- биндинг логической переменной, 20
- цель (Пролог), 26
- дерево вывода, 28, 38
- дерево заключений, 24
- факт, 23, 26
- граф смежности, 21
- грамматика, 86
- инкрементная компоновка, 130
- канадский крест, 163
- компоновка, 130
- конъюнктивная цель, 20
- консеквенция, 24
- координата точки, 195
- линкер, 121
- линовка, 122
- логическая переменная, 20
- монитор **Qemu**, 124
- назначение адресов, 122
- низкоуровневое программирование, 117
- объектный код, 120
- оператор, 79
- переменная цели, 26
- правило, 26
- привязка логической переменной, 20
- разрешение символов, 131
- релокация символов, 132
- секционирование, 133
- семантическое дерево, 21
- символ, 79, 128
- символьный тип, 78
- синтаксическое дерево, 86
- скрипт линкера, 137
- состояние лексера, 88
- спецификация MultiBoot, 173, 175
- строчный комментарий, 87
- таблица символов, 128
- токен, 86
- трассировка, 28
- указатель адреса размещения, 137
- унификация, 31, 38
- вывод *Prolog*-программы, 24
- заголовок правила, 26
- загрузчик, 173
- ABI, 139
- application term, 55
- backtracking, 39
- bare metal, 117
- closure, 56
- constant, 70
- ELF, 122
- functor, 70
- functor arity, 70
- lambda term, 55
- LMA, 141
- make-правило, 154
- program term, 70

query term, 70

standalone, 117

startup код, 141

structure, 70

subterm, 70

variable, 70

VMA, 141