

Оглавление

Об этом сборнике	12
I ML & функциональное программирование	13
1 Основы Standard ML © Michael P. Fourman	14
1.1 Введение	14
2 Programming in Standard ML'97	16
An On-line Tutorial © Stephen Gilmore	16
II Язык Prolog: логическое программирование и искусственный интеллект	17
3 Adventure in Prolog	18
Preface	18
Prolog tools	19
3.1 Getting Started	20
3.1.1 Jumping In	23
3.1.2 Logic Programming	25
3.1.3 Jargon	26
3.2 Facts	27
3.2.1 Nani Search	30
3.2.2 Exercises	32
3.2.3 Genealogical Logicbase	32
3.2.4 Customer Order Entry	33
3.3 Simple Queries	34
3.3.1 How Queries Work	37
3.3.2 Exercises	40
3.4 Compound Queries	42
3.4.1 Built-in Predicates	46
3.4.2 Exercises	50
3.5 Rules	51

3.5.1	How Rules Work	53
3.5.2	Using Rules	56
3.5.3	Exercises	60
3.6	Arithmetic	63
3.6.1	Exercises	65
3.7	Managing Data	65
3.7.1	Exercises	69
Appendix	70
4	[LPN] Learn Prolog Now!	71
4.1	Факты, правила и запросы	71
4.1.1	1.1 Some Simple Examples	71
4.1.2	1.2 Prolog Syntax	76
4.1.3	1.3 Exercises	76
4.1.4	1.4 Practical Session	76
4.2	Chapter 2 Unification and Proof Search	76
4.3	Chapter 3 Recursion	76
4.4	Chapter 4 Lists	76
4.5	Chapter 5 Arithmetic	76
4.6	Chapter 6 More Lists	76
4.7	Chapter 7 Definite Clause Grammars	76
4.8	Chapter 8 More Definite Clause Grammars	76
4.9	Chapter 9 A Closer Look at Terms	76
4.10	Chapter 10 Cuts and Negation	76
4.11	Chapter 11 Database Manipulation and Collecting Solutions	76
4.12	Chapter 12 Working With Files	76
5	Учебник Фишера	77
	Введение	77
5.1	Установка и запуск <i>Prolog</i> -системы	79
5.2	Разбор примеров программ	83
5.2.1	Раскраска карт	83
5.2.2	Два определения факториала	87
5.2.3	Классическая задача “Ханойские башни”	91
5.2.4	Загрузка, редактирование, хранение программ	94
5.2.5	2.5 Negation as failure	96
5.2.6	2.6 Tree data and relations	96
5.2.7	2.7 Prolog lists and sequences	97
5.2.8	2.8 Change for a dollar	97
5.2.9	2.9 Map coloring redux	97
5.2.10	2.10 Simple I/O	97
5.2.11	2.11 Chess queens challenge puzzle	97
5.2.12	2.12 Finding all answers	97
5.2.13	2.13 Truth table maker	97

5.2.14	2.14 DFA parser	97
5.2.15	2.15 Graph structures and paths	98
5.2.16	2.16 Search	98
5.2.17	2.17 Animal identification game	98
5.2.18	2.18 Clauses as data	98
5.2.19	2.19 Actions and plans	98
5.3	Как работает <i>Prolog</i>	98
5.3.1	Деривационные деревья, выборы и унификация	98
	Унификация термов <i>Prologa</i>	101
5.3.2	3.2 Cut	103
5.3.3	3.3 Meta-interpreters in <i>Prolog</i>	108
5.4	4. Built-in Goals	108
5.4.1	4.1 Utility goals	108
5.4.2	4.2 Universals (true and fail)	108
5.4.3	4.3 Loading <i>Prolog</i> programs	108
5.4.4	4.4 Arithmetic goals	108
5.4.5	4.5 Testing types	108
5.4.6	4.6 Equality of <i>Prolog</i> terms, unification	108
5.4.7	4.7 Control	108
5.4.8	4.8 Testing for variables	108
5.4.9	4.9 Assert and retract	108
5.4.10	4.10 Binding a variable to a numerical value	108
5.4.11	4.11 Procedural negation, negation as failure	108
5.4.12	4.12 Input/output	108
5.4.13	4.13 <i>Prolog</i> terms and clauses as data	108
5.4.14	4.14 <i>Prolog</i> operators	108
5.4.15	4.15 Finding all answers	108
5.5	5. Search in <i>Prolog</i>	108
5.5.1	5.1 The A* algorithm in <i>Prolog</i>	108
5.5.2	5.2 The 8-puzzle	108
5.5.3	5.3 $\alpha\beta$ search in <i>Prolog</i>	108
5.6	6. Logic Topics	108
5.6.1	6.1 Chapter 6 notes	108
5.6.2	6.2 Positive logic	108
5.6.3	6.3 Convert first-order logic to normal form	108
5.6.4	6.4 A normal rulebase goal interpreter	108
5.6.5	6.5 Evidentiary soundness and completeness	108
5.6.6	6.6 Rule tree visualization using Java	108
5.7	7. Introduction to Natural Language Processing	108
5.7.1	7.1 <i>Prolog</i> grammar parser generator	108
5.7.2	7.2 <i>Prolog</i> grammar for simple English phrase structures	108
5.7.3	7.3 Idiomatic natural language command and question interfaces	108
5.8	8. Prototyping with <i>Prolog</i>	108

5.8.1	8.1 Action specification for a simple calculator	108
5.8.2	8.2 Animating the 8-puzzle (\$5.2) using character graphics	108
5.8.3	8.3 Animating the blocks mover (\$2.19) using character graphics	108
5.8.4	8.4 Java Tic-Tac-Toe GUI plays against <i>Prolog</i> opponent (\$5.3)	108
5.8.5	8.5 Structure diagrams and <i>Prolog</i>	108
References		108
6	ASTLOG: Язык для анализа синтаксических деревьев	109
Abstract		109
6.1	Introduction	110
6.1.1	The awk Approach	110
6.1.2	The Logic Programming Approach	112
6.2	Elements of ASTLOG	113
Figure 1: Complete Syntax of ASTLOG		113
6.2.1	Objects	114
6.2.2	The Current Object	114
Figure 2: Some core ASTLOG primitives		116
Figure 3: Some primitive node and symbol predicates		117
6.2.3	Examples	117
Figure 4: Actual ASTLOG code for follow_stmt		120
Figure 5: Definition of flatten		121
Figure 6: Parameterized version, flatten2		121
6.3	Higher order features	122
6.3.1	3.1 Lambdas and Applications	122
Figure 7: Parameterized version of sametree		123
Figure 8: Embedded Query State Primitives		124
6.3.2	Queries as Objects	124
Figure 9: Query Accumulators qcount and qlist		125
6.4	Implementation	126
6.5	Conclusions and Future Work	127
6.6	Acknowledgements	129
References		129
Appendix		130
Figure 10: Outline of astlog Operational Semantics		131
7	Warren's Abstract Machine	132
Абстрактная машина Варрена		
Предисловие к репринтному изданию		132
Предисловие		133
Реализация машины вывода на C_+^+		134
7.0.1	Makefile	134
7.0.2	hpp.hpp	135
7.0.3	cpp.cpp	135
7.0.4	upp.hpp/lpp.hpp: синтаксический анализатор	136

7.1	Введение	136
7.1.1	Существующая литература	136
7.1.2	Этот учебник	137
7.2	Унификация — ясно и просто	139
7.2.1	Представление термов	140
7.2.2	Компиляция \mathcal{L}_0 запросов	142
7.2.3	Compiling L_0 programs	144
7.2.4	Argument registers	147
7.3	Flat Resolution	149
7.3.1	Facts	150
7.3.2	Rules and queries	151
7.4	Prolog	153
7.4.1	Environment protection	154
7.4.2	What's in a choice point	156
7.5	Optimizing the Design	161
7.5.1	Heap representation	161
7.5.2	Constants, lists, and anonymous variables	162
7.5.3	A note on <code>set</code> instructions	165
7.5.4	5.4 Register allocation	54
7.5.5	5.5 Last call optimization	56
7.5.6	5.6 Chain rules	57
7.5.7	5.7 Environment trimming	58
7.5.8	5.8 Stack variables	60
7.5.9	5.9 Variable classification revisited	69
7.5.10	5.10 Indexing	75
7.5.11	5.11 Cut	83
7.6	6 Conclusion 89	166
7.7	A Prolog in a Nutshell 91	166
7.8	B The WAM at a glance 97	166
7.8.1	B.1 WAM instructions	97
7.8.2	B.2 WAM ancillary operations	112
7.8.3	B.3 WAM memory layout and registers	117
8	An Efficient Unification Martelli/Montanary Algorithm	167
	Abstract	167
8.1	INTRODUCTION	168
8.2	UNIFICATION AS THE SOLUTION OF A SET OF EQUATIONS: A NONDETERMINISTIC ALGORITHM	169
8.3	AN ALGORITHM WHICH EXPLOITS A PARTIAL ORDERING AMONG SETS OF VARIABLES	173
8.3.1	Basic Definitions	173
8.3.2	Transformations of Sets of Multiequations	175
8.3.3	Solving Systems of Multiequations	176

8.3.4	THE UNIFICATION ALGORITHM	180
8.4	EFFICIENT MULTIEQUATION SELECTION	182
8.5	IMPROVING THE UNIFICATION ALGORITHM FOR NONUNIFYING DATA	183
8.6	IMPLEMENTATION	185
8.7	COMPARISONS WITH OTHER ALGORITHMS	187
8.8	CONCLUSION	192
	REFERENCES	192
9	Parsing and Compiling Using Prolog	195
	Abstract	195
9.1	INTRODUCTION	196
9.2	PARSING	197
9.2.1	Bottom-Up	198
9.2.2	Top-Down	200
9.2.3	Recursive Descent	202
9.3	SYNTAX-DIRECTED TRANSLATION	205
9.4	M-GRAMMARS AND DCGs	206
9.5	GRAMMAR PROPERTIES	211
9.6	LEXICAL SCANNERS AND PARSER GENERATION	215
9.7	CODE GENERATION	217
9.7.1	Generating Code from Polish	217
9.7.2	Generating Code from Trees	221
9.7.3	A Machine-Independent Algorithm for Code Generation	223
9.7.4	Code Generation from a Labelled Tree	226
9.8	OPTIMIZATIONS	228
9.8.1	Compile-Time Evaluation	228
9.8.2	Peephole Optimization	229
9.9	USING PROPOSED EXTENSIONS	230
9.10	FINAL REMARKS	234
	ACKNOWLEDGMENTS	236
	REFERENCES	236
10	Using Definite Clause Grammars in SWI-Prolog	239
	Introduction	239
	Who This Course Is For	239
	Getting The Most From This Course	240
	Other resources	240
	Getting Stuck	240
10.1	1 Definite Clause Grammars	240
10.1.1	1 DCG rules	241
10.1.2	2 More DCG Syntax	243
10.1.3	3 Capturing Input	243
10.1.4	4 Variables in Body	243

10.2	2 Relating Trees To Lists	243
10.3	3 Left Recursion	243
10.4	4 Right-hand Context Notation	243
10.5	5 Implicitly Passing States Around	243
10.6	6 Parsing From Files	243
10.7	7 Implementation	243
10.8	8 Error Handling	243
10.8.1	1 Resynching The Parser	243
10.8.2	2 Printing Line Numbers	243
10.9	9 A Few Practical Hints	243
10.9.1	1 basics.pl	243
10.9.2	2 Lexical Issues	243
10.9.3	3 Regular Expressions	243
	Conclusion	243

III Дурдом на дереве 245

11	The Tree Processing Language	
	Defining the structure and behaviour of a tree	246
	Abstract	246
	Preface	247
11.1	Introduction	247
11.1.1	Compiler Construction and Abstract Syntax Trees	248
11.1.2	Problem Statement	249
11.1.3	Outline	249

IV Язык *bI* 250

12	DLR: Dynamic Language Runtime	251
----	-------------------------------	------------

13	Система динамических типов	254
13.1	sym : символ = Абстрактный Символьный Тип /AST/	254
13.2	Скаляры	256
13.2.1	str : строка	256
13.2.2	int : целое число	256
13.2.3	hex : машинное hex	256
13.2.4	bin : бинарная строка	256
13.2.5	num : число с плавающей точкой	256
13.3	Композиты	256
13.3.1	list : плоский список	256
13.3.2	cons : cons-пара и списки в <i>Lisp</i> -стиле	256
13.4	Функционалы	256

13.4.1 ор: оператор	256
13.4.2 fn: встроенная/скомпилированная функция	256
13.4.3 lambda: лямбда	256
14 Программирование в свободном синтаксисе: FSP	257
14.1 Типичная структура проекта FSP: <i>lexical skeleton</i>	257
14.1.1 Настройки (g) Vim	258
14.1.2 Дополнительные файлы	258
14.1.3 Makefile	259
15 Синтаксический анализ текстовых данных	260
15.1 Универсальный Makefile	260
15.2 C₊⁺ интерфейс синтаксического анализатора	261
15.3 Минимальный парсер	261
15.4 Добавляем обработку комментариев	263
15.5 Разбор строк	264
15.6 Добавляем операторы	265
15.7 Обработка вложенных структур (скобок)	268
16 Синтаксический анализатор	270
16.1 lpp.lpp: лексер <i>/flex/</i>	270
16.2 ypp.ypp: парсер <i>/bison/</i>	272
V skelex: скелет программы в свободном синтаксисе	275
Структура проекта	276
Makefile	276
ypp.ypp: синтаксический парсер	277
lpp.lpp: лексер	279
hpp.hpp: хедеры	280
cpr.cpr: ядро интерпретатора	282
Тестирование интерпретатора	284
Комментарии	284
Скаляры и базовые композиты	284
Операторы	285
VI emLinux для встраиваемых систем	287
Структура встраиваемого микроЛinuxа	288
Процедура сборки	289
17 clock: коридорные электронные часы = контроллер умного дурдома	290
18 gambox: игровая приставка	291

19 Программирование встраиваемых систем с использованием GNU Toolchain [23]	293
19.1 Введение	293
19.2 Настройка тестового стенда	294
19.2.1 Qemu ARM	294
19.2.2 Инсталляция Qemu на <i>Debian GNU/Linux</i>	294
19.2.3 Установка кросс-компилятора GNU Toolchain для ARM	294
19.3 Hello ARM	295
19.3.1 Сборка бинарника	296
19.3.2 Выполнение в Qemu	299
19.3.3 Другие команды монитора	301
19.4 Директивы ассемблера	301
19.4.1 Суммирование массива	301
19.4.2 Вычисление длины строки	303
19.5 Использование ОЗУ (адресного пространства процессора)	304
19.6 Линкер	306
19.6.1 Разрешение символов	306
19.6.2 Релокация	308
19.7 Скрипт линкера	313
19.7.1 Пример скрипта линкера	314
19.7.2 Анализ объектного/исполняемого файла утилитой objdump	315
19.8 Данные в RAM, пример	316
19.8.1 RAM энергозависима (<i>volatile</i>)!	317
19.8.2 Спецификация адреса загрузки LMA	318
19.8.3 Копирование ‘.data’ в ОЗУ	319
19.9 Обработка аппаратных исключений	321
19.10 Стартап-код на Си	322
19.10.1 Стек	323
19.10.2 Глобальные переменные	325
19.10.3 Константные данные	325
19.10.4 Секция .eeprom (AVR8)	325
19.10.5 Стартовый код	325
19.11 Использование библиотеки Си	329
19.12 Inline-ассемблер	330
19.13 Использование ‘make’ для автоматизации компиляции	330
19.13.1 Выбор конкретной <i>цели</i>	331
19.13.2 Переменные	332
19.14 Contributing	332
19.15 Credits	332
19.15.1 14.1. People	332
19.15.2 14.2. Tools	332

19.16.15. Tutorial Copyright	332
19.17A. ARM Programmer's Model	332
19.18B. ARM Instruction Set	332
19.19C. ARM Stacks	332
20 Embedded Systems Programming in C₊ [22]	333
21 Сборка кросс-компилятора GNU Toolchain из исходных текстов	334
APP/HW: приложение/платформа	335
Подготовка BUILD-системы: необходимое ПО	335
dirs: создание структуры каталогов	335
Сборка в ОЗУ на ramdiske	336
Пакеты системы кросс-компиляции	336
gz: загрузка исходного кода для пакетов	337
Макро-правила для автоматической распаковки исходников	338
Общие параметры для ./configure	338
21.1 Сборка кросс-компилятора	339
21.1.1 cclibs0: библиотеки поддержки gcc	339
21.1.2 binutils0: ассемблер и линкер	340
21.1.3 gcc00: сборка stand-alone компилятора Си	342
21.1.4 newlib: сборка стандартной библиотеки libc	343
21.1.5 gcc0: пересборка компилятора Си/C ₊	343
21.2 Поддерживаемые платформы	343
21.2.1 i386: ПК и промышленные PC104	343
21.2.2 x86_64: серверные системы	343
21.2.3 AVR: Atmel AVR Mega	343
21.2.4 arm: процессоры ARM Cortex-Mx	343
21.2.5 armhf: SoCi Cortex-A, PXA270,.	343
21.3 Целевые аппаратные системы	344
21.3.1 x86: типовой компьютер на процессоре i386+	344
22 Porting The GNU Tools To Embedded Systems	345
23 Оптимизация кода	346
23.1 PGO оптимизация	346
VIII Микроконтроллеры Cortex-Mx	347
IX os86: низкоуровневое программирование i386	348
Специализированный GNU Toolchain для i386-pc-gnu	349
MultiBoot-загрузчик	349

X Спецификация MultiBoot	350
24 Introduction to Multiboot Specification	352
24.1 The background of Multiboot Specification	352
24.2 The target architecture	352
24.3 The target operating systems	353
24.4 Boot sources	353
24.5 Configure an operating system at boot-time	353
24.6 How to make OS development easier	353
24.7 Boot modules	354
The definitions of terms used through the specification	354
25 The exact definitions of Multiboot Specification	356
25.1 OS image format	356
25.1.1 The layout of Multiboot header	356
25.1.2 The magic fields of Multiboot header	357
25.1.3 The address fields of Multiboot header	358
25.1.4 The graphics fields of Multiboot header	359
25.2 Machine state	359
25.3 Boot information format	360
Examples	365
History	365
Index	365
XI Технологии	366
XII Сетевое обучение	367
XIII Базовая теоретическая подготовка	368
26 Математика	369
26.1 Высшая математика в упражнениях и задачах [68]	369
Запуск Maxima и Octave в пакетном режиме	370
26.1.1 Аналитическая геометрия на плоскости	370
XIV Прочее	378
Ф.И.Атауллаханов об учебниках США и России	379
27 Настройка редактора/IDE (g)Vim	380
27.1 для вашего собственного скриптового языка	380

Книги	380
Книги must have любому техническому специалисту	380
Математика, физика, химия	380
Обработка экспериментальных данных и метрология	381
Программирование	381
САПР, пакеты математики, моделирования, визуализации	382
Разработка языков программирования и компиляторов	383
Lisp/Sheme	385
Haskell	385
ML	385
Электроника и цифровая техника	386
Конструирование и технология	387
Приемы ручной обработки материалов	387
Механообработка	387
Использование OpenSource программного обеспечения	388
\LaTeX	388
Математическое ПО: Maxima, Octave, GNUPlot,..	389
САПР, электроника, проектирование печатных плат	390
Программирование	390
GNU Toolchain	390
JavaScript, HTML, CSS, Web-технологии:	390
Python	390
Prolog и логическое программирование	391
Разработка операционных систем и низкоуровневого ПО	393
Базовые науки	393
Математика	393
Символьная алгебра	396
Численные методы	398
Теория игр	399
Физика	399
Химия	401
Задачники	402
Математика	402
Стандарты и ГОСТы	403
Индекс	403

Об этом сборнике

В этот сборник (блогбук) я пишу отдельные статьи и переводы, сортированные только по общей тематике, и добавляю их, когда у меня в очередной раз зачесется **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 микроформат¹ для просмотра на телефонах и мобильных девайсах, проверялось на удобство чтения на Alcatel Onetouch 4007D Pixi: в горизонтальной ориентации вполне читается в транспорте.

¹ 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.¹

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

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.

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

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

Adventure in Prolog

1

© Published by: Amzi! inc.

Enjoy the adventure...

-Dennis Merritt

Adventure established the architecture of all fantasy computer games to follow. It was the first to create structures representing places and items and characters and to let the player explore and interact with the environment.

It was as totally addictive as today's wonderous graphical games that have built on the that very same architecture.

To learn more about the granddaddy of all such games, see: [Colossal Cave Adventure](#) and/or [google Adventure](#) and [Willie Crowther](#).

Preface

I was working for an aerospace company in the 1970s when someone got a copy of the original **Adventure** game and installed it on our mainframe computer. For the next month my lunch hours, evenings and weekends, as well as normal work hours, were consumed with fighting the fierce green dragon and escaping from the twisty little passages. Finally, with a few hints about the plover's egg and dynamite, I had proudly earned all the points in the game.

My elation turned to terror as I realized it was time for my performance review. My boss was a stern man, who was more comfortable with machines than with people. He opened up a large computer printout containing a log of the hours each of his programmers spent on the mainframe computer. He said he noticed that recently I

¹ © <http://www.amzi.com/AdventureInProlog/index.php>

had been working evenings and weekends and that he admired that type of dedication in his employees. He gave me the maximum raise and told me to keep up the good work.

Ever since I've had a warm spot in my heart for adventure games. Years later, when I got my first home computer, I immediately started to write my own adventure game in C#. First came the tools, a simple dynamic database to keep track of the game state and pattern matching functions to search that database. Then came a natural language parser for the front end. Functions implemented the various rules of the game.

At around the same time I joined the Boston Computer Society and attended a lecture of the newly formed Artificial Intelligence group. The lecture was about *Prolog*. I was amazed — here was a language that included all of the tools needed for building adventure games and more.

It had a much richer dynamic database and more powerful pattern matcher than the one I had written, plus its syntax was rules, which are much more natural for coding the specification of the game. It had a built-in search engine and, to top it all off, had tools for natural language processing.

I learned *Prolog* from the classic Clocksin and Mellish [31] text and started writing adventure games anew.

I went on to use *Prolog* for a number of expert system applications at my then current job, including a mainframe database performance tuning system and installation expert. This got others interested in the language and I began teaching it as well.

While the applications we were using Prolog for were serious and performed a key role in improving technical support for the growing company, I still found the adventure game to be an excellent showcase for teaching the language.

This book is the result of that work. It takes a pragmatic, rather than theoretical, approach to the language and is designed for programmers interested in adding this powerful language to their bag of tools.

I offer my thanks to Will Crowther and Don Woods for writing the first (and in my opinion still the best) adventure game and to the Boston Computer Society for testing the ideas in the book. Thanks also to Ray Reeves, who speaks fluent *Prolog*, and Nancy Wilson, who speaks fluent English, for their careful reading of the text.

© Dennis Merritt
Stow, Massachusetts, April 1996

Prolog tools

For either learning or deploying Prolog we recommend:

[Amzi! Prolog + Logic Server](#)
FREE (IDE only)

The Amzi! Prolog IDE, with its source code debugger, is an excellent tool for getting a solid understanding of Prolog's dynamic variable binding and built-in search.

The Amzi! Logic Server provides the tools for deploying Prolog with other development tools.

The full Amzi! Prolog + Logic Server package is available on either:

- an individual basis or
- an institutional site license.

Download

Other resources for learning Prolog

Checkout the numerous articles on Prolog:

Prolog Articles

and the archives from the years when Amzi! was editor of the AI Expert magazine. Many of the AI techniques are illustrated with Prolog code.

AI Newsletter

3.1 Getting Started

Prolog stands for PROgramming in LOGic. It was developed from a foundation of logical theorem proving and originally used for research in natural language processing. Although its popularity has sprung up mainly in the artificial intelligence (AI) community where it has been used for applications such as expert systems, natural language, and intelligent databases, it is also useful for more conventional types of applications. It allows for more rapid development and prototyping than most languages because it is semantically close to the logical specification of a program. As such, it approaches the ideal of executable program specifications².

Programming in Prolog is significantly different from conventional procedural programs and requires a readjustment in the way one thinks about programming. Logical relations are asserted, and Prolog is used to determine whether or not certain statements are true, and if true, what variable bindings make them true. This leads to a very declarative style of programming.

In fact, the term program does not accurately describe a Prolog collection of executable facts, rules and logical relationships, so you will often see term *logicbase* used in this book as well.

While Prolog is a fascinating language from a purely theoretical viewpoint, this book will stress Prolog as a practical tool for application development.

² and fast prototyping and RAD

Much of the book will be built around the writing of a short adventure game. The adventure game is a good example since it contains mundane programming constructs, symbolic reasoning, natural language, data, and logic.

Through exercises you will also build a simple expert system, an intelligent genealogical logicbase, and a mundane customer order entry application.

You should create a source file for the game, and enter the examples from the book as you go. You should also create source files for the other three programs covered in the exercises. Sample source code for each of the programs is included in the appendix [3.7.1](#).

The adventure game is called Nani Search. Your persona as the adventurer is that of a three year old girl. The lost treasure with magical powers is your nani (security blanket). The terrifying obstacle between you and success is a dark room. It is getting late and you're tired, but you can't go to sleep without your nani. Your mission is to find the nani.



This is Nani

Nani Search is composed of

- A read and execute command loop
- A natural language input parser
- Dynamic facts/data describing the current environment
- Commands that manipulate the environment
- Puzzles that must be solved

You control the game by using simple English commands (at the angle bracket ($>$) prompt) expressing the action you wish to take. You can go to other rooms, look at your surroundings, look in things, take things, drop things, eat things, inventory the things you have, and turn things on and off.

Figure 1.1. A sample run of Nani Search

You are in the kitchen.

You can see: apple, table, broccoli

You can go to: cellar, office, dining room

```
> go to the cellar
```

You can't go to the cellar because it's dark in the cellar,
and you're afraid of the dark.

```
> turn on the light
```

You can't reach the switch and there's nothing to stand on.

```
> go to the office
```

You are in the office.

You can see the following things: desk

You can go to the following rooms: hall, kitchen

```
> open desk
```

The desk contains:

flashlight

crackers

```
> take the flashlight
```

You now have the flashlight

```
> kitchen
```

You are in the kitchen

```
> turn on the light
```

flashlight turned on.

...

Figure 1.1 shows a run of a completed version of Nani Search. As you develop your own version you can of course change the game to reflect your own ideas of adventure.

The game will be implemented from the bottom up, because that fits better with the order in which the topics will be introduced. Prolog is equally adept at supporting top-down or inside-out program development.

A Prolog logicbase exists in the listener's workspace as a collection of small modular units, called *predicates*. They are similar to subroutines in conventional languages, but on a smaller scale.

The predicates can be added and tested separately in a Prolog program, which

makes it possible to incrementally develop the applications described in the book. Each chapter will call for the addition of more and more predicates to the game. Similarly, the exercises will ask you to add predicates to each of the other applications.

We will start with the Nani Search logicbase and quickly move into the commands that examine that logicbase. Then we will implement the commands that manipulate the logicbase.

Along the way there will be diversions where the same commands are rewritten using a different approach for comparison. Occasionally a topic will be covered that is critical to Prolog but has little application in Nani Search.

One of the final tasks will be putting together the top-level command processor. We will finish with the natural language interface.

The goal of this book is to make you feel comfortable with

- The Prolog logicbase of facts and rules
- The built-in theorem prover that allows Prolog to answer questions about the logicbase (backtracking search)
- How logical variables are used (They are different from the variables in most languages.)
- Unification, the built in pattern matcher
- Extra-logical features (like read and write that make the language practical)
- How to control Prolog's execution behavior

3.1.1 Jumping In

As with any language, the best way to learn Prolog is to use it. This book is designed to be used with a Prolog listener, and will guide you through the building of four applications.

1. Adventure game
2. Intelligent genealogical logicbase
3. Expert system
4. Customer order entry business application

The adventure game will be covered in detail in the main body of the text, and the others you will build yourself based on the exercises at the end of each chapter.

There will be two types of example code throughout the book. One is code, meant to be entered in a source file, and the other is interactions with the listener. The listener interactions are distinguished by the presence of the question mark and dash (-) listener prompt.

Here is a two-line program, meant to help you learn the mechanics of the editor and your listener.

```
mortal(X) :- person(X).  
person(socrates).
```

In the **Amzi! Eclipse IDE**, first create a project for your source files. Select **File** **New** **Project** on the main menu, then click on **Prolog** and **Project**, and enter the name of your project, **adventure**. Next, create a new source file. Select **File** **New** **File**,

and enter the name of your file, **mortal.pro**. Enter the program in the edit window, paying careful attention to upper and lowercase letters and punctuation. Then select **File > Save** from the menu.

Next, start the Prolog listener by selecting **Run > Run As > Interpreted Project**. Loading the source code in the **Listener** is called *consulting*. You should see a message indicating that your source file, **mortal.pro**, was consulted. This message is followed by the typical listener prompt.

```
?-
```

Entering the source code in the Listener is called *consulting*. Select **Listener > Consult** from the main menu, and select **mortal.pro** from the file menu. You can also consult a Prolog source file directly from the listener prompt like this.

```
?- consult(mortal).  
yes
```

See the documentation and/or online help for details on the Amzi! listener and Eclipse IDE.

In all the listener examples in this book, you enter the text after the prompt (?), the rest is provided by Prolog. When working with Prolog, it is important to remember to include the final period **.** and to press the **return** key. If you forget the period (and you probably will), you can enter it on the next line with a **.** **return**.

Once you've loaded the program, try the following Prolog queries.

```
?- mortal(socrates).  
yes  
?- mortal(X).  
X = socrates.
```

Now let's change the program. First type **quit.** to end the listener. Go back to the edit window and add the line

```
person(plato).
```

after the **person(socrates)** line.

Select **Run > Run As > Interpreted Project** to start the listener again with your updated source file. And test it.

```
?- mortal(plato).  
yes
```

One more test. Enter this query in the listener.

```
?- write('Hello World').  
Hello World  
yes
```

You are now ready to learn Prolog.

3.1.2 Logic Programming

Let's look at the simple example in more detail. In classical logic we might say "All people are mortal", or, rephrased for Prolog, "For all X, X is mortal if X is a person".

```
mortal(X) :- person(X).
```

Similarly, we can assert the simple fact that Socrates is a person.

```
person(socrates).
```

From these two logical assertions, Prolog can now prove whether or not Socrates is mortal.

```
?- mortal(socrates).
```

The listener responds

```
yes
```

We could also ask "Who is mortal?" like this

```
?- mortal(X).
```

and receive the response

```
X = socrates
```

This declarative style of programming is one of Prolog's major strengths. It leads to code that is easier to write and easier to maintain. For the most part, the programmer is freed from having to worry about control structures and transfer of control mechanisms. This is done automatically by Prolog.

By itself, however, a logical theorem prover is not a practical programming tool. A programmer needs to do things that have nothing to do with logic, such as read and write terms. A programmer also needs to manipulate the built-in control structure of Prolog in order for the program to execute as desired.

The following example illustrates a Prolog program that prints a report of all the known mortals. It is a mixture of pure logic from before, extra-logical I/O, and forced control of the Prolog execution behavior. The example is illustrative only, and the concepts involved will be explained in later chapters.

First add some more philosophers to the `mortal` source in order to make the report more interesting. Place them after `person(plato)`.

```
person(zeno).  
person(aristotle).
```

Next add the report-writing code, again being careful with punctuation and upper- and lowercase. Note that the format of this program is the same as that used for the logical assertions.

```
mortal_report:-  
    write('Known mortals are:'),nl,  
    mortal(X),  
    write(X),nl,  
    fail.
```

Figure 1.2. Sample program

% This is the syntax for comments.
% MORTAL – The first illustrative Prolog program

```
mortal(X) :- person(X).
```

```
person(socrates).  
person(plato).  
person(zeno).  
person(aristotle).
```

```
mortal_report:-  
    write('Known mortals are:'),nl,  
    mortal(X),  
    write(X),nl,  
    fail.
```

Listing 1.2 contains the full program, with some optional comments, indicated by the percent sign (%) at the beginning of a line. Load the program in the listener and try it. Note that the syntax of calling the report code is the same as the syntax used for posing the purely logical queries.

```
?- mortal_report.  
Known mortals are:  
socrates  
plato  
zeno  
aristotle  
false.
```

The final no or false is from Prolog, and will be explained later.

You should now be able to create and edit source files for Prolog, and be able to load and use them from a Prolog listener.

You have had your first glimpse of Prolog and should understand that it is fundamentally different from most languages, but can be used to accomplish the same goals and more.

3.1.3 Jargon

With any field of knowledge, the critical concepts of the field are embedded in the definitions of its technical terms. Prolog is no exception. When you understand terms

such as *predicate*, *clause*, *backtracking*, and *unification* you will have a good grasp of Prolog. This section defines the terms used to describe Prolog programs, such as predicate and clause. Execution-related terms, such as backtracking and unification will be introduced as needed throughout the rest of the text.

Prolog jargon is a mixture of programming terms, database terms, and logic terms. You have probably heard most of the terms before, **but** in Prolog **they don't necessarily mean what you think they mean**.

In Prolog the normally clear distinction between data and procedure becomes blurred. This is evident in the vocabulary of Prolog. Almost every concept in Prolog can be referred to by synonymous terms. One of the terms has a procedural flavor, and the other a data flavor.

We can illustrate this at the highest level. A Prolog *program* is a Prolog *logicbase*. As we introduce the vocabulary of Prolog, synonyms (from Prolog or other computer science areas) for a term will follow in parentheses. For example, at the highest level we have a Prolog *program* (*logicbase*).

The Prolog program is composed of *predicates* (*procedures*, *record types*, *relations*). Each is defined by its name and a number called *arity*. The arity is the fixed number of *arguments* (*attributes*, *fields*) the predicate has. Two predicates with the same name and different arity are considered to be **different** predicates.

In our sample program we saw three examples of predicates. Each of these three predicates has a distinctly different flavor.

person/1

looks like multiple **data records** with one data field for each.

mortal_report/0

looks like a **procedure** with no arguments.

mortal/1

a logical assertion or **rule** that is somewhere in between data and procedure.

Each predicate in a program is defined by the existence of one or more *clauses* in the logicbase. In the example program, the predicate **person/1** has four clauses. The other predicates have only one clause.

A clause can be either a *fact* or a *rule*. The three clauses of the **person/1** predicate are all **facts**. The single clauses of **mortal_report/0** and **mortal/1** are both **rules**.

3.2 Facts

This chapter describes the basic Prolog facts. They are the simplest form of Prolog predicates, and are **similar to records in a relational database**. As we will see in the next chapter they can be queried like database records.

The syntax for a fact is

```
pred(arg1, arg2, ... argN).
```

where

pred

The name of the predicate

arg1,..

The arguments

N

The arity

.

The syntactic end of all Prolog clauses

A predicate `pred/0` of arity 0 is simply

pred.

The arguments can be any legal Prolog *term*. The basic Prolog terms are **integer**

A positive or negative number whose absolute value is less than some implementation specific power of 2

atom

A text constant beginning with a lowercase letter

variable

Begins with an uppercase letter or underscore (`_`)

structure

Complex terms, which will be covered in chapter ??

Various Prolog implementations enhance this basic list with other data types, such as floating point numbers, or strings.

The Prolog character set is made up of

- Uppercase letters, A-Z
- Lowercase letters, a-z
- Digits, 0-9
- Symbols, + - * / \ ^ , . ~ : . ? @ # \$ &

Integers are made from digits. Other numerical types³ are allowed in some Prolog implementations.

Atoms are usually made from letters and digits with the first character being a **lowercase** letter, such as

`hello`

`twoWordsTogether`

`x14`

For readability, the underscore (`_`), **but not the hyphen (-)**, can be used as a separator in longer names. So the following are legal.

`a_long_atom_name`

`z_23`

³ 0x... hex numbers or floating point numbers

The following are not legal atoms.

```
no-embedded-hyphens
123nodigitsatbeginning
_nounderscorefirst
Nocapsfirst
```

Use single quotes to make any character combination a legal atom as follows.

```
'this-hyphen-is-ok'
'UpperCase'
'embedded blanks'
```

Do not use double quotes ("") to build atoms. This is a special syntax that causes the character string to be treated as a list of ASCII character codes (string).

Atoms can also be legally made from symbols, as follows.

```
-->
++
```

Variables are similar to atoms, but are distinguished by beginning with either an **uppercase** letter or the underscore (_).

```
X
Input_List
_4th_argument
Z56
```

Using these building blocks, we can start to code facts. The predicate name follows the rules for atoms. The arguments can be any Prolog terms.

Facts are often used to store the data a program is using. For example, a business application might have `customer/3`.

```
customer('John Jones', boston, good_credit).
customer('Sally Smith', chicago, good_credit).
```

The single quotes are needed around the names because they begin with uppercase letters and because they have embedded blanks.

Another example is a windowing system that uses facts to store data about the various windows. In this example the arguments give the window name and coordinates of the upper left and lower right corners.

```
window(main, 2, 2, 20, 72).
window(errors, 15, 40, 20, 78).
```

A medical diagnostic expert system might have `disease/2`.

```
disease(plague, infectious).
```

A Prolog listener provides the means for dynamically recording facts and rules in the logicbase, as well as the means to **query (call)** them. The logicbase is updated by **consulting** or **reconsulting** program source. Predicates can also be typed directly into the listener, but they are not saved between sessions.

3.2.1 Nani Search

We will now begin to develop **Nani Search** by defining the basic facts that are meaningful for the game. These include

- The rooms and their connections
- The things and their locations
- The properties of various things
- Where the player is at the beginning of the game

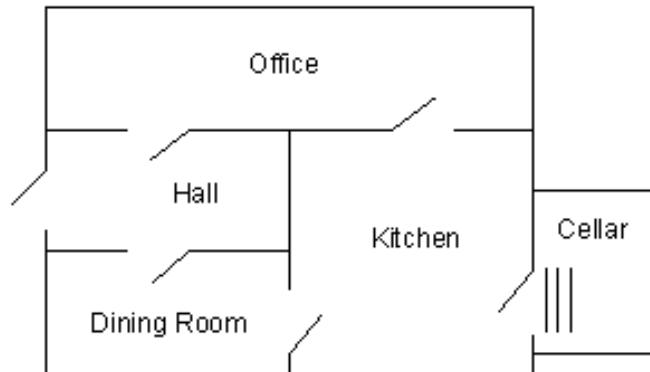


Figure 2.1. The rooms of **Nani Search**

Open a new source file and save it as **myadven.pro**⁴, or whatever name you feel is appropriate. You will make your changes to the program in that source file. (A completed version of **nanisrch.pro** is in the Prolog *samples/* directory, **samples/prolog/misc_one_file**.)

First we define the **rooms** with the predicate **room/1**, which has five clauses, all of which are facts. They are based on the game map in figure 2.1.

```
room(kitchen).  
room(office).  
room(hall).  
room('dining room').  
room(cellar).
```

We define the **locations of things** with a two-argument predicate **location/2**. The **first** argument will mean the **thing** and the **second** will mean its **location**. To begin with, we will add the following things⁵.

```
location(desk, office).  
location(apple, kitchen).  
location(flashlight, desk).  
location('washing machine', cellar).  
location(nani, 'washing machine').
```

⁴ or **nani.pl**, .pl is ProLog program extension

⁵ is can be thought as **relations** in traditional RDBMS

```
location(broccoli, kitchen).  
location(crackers, kitchen).  
location(computer, office).
```

The symbols we have chosen, such as **kitchen** and **desk** have meaning to us, but none to Prolog. The relationship between the arguments should also accurately reflect our meaning.

For example, the meaning we attach to **location/2** is “The first argument is located in the second argument”. Fortunately Prolog considers **location(sink, kitchen)** and **location(kitchen, sink)** to be different. Therefore, as long as we are consistent in our use of arguments, we can accurately represent our meaning and avoid the potentially ambiguous interpretation of the kitchen being in the sink.

We are not as lucky when we try to represent the **connections between rooms**. Let's start, however, with **door/2**, which will contain facts such as

```
door(office, hall).
```

We would like this to mean “There is a connection from the office to the hall, **or** from the hall to the office”.

Unfortunately, Prolog considers **door(office, hall)** to be different from **door(hall, office)**. If we want to accurately represent a two-way connection, we would have to define **door/2** twice for each connection⁶.

```
door(office, hall).  
door(hall, office).
```

The strictness about order serves our purpose well for location, but it creates this problem for connections between rooms. If the office is connected to the hall, then we would like the reverse to be true as well.

For now, we will just add **one-way doors** to the program; we will address the symmetry problem again in the next chapter **3.3** and resolve it in chapter **5 3.5**.

```
door(office, hall).  
door(kitchen, office).  
door(hall, 'dining room').  
door(kitchen, cellar).  
door('dining room', kitchen).
```

Here are some other facts about properties of things the game player might try to eat.

```
edible(apple).  
edible(crackers).  
  
tastes_yucky(broccoli).
```

⁶ it is *symmetric relation*

Finally we define the initial status of the flashlight, and the player's location at the beginning of the game.

```
turned_off(flashlight).  
here(kitchen).
```

We have now seen how to use basic facts to represent data in a Prolog program.

3.2.2 Exercises

During the course of completing the exercises you will develop three Prolog applications in addition to **Nani Search**. The exercises from each chapter will build on the work of previous chapters. Suggested solutions to the exercises are contained in the Prolog source files listed in the appendix [3.7.1](#), and are also included in **samples/prolog/misc**. The files

gene.pl

A genealogical intelligent logicbase

custord.pl

A customer order entry application

birds.pl

An expert system that identifies birds

Not all applications will be covered in each chapter. For example, the expert system requires an understanding of rules and will not be started until the end of chapter [5.3.5](#).

3.2.3 Genealogical Logicbase

1- First create a source file **gene.pl** for the genealogical logicbase application. Start by adding a few members of your family tree. It is important to be accurate, since we will be exploring family relationships. Your own knowledge of who your relatives are will verify the correctness of your Prolog programs.

Start by recording the gender of the individuals. Use two separate predicates, **male/1** and **female/1**. For example

```
male(dennis).  
male(michael).
```

```
female(diana).
```

Remember, if you want to include **uppercase** characters or embedded **blanks** you must enclose the name in 'single' (not double) quotes. For example

```
male('Ghenghis Khan').
```

2- Enter a two-argument predicate that records the parent-child relationship. One argument represents the parent, and the other the child. It doesn't matter in which **order** you enter the arguments, as long as you are **consistent**. Often Prolog programme adopt the convention that `parent(A,B)` is interpreted "A is the parent → of B". For example

```
parent(dennis, michael).  
parent(dennis, diana).
```

3.2.4 Customer Order Entry

3- Create a source file for the customer order entry program. We will begin it with three record types (predicates). The first is `customer/3` where the three arguments are

arg1

Customer name

arg2

City

arg3

Credit rating (aaa, bbb, etc)

Add as many customers as you see fit.

4- Next add clauses that define the items that are for sale. It should also have three arguments

arg1

Item identification numbers

arg2

Item name

arg3

The reorder point for inventory (when at or below this level, reorder)

5- Next add an inventory record for each item. It has two arguments.

arg1

Item identification number (same as in the item record)

arg2

Amount in stock

3.3 Simple Queries

Now that we have some facts in our Prolog program, we can **consult** the program in the listener and **query**, or **call**, the facts. This chapter, and the next 3.4, will assume the Prolog program contains only facts. Queries against programs with rules will be covered in a later chapter ??.

Prolog queries work by pattern matching. The query pattern is called a **goal**. If there is a fact that matches the **goal**, then the **query succeeds** and the listener responds with **yes**.⁷ If there is **no matching** fact, then the query **fails** and the listener responds with **no**.⁸

Prolog's **pattern matching** is called **unification**. In the case where the logicbase contains only facts, unification succeeds if the following three conditions hold simultaneously:

- The predicate named in the goal and logicbase are the same.
- Both predicates have the same arity.
- All of the arguments are the same.

Before proceeding, review figure 3.1, which has a listing of the program so far.

Figure 3.1. The listing of **Nani Search** entered at this point

```
/* /block comment/
               Nani Search
*/
% locations /line comment/
room(kitchen).
room(office).
room(hall).
room('dining_room').
room(cellar).

% doors
door(office, hall).
door(kitchen, office).
door(hall, 'dining_room').
door(kitchen, cellar).
door('dining_room', kitchen).

% things places
location(desk, office).
location(apple, kitchen).
location(flashlight, desk).
location('washing_machine', cellar).
location(nani, 'washing_machine').
```

⁷ or **true**.

⁸ or **false**.

```
location(broccoli, kitchen).  
location(crackers, kitchen).  
location(computer, office).
```

% edible items

```
edible(apple).  
edible(crackers).
```

```
tastes_yucky(broccoli).
```

% initial state

```
here(kitchen).
```

The first query we will look at asks if the office is a room in the game. To pose this, we would enter that goal followed by a period at the listener prompt.

```
?- room(office).  
yes
```

Prolog will respond with a `yes` or `true`. If we wanted to know if the attic was a room, we would enter that goal.

```
?- room(attic).  
no
```

Prolog will respond with a `no` if no match was found. Likewise, we can ask about the locations of things.

```
?- location(apple, kitchen).  
yes
```

```
?- location(kitchen, apple).  
no
```

Prolog responds to our location query patterns in a manner that makes sense to us. That is, the kitchen is not located in the apple.

However, here is the problem with the one-way doors, which we still haven't fixed. It is mentioned again to stress the importance of the order of the arguments.

```
?- door(officE, hall).  
yes  
  
?- door(hall, officE).  
no
```

Goals can be generalized by the use of Prolog **variables**. They do not behave like the variables in other languages, and are better called **logical variables** (although Prolog does not precisely correspond to logic). The logical variables replace one or more of the arguments in the goal.

Logical variables add a new dimension to unification. As before, the predicate names and arity must be the same for unification to succeed. However, when the corresponding arguments are compared, an (unbound) **variable** will successfully **match any term**.

After successful unification, the logical variable takes on the value of the term it was matched with. This is called **binding** the variable. When a goal with a variable successfully unifies with a fact in the logicbase, Prolog returns the value of the newly bound variable.

Since there may be more than one value a variable can be bound to and still satisfy the goal, Prolog provides the means for you to ask for alternate values. After an answer you can enter a semicolon `;`. It causes Prolog to look for alternative bindings for the variables. Entering anything else at the prompt ends the query.

For example, we can use a logical variable to find all of the rooms.

```
?- room(X).  
X = kitchen ;  
X = office ;  
X = hall ;  
X = 'dining room' ;  
X = cellar ;  
no
```

The last `no` means there are no more answers.

Here's how to find all the things in the kitchen. (Remember, variables begin with uppercase letters.)

```
?- location(Thing, kitchen).  
Thing = apple ;  
Thing = broccoli ;  
Thing = crackers ;  
no
```

We can use two variables to see everything in every place.

```
?- location(Thing, Place).  
Thing = desk  
Place = office ;  
  
Thing = apple  
Place = kitchen ;
```

```
Thing = flashlight
Place = desk ;
...
no
```

Other sample applications might have the following queries.

What customers live in Boston, and what is their credit rating?

```
?- customer(X, boston, Y).
```

What is the title of chapter 2 **3.2** ?

```
?- chapter(2,Title).
```

What are the coordinates of window **main** ?

```
?- window(main,Row1,Col1,Row2,Col2).
```

3.3.1 How Queries Work

When Prolog tries to satisfy a goal about a predicate, such as **location/2**, it searches through the clauses defining **location/2**. When it finds a match for its variables, it marks the particular clause that was used to satisfy the goal. Then, if the user asks for more answers, it resumes its search of the clauses at that place marker.

Referring to the list of clauses in figure 3.1, let's look closer at this process with the query **location(X, kitchen)**. First, unification is attempted between the query pattern and the first clause of **location/2**.

Pattern	Clause #1
location(X, kitchen)	location(desk, office)

This unification fails. The predicate names are the same, the number of arguments is the same, but the second argument in the pattern, **kitchen**, is **different** from the second argument in the clause, **office**.

Next, unification is attempted between the pattern and the second clause of **location**.

Pattern	Clause #2
location(X, kitchen)	location(apple, kitchen)

This unification succeeds. The predicate names, **arity** (number of arguments), and second arguments **are the same**. The first arguments can be made the same if the variable **X** in the pattern takes the value **apple**.

Now that unification succeeds, the Prolog listener reports its success, and the binding of the variable **X**.

```
?- location(X, kitchen).
```

```
X = apple
```

If the user presses a key other than the semicolon ; at this point, the listener responds with yes indicating the query ended successfully.

If the user presses the semicolon (;) key, the listener looks for other solutions. First it *unbinds* the variable X. Next it resumes the search using the clause following the one that had just satisfied the query. This is called **backtracking**. In the example that would be the third clause.

Pattern	Clause #3
location(X, kitchen)	location(flashlight, desk)

This fails, and the search continues. Eventually the sixth clause succeeds.

Pattern	Clause #6
location(X, kitchen)	location(broccoli, kitchen)

As a result, the variable X is now rebound to broccoli, and the listener responds

X = broccoli ;

Again, entering a semicolon ; causes X to be unbound and the search to continue with the seventh clause, which also succeeds.

X = crackers ;

As before, entering anything except a semicolon (;) causes the listener to respond yes, indicating success. A semicolon (;) causes the unbinding of X and the search to continue. But now, there are no more clauses that successfully unify with the pattern, so the listener responds with no indicating the final attempt has failed.

no

The best way to understand Prolog execution is to trace its execution in the debugger. But first it is necessary to have a deeper understanding of goals.

A Prolog goal has four **ports** representing the flow of control through the goal: call, exit, redo, and fail. First the goal is called. If successful it is exited. If not it fails. If the goal is retried, by entering a semicolon (;) the redo port is entered. Figure 3.2 shows the goal and its ports.



Figure 3.2. The ports of a Prolog goal

The behaviors at each port are

call

Begins searching for clauses that unify with the goal

exit

Indicates the goal is satisfied, sets a place marker at the clause and binds the variables appropriately

redo

Retries the goal, unbinds the variables and resumes search at the place marker

fail

Indicates no more clauses match the goal

Prolog debuggers use these ports to describe the state of a query. Figure 3.3 shows a trace of the `location(X, kitchen)` query. Study it carefully because it is the key to your understanding of Prolog. The number in parentheses indicates the current clause.

Figure 3.3. Prolog trace of `location(X,kitchen)`

```
?- location(X, kitchen).  
CALL: - location(X, kitchen)  
EXIT:(2) location(apple , kitchen)  
    X = apple ;  
REDO: location(X, kitchen)  
EXIT:(6) location(broccoli , kitchen)  
    X = broccoli ;  
REDO: location(X, kitchen)  
EXIT:(7) location(crackers , kitchen)  
    X = crackers ;  
FAIL - location(X, kitchen)  
    no
```

Because the trace information presented in this book is designed to teach Prolog rather than debug it, the format is a little different from that used in the actual debugger. Run the Amzi! Source Code Debugger on these queries to see how they work for real.

To start the **Amzi! Debugger**, highlight your project name or edit a source file in your project, then select `Run > Debug As > Interpreted Project` from the main menu.

You will see a separate perspective with multiple views that contain trace information. Enter the query `location(X, kitchen)` in the `Debug Listener` view. You will see the trace start in the debugger view.

Use the `Step Over` button in the debugger to creep from port to port. When output appears in the listener view, enter semicolons `;` to continue the search. See the help files for more details on the debugger.

Unification between goals and facts is actually more general than has been presented. Variables can also occur in the facts of the Prolog logicbase as well.

For example, the following fact could be added to the Prolog program. It might mean everyone sleeps.

```
sleeps(X).
```

You can add it directly in the listener, to experiment with, like this.

```
?- assert(sleeps(X)).  
yes
```

Queries against a logicbase with this fact give the following results.

```
?- sleeps(jane).  
yes  
  
?- sleeps(tom).  
yes
```

Notice that the listener does not return the variable bindings of `X=jane` and `X=tom`. While they are surely bound that way, the listener only lists variables mentioned in the query, not those used in the program.

Prolog can also bind variables to variables.

```
?- sleeps(Z).  
Z = H116  
  
?- sleeps(X).  
X = H247
```

When two unbound variables match, they are both bound, but **not to a value**. They are bound together, so that if either one takes a value, the other takes the same value. This is usually implemented by binding both variables to a common internal variable. In the first query above, both `Z` in the query and `X` in the fact are bound to internal variable `H116`. In this way Prolog remembers they have the same value. If either one is bound to a value later on, both automatically bind to that value. This feature of Prolog distinguishes it from other languages and, as we will discover later, gives Prolog much of its power.

The two queries above are the same, even though one uses the same character `X` that is used in the fact `sleeps(X)`. The variable in the fact is considered different from the one in the query.

3.3.2 Exercises

The exercise sections will often contain nonsense Prolog questions. These are queries against a meaningless logicbase to strengthen your understanding of Prolog without the benefit of meaningful semantics. You are to predict the answers to the query and then try them in Prolog to see if you are correct. If you are not, trace the queries to better understand them.

Nonsense Prolog

1- Consider the following Prolog logicbase

```
easy(1).  
easy(2).  
easy(3).  
  
gizmo(a,1).  
gizmo(b,3).  
gizmo(a,2).  
gizmo(d,5).  
gizmo(c,3).  
gizmo(a,3).  
gizmo(c,4).
```

and predict the answers to the queries below, including all alternatives when the semicolon (;) is entered after an answer.

```
?- easy(2).  
?- easy(X).
```

```
?- gizmo(a,X).  
?- gizmo(X,3).  
?- gizmo(d,Y).  
?- gizmo(X,X).
```

2- Consider this logicbase,

```
harder(a,1).  
harder(c,X).  
harder(b,4).  
harder(d,2).
```

and predict the answers to these queries.

```
?- harder(a,X).  
?- harder(c,X).  
?- harder(X,1).  
?- harder(X,4).
```

Adventure Game

3- Enter the listener and reproduce some of the example queries you have seen against `location/2`. List or print `location/2` for reference if you need it. Remember to respond with a semicolon [;] for multiple answers. Trace the query.

Genealogical Logicbase

4- Pose queries against the genealogical logicbase that:

- Confirm a parent relationship such as parent(dennis, diana)
- Find someone's parent such as parent(X, diana)
- Find someone's children such as parent(dennis, X)
- List all parent-children such as parent(X,Y)

5- If `parent/2` seems to be working, you can add additional family members to get a larger logicbase. Remember to include the corresponding `male/1` or `female/1` predicate for each individual added.

Customer Order Entry

6- Pose queries against the customer order entry logicbase that

- find customers in a given city
- find customers with a given credit rating
- confirm a given customer's credit rating
- find the customers in a given city with a given credit rating
- find the reorder quantity for a given item
- find the item number for a given item name
- find the inventory level for a given item number

3.4 Compound Queries

Simple goals can be combined to form compound queries. For example, we might want to know if there is anything good to eat in the kitchen. In Prolog we might ask

```
?- location(X, kitchen), edible(X).
```

Whereas a simple query had a single goal, the compound query has a **conjunction** of goals. The comma separating the goals is read as **and**.

Logically (declaratively) the example means “Is there an X such that X is located in the `kitchen` and X is `edible`?” If the **same variable** name appears **more than once** in a query, it must have the same value in all places it appears. The query in the above example will only succeed if there is a single value of X that can satisfy both goals.

However, the variable name has no significance to any other query, or clause in the logicbase. If X appears in other queries or clauses, that query or clause gets its own copy of the variable. We say the **scope of a logical variable is a query**.

Trying the sample query we get

```
?- location(X, kitchen), edible(X).  
X = apple ;  
X = crackers ;  
no
```

The `broccoli` does not show up as an answer because we did not include it in the `edible/1` predicate.

This logical query can also be interpreted procedurally, using an understanding of Prolog's execution strategy. The procedural interpretation is: "First find an `X` located in the `kitchen`, and then test to see if it is `edible`. If it is not, go back and find another `X` in the `kitchen` and test it. Repeat until successful, or until there are no more `Xs` in the `kitchen`".

To understand the execution of a compound query, think of the goals as being arranged from left to right. Also think of a separate table which is kept for the current variable bindings. The flow of control moves back and forth through the goals as Prolog attempts to find variable bindings that satisfy the query.

Each goal can be entered from either the left or the right, and can be left from either the left or the right. These are the ports of the goal as seen in the last chapter.

A compound query begins by calling the first goal on the left. If it succeeds, the next goal is called with the variable bindings as set from the previous goal. If the query finishes via the exit port of the rightmost goal, it succeeds, and the listener prints the values in the variable table.

If the user types semicolon `;` after an answer, the query is re-entered at the redo port of the rightmost goal. Only the variable bindings that were set in that goal are undone.

If the query finishes via the fail port of the leftmost goal, the query fails. Figure 4.1 shows a compound query with the listener interaction on the ending ports.

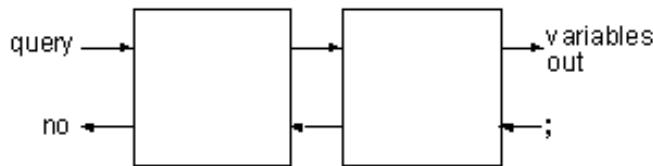


Figure 4.1. Compound queries

Annotated trace of compound query

Next log contains the annotated trace of the sample query. Make sure you understand it before proceeding.

```
?- location(X, kitchen), edible(X).
```

The trace has a new feature, which is a number in the first column that indicates the goal being worked on.

First the goal `location(X, kitchen)` is called, and the trace indicates that pattern matches the second clause of `location`.

```
1 CALL location(X, kitchen)
```

It succeeds, and results in the binding of X to `apple`.

```
1 EXIT (2)location(apple, kitchen)
```

Next, the second goal `edible(X)` is called. However, X is now bound to `apple`, so it is called as `edible(apple)`.

```
2 CALL edible(apple)
```

It succeeds on the first clause of `edible/1`, thus exiting the query successfully.

```
2 EXIT (1) edible(apple)
```

```
X = apple ;
```

Entering semicolon (`;`) causes the listener to backtrack into the rightmost goal of the query.

```
2 REDO edible(apple)
```

There are no other clauses that match this pattern, so it fails.

```
2 FAIL edible(apple)
```

Leaving the fail port of the second goal causes the listener to enter the redo port of the first goal. In so doing, the variable binding that was established by that goal is undone, leaving X unbound.

```
1 REDO location(X, kitchen)
```

It now succeeds at the sixth clause, rebinding X to `broccoli`.

```
1 EXIT (6) location(broccoli, kitchen)
```

The second goal is called again with the new variable binding. This is a fresh call, just as the first one was, and causes the search for a match to begin at the first clause

```
2 CALL edible(broccoli)
```

There is no clause for `edible(broccoli)`, so it fails.

```
2 FAIL edible(broccoli)
```

The first goal is then re-entered at the redo port, undoing the variable binding.

```
1 REDO location(X, kitchen)
```

It succeeds with a new variable binding.

```
1 EXIT (7) location(crackers, kitchen)
```

This leads to the second solution to the query.

```
2 CALL edible(crackers)
2 EXIT (2) edible(crackers)
X = crackers ;
```

Typing semicolon (;) initiates backtracking again, which fails through both goals and leads to the ultimate failure of the query.

```
2 REDO edible(crackers)
2 FAIL edible(crackers)
1 REDO location(X, kitchen)
1 FAIL location(X, kitchen)
no
```

In this example we had a single variable, which was bound (given a value) by the first goal and tested in the second goal. We will now look at a more general example with two variables. It is attempting to ask for all the things located in rooms adjacent to the kitchen.

In logical terms, the query says “Find a T and R such that there is a door from the kitchen to R and T is located in R”. In procedural terms it says “First find an R with a door from the kitchen to R. Use that value of R to look for a T located in R“.

```
?- door(kitchen, R), location(T,R).
```

```
R = office
```

```
T = desk ;
```

```
R = office
```

```
T = computer ;
```

```
R = cellar
```

```
T = 'washing machine' ;
```

```
no
```

In this query, the backtracking is more complex. Figure 4.3 shows its trace.

Figure 4.3. Trace of a compound query

```
Goal: door(kitchen, R), location(T,R)
```

```
1 CALL door(kitchen, R)
1 EXIT (2) door(kitchen, office)
2 CALL location(T, office)
2 EXIT (1) location(desk, office)
R = office
```

```

T = desk ;
2 REDO location(T, office)
2 EXIT (8) location(computer, office)
    R = office
    T = computer ;
2 REDO location(T, office)
2 FAIL location(T, office)
1 REDO door(kitchen, R)
1 EXIT (4) door(kitchen, cellar)
2 CALL location(T, cellar)
2 EXIT (4) location('washing machine', cellar)
    R = cellar
    T = 'washing machine' ;
2 REDO location(T, cellar)
2 FAIL location(T, cellar)
1 REDO door(kitchen, R)
1 FAIL door(kitchen, R)
no

```

Notice that the variable `R` is bound by the first goal and `T` is bound by the second. Likewise, the two variables are unbound by entering the redo port of the goal that bound them. After `R` is first bound to `office`, that binding sticks during backtracking through the second goal. Only when the listener backtracks into the first goal does `R` get unbound.

3.4.1 Built-in Predicates

Up to this point we have been satisfied with the format Prolog uses to give us answers. We will now see how to generate output that is customized to our needs. The example will be a query that lists all of the items in the kitchen. This will require performing I/O and forcing the listener to automatically backtrack to find all solutions.

To do this, we need to understand the concept of the *built-in* (evaluable) predicate. A built-in predicate is predefined by Prolog. There are no clauses in the logicbase for built-in predicates. When the listener encounters a goal that matches a built-in predicate, it calls a predefined procedure.

Built-in predicates are usually written in the language used to implement the listener. They can perform functions that have nothing to do with logical theorem proving, such as writing to the console. For this reason they are sometimes called extra-logical predicates.

However, since they appear as Prolog goals they must be able to respond to either a call from the left or a redo from the right. Its response in the redo case is referred to as its behavior on backtracking.

We will introduce specific built-in predicates as we need them. Here are the I/O predicates that will let us control the output of our query.

`write/1`

This predicate always succeeds when called, and has the side effect of writing its

argument to the console. It always fails on backtracking. Backtracking does not undo the side effect.

nl/0

Succeeds, and starts a new line. Like write, it always succeeds when called, and fails on backtracking.

tab/1

It expects the argument to be an integer and tabs that number of spaces. It succeeds when called and fails on backtracking.

Figure 4.4 is a stylized picture of a goal showing its internal control structure. We will compare this with the internal flow of control of various built-in predicates.

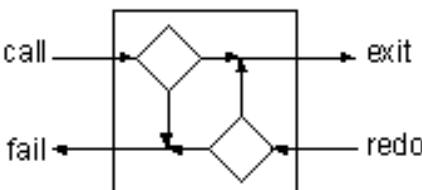


Figure 4.4. Internal flow of control through a normal goal

In figure 4.4, the upper left diamond represents the decision point after a call. Starting with the first clause of a predicate, unification is attempted between the query pattern and each clause, until either unification succeeds or there are no more clauses to try. If unification succeeded, branch to exit, marking the clause that successfully unified, if it failed, branch to fail.

The lower right diamond represents the decision point after a redo. Starting with the most recent clause found in the predicate, unification is again attempted between the query pattern and remaining clauses. If it succeeds, branch to exit, if not, branch to fail.

The I/O built-in predicates differ from normal goals in that they never change the direction of the flow of control. If they get control from the left, they pass control to the right. If they get control from the right, they pass control to the left as shown in figure 4.5.

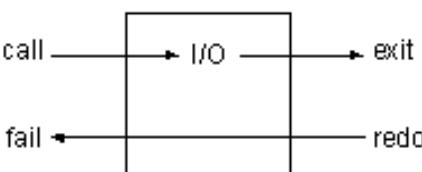


Figure 4.5. Internal flow of control through an I/O predicate

The output I/O predicates do not affect the variable table; however, they may output values from it. They simply leave their mark at the console each time control passes through them from left to right.

There are built-in predicates that do affect backtracking, and we have need of one of them for the first example. It is `fail/0`, and, as its name implies, it always fails.

If `fail/0` gets control from the left, it immediately passes control back to the redo port of the goal on the left. It will never get control from the right, since it never allows control to pass to its right. Figure 4.6 shows its internal control structure.

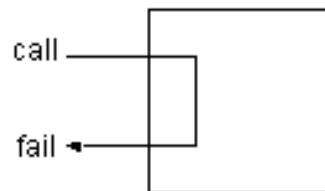


Figure 4.6. Internal flow of control through the `fail/0` predicate

Previously we relied on the listener to display variable bindings for us, and used the semicolon (`;`) response to generate all of the possible solutions. We can now use the I/O built-in predicates to display the variable bindings, and the `fail/0` predicate to force backtracking so all solutions are displayed.

Here then is the query that lists everything in the kitchen.

```
?- location(X, kitchen), write(X) ,nl, fail.  
apple  
broccoli  
crackers  
no
```

The final `no` means the query failed, as it was destined to, due to the `fail/0`.

Figure 4.7 shows the control flow through this query.

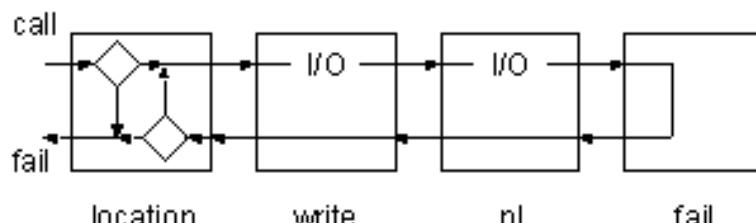


Figure 4.7. Flow of control through query with built-in predicates

Figure 4.8. Trace of query with built-in predicates

Goal: `location(X, kitchen), write(X), nl, fail.`

```
1 CALL location(X, kitchen)  
1 EXIT (2) location(apple, kitchen)  
2 CALL write(apple)
```

```
apple
2 EXIT write(apple)
3 CALL nl

3 EXIT nl
4 CALL fail
4 FAIL fail
3 REDO nl
3 FAIL nl
2 REDO write(apple)
2 FAIL write(apple)
1 REDO location(X, kitchen)
1 EXIT (6) location(broccoli, kitchen)
2 CALL write(broccoli)
    broccoli
2 EXIT write(broccoli)
3 CALL nl

3 EXIT nl
4 CALL fail
4 FAIL fail
3 REDO nl
3 FAIL nl
2 REDO write(broccoli)
2 FAIL write(broccoli)
1 REDO location(X, kitchen)
1 EXIT (7) location(crackers, kitchen)
2 CALL write(crackers)
    crackers
2 EXIT write(crackers)
3 CALL nl

3 EXIT nl
4 CALL fail
4 FAIL fail
3 REDO nl
3 FAIL nl
2 REDO write(crackers)
2 FAIL write(crackers)
1 REDO location(X, kitchen)
1 FAIL location(X, kitchen)
    no
```

3.4.2 Exercises

Nonsense Prolog

- 1- Consider the following Prolog logicbase.

```
easy(1).  
easy(2).  
easy(3).
```

```
gizmo(a,1).  
gizmo(b,3).  
gizmo(a,2).  
gizmo(d,5).  
gizmo(c,3).  
gizmo(a,3).  
gizmo(c,4).
```

```
harder(a,1).  
harder(c,X).  
harder(b,4).  
harder(d,2).
```

Predict the results of the following queries. Then try them and trace them to see if you were correct.

```
?- gizmo(a,X),easy(X).  
?- gizmo(c,X),easy(X).  
?- gizmo(d,Z),easy(Z).
```

```
?- easy(Y),gizmo(X,Y).
```

```
?- write('report'), nl, easy(T), write(T),  
    gizmo(M,T), tab(2), write(M), fail.
```

```
?- write('buggy'), nl, easy(Z), write(X),  
    gizmo(Z,X), tab(2), write(Z), fail.
```

```
?- easy(X),harder(Y,X).  
?- harder(Y,X),easy(X).
```

Adventure Game

- 2- Experiment with the queries you have seen in this chapter.

- 3- Predict the results of this query before you execute it. Then try it. Trace it if you were wrong.

```
?- door(kitchen, R), write(R), nl, location(T,R),
   tab(3), write(T), nl, fail.
```

Genealogical Logicbase

4- Compound queries can be used to find family relationships in the genealogical logicbase. For example, find someone's mother with

```
?- parent(X, someone), female(X).
```

Write similar queries for fathers, sons, and daughters. Trace these queries to understand their behavior (or misbehavior if they are not working right for you).

5- Experiment with the ordering of the goals. In particular, contrast the queries.

```
?- parent(X, someone), female(X).
```

```
?- female(X), parent(X, someone).
```

Do they both give the same answer? Trace both queries and see which takes more steps.

6- The same predicate can be used multiple times in the same query. For example, we can find grandparents

```
?- parent(X, someone), parent(GP, X).
```

7- Write queries which find grandmothers, grandfathers, and great-great grandparents.

Customer Order Entry

8- Write a query against the item and inventory records that returns the inventory level for an item when you only know the item name.

3.5 Rules

We said earlier a predicate is defined by clauses, which may be facts or rules. A rule is no more than a stored query. Its syntax is

head :- **body**.

where

head

a predicate definition (just like a fact)

:-

the neck symbol, sometimes read as "if"

body

one or more goals (a query)

For example, the compound query that finds out where the good things to eat are can be stored as a rule with the predicate name `where_food/2`.

```
where_food(X,Y) :-  
    location(X,Y),  
    edible(X).
```

It states “There is something X to eat in room Y if X is located in Y, and X is edible”.

We can now use the new rule directly in a query to find things to eat in a room. As before, the semicolon (;) after an answer is used to find all the answers.

```
?- where_food(X, kitchen).  
X = apple ;  
X = crackers ;  
no  
  
?- where_food(Thing, 'dining room').  
no
```

Or it can check on specific things.

```
?- where_food(apple, kitchen).  
yes
```

Or it can tell us everything.

```
?- where_food(Thing, Room).  
Thing = apple  
Room = kitchen ;  
  
Thing = crackers  
Room = kitchen ;  
no
```

Just as we had multiple facts defining a predicate, we can have multiple rules for a predicate. For example, we might want to have the broccoli included in `where_food/2`. (Prolog doesn’t have an opinion on whether or not broccoli is legitimate food. It just matches patterns.) To do this we add another `where_food/2` clause for things that taste_yucky.

```
where_food(X,Y) :-  
    location(X,Y),  
    edible(X).  
where_food(X,Y) :-  
    location(X,Y),  
    tastes_yucky(X).
```

Now the broccoli shows up when we use the semicolon (;) to ask for everything.

```
?- where_food(X, kitchen).  
X = apple ;  
X = crackers ;  
X = broccoli ;  
no
```

Until this point, when we have seen Prolog try to satisfy goals by searching the clauses of a predicate, all of the clauses have been facts.

3.5.1 How Rules Work

With rules, Prolog unifies the goal pattern with the head of the clause. If unification succeeds, then Prolog initiates a new query using the goals in the body of the clause.

Rules, in effect, give us multiple levels of queries. The first level is composed of the original goals. The next level is a new query composed of goals found in the body of a clause from the first level.

Each level can create even deeper levels. Theoretically, this could continue forever. In practice it can continue until the listener runs out of space.

Figure 5.1 shows the control flow after the head of a rule has been matched. Notice how backtracking from the third goal of the first level now goes into the second level.

Figure 5.1. Control flow with rules In this example, the middle goal on the first level succeeds or fails if its body succeeds or fails. When entered from the right (redo) the goal reenters its body query from the right (redo). When the query fails, the next clause of the first-level goal is tried, and if the next clause is also a rule, the process is repeated with the second clause's body.

As always with Prolog, these relationships become clearer by studying a trace. Figure 5.2 contains the annotated trace of the `where_food/2` query. Notice the appearance of a two-part number. The first part of the number indicates the query level. The second part indicates the number of the goal within the query, as before. The parenthetical number is the clause number. For example

2-1 EXIT (7) location(crackers, kitchen) means the exit occurred at the second level, first goal using clause number seven.

The query is

```
?- where_food(X, kitchen).
```

First the clauses of `where_food/2` are searched.

```
1-1 CALL where_food(X, kitchen)
```

The pattern matches the head of the first clause, and while it is not at a port, the trace could inform us of the clause it is working on.

```
1-1 try (1) where_food(X, kitchen)
```

The body of the first clause is then set up as a query, and the trace continues.

```
2-1 CALL location(X, kitchen)
```

From this point the trace proceeds exactly as it did for the compound query in the previous chapter.

```
2-1 EXIT (2) location(apple, kitchen)
2-2 CALL edible(apple)
2-2 EXIT (1) edible(apple)
```

Since the body has succeeded, the goal from the previous (first) level succeeds.

```
1-1 EXIT (1) where_food(apple, kitchen)
      X = apple ;
```

Backtracking goes from the first-level goal, into the second level, proce-

```
\begin{verbatim}
1-1 REDO where_food(X, kitchen)
  2-2 REDO edible(apple)
  2-2 FAIL edible(apple)
  2-1 REDO location(X, kitchen)
  2-1 EXIT (6) location(broccoli, kitchen)
  2-2 CALL edible(broccoli)
  2-2 FAIL edible(broccoli)
  2-1 REDO location(X, kitchen)
  2-1 EXIT (7) location(crackers, kitchen)
  2-2 CALL edible(crackers)
  2-2 EXIT (2) edible(crackers)
1-1 EXIT (1) where_food(crackers, kitchen)
      X = crackers ;
```

Now any attempt to backtrack into the query will result in no more answers, and the query will fail.

```
2-2 REDO edible(crackers)
2-2 FAIL edible(crackers)
2-1 REDO location(X, kitchen)
2-1 FAIL location(X, kitchen)
```

This causes the listener to look for other clauses whose heads match the query pattern. In our example, the second clause of `where_food/2` also matches the query pattern.

```
1-1 REDO where_food(X, kitchen)
```

Again, although traces usually don't tell us so, it is building a query from the body of the second clause.

```
1-1 try (2) where_food(X, kitchen)
```

Now the second query proceeds as normal, finding the broccoli, which `tastes_yucky`

```
2-1 CALL location(X, kitchen)
2-1 EXIT (2) location(apple, kitchen)
2-2 CALL tastes_yucky(apple)
2-2 FAIL tastes_yucky(apple)
2-1 REDO location(X, kitchen)
2-1 EXIT (6) location(broccoli, kitchen)
2-2 CALL tastes_yucky(broccoli)
2-2 EXIT (1) tastes_yucky(broccoli)
1-1 EXIT (2) where_food(broccoli, kitchen)
X = broccoli ;
```

Backtracking brings us to the ultimate no, as there are no more `where_food/2` clauses to try.

```
2-2 REDO tastes_yucky(broccoli)
2-2 FAIL tastes_yucky(broccoli)
2-1 REDO location(X,kitchen)
2-1 EXIT (7) location(crackers, kitchen)
2-2 CALL tastes_yucky(crackers)
2-2 FAIL tastes_yucky(crackers)
2-2 REDO location(X, kitchen)
2-2 FAIL location(X, kitchen)
1-1 REDO where_food(X, kitchen)
1-1 FAIL where_food(X, kitchen)
no
```

Figure 5.2. Trace of a query with rules It is important to understand the relationship between the first-level and second-level variables in this query. These are independent variables, that is, the X in the query is not the same as the X that shows up in the body of the `where_food/2` clauses, values for both happen to be equal due to unification.

To better understand the relationship, we will slowly step through the process of transferring control. Subscripts identify the variable levels.

The goal in the query is

```
?- where_food(X1, kitchen)
```

The head of the first clause is

```
\begin{verbatim}
```

```
where_food(X2, Y2)
```

Remember the 'sleeps' example in chapter 3 where a query with a variable

So, after unification between the goal and the head, the variable binding

```
\begin{verbatim}
```

```
X1 = _01
```

```
X2 = _01
```

```
Y2 = kitchen
```

The second-level query is built from the body of the clause, using these bindings.

```
location(_01, kitchen), edible(_01).
```

When internal variable `_01` takes on a value, such as `apple`, both `X`'s then take on the same value. This is fundamentally different from the assignment statements that set variable values in most computer languages.

3.5.2 Using Rules

Using rules, we can solve the problem of the one-way doors. We can define a new two-way predicate with two clauses, called `connect/2`.

```
connect(X,Y) :- door(X,Y).
```

```
connect(X,Y) :- door(Y,X).
```

It says "Room X is connected to a room Y if there is a door from X to Y, or if there is a door from Y to X". Note the implied *or* between clauses. Now `connect/2` behaves the way we would like.

```
?- connect(kitchen, office).
```

```
yes
```

```
?- connect(office, kitchen).
```

```
yes
```

We can list all the connections (which is twice the number of doors) with a general query.

```
?- connect(X,Y).
```

```
X = office
```

```
Y = hall ;
X = kitchen
Y = office ;
...
X = hall
Y = office ;

X = office
Y = kitchen ;
...
```

With our current understanding of rules and built-in predicates we can now add more rules to Nani Search. We will start with `look/0`, which will tell the game player where he or she is, what things are in the room, and which rooms are adjacent.

To begin with, we will write `list_things/1`, which lists the things in a room. It uses the technique developed at the end of chapter 4 to loop through all the pertinent facts.

```
list_things(Place) :-
    location(X, Place),
    tab(2),
    write(X),
    nl,
    fail.
```

We use it like this.

```
?- list_things(kitchen).
apple
broccoli
crackers
no
```

There is one small problem with `list_things/1`. It gives us the list, but it always fails. This is all right if we call it by itself, but we won't be able to use it in conjunction with other rules that follow it (to the right as illustrated in our diagrams). We can fix this problem by adding a second `list_things/1` clause which always succeeds.

```
list_things(Place) :-
    location(X, Place),
    tab(2),
    write(X),
    nl,
    fail.

list_things(AnyPlace).
```

Now when the first clause fails (because there are no more `location/2`s to try) the second `list_things/1` clause will be tried. Since its argument is a variable it will successfully match with anything, causing `list_things/1` to always succeed and leave through the `exit` port.

As with the second clause of `list_things/1`, it is often the case that we do not care what the value of a variable is, it is simply a place marker. For these situations there is a special variable called the anonymous variable, represented as an underscore (`_`). For example

```
list_things(_).
```

Next we will write `list_connections/1`, which lists connecting rooms. Since rules can refer to other rules, as well as to facts, we can write `list_connections/1` just like `list_things/1` by using the `connection/2` rule.

```
list_connections(Place) :-  
    connect(Place, X),  
    tab(2),  
    write(X),  
    nl,  
    fail.  
list_connections(_).
```

Trying it gives us

```
?- list_connections(hall).  
dining room  
office  
yes
```

Now we are ready to write `look/0`. The single fact `here(kitchen)` tells us where we are in the game. (In chapter 7 we will see how to move about the game by dynamically changing `here/1`.) We can use it with the two list predicates to write the full `look/0`.

```
look :-  
    here(Place),  
    write('You are in the '), write(Place), nl,  
    write('You can see:'), nl,  
    list_things(Place),  
    write('You can go to:'), nl,  
    list_connections(Place).
```

Given we are in the kitchen, this is how it works.

```
?- look.  
You are in the kitchen  
\begin{verbatim}  
You can see:  
    apple  
    broccoli  
    crackers  
You can go to:  
    office  
    cellar  
    dining room  
yes
```

We now have an understanding of the fundamentals of Prolog, and it is worth summarizing what we have learned so far.

We have seen the following about rules in Prolog.

- A Prolog program is a logicbase of interrelated facts and rules.
- The rules communicate with each other through unification, Prolog's built-in pattern matcher.
- The rules communicate with the user through built-in predicates such as `write/1`.
- The rules can be queried (called) individually from the listener.

We have seen the following about Prolog's control flow.

- The execution behavior of the rules is controlled by Prolog's built-in backtracking search mechanism.
- We can force backtracking with the built-in predicate `fail`.
- We can force success of a predicate by adding a final clause with dummy variables as arguments and no body.

We now understand the following aspects of Prolog programming.

- Facts in the logicbase (locations, doors, etc.) replace conventional data definition.
- The backtracking search (`list_things/1`) replaces the coding of many looping constructs.
- Passing of control through pattern matching (`connect/2`) replaces conditional test and branch structures.
- The rules can be tested individually, encouraging modular program development.

- Rules that call rules encourage the programming practices of procedure abstraction and data abstraction. (For example, `look/0` doesn't know how `list_things/1` works, or how the location data is stored.)

With this level of understanding, we can make a lot of progress on the exercise applications. Take some time to work with the programs to consolidate your understanding before moving on to the following chapters.

3.5.3 Exercises

Nonsense Prolog

1- Consider the following Prolog logicbase.

```
a(a1,1).  
a(A,2).  
a(a3,N).  
  
b(1,b1).  
b(2,B).  
b(N,b3).  
  
c(X,Y) :- a(X,N), b(N,Y).  
  
d(X,Y) :- a(X,N), b(Y,N).  
d(X,Y) :- a(N,X), b(N,Y).
```

Predict the answers to the following queries, then check them with Prolog, tracing.

```
?- a(X,2).  
  
?- b(X,kalamazoo).  
  
?- c(X,b3).  
?- c(X,Y).  
  
?- d(X,Y).
```

Adventure Game

2- Experiment with the various rules that were developed during this chapter, tracing them all.

3- Write `look_in/1` for Nani Search. It should list the things located in its argument. For example, `look_in(desk)` should list the contents of the desk.

Genealogical Logicbase

4- Build rules for the various family relationships that were developed as queries in the last chapter. For example

```
mother(M,C) :-  
    parent(M,C),  
    female(M).
```

5- Build a rule for siblings. You will probably find your rule lists an individual as his/her own sibling. Use trace to figure out why.

6- We can fix the problem of individuals being their own siblings by using the built-in predicate that succeeds if two values are unequal, and fails if they are the same. The predicate is $\neq(X,Y)$. Jumping ahead a bit (to operator definitions in chapter 12), we can also write it in the form $X \neq Y$.

7- Use the sibling predicate to define additional rules for brothers, sisters, uncles, aunts, and cousins.

8- If we want to represent marriages in the family logicbase, we run into the two-way door problem we encountered in Nani Search. Unlike parent/2, which has two arguments with distinct meanings, married/2 can have the arguments reversed without changing the meaning.

Using the Nani Search door/2 predicate as an example, add some basic family data with a spouse/2 predicate. Then write the predicate married/2 using connect/2 as a model.

9- Use the new married predicate to add rules for uncles and aunts that get uncles and aunts by marriage as well as by blood. You should have two rules for each of these relationships, one for the blood case and one for the marriage case. Use trace to follow their behavior.

10- Explore other relationships, such as those between in-laws.

11- Write a predicate for grandparent/2. Use it to find both a grandparent and a grandchild.

```
grandparent(someone, X).  
grandparent(X, someone).
```

Trace its behavior for both uses. Depending on how you wrote it, one use will require many more steps than the other. Write two predicates, one called grandparent/2 and one called grandchild/2. Order the goals in each so that they are efficient for their intended uses.

Customer Order Entry

12- Write a rule item_quantity/2 that is used to find the inventory level of a named item. This shields the user of this predicate from having to deal with the item numbers.

13- Write a rule that produces an inventory report using the `item_quantity/2` predicate. It should display the name of the item and the quantity on hand. It should also always succeed. It will be similar to `list_things/2`.

14- Write a rule which defines a good customer. You might want to identify different cases of a good customer.

Expert Systems

Expert systems are often called rule-based systems. The rules are "rules of thumb" used by experts to solve certain problems. The expert system includes an inference engine, which knows how to use the rules.

There are many kinds of inference engines and knowledge representation techniques that are used in expert systems. Prolog is an excellent language for building any kind of expert system. However, certain types of expert systems can be built directly using Prolog's native rules. These systems are called structured selection systems.

The code listing for 'birds' in the appendix contains a sample system that can be used to identify birds. You will be asked to build a similar system in the exercises. It can identify anything, from animals to cars to diseases.

15- Decide what kind of expert system you would like to build, and add a few initial identification rules. For example, a system to identify house pets might have these rules.

```
pet(dog):- size(medium), noise(woof).  
pet(cat):- size(medium), noise(meow).  
pet(mouse):- size(small), noise(squeak).
```

16- For now, we can use these rules by putting the known facts in the logicbase. For example, if we add `size(medium)` and `noise(meow)` and then pose the query `pet(X)` we will find `X=cat`.

Many Prologs allow clauses to be entered directly at the listener prompt, which makes using this expert system a little easier. The presence of the neck symbol (`(:-)`) signals to the listener that the input is a clause to be added. So to add facts directly to the listener workspace, they must be made into rules, as follows.

```
?- size(medium) :- true.  
recorded  
  
?- noise(meow) :- true.  
recorded
```

Jumping ahead, you can also use `assert/1` like this

```
?- assert(size(medium)).  
yes  
?- assert(noise(meow)).  
yes
```

These examples use the predicates in the general form `attribute(value)`. In this simple example, the pet attribute is deduced. The size and noise attributes must be given.

17- Improve the expert system by having it ask for the attribute/values it can't deduce. We do this by first adding the rules

```
size(X):- ask(size, X).  
noise(X):- ask(noise, X).
```

For now, `ask/2` will simply check with the user to see if an attribute/value pair is true or false. It will use the built-in predicate `read/1` which reads a Prolog term (ending in a period of course).

```
ask(Attr, Val):-  
    write(Attr), tab(1), write(Val),  
    tab(1), write('yes/no'), write(?),  
    read(X),  
    X = yes.
```

The last goal, `X = yes`, attempts to unify `X` and `yes`. If `yes` was read, then it succeeds, otherwise, it fails.

3.6 Arithmetic

Prolog must be able to handle arithmetic in order to be a useful general purpose programming language. However, arithmetic does not fit nicely into the logical scheme of things.

That is, the concept of evaluating an arithmetic expression is in contrast to the straight pattern matching we have seen so far. For this reason, Prolog provides the built-in predicate `is` that evaluates arithmetic expressions. Its syntax calls for the use of operators, which will be described in more detail in chapter 12.

```
X is <arithmetic expression>
```

The variable `X` is set to the value of the arithmetic expression. On backtracking it is unassigned.

The arithmetic expression looks like an arithmetic expression in any other programming language.

Here is how to use Prolog as a calculator.

```
?- X is 2 + 2.
```

```
X = 4
```

```
?- X is 3 * 4 + 2.
```

```
X = 14
```

Parentheses clarify precedence.

```
?- X is 3 * (4 + 2).
```

```
X = 18
```

```
?- X is (8 / 4) / 2.
```

```
X = 1
```

In addition to `is`, Prolog provides a number of operators that compare two numbers. These include `greater than`, `less than`, `greater or equal than`, and `less or equal than`. They behave more logically, and succeed or fail according to whether the comparison is true or false. Notice the order of the symbols in the greater or equal than and less than or equal operators. They are specifically constructed not to look like an arrow, so that you can use arrow symbols in your programs without confusion.

```
X > Y
```

```
X < Y
```

```
X >= Y
```

```
X =< Y
```

Here are a few examples of their use.

```
?- 4 > 3.
```

```
yes
```

```
?- 4 < 3.
```

```
no
```

```
?- X is 2 + 2, X > 3.
```

```
X = 4
```

```
?- X is 2 + 2, 3 >= X.
```

```
no
```

```
?- 3+4 > 3*2.
```

```
yes
```

They can be used in rules as well. Here are two example predicates. One converts centigrade temperatures to Fahrenheit, the other checks if a temperature is below freezing.

```
c_to_f(C,F) :-
```

```
    F is C * 9 / 5 + 32.
```

```
freezing(F) :-
```

```
    F =< 32.
```

Here are some examples of their use.

```
?- c_to_f(100,X).
```

```
X = 212
```

```
yes
```

```
?- freezing(15).
```

```
yes
```

```
?- freezing(45).
```

```
no
```

3.6.1 Exercises

Customer Order Entry

1- Write a predicate `valid_order/3` that checks whether a customer order is valid. The arguments should be customer, item, and quantity. The predicate should succeed only if the customer is a valid customer with a good credit rating, the item is in stock, and the quantity ordered is less than the quantity in stock.

2- Write a `reorder/1` predicate which checks inventory levels in the inventory record against the reorder quantity in the item record. It should write a message indicating whether or not it's time to reorder.

3.7 Managing Data

We have seen that a Prolog program is a logicbase of predicates, and so far we have entered clauses for those predicates directly in our programs. Prolog also allows us to manipulate the logicbase directly and provides built-in predicates to perform this function. The main ones are:

`asserta(X)`

Adds the clause `X` as the first clause for its predicate. Like the other I/O predicates, it always fails on backtracking and does not undo its work.

`assertz(X)`

Same as `asserta/1`, only it adds the clause `X` as the last clause for its predicate.

`retract(X)`

Removes the clause `X` from the logicbase, again with a permanent effect that is not undone on backtracking.

The ability to manipulate the logicbase is obviously an important feature for Nani Search. With it we can dynamically change the location of the player, as well as the stuff that has been picked up and moved.

We will first develop `goto/1`, which moves the player from one room to another. It will be developed from the top down, in contrast to `look/0` which was developed from the bottom up.

When the player enters the command `goto`, we first check if they can go to the place and if so move them so they can look around the new place. Starting from this description of `goto/1`, we can write the main predicate.

```
goto(Place):-  
    can_go(Place),  
    move(Place),  
    look.
```

Next we fill in the details. We can go to a room if it connects to where we are.

```
can_go(Place):-  
    here(X),  
    connect(X, Place).
```

We can test `can_go/1` immediately (assuming we are in the kitchen).

```
?- can_go(office).  
yes  
  
?- can_go(hall).  
no
```

Now, `can_go/1` succeeds and fails as we want it to, but it would be nice if it gave us a message when it failed. By adding a second clause, which is tried if the first one fails, we can cause `can_go/1` to write an error message. Since we want `can_go/1` to fail in this situation we also need to add a `fail` to the second clause.

```
can_go(Place):-  
    here(X),  
    connect(X, Place).  
can_go(Place):-  
    write('You can''t get there from here.'), nl,  
    fail.
```

This version of `can_go/1` behaves as we want.

```
?- can_go(hall).  
You can't get there from here.  
no
```

Next we develop `move/1`, which does the work of dynamically updating the logicbase to reflect the new location of the player. It retracts the old clause for `here/1` and replaces it with a new one. This way there will always be only one `here/1` clause representing the current place. Because `goto/1` calls `can_go/1` before `move/1`, the new `here/1` will always be a legal place in the game.

```
move(Place) :-  
    retract(here(X)),  
    asserta(here(Place)).
```

We can now use `goto/1` to explore the game environment. The output it generates is from `look/0`, which we developed in chapter 5.

```
?- goto(office).  
You are in the office  
You can see:  
    desk  
    computer  
You can go to:  
    hall  
    kitchen  
yes
```

```
?- goto(hall).  
You are in the hall  
You can see:  
You can go to:  
    dining room  
    office  
yes
```

```
?- goto(kitchen).  
You can't get there from here.  
no
```

We will also need 'asserta' and 'retract' to implement 'take' and 'put' commands in the game.

Here is `take/1`. For it we will define a new predicate, `have/1`, which has one clause for each thing the game player has. Initially, `have/1` is not defined because the player is not carrying anything.

```
take(X) :-  
    can_take(X),  
    take_object(X).
```

`can_take/1` is analogous to `can_go/1`.

```
can_take(Thing) :-  
    here(Place),  
    location(Thing, Place).  
can_take(Thing) :-  
    write('There is no '), write(Thing),  
    write(' here.'),  
    nl, fail.
```

`take_object/1` is analogous to `move/1`. It retracts a `location/2` clause and asserts a `have/1` clause, reflecting the movement of the object from the place to the player.

```
take_object(X):-  
    retract(location(X,_)),  
    asserta(have(X)),  
    write('taken'), nl.
```

As we have seen, the variables in a clause are local to that clause. There are no global variables in Prolog, as there are in many other languages. The Prolog logicbase serves that purpose. It allows all clauses to share information on a wider basis, replacing the need for global variables. 'asserts' and 'retracts' are the tools used to manipulate this global data.

As with any programming language, global data can be a powerful concept, easily overused. They should be used with care, since they hide the communication of information between clauses. The same code will behave differently if the global data is changed. This can lead to hard-to-find bugs.

Eliminating global data and the `assert` and `retract` capabilities of Prolog is a goal of many logic programmers. It is possible to write Prolog programs without dynamically modifying the logicbase, thus eliminating the problem of global variables. This is done by carrying the information as arguments to the predicates. In the case of an adventure game, the complete state of the game could be represented as predicate arguments, with each command called with the current state and returning a new modified state. This approach will be discussed in more detail in chapter 14.

Although the database-like approach presented here may not be the purest method from a logical standpoint, it does allow for a very natural representation of this game application.

Various Prologs provide varying degrees of richness in the area of logicbase manipulation. The built-in versions are usually unaffected by backtracking. That is, like the other I/O predicates, they perform their function when called and do nothing when entered from the redo port.

Sometimes it is desirable to have a predicate retract its assertions when the redo port is entered. It is easy to write versions of `assert` and `retract` that undo their work on backtracking.

```
backtracking_assert(X) :-  
    asserta(X).  
backtracking_assert(X) :-  
    retract(X), fail.
```

The first time through, the first clause is executed. If a later goal fails, backtracking will cause the second clause to be tried. It will undo the work of the first and fail, thus giving the desired effect.

3.7.1 Exercises

Adventure Game

1- Write `put/1\verb` which retracts a `have/1` clause and asserts a `location/2` clause in the current room.

2- Write `inventory/0` which lists the `have/1` things.

3- Use `goto/1`, `take/1`, `put/1`, `look/0`, and `inventory/0` to move about and examine the game environment so far.

4- Write the predicates `turn_on/1` and `turn_off/1` for Nani Search. They will be used to turn the flashlight on or off.

5- Add an open/closed status for each of the doors. Write `open` and `close` predicates that do the obvious. Fix `can_go/1` to check whether a door is open and write the appropriate error message if its not.

Customer Order Entry

6- In the order entry application, write a predicate `update_inventory/2` that takes an item name and quantity as input. Have it retract the old inventory amount, perform the necessary arithmetic and assert the new inventory amount.

NOTE: `retract(inventory(item_id,Q))` binds `Q` to the old value, thus alleviating the need for a separate goal to get the old value of the inventory.

7- We can now use the various predicates developed for the customer order entry system to write a predicate that prompts the user for order information and generates the order. The predicate can be simply `order/0`.

`order/0` should first prompt the user for the customer name, the item name and the quantity. For example

```
write('Enter customer name:'), read(C),
```

It should then use the rules for `good_customer` and `valid_order` to verify that this is a valid order. If so, it should assert a new type of record, `order/3`, which records the order information. It can then `update_inventory` and check whether its time to reorder.

The customer order entry application has been designed from the bottom up, since that is the way the material has been presented for learning. The order predicate should suggest that Prolog is an excellent tool for top-down development as well.

One could start with the concept that processing an order means reading the date, checking the order, updating inventory, and reordering if necessary. The necessary details of implementing these predicates could be left for later.

Expert System

8- The expert system currently asks for the same information over and over again. We can use the logicbase to remember the answers to questions so that `ask/2` doesn't re-ask something.

When `ask/2` gets a `yes` or `no` answer to a question about an attribute-value pair, assert a fact in the form

```
known(Attribute, Value, YesNo).
```

Add a first clause to `ask/2` that checks whether the answer is already known and, if so, succeeds. Add a second clause that checks if the answer is known to be false and, if so, fails.

The third clause makes sure the answer is not already known, and then asks the user as before. To do this, the built-in predicate `not/1` is used. It fails if its argument succeeds.

```
not (known(Attr, Val, Answer))
```

Appendix

Глава 4

[LPN] Learn *Prolog* Now!

1

© Patrick Blackburn, Johan Bos, Kristina Striegnitz

4.1 Факты, правила и запросы

Это раздел имеет две главных цели:

1. Дать несколько простых примеров программ на *Prolog*. Это будет введением в три базовых конструкции *Prologa*: *факт* (fact), *правило* (rule) и *запрос* (query). Также мы рассмотрим несколько других тем, таких как роль логики в *Prologе*, и важную идею *унификации* с выводом значений переменных.
2. Начало систематического изучения *Prologa* через определение *термов*, *атомов*, *переменных* и других синтаксических конструкций.

4.1.1 1.1 Some Simple Examples

There are only three basic constructs in Prolog: facts, rules, and queries. A collection of facts and rules is called a knowledge base (or a database) and Prolog programming is all about writing knowledge bases. That is, Prolog programs simply are knowledge bases, collections of facts and rules which describe some collection of relationships that we find interesting.

So how do we use a Prolog program? By posing queries. That is, by asking questions about the information stored in the knowledge base.

Now this probably sounds rather strange. It's certainly not obvious that it has much to do with programming at all. After all, isn't programming all about telling a computer what to do? But as we shall see, the Prolog way of programming makes a lot of sense, at least for certain tasks; for example, it is useful in computational linguistics

¹ © <http://www.learnprolognow.org/lpnpage.php?pageid=online>

and Artificial Intelligence (AI). But instead of saying more about Prolog in general terms, let's jump right in and start writing some simple knowledge bases; this is not just the best way of learning Prolog, it's the only way.

Knowledge Base 1

Knowledge Base 1 (KB1) is simply a collection of facts. Facts are used to state things that are unconditionally true of some situation of interest. For example, we can state that Mia, Jody, and Yolanda are women, that Jody plays air guitar, and that a party is taking place, using the following five facts:

```
woman(mia).  
woman(jody).  
woman(yolanda).  
playsAirGuitar(jody).  
party.
```

This collection of facts is KB1. It is our first example of a Prolog program. Note that the names mia , jody , and yolanda , the properties woman and playsAirGuitar , and the proposition party have been written so that the first letter is in lower-case. This is important; we will see why a little later on.

How can we use KB1? By posing queries. That is, by asking questions about the information KB1 contains. Here are some examples. We can ask Prolog whether Mia is a woman by posing the query:

```
?- woman(mia).
```

Prolog will answer

```
yes
```

for the obvious reason that this is one of the facts explicitly recorded in KB1. Incidentally, we don't type in the ?- . This symbol (or something like it, depending on the implementation of Prolog you are using) is the prompt symbol that the Prolog interpreter displays when it is waiting to evaluate a query. We just type in the actual query (for example woman(mia)) followed by . (a full stop). The full stop is important. If you don't type it, Prolog won't start working on the query.

Similarly, we can ask whether Jody plays air guitar by posing the following query:

```
?- playsAirGuitar(jody).
```

Prolog will again answer yes, because this is one of the facts in KB1. However, suppose we ask whether Mia plays air guitar:

```
?- playsAirGuitar(mia).
```

We will get the answer

no

Why? Well, first of all, this is not a fact in KB1. Moreover, KB1 is extremely simple, and contains no other information (such as the rules we will learn about shortly) which might help Prolog try to infer (that is, deduce) whether Mia plays air guitar. So Prolog correctly concludes that `playsAirGuitar(mia)` does not follow from KB1.

Here are two important examples. First, suppose we pose the query:

```
?- playsAirGuitar(vincent).
```

Again Prolog answers no. Why? Well, this query is about a person (Vincent) that it has no information about, so it (correctly) concludes that `playsAirGuitar(vincent)` cannot be deduced from the information in KB1.

Similarly, suppose we pose the query:

```
?- tatooed(jody).
```

Again Prolog will answer no. Why? Well, this query is about a property (being tatooed) that it has no information about, so once again it (correctly) concludes that the query cannot be deduced from the information in KB1. (Actually, some Prolog implementations will respond to this query with an error message, telling you that the predicate or procedure `tatooed` is not defined; we will soon introduce the notion of predicates.)

Needless to say, we can also make queries concerning propositions. For example, if we pose the query

```
?- party.
```

then Prolog will respond

yes

and if we pose the query

```
?- rockConcert.
```

then Prolog will respond

no

exactly as we would expect.

Knowledge Base 2

Here is KB2, our second knowledge base:

```
happy(yolanda).  
listens2Music(mia).  
listens2Music(yolanda) :- happy(yolanda).  
playsAirGuitar(mia) :- listens2Music(mia).  
playsAirGuitar(yolanda) :- listens2Music(yolanda).
```

There are two facts in KB2, `listens2Music(mia)` and `happy(yolanda)`. The last three items it contains are rules.

Rules state information that is conditionally true of the situation of interest. For example, the first rule says that Yolanda listens to music if she is happy, and the last rule says that Yolanda plays air guitar if she listens to music. More generally, the `:`- should be read as “if”, or “is implied by”. The part on the left hand side of the `:`- is called the head of the rule, the part on the right hand side is called the body. So in general rules say: if the body of the rule is true, then the head of the rule is true too.

And now for the key point:

If a knowledge base contains a rule head `: -` body, and Prolog knows that body follows from the information in the knowledge base, then Prolog can infer head.

This fundamental deduction step is called *modus ponens*.

Let's consider an example. Suppose we ask whether Mia plays air guitar:

```
?- playsAirGuitar(mia).
```

Prolog will respond yes. Why? Well, although it can't find `playsAirGuitar(mia)` as a fact explicitly recorded in KB2, it can find the rule

```
playsAirGuitar(mia) :- listens2Music(mia).
```

Moreover, KB2 also contains the fact `listens2Music(mia)`. Hence Prolog can use the rule of modus ponens to deduce that `playsAirGuitar(mia)`.

Our next example shows that Prolog can chain together uses of modus ponens. Suppose we ask:

```
?- playsAirGuitar(yolanda).
```

Prolog would respond yes. Why? Well, first of all, by using the fact `happy(yolanda)` and the rule

```
listens2Music(yolanda) :- happy(yolanda).
```

Prolog can deduce the new fact `listens2Music(yolanda)`. This new fact is not explicitly recorded in the knowledge base — it is only implicitly present (it is inferred knowledge). Nonetheless, Prolog can then use it just like an explicitly recorded fact. In particular, from this inferred fact and the rule

```
playsAirGuitar(yolanda) :- listens2Music(yolanda).
```

it can deduce `playsAirGuitar(yolanda)` , which is what we asked it. Summing up: any fact produced by an application of modus ponens can be used as input to further rules. By chaining together applications of modus ponens in this way, Prolog is able to retrieve information that logically follows from the rules and facts recorded in the knowledge base.

The facts and rules contained in a knowledge base are called clauses. Thus KB2 contains five clauses, namely three rules and two facts. Another way of looking at KB2 is to say that it consists of three predicates (or procedures). The three predicates are:

```
listens2Music  
happy  
playsAirGuitar
```

The happy predicate is defined using a single clause (a fact). The `listens2Music` and `playsAirGuitar` predicates are each defined using two clauses (in one case, two rules, and in the other case, one rule and one fact). It is a good idea to think about Prolog programs in terms of the predicates they contain. In essence, the predicates are the concepts we find important, and the various clauses we write down concerning them are our attempts to pin down what they mean and how they are inter-related.

One final remark. We can view a fact as a rule with an empty body. That is, we can think of facts as conditionals that do not have any antecedent conditions, or degenerate rules.

Knowledge Base 3

Knowledge Base 4

Knowledge Base 5

4.1.2 1.2 Prolog Syntax

Atoms

Numbers

Variables

Complex terms

4.1.3 1.3 Exercises

4.1.4 1.4 Practical Session

4.2 Chapter 2 Unification and Proof Search

4.3 Chapter 3 Recursion

4.4 Chapter 4 Lists

4.5 Chapter 5 Arithmetic

4.6 Chapter 6 More Lists

4.7 Chapter 7 Definite Clause Grammars

4.8 Chapter 8 More Definite Clause Grammars

4.9 Chapter 9 A Closer Look at Terms

4.10 Chapter 10 Cuts and Negation

4.11 Chapter 11 Database Manipulation and Collecting Solutions

4.12 Chapter 12 Working With Files

Глава 5

Учебник Фишера

© J.R.Fisher 's *PrologTutorial* ¹

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 Project², 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 университета³, как это сделано в классической книге по *Prologу* под авторством Clocksin и Mellish (1981,1992).

¹ © https://www.cpp.edu/~jrfisher/www/prolog_tutorial/contents.html

² компьютерный проект пятого поколения

³ 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 5.1 рассматривает среду программирования на *Prolog* для начинающих.

Раздел 5.2 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 5.3 рассматривает работу внутренних механизмов *Prolog*-движка. Раздел 5.3 рекомендуется просмотреть сразу после того, как студент изучил 2-3 примера программ из раздела 5.2. Последняя секция этого раздела рассматривает **мета-интерпретаторы Prologa**.

Раздел 5.4 дает краткий обзор основных встроенных предикатов, многие из которых используются в разделе 5.2..

Раздел 5.5 рассматривает разработку программ A*-поиска на *Prolog*. Раздел 5.5.3 содержит программу $\alpha\beta$ -поиска для игры tic tac toe.

Раздел 5.6 представляет уникальное и обширное описание логического мета-интерпретатора для нормальных логических баз правил.⁴

Раздел 5.7 представляет введение во встроенный в *Prolog* генератор парсеров

⁴ 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 5.8 показывает приемы реализации различных *Prolog*-прототипов. Новый раздел 5.8.4 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 автором⁵ 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

5.1 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.

⁵ 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 [5.2.2](#). 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 [5.2.2](#). 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. [5.2.7](#)).
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 [5.2.7](#). 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.

5.2 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.

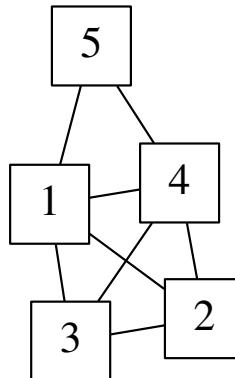
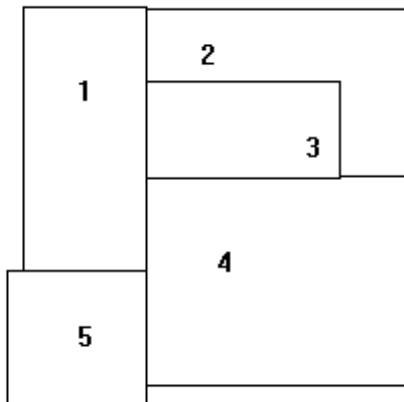
5.2.1 Раскраска карт

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 регионов⁶, 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, например⁷. По данным численным именам регионов строим представление в виде *графа смежности*:

⁶ таких как географические карты

⁷ 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”⁸.

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 переменной⁹ (последний случай), мы получаем от *Prolog*-системы заключение: выполняется ли запрошенное отношение-*целк* и при каких значениях переменных, имея в виду ранее

⁸ причем не указывается какой элемент главный или подчиненный, все элементы отношения равноправны

⁹ имя с большой буквы

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

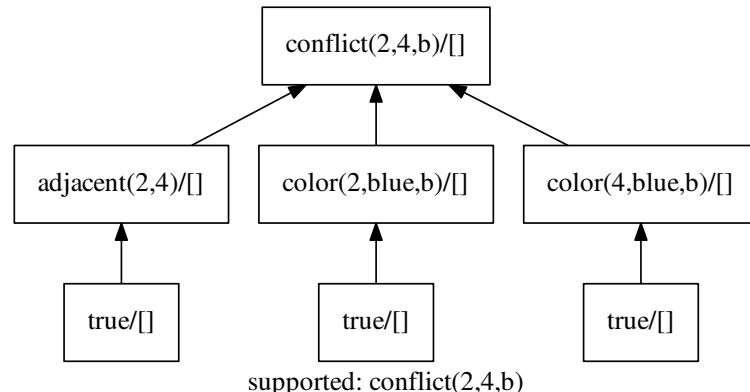
Можно определить другое отношение с тем же именем **conflict** но с другим количеством логических параметров:

```
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**.¹¹

```
?- 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:

¹⁰ которые являются *базой знаний*, или *экспертной системой*

¹¹ /цифра имеет название *аристотель*

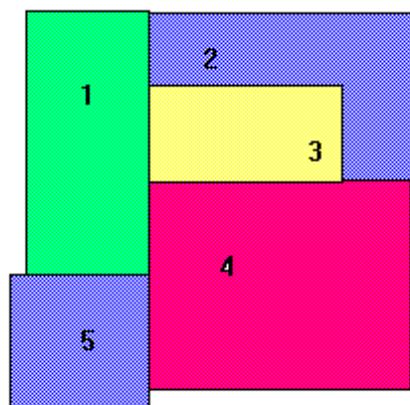
`adjacent(2,4).`

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 раскраски¹². 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 программу¹³ 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.

5.2.2 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*.

¹² given colors to color with

¹³ 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”¹⁴, и второе предложение заявляет что “факториал N есть F ¹⁵ если $N>0$ и $N1$ есть $N-1$, и факториал $N1$ есть $F1$, и F есть $N*F1$.

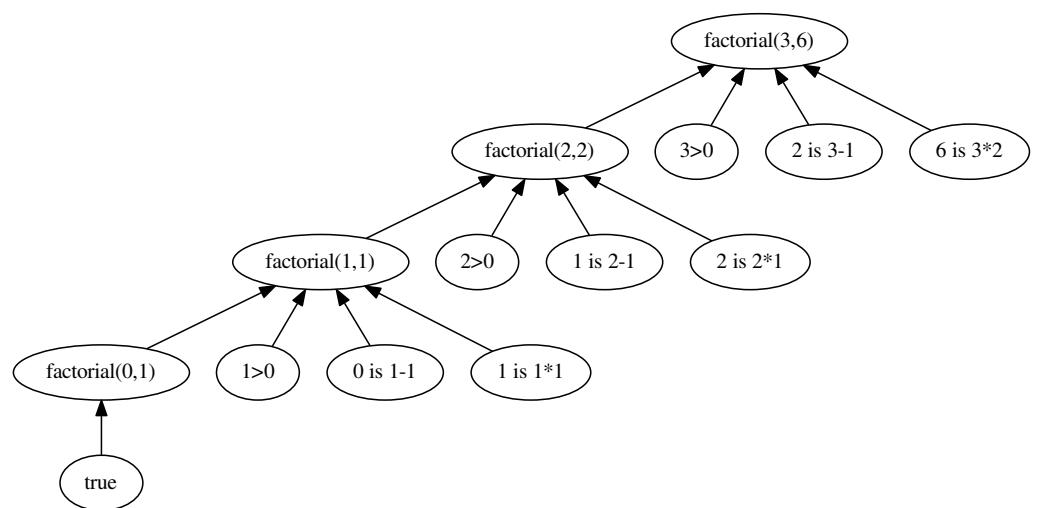
Prolog-цель (goal) для вычисления факториала от 3 дает ответ в W — **переменной цели**:

```
?- 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.

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

¹⁵ точнее: N и F связаны отношением “факториал”



Все арифметические листья |true| при исполнении¹⁶, 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 успешна¹⁷. 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 5.6 мы рассмотрим другой подход к семантике программ. Clause-деревья представляют интуитивный и корректный подход к описанию семантики.

¹⁶ в соответствии с предполагаемой интерпретацией

¹⁷ `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 **5.3.1**, 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 5.2.2.1 Используя первый вариант программы факториала, четко покажите что не существует clause-дерева с корнем `factorial(5,2)`, имеющего все true листья.

Упражнение 5.2.2.2 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.

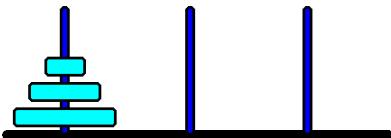
5.2.3 Классическая задача “Ханойские башни”

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 храненилище. 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 5.2.3.1 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 5.2.3.2 Попробуйте *Prolog*-цель `?-move(3, left, right, left)`. Что не так? Предложите способ исправления, и проследите процесс работы исправления.

5.2.4 Загрузка, редактирование, хранение программ

Примеры показывают различные способы хранения и загрузки *Prolog*-программ, и пример вызова системного редактора. Читателю предлагается предварительно заглянуть в разделы 5.3.1, 5.3.2 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 редактора¹⁸ использовать цель

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

для перезагрузки утверждений программы в память, автоматически замещая предыдущие определения. Если использовать `consult` вместо `reconsult`, старая¹⁹ версия утверждений программы останется в памяти наряду с новыми определениями²⁰.

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

¹⁸ предполагается что новая версия файла была сохранена в том же файле

¹⁹ и скорее всего неправильная

²⁰ 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)²¹.

```
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 загружен²²

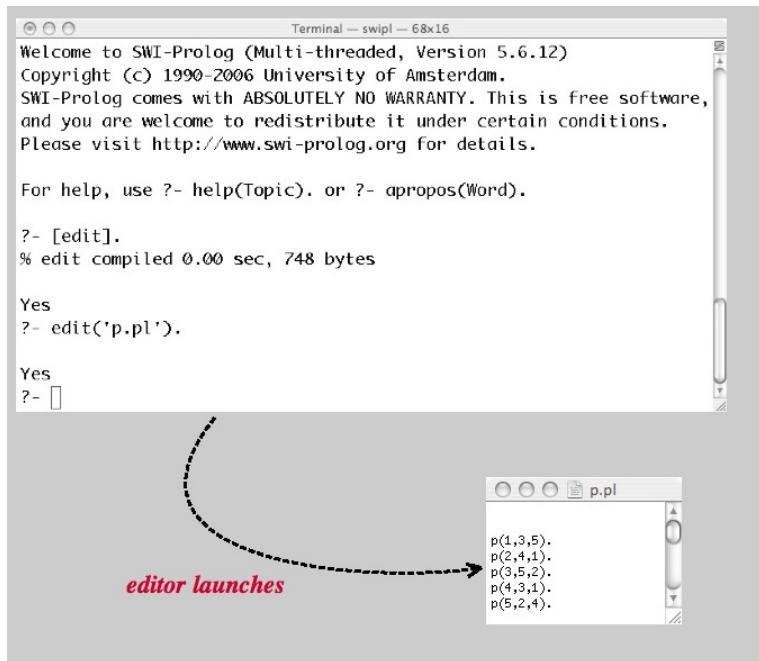
```
?- [edit].
```

```
yes
```

и цель `edit` может быть использована²³

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

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



²¹ это просто пример; мы не используем конкретно **TextEdit**

²² предполагаем локальную *Prolog*-сессию

²³ 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 5.2.4 Проанализируйте как работает редактирование программы. Сначала попробуйте цели

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

и

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

`name/2` описана в разделе [5.4.13](#).

Теперь хороший момент для читателя немного заглянуть вперед и попробовать почитать первые две секции из раздела [5.3](#) “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.

5.2.5 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.

5.2.6 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.

5.2.7 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.

5.2.8 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*.

5.2.9 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.

5.2.10 2.10 Simple I/O

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

5.2.11 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.

5.2.12 2.12 Finding all answers

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

5.2.13 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.

5.2.14 2.14 DFA parser

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

5.2.15 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.

5.2.16 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.

5.2.17 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.

5.2.18 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.

5.2.19 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.

5.3 Как работает *Prolog*

5.3.1 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 5.3.1.1 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 5.3.1.2 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

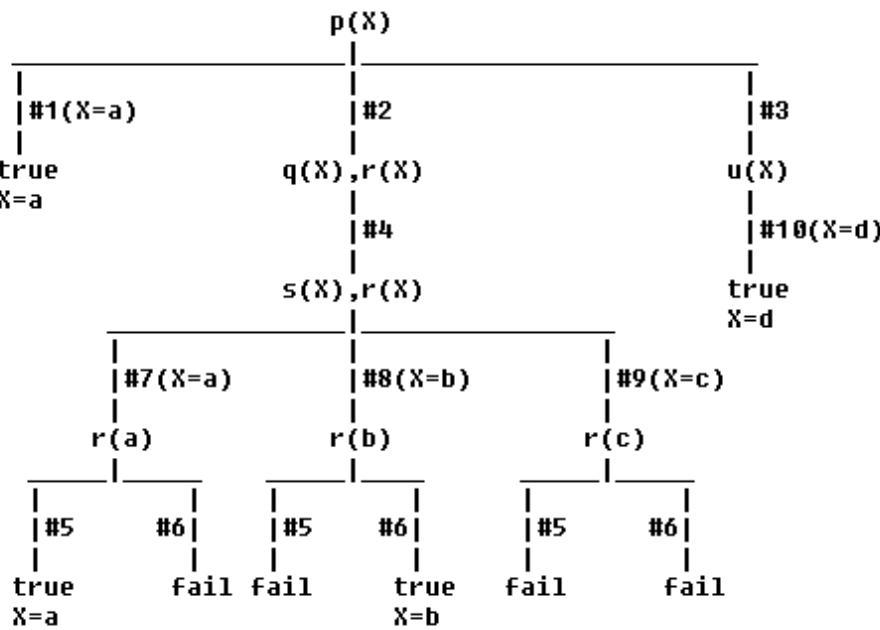
```

```

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.

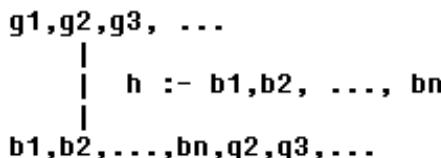


Трассировка упражнения 5.3.1.2 для цели $?-p(X)$ соответствует обходу дерева вывода вглубь. Каждый узел дерева вывода *Prologa* в определенный момент времени становится текущей целью. Аналогично каждый узел — последовательность субцелей. Ребра сразу ниже узла соответствуют доступным выборам замены для текущего узла. Текущий side clause, номер которого отмечает дугу в дереве вывода²⁴, тестируется следующим способом: если самая левая подцель текущего узла²⁵ унифицируется головой side clause²⁵, затем самая левая подцель заменяется телом side clause²⁶. Графически мы можем это показать вот так:

²⁴ отмечена как **g1** в небольшой диаграмме ниже

²⁵ отмечена как **h** в диаграмме

²⁶ **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 значит 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 длядачи ответа на оригинальный запрос.

Унификация термов *Prologa*

Prolog unification matches two Prolog terms T1 and T2 by finding a substitution of variables mapping M such that if M is applied T1 and M is applied to T2 then the results are equal.

For example, Prolog uses unification in order to satisfy equations (T1=T2) ...

```
?- p(X,f(Y),a) = p(a,f(a),Y).
X = a    Y = a
```

```
?- p(X,f(Y),a) = p(a,f(b),Y).
```

No

In the first case the successful substituton is {X/a, Y/b}, and for the second example there is no substitution that would result in equal terms. In some cases the unification does not bind variables to ground terms but result in variables sharing references ...

```
?- p(X,f(Y),a) = p(Z,f(b),a).
X = _G182    Y = b    Z = _G182
```

In this case the unifying substitution is {X/_G182, Y/b, Z/_G182}, and X and Z share reference, as can be illustrated by the next goal ...

```
?- p(X,f(Y),a) = p(Z,f(b),a), X is d.
X = d    Y = b    Z = d
```

{X/_G182, Y/b, Z/_G182} was the most general unifying substitution for the previous goal, and the instance {X/d, Y/b, Z/d} is specialized to satisfy the last goal.

Prolog does not perform an occurs check when binding a variable to another term, in case the other term might also contain the variable. For example (SWI-Prolog) ...

```
?- X=f(X).  
X = f(**)
```

The circular reference is flagged (**) in this example, but the goal does succeed {X/f(f(f(...)))}. However ...

```
?- X=f(X), X=a.  
No
```

The circular reference is checked by the binding, so the goal fails. "a canNOT be unified with f(_Anything)" ...

```
?- a \=f(_).  
Yes
```

Some Prologs have an occurs-check version of unification available for use. For example, in SWI-Prolog ...

```
?- unify_with_occurs_check(X,f(X)).  
No
```

The Prolog goal satisfaction algorithm, which attempts to unify the current goal with the head of a program clause, uses an instance form of the clause which does not share any of the variables in the goal. Thus the occurs-check is not needed for that.

The only possibility for an occurs-check error will arise from the processing of Prolog terms (in user programs) that have unintended circular reference of variables which the programmer believes should lead to failed goals when they occur . Some Prologs might succeed on these circular bindings, some might fail, others may actually continue to record the bindings in an infinite loop, and thus generate a run-time error (out of memory). These rare situations need careful programming.

It is possible to mimic the general unification algorithm in Prolog. But here we present a specialized version of unification, whose computational complexity is linear in the size of the input terms, and simply matches terms left-to-right. The match(+General predicate attempts to match its first argument (which may contain variables) against its second argument (which must be grounded). This little program should be considered just as an illustration, or a programming exercise, although we do know of cogent applications for the LR matching algorithm in situations where general unification is not needed. We would not use match, however, in a Prolog application because built-in unification would be so much faster; we would simply have to ensure that the

presuppositions for match are appropriately checked when built-in unification is used. The reference Apt and Etalle (1993) discusses the question in general regarding how much of general unification is actually NOT needed by Prolog.

```
%%%%%%%%%%%%%%%%
%% match(U,V) :
%%   U may contain variables
%%   V must be ground
%%%%%%%%%%%%%%%
% match a variable with a ground term
match(U,V) :-  
    var(U),  
    ground(V),  
    U = V. % U assigned value V

% match grounded terms
match(U,V) :-  
    ground(U),  
    ground(V),  
    U == V.

% match compound terms
match(U,V) :-  
    \+var(U),  
    ground(V),  
    functor(U,Functor,Arity),  
    functor(V,Functor,Arity),  
    matchargs(U,V,1,Arity).

% match arguments, left-to-right
matchargs(_,_,N,Arity) :-  
    N > Arity.
matchargs(U,V,N,Arity) :-  
    N =< Arity,  
    arg(N,U,ArgU),  
    arg(N,V,ArgV),  
    match(ArgU,ArgV),  
    N1 is N+1,  
    matchargs(U,V,N1,Arity).
```

5.3.2 3.2 Cut

The Prolog **cut** predicate, or **!**, eliminates choices is a Prolog derivation tree. To illustrate, first consider a cut in a goal. For example, consider goal `?-p(X),!.` for the

The cut goal succeeds whenever it is the current goal, AND the derivation tree is trimmed of all other choices on the way back to and including the point in the derivation tree where the cut was introduced into the sequence of goals.

For the previous derivation tree, this means that the branches labeled #2 and #3 are eliminated, and hence the entire subtrees below these two edges are also cut off. This then corresponds to only producing the first answer X=a:

```
?- p(X),!.  
X=a ;  
no
```

Here we tried to get Prolog to find some more answers using ';' but they have already been cut off. Consider also:

```
?- r(X),!,s(Y).  
X=a Y=a ;  
X=a Y=b ;  
X=a Y=c ;  
no
```

Note that there is no backtracking into that first goal. Also,

```
?- r(X), s(Y), !.  
X=a Y=a ;  
no
```

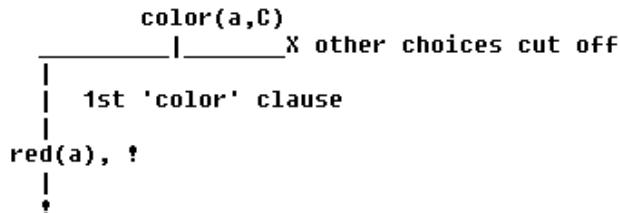
as expected.

Suppose that a cut occurs in the body of the program. The cut rule (above) still applies when the cut appears as a called subgoal. The cut is used in the body of a given clause so as to avoid using clauses appearing after the given clause in the program. To illustrate, consider the following program:

```
part(a). part(b). part(c).  
red(a). black(b).  
color(P,red) :- red(P),!.  
color(P,black) :- black(P),!.  
color(P,unknown).
```

The intention is to determine a color for a part based upon specific stored information or else conclude that the color is 'unknown' otherwise.

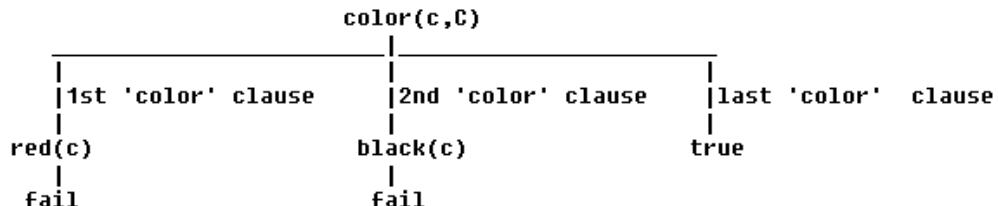
A derivation tree for goal ?- color(a,C) is



which corresponds with

```
?- color(a,C).
C = red
```

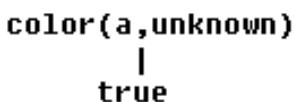
and a derivation tree for goal ?- color(c,C) is



which corresponds with

```
?- color(c,C).
C = unknown
```

The Prolog cut is a procedural device for controlling goal satisfaction. The use of cut affects the meanings of programs. For example, in the 'color' program, the following program clause tree says that 'color(a,unknown)' should be a consequence of the program:



5.3.3 3.3 Meta-interpreters in *Prolog*

5.4 4. Built-in Goals

- 5.4.1 4.1 Utility goals
- 5.4.2 4.2 Universals (true and fail)
- 5.4.3 4.3 Loading *Prolog* programs
- 5.4.4 4.4 Arithmetic goals
- 5.4.5 4.5 Testing types
- 5.4.6 4.6 Equality of *Prolog* terms, unification
- 5.4.7 4.7 Control
- 5.4.8 4.8 Testing for variables
- 5.4.9 4.9 Assert and retract
- 5.4.10 4.10 Binding a variable to a numerical value
- 5.4.11 4.11 Procedural negation, negation as failure
- 5.4.12 4.12 Input/output
- 5.4.13 4.13 *Prolog* terms and clauses as data
- 5.4.14 4.14 *Prolog* operators
- 5.4.15 4.15 Finding all answers

5.5 5. Search in *Prolog*

- 5.5.1 5.1 The A* algorithm in *Prolog*
- 5.5.2 5.2 The 8-puzzle
- 5.5.3 5.3 $\alpha\beta$ search in *Prolog*

5.6 6. Logic Topics

- 5.6.1 6.1 Chapter 6 notes
- 5.6.2 6.2 Positive logic
- 5.6.3 6.3 Convert first-order logic to normal form
- 5.6.4 6.4 A normal rulebase goal interpreter
- 5.6.5 6.5 Evidentiary soundness and completeness

Глава 6

ASTLOG: Язык для анализа синтаксических деревьев

¹ © Roger F. Crew <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 to 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.

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

6.1 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.

6.1.1 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()`; ². 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>)
```

² 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.

6.1.2 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.

6.2 Elements of ASTLOG

Section 6.2 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>

6.2.1 Objects

ASTLOG refers to external objects. Given a Cи/C₊⁺ compiler front end that provides a (C₊⁺) 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₊⁺ 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 constants³.

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.

6.2.2 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

³ “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 6.3.1 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 object⁴, not necessarily the same current object as that which the full term is being matched against. For example, the operand of `strlen` (see 6.2.2) 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)`

⁴ 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()`.

6.2.3 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);

```

6.3 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.

6.3.1 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 6.3.1. 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.

6.3.2 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 6.2.3). 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 6.3.1 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 6.3.2. 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.

6.4 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_{II} 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.

6.5 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.

6.6 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 $i = 1$. Likewise, if `Fk ! Bk ‘ yes :: Fk+1` then we have a $|(k + 1)\text{th}$ hit.

When we have a $(k + 1)\text{th}$ hit. The semantic function `mgu(B; f; [[t1;:::;tn]]; [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 ; [[t01]]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” [?] 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 7

Warren's Abstract Machine Абстрактная машина Варрена

¹

© Hassan Aït-Kaci <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 версия² 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

¹ © <http://wambook.sourceforge.net/>

² 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*³ для представления *термов Prologa*, метод *общих структур*⁴, 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-

³ structure copying

⁴ 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

Реализация машины вывода на C_+^+

В перевод книги Варрена мной⁵ добавлен пример реализации виртуальной машины вывода на C_+^+ . Исходные тексты находятся в каталоге [prolog/warren/](#). Для вставки отдельных частей исходника по ходу книги полные файлы **hpp.hpp** и **cpr.cpp** разделены на отдельные небольшие фрагменты в каталогах **hpp/** и **cpr/**.

7.0.1 Makefile

Файл сборки **prolog/warren/Makefile** содержит не только типовые заклинания для *лексической* программы, использующей связку Flex/Bison для парсера входного языка, но и скрипты склейки файлов исходников из частей в каталогах **ypp/ lpp/ hpp/ cpp/ mk/**.

Makefile: запуск программы и генерация **log.log**

```
log.log: ./exe.exe src.src
    ./exe.exe < src.src > log.log && tail $(TAIL) log.log
```

Makefile: типовой блок компиляции *лексической* программы

```
C = cpp.cpp ypp.tab.cpp lex.yy.c
H = hpp.hpp ypp.tab.hpp
CXXFLAGS = -std=gnu++11 -DMODULE=\$(notdir \$(CURDIR)) \
./exe.exe: \$(C) \$(H) Makefile
\$(CXX) \$(CXXFLAGS) -o \$(C)
```

Makefile: генерация кода *синтаксического анализатора*

```
ypp.tab.cpp: ypp.ypp
bison \$<
```

Makefile: генерация кода *лексера*

```
lex.yy.c: lpp.lpp
flex \$<
```

⁵ <dponyatov@gmail.com>

7.0.2_hpp.hpp

hpp.hpp: обертка одиночного #include

```
#ifndef _H_WARREN  
#define _H_WARREN
```

hpp.hpp

```
#endif // _H_WARREN
```

hpp.hpp: типовые #include

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

hpp.hpp: базовый класс для структур WAM

```
struct WAM {  
    string val;  
    WAM(string);  
    virtual string dump(int=0);  
};
```

7.0.3_cpp.cpp

cpp.cpp

```
#include "hpp.hpp"  
int main() { return yyparse(); }
```

callback для обработки ошибок синтаксического анализатора

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

реализация методов WAM

```
WAM::WAM(string V) { val=V; }
```

```
string WAM::dump(int depth) { return "<" + val + ">"; }
```

7.0.4 урр.урр/lpp.lpp: синтаксический анализатор

Ввод входных данных и их синтаксис немного отличается от классического *Prolog*. Исходные тексты программ и команды *Prolog*-машины читаются с `stdin` аналогично другим скриптовым языкам, и не поддерживается традиционная для *Prolog* кнопка `[;]`. Для ускорения (и упрощения) синтаксического разбора входного потока была использована классическая связка **flex/bison**.

7.1 Введение

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*⁶. 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.

7.1.1 Существующая литература

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 Варрена [?], 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 Варрена. 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. Warren⁷ [?]. В этой работе можно найти описание техник компиляции *Prologa* похожие на принципы WAM⁸. Тем не менее мы считаем что эта похвальная попытка все еще страдает от нескольких недостатков, если его

⁶ 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”

⁷ 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.

⁸ chap.9

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 свердетализированные 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-predicate⁹.

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.

7.1.2 Эта книга

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

⁹ chap.10

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 7.2, we consider unification alone. Then, we look at flat resolution (that is, Prolog without

backtracking) in 7.3. Following that, we turn to disjunctive definitions and backtracking in 7.4. At that point, we will have a complete, albeit naïve, design for pure Prolog. In 7.5, 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 7.7. As for notation, we implicitly use the syntax of so-called Edinburgh Prolog (see, for instance, [?]), which we also recall in that appendix. Finally, 7.8 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.

7.2 Унификация — ясно и просто

Напомним что терм (первого порядка) — **переменная** (задается большой буквой в начале имени), **константа** (задается маленькой буквой в начале имени) или **терм** — структура вида $f(t_1, \dots, t_n)$, где f символ называемый **функцией** (записывается аналогично константе, с маленькой буквы), а элементы t_i тоже термы первого порядка — **субтермы**. Число субтермов для данного функционара предопределено, и называется **арностью** функционара. Для обеспечения возможности использовать один и тот же символ с разной арностью, мы должны использовать запись f/n , что обозначает функцию f с арностью n . Таким образом, два функционара равны только в том случае, если они имеют **одинаковые символ f и арность n** . Разрешая случай $n = 0$ можно рассматривать константу как особый случай терма: константе c соответствует функция $c/0$ с нулевой арностью.

Мы рассмотрим очень простой низкоуровневый¹⁰ язык \mathcal{L}_0 . На этом языке мы можем описать два вида объектов: **терм программы** и **терм запроса**. Оба этих вида запросов являются термами первого порядка, но не переменными. Семантика \mathcal{L}_0 равносильна вычислению самого общего унифициатора программы или запроса. Что касается синтаксиса, \mathcal{L}_0 будет описывать программу как t и запрос как $?-t$ где t является термом. Область видимости переменных ограничена термом программы/запроса. Таким образом, **значение программы не зависит от имен ее переменных**. Интерпретатор для \mathcal{L}_0 будет использовать определенное представление данных для термов и использовать алгоритм унификации для ее операционной семантики. Затем мы опишем $\mathcal{M}_0 = (\mathcal{D}_0, \mathcal{I}_0)$, дизайн абстрактной машины для \mathcal{L}_0 содержащий представление данных \mathcal{D}_0 , над которыми выполняется множество \mathcal{I}_0 машинных инструкций.

Идея достаточно проста: имея определенных программный терм p , мы можем выполнить любой запрос $?-q$, и выполнение запроса завершится с ошибкой ес-

¹⁰ IL – Intermediate Language

ли p и q не унифицируются, или будет успешным с привязкой переменных в q полученной при унификации запроса с p .

7.2.1 Представление термов

Для начала давайте определим внутреннее представление термов в языке \mathcal{L}_0 . Мы будем использовать глобальный блок хранения данных в форме адресуемой **кучи** который мы назовем **HEAP**: массив ячеек данных. Адресом ячейки в куче является индекс элемента массива **HEAP**.

Для представления произвольных термов в **HEAP** будет достаточно закодировать переменные и “структуры” имеющие форму $f(@_1, \dots, @_n)$ где f/n функтор и $@_i$ ссылки на адреса кучи для n субтермов. Таким образом существует два вида данных, хранимых в куче: переменные и структуры термов. Явно заданные **тэги**, появляющиеся как часть внутреннего формата ячеек кучи, будут использоваться для различия между этими двумя типами данных.¹¹

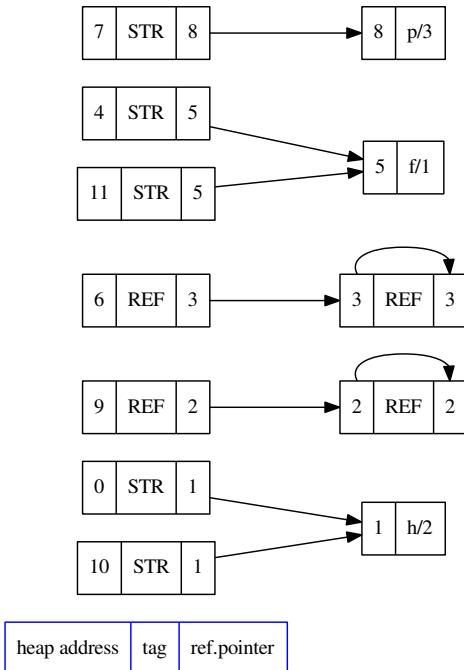
Переменная будет идентифицироваться как указатель, и представляться как одна ячейка кучи, так что мы должны говорить о **ячейках переменных**. Ячейка переменной отмечается тэгом **REF**, и обозначается как $\langle \text{REF}, k \rangle$ где k адрес хранения, т.е. индекс в **HEAP**. Этот механизм предназначен для облегчения связывания переменных через установление ссылки на терм в переменной, которая связывается с этим термом. Таким образом при связывании переменной адресная часть **REF**-ячейки получает значение соответствующего адреса терма. Соглашение о представлении **несвязанной переменной** — адресная часть **REF**-ячейки указывает на саму переменную. Таким образом **несвязанные переменные представляются REF-ячейкой со ссылкой на саму себя**.

Структуры — термы не являющиеся переменными. Формат кучи для представления структуры $f(t_1, \dots, t_n)$ содержит $n + 2$ ячеек кучи. Первые две ячейки не обязательно смежные. По сути первая из этих двух ячеек выступает в роли сортированного указателя на вторую ячейку, и в то же время сама выступает как представление функтора f/n .¹² Остальные n ячеек предназначены для упорядоченного хранения ссылок на корни соответствующих n субтермов.

Детальнее, первая из $n + 2$ ячеек представляющая терм $f(t_1, \dots, t_n)$ форматирована как тэгированная **структурная ячейка**, которую можно записать как $\langle \text{STR}, k \rangle$, содержит тэг **STR** и указатель k на **ячейку функтора**, хранящую представление функтора f/n . Важно отметить что **непосредственно за ячейкой функтора в смежных адресах всегда следуют n структурных ячеек, представляющих каждый из t_i субтермов**. Так что если $\text{HEAP}[k] = f/n$ то $\text{HEAP}[k+1]$ будет ссылаться на первый субтерм t_1 , а $\text{HEAP}[k+n]$ будет ссылаться на последний субтерм t_n .

¹¹ интересно рассмотреть расширение тэгирования для реализации ООП и динамического контроля типов

¹² причина использования этой кажущейся странной косвенной адресации — реализация разделяемых структур (structure sharing) — будет вскоре ясна



Фиг. 2.1: Представление кучи для терма $p(Z, h(Z, W), f(W))$

0	STR	1
1	$h/2$	
2	REF	2
3	REF	3
4	STR	5
5	$f/1$	
6	REF	3
7	STR	8
8	$p/3$	
9	REF	2
10	STR	1
11	STR	5

Например, рассмотрим представление кучи для терма $p(Z, h(Z, W), f(W))$, начальная ячейка которого находится по адресу 7 (иллюстрация 7.2.1). Отметим что **для каждой** непривязанной переменной существует только одно вхождение, представленное как **REF**-ячейка, в то время как другие ее вхождения в исходный терм представляются как ссылки на первое вхождение ($Z = \text{HEAP}[2]$, $W = \text{HEAP}[3]$). Также обратите внимание что за структурными ячейками по адресам 0, 4 и 7 **сразу** следуют их ячейки функторов, но это не так для адресов

7.2.2 Компиляция \mathcal{L}_0 запросов

Согласно операционной семантике \mathcal{L}_0 обработка запроса состоит из подготовки в решению уравнения с одной стороны. А именно, терм запроса q транслируется в последовательность инструкций, целью которой является построение экземпляра q на куче из текстового представления q . Таким образом, из-за древовидной структуры терма и множественных вхождениях переменных, необходимо, чтобы при обработке части терма где-то временно сохранялись части терма, которые еще предстоит обработать, или переменные которые могут встретиться еще раз далее по ходу работы. Для этой цели виртуальная машина M_0 наделена достаточным количеством (изменяемых) **регистров** X_1, X_2, \dots которые используются для временного хранения данных кучи по мере создания промежуточных термов. Таким образом, содержимое каждого регистра должно иметь формат ячейки кучи. Эти изменяемые регистры выделяются для терма по мере доступности, так что (1) регистр X_1 всегда распределяется для охватывающего терма, и (2) тот же регистр распределяется для всех вхождений определенной переменной. Например регистры для переменных терма $p(Z, h(Z, W), f(W))$ распределяются

$$\begin{aligned} X_1 &= p(X_2, X_3, X_4) \\ X_2 &= Z \\ X_3 &= h(X_2, X_5) \\ X_4 &= f(X_5) \\ X_5 &= W \end{aligned}$$

Это равносильно тому что терм рассматривается как сплющенный конъюктивный набор уравнений в форме $X_i = X$ или $X_i = f(X_{i_1}, \dots, X_{i_n})$, ($n \geq 0$) , где члены X_i различные новые имена переменных. Есть два последствия распределения регистров: (1) все внешние имена переменных (такие как Z and W в нашем примере) могут быть забыты; и (2) терм запроса может быть трансформирован в его **сплющенную форму**, т.е. последовательность назначений регистров только в форме $X_i = f(X_{i_1}, \dots, X_{i_n})$. Эта форма — то, что контролирует построение представления терма в куче. Таким образом, чтобы генерация кода слева направо была хорошо обоснована, необходимо сформировать сплющенный терм запроса, так чтобы гарантировать что **имена регистров не могут использоваться в правых частях присвоений (например как субтерм) до их инициализации**¹³. Например сплющенная форма терма запроса $p(Z, h(Z, W), f(W))$ это последовательность $X_3 = h(X_2, X_5)$, $X_4 = f(X_5)$, $X_1 = p(X_2, X_3, X_4)$ ¹⁴.

Сканируя сплющенный терм запроса слева направо, каждый компонент в форме $X_i = f(X_{i_1}, \dots, X_{i_n})$ токенизируется в последовательность $X_i = f/n, X_{i_1}, \dots, X_{i_n}$

¹³ if it has one (viz., being the lefthand side)

¹⁴ исключена привязка переменных на регистры X_2, X_5

такую что после регистра ассоциированного с п-арным функтором идет последовательность n имен регистров. Так что в потоке таких токенов полученных в результате токенизации полного сплющенного терма, существует три вида элементов для обработки:

1. регистр ассоциированный со структурным функтором;
2. регистр-аргумент который не был нигде равнее встречен в потоке;
3. регистр-аргумент который уже был упомянут в потоке.

Из такого потока легко получить представление кучи используя метод управляемого потоком токенов синтеза. Для реализации этого нужно выполнить соответствующие действия для каждого типа токенов:

1. создать на куче новую ячейку STR (и примыкающий функтор) и скопировать эту ячейку в указанный регистр;
2. создать на куче новую ячейку REF содержащую собственный адрес, и скопировать ее в указанный регистр;
3. создать на куче новую ячейку и копировать в нее значение регистра.

Each of these three actions specifies the effect of respective instructions of the machine M_0 that we note:

1. put structure f in X_i
2. set variable X_i
3. set value X_i

respectively.

From the preceding considerations, it has become clear that the heap is implicitly used as a stack for building terms. Namely, term parts being constructed are incremental piled on top of what already exists in the heap. Therefore, it is necessary to keep the address of the next free cell in the heap somewhere, precisely as for a stack.¹⁵ Adding to M a global register H containing at all times the next available address on the heap, these three instructions are given explicitly in Figure 2.2. For example, given that registers are allocated as above, the sequence of instructions to build the query term pZh is shown in Figure 2.3.

Exercise 2.1 Verify that the effect of executing the sequence of instructions shown in Figure 2.3 (starting with H) does indeed yield a correct heap representation for the term pZh the one shown earlier as Figure 2.1, in fact.

¹⁵ As a matter of fact, in [War83], Warren refers to the heap as the **global stack**.

7.2.3 Compiling L_0 programs

Compiling a program term p is just a bit trickier, although not by much. Observe that it assumes that a query $?-q$ will have built a term on the heap and set register X to contain its address. Thus, unifying q to p can proceed by following the term structure already present in X as long as it matches functor for functor the structure of p . The only complication is that when an unbound REF cell is encountered in the query term in the heap, then it is to be bound to a new term that is built on the heap as an exemplar of the corresponding subterm in p . Therefore, an L program functions in two modes: a read mode in which data on the heap is matched against, and a write mode in which a term is built on the heap exactly as is a query term.

Figure 2.2: M machine instructions for query terms

```
put structure f n  
Xi
```

Figure 2.3: Compiled code for L query $?-pZh$

```
put structure h  
X % ?-X  
h  
set variable X  
set variable X  
put structure f  
f  
set value  
put structure p  
p  
set value X %  
set value X % X  
set value X
```

As with queries, register allocation precedes translation of the textual form of a program term into a machine instruction sequence. For example, the following registers are allocated to program term $p\text{fh}y$

```
x1  
..  
x7
```

Recall that compiling a query necessitates ordering its flattened form in such a way as to build a term once its subterms have been built. Here, the situation is reversed because query data from the heap are assumed available, even if only in the form of unbound REF cells. Hence, a program term's flattened form follows a top-down order. For example, the program term $p\text{fh}y$ is put into the flattened sequence: $X1..x7$

As with query compiling, the flattened form of a program is tokenized for left-to-right processing and generates three kinds of machine instructions depending on whether is met:

1. a register associated with a structure functor;
2. a first-seen register argument; or,
3. an already-seen register argument.

These instructions are,

1. get structure f n Xi
2. unify variable Xi
3. unify value Xi

respectively.

Figure 2.4: Compiled code for L program pfhY

Taking for example the program term pf , the M machine instructions shown in Figure 2.4 are generated. Each of the two unify instructions functions in two modes depending on whether a term is to be matched from, or being built on, the heap. For building (write mode), the work to be done is exactly that of the two set query instructions of Figure 2.2. For matching (read mode), these instructions seek to recognize data from the heap as those of the term at corresponding positions, proceeding if successful and failing otherwise. In L, failure aborts all further work. In read mode, these instructions set a global register S to contain at all times the heap address of the next subterm to be matched.

Variable binding creates the possibility that reference chains may be formed. Therefore dereferencing is performed by a function deref which, when applied to a store address, follows a possible reference chain until it reaches either an unbound REF cell or a non-REF cell, the address of which it returns. The effect of dereferencing is none other than composing variable substitutions. Its definition is given in Figure 2.5. We shall use the generic notation STORE[a] to denote the contents of a term data cell at address a (whether heap, X register, or any other global structure, yet to be introduced, containing term data cells). We shall use specific area notation (e.g., HEAP[a]) whenever we want to emphasize that the address a must necessarily lie within that area.

Figure 2.5: The deref operation

```
function deref
then return deref
else return a
end deref
F
```

Mode is set by get structure f n Xi as follows: if the dereferenced value of Xi is an unbound REF cell, then it is bound to a new STR cell pointing to f n pushed onto the heap and mode is set to write; otherwise, if it is an STR cell pointing to functor f n, then register S is set to the heap address following that functor cell's and mode is set to read. If it is not an STR cell or if the functor is not f n, the program fails. Similarly, in read mode, unify variable Xi sets register Xi to the contents of the heap at address S; in write mode, a new unbound REF cell is pushed on the heap and copied into Xi. In both modes, S is then incremented by one. As for unify value Xi, in read mode, the value of Xi must be unified with the heap term at address S; in write mode, a new cell is pushed onto the heap and set to the value of register Xi. Again, in either mode, S is incremented. All three instructions are expressed explicitly in Figure 2.6. In

In the definition of get structure f n Xi, we write bindaddr H to effectuate the binding of the heap cell rather than HEAP[addr] h REF H i for reasons that will become clear later. The bind operation is performed on two store addresses, at least one of which is that of an unbound REF cell. Its effect, for now, is to bind the unbound one to the other—i.e., change the data field of the unbound REF cell to contain the address of the other cell. In the case where both are unbound, the binding direction is chosen arbitrarily. Later, this will change as a correctness-preserving measure in order to accommodate an optimization. Also, we will see that bind is the logical place, when backtracking needs to be considered, for recording effects to be undone upon failure (see Chapter 4, and appendix Section B.2 on Page 113). If wished, bind may also be made to perform the occurs-check test in order to prevent formation of cyclic terms (by failing at that point). However, the occurs-check test is omitted in most actual Prolog implementations in order not to impede performance.

Figure 2.6: M machine instructions for programs

We must also explicate the unify operation used in the matching phase (in read mode). It is a unification algorithm based on the UNION/FIND method [AHU74], where variable substitutions are built, applied, and composed through dereference pointers. In M (and in all later machines that will be considered here), this unification operation is performed on a pair of store addresses. It uses a global dynamic structure, an array of store addresses, as a unification stack (called PDL, for Push-Down List). The unification operation is defined as shown in Figure 2.7, where empty, push, and pop are the expected stack operations.

Exercise 2.2 Give heap representations for the terms $fXgX$ and $f b Y$. Let a and b be their respective heap addresses, and let aX and aY be the heap addresses corresponding to variables X and Y , respectively. Trace the effects of executing $\text{unify } a$, verifying that it terminates with the eventual dereferenced bindings from aX and aY corresponding to $X b$ and $Y g b a$.

Exercise 2.3 Verify that the effect of executing the sequence of instructions shown in Figure 2.4 right after that in Figure 2.3 produces the MGU of the terms $p \text{ Z h ZW f W}$ and $p \text{ f X h Yf a Y}$. That is, the (dereferenced) bindings corresponding to W f a , X f a , Y f f a , Z f f a .

Exercise 2.4 What are the respective sequences of M instructions for L query term $?-p \text{ f X h Y f a Y}$ and program term $p \text{ Z h ZW f W}$?

Exercise 2.5 After doing Exercise 2.4, verify that the effect of executing the sequence you produced yields the same solution as that of Exercise 2.3.

7.2.4 Argument registers

Since we have in mind to use unification in Prolog for procedure invocation, we can introduce a distinction between atoms (terms whose functor is a predicate) and terms (arguments to a predicate). We thus extend L into a language L₀ similar to L but where a program may be a set of first-order atoms each defining at most one fact per predicate name. Thus, in the context of such a program, execution of a query connects to the appropriate definition to use for solving a given unification equation, or fails if none exists for the predicate invoked.

Figure 2.7: The *unify* operation

The set of instructions I contains all those in I. In M, compiled code is stored in a code area (CODE), an addressable array of data words, each containing a possibly labeled instruction over one or more memory words consisting of an opcode followed by operands. For convenience, the size of an instruction stored at address a (i.e., CODE[a]) will be assumed given by the expression instruction sizea. Labels are symbolic entry points into the code area that may be used as operands of instructions for transferring control to the code labeled accordingly. Therefore, there is no need to store a procedure name in the heap as it denotes a key into a compiled instruction sequence. Thus, a new instruction call pn can be used to pass control over to the instruction labeled with pn, or fail if none such exists.

A global register P is always set to contain the address of the next instruction to execute (an instruction counter). The standard execution order of instructions is sequential. Unless failure occurs, most machine instructions (like all those seen before) are implicitly assumed, to increment P by an appropriate offset in the code area as an ultimate action. This offset is the size of the instruction at address P. However, some instructions have for purpose to break the sequential order of execution or to connect to some other instruction at the end of a sequence. These instructions are called control instructions as they typically set P in a non-standard way. This is the case of call pn,

whose explicit effect, in the machine M0, is:

callpnPpnwhe

where the notation pn stands for the address in the code area of instruction labeled pn. If the procedure pn is not defined (i.e., if that address is not allocated in the code area), a unification failure occurs and overall execution aborts. W

We also introduce another control instruction, proceed, which indicates the end of a fact's instruction sequence. These two new control instructions' effects are trivial for now, and they will be elaborated later. For our present purposes, it is sufficient that proceed be construed as a no-op (i.e., just a code terminator), and call pn as an unconditional "jump" to the start address of the instruction sequence for program term with functor pn.

Having eliminated predicate symbols from the heap, the unification problem between fact and query terms amounts to solving, not one, but many equations, simultaneously. Namely, there are as many term roots in a given fact or query as there are arguments to the corresponding predicate. Therefore, we must organize registers quite specifically so as to reflect this situation. As we privileged X before to denote the (single) term root, we generalize the convention to registers X

to X_n which will now always refer to the first to n -th arguments of a fact or query atom. In other words, registers $X_i X_n$ are systematically allocated to term roots of an n-ary predicate's arguments. To emphasize this, we use a conspicuous notation, writing a register A_i rather than X_i when it is being used as an argument of an atom. In that case, we refer to that register as an argument register. Otherwise, where register X_i is not used as an argument register, it is written X_i , as usual. Note that this is just notation as the A_i 's are not new registers but the same old X_i 's used thus far. For example, registers are now allocated for the variables of the atom $pZ hZW f W$ as follows:

A_1

A_2

A_3

$A_4 = W$

Observe also that a new situation arises now as variables can be arguments and thus must be handled as roots. Therefore, provision must be made for variables to be loaded into, or extracted from, argument registers for queries and facts, respectively. As before, the necessary instructions correspond to when a variable argument is a first or later occurrence, either in a query or a fact. In a query,

1. the first occurrence of a variable in i -th argument position pushes a new unbound REF cell onto the heap and copies it into that variable's register as well as argument register A_i ; and,
2. a later occurrence copies its value into argument register A_i . Whereas, in a fact,

3. the first occurrence of a variable in i-th argument position sets it to the value of argument register A_i ; and,

4. a later occurrence unifies it with the value of A_i .

The corresponding instructions are, respectively:

1. put variable $X_n A_i$

2. put value $X_n A_i$

3. get variable $X_n A_i$

4. get value $X_n A_i$

and are given explicitly in Figure 2.8. For example, Figure 2.9 shows code generated for query $?- pZ hZW f W$ and Figure 2.10 that for fact $pF X hY f a Y$.

Figure 2.8: M_0 instructions for variable arguments

Exercise 2.6 Verify that the effect of executing the sequence of M instructions shown in Figure 2.9 produces the same heap representation as that produced by the M code of Figure 2.3 (see Exercise 2.1).

Exercise 2.7 Verify that the effect of executing the sequence of M instructions shown in Figure 2.10 right after that in Figure 2.9 produces the MGU of the terms $p Z h ZW f W$ and $p f X h Y f a Y$. That is, the binding $W f a , X f a , Y f f a , Z f f a$.

Exercise 2.8 What are the respective sequences of M_0 instructions for L query term $?- p f X h Y f a Y$ and L program term $p Z h ZW f W ?$

Exercise 2.9 After doing Exercise 2.8, verify that the effect of executing the sequence you produced yields the same solution as that of Exercise 2.7.

Figure 2.9: Argument registers for L query $?- pZ hZW f W$

Figure 2.10: Argument registers for L0 fact $pF X hY f a Y$

7.3 Flat Resolution

We now extend the language L_0 into a language L where procedures are no longer reduced only to facts but may also have bodies. A body defines a procedure as a conjunctive sequence of atoms. Said otherwise, L is Prolog without backtracking.

An L program is a set of procedure definitions or (definite) clauses, at most one per predicate name, of the form ' $a :- a_0 \dots a_n$ ' where $n \geq 0$ and the a_i 's are atoms. As

before, when $n = 0$, the clause is called a fact and written without the ‘`:`’ implication symbol. When $n > 0$, the clause is called a rule, the atom a is called its head, the sequence of atoms $a_0 \dots a_{n-1}$ is called its body and atoms composing this body are called goals. A rule with exactly one body goal is called a chain (rule). Other rules are called deep rules. L queries are sequences of goals, of the form ‘`?-g0000gk0`’ where $k \geq 0$. When $k = 0$, the query is called the empty query. As in Prolog, the scope of variables is limited to the clause or query in which they appear.

Executing a query ‘`?-g0000gk0`’ in the context of a program made up of a set of procedure-defining clauses consists of repeated application of leftmost resolution until the empty query, or failure, is obtained. Leftmost resolution amounts to unifying the goal g_0 with its definition’s head (or failing if none exists) and, if this succeeds, executing the query resulting from replacing g_0 by its definition body, variables in scope bearing the binding side-effects of unification. Thus, executing a query in L either terminates with success (i.e., it simplifies into the empty query), or terminates with failure, or never terminates. The “result” of an L query whose execution terminates with success is the (dereferenced) binding of its original variables after termination.

Note that a clause with a non-empty body can be viewed in fact as a conditional query. That is, it behaves as a query provided that its head successfully unifies with a predicate definition. Facts merely verify this condition, adding nothing new to the query but a contingent binding constraint. Thus, as a first approximation, since an L query (resp., clause body) is a conjunctive sequence of atoms interpreted as procedure calls with unification as argument passing, instructions for it may simply be the concatenation of the compiled code of each goal as an L0 query making it up. As for a clause head, since the semantics requires that it retrieves arguments by unification as did facts in L0, instructions for L0’s fact unification are clearly sufficient.

Therefore, M0 unification instructions can be used for L clauses, but with two measures of caution: one concerning continuation of execution of a goal sequence, and one meant to avoid conflicting use of argument registers.

7.3.1 Facts

Let us first only consider L facts. Note that L0 is all contained in L. Therefore, it is natural to expect that the exact same compilation scheme for facts carries over untouched from L0 to L. This is true up to a wee detail regarding the proceed instruction. It must be made to continue execution, after successfully returning from a call to a fact, back to the instruction in the goal sequence following the call. To do this correctly, we will use another global register CP, along with P, set to contain the address (in the code area) of the next instruction to follow up with upon successful return from a call (i.e., set to P instruction size P at procedure call time). Then, having exited the called procedure’s code sequence, execution could thus be resumed as indicated by CP. Thus, for L’s facts, we need to alter the effect of M0’s call pn to:cal

$pnCPPinstructionsizeP0$

and that of proceed to:

PCP

As before, when the procedure pn is not defined, execution fails. In summary, with the simple foregoing adjustment, L facts are translated exactly as were L0 facts.

7.3.2 Rules and queries

We now must think about translating rules. A query is a particular case of a rule in the sense that it is one with no head. It is translated exactly the same way, but without the instructions for the missing head. The idea is to use L0's instructions, treating the head as a fact, and each goal in the body as an L0 query term in sequence; that is, roughly translate a rule 'p0000 :- p00000000pn00000' following the pattern:

```
get arguments of p
put arguments of p0
call p0
.
.
.
put arguments of pn
call pn
```

Here, in addition to ensuring correct continuation of execution, we must arrange for correct use of argument registers. Indeed, since the same registers are used by each goal in a query or body sequence to pass its arguments to the procedure it invokes, variables that occur in many different goals in the scope of the sequence need to be protected from the side effects of put instructions. For example, consider the rule 'pX Y 0 :- qX Z0 rZ Y 00'. If the variables YZ were allowed to be accessible only from an argument register, no guarantee could be made that they still would be after performing the unifications required in executing the body of p.

Therefore, it is necessary that variables of this kind be saved in an environment associated with each activation of the procedure they appear in. Variables which occur in more than one body goal are dubbed permanent as they have to outlive the procedure call where they first appear. All other variables in a scope that are not permanent are called temporary. We shall denote a permanent variable as Y_i , and use X_i as before for temporary variables. To determine whether a variable is permanent or temporary in a rule, the head atom is considered to be part of the first body goal. This is because get and unify instructions do not load registers for further processing. Thus, the variable X in the example above is temporary as it does not occur in more than one goal in the body (i.e., it is not affected by more than one goal's put instructions).

Clearly, permanent variables behave like conventional local variables in a procedure. The situation is therefore quite familiar. As is customary in programming languages, we protect a procedure's local variables by maintaining a run-time stack of procedure

activation frames in which to save information needed for the correct execution of what remains to be done after returning from a procedure call. We call such a frame an environment frame. We will keep the address of the latest environment on top of the stack in a global register E.¹⁶

As for continuation of execution, the situation for rules is not as simple as that for facts. Indeed, since a rule serves to invoke further procedures in its body, the value of the program continuation register CP, which was saved at the point of its call, will be overwritten. Therefore, it is necessary to preserve continuation information by saving the value of CP along with permanent variables.

Let us recapitulate: M is an augmentation of M0 with the addition of a new data area, along with the heap (HEAP), the code area (CODE), and the push-down list (PDL). It is called the stack (STACK) and will contain procedure activation frames. Stack frames are called environments. An environment is pushed onto STACK upon a (non-fact) procedure entry call, and popped from STACK upon return. Thus, an allocate/deallocate pair of instructions must bracket the code generated for a rule in order to create and discard, respectively, such environment frames on the stack. In addition, deallocate being the ultimate instruction of the rule, it must connect to the appropriate next instruction as indicated by the continuation pointer that had been saved upon entry in the environment being discarded.

Since the size of an environment varies with each procedure in function of its number of permanent variables, the stack is organized as a linked list through a continuation environment slot; i.e., a cell in each environment frame bearing the stack index of the environment previously pushed onto the stack.

To sum up, two new I instructions for M are added to the ones we defined for I0:

1. allocate
2. deallocate

with effect, respectively:

1. to allocate a new environment on the stack, setting its continuation environment field to the current value of E, and its continuation point field to that of CP; and,
2. to remove the environment frame at stack location E from the stack and proceed, resetting P to the value of its CP field and E to the value of its CE field.

To have proper effect, an allocate instruction needs to have access to the size of the current environment in order to increment the value of E by the right stack offset. The necessary piece of information is a function of the calling clause (i.e., the number of permanent variables occurring in the calling clause). Therefore, it is easily statically available at the time the code for the calling clause is generated. Now, the problem is to transmit this information to the called procedure that, if defined as a rule (i.e., starting with an allocate), will need it dynamically, depending on which clause calls it. A simple solution is to save this offset in the calling clause's environment frame from where it can be retrieved by a callee that needs it. Hence, in M, an additional slot in an environment is set by allocate to contain the number of permanent variables in the

¹⁶ In [War83], this stack is called the local stack to distinguish it from the global stack (see Footnote 1 at the bottom of Page 13).

clause in question.

Summing up again, an M stack environment frame contains:

1. the address in the code area of the next instruction to execute upon (successful) return from the invoked procedure;
2. the stack address of the previous environment to reinstate upon return (i.e., where to pop the stack to);
3. the offset of this frame on the stack (the number of permanent variables); and,
4. as many cells as there are permanent variables in the body of the invoked procedure (possibly none).

Such an M environment frame pushed on top of the stack looks thus:

ECEcontinuationenvironment

This necessitates giving allocate an explicit argument that is the number of permanent variables of the rule at hand, such that, in M:

allocateN =

Similarly, the explicit definition of M's deallocate is:

deallocate =

With this being set up, the general translation scheme into M instructions for an L rule ‘p0000 :- p0000000pn00000’ with N permanent variables will follow the pattern:

pallocateN

For example, for L clause ‘pX Y 0 :- qX Z0 rZ Y 00’, the corresponding M code is shown in Figure 3.1.

Figure 3.1: M machine code for rule pX Y 0 :- qX Z0 rZ Y 00

Exercise 3.1 GiveM code for L facts qa0 b and rb0 c and L query ?-pU0 V , then trace the code shown in Figure 3.1 and verify that the solution produced is U a0 V c. C

7.4 Prolog

The language L (resp., the machine M) corresponds to pure Prolog, as it extends the language L (resp., the machine M) to allow disjunctive definitions. As in L, an L program is a set of procedure definitions. In L, a definition is an ordered sequence of clauses (i.e., a sequence of facts or rules) consisting of all and only those whose head atoms share the same predicate name. That name is the name of the procedure specified

by the definition. L queries are the same as those of L. The semantics of L operates using top-down leftmost resolution, an approximation of SLD resolution. Thus, in L, a failure of unification no longer yields irrevocable abortion of execution but considers alternative choices of clauses in the order in which they appear in definitions. This is done by chronological backtracking; i.e., the latest choice at the moment of failure is reexamined first.

It is necessary to alter M's design so as to save the state of computation at each procedure call offering alternatives to restore upon backtracking to this point of choice. We call such a state a choice point. We thus need to analyze what information must be saved as a choice point in order to create a record (a choice point frame) wherefrom a correct state of computation can be restored to offer another alternative, with all effects of the failed computation undone. Note that choice point frames must be organized as a stack (just like environments) in order to reflect the compounding of alternatives as each choice point spawns potentially more alternatives to try in sequence.

To distinguish the two stacks, let us call the environment stack the AND-stack and the choice point stack the OR-stack. As with the AND-stack, we organize the OR-stack as a linked list. The head of this list always corresponds to the latest choice point, and will be kept in a new global register B, such that upon failure, computation is made to resume from the state recovered from the choice point frame indicated by B. When the latest frame offers no more alternatives, it is popped off the OR-stack by resetting B to its predecessor if one exists, otherwise computation fails terminally.

Clearly, if a definition contains only one clause, there is no need to create a choice point frame, exactly as was the case in M. For definitions with more than one alternative, a choice point frame is created by the first alternative; then, it is updated (as far as which alternative to try next) by intermediate (but non ultimate) alternatives; finally, it is discarded by the last alternative.

7.4.1 Environment protection

Before we go into the details of what exactly constitutes a choice frame, we must ponder carefully the interaction between the AND-stack and the OR-stack. As long as we considered (deterministic) L program definitions, it was clearly safe to deallocate an environment frame allocated to a rule after successfully falling off the end of the rule. Now, the situation is not quite so straightforward as later failure may force reconsidering a choice from a computation state in the middle of a rule whose environment has long been deallocated. This case is illustrated by the following example program:

```
a :- bX0
cX00
bX0
:- eX00
c
00
eX0
```

```

:- f X00
eX0
:- gX00
f
00
g
00

```

Executing ‘?-a0’ allocates an environment for a, then calls b. Next, an environment for b is allocated, and e is called. This creates a choice point on the OR-stack, and an environment for e is pushed onto the AND-stack. At this point the two stacks look thus:¹⁷

EEnvironment fore

BChoicepoint fore

The following call to f succeeds binding X to . The environment for e is deallocated, then the environment for b is also deallocated. This leads to stacks looking thus:

EEnvironment for a

BChoicepoint fore

Next, the continuation follows up with execution of a’s body, calling c, which immediately hits failure. The choice point indicated by B shows an alternative clause for e, but at this point b’s environment has been lost. Indeed, in a more involved example where c proceeded deeper before failing, the old stack space for b’s environment would have been overwritten by further calls in c’s body

Therefore, to avoid this kind of misfortune, a setup must be found to prevent unrecoverable deallocation of environment frames whose creation chronologically preceded that of any existing choice point. The idea is that every choice point must “protect” from deallocation all environment frames already existing before its creation. Now, since a stack reflects chronological order, it makes sense to use the same stack for both environments and choice points. A choice point now caps all older environments. In effect, as long as it is active, it forces allocation of further environments on top of it, preventing the older environments’ stack space to be overwritten even though they may explicitly be deallocated. This allows their safe resurrection if needed by coming back to an alternative from this choice point. Moreover, this “protection” lasts just as long as it is needed since as soon as the choice point disappears, all explicitly deallocated environments can be safely overwritten.

Hence, there is no need to distinguish between the AND-stack from the OR-stack, calling the single one the stack. Choice point frames are stored in the stack along with environments, and thus B’s value is an address in the stack.

¹⁷ In these diagrams, the stacks grow downwards; i.e., the stack top is the lower part.

Going back to our example above, the snapshot of the single stack at the same first instant looks thus:

Environment for a
Environment for b
BChoicepoint for e
EEnvironment for e

and at the same second instant as before, the stack is such that having pushed on it the choice point for e protects b's deallocated environment (which may still be needed by future alternatives given by e's choice point), looking thus:

EEnvironment for a
Deallocatedenvironment for b
BChoicepoint for e

Now, the computation can safely recover the state from the choice point for e indicated by B, in which the saved environment to restore is the one current at the time of this choice point's creation—i.e., that (still existing) of b. Having no more alternative for e after the second one, this choice point is discarded upon backtracking, (safely) ending the protection. Execution of the last alternative for e proceeds with a stack looking thus:

B
Environment for a
Environment for b
EEnvironment for e

7.4.2 What's in a choice point

When a chosen clause is attempted among those of a definition, it will create side effects on the stack and the heap by binding variables residing there. These effects must be undone when reconsidering the choice. A record must be kept of those variables which need to be reset to ‘unbound’ upon backtracking. Hence, we provide, along with the heap, the stack, the code area, and the PDL, a new (and last!) data area called the trail (TRAIL). This trail is organized as an array of addresses of those (stack or heap) variables which must be reset to ‘unbound’ upon backtracking. Note that it also works as a stack, and we need a new global register TR always set to contain the top of the trail.

It is important to remark that not all bindings need to be remembered in the trail. Only conditional bindings do. A conditional binding is one affecting a variable existing before creation of the current choice point. To determine this, we will use a new global

register HB set to contain the value of H at the time of the latest choice point.¹⁸ Hence only bindings of heap (resp., stack) variables whose addresses are less than HB (resp., B) need be recorded in the trail. We shall write `traila` when that this operation is performed on store address a. As mentioned before, it is done as part of the bind operation.

Let us now think about what constitutes a computation state to be saved in a choice point frame. Upon backtracking, the following information is needed:

The argument registers A₁, ..., An, where n is the arity of the procedure offering alternative choices of definitions. This is clearly needed as the argument registers, loaded by put instructions with the values of arguments necessary for goal being attempted, are overwritten by executing the chosen clause.

The current environment (value of register E), to recover as a protected environment as explained above.

The continuation pointer (value of register CP), as the current choice will overwrite it.

The latest choice point (value of register B), where to backtrack in case all alternatives offered by the current choice point fail. This acts as the link connecting choice points as a list. It is reinstated as the value of the B register upon discarding the choice point.

The next clause, to try in this definition in case the currently chosen one fails. This slot is updated at each backtracking to this choice point if more alternatives exist.

The current trail pointer (value of register TR), which is needed as the boundary where to unwind the trail upon backtracking. If computation comes back to this choice point, this will be the address in the trail down to which all variables that must be reset have been recorded.

The current top of heap (value of register H), which is needed to recover (garbage) heap space of all the structures and variables constructed during the failed attempt which will have resulted in coming back to this choice point.

In summary, a choice point frame is allocated on the stack looking thus:¹⁹

¹⁸ Strictly speaking, register HB can in fact be dispensed with since, as we see next, its value is that of H which will have been saved in the latest choice point frame.

¹⁹ In [War83], David Warren does not include the arity in a choice point, as we do here. He sets up things slightly differently so that this number can always be quickly computed. He can do this by making register B (and the pointers linking the choice point list) reference a choice point frame at its end, rather than its start as is the case for environment frames. In other words, register B contains the stack address immediately following the latest choice point frame, whereas register E contains the address of the first slot in the environment. Thus, the arity of the latest choice point predicate is always given by n B STACK[B]. For didactic reasons, we chose to handle E and B identically, judging that saving one stack slot is not really worth the entailed complication of the code implementing the instructions.

```

B n number
of arguments
B
A
argument
register
.

.

.

B n An argument
register n
B n
CE continuation
environment
B n CP continuation
pointer
B n B previous
choice point
B n BP next
clause
B n TR trail
pointer
B n H heap
pointer

```

Note in passing that M's explicit definition for allocate N must be altered in order to work forM. This is because the top of stack is now computed differently depending on whether an environment or choice point is the latest frame on the stack. Namely, in M:

```

allocate N if E B
then newE E STACK[E ]
else newE B STACK[B]
STACK[newE] E
STACK[newE
] CP
STACK[newE ] N
E newE
P P instruction sizeP

```

To work with the foregoing choice point format, three new I instructions are added to those already in I. They are to deal with the choice point manipulation needed for multiple clause definitions. As expected, these instructions correspond, respectively, to

(1) a first, (2) an intermediate (but non ultimate), and (3) a last, clause of a definition. They are:

1. try me else L
2. retry me else L
3. trust me

where L is an instruction label (i.e., an address in the code area). They have for effect, respectively:

1. to allocate a new choice point frame on the stack setting its next clause field to L and the other fields according to the current context, and set B to point to it;
2. having backtracked to the current choice point (indicated by the current value of the B register), to reset all the necessary information from it and update its next clause field to L; and,
3. having backtracked to the current choice point, to reset all the necessary information from it, then discard it by resetting B to its predecessor (the value of the link slot).

With this setup, backtracking is effectively handled quite easily. All instructions in which failure may occur (i.e., some unification instructions and all procedure calls) must ultimately test whether failure has indeed occurred. If such is the case, they must then set the instruction counter accordingly. That is, they perform the following operation:

$\text{backtrack } P \text{STACK}[B] \text{STACK}[B]$

as opposed to having P be unconditionally set to follow its normal (successful) course. Naturally, if no more choice point exists on the stack, this is a terminal failure and execution aborts. All the appropriate alterations of instructions regarding this precaution are given in Appendix B.

The three choice point instructions are defined explicitly in Figures 4.1, 4.2, and 4.3, respectively. In the definition of try me else L, we use a global variable num of args giving the arity of the current procedure. This variable is set by call that we must accordingly modify for M from its M form as follows.²⁰

$\text{call } pn \text{ CPP } \text{instructionsize } P$

$\text{numofargs } n$

Ppn

As we just explained, we omit treating the case of failure (and therefore of backtracking) where pn is not defined in this explicit definition of call pn. Its obvious complete form is, as those of all instructions of the full WAM, given in Appendix B.

²⁰ As for num of args, it is legitimate to ask why this is not a global register like E, P, etc., in the design. In fact, the exact manner in which the number of arguments is retrieved at choice point creation time is not at all explained in [War83, War88]. Moreover, upon private inquiry, David H. D. Warren could not remember whether that was an incidental omission. So we chose to introduce this global variable as opposed to a register as no such explicit register was specified for the original WAM.

Finally, the definitions of retry me else L and trust me, use an ancillary operation, unwind trail, to reset all variables since the last choice point to an unbound state. Its explicit definition can be found in Appendix B.

In conclusion, there are three patterns of code translations for a procedure definition in L, depending on whether it has one, two, or more than two clauses. The code generated in the first case is identical to what is generated for an L program on M. In the second case, the pattern for a procedure pn is:

```
pn : try me else L  
      code for first clause  
      L : trust me  
      code for second clause
```

```
pn : try me else L  
code for first clause  
L : retry me else L  
code for second clause .  
. .  
. Lk  
: retry me else Lk  
code for penultimate clause  
Lk : trust me  
code for last clause
```

where each clause is translated as it would be as a single L clause for M. For example, M code for the definition:

```
pX  
a  
pb  
X  
pX  
Y :- pX  
a  
pb  
Y
```

is given in Figure 4.4.

Figure 4.1: M choice point instruction try me else

Figure 4.2: M choice point instruction retry me else

Figure 4.3: M choice point instruction trust mentioned

Figure 4.4: M code for a multiple clause definition

Exercise 4.1 Trace the execution of L query `?-pc d` with code in Figure 4.4, giving all the successive states of the stack, the heap, and the trail.

Exercise 4.2 It is possible to maintain separate AND-stack and OR-stack. Discuss the alterations that would be needed to the foregoing setup to do so, ensuring a correct management of environments and choice points.

7.5 Optimizing the Design

Now that the reader is hopefully convinced that the design we have reached forms an adequate target language and architecture for compiling pure Prolog, we can begin transforming it in order to recover Warren's machine as an ultimate design. Therefore, since all optimizations considered here are part of the definitive design, we shall now refer to the abstract machine gradually being elaborated as the WAM. In the process, we shall abide by a few principles of design pervasively motivating all the conception features of the WAM. We will repeatedly invoke these principles in design decisions as we progress toward the full WAM engine, as more evidence justifying them accrues.

WAM PRINCIPLE 1 Heap space is to be used as sparingly as possible, as terms built on the heap turn out to be relatively persistent.

WAM PRINCIPLE 2 Registers must be allocated in such a way as to avoid unnecessary data movement, and minimize code size as well.

WAM PRINCIPLE 3 Particular situations that occur very often, even though correctly handled by general-case instructions, are to be accommodated by special ones if space and/or time may be saved thanks to their specificity.

In the light of WAM Principles 1, 2, and 3, we may now improve on M.

Figure 5.1: Better heap representation for term pZ hZW f W

```
h
REF
REF
f
REF
p
REF
STR
STR
```

7.5.1 Heap representation

As many readers of [AK90] did, this reader may have wondered about the necessity of the extra level of indirection systematically introduced in the heap by an STR cell for

each functor symbol. In particular, Fernando Pereira [Per90] suggested that instead of that shown in Figure 2.1 on Page 11, a more economical heap representation for pZ $hZW f W$ ought to be that of Figure 5.1, where reference to the term from elsewhere must be from a store (or register) cell of the form h STR

i. In other words, there is actually no need to allot a systematic STR cell before each functor cell.

As it turns out, only one tiny modification of one instruction is needed in order to accommodate this more compact representation. Namely, the put structure instruction is simplified to:

```
put structure f n
Xi  HEAP[H]  f n
Xi  h STR
H i
H H
```

Clearly, this is not only in complete congruence with WAM Principle 1, but it also eliminates unnecessary levels of indirection and hence speeds up dereferencing.

The main reason for our not having used this better heap representation in Section 2.1 was essentially didactic, wishing to avoid having to mention references from outside the heap (e.g., from registers) before due time. In addition, we did not bother bringing up this optimization in [AK90] as we are doing here, as we had not realized that so little was in fact needed to incorporate it.²¹

7.5.2 Constants, lists, and anonymous variables

To be fully consistent with the complete WAM unification instruction set and in accordance with WAM Principle 3, we introduce special instructions for the specific handling of -ary structures (i.e., constants), lists, and variables which appear only once within a scope—so-called anonymous variables. These enhancements will also be in the spirit of WAM Principles 1 and 2 as savings in heap space, code size, and data movement will ensue.

Constants and lists are, of course, well handled by the structure oriented get, put, and unify instructions. However, work and space are wasted in the process, that need

²¹ After dire reflection seeded by discussions with Fernando Pereira, we eventually realized that this optimization was indeed cheap—a fact that had escaped our attention. We are grateful to him for pointing this out. However, he himself warns [Per90]:

“Now, this representation (which, I believe, is the one used by Quintus, SICStus Prolog, etc.) has indeed some disadvantages:

1. If there aren’t enough tags to distinguish functor cells from the other cells, garbage collection becomes trickier, because a pointed-to value does not in general identify its own type (only the pointer does).

2. If you want to use [the Huet-Fages] circular term unification algorithm, redirecting pointers becomes messy, for the [same] reason... In fact, what [the term representation in Section 2.1 is] doing is enforcing a convention that makes every functor application tagged as such by the appearance of a STR cell just before the functor word.”

not really be. Consider the case of constants as, for instance, the code in Figure 2.10, on Page 24. There, the sequence of instructions:

unifyvariableX

getstructurea, X7

simply binds a register and proceeds to check the presence of, or build, the constant a on the heap. Clearly, one register can be saved and data movement optimized with one specialized instruction: unify constant a. The same situation in a query would simplify a sequence:

putstructurec

Xi

into one specialized instruction set constant c. Similarly, put and get instructions can thus be specialized from those of structures to deal specifically with constants. Thus, we define a new sort of data cells tagged CON, indicating that the cell's datum is a constant. For example, a heap representation starting at address

for the structure f b ga could be:

g
CON a

f
CON b
STR
E

Exercise 5.1 Could the following (smaller) heap representation starting at address be an alternative for the structure f b ga? Why?

f
CON b
g
CON a
He

Heap space for constants can also be saved when loading a register with, or binding a variable to, a constant. Rather than systematically occupying a heap cell to reference, a constant can be simply assigned as a literal value. The following instructions are thus added to I:

1. put constant c Xi
2. get constant c Xi
3. set constant c
4. unify constant c

and are explicitly defined in Figure 5.2.

Figure 5.2: Specialized instructions for constants

Figure 5.3: Specialized instructions for lists Programming with linear lists being so privileged in Prolog, it makes sense to tailor the design for this specific structure. In particular, non-empty list functors need not be represented explicitly on the heap. Thus again, we define a fourth sort for heap cells tagged LIS, indicating that the cell's datum is the heap address of the first element of a list pair. Clearly, to respect the subterm contiguity convention, the second of the pair is always at the address following that of the first. The following instructions (defined explicitly in Figure 5.3) are thus added to I:

1. put list Xi
2. get list Xi

For example, the code generated for query `?-pZ ZW f W`, using Prolog's notation for lists, is shown in Figure 5.4 and that for fact `pf X Y f a Y`, in Figure 5.5. Note the hidden presence of the atom as list terminator. Of c

Of course, having introduced specially tagged data cells for constants and nonempty lists will require adapting accordingly the general-purpose unification algorithm given in Figure 2.7. The reader will find the complete algorithm in appendix Section B.2, on Page 117.

Exercise 5.2 In [War83], Warren also uses special instructions `put nil Xi`, `get nil Xi`, and to handle the list terminator constant `.`. Define the effect of these instructions, and give explicit pseudo-code implementing them. Discuss their worth being provided as opposed to using `put constant Xi`, `get constant Xi`, `set constant ,` and `unify constant .`

Figure 5.4: Specialized code for query `?-pZ ZW f W p`

Figure 5.5: Specialized code for fact `pf X Y f a Y` Last in the rubric of specialized instructions is the case of single-occurrence variables in non-argument positions (e.g., `X` in Figure 2.4 on Page 16, Figure 2.10 on Page 24, and Figure 5.5 on Page 51). This is worth giving specialized treatment insofar as no register need be allocated for these. In addition, if many occur in a row as in `f`, say, they can be all be processed in one swoop, saving in generated code size and time. We introduce two new instructions:

1. set void n
2. unify void n

whose effect is, respectively:

1. to push n new unbound REF cells on the heap;
2. in write mode, to behave as `set void n` and, in read mode, to skip the next n heap cells starting at location S.

These are given explicitly in Figure 5.6.

Note finally, that an anonymous variable occurring as an argument of the head of a clause can be simply ignored. Then indeed, the corresponding instruction:

getvariableXi

is clearly vacuous. Thus, such instructions are simply eliminated. The code for fact p
gX f
Y , for example, shown in Figure 5.7, illustrates this point.

Exercise 5.3 What is the machine code generated for the fact p ? What about the query ?-p ?

7.5.3 A note on set instructions

Defining the simplistic language L has allowed us to introduce, independently of other Prolog considerations, all WAM instructions dealing with unification. Strictly speaking, the set instructions we have defined are not part of the WAM as described in [War83] or in [War88]. There, one will find that the corresponding unify instructions are systematically used where we use set instructions. The reason is, as the reader may have noticed, that indeed this is possible provided that the put structure and put list instructions set mode to write. Then, clearly, all set instructions are equivalent to unify instructions in write mode. We chose to keep these separate as using set instructions after put instructions is more efficient (it saves mode setting and testing) and makes the code more perspicuous. Moreover, these instructions are more natural, easier to explain and motivate as the data building phase of unification before matching work comes into play.

Figure 5.6: Anonymous variable instructions

Figure 5.7: Instructions for fact pgXY Incidentally, these instructions together with their unify homologues, make “onthe-fly” copying part of unification, resulting in improved space and time consumption, as opposed to the more naive systematic copying of rules before using them.

7.5.4	5.4 Register allocation	54
7.5.5	5.5 Last call optimization	56
7.5.6	5.6 Chain rules	57
7.5.7	5.7 Environment trimming	58
7.5.8	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
7.5.9	5.9 Variable classification revisited	69
7.5.10	5.10 Indexing	75
7.5.11	5.11 Cut	83
7.6	6 Conclusion	89
7.7	A Prolog in a Nutshell	91
7.8	B The WAM at a glance	97
7.8.1	B.1 WAM instructions	97
7.8.2	B.2 WAM ancillary operations	112
7.8.3	B.3 WAM memory layout and registers	117

Глава 8

An Efficient Unification Martelli/Montanary Algorithm

¹

© ALBERTO MARTELLI Consiglio Nazionale delle Ricerche
and
UGO MONTANARI Universita di Pisa ²

Abstract

The unification problem in first-order predicate calculus is described in general terms as the solution of a system of equations, and a nondeterministic algorithm is given. A new unification algorithm, characterized by having the acyclicity test efficiently embedded into it, is derived from the nondeterministic one, and a PASCAL implementation is given. A comparison with other well-known unification algorithms shows that the algorithm described here performs well in all cases.

Categories and Subject Descriptors: F.2.2 [Analysis of Algorithms and Problem Complexity]: Nonnumerical Algorithms and Problems—complexity of proof procedures; F.4.1 [Mathematical Logic and Formal Languages]: Mathematical Logic—mechanical theorem proving; I.2.3 [Artificial Intelligence]: Deduction and Theorem Proving—resolution

¹ © <http://www.nsl.com/misc/papers/martelli-montanari.pdf>

² Authors' present addresses: A. Martelli, Istituto di Scienze della Informazione, Università di Torino, Corso M. d'Aeglio 42, 1-10125 Torino, Italy; U. Montanari, Istituto di Scienze della Informazione, Università di Pisa, Corso Italia 40, 1-56100 Pisa, Italy.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1982 ACM 0164-0925/82/0400-0258 \$00.75

ACM Transactions on Programming Languages and Systems, Vol. 4, No. 2, April 1982, Pages 258-282.

8.1 INTRODUCTION

In its essence, the unification problem in first-order logic can be expressed as follows: Given two terms containing some variables, find, if it exists, the simplest substitution (i.e., an assignment of some term to every variable) which makes the two terms equal. The resulting substitution is called the *most general unifier* and is unique up to variable renaming.

Unification was first introduced by Robinson [17, 18] as the central step of the inference rule called resolution. This single, powerful rule can replace all the axioms and inference rules of the first-order predicate calculus and thus was immediately recognized as especially suited to mechanical theorem provers. In fact, a number of systems based on resolution were built and tried on a variety of different applications [5]. Even though further research made it apparent that resolution systems are difficult to direct during proof search and thus are often prone to combinatorial explosion [6], new impetus to the research in this area was given by Kowalski's idea of interpreting predicate logic as a programming language [10]. Here predicate logic clauses are seen as procedure declarations, and procedure invocation represents a resolution step. From this viewpoint, theorem provers can be regarded as interpreters for programs written in predicate logic, and this analogy suggests efficient implementations [3, 25].

Resolution, however, is not the only application of the unification algorithm. In fact, its pattern matching nature can be exploited in many cases where symbolic expressions are dealt with, such as, for instance, in interpreters for equation languages [4, 11], in systems using a database organized in terms of productions [19], in type checkers for programming languages with a complex type structure [14], and in the computation of critical pairs for term rewriting systems [9].

The unification algorithm constitutes the heart of all the applications listed above, and thus its performance affects in a crucial way the global efficiency of each. The unification algorithm as originally proposed can be extremely inefficient; therefore, many attempts have been made to find more efficient algorithms [2, 7, 13, 15, 16, 22]. Unification algorithms have also been extended to the case of higher order logic [8] and to deal directly with associativity and commutativity [20]. The problem was also tackled from a computational complexity point of view, and linear algorithms were proposed independently by Martelli and Montanari [13] and Paterson and Wegman [15].

In the next section we give some basic definitions by representing the unification problem as the solution of a system of equations. A nondeterministic algorithm, which comprehends as special cases most known algorithms, is then defined and proved correct. In Section 3 we present a new version of this algorithm obtained by grouping together all equations with some member in common, and we derive from it a first version of our unification algorithm.

In Sections 4 and 5 we present the main ideas which make the algorithm efficient,

and the last details are described in Section 6 by means of a PASCAL implementation.

Finally, in Section 7, the performance of this algorithm is compared with that of two well-known algorithms, Huet's [7] and Paterson and Wegman's [15]. This analysis shows that our algorithm has uniformly good performance for all classes of data considered.

8.2 UNIFICATION AS THE SOLUTION OF A SET OF EQUATIONS: A NONDETERMINISTIC ALGORITHM

In this section we introduce the basic definitions and give a few theorems which are useful in proving the correctness of the algorithms. Our way of stating the unification problem is slightly more general than the classical one due to Robinson [18] and directly suggests a number of possible solution methods.

Let

$$A = \bigcup_{i=0,1,\dots} A_i \quad (A_i \cap A_j = \emptyset, i \neq j)$$

be a ranked alphabet, where A_i contains the i -adic function symbols (the elements of A_0 are constant symbols). Furthermore, let V be the alphabet of the variables. The *terms* are defined recursively as follows:

- (1) constant symbols and variables are terms;
- (2) if t_1, \dots, t_n ($n \geq 1$) are terms and $f \in A_n$, then $f(t_1, \dots, t_n)$ is a term.

A *substitution* ϑ is a mapping from variables to terms, with $\vartheta(x) = x$ almost everywhere. A substitution can be represented by a finite set of ordered pairs $\vartheta = (t_1, x_1), (t_2, x_2), \dots, (t_m, x_m)$ where t_i are terms and x_i are distinct variables, $i = 1, \dots, m$. To apply a substitution ϑ to a term t , we simultaneously substitute all occurrences in t of every variable x_i in a pair (t_i, x_i) of ϑ with the corresponding term t_i . We call the resulting term t_ϑ .

For instance, given a term $t = f(x_1, g(x_2, a))$ and a substitution $\vartheta = (h(x_2), x_1), (b, x_2)$ we have $t_\vartheta = f(f(x_2), g(b), a)$ and $t_{\vartheta\vartheta} = f(h(b), g(b), a)$.

The standard unification problem can be written as an equation

$$t' = t''$$

A solution of the equation, called a *unifier*, is any substitution ϑ , if it exists, which makes the two terms identical. For instance, two unifiers of the equation $f(x_1, h(x_1), x_2, f(g(x_3), x_4, x_3))$ are $\vartheta_1 = (g(x_3), x_1), (x_3, x_2), (h(g(x_3)), x_4)$ and $\vartheta_2 = (g(a), x_1), (a, x_2)$.

In what follows it is convenient also to consider sets of equations

$$t'_j = t''_j, \quad j = 1, \dots, k$$

Again, a *unifier* is any substitution which makes all pairs of terms t'_j, t''_j identical simultaneously.

Now we are interested in finding transformations which produce **equivalent** sets of equations, namely, transformations which preserve the sets of all unifiers. Let us introduce the following two transformations:

(1) Term Reduction. Let

$$f(t'_1, t'_2, \dots, t'_n) = f(t''_1, t''_2, \dots, t''_n), \quad f \in A_n \quad (8.1)$$

be an equation where both terms are not variables and where the two root function symbols are equal. The new set of equations is obtained by replacing this equation with the following ones:

$$t'_1 = t''_1 \quad (8.2)$$

$$t'_2 = t''_2 \quad (8.3)$$

$$\dots \quad (8.4)$$

$$\dots \quad (8.5)$$

$$\dots \quad (8.6)$$

$$t'_n = t''_n \quad (8.7)$$

If $n = 0$, then f is a constant symbol, and the equation is simply erased.

(2) Variable Elimination. Let $x = t$ be an equation where x is a variable and t is any term (variable or not). The new set of equations is obtained by applying the substitution $\vartheta = (t, x)$ to both terms of all other equations in the set (without erasing $x = t$).

We can prove the following theorems:

THEOREM 2.1. *Let S be a set of equations and let $f'(t'_1, \dots, t'_n) = f''(t''_1, \dots, t''_n)$ be an equation of S . If $f' \neq f''$, then S has no unifier. Otherwise, the new set of equations S' , obtained by applying term reduction to the given equation, is equivalent to S .*

PROOF. If $f' \neq f''$, then no substitution can make the two terms identical. If $f' = f''$, any substitution which satisfies 8.2 also satisfies 8.1, and conversely for the recursive definition of term. \square

THEOREM 2.2. *Let S be a set of equations, and let us apply variable elimination to some equation $x = t$, getting a new set of equations S' . If variable x occurs in t (but t is not x), then S has no unifier; otherwise, S and S' are equivalent.*

PROOF. Equation $x = t$ belongs both to S and to S' , and thus any unifier ϑ (if it exists) of S or of S' must unify x and t ; that is, x_ϑ and t_ϑ are identical. Now let $t_1 = t_2$ be any other equation of S , and let $t'_1 = t'_2$ be the corresponding equation in S' . Since t'_1 and t'_2 have been obtained by substituting t for every occurrence of x in t_1 and t_2 , respectively, we have $t_{1\vartheta} = t'_{1\vartheta}$ and $t_{2\vartheta} = t'_{2\vartheta}$. Thus, any unifier of S is also a unifier of S' and vice versa. Furthermore, if variable x occurs in t (but t is not x), then no substitution ϑ can make x and t identical, since x_ϑ becomes a subterm of t_ϑ , and thus S has no unifier. \square

A set of equations is said to be *in solved form* iff it satisfies the following conditions:

- (1) the equations are $x_j = t_j, j = 1, \dots, k$;
- (2) every variable which is the left member of some equation occurs only there.

A set of equations in solved form has the obvious unifier

$$\vartheta = (t_1, x_1), (t_2, x_2), \dots, (t_k, x_k)$$

If there is any other unifier, it can be obtained as

$$0 = (t_1, x_1), (t_2, x_2), \dots, (t_k, x_k) U a$$

where a is any substitution which does not rewrite variables x_1, \dots, x_k . Thus t is called a most general unifier (mgu).

The following nondeterministic algorithm shows how a set of equations can be transformed into an equivalent set of equations in solved form.

Algorithm 1

Given a set of equations, repeatedly perform any of the following transformations. If no transformation applies, stop with success.

- (a) Select any equation of the form

$$t = x$$

where t is not a variable and x is a variable, and rewrite it as

$$x = t.$$

- (b) Select any equation of the form

$$X = X$$

where X is variable, and erase it.

- (c) Select any equation of the form

$$t' = t''$$

where t' and t'' are not variables. If the two root function symbols are different, stop with failure; otherwise, apply term reduction.

- (d) Select any equation of the form

$$x = t$$

where x is a variable which occurs somewhere else in the set of equations and where $t \neq x$. If x occurs in t , then stop with failure; otherwise, apply variable elimination.

As an example, let us consider the following set of equations:

$$g(x_2) = xl;$$

$$f(xl, h(xl), x2) = f(g(x3), x4, x3).$$

By applying transformation (c) of Algorithm 1 to the second equation we get

$$g(x2) = xl;$$

$$xl = g(x3);$$

$$h(x) = x4;$$

$$X2 = X3.$$

By applying transformation (d) to the second equation we get

$$g(x2) = g(xs);$$

$$xl = g(x3);$$

$$h(g(x3)) = x4;$$

$$X2 - X3.$$

We now apply transformation (c) to the first equation and transformation (a) to the third equation:

$$X2 \quad X3$$

$$xl = g(x3);$$

$$Xa = h(g(x3));$$

$$X2 - - - X3.$$

Finally, by applying transformation (d) to the first equation and transformation (b) to the last equation, we get the set of equations in solved form:

$$X2 - X3;$$

$$xl = g(x3);$$

$$x4 = h(g(x3)).$$

Therefore, an mgu of the given system is

$$= (g(x), x), (x3, x2), (h(g(x3)), x4).$$

The following theorem proves the correctness of Algorithm 1.

THEOREM 2.3 . Given a set of equations S,

(i) Algorithm 1 always terminates, no matter which choices are made.

(ii) If Algorithm 1 terminates with failure, S has no unifier. If Algorithm 1 terminates with success, the set S has been transformed into an equivalent set in solved form.

PROOF

(i) Let us define a function F mapping any set of equations S into a triple of natural numbers (n_1, n_2, n_3) . The first number, n_1 , is the number of variables in S which do not occur only once as the left-hand side of some equation. The second number, n_2 , is the total number of occurrences of function symbols in S . The third number, n_3 , is the sum of the numbers of equations in S of type $x = x$ and of type $t = x$, where x is a variable and t is not. Let us define a total ordering on such triples as follows:

$$(n_1, n_2, n_3) > (n'_1, n'_2, n'_3) \text{ if } n_1 > n'_1 \\ \text{or } n_1 = n'_1 \text{ and } n_2 > n'_2 \\ \text{or } n_1 = n'_1 \text{ and } n_2 = n'_2 \text{ and } n_3 > n'_3.$$

With the above ordering, N_3 becomes a well-founded set, that is, a set where no infinite decreasing sequence exists. Thus, if we prove that any transformation of Algorithm 1 transforms a set S in a set S' such that $F(S') < F(S)$, we have proved the termination. In fact, transformations (a) and (b) always decrease n_3 and, possibly, n_1 . Transformation (c) can possibly increase n_3 and decrease n_1 , but it surely decreases n_2 (by two). Transformation (d) can possibly change n_3 and increase n_2 , but it surely decreases n_1 .

(ii) If Algorithm 1 terminates with failure, the thesis immediately follows from Theorems 2.1 and 2.2. If Algorithm 1 terminates with success, the resulting set of equations S' is equivalent to the given set S . In fact, transformations (a) and (b) clearly do not change the set of unifiers, while for transformations (c) and (d) this fact is stated in Theorems 2.1 and 2.2. Finally, S' is in solved form. In fact, if (a), (b), and (c) cannot be applied, it means that the equations are all in the form $x = t$, with $t \neq x$. If (d) cannot be applied, that means that every $v.\text{arialSle}$ which is the left-hand side of some equation occurs only there. \square

The above nondeterministic algorithm provides a widely general version from which most unification algorithms [2, 3, 7, 13, 15, 16, 18, 22-24] can be derived by specifying the order in which the equations are selected and by defining suitable concrete data structures. For instance, Robinson's algorithm [18] might be obtained by considering the set of equations as a stack.

8.3 AN ALGORITHM WHICH EXPLOITS A PARTIAL ORDERING AMONG SETS OF VARIABLES

8.3.1 Basic Definitions

In this section we present an extension of the previous formalism to model our algorithm more closely. We first introduce the concept of multiequation. A multiequation is the

generalization of an equation, and it allows us to group together many terms which should be unified. To represent multiequations we use the notation $S - M$ where the left-hand side S is a nonempty set of variables and the right-hand side M is a multiset¹ of nonvariable terms. An example is

$$xl, x2, x3 = (tl, t2).$$

3

The solution (unifier) of a multiequation is any substitution which makes all terms in the left- and right-hand sides identical.

A multiequation can be seen as a way of grouping many equations together. For instance, the set of equations

$$Xl --- X2;$$

$$X3 = Xl;$$

$$tl = Xl;$$

$$X2 --- t2;$$

$$tl = t2$$

can be transformed into the above multiequation, since every unifier of this set of equations makes the terms of all equations identical. To be more precise, given a set of equations SE , let us define a relation RSE between pairs of terms as follows: $tl RSE t2$ iff the equation $tl = t2$ belongs to SE . Let $/tSE$ be the reflexive, symmetric, and transitive closure of RSE .

Now we can say that a set of equations SE corresponds to a multiequation $S = M$ iff all terms of SE belong to $S \cup M$ and for every tr and ts $E S \cup M$ we have $tr RSE ts$.

It is easy to see that many different sets of equations may correspond to a given multiequation and that all these sets are equivalent. Thus the set of solutions (unifiers) of a multiequation coincides with the set of solutions of any corresponding set of equations.

Similar definitions can be given for a set of multiequations Z by introducing a relation Rz between pairs of terms which belong to the same multiequation. A set of equations SE corresponds to a set of multiequations Z iff

$$ti / SE \wedge j * ti Rz t j$$

for all terms $t, t j$ of SE or Z .

³ A multiset is a family of elements in which no ordering exists but in which many identical elements may occur.

8.3.2 Transformations of Sets of Multiequations

We now introduce a few transformations of sets of multiequations, which are generalizations of the transformations presented in Section 2.

We first define the common part and the frontier of a multiset of terms (variables or not). The common part of a multiset of terms M is a term which, intuitively, is obtained by superimposing all terms of M and by taking the part which is common to all of them starting from the root. For instance, given the multiset of terms

$$(f(xl, g(a, f(xs, b))), f(h(c), g(x2, f(b, xs))), f(h(x4), g(x6, x3))),$$

the common part is

$$f(xl, g(x2, x3)).$$

The frontier is a set of multiequations, where every multiequation is associated with a leaf of the common part and consists of all subterms (one for each term of M) corresponding to that leaf. The frontier of the above multiset of terms is

$$\begin{aligned}\{\{x\}\} &= (h(c), h(x4)), \\ \{x2, x6\} &= (a), \\ \{x3\} &= (f(xs, b), f(b, xD)).\end{aligned}$$

Note that if there is a clash of function symbols among some terms of a multiset of terms M, then M has no common part and frontier. In this case the terms of M are not unifiable.

The common part and the frontier can be defined more precisely by means of a function DEC which takes a multiset of terms M as argument and returns either "failure," in which case M has neither common part nor frontier, or a pair (C(M), F(M)) where C(M) is the common part of M and F(M) is the frontier of M.

In the definition of DEC we use the following notation:

head(t) is the root function symbol of term t, for $t \in V$.

P_i is the i th projection, defined by

$$\lambda [P_i(f(t_1, \dots, t_n)) = t_i \text{ for } f \in A \text{ and } 1 \leq i \leq n]$$

make is a function which transforms a multiset of terms M into a multiequation whose left-hand side is the set of all variables in M and whose right-hand side is the multiset of all terms in M which are not variables; and

U is the union for multisets.

```
DEC(M) = ff 3t ~ M, t E V
        then (t, {makemulteq(M)} )
        else if 3n, 3 f E A, , Yt E M, head(t) = f
              then if n ffi 0
                  then ( f, O)
                  else if Vi (1 -- i -- n), DEC(Mi) ~ failure
                        where Mi --- OteM Pi(t)
                  then (f(C(M1) ..... C(M)), UTffil F(Mi))
                  else failure
                  else failure .
```

We can now define the following transformation:

Multiequation Reduction. Let Z be a set of multiequations containing a multiequation $S - M$ such that M is nonempty and has a common part C and a frontier F . The new set Z' of multiequations is obtained by replacing $S = M$ with the union of the multiequation $S = (C)$ and of all the multiequations of F :

$$Z' \text{ ffi} (Z - S \text{ ffi} M) US = (C) UF.$$

THEOREM 3.1. Let $S = M$ (M nonempty) be a multiequation of a set Z of multiequations. If M has no common part, or if some variable in S belongs to the left-hand side of some multiequation in the frontier F of M , then Z has no unifier. Otherwise, by applying multiequation reduction to the multiequation $S = M$ we get an equivalent set Z' of multiequations.

PROOF. If the common part of M does not exist, then the multiequation $S - M$ has no unifier, since two terms should be made equal having a different function symbol in the corresponding subterms. Moreover, if some variable x of S occurs in some left-hand side of the frontier, then it also occurs in some term t of M , and thus the equation $x = t$, with x occurring in t , belongs to a set of equations equivalent to Z . But, according to Theorem 2.2, this set has no unifier.

To prove that Z and Z' are equivalent, we show first that a unifier of Z is also a unifier of Z' . In fact, if a substitution makes all terms of M equal, it also makes equal all the corresponding subterms, in particular, all terms and variables which belong to left- and right-hand sides of the same multiequation in the frontier. The multiequation $S = (C)$ is also satisfied by construction. Conversely, if σ satisfies Z' , then the multiequation $S - M$ is also satisfied. In fact, all terms in S and M are made equal—in their upper part (the common part) due to the multiequation $S = (C)$ and in their lower part (the subterms not included in the common part) due to the set of multiequations F . \square

We say that a set Z of multiequations is compact iff

$$\forall [Y(S = M), (S' = M')] \sim Z: SA S' = \sim . \forall]$$

We can now introduce a second transformation, which derives a compact set of multiequations.

Compactification. Let Z be a noncompact set of multiequations. Let R be a relation between pairs of multiequations of Z such that $iS = M \sim iS' = M'$ iff $S \sim S' \# O$, and let $/t$ be the transitive closure of R . The relation $/t$ partitions the set Z into equivalence classes. To obtain the final compact set Z' , all multiequations belonging to the same equivalence class are merged; that is, they are transformed into single multiequations by taking the union of their left- and right-hand sides.

Clearly, Z and Z' are equivalent, because the relation $/t$ between pairs of terms, defined in Section 3.1, does not change by passing from Z to Z' .

8.3.3 Solving Systems of Multiequations

For convenience, in what follows, we want to give a structure to a set of multiequations. Thus we introduce the concept of system of multiequations. A system R is a pair $(T,$

U), where T is a sequence and U is a set of multiequations (either possibly empty), such that

- (1) the sets of variables which constitute the left-hand sides of all multiequations in both T and U contain all variables and are disjoint;
- (2) the right-hand sides of all multiequations in T consist of no more than one term; and
- (3) all variables belonging to the left-hand side of some multiequation in T can only occur in the right-hand side of any preceding multiequation in T .

We now present an algorithm for solving a given system R of multiequations. When the computation starts, the T part is empty, and every step of the following Algorithm 2 consists of "transferring" a multiequation from the U part, that is, the unsolved part, to the T part, that is, the triangular or solved part of R . When the U part of R is empty, the system is essentially solved. In fact, the solution can be obtained by substituting the variables backward. Notice that, by keeping a solved system in this triangular form, we can hope to find efficient algorithms for unification even when the mgu has a size which is exponential with respect to the size of the initial system. For instance, the mgu of the set of multiequations

```
{ $\{x_1\}$  =  $\sim$ ,  
 $\{x^\sim\}$  =  $\sim$ ,  
 $\{x_3\}$  = 0,  
 $\{x_4\}$  = ( $h(x_3, h(x_2, x_2))$ ,  $h(h(h(x_1, x_1), x_2), x_3))$ }
```

is

```
{( $h(x_1, x_1), x_2$ ), ( $h(h(x_1, x_1), h(x_1, x_1)), x_3$ ),  
( $h(h(h(x_1, x_1), h(x_1, x_1)), h(h(x_1, x_1), h(x_1, x_1)))$ ),  $x_4$ )}.
```

However, we can give an equivalent solved system with empty U part and whose T part is

```
( $\{x\}$  --- ( $h(x_3, x_3)$ ),  
 $\{x_3\}$  = ( $h(x_2, x_2)$ ),  
 $\{x_2\}$  = ( $h(x_1, x_1)$ ),  
 $\{x_1\}$  = 0),
```

from which the mgu can be obtained by substituting backward.

Given a system $R = (T, U)$ with an empty T part, an equivalent system with an empty U part can be computed with the following algorithm.

Algorithm 2

1. (1) repeat
 - (a) (1.1) Select a multiequation $S = M$ of U with $M \# 5$.

- (b) (1.2) Compute the common part C and the frontier F of M. If M has no common part, stop with failure (clash).
 - (c) (1.3) If the left-hand sides of the frontier of M contain some variable of S, stop with failure (cycle).
 - (d) (1.4) Transform U using multiequation reduction on the selected multiequations and compactification.
 - (e) (1.5) Let $S = \{x_1, \dots, x_n\}$. Apply the substitution $x_i = (C, x_1), \dots, (C, x_n)$ to all terms in the right-hand side of the multiequations of U.
 - (f) (1.6) Transfer the multiequation $S = (C)$ from U to the end of T, until the U part of R contains only multiequations, if any, with empty right-hand sides.

2. (2) Transfer all the multiequations of U (all with $M = D$) to the end of T, and stop with success.

Of course, if we want to use this algorithm for unifying two terms t_1 and t_2 , we have to construct an initial system with empty T part and with the following U part:

```
\[\{\{x\} = (t1, t2), \{x1\} = 6, \{x2\} = 0 \dots \{x,\} = 6\}\]]
```

where x_1, x_2, \dots, X_n are all the variables in t_1 and t_2 and x is a new variable which does not occur in t_1 and t_2 . For instance, let $t_1 = f(x_1, g(x_2, x_3), x_2, b)$ and $t_2 = f(g(h(a, x_3), x_2), x_1, h(a, x_4), x_4)$. The initial system is as follows:

J: $\{\{x\} = (f(x_1, g(x_2, x_3), x_2, b), f(g(h(a, x^~), x_2), x_1, h(a, x_4), x_4))$
 $\{x^~\} = 6, \{x_2\} = 6, \{x_3\} = ;D, \{x_4\} = 6, \{xs\} = 6\}; (3)$
T:().

After the first iteration of Algorithm 2 we get

```

U: {x~} = (g(h(a, x~), x2), g(x2, x3)),
{x2} = (h(a, x4)),
(x~) = 0,
(x4) = (b),
(xs) = ~);
T: ( {x} = (f(x1, x1, x2, x4))).
```

We now eliminate variable x_2 , obtaining

```

U: ({X1} = (g(h(a, xs), h(a, x4)), g(h(a, x4), x3)),
{x3} = 6,
(x4) = (b),
{x5} = 0};

T: ( (x) = (f(x1, X1, x2, x4)),
{x2} = (h(a, x4))).
```

By eliminating variable x_1 , we get

```
U: {x3} = (h(a, x4)),  
{x,, xs} = (b));  
T: ( (x) = (f(x1, xi, x2, x4)),  
(x2) = (h(a, x4)),  
(x1) = (g(h(a, x4), x3))).
```

Finally, by eliminating first the set x_4 , xs and then x_3 , we get the solved system

```
U: 0;  
T: ((x) = (f(x~, x1, x2, x4)),  
(x2) = (h(a, x4)),  
{x1) = (g(h(a, x4), xz)),  
(x4, xs) = (b),  
{x3) = (h(a, b))).
```

We can now prove the correctness of Algorithm 2.

THEOREM 3.2. Algorithm 2 always terminates. If it stops with failure, then the given system has no unifier. If it stops with success, the resulting system is equivalent to the given system and has an empty unsolved part.

PROOF. All transformations obtain systems equivalent to the given one. In fact, in step (1.4) multiequation reduction obtains a set of equations which (according to Theorem 3.1) is equivalent, and compactification transforms it again into a system. Step (1.5) applies substitution only to the terms in U , and its feasibility can be proved as in Theorem 2.2. Step (1.6) can be applied since the multiequation $S = (C)$, introduced during multiequation reduction, has not been modified by compactification, due to the condition tested in step (1.3). For the same condition, transferring multiequation $S = (C)$ from U to T still leaves a system. Step (2) is clearly feasible.

If the algorithm stops with failure, then, by Theorem 3.1, the system presently denoted by R (equivalent to the given one) has no solution. Otherwise, the final system clearly has an empty U part. Finally, the algorithm always terminates since at every cycle some variable is eliminated from the U part. \square

It is easy to see that, for a given system, the size of the final system depends heavily on the order of elimination of the multiequations. For instance, given the same system as discussed earlier,

```
U: {{x1)} = ~,  
{x2} = (h(x1, X1)),  
{x3} = (h(x2, x2)),  
{x,) = (h(x3, x3))};  
T:(),
```

and eliminating the variables in the order x_2 , xz , x_4 , Xx , we get the final system

```

U: 0;
T: ({x2} -- (h(xm, Xl)),
{x3} = (h(h(Xl, xl), h(Xl, xl))),
{x4} = (h(h(h(Xl, Xl), h(xl, xl)), h(h(xl, xl), h(Xl, Xl)))),
{x~ } = 0).

```

If instead we eliminate the variables in the order x4, x3, x2, xl, we get

```

U: 0;
T: ({x4} = (h(x3, x3)),
(x3} = (h(x2, x2)),
{x2} -- (h(Xl, Xx)),
{x, } = 0).

```

8.3.4 The Unification Algorithm

Looking at Algorithm 2, it is clear that the main source of complexity is step (1.5), since it may make many copies of large terms. In the following—and this is the heart of our algorithm—we show that, if the system has unifiers, then there always exists a multiequation in U (if not empty) such that by selecting it we do not need step (1.5) of the algorithm, since the variables in its left-hand side do not occur elsewhere in U. We need the following definition.

Given a system R, let us consider the subset Vu of variables obtained by making the union of all left-hand sides Si of the multiequations in the U part of R. Since the sets Si are disjoint, they determine a partition of Vu. We now define a relation on the classes Si of this partition: we say that $S_i < S_j$ iff there exists a variable of S_i occurring in some term of M_j , where M_j is the right-hand side of the multiequation whose left-hand side is S_j . We write $<^*$ for the transitive closure of $<$.

Now we can prove the following theorem and corollary.

THEOREM 3.3. If a system R has a unifier, then the relation $<^*$ is a partial ordering. **PROOF.** If $S_i < S_j$, then, in all unifiers of the system, the term substituted for every variable in S_i must be a strict subterm of the term substituted for every variable in S_j . Thus, if the system has a unifier, the graph of the relation $<$ cannot have cycles. Therefore, its transitive closure must be a partial ordering. \square

COROLLARY. If the system R has a unifier and its U part is nonempty, there exists a multiequation S ffi M such that the variables in S do not occur elsewhere in U.

PROOF. Let S = M be a multiequation such that S is "on top" of the partial ordering $<^*$ (i.e., $\exists S_i, S <^* S_i$). The variables in S occur neither in the other lefthand sides of U (since they are disjoint) nor in any right member M_i of U, since otherwise $S < S_i$. \square

We can now refine the nondeterministic Algorithm 2 giving the general version of our unification algorithm for a system of multiequations $R = (T, U)$.

Algorithm 3: UNIFY, the Unification Algorithm

1. (1) repeat

- (a) (1.1) Select a multiequation $S = M$ of U such that the variables in S do not occur elsewhere in U . If a multiequation with this property does not exist, stop with failure (cycle).
- (b) (1.2) if M is empty then transfer this multiequation from U to the end of T . else begin
 - i. (1.2.1) Compute the common part C and the frontier F of M . If M has no common part, stop with failure (clash).
 - ii. (1.2.2) Transform U using multiequation reduction on the selected multiequation and compactification.
 - iii. (1.2.3) Transfer the multiequation $S = (C)$ from U to the end of T . end until the U part of R

(2) stop with success.

A few comments are needed. Besides step (1.5) of Algorithm 2, we have also erased step (1.3) for the same reason. Furthermore, in Algorithm 2 we were forced to wait to transfer multiequations with empty right-hand sides since substitution in that case would have required a special treatment.

By applying Algorithm UNIFY to the system which was previously solved with Algorithm 2, we see that we must first eliminate variable x_1 , then variable x_2 , then variables x_2 and x_3 together, and, finally, variables x_4 and x_5 together, getting the following final system:

```

U: ~
T: ({x} = (f(x1, x1, x2, x, )), ,
{X1} = (g(x2, x3)), ,
{x2, x3} = (h(a, x4)), ,
{x,, xs} = (b)).
```

Note that the solution obtained using Algorithm UNIFY is more concise than the solution previously obtained using Algorithm 2, for two reasons. First, variables x_2 and x_3 have been recognized as equivalent; second, the right member of x is more factorized. This improvement is not casual but is intrinsic in the ordering behavior of Algorithm UNIFY.

To summarize, Algorithm UNIFY is based mainly on the two ideas of keeping the solution in a factorized form and of selecting at each step a multiequation in such a way that no substitution ever has to be applied. Because of these two facts, the size of the final system cannot be larger than that of the initial one. Furthermore, the operation of selecting a multiequation fails if there are cycles among variables, and thus the so-called occur-check is built into the algorithm, instead of being performed at the last step as in other algorithms [2, 7].

8.4 EFFICIENT MULTIEQUATION SELECTION

In this section we show how to implement efficiently the operation of selecting a multiequation "on top" of the partial ordering in step (1.1) of Algorithm 3.

The idea is to associate with every multiequation a counter which contains the number of other occurrences in U of the variables in its left-hand side. This counter is initialized by scanning the whole U part at the beginning. Of course, a multiequation whose counter is set to zero is on top of the partial ordering.

For instance, let us again consider system (3):

```
U: {[0] {x} = (f(x1,g(x2, x3), x2, b), f(g(h(a, x~), xe), x1,
h(a, x4), x4)),
[2] {x1} = 6,
[3] {x2} = 6,
[1] (x3) = 6,
[2] {x4} = 6,
[1] {xa} -- 6};
T:();
```

Here square brackets enclose the counters associated with each multiequation. Since only the first multiequation has its counter set to zero, it is selected to be transferred. Counters of the other multiequations are easily updated by decrementing them whenever an occurrence of the corresponding variable appears in the left-hand side of a multiequation in the frontier computed in step (1.2.1). When two or more multiequations in U are merged in the compactification phase, the counter associated with the new multiequation is obviously set to a value which is the sum of the contents of the old counters.

The next steps are as follows:

```
U: {[0] (X1} = (g(h(a, x~), x2), g(x2, x3)),
[2] {x2} = (h(a, x4)),
[1] (x~) = o,
[1] {x,} = (b),
[1] {x~} = ~};
T: ( (x} = (f(x1, x~, x2, x4))).
U: {[0] {x2, x3} = (h(a, x4), h(a, x~)),
[1] {x4} = (b),
[1] {x~} = o};
T: ({x} = (f(x,, x,, x2, x,)), 
{x,} = (g(x2, x3)));
U: {[0] {x4, xs} = (b)};
T: ({x} = (f(x,, xl, x2, x4)),
{x,} = (g(x2, x3)),
{x2, x3} = (h(a, x4)));
U: ~;
```

```

T: ({x} = (f(x1, x1, x2, x4)),
{x1} = (g(x2, x3)),
{x2, x3} = (h(a, x4)),
{x4, x~} = (b)).

```

8.5 IMPROVING THE UNIFICATION ALGORITHM FOR NONUNIFYING DATA

In the case of nonunifying data, Algorithm 3 can stop with failure in two ways: either in step (1.1) if a cycle has been detected, or in step (1.2.1) if a clash occurs. In this section we show how to anticipate the latter kind of failure without altering the structure of the algorithm.

Let us first give the following definition. Two terms are consistent iff either at least one of them is a variable or they are both nonvariable terms with the same root function symbol and pairwise consistent arguments. This definition can be extended to the case of more than two terms by saying that they are consistent iff all pairs of terms are consistent. For instance, the three terms $f(x, g(a, y))$, $f(b, x)$, and $f(x, y)$ are consistent although they are not unifiable.

We now modify Algorithm UNIFY by requiring all terms in the right-hand side of a multiequation to be consistent, for every multiequation. Thus, we stop with clash failure as soon as this requirement is not satisfied. This new version of the algorithm is still correct since, if there are two inconsistent terms in the same multiequation, they will never unify.

In this way, clashes are detected earlier. In fact, in the Algorithm 3 version of UNIFY a clash can be detected while computing the common part and the frontier of the right-hand side of the selected multiequation, whereas in the new version of UNIFY the same error is detected in the compactification phase of a previous iteration.

An efficient implementation of the consistency check when two multiequations are merged requires a suitable representation for right-hand sides of multiequations. Thus, instead of choosing the obvious solution of representing every righthand side as a list of terms, we represent it as a multiterm, defined as follows.

A multiterm can be either empty or of the form $f(P_1 \dots P_n)$ where $f \in A$, and P_i ($i = 1 \dots n$) is a pair (S_i, M_i) consisting of a set of variables S_i and a multiterm M_i . Furthermore, S_i and M_i cannot both be empty.

For instance, the multiset of consistent terms

$$(f(x, g(a, y)), f(b, x), f(x, y))$$

can be represented with the multiterm

$$f(((x), b), (x, y, g((O, a), ((y),)))).$$

By representing right-hand sides in this way we have no loss of information, since the only operations which we have to perform on them are the operation of merging

two right-hand sides and the operation of computing the common part and the frontier, which can be described as follows:

```
MERGE (M', M" ) =
case M' of
0: M";
f'((Si M~), , tS, M' ~"
case M" of
0: M';
f"((S~,M~) ..... (Sn", M~")):
iff' -- f" and MERGE(M~, M[ ]) # failure (i -- 1 ..... n)
then f'((Si 0 S~, MERGE(MI, M; )) ..... 
(S~ ~J S t'n, MERGE(M~, M,,))"~
else failure
endcase
endcase
COMMONPART(f((S1, M1) ..... (S,, Mn))) = f(P1, --., Pn)
where Pi = if Si = ~ then COMMONPART(Mi)
else ANYOF(SD (i = 1 ..... , n)
```

where function ANYOF(S~) returns an element of set Si.

```
UPart = record
MultEqNumber: Integer;;
ZeroCounterMultEq, Equations: ListOfMultEq
end;
System = TPSystem;
PSystem = record
T: ListOfMultEq;
U: UPart
end;
MultiTerm = ~PMultiTerm;
PMultiTerm = record
Fsymb: FunName;
Args: ListOfTempMultEq
end;
MultiEquation = ~PMultiEquation;
PMultiEquation = record
Counter, VarNumber: Integer;
S: ListOfVariables;
M: Multi Term
end;
TempMultEq = ~PTempMultEq;
PTempMultEq = record
```

```

S: QueueOfVariables;
M: MultiTerm
end;
Variable = TPVariable;
P Variable = record
Name: VarName;
M: MultiEquation
end;

FRONTIER(f((S,, M1) ..... (Sn, Mn) )) = F1 [..J ... (.J Fn
where Fi = if Si = 0 then FRONTIER(M/)
else {Si = Mi} (i = 1, ..., n).

```

Note that the common part and the frontier are defined only for nonempty multiterm and that they always exist.

8.6 IMPLEMENTATION

In order to describe the last details of our algorithm, we present here a PASCAL implementation. In Figure 1 we have the definitions of data types. All data structures used by the algorithm are dynamically created data structures connected through pointers. The UPart of a system has two lists of multiequations: Equations, which contains all initial multiequations, and ZeroCounterMultEq, which contains all multiequations with zero counter. Furthermore, the field MultEqNumber contains the number of multiequations in the UPart. A multiequation, besides having the fields Counter, S, and M, has a field VarNumber, which contains the number of variables in S and is used during compactification. The pairs Pi = (S i, Mi), which are the arguments of a multiterm, have type TempMultEq. Finally, all occurrences of a variable point to the same Variable object, which points to the multiequation containing it in its left-hand side.

Figure 2

```

procedure Unify(R: System);
var Mult: MultiEquation ;
Frontier: ListOf TempMultEq ;
begin
repeat
SelectMultiEquation(R ~. U, Mult);
if not(Mult^.M=Nil) then
begin
Frontier := Nil;
Reduce(Multi.M, Frontier);
Compact(Frontier, R ~. U)

```

```

end;
R ~.T := NewListOfMultEq(Mult, R ~.T)
until R ~.U.MultEqNumber = 0
end (*Unify*);
```

Figure 3

```

procedure SelectMultiEquation(var U: UPart; var Mult: MultiEquation);
begin
  if U.ZeroCounterMultEq = Nil then fail('cycle');
  Mult := U^.ZeroCounterMultEq^.Value;
  U^.ZeroCounterMultEq := U^.ZeroCounterMultEq^.Next;
  U^.MultEqNumber := U^.MultEqNumber - 1
end (* SelectMultiEquation *);
```

ject, which points to the multiequation containing it in its left-hand side. The types "ListOf...", not given in Figure 1, are all implemented as records with two fields: Value and Next. Finally, QueueOfVariables is an abstract type with operations CreateListOfVars, IsEmpty, HeadOf, RemoveHead, and Append, which have a constant execution time.

In Figure 2 we rephrase Algorithm UNIFY as a PASCAL procedure. Procedure SelectMultiEquation selects from the UPart of the system a multiequation which is "on top" of the partial ordering, by taking it from the ZeroCounterMultEq list. Its implementation is given in Figure 3.

Procedure Reduce, given in Figure 4, computes the common part and the frontier of the selected multiequation. This procedure modifies the right-hand side of this multiequation so that it contains directly the common part. Note that the frontier is represented as a list of TempMultEq instead of as a list of multiequations.

Finally, in Figure 5 we give procedure Compact, which performs compactification by repeatedly merging multiequations. When two multiequations are merged, one of them is erased, and thus all pointers to it from its variables must be moved to the other multiequation. To minimize the computing cost, we always erase the multiequation with the smallest number of variables in its left-hand side. Procedure MergeMultiTerms is given in Figure 6.

A detailed complexity analysis of a similar implementation is given in [13]. There it is proved that an upper bound to execution time is the sum of two terms, one linear with the total number of symbols in the initial system and another one $n \log n$ with the number of distinct variables.

Figure 4

```

procedure Reduce(M: MultiTerm; var Frontier: ListOfTempMultEq);
var Arg" ListOfTempMultEq;
begin
```

```

Arg := MT.Args;
while not(Arg = Nil) do
begin
if IsEmpty(Arg T. Value T.S) then Reduce(Arg0. Value T.M, Frontier)
else
begin
Frontier := NewListOfTempMultEq(Arg T. Value, Frontier);
ArgT. Value := NewTempMultEq( CreateQueueOfVars(HeadOf(Arg~. ValueT.S) ) )
end;
Arg := ArgT.Next
end
end (*Reduce* );

```

Here we want only to point out that the nonlinear behavior stems from the operation described above of moving all pointers directed from variables to multiequations, whenever two multiequations are merged. To see how this can happen, let us consider the problem of unifying the two terms

$$f(xl, x3, xs, xT, xl, xs, xl)$$

and

$$f(x2, x4, x6, x8, x3, x7, x5).$$

During the first iteration of Unify we get a frontier whose multiequations are the pairs $(xl, x2)$, $(x3, x4)$, $(x , x6)$, (xT, xs) , $(xl, x3)$, (xs, xT) , and (xl, xs) . By executing Compact with this frontier, we see that it moves one pointer for each of the first four elements of the frontier, two pointers for each of the next two elements, and four pointers for the last element. Thus, it has an $n \log n$ complexity.

As a final remark, we point out that we might modify the worst-case behavior of our algorithm with a different implementation of the operation of multiequation merging. In fact, we might represent sets of variables as trees instead of as lists, and we might use the well-known UNION-FIND algorithms [1] to add elements and to access them. In this case the complexity would be of the order of $m \cdot G(m)$, where G is an extremely slowly growing function (the inverse of the Ackermann function). However, m would be, in this case, the number of variable occurrences.

8.7 COMPARISONS WITH OTHER ALGORITHMS

In this section we compare the performance of our algorithm with that of two well-known algorithms: Huet's algorithm [7], which has an almost linear time complexity, and Paterson and Wegman's algorithm [15], which is theoretically the best having a linear complexity.

Figure 5

```
procedure Compact(Frontier: ListOfTempMultEq; var U: UPart);
var Vars: QueueOfVariables;
V: Variable;
Mult, Mult 1: MultiEquation ;
procedure MergeMultEq(var Mult: MultiEquation ; Mult 1: MultiEquation );
vat Multt: MultiEquation;
V: Variable;
Vars : L istOfVariab les ;
begin
if not(Mult = Mult 1) then
begin
if Mult T. VarNumber < Mult 1T. VarNumber then
begin
Multt := Mult;
Mult := Multl;
Mult 1 := Multt
end;
MultT.Counter := MultT.Counter + Multl^.Counter;
Mult T. VarNumber := Mult ^. VarNumber + Mult 1 T. VarNumber;
Vars := Mult 1 T.S;
repeat
V := Vars'~.Value;
Vars := VarsT.Next;
V ~.M := Mult;
Mult T.S := NewListOfVariables( V, Mult T.S)
until Vars = Nil;
MergeMultiTerms(MultT.M, Mult 1 T.M);
U.MultEqNumber := U.MultEqNumber- 1
end
end (*MergeMultEq* );
begin
while not(Frontier = Nil) do
begin
Vars := Frontier T. ValueT.S;
V := HeadOf(Vars);
RemoveHead( Vars);
Mult := VT.M;
MultT.Counter := MultT.Counter - 1;
while not IsEmpty(Vars) do
begin
V := HeadOf(Vars);
RemoveHead( Vars);
Multl ::= VT.M;
```

```

Mult l T.Counter := Mult l ~.Counter - 1;
MergeMultiEq(Mult, Mult 1)
end;
MergeMulti Terms(Mult T.M, Frontier T. Value^.M ) ;
ifMultT.Counter = 0 then
U.ZeroCounterMultEq := NewListOfMultEq(Mult, U.ZeroCounterMultEq);
Frontier := FrontierT.Next
end
end (*Compact* );

```

Figure 6

```

procedure MergeMultiTerms(var M: MultiTerm ; MI: MultiTerm);
var Arg, Arg1: ListOfTempMultEq;
begin
ifM = Nil then M := M1
else if not(M1 = Nil) then
begin
if not (M "f .Fsymb = M1 ~.Fsymb) then fail(' clash' )
else
begin
Arg := M^.Args;
Arg1 := MIT.Args;
while not(Arg = Nil) do
begin
Append(Arg^. Value^.S, Arg1 ^ . Value^.S);
MergeMultiTerms(Arg^. Value^.M, Arg1 ^ . Value^.M );
Arg := ArgT.Next;
Arg1 := Arg1T.Next
end
end
end
end (*MergeMultiTerms* );

```

As an example of the assertion made at the end of Section 2, let us give a sketchy description of the two algorithms using the terminology of this paper. Both algorithms deal with sets of multiequations whose left-hand sides are disjoint and whose right-hand sides consist of only one term of depth one, that is, of the form $f(x_1, \dots, x_n)$ where x_1, \dots, x_n are variables. For instance,

```

{x1} = f(x~, x3, x, );
{x2} --- a;
(xa) = g(x2);
(x4) = a;
(x5) ffi f(x6, xT, xs);

```

```

{x6} = a;
{x7} = g(xa);
(4)
{xs} = 0.

```

Furthermore, we have a set S of equations whose left- and right-hand sides are variables; for instance,

S: {x₁ = xs}.

A step of both algorithms consists of choosing an equation from S, merging the two corresponding multiequations, and adding to S the new equations obtained as the outcome of the merging. For instance, after the first step we have

```

{x1, xs} = f(x2, x3, x4);
{x2} = a;
{x3} = g(x2);
{x4} = a;
{x3} = a;
{xv} = g(xa);
{x8} = 0;
S: {x2 = x6, x3 ffi x7, x4 = xs}.

```

The two algorithms differ in the way they select the equation from S. In Huet's algorithm S is a list; at every step, the first element of it is selected, and the new equations are added at the end of the list. The algorithm stops when S is empty, and up to this point it has not yet checked the absence of cycles. Thus, there is a last step which checks whether the final multiequations are partially ordered.

The source of the nonlinear behavior of this algorithm is the same as for our algorithm, that is, the access to multiequations after they have been merged. To avoid this, Paterson and Wegman choose to merge two multiequations only when their variables are no longer accessible. For instance, from (5) their algorithm selects x₃ = x₇ because x₂ and xs are still accessible from the third and sixth multiequation, respectively, getting

```

{x1, xs} = f(x2, x3, x4);
{x2} = a;
{x3, xT} = g(x2);
{x4} = a;
{x6} == a;
{xs} = 0;
S: {x2 = xs, x2 = x6, x4 = xs}.

```

To select the multiequations to be merged, this algorithm "climbs" the partial ordering among multiequations until it finds a multiequation which is "on top"; thus the detection of cycles is intrinsic in this algorithm.

Let us now see how our algorithm works with the above example. The initial system of multiequations is

```

U: {[0] {x1, x5} = f(( {x2, x6}, 0), ({x3, xT}, gD), ({x4, xs}, ~)),
[2] {x2} -= a,
[1] {x3} = g(({x2}, 0)),
[1] (x4) = a,
[1] {x6} = a,
[1] {xT} = g({{xs}, 0}),
[2] {xs } = ~};
T: () .

```

The next step is

```

U: {[1] {x2, x6} = a,
[0] {x3, xT} = g(({x2, xs}, ~)),
[1] {x4, xs} = a};
T: ((x~, xs} = f(x2, x~, x4)),

```

and so on.

In this algorithm the equations containing the pairs of variables to be unified are kept in the multiterms, and the mergings are delayed until the corresponding multiequation is eliminated.

An important difference between our algorithm and the others is that our algorithm may use terms of any depth. This fact entails a gain in efficiency, because it is certainly simpler to compute the common part and the frontier of deep terms than to merge multiequations step by step. Note, however, that this feature might also be added to the other algorithms. For instance, by adding the capability of dealing with deep terms to Paterson and Wegman's algorithm, we essentially obtain a linear algorithm which was independently discovered by the authors [13].

In order to compare the essential features of the three algorithms, we notice that they can stop either with success or with failure for the detection of a cycle or with failure for the detection of a clash. Let P_m , P_c , and P_t be the probabilities of stopping with one of these three events, respectively. We consider three extreme cases:

(1) $P_m \gg P_c, P_t$ (very high probability of stopping with success). Paterson and Wegman's algorithm is asymptotically the best, because it has a linear complexity whereas the other two algorithms have a comparable nonlinear complexity.

However, in a typical application, such as, for example, a theorem prover, the unification algorithm is not used for unifying very large terms, but instead it is used a great number of times for unifying rather small terms each time. In this case we cannot exploit the asymptotically growing difference between linear and nonlinear algorithms, and the computing times of the three algorithms will be comparable, depending on the efficiency of the implementation.

An experimental comparison of these algorithms, together with others, was carried out by Trum and Winterstein [21]. The algorithms were implemented in the same language, PASCAL, with similar data structures, and tried on five different classes of unifying test data. Our algorithm had the lowest running time for all test data. In fact, our algorithm is more efficient than Huet's because it does not need a final acyclicity

test, and it is more efficient than Paterson and Wegman's because it needs simpler data structures.

(2) $P_c \gg P_t \gg P_m$ (very high probability of detecting a cycle). Paterson and Wegman's algorithm is the best because it starts merging two multiequations only when it is sure that there are no cycles above them. Our algorithm is also good because cycle detection is embedded in it. In contrast, Huet's algorithm must complete all mergings before being able to detect a cycle, and thus it has a very poor performance.

(3) $P_t \gg P_c \gg P_m$ (very high probability of detecting a clash). Huet's algorithm is the best because, if it stops with a clash, it has not paid any overhead for cycle detection. Our algorithm is better than Paterson and Wegman's because clashes are detected during multiequation merging and because our algorithm may merge some multiequations earlier, like x_2, x_6 and x_4, X_s in the above example. On the other hand, mergings which are delayed by our algorithm, by putting them in multiterms, cannot be done earlier by the other algorithm because they refer to multiequations which are still accessible. The difference in the performance of the two algorithms may become quite large if terms of any depth are allowed.

8.8 CONCLUSION

A new unification algorithm has been presented. Its performance has been compared with that of other well-known algorithms in three extreme cases: high probability of stopping with success, high probability of detecting a cycle, and high probability of detecting a clash. Our algorithm was shown to have a good performance in all the cases, and thus presumably in all the intermediate cases, whereas the other algorithms had a poor performance in some cases.

Most applications of the unification algorithm, such as, for instance, a resolution theorem prover or the interpreter of an equation language, require repeated use of the unification algorithm. The algorithm described in this paper can be very efficient even in this case, as the authors have shown in [12]. There they have proposed to merge this unification algorithm with Boyer and Moore's technique for storing shared structures in resolution-based theorem provers [3] and have shown that, by using the unification algorithm of this paper instead of the standard one, an exponential saving of computing time can be achieved. Furthermore, the time spent for initializations, which might be heavy for a single execution of the unification algorithm, is there reduced through a close integration of the unification algorithm into the whole theorem prover.

REFERENCES

1. 1. AHO, A.V., HOPCROFT, J.E., AND ULLMAN, J.D. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Mass., 1974.
2. 2. BAXTER, L.D. A practically linear unification algorithm. Res. Rep. CS-76-13, Dep. of Applied Analysis and Computer Science, Univ. of Waterloo, Waterloo,

3. 3. BOYER, R.S., AND MOORE, J.S. The sharing of structure in theorem-proving programs. In Machine Intelligence, vol. 7, B. Meltzer and D. Michie (Eds.). Edinburgh Univ. Press, Edinburgh, Scotland, 1972, pp. 101-116.
4. 4. BURSTALL, R.M., AND DARLINGTON, J. A transformation system for developing recursive programs. J. ACM 24, 1 (Jan. 1977), 44-67.
5. 5. CHANG, C.L., AND LEE, R.C. Symbolic Logic and Mechanical Theorem Proving. Academic Press, New York, 1973.
6. 6. HEWITT, C. Description and Theoretical Analysis (Using Schemata) of PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot. Ph.D. dissertation, Dep. of Mathematics, Massachusetts Institute of Technology, Cambridge, Mass., 1972.
7. 7. HUET, G. R6solution d'6quations dans les langages d'ordre 1, 2 0:. Th se d'6tat, Sp6cialit6 Math matiques, Universit Paris VII, 1976.
8. 8. HUET, G.P. A unification algorithm for typed ?, -calculus. Theor. Comput. Sci. 1, 1 (June 1975), 27-57.
9. 9. KNUTH, D.E., AND BENDIX, P.B. Simple word problems in universal algebras. In Computational Problems in Abstract Algebra, J. Leech (Ed.). Pergamon Press, Eimsford, N.Y., 1970, pp. 263-297.
10. 10. KOWALSKI, R. Predicate logic as a programming language. In Information Processing 74, Elsevier North-Holland, New York, 1974, pp. 569-574.
11. 11. LEVI, G., AND SIROVICH, F. Proving program properties, symbolic evaluation and logical procedural semantics. In Lecture Notes in Computer Science, vol. 32: Mathematical Foundations of Computer Science 1975. Springer-Verlag, New York, 1975, pp. 294-301.
12. 12. MARTELLI, A., AND MONTANARI, U. Theorem proving with structure sharing and efficient unification. Internal Rep. S-77-7, Ist. di Scienze della Informazione, University of Pisa, Pisa, Italy; also in Proceedings of the 5th International Joint Conference on Artificial Intelligence, Boston, 1977, p. 543.
13. 13. MARTELLI, A., AND MONTANARI, V. Unification in linear time and space: A structured presentation. Internal Rep. B76-16, Ist. di Elaborazione delle Informazioni, Consiglio Nazionale delle Ricerche, Pisa, Italy, July 1976.
14. 14. MILNER, R. A theory of type polymorphism in programming. J. Comput. Syst. Sci. 17, 3 (Dec. 1978), 348-375.
15. 15. PATERSON, M.S., AND WEGMAN, M.N. Linear unification. J. Comput. Syst. Sci. 16, 2 (April 1978), 158-167.

16. 16. ROBINSON, J.A. Fast unification. In Theorem Proving Workshop, Oberwolfach W. Germany, Jan. 1976.
17. 17. ROBINSON, J.A. Computational logic: The unification computation. In Machine Intelligence, vol. 6, B. Meltzer and D. Michie (Eds.). Edinburgh Univ. Press, Edinburgh, Scotland, 1971, pp. 63-72.
18. 18. ROBINSON, J.A. A machine-oriented logicbased on the resolution principle. J. ACM 12, 1 (Jan. 1965), 23-41.
19. 19. SHORTLIFFE, E.H. Computer-Based Medical Consultation: MYCIN. Elsevier North-Holland, New York, 1976.
20. 20. STICKEL, M.E. A complete unification algorithm for associative-commutative functions. In Proceedings of the 4th International Joint Conference on Artificial Intelligence, Tbilisi, U.S.S.R., 1975, pp. 71-76.
21. 21. TRUM, P., AND WINTERSTEIN, G. Description, implementation, and practical comparison of unification algorithms. Internal Rep. 6/78, Fachbereich Informatik, Univ. of Kaiserslautern, W. Germany.
22. 22. VENTURINI ZILLI, M. Complexity of the unification algorithm for first-order expressions. Calcolo 12, 4 (Oct.-Dec. 1975), 361-372.
23. 23. VON HENKE, F.W., AND LUCKHAM, D.C. Automatic program verification: A methodology for verifying programs. Stanford Artificial Intelligence Laboratory Memo AIM-256, Stanford Univ., Stanford, Calif., Dec. 1974.
24. 24. WALDINGER, R.J., AND LEVITT, K.N. Reasoning about programs. Artif. Intell. 5, 3 (Fall 1974), 235-316.
25. 25. WARREN, D.H.D., PEREIRA, L.M., AND PEREIRA, F. PROLOG-The language and its implementation compared with LISP. In Proceedings of Symposium on Artificial Intelligence and Programming Languages, Univ. of Rochester, Rochester N.Y., Aug. 15-17, 1977. Appeared as joint issue: SIGPLAN Notices (ACM) 12, 8 (Aug. 1977), and SIGART Newslet. 64 (Aug. 1977), 109-115.

Received September 1979; revised July 1980 and September 1981; accepted October 1981

Глава 9

Parsing and Compiling Using Prolog

pdf ¹

© JACQUES COHEN and TIMOTHY J. HICKEY
Brandeis University

Abstract

This paper presents the material needed for exposing the reader to the advantages of using Prolog as a language for describing succinctly most of the algorithms needed in prototyping and implementing compilers or producing tools that facilitate this task. The available published material on the subject describes one particular approach in implementing compilers using Prolog. It consists of coupling actions to recursive descent parsers to produce syntax-trees which are subsequently utilized in guiding the generation of assembly language code. Although this remains a worthwhile approach, there is a host of possibilities for Prolog usage in compiler construction. The primary aim of this paper is to demonstrate the use of Prolog in parsing and compiling. A second, but equally important, goal of this paper is to show that Prolog is a labor-saving tool in prototyping and implementing many nonnumerical algorithms which arise in compiling, and whose description using Prolog is not available in the literature. The paper discusses the use of unification and nondeterminism in compiler writing as well as means to bypass these (costly) features when they are deemed unnecessary. Topics covered include bottom-up and top-down parsers, syntax-directed translation, grammar properties, parser generation, code generation, and optimizations. Newly proposed features that are useful in compiler construction are also discussed. A knowledge of Prolog is assumed.

Categories and Subject Descriptors: D.1.O [Programming Techniques]: General; D.2.m [Software Engineering]: Miscellaneous—rapid prototyping; D.3.4 [Programming

¹ © <https://pdfs.semanticscholar.org/dd8d/c0deb336d90912a21ba8ec6f6c6fef4b4024.pdf>

Languages]: Processors; F.4.1. [Mathematical Logic and Formal Languages]: Mathematical Logic–logic programming 1.2.3 [Artificial Intelligence]: Deduction and Theorem Proving logic programming

General Terms: Algorithms, Languages, Theory, Verification

Additional Key Words and Phrases: Code generation, grammar properties, optimization, parsing

This work was supported by the NSF under grant DCR 8590881.

Authors' address: Computer Science Department, Ford Hall, Brandeis University, Waltham, MA 02254.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

0 1987 ACM 0164-0925/87/0400-0125 \$00.75

ACM Transactions on Programming Languages and Systems, Vol. 9, No. 2, April 1967, Pages 125-163.

9.1 INTRODUCTION

The seminal paper by Alain Colmerauer on Metamorphosis Grammars first appeared in 1975 [9]. That paper spawned most of the developments in compiler writing using Prolog, a great many of them due to David H. D. Warren. Warren's thesis [30], the paper summarizing it [31], and the related work on Definite Clause Grammars [25] are practically the sole sources of reference on the subject.²

The available published material on the subject describes one particular approach in implementing compilers using Prolog. It consists of coupling actions to recursive descent parsers to produce syntax-trees which are subsequently utilized in guiding the generation of assembly language code. Although this remains a worthwhile approach, there is a host of possibilities for Prolog usage in compiler construction. The primary aim of this paper is to present the material needed for exposing the reader to the advantages of using Prolog in parsing and compiling. A second, but equally important, goal of this paper is to show that Prolog is a labor-saving tool in prototyping and implementing many nonnumerical algorithms which arise in compiling, and whose description using Prolog is not available in the literature. Finally, a third goal is to present new approaches to compiler design which use proposed extensions to Prolog.

This paper is directed to compiler designers moderately familiar with Prolog, who wish to explore the advantages and present drawbacks of using this language for implementing language processors. The advantages of Prolog stem from two important features of the language.

² A recent book edited by Campbell [3] mostly covers the implementation of Prolog itself.

(1) The use of unification as a general pattern-matching operation allowing procedure parameters (logical variables) to be both input and output or to remain unbound. Unification replaces the conditionals and assignments which exist in most languages.

(2) The ability to cope with nondeterministic situations, and therefore allow the determination of multiple solutions to a given problem.

From a subjective point of view, the main advantage of Prolog is that the language has its foundations in logic, and it therefore encourages the user to describe problems in a logical manner which facilitates the checking for correctness, enhances program readability, and reduces the debugging effort. It will be seen that unification and nondeterminism play an important role in compiler design; however, using their full generality is often costly and unnecessary. These issues are discussed throughout the paper whenever they become relevant. Remarks are made in the last section about the efficiency of Prolog-written compilers and the means to improve their performance.

The Prolog proficiency assumed in this paper can be acquired by reading the first few chapters of either Kowalski's [20] or Clocksin and Mellish's [6] books. In particular, the reader should be at ease with elementary list processing and with the predicate append. The concrete syntax used in this paper is that of Edinburgh Prolog [6]. It is also assumed that the reader is familiar with compiler design topics such as parsing, lexical analysis, code generation, optimizations, and so on. These topics are covered in standard texts [1, 17, 29].

9.2 PARSING

In this section we present parsers belonging to two main classes of parsing algorithms, namely, bottom-up and top-down. Due to the backtracking capabilities of Prolog, these parsers can in general handle nondeterministic and ambiguous languages. An early paper by Griffiths and Petrick [18] describes various parsing algorithms and their simulation by automata. There the amount of nondeterminism is roughly specified by a selectivity matrix which guides the parser in avoiding states leading to backtracking. A similar situation occurs in the Prolog parsers described here. In compilers, interest is commonly restricted to deterministic languages. Backtracking may be prevented by a judicious use of cuts(!) and/or by introducing assertions in the database that guide the parser in avoiding dead-ends.

A word about notation is in order. The grammar conventions are those in [1]. Edinburgh Prolog uses capital letters as variables, and therefore capitals cannot be used to represent nonterminals unless they are quoted. In this paper, the terms t() and n() denote, respectively, terminals and nonterminals. Quoting may be necessary for specifying certain terminals (e.g., parentheses). For example, the right-hand side (rhs) of the rule

$$F \rightarrow (E)$$

is described by the list

$$[WV), de), W)^{'} l.$$

Whenever stacks are used, they are also represented by lists whose leftmost element is the top of the stack.

This section does not pretend to make an exhaustive treatment of parsers. We describe bottom-up and top-down parsers for both nondeterministic and deterministic languages. A nondeterministic shift-reduce and a deterministic weak-precedence parser are the bottom-up representatives. Their top-down counterparts are, respectively, a predictive and an LL(1) parser. A recursive descent version of the latter is also considered. Besides those described herein, we have programmed and tested Earley's algorithm [13] and a parser generator that produces the necessary tables for parsing SLR(1) grammars [11].

9.2.1 Bottom-Up

A very simple (albeit inefficient) shift-reduce parser can be readily programmed in Prolog. Its action consists of attempting to reduce whenever possible; otherwise the window is shifted on to a stack and repeated reductions (followed by shifts) take place until the main nonterminal appears by itself in the top of the stack. Note that a reduction may be immediately followed by other reductions. A reduction corresponds to the recognition of a grammar rule; for instance, the reduction for the rule $E + E + T$ occurs when $E + T$ lies on the top of the stack. It is then replaced by an E . This action is expressed by the unit clause

$$\text{redme}(Idt), H+), n(e)IXl, Me)IXl).$$

Let us consider the classical grammar describing arithmetic expressions:

$$G1 : E + E + T$$

$$E - T$$

$$T - +T * F$$

$$T + F$$

$$F + (E)$$

$$F + (\text{letter})$$

The appropriate sequence of reduce clauses follows immediately from the above rules. To decrease the amount of backtracking it is convenient to order these clauses so that rules with longer right-hand sides are tried before those with shorter rhs. We are now ready to present the parser. It has two parameters: (1) a list representing the string being parsed, and (2) the list representing the current stack.

```
% try-reduce %
sr-parse(Input, Stack) :- reduce(Stack, NewStack),
% try-shift %
sr-parse(Input, NewStack).
sr-parse([ Window I Rest], Stack) :- sr-parse(Rest, [Window 1 Stack]).
```

Assume that a marker (\$) is to be placed at the end of each input string. The following acceptance clause accepts a string only when the marker is in the window and the stack contains just an E.

```
% acceptance %
sr-paWL§1, Me)lh
```

Consider the input string a^*b . We assume that a scanner is available to translate it into the suitable list, understandable by the parser. Then the query

```
?- sr-PamWa), t(*), t(b), $1, [ I ].
```

will succeed.

The above parser is very inefficient, since it relies heavily on backtracking to eventually accept or refuse a string. Note that in parsing the string a^*b , $t(a)$ is first shifted and successively reduced to an F, T, and (even) an E; the latter being a faulty reduction. The parser is, however, capable of undoing these reductions through backtracking. This inordinate amount of backtracking can be controlled by a careful selection of the reductions and shifts that eliminate possible blind-alleys. This is done in our next bottom-up parser, which is the weak-precedence type [1,19].

The basic strategy is to consult a table made of unit clauses like

$$try-reduce(Top-of-stack, Window). \text{and} try-shift(Top-of-stack, Window).$$

which command a reduction or a shift, depending on the elements lying on the window and on the top of the stack. The problem of automatically generating the above clauses from the grammar rules is addressed in Section 5. The weakprecedence relations for the grammar G_i are represented by the clauses

$$try_{reduce}(n(t), \$).$$

$$try_{reduce}(n(f), \$9.$$

...

$$try-reduce(t('`), t(+)).$$

$$&-reduce(t('`), t(``)).$$

and

$$try-shift(t(+), t(``)).$$

$$tryshift(n(e), t(+)).$$

and so on.

We now transform the previous sr-parser into a wp-parser which takes advantage of the additional information to avoid backtracking. Using Griffiths and Petrick's terminology [18], these unit clauses render the algorithms selective.

```

% acceptance %
w-~~~~4$1, [de)1).
% try-reduce %
wp-parse([ W 1 Input], [S 1 Stack]) :- try_reduce(S, W),
reduce([S 1 Stack], NewStack),
wp-purse([ W 1 Input], NewStack).
% try shift %
wp-parse([ WI Input], [S 1 Stack]) :- try-shift(S, W),
wp-purge(Zrzput, [W, S 1 Stack]).
```

Notice that if a grammar is truly a weak-precedence grammar (i.e., there are no precedence conflicts and rules have distinct rhs), then backtracking will only occur when try-reduce fails and try-shift has to be tried. Thus the query

```
?- wp-parse(Znput, [ I], print(accept)).
```

will print “accept”, and succeed if the Input string is in the language. If the string is not in the language, the query will fail. The time complexity is proportional to the length of Input. Error detection and recovery are discussed in Section 9.

A comment about the efficiency of this version of wp-parse is in order. Since there will in general be a large number of try-reduce and try-shift rules, the execution time of the wp-parser could be significantly reduced if a Prolog compiler could branch directly to a clause having the appropriate constant as its first term (for example, by constructing a hashing table at compile time). Recent and planned Prolog optimizing compilers can indeed perform this branching [30]. The reader should also refer to [21] for a discussion of optimizations applicable to deterministic Prolog programs, which render their efficiency closer to those of conventional programs.

Finally, note that it would be straightforward to extend this type of parser to cover the syntactical analysis of bounded-context grammars, that is, those for which a decision to reduce or shift is based on an inspection of m elements in the top of the stack and a look-ahead of n elements in the input string.

9.2.2 Top-Down

A Prolog implementation of predictive parsers [1] follows readily from the programs described in the previous section. The grammar G2, below, generates the same language as G1, but left-recursion has been replaced by right-recursion.

$$Gz : E - +TE'$$

$$E' - + + TE'$$

$$E'3e$$

$$T + FT'$$

$$T' -- B * FT'$$

$T' + E$ $F * U - O$ $F + (\text{letter})$

The above rules are placed in the database using the unit clauses

```
rule(Non-terminal, Rhs).
```

Examples are

```
ruMn(tprim), [t(*), n(f), nt@hne)l].
rule(n(tprime), [ I]).
```

The parser predict(Input, Stack) has the same parameters as its predecessors, namely: (1) the input string and (2) the current stack contents (initially n(e), where E is the main nonterminal). The parser succeeds if the Input string is in the language, and fails otherwise.

The basic action of predict is to replace a nonterminal on the top of the parse stack by the rhs of the rule defining that nonterminal. If a terminal element lies on the top of the stack and if it matches the element W in the window, then parsing proceeds by popping W and considering the next element of the input string to be in the window. A string is accepted when the stack is empty and the window contains the marker. In Prolog we have

```
% acceptance.
predict(I$1, 1 I).
% try a possible rule.
predict(linput, [n(N) 1 Stack]) :-
ruldn(N), Rh),
append(Rhs, Stack, NewStack),
predict&put, NewStack).
% match the terminals.
predict([t(W) 1 Input], [t(W) 1 Stack]) :- predict(linput, Stack).
```

The above parser can handle nondeterministic or even ambiguous grammars, but may become trapped in an infinite recursion loop if the grammar is left-recursive.

To improve the efficiency when processing deterministic grammars, one could again resort to placing additional information in the database. This is the case for the next parser we consider, which is applicable to LL(1) grammars, and does not rely on backtracking. It will become apparent in Section 5 that it is straightforward to generate tables for LL(1) grammars [1]. These tables have as entries the contents of the window t(W) and the nonterminal n(N) on the top of the stack, and they specify the appropriate (unique) replacement by the rhs of the rule defining N. Entries may be defined by unit clauses of the form

```
entv(t(W), n(N), Rhs).
```

for all pairs (W, N) such that $N \in L$. An LL(1) deterministic parser is obtained by replacing the middle clause of predict by

```
predict([t(W) 1 Input], [n(N) 1 Stack]) :- entry(t(W), n(N), Rhs),  
append(Rhs, Stack, NewStack),  
predict([t(W) I Input], NewStack).
```

By properly selecting one among multiple entries, predict can deterministically parse languages defined by ambiguous grammars, as is the case of the if then else construct considered in [1, p. 191]. Moreover, the parser does not rely on backtracking to accept a string. The complexity of the LL(1) parser is therefore $O(n)$ where n is the length of the input string.

9.2.3 Recursive Descent

All of the previously described parsers contain a general nucleus which drives the parsing, the grammar rules being specified by unit clauses in the database. Parser efficiency can be increased by establishing a direct mapping between grammar rules and Prolog clauses. This is accomplished as in recursive descent parsing: each procedure directly corresponds to a given grammar rule. As usual, left-recursion is not allowed and has to be replaced by right-recursion to avoid endless loops.

There are three manners in which these parsers can be implemented in Prolog, depending on the form of the input string. The first and the least efficient of these is the one that uses the predicate append. The second uses links to define the input string that appears as unit clauses in the database. Finally, the third, which uses difference lists, is the most efficient, as will be seen by estimates of the various complexities. The implementation of these versions is illustrated using the grammar G3, generating a'kb", n I 0. The notation $t(T)$ and $n(N)$ will no longer be needed to differentiate between terminals and nonterminals, since the nonterminals will be transformed into Prolog procedures which manipulate terminal strings.

$$G3 : S + aSb$$

$$S4C$$

Every grammar rule is transformed into a clause whose argument is the list of terminals derived from the defined nonterminal. Terminals are similarly handled using unit clauses. We have

```
s(ASB) :- append(A, SB, ASB),  
~PP~W, B, SB),  
a(A),  
SW,  
MB).  
s(C) :- c(C).  
m1).
```

WI).
4c1).

The appends are used to partition the list ASB as the concatenation of three sublists A, S, B. Although the only partition for which the parser will succeed is

$$A = a, S = an - lcbnml, B = b,$$

this program will generate at least $2n$ incorrect partitions. Hence the number of calls needed to append is at least n^2 . Note that the appends should precede the calls of $a(A)$, $s(S)$, $b(B)$. Otherwise, an infinite loop would occur. The above program can be optimized by symbolic execution: the terms $a(A)$, $b(B)$, and $c(C)$ can be directly replaced by their unit clause counterparts, yielding

```
s(ASB) :- append([a], SB, ASB),  
wwMS, PI, SB),  
SW.  
m1).
```

The second approach for programming recursive descent parsers in Prolog is the use of links. An input string such as [a, a, c, b, b] is represented by the unit clauses $link(i, t, i + 1)$, stating that there is a terminal t located between positions i and $i + 1$. In our case the input string aacbb becomes $hk(1, a, 2)$.

```
link(2, a, 3).  
link(3, c, 4).  
link(4, b, 5).  
link(5, b, 6).
```

A clause recognizing a nonterminal will now have two parameters denoting the leftmost and rightmost positions in the input string that will parse into the given nonterminal. In our particular example we have

```
s(X1, X4) :- link(X1, a, X2),  
s(X2, X3),  
link(X3, b, X4).  
s(X1, X2) :- link(X1, c, X2).
```

The as will be consumed by the n successive calls of the first two literals. Then, only the second clause is applicable and the c is consumed. Finally, the unbound variables X3, X4 are successively bound to the points separating the remaining bs. The algorithm's complexity is therefore linear.

An efficient implementation of recursive descent parsers in Prolog makes use of difference lists. If a nonterminal A generates a terminal string $a!$ (i.e., $A == a^* a$), that string can be represented by the difference of two lists U and V; V is a sublist of U which has the same tail as U. For example, if U is [a, c, b, b, b] and V is [b, b] the difference $U - V$ defines the list [a, c, b], which for G3 parses into an S. Warren [31] points out that the use of difference lists corresponds to having the general link-like clause:

```
link([H 1 2'1, H, T)
```

which can be read as “the string position labelled by the list with head H and tail T is connected by a symbol H to the string position labelled T.” A parser for G3 using difference lists can be written as follows:

```
s(U, V) :- a(U, V1), s(V1, V2), b(V2, V).  
s(w, 2) :- c(w, 2).
```

For the terminals a, b, and c we have

```
a(I VII, vu).  
b([b I U21, U2]).  
c(k I u31, U3).
```

Symbolic execution allows us to find the values of U and V1 in the first clause:

```
U=[a|U1], V1=U1
```

Similarly,

```
V2=[b|U2], V=U2  
W=[c|U3], Z=U3
```

Substituting the values of the above variables, we obtain the optimized program

```
s([a 1 U1], U2) :- s(U1, [b I U2]).  
4kl u31, U3).
```

(The above program could also have been derived using symbolic execution by considering the first version of the parser with append and noticing that if X - Y and Y - Z are difference lists, then append(X - Y, Y - Z, X - Z) is a fact.)

Let us follow the execution of the call

$$s(b, 0, c, b, bl, [I]).$$

Notice that U1 becomes [a, c, b, b] and U2 is []. The next calls of S are

$$([a, c, hbl, PII$$
$$s(k, hbl, P, bl)$$

This last call matches only the second clause thus indicating a valid string. An informal English description of the acceptance is as follows: successively remove each a in the head of the first of the difference lists and add a b to the second one. A string is accepted when no more as can be removed, the head of the first list is a c, and the two lists contain the same number of bs. Therefore, for this particular grammar, G3, the parsing is done in linear time with no backtracking. The reader might have already surmised that the use of difference lists and of symbolic executions illustrated in this example could be carried out automatically from the given grammar rules. Clocksin and Mellish ([6, 1st ed., p. 237-2381] present a short Prolog program that does the translation.

9.3 SYNTAX-DIRECTED TRANSLATION

This type of translation consists of triggering semantic actions specified by the programmer once selected syntactic constructs are found by a parser. In the case of the bottom-up parsers described in Section 2.1, it suffices to add a third parameter to the reduce clauses specifying the rule number and to modify the parser so that a semantic action (specified by the rule number) will take place just after the reduction. For example, in order to translate arithmetic expressions into postfix Polish notation, the corresponding reduce for the first rule of G1 becomes

```
reduce(W), t(+), n(e) I X1, [n(e)] I X1, 0
```

The modified parser contains two additional parameters: (1) a stack, Sem, which will be manipulated by the action procedure and (2) a parameter, Result, which will be bound to the final result of the semantic actions:

```
% accept and bind Result to the semantic parameter.  
wp-translate([$_], [n(e)], Result, Result).  
% try to perform a reduction and a semantic action.  
wp-transkzte([ W 1 Input], [S I Stack], Sem, Result) :-  
&y-reduce@, W),  
reduce([S I Stack], NewStack, RuleNumber),  
action(RuleNumber, [S I Stack], Sem, NewSem),  
wp-translute([ W I Input], NewStack, NewSem, Result).  
  
% try a shift.  
wp-transZate([ W 1 Input], [S 1 Stack], Sem, Result) :-  
try_shift(S, W),  
wp-translute(Zinput, [W, S 1 Stack], Sem, Result).
```

The parser can then be equipped with actions by adding rules which specify how the temporary semantic parameter is to be modified for each rule. The following action procedure constructs parse trees for the arithmetic expressions defined by grammar G1:

```
syntax-tree(Znput, Tree) :- wp-trandute(Znput, [ 1, [ 1, Tree]).  
action(1, Stack, [X1, X2 I T], [plus(X2, X1) IT]).  
action(3, Stack, [X1, X2 I 2'1, [times(X2, X1) I 2'1]).  
action(6, [t(Z,etter) I Stack], Temp, [Letter I Temp]).  
action(X, Stack, Temp, Temp) :- X # 1, X # 3, X # 6.
```

The body of the last clause guarantees that no spurious actions are performed should backtracking ever occur. Notice that the action procedure must have access to the parsing stack (as is the case for rule 6) so that specific terminals may be incorporated into the actions. A similar strategy is applicable in adding actions to predictive parsers.

All of the above descriptions of semantic actions utilize inherited attributes and are admittedly standard. The main purpose of presenting them here is to point out how succinct the descriptions become when Prolog is used. The truly novel way of performing syntax-directed translation is that pioneered by Colmerauer and widely utilized by Warren. That approach does not strictly separate syntax from semantics as was done in this section. They have added new parameters to the recursive descent parser described in Section 2.3, so that the translation takes full advantage of the unification and goal-seeking features of Prolog. Colmerauer's approach is the subject of the next section.

9.4 M-GRAMMARS AND DCGs

A metamorphosis (or M-) grammar is a formalism which combines a Chomskytype language definition with logic programming capabilities for manipulating the semantic attributes needed to perform syntax-directed translations. Colmerauer [9] maps general type-0 Chomsky rules into general logicprogramming clauses, (i.e., those that may contain more than one predicate in the left-hand side). A very useful subset of M-Grammars are Definite Clause Grammars (DCGs), which are based on Chomsky's context-free grammars. The reader has undoubtedly noticed the similarity between Prolog clauses and context-free grammar rules: they both have one term in the lhs and several (or none, i.e., t) in the rhs. Prolog restricts itself to those special clauses called Horn or Definite clauses, thus explaining the acronym. It will be seen shortly that although DCGs are based on context-free grammars they are able to parse context-sensitive ones as well. (In fact, any recursively enumerable language can be recognized using DCGs with parameters.)

DCGs are translated directly into Prolog clauses which include a recursive descent parser using difference lists. For example, the DCG rules for recognizing strings in G3 are

$$s -- + [cl.$$

$$s -- + [a], s, [b].$$

The syntax of DCGs is close to that of Prolog clauses. The ‘:-’ is replaced by ‘-+’, and terminals appear within square brackets. Most Prolog interpreters automatically translate the above into the clauses:

$$s([cIm, LO).$$

$$s([a1LO], Ll) : -s(L0, [bILL]).$$

which have already been explained in Section 2.3. DCG terms usually contain one or more arguments which are directly copied into their Prolog counterparts, which also contain the difference list parameters. Our first example of usage of DCGs is to determine the value of n for a given input string a”&” (generated by grammar GB).

$$s(0) -- + [cl.$$

$$s(succW) -- + [al, SW), PI.$$

The added argument specifies that the recognition of a c implies a value of N = 0. Each time an s surrounded by an a and a b is recognized, the value of N increases by one (succ indicates the successor). The above DCGs are automatically translated into

$$SK4[cIw, LO).$$

$$s(succ(N), [a1LO], Ll) : -s(N, LO, [bILL]).$$

The call $s(X, [a, a, c, b, b], [])$ yields $X = \text{succ}(\text{succ}(0))$. The backtracking capabilities of Prolog allow the call $s(\text{succ}(\text{succ}(O)), X, [])$ which yields $X = [a, a, c, b, b]$.

By employing a technique similar to the one illustrated by the previous example, we can construct a parser s to recognize the language "a"b"c". It uses the auxiliary procedure sequ.ence(X, N) (defined below) which parses a list of Xs and binds N to the number of Xs found.

$$\text{sequence}(X, 0) -- + [1.$$

$$\text{sequence}(X, \text{succ}(N)) -- + [Xl, \text{sequeme}(X, IV)$$

$$s(N) -- + \text{sequence}(a, N), \text{sequence}(b, N), \text{sequeme}(c, IV)$$

Let us now consider the use of DCGs for translating arithmetic expressions into their syntax-trees. We start with the simplified right-recursive grammar rules:

$$E - T + E$$

$$E - T$$

$$T - a$$

Initially one would be tempted to use the DCG

$$eWw(X, Y)) -- + W3, [+I, e(y).$$

$$e(X) -- + t(X).$$

$$t(a) -- 9[a].$$

These rules, however, translate $a + a + a$ into plus (a, pZus(u, a)) which is rightassociative and therefore semantically incorrect. Some cunning is needed to circumvent this difficulty. Let us first rewrite the grammar rules as

$$E + TR$$

$$R + +TR$$

$$R-, C$$

$$T + a$$

Our goal is to generate plus(plus(a, a), a) for the input string a + a + a. The following DCG will do the proper translation:

$$eqv\$?3) -- 9term(Tl), restexpr(T1, E).$$

$$restexpr(T1, E) -- a[+I, term(T2), restexpr(plus(T1, T2), E)].$$

$$restexpr(\&, E) -- + [1.$$

$$term(a) -- 3[a].$$

When the clause expr recognizes the first term T1 in the expression, it passes this term to the second clause, restexpr. If there is another term T2 following T1, then the composite term pb(T1, T2) is constructed and recursively passed to restexpr. The first parameter of restexpr is used to build a left-recursive parse tree, which is finally transmitted back to erpr by the third clause.

Unfortunately, the above “contortions” are needed if one insists on using a recursive descent parser to construct a left-associative syntax tree. This particular Prolog technique has become a standard idiom among DCG writers. A way out of this predicament is to implement DCGs using bottom-up parsers. (This has been proposed in [28].) At present these capabilities are not generally available in existing Prolog interpreters and compilers.

It is straightforward to generalize the above translation by introducing multilevel grammar rules such as

$$Ei + TiRi$$

$$Ri + OpiTiRi$$

$$\& + \& + I$$

$$Ri + c$$

with 1 5 i 5 n and En+, + letter] (El), where i denotes the precedence of the operator opi. The corresponding DCG contains i as a parameter, and could allow for redefining the priorities of the operators, therefore rendering the language extensible. This approach is used in the Edinburgh version of Prolog.

A very useful feature of DCGs is that parts of Prolog programs may appear in their right-hand sides. This is done by surrounding the desired Prolog predicates within curly brackets. Our next example illustrates the use of this feature to perform the translation of arithmetic expressions into postfix notation directly by a DCG that does not construct syntax trees. Our first example of this technique will output the postfix notation.

$$e -- + t, r.$$

$$r -- + [+I, t, write(+)), r.$$

$$r -- + [1.$$

$$t -- + [a], (write(a)).$$

This procedure produces the postfix expression using side effects, and this technique can be a drawback. One solution to this problem is to use difference lists to simulate the write procedure. To each DCG clause corresponding to a nonterminal N we add difference list parameters representing the list of symbols that are output during the recognition of N. (These difference lists are in addition to those used in syntactic analysis). We have

$$e(F1, F3) -- at(F1, F2), r(F2, F3).$$

$$r(F1, F4) -- + [+I, t(F1, F2), writefik(+, F2, F3)), r(F3, F4).$$

$$r(F, F) -- + [1.$$

$$t(F1, F2) -- + [a], (writefik(a, Fl, F2)).$$

The output is simulated by the procedure `writefile(Symbo1, Pos, NewPos)` defined by the unit clause

$$writefile(X, [X1B], B).$$

The call `?- e(F, [1, [a, +, a, +, a], []])` produces $F = [a, a, +, a, +]$. This example shows that difference lists can be used both to select parts of a list and to construct a list. It is not hard to write a program that automatically performs the translation from a DCG using `write` to a DCG using `writefile` and additional difference lists. In the remainder of the paper we use the procedure `write`, and leave to the interested reader the task of adding difference lists to avoid side effects.

The BNF of full-fledged programming languages can be readily transcribed into DCGs that translate source programs into syntax-trees which can then be either interpreted or used to generate code. We have tested the DCG needed to process the entire Pascal language by translating input programs into syntaxtrees. The following program fragments illustrate this construction for parts of a mini-language. A while statement is defined by the DCG:

```
statement(while(Test, Do)) --+
[while], test(Test), [do], statement(Do).
```

A test may be defined by

```
test(test(Op, E1, E2)) --+ expr(&l),
comP(oP),
expr(E2).
camp(=) --+ [=I.
cow(( )I --+ [( )I.
etc . . .
```

The translation of statements into P-code-like instructions is also easily achieved. For example the statement `while T do S` can be directly translated into the sequence

```

L: code for test T
jif(i.e, jump if false) to Exit
S
jump to L
Exit:

```

If labels are represented by terms of the form `label(L)` and the instructions by `instr(jif, L)` or `instr(jump, L)`, the translation is performed by the DCG:

```

statement([label(L), Test, S, instr(jump, L), label(Exit)]) --+
[while], test(Test, Exit), [do], statement(S).

```

where

```

test([R1, R2, Op, in.str(jif, Exit)], Exit) --+
expr(E1, R1),
comP(Op),
expr(E2, R2).

```

This example illustrates the elegant use of Prolog's logical variables and unification in compiling. Each of the variables `L` and `Exit` occur twice in the generated code. When instantiated, each pair will be bound to the same actual value. This instantiation may occur at a later stage when the final program is assembled and storage is allocated. Even when using special compiler-writing tools such as YACC, the implementation of similar constructs requires lengthier programs since one has to keep track of locations that have to be updated when final addresses become known. Prolog's ability to postpone bindings is therefore of great value in compiling.

The advantage of using logical variables and delayed binding is also apparent in managing symbol tables. Consider the procedure `Zookup`(Identifier, Property, Dictionary) in which `Dictionary` is a list containing the pairs (identifier-property); `lookup`'s behavior is similar to that of the procedure `member(E, L)` which tests if an element `E` is present or not in the list `L`. However, `lookup` adds the pair to the `Dictionary` if it has not been previously added. We have

```

lookup(I, P, [[I, P] 1 T] :- !.
lookup(I, P, [[II, P1] 1 T] :- lookup(I, P, T).

```

If `lookup` is initially called with an uninstantiated variable, the first clause will create the new pair `[I, P]` as well as a new uninstantiated variable `T`. The cut is needed to prevent backtracking once the desired pair is found or is created. Consider now the sequence of calls to `lookup`:

```

lookup(a, X1, D), lookup(b, X2, D), lookup(a, X3, D).

```

The net effect of the above calls is to store the two pairs `[a, X1]` and `[b, X2]` in `D` and to bind `X3` to `X1`. Later on, when `X1` and `X2` are instantiated, `X3` will automatically be bound to the value of `X1`.

A similar approach is used in [31] to implement binary tables. In that paper, table lookup is done in the code-generation phase after constructing the syntax trees (see Section 7). If one wished to perform that operation while parsing, the DCG rule defining a factor could be

```
WU, PI, D) --+ Went(Z (bokup(V, PI, D)J.
```

where `ident(1)` is constructed in a previous scanning pass and the property `P` is determined while processing declarations. In this case lookup should be modified to handle semantic errors such as undeclared identifiers.

9.5 GRAMMAR PROPERTIES

This section makes extensive use of the built-in predicate `setof` which implicitly relies on the nondeterministic capabilities of Prolog. In our view the use of this and similar predicates in determining grammar properties is perfectly justifiable, since, in this context, efficiency plays a secondary role: grammar properties are usually determined only a few of times when generating the parser and, although it is important that the generated parser itself be efficient (and deterministic), longer generation times are usually tolerable.

We start by pointing out that it is easy to test whether a grammar is strictly weak-precedence or LL(1), provided one knows the sets `first(N)`, `follow(N)`, and `last(N)`. The Prolog procedures for performing these tests follow the declarative definitions closely and appear at the end of this section. We first show how Prolog can be used to calculate these sets in the general case of context-free grammars which may contain left-recursive nonterminals and e-rules.

We assume that the rules for a grammar are stored in the database by assertions like

$$\text{rule}(RuleNum, n(A), Rhs)$$

in which `RuleNum` is an integer number identifying a rule, `Rhs` is the list representing the right-hand side of the rule defining the nonterminal `A`. Recall that the elements of `Rhs` are identified by the terms of the form `t(T)` and `n(N)` representing terminals and nonterminals.

For each nonterminal `N` in the grammar, `first(N)` is the set of all (terminal or nonterminal) symbols `V` such that `N a* V . . .`. To calculate the set `first(A)` for a nonterminal `A`, we use the built-in procedure `setof` in conjunction with a procedure `first` which finds a single element of this set. Thus, we make the toplevel call:

$$\text{allfirst}(N, L) :- \text{setof}(X, \text{first}(N, [1, X], L).$$

The procedure `first(Input, &a&, V)` has three parameters:

- (1) an Input list representing a sequence (`Y` of terminals and/or nonterminals,
- (2) a Stuck of rule numbers which keeps track of the already considered rules,
- (3) a terminal or nonterminal element `V` such that (`Y = J* V . . .`.

There are three ways in which a symbol T can be the first element of a sentential form derived from (Y: (1) it can be the first element of (Y, (2) it can be the first element of a sentential form obtained by rewriting the first element in cu (which must be a nonterminal in this case), or (3) it can be the first element of a sentential form obtained by rewriting some of the initial nonterminals of (Y into the empty string t. The following procedure contains a clause for each of these three cases. The middle parameter &a& is used to prevent looping by prohibiting the consideration of previously used rules. The third clause uses the procedure reduces-to-epsilon (defined below) to determine if a sequence of nonterminals rewrites into 6

```
first([Symbol 1 Rest], Stack, Symbol).
first([n(N) 1 Rest], Stack, Symbol) :-
rule(Number, n(N), Rks),
not(member(Number, Stack)),
first(Rks, [Number 1 Stuck], Symbol).
first(List, Stack, Symbol) :-
append(A, B, List),
Af[1,
reduces-to-epsilon(A),
first@, Stuck, Symbol).
```

The predicate reduces-to-epsilon(A) will succeed if A represents a sequence cu of nonterminals which rewrite into the empty string. If a sentential form reduces to epsilon, then it must consist entirely of nonterminals that reduce to epsilon. Moreover, if a nonterminal rewrites to epsilon, then there is a parse tree representing this reduction such that no branch of the parse tree contains more than one occurrence of any nonterminal. The translation of these two statements into Prolog is straightforward. The procedure list-reduces-to-epsilon asserts that a sequence of nonterminals List rewrites into epsilon if each of the nonterminals does, and the procedure nt-reduces-to-epsilon asserts that a nonterminal N reduces to epsilon if it rewrites into a sentential form that reduces to epsilon. The stack parameter is used to guarantee that no branch of the parse tree contains multiple occurrences of any nonterminal.

```
reduces-to-epsilon(List) :-
list-reduces-to-epsilon(List, [ I ]).
list-reduces-to-epsilon([ 1, Stock].
list-reduces-to-epsilon([n(N) 1 Rest], Stack) :-
nt-reduces-to-epsih(n(N), [n(N) 1 Stack]),
list-reduces-to-epsilon(Rest, Stack).
nt-reduces-to-epsih(n(N), Stack) :-
rule(Number, n(N), Rhs),
not(intersect(Rhs, Stuck)),
list-reduces-to-epsilon(Rhs, Stack).
intersect(List1, L&2) :- member(X, L&1), member(X, L&2).
```

In weak-precedence, parsing reductions are called for when $S > IV$, where

- (1) W is the terminal element in the window,
- (2) S is the (terminal or nonterminal) element in the top of the stack,
- (3) $S > W$ if there exists a grammar rule

$$Y - a \ast \ast - x, x, \dots,$$

where $X_1 \dots S$ and $WE \text{ first}(X_z)$.

Shifting occurs when $S \in W$, that is, if there is a rule

$$Y - \ast \ast \ast sx, \dots \text{ where } WE \text{ first}(X\ast).$$

To determine whether a language is of the weak-precedence type and to construct the parsing tables, one needs to find for each nonterminal X the set $\text{last}^+(X)$, consisting of all terminals and nonterminals V such that $X \rightarrow \dots V$. This can be done by finding the sets $\text{first}(X)$ for the grammar that is obtained by reversing the right-hand sides of the rules in the original grammar. It is easy to define a procedure `first-rev` that finds the sets $\text{first}(A)$ for the reversed grammar by modifying the procedure `first`. The procedure to compute $\text{last}^+(A)$ is then concisely expressed as follows:

```
last-ph(n(X), 2) :-  
rule(Number, n(X), Rhs),  
reverse(Rhs, RRhs),  
first-rev(RRhs, [1, Z]).
```

As before, the set $\text{last}^+(A)$ can then be found using the `setof` predicate:

```
aUo.st - ph(n(A), L) :- setof(X, last-plus(n(A), [1, X]), L).
```

The set $\text{follow}(N)$ is also succinctly expressed in Prolog. There are two ways in which a symbol V can be in the set $\text{follow}(N)$: (1) there is a rule $X \rightarrow \ast V / 3$ such that $V \in \text{first}(@)$, or (2) there is a rule $X \rightarrow \alpha N \beta$ such that B rewrites into epsilon, and $V \in \text{follow}(X)$. The Prolog procedure for `follow` consists of two clauses closely paralleling these two cases. The middle parameter `&` is again used to prevent looping by prohibiting the multiple use of rules:

```
follow(n(N), Stack, Terminal) :-  
rule(Number, n(X), Rhs),  
not(member(Number, Stack)),  
wwW& In(N) I B1, Rh),  
first(B, [1, Terminal]).  
follow(n(N), Stack, Terminal) :-  
rule(Number, n(X), Rhs),  
not(member(Number, Stack)),  
~PP=U, [n(N) I B1, Rh),  
reduces-to-epsilon(B),  
follow(n(X), [Number 1 Stack], Terminal).
```

The predicate all-follow below calculates the list of all follow symbols of a nonterminal N:

```
all - follow(N, L) :- setof(X, follow(N, [1, X], L), L).
```

To assess the gains in program size and readability the reader may want to compare the above programs with the English description of first and follow in [1, p. 184] and with a Pascal version in [2]. As to efficiency, these programs could be significantly improved by using the assert procedure to memorize previously computed firsts and follows, thereby avoiding recomputation. (This technique, called memoization, has been considered in [24].)

The predicates first and follow and last-plus can be used to test for the LL(1) and weak-precedence grammar properties and to generate the parsing tables for each of these types of grammars. For example, the clauses of the try_reduce procedure can be computed by the following procedure:

```
generate-reduces(L) :-
setof(try-reduce(X, Y), wp-greater(X, Y), L).
wp-greater(X, Y) :-
rule(RuleNum, n(N), Rhs),
appe~(Awl, [A, B I AnyPI, Rh]),
last-plus(A, X),
first([B], Y).
```

and the try_shift clauses can be generated in a similar manner. Once these clauses have been computed and stored in the database, the grammar can be tested for weak-precedence by the query:

```
not-weak-precedence :- try-reduce(S, W), try-shift(S, W).
not-weak-precedence :- rule(N, X, Rhs), rule(M, Y, Rhs), N # M
```

The first clause tests for reduce-shift conflicts, which could easily be reported to the user for selecting the desired action. This choice enables the processing of ambiguous grammars. The second clause tests if two grammar rules have identical right-hand sides.

The procedures to generate LL(1) tables and to test whether a grammar is LL(1) can also be written concisely. The procedure that generates the tables consists of a call to setof, combined with a procedure to find the firsts and follows of the right-hand side of a rule:

```
generate-ill-table(L) :-
setof(entry(t(W), n(X), Rh), first-of-ruk(t(W), n(X), Rh), L).
first-of-r&( W, N, Rh) :-
ruk(Number, N, Rhs),
first(Rhs, W).
first-of-r&( W, N, Rh) :-
rule(Number, N, Rh),
```

```
reduces-to-epsilon(Rhs),  
follow(N, W).
```

To test whether a language is LL(1) we must show that the table constructed above has no multiple entries. This can be done with a call to the procedure

n&-111, defined as follows:

```
not-ill :- entry(t( W), n(X), Rhs1), entry(t( W), n(X), Rhs2), Rhs1 # Rhs2.
```

The generation of the unit clauses for weak-precedence and LL(1) parsing actually amounts to prototyping a parser generator. Additional discussion on this topic is given in Section 6. The predicates first and last can also be used to determine if a grammar contains a nonterminal that is left-recursive and also right-recursive. This is a commonly used test for attempting to detect ambiguity in context-free grammars.

There is a host of grammar properties and transformations that could be succinctly described in Prolog. A few that we have programmed are elimination of rules, general replacement of left-recursive rules by right-recursive ones, and reduction to Chomsky and standard normal forms. Other properties that seem likely candidates for description in Prolog are an attempt to determine if a grammar is LL(k) or LR(k), and the reduction of an LR(K) grammar to LR(l).

9.6 LEXICAL SCANNERS AND PARSER GENERATION

We first note that the syntax of regular expressions is quite similar to that of arithmetic expressions. The union (\sqcup) replaces the add operator and concatenation (represented by a blank or period) replaces the multiplication operator. The star operation may be represented by surrounding a starred sequence by curly brackets. The translation of a regular expression into its syntax-tree is performed either using DCGs (Section 4) or triggering the semantic actions described in Section 3. For example, the expression $((a \sqcup b).c).d$ is translated into the tree: cone (star (cpnc (union (a, b), c)), d). We now present a recognizer accepting strings defined by a regular expression given by its syntax-tree. The first argument of the procedure ret is the syntax-tree, the other two are difference lists (as described in Section 4).

```
rec(L, [L 1 U], U) :- letter(L).  
rec(stur(X), U, V) :- rec(X, U, W), rec(stur(X), W, V).  
rec(star(X), U, U).  
rec(unzbn(X, Y), U, V) :- rec(X, U, V).  
rec(union(X, Y), U, V) :- rec(Y, U, V).  
rec(conc(X, Y), U, V) :- rec(X, U, W), rec(Y, W, V).
```

The above interpreter for regular expressions is admittedly inefficient, since it relies heavily on backtracking. Nevertheless, it might be suitable for fast prototyping. An efficient version of the recognizer may be obtained in three steps:

- (1) translation of regular expressions into a nondeterministic automaton containing t moves;
- (2) reduction of the automaton in (1) to a deterministic one not containing e moves (in the cases where the empty symbol is in the language, a complete elimination of t moves is not possible);
- (3) minimization of the automaton obtained in (2).

The above steps are those performed by LEX, a scanner-generator package developed at Bell Labs. A Brandeis student, Peter Appel, has prototyped a Prolog version of LEX in less than one month. His program can handle practically all features of LEX, but is admittedly slow compared with the original C-version of that package. When compiled, his program can generate a scanner for a minilanguage similar to that in the appendix of [1] in about four minutes. However, it should be noted that Appel's program is considerably (about five times) shorter than the C-counterpart, and it took a fairly short time to develop. Since the Prolog programs are deterministic, further gains in efficiency could be expected by applying the optimizations suggested by Mellish [21]. In Section 9 we briefly describe an alternate approach to scanner-generation using proposed extensions of Prolog.

In the remainder of this section we sketch two approaches for prototyping parser generators. The first generates recursive descent parsers, whereas the second produces SLR(1) parsers of the type used in YACC [1].

The recognizer of regular expressions presented earlier in this section can be easily modified to recognize context-free languages specified by rules whose righthand sides are themselves regular expressions. For example the rule

$$E --> T(T)$$

can be described by the unit clause

$$\text{ruk}(n(e), \text{conc}(n(t), \text{star}(\text{conc}(t(+), n(t))))).$$

The new clause for ret becomes

$$\text{rec}(n(A), U, V) : -\text{rule}(n(A), R), \text{rec}(R, U, V).$$

It is straightforward to prototype a parser generator by implementing the following steps.

- (a) Determine manually the syntax-trees for a grammar B specifying the syntax of the grammar rules themselves. Each nonterminal N has its corresponding syntax-tree TN asserted in the database by rule(N, 7).
- (b) Use the modified recognizer ret to parse strings of B, that is, a set of grammar rules specifying a context-free grammar G.
- (c) Attach actions to ret so that it produces the syntax-trees for the grammar G being read. This step has been described in Section 3.
- (d) Once the trees for G are generated, ret itself can be reused to parse the strings generated by G.

A detailed description of the above steps appears in [14]. A further advantage of this approach is the possibility it offers to generate efficient recursive descent parsers [7]. One may “compile” assembly language code for a parser by “walking” on the syntax-tree of a grammar G.

Another option for parser generation is to use Prolog for producing the tables for SLR(1) parsers, given a set of grammar rules. An item of a grammar G is a production of G with a dot at some position of the right-hand side. Each item can be computed as a triplet (N, D, L) in which N is the rule number, D the dot position, also an integer, and L is the length of the right-hand side. (One could also have used only N and D and recomputed L for each rule N whenever needed.) States are implemented as lists of triplets. The main procedure generates all new transition states stemming from a given state. Termination occurs when no new states are generated. Ancillary procedures are needed to check if the element preceding a dot in a triplet is a nonterminal, or to test if an item has the dot at the end of a rule. This latter check is readily achieved by testing for (N, L, L). Another auxiliary procedure determines all triplets that should be added to a given state once it is found that that state contains items having a dot preceding a nonterminal.

The predicate `follow` (see Section 5) is called to determine the expected window contents that trigger reductions. These correspond to states containing items ending with a dot. As in YACC, the parser generator can produce tables with multiple entries, allowing the user to select the appropriate entry which renders the parsing deterministic.

A Prolog version of YACC has been prototyped at Brandeis by Cindy Lurie. Her program was developed in a couple of months. In addition to generating a parser, it also produces the code embodying the error-detection and recovery capabilities suggested by Mickunas and Modry [22]; the correction costs being interactively supplied by the user (see Section 9). The performance of the Prolog version of YACC is comparable to that of the LEX counterpart. The previous remarks about the efficiency remain applicable. A word is in order about the generated scanners and parsers. They are C-programs which, when optimized, can approach the efficiency of those generated by LEX and YACC.

9.7 CODE GENERATION

9.7.1 Generating Code from Polish

We start by describing a simple program that generates code for a single register computer having the usual arithmetic operations, as well as the LOAD Vur and STORE VW instructions, where Vur is the location of a variable. The DCG for performing the translation is basically that used to generate the postfixed Polish described in Section 4. The algorithm essentially operates as follows:

- (1) When a variable is recognized it is placed on a stack.
- (2) When an operation is recognized its two operands are on the top of the stack.

If these are variables the following instructions are generated:

LOAD1st operand

Operation2nd operand

Step (2) is continued by replacing the top elements of the stack with the mark ccc to indicate that, at execution time, the result will be in the accumulator. To take into account this mark we introduce the revised versions of (1) and (2), which handle the cases where one of the operands is an occ mark.

(la) Before pushing a variable onto the stack it is necessary to check if the mark ccc occupies the position just below the top of the stack. This indicates the need of a temporary storage, since the accumulator was already utilized in a previous operation and it contains a result that should not be destroyed. Thus the mark act is replaced by T_i , the i th element of a pool of temporary locations, and the following instruction is generated: $ST0\ Tie\ It$ is then possible to push the recognized variable onto the stack.

(lb) If the penultimate element in the stack is not ccc, the variable is simply pushed onto the stack.

As for operators, two cases need to be considered: one for commutative operations, the other for noncommutative ones. Let S_1 be the top of the stack and S_2 the element just below it.

(2a) If neither S_1 , nor S_2 is an act, then code is generated as in step (2) above.

(2b, c) For the commutative operations (addition and multiplication) it suffices to generate Operation S_1 if S_2 is an ccc, or to generate Operation S_2 if S_1 is an occ.

(2d) Noncommutative operations (subtraction and division) will check if S_1 is an act, in which case the instruction $ST0\ Ti$ has to be generated and the stack updated with T_i instead of ccc, as is done in (la). The generation proceeds as indicated in (2). The case where S_2 is an ccc is processed as in the case of commutative operations (2b).

The above description can be easily summarized in Prolog. For presentation purposes we assume that the arithmetic expression has been parsed into postfix Polish notation. We also assume that variables are represented by terms of the form u (Name) and operators by terms $op(Op)$. In an actual implementation the semantic actions described below would be triggered directly from the DCG rules.

The procedure `gen-code(Polish, Stack, Temps)` traverses the list Polish and outputs the code as soon as it is generated. We remind the reader that a program that produces output using `writes` can easily be modified so that it stores the output in a list and thereby avoids relying on side effects to generate results (see Section 4). The `gen-code` procedure is initiated with a call to the procedure `execute`, defined by

$$execute(L) :- gen - code & [1, [O]).$$

where L is the input in postfix. The operators and operands in the list L trigger calls to the corresponding operator and operand clauses, which modify the Stack as described above, and may either remove or return a location from the list $Temps$ of available temporary locations:

```

gen-code([op(Op) 1 Rest], Stack, Temps) :-
operator(Op, Stack, NewStack, Temps, NewTemps),
gen-code(Rest, NewStack, NewTemps).

gen-code([u(X) 1 Rest], Stack, Temps) :-
operand(X, Stack, NewStack, Temps, NewTemps),
gen-code(Rest, NewStack, NewTemps).

gen-code([ 1, AnyStack, AnyTemps]).
```

The operator and operand clauses have five parameters:

- (1) the variable (or operator) being examined,
- (2,3) the starting and resulting stack configurations,
- (4,5) the starting and resulting lists of available temporary locations.

The following remarks will help in understanding the semantic actions of the procedures. The program assumes the availability of an unlimited number of temporary locations which are reused whenever possible: a temporary is returned to its stack after emitting an instruction of the type LOAD Ti or Op Tie. The list of available temporary locations is initialized to contain only the location T_{..}. Whenever a new temporary is needed, it is taken from this list, and if the list contains only one element a new temporary is generated (see the second clause of get-temp below). The term t(X) is used to represent a temporary location.

```
% Case (1a). '
operand(X, [A, act 1 Stack], [u(X), A, t(I) 1 Stack], Temps, NewTemps) :-
get-temp(t(I), Temps, NewTemps),
write(st0, t(I)).
% Case (1b).
operand(X, Stack, [u(X) 1 Stack], Temps, Temps).
```

The first clause of operand guarantees that the accumulator is always the first or second element of the stack, if it occurs at all. The other elements in the Stack are either temporaries or variables:

```
% Case (2b).
operator(Op, [A, act I Stack], [act 1 Stack], Temps, NewTemps) :-
codeop(Op, Instruction, AnyOpType),
gen-instr(Instruction, A, Temps, NewTemps).
% Case (2~).
operator(Op, [act, A I Stack], [act I Stack], Temps, NewTemps) :-
codeop(Op, Instruction, commute),
gen-instr(Instruction, A, Temps, NewTemps).
% Case (2d).
operator(Op, [act, A I Stack], [act I Stack], Temps, NewTemps) :-
codeop(Op, Instruction, rumcommute),
get-temp(t(Z), Temps, Temps0),
write(st0, t(Z)),
```

```

gen-instr(load, A, Temps0, Temps1),
gen-instr(Instruction, t(I), Temps1, NewTemps).
% Case (2a).
operator(Op, [A, B I Stack], [act I Stack], Temps, NewTemps) :- ,
A#acc,B#acc,
codeop(Op, Instruction, OpType),
gen-instr(load, B, Temps, Temps1),
gen-instr(Instruction, A, Temps1, NewTemps).

```

Notice that at most one of the clauses for operator can succeed, since there can be at most one act in the stack. Thus the ordering of the clauses is unimportant, and there is no need for cuts.

The remainder of the program consists of a few auxiliary procedures. The procedure get-temp simulates the pop operation for a stack containing the currently available temporary locations. Temporary locations are returned to the stack by the first clause of gen-instr.

```

codeop(+, add, commute).
codeop (-, sub, noncommute).
codeop(*, mult, commute).
codeop(/, diu, noncommute).
gen-temp(tU), V, J I RI, [J I fW.
getj%y?y), 14, [J1] :-
genhstr(Znstruction, t(Z), Temps, [I 1 Temps]) :-
write(Znstruction, t(Z)).
gen-instr(Znstruction, v(A), Temps, Temps) :- write(Znstruction, A).

```

The code generated for the expression $A * (A * B + C - C * D)$ is

```

LOAD A
MULT B
ADD C
STO To
LOAD C
MULT D
STO T1
LOAD T,,,
SUB T1
MULT A

```

An alternative approach to the method presented here is to generate new Prolog variables to represent the temporaries as they are needed and to ensure, in a subsequent pass, that their usage is optimized.

9.7.2 Generating Code from Trees

A more general approach to code generation is based on “walks” in the syntaxtree of a program. We start by describing Warren’s approach [31] for generating code for a fictitious machine. This computer performs arithmetic operations using a single accumulator. The corresponding instructions are ADD, MULT, SUB, and DIV. Operations of the type ADDI, MULTI, and so on, are also available, and consider the value immediately following them as the second operand in the computation. LOAD and ST0 commands are of course present, as well as the unconditional transfer (JUMP) or conditional ones such as J xx, where xx is EQ, NE, GT, and so on. The input/output commands are simply READ and WRITE. The generator consists of the clause encode-statement which identifies the node of the syntax-tree and constructs the corresponding code. The generated code is a list of instructions and labels, (possibly containing embedded sublists), for instance,

```
[. . . label(LI), [instr(LOAD, X), instr(ADDI, 3)], . . . -1]
```

In Warren’s paper the arguments of instructions are stored in a dictionary, but remain unbound to actual memory addresses until the very final phase of the compiler. At that time an assembler determines the addresses of labels, and an allocator binds the addresses of the variables and reserves the number of memory locations needed to run the compiled program. We now present some fragments of Prolog programs that perform the generation. An assignment of an expression Expr to a variable X is translated into the list whose head is the generated code for the expression followed by the instruction ST0 X. The procedure encodestatement has three arguments: the syntax-tree, the dictionary Diet, and the resulting code. We have

```
encode-statement(assign(name(X), Expr),
Diet,
[Exprcode, instr(sto, Addr)]) :-
lookup(X, Addr, Diet),
encode-expr(Expr, Diet, Exprcode).
```

The procedure lookup stores the new variable X if it is not yet entered in Diet and retrieves the unbound variable representing its address (see Section 4).

The procedure encode-expr can handle two shapes of arithmetic expression syntax-trees. In the first the right operand is a leaf (i.e., a variable or a constant). In the second the right operand is a subtree. The syntax-tree for arithmetic expressions has internal nodes labeled by the operator Op. The more complex case where the right operand is a subtree is presented below. Its action is to translate expr(Op, Expr1, Expr2) (in which Expr2 is of the form expr(Op, Any1, Any2)) into the sequence containing

- (1) the code for Expr2,
- (2) the instruction ST0 temp,
- (3) the code for Expr1, and finally,
- (4) the code for the instruction specified by Op.

An added argument N is needed to specify the pool of temporary locations. Its initial value is zero. In Prolog we have

```
encode-subexpr(expr(Op, Expr1, Expr2), N, Diet,
[Exprkode, instr(sto, Addr), Exprlcode, instr(Opcode, Addr)]) :-  
complex (Expr2),  
lookup(N, Addr, Diet),  
encode-subexpr(Expr2, N, Diet, ExprZcode),  
N is N+1,  
encode-szbexpr(Expr1, N1, Diet, Exprlcode),  
memoryop (Op, Opcode).  
complex(expr(Op, AnyL, AnyP)).  
memoryop(+, add).  
memoryop(*, m&t).
```

The code generated for the previous expression $A^* (A^* B + C - C^* D)$ now becomes

```
LOAD C  
MULT D  
STO TO  
LOAD  
MULT ii  
ADD C  
SUB To  
STO T,,  
LOAD A  
MULT T,,
```

Note that since a right subtree is evaluated before a left one, the code for $C^* D$ is the first to be generated.

The use of labels is illustrated by the generation of code for while statements. The translation consists of transforming the syntax-tree while (Test, Do) into the code

```
hbel(L1): (encode Test)  
(encode Do )  
jump L1  
lubel(L2):
```

Note that a new argument (L2) is needed in the procedure that encodes tests to generate the jump to the exit label. The Prolog program to achieve the translation parallels the above description.

```
encode-statement (while (Test, Do), Diet,  
[hbel(L1), Testcode, Docode, in(& jump, L1), bbel(L2)]) :-  
encode-test (Test, Diet, L2, Testcode),  
encode-statement(Do, Diet, Docode).
```

9.7.3 A Machine-Independent Algorithm for Code Generation

An alternate approach to code generation is that proposed by Glanville and Graham [15, 16]. It is assumed that by syntax-directed translation a source program is translated into its prefix Polish counterpart. A second syntax-directed translation of the prefix code then produces actual machine code. The interesting feature of this approach is that the grammar used to recognize the prefix takes into consideration the description of the machine for which code is generated. Consider a register machine whose operations are of the type

$$LOADM, R$$

$$ADDR1, R2 \text{ or } ADDM, R$$

$$ST0R, M$$

$$ADDIC, R$$

where M is a memory address, C is a constant, and R is a register, and the first argument is the source, the second the destination. To simplify the presentation, we assume an unlimited pool of registers. The problem of dealing with a limited number of registers is discussed in the next section.

The grammar rule

$$R + opRvar1var$$

describes a prefix string in which the last operand is always a variable, (e.g., as in $+ a b c$). The code to be generated in this case can be triggered by semantic actions corresponding to the rules

$$R + var$$

Action: Load variable into register r

$$R + opRvar$$

Action: 1. recognize (recursively) the left operand R assuming that it will use register r

2. generate the code: op var r

Similar grammar rules are applicable for generating code when the last operand is a constant. The more general case corresponds to the grammar rule

$$R + opRR1var1const$$

In this case a new register is needed before recursing to the second R. Also, a register becomes available after recognizing the second R. A natural way of implementing the Glanville-Graham approach is through the use of DCGs. The following simplified grammar rules express assignments:

$$A+ := varR$$

$R + opRvar \ opRconst$ $R - +opRR$ $R + varIconst$

Note that this is an ambiguous grammar, and therefore the use of cuts at the end of each clause is recommended to avoid generating multiple solutions. The recursive descent compiler generated from the DCGs opts, whenever possible, to the first rule defining R, instead of the more general second rule.

The procedures listed below specify the syntax-directed translation of prefix Polish into assembly language according to the above grammar rules. The procedure reg corresponds to the nonterminal R and has three parameters:

- (1) generated assembly language sequence,
- (2) register containing the final result,
- (3) dictionary for storing variables.

Although the presented program assumes an unlimited number of registers, it is fairly straightforward to modify it to consider a finite number only. This can be done by adding extra parameters to the procedure reg.

The first two clauses of reg treat the special cases where the second parameter is a variable or a constant:

```
% Rule:R+OpRvar.
reg([S1, imtr(Op, Addr, R1)], R1, D) --+
arithop(Op, Optwe),
regC% RL D),
[uar(Var)l,
(lookup( Var, D, Addr), !).

% Rule: R --, Op R const.
reg([S1, instr(Constop, C, R1)], R1, D) --+
arithop(Op, Optwe),
redsl, RI, D),
bdC)l,
{constop(op, Con-stop), !}.
```

where arithop and constop are defined as

$$\begin{aligned} \text{arithop}(\text{sub}, \text{noncommute}) &\rightarrow [-I. \\ \text{arithop}(\text{diu}, \text{noncommute}) &\rightarrow [/I. \\ &\quad \text{constop}(\text{sub}, \text{subi}). \\ &\quad \text{constop}(\text{add}, \text{addi}). \end{aligned}$$

$$\begin{aligned} \text{arithopbdd, commute}) &\rightarrow [- + [+I. \\ \text{arithp}(\text{nult}, \text{commute}) &\rightarrow [- + [*I. \\ &\quad \text{constop}(\text{diu}, \text{diui}). \\ \text{constop}(\text{mult}, \text{multi}) &\rightarrow [- + [*I. \end{aligned}$$

It is possible to perform some optimization in the case of commutative operations. For that purpose two additional DCG clauses are included to process the rules:

$$R + opvarRandR + opconstR$$

The DCG clause for the first of these rules is given below.

```
% Rule: R -+ op uar R(op is commutative).
reg([S1, instr(Op, Addr, R1)], R1, D) --+
arithop(Op, commute),
[udVar]1,
reg(S1, RI, D),
(lookup( Var, Addr, D), !).
```

The more complex DCG given below corresponds to the rule $R + op R R$.

```
% Rule: R-*opRR.
reg([S1, S2, instr(Op, R2, R1)], R1, D) --+
arithop(Op),
redS1, RI, D),
(R2 is R1 + 1),
reg(S2, R2, D), (!).
```

Two recursive calls are made to reg to determine the subsequences S1 and S2 representing the code for calculating the two operands. The simple DCG clauses for the rules $R + uar$ and $R + const$ generate the necessary instructions that load a register with a variable or with a constant.

```
% Rule: R + var.
reg(instr(load, Addr, R1), R1, D) --+
[MVar]1,
{lookup( Var, D, Addr), !}.
% Rule: R + const.
reg(instr(hzdc, C, R1), R1, D) --+
[co=dC]1, 1%
```

Finally, we present the DCG clause for generating an assignment expressed in prefix by the rule

$$A+ := varR.$$

```
% Code generator for assignments.
instruction([S1, instr(store, 1, Addr)], D) --+
[assign, uar (Vur)],
red% 0, D),
(lookup( Var, Addr, D)].
```

Notice that the chosen grammar relies extensively on backtracking for recognizing the appropriate rule. For example, consider the two rules

$$R -- + OpRconst$$

$$R -- + OpRvar$$

and the input string (+ + 5 c d). Although the first rule will not apply, it will nonetheless be tried, and the code for the expression (+ 5 c) will be generated before backtracking. The same code will then have to be regenerated when the second rule is applied. This can be avoided by considering the following transformed equivalent grammar:

$$R -- + OpRR2$$

$$R2 -- + Var1Const$$

This transformation can be easily generalized to the case at hand, and the resulting parser will not rely on backtracking so there will be no need to insert cuts into the program.

An example of the code generated by this technique for the expression ,*(A* B + C - C*(D - E)) is

```
LOAD B, R0
MULT A, R0
ADD C, R0
LOAD D, R1
SUB E, R1
MULT C, R1
SUB R1, R0
MULT A, R0
```

9.7.4 Code Generation from a Labelled Tree

We conclude this section by presenting the Prolog programs implementing the optimal code generation applicable to labelled trees as described in [1]

The labelling phase consists of a postorder walk on a syntax-tree in which left leaves are labelled with a 1 and right leaves with a 0. Interior nodes are labelled by $\max(\text{left}, \text{right})$ if the left label is different from the right one; otherwise the interior node is labelled with $\text{right} + 1$. The label of the root specifies the total (optimal) number of registers needed to code the syntax-tree without using temporary locations. In Prolog the labelling is accomplished by the clause `Zubel`, having four parameters: (1) the original syntax-tree, (2) a mark denoting a left or right branch, (3) the generated labelled tree, and (4) the node label itself.

```
label(uar(X), left, uar(X, 1), 1).
label(uar(X), right, uar(X, 0), 0).
label(expr(Op, Left, Right), 2, expr(Op, E1, E2, Label), N) :-
```

```

hbel(Left, left, El, L.ubell),
lubel(Right, right, E2, LabelZ),
nmx(Label1, LabelP, Label).
mux(N, N, N1) :- N1 is N + 1.
m&N, N1, N) :- N > N1.
max(N1, N, N) :- N=c N1.

```

The actual code generation algorithm is practically the same as that presented on p. 544 of [l]. The parameters of gencode are (1) the labelled syntax-tree, (2) the register stack, (3) the maximum number of registers, and (4) the next available temporary location. The procedure will output code for the expression in such a way that the result of the expression is stored in the register at the top of the register stack. (We remind the reader that any program that uses write to produce its results can easily be modified to store the results in a list, as described in Section 4).

The first two clauses consider the simplest cases dealing with leaves. The third clause is applicable only when the right subexpression is a tree. It finds the labels of the two subexpressions and calls gencodel which generates code using the minimal number of registers and temporaries. We purposely avoided the use of cuts by ensuring that each of the clauses deal with mutually exclusive cases, and so no backtracking is possible.

% case 0: left expression is a leaf.

```
gencode(var(X, 1), [Reg] RestR, Max, Temp) :- print(moue, X, Reg).
```

% case 1: right expression is a leaf.

```
gencode(expr(Op, X, uar(Y, 0), Label), [Reg] RestR, Max, Temp) :-
    gencode(X, [Reg] RestR, Max, Temp),
    write(Op, Y, Reg).
```

% cases 2a, 2b, and 2c: left and right expressions are trees.

```
gencode(expr(Op, L, R, Any-Label), Regs, Max, Temp) :-
    labelvalue(L, NL),
    labelvalue(R, NR),
    NR > 0
    gencodel(expr(Op, L, R, Any-Label), Regs, Max, NL, NR, Temp).
```

% case 2a: Left expression can be computed without temporaries.

```
gencodel(expr(Op, L, R, Any-Label), [Reg1, Reg2] RestR, Max, N1, N2) :-
    N1 < N2, N1 < Max,
    gencode(R, [RegZ, Reg1] RestR, Max, Temp),
    gencode(L, [Reg1] RestR, Max, Temp),
    write(Op, Reg2, Reg1).
```

% case 2b: Right expression can be computed without temporaries.

```
gencodel(expr(Op, L, R, Any-Label), [Reg1, Reg2] RestR, Max, N1, N2) :-
    N2 =< Max,
    gencode(L, [Reg1, Reg2] RestR, Max, Temp),
    gencode(R, [Reg1] RestR, Max, Temp),
    write(Op, Reg2, Reg1).
```

% case 2~: temporaries are required.

```
gencodel(expr(Op, L, R, Any-Label), [Reg] RestR, Max, N1, N2, Temp) :-
    N1 = Max, N2 > Max,
    gencode(R, [Reg] RestR, Max, Temp),
```

```

write(move, Reg, t(Temp)),
NextTemp is Temp + 1,
gencode(L, [Reg] RestR], Max, NextTemp),
write(Op, t(Temp), Reg).

```

% miscellaneous.

```

labelvalue(expr(Any1, Any2, Any3, N), N).
labelvalue(var(Any1, N), N).

```

Assuming that two registers are available, the code generated for the expression $((A - B) / (C - D)) \cdot l((E - J) \cdot MG - f_0)$ is

```

MOVE E, R0
SUB F, R0
MOVE G, R,
SUB H, RI
DIV RI, R0
MOVE R,,, T,,,
MOVE A, R,,,
SUB B, R,,,
MOVE C, R,
SUB D, R,
DIV RI, R0
DIV To, R0

```

We conclude this section by pointing out that Cattell's method of code generation is a prime candidate for prototyping using Prolog. In his dissertation, Cattell [5] proposes a method for formalizing and automatically deriving code generators from machine descriptions. The method consists of constructing a syntax-tree-like description for each instruction in the machine's repertoire. A special tree-matching program then generates code sequences by combining the available instruction syntax-trees so that they match the syntax-tree representing a source program. Cattell's approach combines AI techniques with those in current use in compiler construction.

Although the code generators described herein are specialized to the case of Algol-like programs, Prolog has already proved its usefulness in writing Prolog compilers [3]. At Brandeis we have developed a Prolog compiler that compiles Prolog programs into equivalent C programs [8]. A remarkable feature of these compilers is their conciseness and the ease with which they can be changed to generate code in various target languages.

9.8 OPTIMIZATIONS

9.8.1 Compile-Time Evaluation

Compile-time evaluation of numerical expressions and algebraic simplification are easily performed by transforming the syntax-trees of arithmetic expressions into equivalent

trees containing fewer nodes. Both of the procedures evaluate and simplify have as arguments the initial and final trees. They also have a similar structure: recursive calls are made to process the left and right branches until the leaves are reached. Then the auxiliary procedure simp is called to perform the actual simplifications. This allows successive simplifications to be performed.

```
% leaves are left unchanged
evaluate(const(X), const(X)).
evahte(var(X), var(X)).
% internal nodes are optimized after each of its subtrees
% has been optimized
evaluate(expr(Op, Left, Right), Optexp) :-
    evaluate(left, Optleft),
    evaluate(Right, Optright),
    simp(expr(Op, Optleft, Optright), Optexp).
simp(expr(Op, const(X), const(Y)), const(Z)) :-
    Temp =.. [Op, X, Y], Z is Temp.
```

(In the Edinburgh syntax [6], the operation $\text{Temp} = \dots [\text{Op}, \text{X}, \text{Y}]$ binds Temp to the term $\text{Op}(\text{X}, \text{Y})$). Note that, unfortunately, this procedure is unable to simplify expressions such as $a + 3 + 2$ into $a + 5$. This may be achieved by writing a simple procedure that transforms left-associative expressions into equivalent right-associative expressions. The procedure that performs algebraic simplifications is

```
simplify(expr(Op, X, Y), U) :-
    simplify(X, Left),
    simplify(Y, Right),
    simp(expr(Op, Left, Right), U).
simplify(X, X).
```

As before, the auxiliary procedure simp performs the actual simplifications.

```
simp(expr(Op, X, const(0)), X) :- addop(Op).
simp(expr(Op, X, const(1)), X) :- multop(Op).
simp(expr(*, X, const(0)), const(0)).
simp(expr(*, const(0), X), const(0)).
simp(X, X).
addop(+). addop(-). multop(*) . multop(/).
```

9.8.2 Peephole Optimization

Table I summarizes some of the typical peephole optimizations that can be performed after code generation. The table also indicates the source program segments, resulting in code that can be optimized in this manner.

We first note that if Warren's approach (Section 7.2) is used for code generation, an additional pass is needed to "flatten" the list that makes up the generated code. This list contains sublists resulting from the order in which the clauses encode statements

are activated. Assuming that the code consists of a list of elements separated by (right-associative) semicolons, it is a simple matter to express in Prolog the optimizations in Table I:

```
% if pattern is found perform the optimization.
peep([instr(sto, X), instr(load, X) 1 L], [instr(sto, X) 1 M]) :- peep
peep([instr(subi, 0) I L], M) :- peep@, M).
peep([lubel(A), instr(jump, A) 1 L], M) :- peep([instr(jump, A), L],
% keep trying with tail of list.
ped[X I L1, IX I M1] :- peep& MI.
pw([ I, 1 I]).
```

Table I. Peephole Optimization

Source code	Compiled code	Optimization code
a := ...	STO a	STO a
b := a ...	LOAD a	
if a<0 then ...	SUBI 0	ε
while ... do	JUMP a	JUMP b
if ... then ... else;
a: JUMP b	a: JUMP b	

Note that the above program can handle the nondeterministic situations arising when the code to be inspected using the first parameter renders more than one peep clause applicable. The resulting nondeterministic searches could result in longer processing times. It is up to the designer to decide whether this overhead brings significant gains in the execution of the optimized code. A careful ordering of the clauses and a judicious introduction of cuts can reduce some of the overhead. It is also possible to control the amount of backtracking by introducing and tallying costs which, when exceeded, trigger the choice of alternate paths (see Section 9.9).

Finally, it should be mentioned that David Hildum and the first author were able to implement, using Prolog, all of the (over one hundred) peephole optimizations applicable to a P-code-like intermediate language [27]. This was achieved by prototyping a language for specifying the transformations. An interesting aspect of this implementation is that DCG rules are used to generate another set of DCG rules needed to match and replace patterns.

9.9 USING PROPOSED EXTENSIONS

Several extensions have been proposed to enhance the capabilities of Prolog. The reader is referred to [S] for a brief description of some of these extensions. Two of them are of special interest in compiler construction and are dealt with in this section: the use of the built-in predicate `freeze` and unification involving infinite trees. These features

are available in Prolog II [lo], in the interpreter developed by Carlsson [4] and in MU-Prolog [23]³.

The predicate `freeze` (also referred to as lazy evaluation, or coroutining) has the form

$$\text{freeze}(\text{Var}, \text{Procedure})$$

Its action is to immediately activate the given Procedure if the variable Var is bound. Otherwise, the Procedure becomes a dormant goal until Var is bound. In that event, Procedure becomes the next goal to be activated. It is straightforward to write a metalevel interpreter that simulates the effect of `freeze` [8]. This interpreter is admittedly inefficient. Nevertheless, when compiled, it is usable for processing small examples.

We illustrate the use of `freeze` in two contexts: (1) coroutining the scanning, parsing, and code generation phases of a compiler, and (2) error detection and recovery.

The coroutining of the phases is particularly useful when parallel processing is available: it allows the intermediate results of one phase to be transmitted to the subsequent phase and therefore speed-up the computation by triggering simultaneous executions whenever possible. (In Warren's compiler [31] the phases are strictly sequential.) Consider the simple procedure

```
readlkt(L) :- read(X), readrest(X, L).  
readrest(stop, [ ]).  
readrest(X, [X|L]) :- readrest(
```

The built-in predicate `read` reads individual atoms, and `readlist` assembles them into a list. The atom `stop` is used as a flag to terminate the reading.

The availability of `freeze` allows one to write a `writelst` procedure which outputs the elements of a list as soon as they are read:

```
writelst([I]).  
writeln([H|T]) :- freeze(H, writeln(H)), freeze(T, writelist(T)).
```

The query is: `?- freeze(L, writelist(L readlist(`

The same ideas can be used to alternately transfer control among the scanner, the parser, and the code generator. The main procedure `compile` is

```
compile :- freeze(Tree, encode-statement(Tree, Diet, Code)),  
        freeze(List, parse(Lkt, Tree)),  
        scan(List).
```

The above states that `purse` can only be activated as soon as (a part of) a List is available. Similarly, `encodestatement` is activated as soon as a (partially) instantiated syntax-tree becomes available. It is of course necessary to "sprinkle" additional freezes within `parse` and `encode-statement`. This is illustrated below by examples. The translated DCG rule for parsing a `while` statement becomes

³ One purpose of presenting them here is to generate interest, so that they will become more generally available.

```

stutement (whik (Test , Do) , [ while 1 D1] , 04) :-
freeze (D1, test (Test , D1 , D2)) ,
freeze (D2, eq (D2, [ do 1 D3])) ,
freeze (D3, statement (Do, D3,04)).

```

The second and third parameters of statement and test are the difference lists for parsing strings derived from the corresponding nonterminals. The procedure eq is simply the unit clause eq(X, X) which unifies its arguments. The effect of freezing on D1, 02, and 03 is to allow test and the recursive call of statement to be activated only when the pertinent information becomes available. Similarly, the code generator for a while node of the syntax-tree (see Section 7.2) becomes

```

encode-statement (while (Test , Do) , Diet , . . .) :-
freeze (Test , encode-test (Test , Diet , L2 , Testcode)) ,
freeze (Do, encode-statement (Do, Diet , Docode)).

```

Figure 1 shows the alternating flow of control among the scanner, parser, and code generator while compiling a small program using the coroutining technique. The reader might have already suspected that the introduction of freezes could be done automatically. We have indeed developed programs that perform this task, based on user-specified mode declarations (input or output) for each parameter of a procedure.

Another usage of freeze is in error detection and recovery. The following example just illustrates the main ideas, which are based on the work of Mickunas and Modry [22]. At the top level the procedure recover has two parameters: (1) the possibly erroneous input string and (2) the corrected string.

```

recover ( Tokens , Tree) :-
freeze (Filtered-tokens , parse (Filtered-tokens , Tree)) ,
correct ( Tokens , Filtered-tokens , 0).

```

Figure 1

The variable Filtered-tokens is initially unbound; purse will call the corresponding procedures that use the difference lists. The third parameter of correct is the initial cost of correction. The approach consists of attempting to insert or to delete tokens in the input string so that an erroneous string becomes parsable. If necessary, different costs for insertion and deletion, applicable to specific terminals, can be specified by the designer. In the simplified version of correct listed below, a unit clause cost-ok(nax) in which mar is a number that controls the amount of backtracking.

% final scan

```

correct ([ 1, [ 1, Cost ]).
70 normal scan
correct ([X 1 R], [X 1 Rl] , Cost) :- correct (R, Rl, Cost).

```

% deletion

```
correct ([X 1 R] , Rl , Cost) :-  
cost-ok(Cost) ,  
Cost1 is cost + 1 ,  
correct(R , Rl , Cost1).
```

% insertion

```
correct(R , [ I 1 Rl] , Cost) :-  
cost-ok(Cost) ,  
Cost1 is cost + 1 ,  
correct(R , Rl , Cost1).
```

Note that the variable I in the insertion clause will be bound by the parser according to the grammar rules.

A more elaborate version of correct could reduce the amount of nondeterminism by making insertions and deletions based on examining the (fragments of the) parse tree constructed prior to encountering an error. This is the approach described in [22].

In addition to the two above uses of freeze, we have explored its application in dataflow analysis. The iterative methods described in [l] can be implemented using a variant of freeze in which the frozen variables simulate the incoming and outgoing flow of information for each block.

The other proposed extension of Prolog that is useful in compiler design deals with the so-called infinite trees. It is Colmerauer's contention that grammars, flowcharts, and programs frequently specify loops or recursion [l], which can be conveniently described using directed graphs. Their use within Prolog requires that the unification operation be extended to handle circular structures instead of trees.

An elegant and novel approach for implementing a scanner generator using infinite trees has been developed by students of the University of Marseilles [12] under the guidance of A. Colmerauer. It consists of using a special type of unification to produce the minimal finite state automaton directly from a given regular expression. Most Prolog interpreters perform unification only on trees. A notable exception is the interpreter developed at Marseilles, which can unify special kinds of graphs called infinite trees [lo]. For example, when the unit clause eq (X, X) is matched with eq (A, stute(a(A))), the resulting unification is expressed by the infinite tree:

Terms representing states have an additional component specifying whether the state is final or not. The procedure to translate a regular expression into the corresponding minimal finite state automaton takes as input the expression given by its syntax-tree and produces as result the infinite tree corresponding to the minimal automaton. The highlights of this translation are given in what follows.

If a node of the syntax-tree is a conc(L.eft, Right), one recursively determines the automata corresponding to the Left and Right branches and “concatenates” the two automata to obtain the result. Concatenation of two automata A1 and A2 is performed by checking whether the starting state of A1 is final or not-find. In the first case the resulting automata is the union of the automata A2 with the concatenation of automata A1' and A2, in which A1' is a modified copy of A1 in which the starting state is considered to be not-find. In the second case the concatenation of the two automata

consists of specifying the proper transitions between the final states of A1 directly to the states that stem from the initial state of A2. The union and star operations are processed similarly.

A dictionary is “carried along” as a parameter to provide the information needed to keep a single copy of each of the generated subautomata needed to construct the desired one. Therefore, before proceeding to generate an automaton corresponding to a subpart of a regular expression, the dictionary is used to check if the translation has already been done. If so, the desired subautomaton is retrieved from the dictionary. Otherwise, the automaton is determined and the corresponding entry is placed in the dictionary. This per se does not guarantee the construction of a minimal automaton. The program that “prints” the desired infinite tree is actually the one responsible for the minimization [26]. Again with the use of a dictionary, the printing program keeps unique copies of each subtree of the given infinite tree and uses them every time identical subtrees are found. This process has been proved to terminate [11], and for the particular problem at hand it yields the desired minimal automaton.

The authors of this program [12] extended its capabilities to handle the difference and intersection of regular expressions. The program hardly exceeds three pages of code; it also uses another feature that is only available in Marseilles’ interpreters: the constraint $\&\&ff(X, Y)$, meaning $X \# Y$ is valid even when X and Y are uninstantiated, therefore allowing the program’s execution to continue in the forward mode. Backtracking is thus postponed until it is found that the ensemble of constraints becomes unsatisfiable.

9.10 FINAL REMARKS

In the previous sections we described in Prolog several algorithms that play an important role in the design and construction of compilers. We hope it has become apparent to the reader that the descriptions using Prolog are substantially more concise than those which appear in current textbooks. For example, Aho and Ullman often use a mixture of English, the language of sets, and control primitives usually found in Pascal-like languages. The reader is urged to compare some of their descriptions to those presented in this paper.

The experience we gained with Prolog has convinced us of its effectiveness as a language for rapid prototyping compilers and for developing ancillary tools. Presently, the highest gains are achieved in the development of tools in which performance is not of prime consideration. This is the case of automatically producing code generators, parsers, and scanners. Even if the generation of these components takes considerable computer time (say a few hours), the combined man-machine effort may be inexpensive when compared to the human resources needed to produce their hand-coded counterpart. Another area in which the language has proved its usefulness is in the writing of compilers for Prolog itself. It is fair to say that most Prolog compilers are written in Prolog. The gains are substantial, especially because they have to process relatively short programs, and compilation can be done incrementally as the procedures are developed.

Yet another advantage of using Prolog programs is their ability to perform computation both in the forward and reverse directions. It should therefore be possible to decompile target code to obtain the corresponding source code. Although this is in principle feasible, the use of "impure" Prolog features such as the cut and the assignment (*is*) render the reverse execution impossible. These problems may be circumvented by using the generalized diff, mentioned in Section 9, and by ensuring that simple assignments such as those incrementing the values of variables become backtrackable.

Among the shortcomings of Prolog, it should be mentioned that the language is still in evolution and that, presently, a suitable environment for developing larger Prolog programs is not yet available. The language also suffers from the nonexistence of a methodology for documentation, the lack of scoping for variables, the ever-increasing number of parameters, and the resulting profusion of identifier names.

Benchmarks of the parsers and code generators described in this paper showed that their interpretation is indeed slower than the compiled equivalent programs written in C or in Pascal. Compiled Prolog programs running on a dedicated workstation exhibit 5- to 10-fold speed-ups compared to their interpreted versions. For example, the compiled version of Warren's minicompiler enabled us to generate code for sample programs containing a few hundred statements in a couple of minutes. Such compilation speeds are still admittedly below those attained by equivalent compilers written in C. However, we feel that there is a great potential for improving considerably the performance of compilers written in Prolog. The justifying arguments are as follows.

The advantages of Prolog basically stem from the use of unification and nondeterminism. The present price paid for the advantages are increasing demands in memory and execution time. Since compilers are usually designed to avoid nondeterministic situations it is important to reduce Prolog's interpreter (or compiled code) overhead for dealing with these situations. Once it is known that a Prolog program is deterministic, several optimizations can be carried out. One of them is to eliminate the need of saving choice points for backtracking purposes. The optimized program can then achieve the efficiency of the corresponding programs written in a functional language (see [30]).

In a recent paper, Mellish [21] provides weak conditions for determining automatically if a set of Prolog procedures is deterministic. His method is based on a dataflow analysis in which properties of programs are determined by iteratively solving a system of equations. The efficiency of the compiled code can also be increased by having the user supply, by a mode declaration, the nature (input or output) of each parameter of a procedure. This allows the compiler not only to discard certain nondeterministic situations, but also to replace costly unifications by the simpler operations of assignment and conditionals.

A possibility that should not be overlooked in the quest to speed-up Prolog programs is the use of parallel processing. In contrast with most other languages, Prolog offers an embarrassment of riches for exploiting parallelism. The experience gained by empirical or theoretical analysis of parallel Prolog compilers may therefore help to shed some light as to which particular approach yields better speed-up gains.

We feel that the initial investment spent in learning Prolog is largely compensated

for by the advantages accrued in having a shorter program-development stage and achieving program descriptions that can easily be tried and tested in a computer. It is also possible that other higher level languages such as SETL could be used with the same purpose. What seems certain is that the availability of these languages will make program description less verbose and more accurate. In addition, they will spur the development of optimization techniques capable of rendering efficient the descriptions that are not directly presented in an efficient form. The history of the development of Fortran and other languages indicates that this is not only a desirable goal but likely an unavoidable one.

ACKNOWLEDGMENTS

The first author's initiation to Prolog developed from the close contacts that he has had with the Groupe d'Intelligence Artificielle (GIA) at the University of Marseilles, Luminy, in France. Alain Colmerauer, Michel van Caneghem, Henri Kanoui, Bob Pasero, and Francis Giannesini were all enthusiastic in sharing with him their knowledge of the language they have developed and refined at GIA. Three graduate students from Marseilles: Sylvie Duchenoy, Robert Kong Win Chang, and Sophie Nabitz helped in testing the programs presented here. In particular, Robert Kong, now at Brandeis, has dedicated countless hours in helping us polish the paper. David Hildum implemented the Glanville-Graham code-generation method described in Section 7. Peter Appel and Cindy Lurie did their honor's projects prototyping Prolog versions of LEX and YACC. We count ourselves lucky to have had the opportunity to interact with the above-mentioned persons.

Finally, we wish to express our gratitude to a referee, David S. Warren, who, following a meticulous reading of the original manuscript, helped identify the major issues of Prolog usage in compiling and urged us to discuss them in the revised paper. The thoughtful and detailed remarks made by this referee provided an added incentive to improve the paper and reaffirmed our respect for the refereeing process.

REFERENCES

1. AHO, A. V., AND ULLMAN, J. D. *Principles of Compiler Design*. Addison-Wesley, Reading, Mass., 1979.
2. BACKHOUSE, R. C. *Syntax of Programming Languages*. Prentice-Hall, Englewood Cliffs, N.J., 1979.
3. CAMPBELL, J. A. (ED.) *Implementations of Prolog*. Wiley, New York, 1984.
4. CARLSSON, M. A microcoded unifier for Lisp machine Prolog. In *IEEE Proceedings 1985 Symposium on Logic Programming* (Boston, July 1985), IEEE, New York, 1985, 162-171.

5. 5. CATTELL, R. G. G. Automatic derivation of code generators from machine descriptions. *ACM Trans. Programm. Long. Syst.* 2,2 (Apr. 1980), 173-190.
6. 6. CLOCKSin, W. F., AND MELLISH, C. S. *Programming in Prolog*. Springer-Verlag, New York, 1981(2nd ed., 1984).
7. 7. COHEN, J., SITVER, R., AND AUTY, D. Evaluating and improving recursive descent parsers. *IEEE Trans. Softw. Eng.* SE-5,5 (Sept. 1979), 472-480.
8. 8. COHEN, J. Describing Prolog by its interpretation and compilation. *Commun. ACM* 28, 12 (Dec. 1985), 1311-1324.
9. 9. COLMERAUER, A. *Les Grammaires de Metamorphose*. Groupe d'Intelligence Artificielle, Univ. of Marseilles-Luminy, 1975. (Appears as *Metamorphosis grammar in Natural Language Communication with Computers*. L. Bale, Ed., Springer-Verlag, New York, 1978, 133-189.)
10. 10. COLMERAUER, A., KANOUI, H., AND VAN CANEGHEM, M. *Prolog II*. Groupe d'Intelligence Artificielle, Univ. of Marseilles-Luminy, 1982.
11. 11. COLMERAUER, A. Prolog and infinite trees. In *Logic Programming*, Clark and Tarnlund (Eds.), Academic Press, New York, 1982, 231-251.
12. 12. COUPET, S., AND DUPLESSIS, F. Prolog programs for transforming regular expressions into the corresponding minimal finite state recognizers. *Memoire de D.E.A.*, GIA, Univ. of Marseilles, June 1984 (in French).
13. 13. EARLEY, J. An efficient context-free parsing algorithm. *Commun. ACM* 13, 2 (Feb. 1970), 94-102.
14. 14. GIANNESINI, F., AND COHEN, J. Parser generation and grammar manipulation using Prolog's infinite trees. *J. Logic Programm.* (Oct. 1984), 253-265.
15. 15. GLANVILLE, R. A machine independent algorithm for code generation and its use in retargetable compilers. Ph.D. dissertation, Univ. of California, Berkeley, 1977.
16. 16. GRAHAM, S. L., HENRY, R. R., AND SHULMAN, R. A. An experiment in table driven generation. In *Proceedings SIGPLAN Symposium on Compiler Construction* 17,6 (June 1982), 32-43.
17. 17. GRIES, D. *Compiler Construction for Digital Computers*. Wiley, New York, 1971.
18. 18. GRIFFITHS, T. V., AND PETRICK, S. R. On the relative efficiencies of context-free grammar recognizers. *Commun. ACM* 8,5 (May 1965), 289-300.

19. 19. ICHBIAH, J. D., AND MORSE, S. P. A technique for generating almost optimal Floyd-Evans productions for precedence grammars. *Commun. ACM* 13,8 (Aug. 1970), 501-508.
20. 20. KOWALSKI, R. Logic for Problem Solving. North-Holland, Amsterdam, 1979.
21. 21. MELLISH, C. S. Some global optimizations for a Prolog compiler. *J. Logic Programm.* 2, 1 (Apr. 1985), 43-66.
22. 22. MICKUNAS, M. D., AND MODRY, J. A. Automatic error recovery for LR parsers. *Commun. ACM* 21,6 (June 1978), 459-465.
23. 23. NAISH, L. Automating control for logic programs. *J. Logic Programm.* 3 (1985), 167-183.
24. 24. PEREIRA, F. C. N., AND WARREN, D. H. D. Parsing as deduction. In *Proceedings of the 21st Annual Meeting of the Association for Computational Linguistics* (Cambridge, Mass., June 1983), Association for Computational Linguistics, 1983, 137-144.
25. 25. PEREIRA, F. C. N., AND WARREN, D. H. D. Definite clause grammars for language analysis. *Artif. Intell.* 13 (1980), 231-278.
26. 26. PIQUE, J. F. Drawing trees and their equations in Prolog. In *Proceedings of the 2nd International Logic Programming Conference* (Uppsala, 1984), Ord and Furm, Uppsala, 1984, 23-33.
27. 27. TANENBAUM, A. S., VAN STAVEREN, H., AND STEVENSON, J. W. Using peephole optimization on intermediate code. *ACM Trans. Programm. Lang. Syst.* 4,1 (Jan. 1982), 21-36.
28. 28. UEHARA, K., OCHITANI, R., AND KAKUSHO, O. A bottom-up parser based on predicate logic. In *IEEE Proceedings 1984, International Symposium on Logic Programming* (Atlantic City, N.J., Feb. 1984), IEEE, New York, 1984, 220-227.
29. 29. WAITE, W. M., AND GOOS, G. Compiler Construction. Springer-Verlag, New York, 1984.
30. 30. WARREN, D. H. D. Applied logic-its use and implementation as a programming tool. Ph.D. dissertation, Univ. of Edinburgh, 1977 (also appeared as Tech. Note 290, SRI International, 1983).
31. 31. WARREN, D. H. D. Logic programming and compiler writing. *Softw. Pratt. Exper.* 10 (Feb. 1980), 97-125.

Глава 10

Using Definite Clause Grammars in SWI-Prolog

¹ © Anne Ogborn <aogborn@uh.edu>

Thanks to Markus Triska. Large sections of this tutorial are taken directly from his tutorial, which is used by permission here.

Introduction

Who This Course Is For

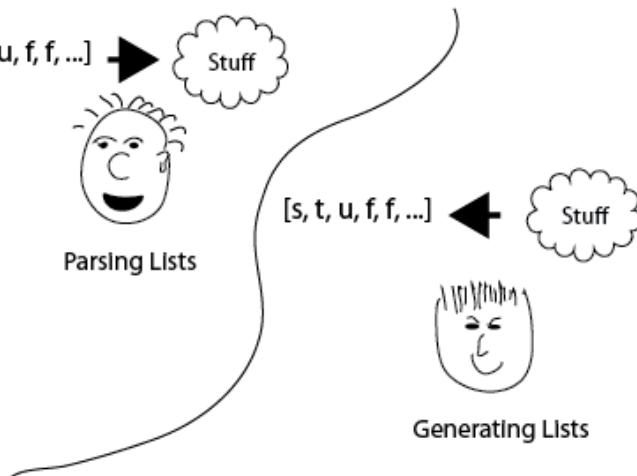
Anyone who:

- knows swi-Prolog reasonably well
- and wants to effectively generate or parse lists.

The second item goes far beyond the usual task of parsing text most programmers associate with DCG's. We'll convert a tree to a list in this tutorial. A DCG could convert a 2D array into a sparse array, or look for patterns in a data stream.

But, we traditionally associate DCG's with parsing text. So we'll give you some tools for parsing text as well.

¹ © <http://www.pathwayslms.com/swipltuts/dcg/>



Getting The Most From This Course

To get the most from this course, you'll need to

- Have a working swi-Prolog install
- Understand basic Prolog be able to use SWI-Prolog's environment
- Read the text
- Try each example program. Experiment!
- A collection of worked exercises and examples is on github
- Do the exercises

Different people will have different backgrounds and learning styles. Whatever works for you works.

Other resources

[Another DCG tutorial](#)

Getting Stuck

If you have questions and **reasonable effort** doesn't answer them, drop me email at aogborn (somechar) uh.edu. Please, if you're taking a beginning Prolog course, ask your instructor. Questions about family trees will be ignored. But if you're working on a real DCG related problem, feel free.

Asking on ##Prolog on freenode.net IRC is also a good way to get answers.

10.1 1 Definite Clause Grammars

A Prolog **definite clause grammar (DCG)** describes a Prolog list.

Operationally, DCGs can be used to parse and generate lists.

10.1.1 1 DCG rules

A DCG is defined by DCG rules. A DCG rule has the form:

head \rightarrow body.

Analogous to normal Prolog rules with:

head :- body.

A rule's head is a (non variable) Prolog term.

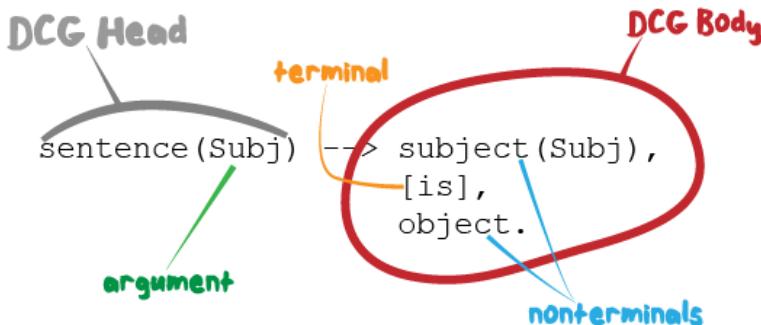
A rule's body is a sequence of terminals and nonterminals, separated by commas.

A terminal is a Prolog list, which stands for the elements it contains.

```
some_termsinals  $\rightarrow$ 
    [this, is, a, teminal],
    [so, is, this],
    "code\_strings\_are\_also\_lists,\_so\_this\_too\_is\_a\_terminal".
```

A nonterminal refers to a DCG rule or other language construct, which stand for the elements they themselves describe.

Declaratively, we can read the comma as "and then" in DCGs.



As an example, let us describe lists that only contain the atom 'a'. We shall use the nonterminal as //0 to refer to such lists:

```
as  $\rightarrow$  [].
as  $\rightarrow$  [a], as.
```

The first rule says: The empty list is such a list. The second rule says: A list containing the atom 'a' and then only atoms 'a' is also such a list.

To execute a grammar rule, we use Prolog's built-in phrase/2 predicate. The first argument is a DCG body. phrase(Body, Ls) is true iff Body describes the list Ls.

The most general query asks for all solutions:

```
?- phrase(as , Ls).  
Ls = [] ;  
Ls = [a] ;  
Ls = [a, a] ;  
Ls = [a, a, a] ;  
Ls = [a, a, a, a] ;  
etc .
```

Examples of more specific queries and the system's answers:

```
?- phrase(as , [a,a,a]).  
true.
```

```
?- phrase(as , [b,c,d]).  
false.
```

```
?- phrase(as , [a,X,a]).  
X = a.
```

Exercises: 1) run 1_1. Add another DCG that creates an alternating series of a's and b's, so your output should look like:

```
Ls = [] ;  
Ls = [a] ;  
Ls = [a, b] ;  
Ls = [a, b, a] ;  
Ls = [a, b, a, b] ;  
etc .
```

2) Try the queries above in 'examples of more specific queries'

- 10.1.2 2 More DCG Syntax
- 10.1.3 3 Capturing Input
- 10.1.4 4 Variables in Body
- 10.2 2 Relating Trees To Lists
- 10.3 3 Left Recursion
- 10.4 4 Right-hand Context Notation
- 10.5 5 Implicitly Passing States Around
- 10.6 6 Parsing From Files
- 10.7 7 Implementation
- 10.8 8 Error Handling
 - 10.8.1 1 Resynching The Parser
 - 10.8.2 2 Printing Line Numbers
- 10.9 9 A Few Practical Hints
 - 10.9.1 1 basics.pl
 - 10.9.2 2 Lexical Issues
 - 10.9.3 Regular Expressions

Conclusion

In conclusion, I'd remind you - if you're working with lists, DCG's can definitely make your life easier. They're not just for parsing any more!

Thanks for taking this tutorial. If I can improve anything please email me at <aogborn@uh.edu>.

If you make something beautiful, drop us a link.

Thanks,

This tutorial is based on a tutorial by [Markus Triska](#), so a special nod to him.

[Ulrich Neumerkel](#), [Richard O'Keefe](#), [Carlo Capelli](#), and [Paulo Moura](#) patiently explained many points on the swipl email list.

[Michael Richter](#) applied his thorough critical eye to the text.

Props to the Sanskrit grammarian [Pāṇini](#), who first formalized grammar.



Annie

Часть III

Дурдом на дереве

Глава 11

The Tree Processing Language Defining the structure and behaviour of a tree

¹ © E. Papegaaij <e.papegaaij@alumnus.utwente.nl>
Supervisors dr. ir. Theo C. Ruys
 ir. Philip K.F. Hözespies
 dr. ir. Arend Rensink
Institute University of Twente
Chair Formal Methods and Tools

Enschede, March 7, 2007

Abstract

Tree structures are commonly used in many applications. One of these is a compiler, in which the tree is called an abstract syntax tree (AST). Different techniques have been developed for building and working with ASTs. However, many of these techniques are limited in their applicability, require major effort to implement or introduce maintenance problems in an evolving application.

This thesis introduces the Tree Processing Language, a language for defining the structure of a tree and adding functionality to this tree. The compiler **TPLc** is used to produce the actual class hierarchy implementing the specified tree. TPL provides a clear separation between the structure of a tree, a **tree definition**, and behaviour of a tree, **logic specifications**. Different aspects of the behaviour of a tree can be provided in separate logic specifications, allowing a clear separation of concerns.

TPLc generates a heterogeneous tree structure with strictly typed children. Functionality in a logic specification is specified using the inheritance pattern. To allow different inheritance trees in different logic specifications, the inheritance pattern is enhanced

with multiple inheritance. For languages that do not support multiple inheritance, the inheritance pattern with composition is developed.

To prove the applicability of TPL, **TPLc** is written in TPL. When compared with an implementation in Java, this implementation provides a better separation of concerns and is easier to maintain.

Preface

Compiler construction has always been one of my favourite fields of software engineering. In the past few years I've written several parsers and compilers. Of these compilers, the compiler for the functional programming language Tina has been the most challenging. I used a hand-written heterogeneous abstract syntax tree as underlying data structure. The most important algorithm applied onto this AST, the transformation of Tina into a core lambda expression language, was written as part of these AST node classes. However, the overwhelming number of AST classes (almost 100) made this approach increasingly difficult to maintain when other algorithms (such as a lambda lifter) were added. At that moment, it became clear that a more structured approach was required. To keep the development of an application, based on a heterogeneous tree, maintainable, different algorithms needed to be separated in different files. The development of tpl is an attempt to provide such an environment.

When I first approached Theo C. Ruys, my premiere supervisor, for an assignment, I had no idea I would be solving this problem, which had bothered me for a long time. At first, ambitious as I was, I proposed to design and implement a completely new parser generator. Luckily, Theo slowed me down a bit and directed me to focus on the real problem: the heterogeneous AST.

For his help in concreting the features of tpl, reading and correcting this thesis and his patience during the endless discussions we had last year, I would like to thank Theo C. Ruys, my premiere supervisor. His guiding helped me structure my thoughts, to be able to write them down. I would also like to thank Philip K.F. Hözzenpies for his help in writing and formatting this thesis. His knowledge of the English language has proven to be far better than mine. Last, but not least, I would like to thank Arend Rensink for having taken the time to examine this thesis.

Emond Papegaaij
Enschede, March 7, 2007

11.1 Introduction

Tree structures have been, and probably will be for a considerable time in the future, a widely used way of organising and working with data. Tree structures are used to represent the structure of an input file², user interface components, the representation

² concrete and abstract syntax trees

of HTML pages³, XML and many more. Due its wide acceptance, extensive research has been spent on working with tree structures.

This thesis is placed in the context of working with tree structures in an object-oriented programming environment. The main focus is on defining the runtime organisation of the tree and applying algorithms on this structure. The origin of the tree — the system responsible for constructing the tree structure — and the actual construction of the tree are discussed, but fall outside the main research area.

In this chapter, an introduction on compiler construction is given, in [11.1.1](#). This section shows how an abstract syntax tree is acquired, and what the typical operations are that need to be performed on an AST. [11.1.2](#) describes the problem statement of this thesis. Finally, the outline of this thesis is given in [11.1.3](#).

11.1.1 Compiler Construction and Abstract Syntax Trees

A multi-pass compiler performs the compilation of a source file in several stages. These stages will be discussed in this section. Compilation starts with reading a source file, and recognising the syntax of the input. Next, an abstract representation of this input is constructed. This is the abstract syntax tree. This AST is used in subsequent phases to perform context checking and code generation. More complex compilers might have more phases, such as optimisers.

Abstract syntax trees are also commonly used in other disciplines, such as communication (eg. a web browser) and source code refactoring in an integrated development environment (IDE) [?]. It is also possible that the abstract syntax tree is not the result of a parser reading an input file, but from speech, or from a graphical programming language. However, the most common usage is a compiler, which reads an input language.

Lexical Analysis and Parsing

In the first stage, the lexical analysis, the compiler reads the input file and produces a stream of tokens. Every token corresponds to a fragment, or construct, found in the input file, such as identifiers, literals, operators and keywords. These tokens are fed to a parser, which discovers (and checks) the structure of the input.

Writing a lexer (or scanner) and parser by hand is tedious, difficult and error prone. Many programs have been developed, which assist the developer in writing the lexer and parser. These tools often take a syntax specification in (E)BNF, and generate a lexer and parser from this specification. Therefore, these tools are commonly called parser generators. Some of these tools are mentioned in ??.

Different strategies exist, on how a parser matches the input language, such as LALR and recursive descent parsing. However, a discussion of these is beyond the scope of this thesis⁴.

³ the document object model

⁴ An explanation of various parsing algorithms, such as LR(k) and LL(k), can be found in [?].

Construction of the AST

In a multi-pass compiler, the task of the parser is to record the structure of the parsed input in an abstract syntax tree. This tree contains all relevant information from the input. What exactly is relevant information, depends on the subsequent phases. Normally, tokens, such as comma's and brackets, are discarded. Also, nesting of parser production rules is removed.

AST construction is exemplified with the grammar presented in fragment 1.1. This grammar matches simple expressions with addition and multiplication. The actual values are represented by numbers and identifiers. Expressions can be nested with brackets.

This grammar matches sentences such as ‘1’, ‘1+1’ and ‘(1+a)*b’. The parse tree of the sentence ‘3+5*(a+b)’ is given in figure 1.1. This figure shows how the complete sentence is matched as an hexpressioni. The hexpressioni consists of a htermi, followed by the literal ‘+’, again followed by a htermi. The left htermi is a simple hatomi, which in turn is a hnumberi. The right htermi consists of two hatomis, separated by a ‘*’. This process is continued until all tokens (the bottom line of the figure) are matched.

The parse tree clearly shows the structure of the parsed text, but this structure is not very practical to work with. If an interpreter for this grammar is needed, a set of four constructs is sufficient: addition, multiplication, numbers and identifiers. The node adds the results of the left and right operands. This node is created when a ‘+’ is matched in hexpressioni. The node multiplies the left operand with the right. It is created when a ‘*’ is matched in htermi. A node is created when a hnumberi is matched, and yields the value of the number. Finally, the node, which is created when an hidentifieri is matched, resolves the value in a symbol table.

Context Checking and Code Generation

11.1.2 Problem Statement

11.1.3 Outline

Часть IV

Язык bI

Глава 12

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

bI-стиль универсальный символьный тип **13.1**, позволяющий хранить как скаляры, так и вложенные элементы; в базовый тип **AST** заложено хранение типа данных **tag**, его значения **value**, и два способа вложенных хранилищ: плоский упорядоченный список **nest** и именованный неупорядоченный со строковыми ключами **pars**.

От базового символьного типа наследуются
скаляры символ, строка, несколько вариантов чисел (целые, плавающие, машинные, комплексные)¹

¹ 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 связностью², Beowulf³ 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-системах⁴

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

² аппаратная разделяемая память через сеть InfiniBand — “Сергей Королев”

³ компьютеры общего назначения (офисные) с передачей сообщений по Gigabit Ethernet

⁴ 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 13

Система динамических типов

13.1 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 деревьев¹. 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 TYPE
struct Sym {
    // _____ тип (класс) и значение элемента данных
    string tag;                                // data type / class
    string val;                                 // symbol value
    // _____ конструкторы (токен используется в лексере)
    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 данных²

```
AST может хранить (и обрабатывать) вложенные элементы
// _____
```

¹ в этом АСТ близок к традиционной аббревиатуре AST: Abstract Syntax Tree

² 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 representation  
virtual Sym* eq(Sym*); // A = B assignment  
virtual Sym* inher(Sym*); // A : B inheritance  
virtual Sym* member(Sym*); // A % B,C named member (class)  
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  
};
```

13.2 Скаляры

- 13.2.1 str: строка
- 13.2.2 int: целое число
- 13.2.3 hex: машинное hex
- 13.2.4 bin: бинарная строка
- 13.2.5 num: число с плавающей точкой

13.3 Композиты

- 13.3.1 list: плоский список
- 13.3.2 cons: cons-пара и списки в *Lisp*-стиле

13.4 Функционалы

- 13.4.1 op: оператор
- 13.4.2 fn: встроенная/скомпилированная функция
- 13.4.3 lambda: лямбда

Глава 14

Программирование в свободном синтаксисе: FSP

14.1 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 &
...

```

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

14.1.1 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 файла¹ для каждого диалекта скриптового языка², и привязать их к расширениям через dot-файлы (g)Vim в вашем домашнем каталоге. Подробно конфигурирование (g)Vim см. 27.

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	

14.1.2 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

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

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

14.1.3 Makefile

Для сборки проекта используем команду **make** или **ming32-make** для Windows/MacOS.

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 исходных файлов ядра транслятора на C_+ :

cpp.cpp реализация системы динамических типов данных, наследованных от символьного типа AST [13.1](#). Библиотека динамических классов языка *bI IV* 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 IV*: 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 16.2**

lex.yy.c код лексического анализатора, генерируемый утилитой **flex 16.1**
`CXXFLAGS += gnu++11` добавляем опцию диалекта C_+ , необходимую для компиляции ядра *bI*

Глава 15

Синтаксический анализ текстовых данных

15.1 Универсальный Makefile

Универсальный Makefile сделан на базе 14.1.3, с добавлением переменной APP указывающей какой пример парсера следует скомпилировать и выполнить.

Для хранения (и возможной обработки) отпарсенных данных используем ядро языка *bi 13* — 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 $< > $@
```

15.2 C_+^+ интерфейс синтаксического анализатора

```
extern int yylex(); // получить код следующего токена, и увлечь  
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 ошибке  
#include "ypp.tab.hpp"
```

15.3 Минимальный парсер

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

¹

Лексер **minimal.lpp** /flex/

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

(.. / bi /) **hpp.hpp** содержит определения интерфейса лексера 15.2, и ядра языка *bI* 13 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.

¹ 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)²

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 [15.3](#)

%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 [13.1](#) и его производными классами.

%token описывает токены, которые может возвращать лексер **??**, причем набор токенов должен быть согласованным между лексером и парсером³

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

REPL выражение, описывающее грамматику, аналогичную простейшему варианту цикла REPL: Read Eval Print Loop⁴. В нашем случае часть вычисления Eval не выполняется⁵, 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**

² точка добавлена, так часто используется в именах файлов

³ определение токенов генерируется в файл `ypp.tab.hpp`

⁴ чтение/вычисление/вывод/повторить

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

`scalar` выражение, представляющее только распознаваемые скаляры:

`SYM` символ,

`STR` строку [или](#)

`NUM` число⁶

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**[15.3](#), распознались как объекты bI , остальные остались символами и попали в лог без изменений.

15.4 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"  
%}
```

⁶ 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) 15.2, 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).

15.5 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]*          {}
\\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 строки⁷

<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'>.⁸

15.6 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)**:

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

⁸ 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:{>
```

15.7 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*[13.1](#). 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 16

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 13.3, и не заставлять пользователя закапываться в клубок *Lisp*овских скобок.

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

Единственной проблемой с точки зрения синтаксиса для начинающего пользователя *bI* может оказаться отказ от скобок при вызове функций, применение оператора явной аппликации \mathfrak{C} , и функциональные наклонности самого *bI*, претендующего на звание универсального **объектного метаязыка** и **языка шаблонов**.

16.1 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();
%%
```

16.2 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* 13.1, 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 NUM¹. Надо отметить, что в принципе можно было бы обойтись единственным SYM, но для дополнительного контроля грамматики полезно выделить несколько токенов: это позволит гарантировать что в определении класса ?? вы сможете использовать в качестве суперкласса и имен полей только символы. По крайне мере до момента, когда в очередном форке *bI* не появится возможность наследовать любые объекты.

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

Часть V

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	bison	парсер синтаксиса
lpp.lpp	flex	лексер
hpp.hpp	<i>C₊⁺</i>	хедеры
cpp.cpp	<i>C₊⁺</i>	ядро интерпретатора
Makefile	make	скрипты сборки проекта
.gitignore	git	маски файлов, не попадающие в git-проект
bat.bat	<i>Windows</i>	helper запуска (g)Vim

.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 V

%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\n-can\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	блокировка вычисления 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> не класс , 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 VI

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 загрузчиком², 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

² 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 17

clock: коридорные электронные
часы = контроллер умного
дурдома

Глава 18

gambox: игровая приставка

Часть VII

GNU Toolchain и C_+^+ для встраиваемых систем

Глава 19

Программирование встраиваемых систем с использованием GNU Toolchain [23]

© Vijay Kumar B.¹ перевод ²

19.1 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

¹ © <http://bravegnu.org/gnu-eprog/>

² © <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 19.18.

19.2 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 программный³ эмулятор нескольких распространенных аппаратных платформ. Вы можете написать программу на ассемблере и C_+ , скомпилировать ее используя **GNU Toolchain** и отладить ее в эмуляторе **Qemu**.

19.2.1 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.

Процессор⁴ 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.

19.2.2 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 требуется⁵. Установим пакет командой:

```
$ sudo apt install qemu
```

19.2.3 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

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

⁴ Точнее SoC: система-на-кристалле

⁵ хотя может быть и очень хочется

1. Годные чуваки из CodeSourcery⁶ 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 [21](#).

19.3 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, цифр⁷, 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:

⁶ подразделение Mentor Graphics

⁷ 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.

19.3.1 Сборка бинарника

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

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

Опция -o указывает выходной файл с *объектным кодом*, имеющий стандартное расширение .o⁹.

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
```

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

⁹ 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**¹⁰ вызовем **линкер ld**:

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

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

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

¹¹ 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 байта¹², 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 кода¹³. Формат ELF применяется для хранения машинного кода, если вы запускаете его в базовой ОС¹⁴, но поскольку мы собираемся запускать нашу программу на bare metal¹⁵, мы должны сконвертировать полученный .elf файл в более простой **бинарный формат**.

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

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

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

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

¹⁴ прежде всего “большой” или встраиваемый *Linux*

¹⁵ голом железе

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

```
$ objcopy -O <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 байтам¹⁷:

```
$ 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**

19.3.2 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 блоками¹⁸ (4096 x 4K):

¹⁷ 4 инструкции по 4 байта каждая

¹⁸ 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.

19.3.3 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
```

19.4 Директивы ассемблера

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

19.4.1 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
eoas:   .align
start:
        ldr r0, =eoas      @ r0 = &eoas
        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.

19.4.2 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**¹⁹.

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.

19.5 Использование ОЗУ (адресного пространства процессора)

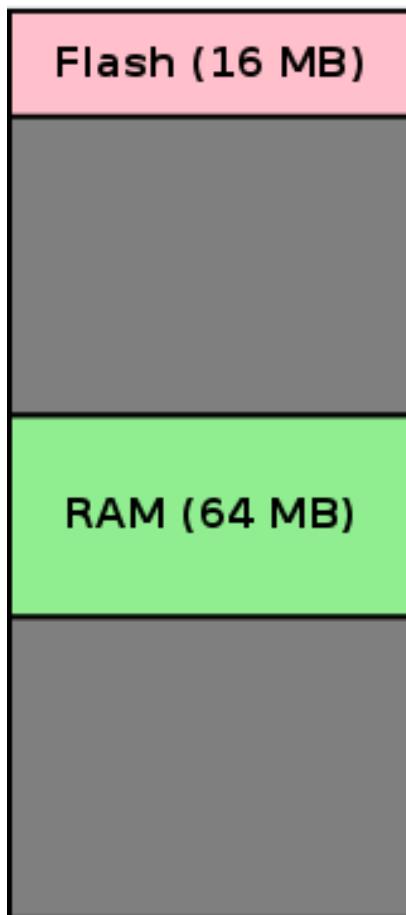
Flash-память описанная ранее, в которой хранится машинный код программы, один из видов EEPROM²⁰. Это вторичный носитель данных, как например жесткий диск, но в любом случае хранить данные и значения переменных во флашне неудобно как с точки зрения возможности перезаписи, так и прежде всего со скоростью чтения флаша, 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:

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

²⁰ 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**²².

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

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

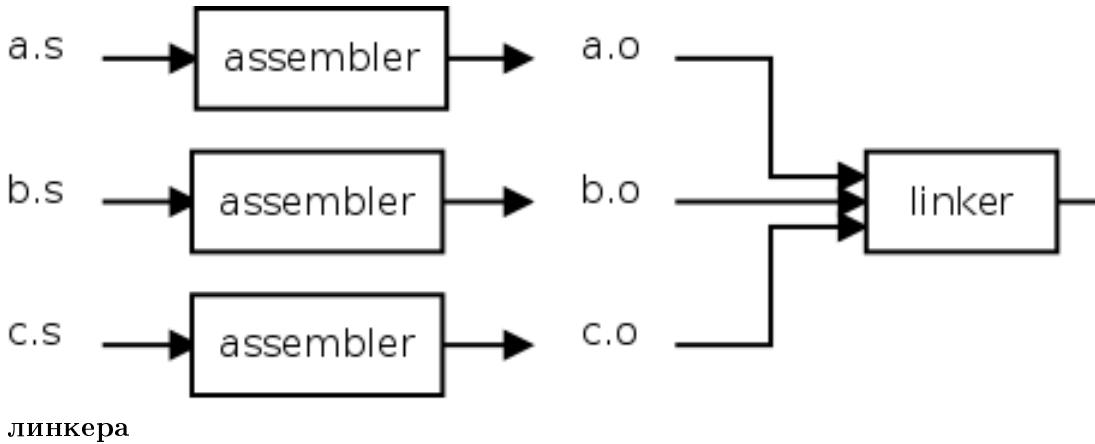
²² или используете малораспространенные клоны ARM-процессоров, типа Миландровского 1986BE9x “чернобыль”

19.6 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.a²³ в один исполняемый файл.

При написании многофайловой программы (еще это называют **инкрементной компоновкой**) каждый файл исходного кода ассемблируется в индивидуальный файл объектного кода. Линкер²⁴ 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.

19.6.1 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

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

²⁴ или компоновщик

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
```

²⁵ 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*²⁶, 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.

19.6.2 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:

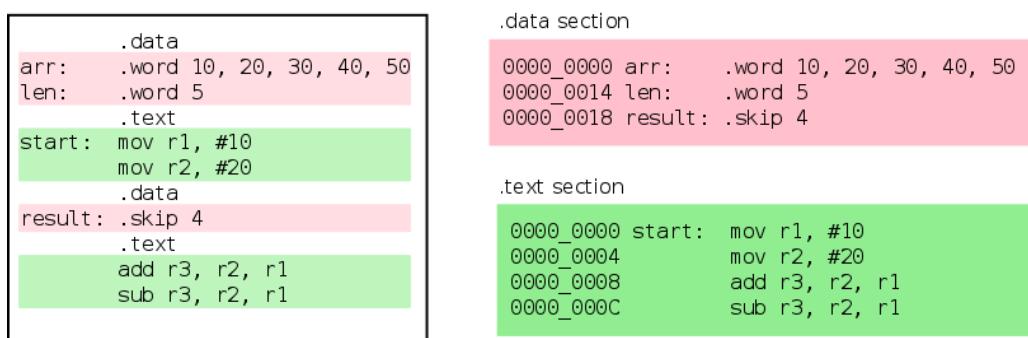
1. Объединение секций
2. Размещение секций

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` ожидаемо используются для переключения между этими секциями.

²⁶ 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

19.7 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.

19.7.1 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.

19.7.2 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 0x0²⁷
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`²⁸ указывает адрес размещения сегмента

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

²⁸ 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`

19.8 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 !**

19.8.1 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).

19.8.2 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 символ²⁹. Так что в результате мы настроили адрес хранения .data на область флеша, помеченную символом **etext**.

²⁹ значением которого является валидный адрес

19.8.3 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.

19.9 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`.

19.10 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

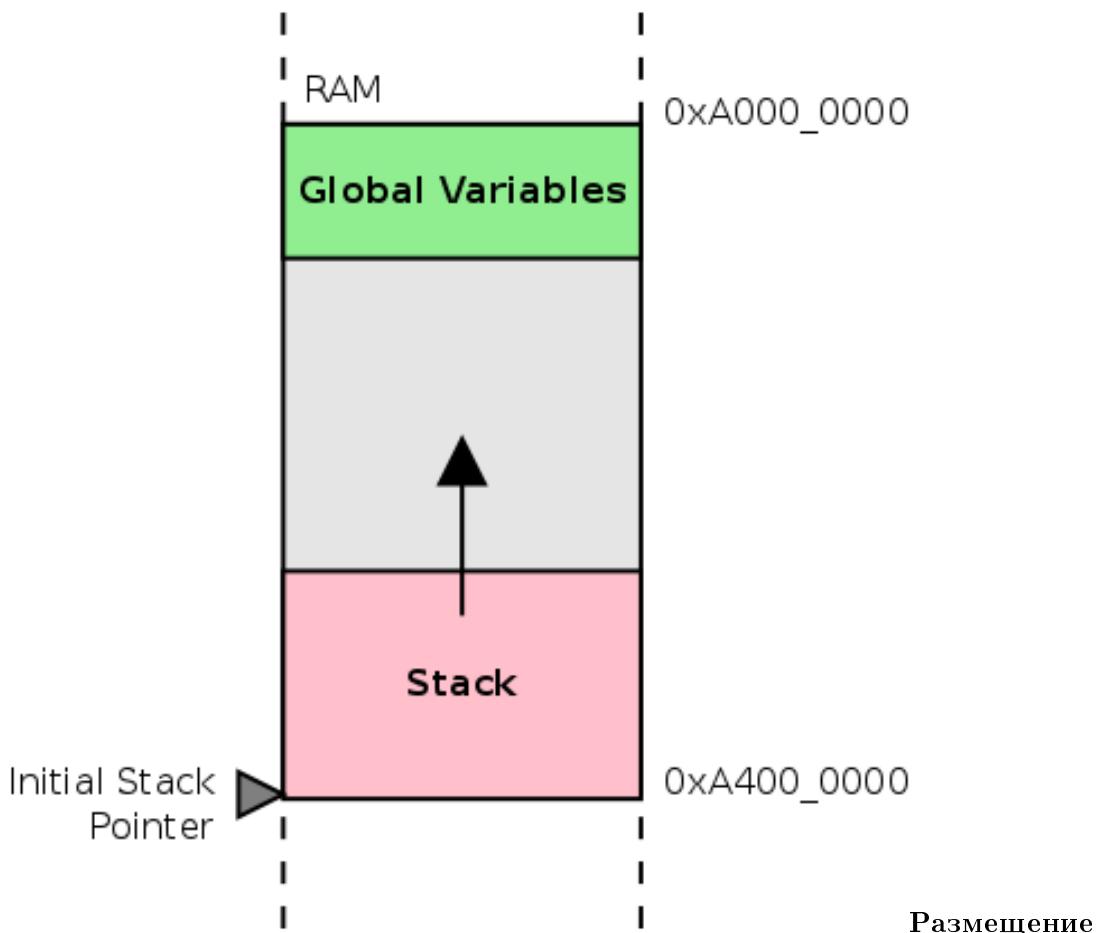
19.10.1 Стек

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.

19.10.2 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 нулем³⁰. 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.

19.10.3 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`.

19.10.4 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).

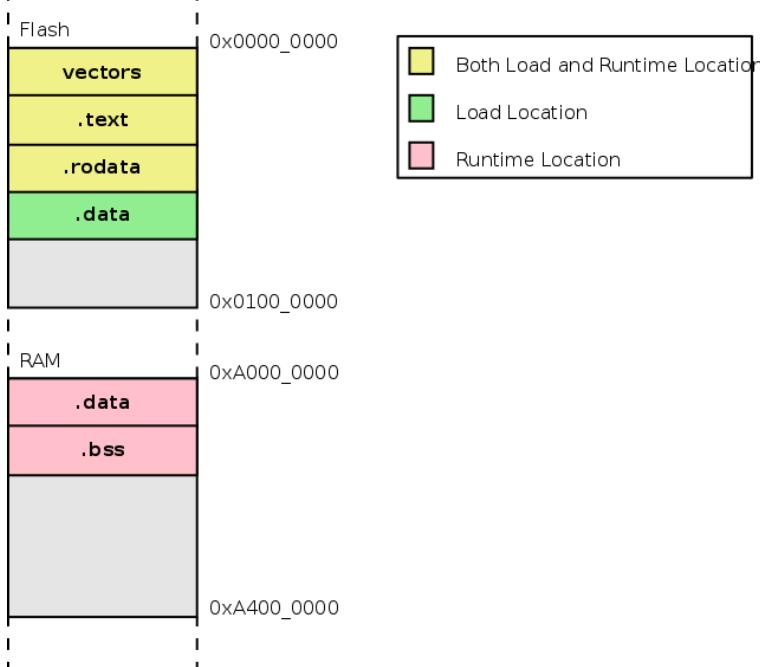
19.10.5 Стартовый код

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:

³⁰ 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
```

19.11 Использование библиотеки Си

FIXME: Эту секцию еще нужно написать.

19.12 Inline-ассемблер

FIXME: Эту секцию еще нужно написать.

19.13 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 ‘проверяться 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.

19.13.1 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 исходников³¹, и вы изменили несколько символов в одном файле. Без **make**³² 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.

19.13.2 Переменные

19.14 13. Contributing

19.15 14. Credits

19.15.1 14.1. People

19.15.2 14.2. Tools

19.16 15. Tutorial Copyright

19.17 A. ARM Programmer's Model

19.18 B. ARM Instruction Set

19.19 C. ARM Stacks

³¹ например тысячи файлов, как у ядра *Linux*

³² используя простой .rc shell-скрипт или .batник

Глава 20

Embedded Systems Programming in C_+^+ [22]

1

Глава 21

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**¹ 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 19.13**, которой можно передать набор переменных конфигурирования. В таблице перечислен набор переменных конфигурирования сборки с указанием их значения по умолчанию² и имя mk-файла, где оно задано:

¹ например для i386 может понадобится сборка кросс-компилятора с **-target=i486-none-elf** IX или **i686-linux-uclibc** вместо типовой компиляции для *Linux* типа **i486-linux-gnu**

² 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 проекта³ 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

³ 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/HW⁴** 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 сразу⁵**. Более простой способ – просто сделать симлинк **ln -s ~/gz project/gz** и не переопределять переменную **GZ** явно.

⁴ по ним могут закачиваться дополнительные файлы исходников, зависящие от платформы — например исходник загрузчика или бинарные файлы (блöбы) драйверов от производителя железки

⁵ 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
```

21.1 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)

21.1.1 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

```

21.1.2 binutils0: ассемблер и линкер

Чтобы побыстрее получить результат, который можно сразу потестировать, соберем сначала кросс-**binutils**, а потом все что относится к Сициальному компилятору⁶.

-target триплет целевой системы, например **i386-pc-elf**

⁶ 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 процессора⁷

-`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

⁷ 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
```

21.1.3 **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
```

21.1.4 newlib: сборка стандартной библиотеки libc

Стандартная библиотека **libc**⁸ 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
```

21.1.5 gcc0: пересборка компилятора Си/ C_+

21.2 Поддерживаемые платформы

21.2.1 i386: ПК и промышленные PC104

arch/i386.mk

```
CFG_ARCH =
```

21.2.2 x86_64: серверные системы

arch/x86_64.mk

21.2.3 AVR: Atmel AVR Mega

arch/avr.mk

21.2.4 arm: процессоры ARM Cortex-Mx

arch/arm.mk

21.2.5 armhf: SoCи Cortex-A, PXA270,..

arch/armhf.mk

⁸ для микроконтроллерных систем — обрезанная версия, **newlib**

21.3 Целевые аппаратные системы

21.3.1 **x86**: типовой компьютер на процессоре i386+

hw/x86.mk

CPU = i386

TARGET = \$(CPU)-pc-elf

Глава 22

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

Глава 23

Оптимизация кода

23.1 PGO оптимизация

¹

Часть VIII

Микроконтроллеры Cortex-Mx

Часть IX

**os86: низкоуровневое
программирование i386**

Если вам по каким-то причинам не подходит одна из типовых распространенных ОС, например требуется сделать систему управления жесткого реального времени², 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 [21](#).

Для упрощения не будем завязываться на особенности конкретного ПК или эмулятора **Qemu**³, 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 X** быть слинкована в формат ELF и включать заголовок multiboot.

² или вы любитель гадить из прикладного ПО в аппаратные порты в обход всех соглашений и средств защиты ОС

³ VMWare, VirtualPC

Часть X

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>

Copyright © 1995,96 Erich Stefan Boleyn <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 24

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.

24.1 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.

24.2 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.

24.3 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.

24.4 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 systems¹, using precalculated *blocklists*², loading from a special *boot partition*³, or even loading from within another operating system⁴. 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.

24.5 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.

24.6 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.

¹ e.g. the BSD/Mach boot loader

² e.g. LILO

³ e.g. OS/2

⁴ 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.

24.7 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 25

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.

25.1 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.

25.1.1 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 25.1.2, the fields ‘header_addr’, ‘load_addr’, ‘load_end_addr’, ‘bss_end_addr’ and ‘entry_addr’ are defined in Header address fields 25.1.1, and the fields ‘mode_type’, ‘width’, ‘height’ and ‘depth’ are defined in Header graphics fields 25.1.4.

25.1.2 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.

25.1.3 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.

25.1.4 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.

25.2 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.

25.3 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 XI

Технологии

Часть XII

Сетевое обучение

Часть XIII

Базовая теоретическая подготовка

Глава 26

Математика

26.1 Высшая математика в упражнениях и задачах [68]

В этом разделе будут размещены решения некоторых задач из [68] 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^AT_EX 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 режиме¹ 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

26.1.1 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:

$$d = |x_2 - x_1| \quad (26.1)$$

¹ например в файлах Makefile 19.13

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 λ , где $\lambda = \frac{AC}{CB}$. Если отрезки AC и CB направлены в одну сторону, то λ приписывают знак “плюс”; если же отрезки AC и CB направлены в противоположные стороны, то λ 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 λ , находится по формуле:

$$\bar{x} = \frac{x_1 + \lambda x_2}{1 + \lambda} \quad (26.2)$$

В частности, при $\lambda = 1$ получается формула для координаты середины отрезка:

$$\bar{x} = \frac{x_1 + x_2}{2} \quad (26.3)$$

Формула 26.2 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

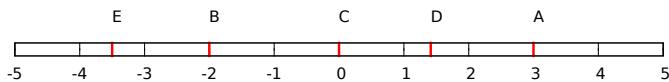


```
number line 3,-2,0,sqrt(2),-3.5 • 3 | • -2 | • 0 | • √2 | • -3.5
```

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
```

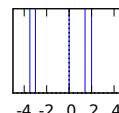


2

² $\sqrt{2}$ пришлось указать численно, значение функции не подставилось

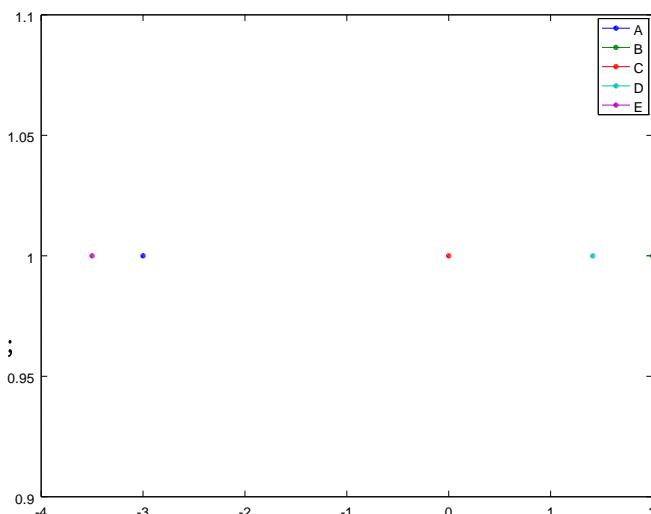
Листинг 23: Maxima

```
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 26.2 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 [68]), 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  
           1  
           -  
           4  
(%i5) (%o5)      C = - 1  
(%i6)
```

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> ...]  
    %<параметр_N>  
    %<параметр_1>  
    %...  
    %<параметр_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 [68]), 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

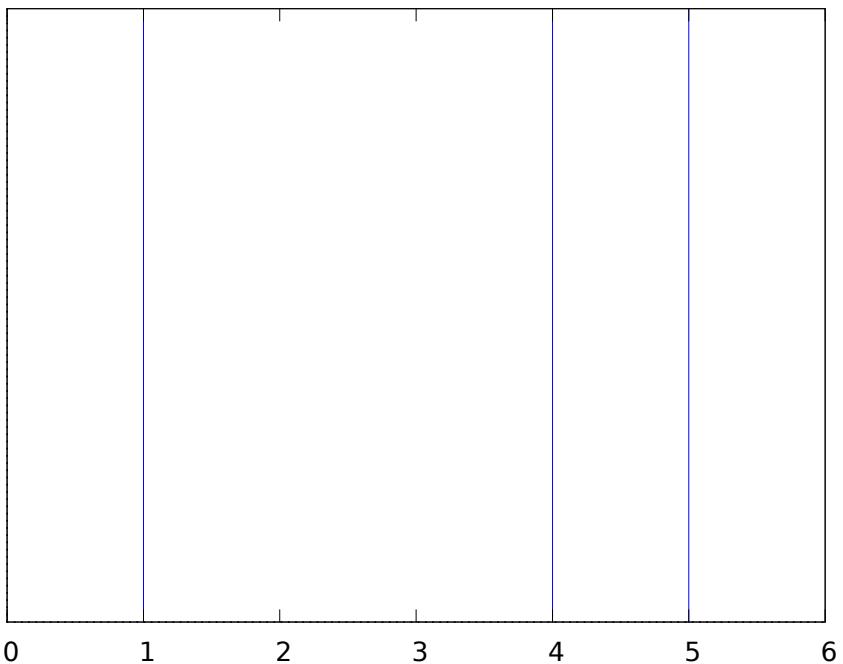
$$C(\bar{x}) = \frac{1 - 3 \cdot 5}{1 - 3} = C(7) \quad (26.4)$$

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 концы³:

1. $A(-6) B(7)$

2. $C(-5) D(0.5)$

³ используем формулу 26.3

```
% [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 26.3:

$$2 = \frac{(-3) + x_2}{2}$$

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 XIV

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 27

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.

27.1 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** [35]
- Тихонов, Самарский **Математическая физика** [44, 69]
- Демидович, Марон **Численные методы** [49, 50]
- Кремер **Теория вероятностей и матстатистика** [40]
- Ван дер Варден **Математическая статистика** [36]
- Кострикин **Введение в алгебру** [38, 39]

- Ван дер Варден **Алгебра** [37]

- Демидович **Сборник задач по математике для втузов. В 4 частях** [70, ?, ?, ?]
- Будак, Самарский, Тихонов **Сборник задач по математической физике** [69]

Фейнмановские лекции по физике

1. Современная наука о природе. Законы механики. [55]
2. Пространство. Время. Движение. [56]
3. Излучение. Волны. Кванты. [57]
4. Кинетика. Теплота. Звук. [58]
5. Электричество и магнетизм [59]
6. Электродинамика. [60]
7. Физика сплошных сред. [61]
8. Квантовая механика 1. [62]
9. Квантовая механика 2. [63]

- Цирельсон **Квантовая химия** [65]
- Розенброк **Вычислительные методы для инженеров-химиков** [66]
- Шрайвер Эткинс **Неорганическая химия** [67]

Обработка экспериментальных данных и метрология

- Смит **Цифровая обработка сигналов** [41]
- Князев, Черкасский **Начала обработки экспериментальных данных** [42]

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 кратко¹, без излишеств профессионального программирования², 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 параметрами³ (электроника, электротехника, механика, гидропневмоавтоматика и системы управления)
- **V-REP** робототехнический симулятор
- **SimChemistry**⁴ интересный демонстрационный симулятор химической кинетики молекул на микроуровне (обсчитывается движение и столкновение отдельных молекул)
- **Avogadro** 3D редактор молекул

¹ наследование, полиморфизм, операторы для пользовательских типов, использование библиотеки STL

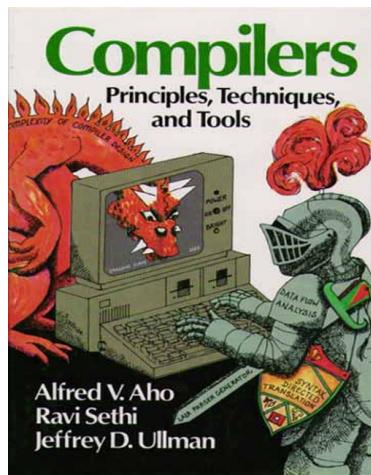
² мегабиблиотека Boost, написание своих библиотек шаблонов и т.п.

³ для описания моделей элементов использует ООП-язык Modelica

⁴ 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

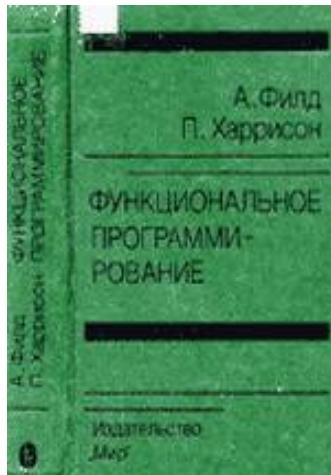
[3]

Структура и интерпретация компьютерных программ

Харольд Абельсон, Джеральд Сассман

ISBN 5-98227-191-8

EN: web.mit.edu/alexmv/6.037/sicp.pdf



[4]

Функциональное программирование

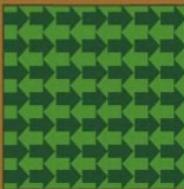
Филд А., Харрисон П.

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

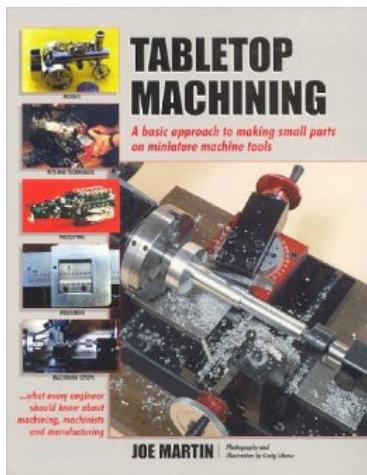
2 edition, May 2014

<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₊⁺**
© <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/>

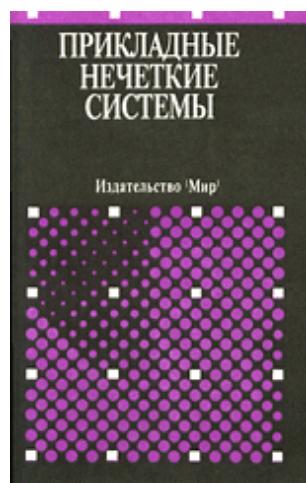
Python

- [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>

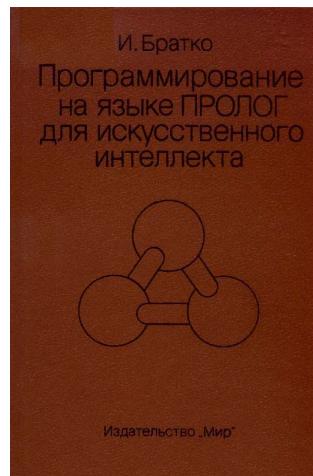
Python является простым и, в то же время, мощным интерпретируемым объектно-ориентированным языком программирования. Он предоставляет структуры данных высокого уровня, имеет изящный синтаксис и использует

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.

Prolog и логическое программирование

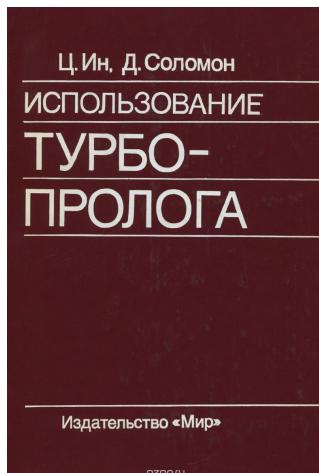


[27] Прикладные нечеткие системы
Тэрано Т., Асай К., Сугэно М. [djvu](#)

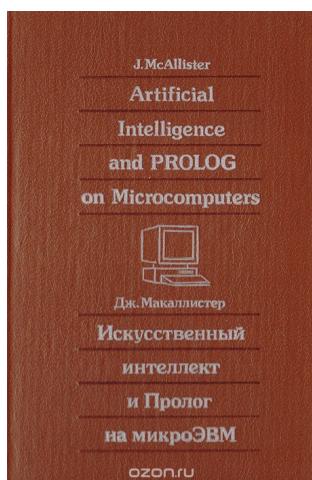


[28] Программирование на языке Пролог для искусственного интеллекта
Иван Братко

Мир, 1990
ISBN 5-03-001425-X, 0-201-14224-4



- [29] **Использование Турбо-Пролога**
Чин Маун Ин, Дэвид Соломон
Мир, 1993
ISBN 5-03-001181-1



- [30] **Искусственный интеллект и Пролог на микроЭВМ**
Дж. Макаллистер
Машиностроение, 1990
ISBN 5-217-00973-X

- [31] **Программирование на языке Пролог**
Клоксин У., Меллиш К.
Мир, 1987

- [32] **Искусство программирования на языке Пролог**
Л. Стерлинг, Э. Шапиро

Мир 1990

ISBN: 5-0300-0406-8

- [33] Интеллектуальные информационные системы. PROLOG- язык разработки интеллектуальных и экспертных систем: учебное пособие Хабаров С.П.
СПб. СПбГЛТУ, 2013.- 138 с. [pdf](#)

Разработка операционных систем и низкоуровневого ПО

- [34] OSDev Wiki
<http://wiki.osdev.org>

Базовые науки

Математика



- [35] 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>

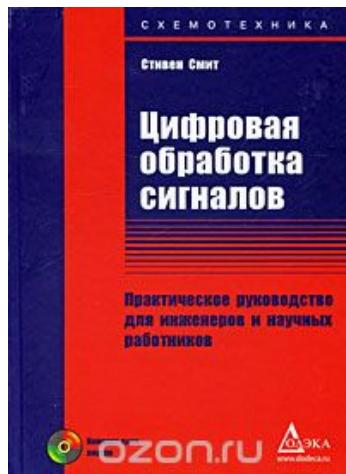
Пятое издание известного учебника, охватывает большинство вопросов программы по высшей математике для инженерно-технических специальностей вузов, в том числе дифференциальное исчисление функций одной переменной

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”.**

- [36] Математическая статистика Б.Л. Ван дер Варден
- [37] Алгебра Б.Л. Ван дер Варден
- [38] Введение в алгебру. В 3 частях. Часть 1. Основы алгебры А.И. Ко стрикин
- [39] Введение в алгебру. В 3 частях. Линейная алгебра. Часть 2 А.И. Кострикин



- [40] Теория вероятностей и математическая статистика
Наум Кремер
М.: Юнити, 2010



[41]

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

[42] Начала обработки экспериментальных данных

Б.А.Князев, В.С.Черкасский

Новосибирский государственный университет, кафедра общей физики, Новосибирск, 1996

http://www.phys.nsu.ru/cherk/Metodizm_old.PDF

Учебное пособие предназначено для студентов естественно-научных специальностей, 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.



[43]

Принципы современной математической физики Р. Рихтмайер

[44] Уравнения математической физики А.Н. Тихонов, А.А. Самарский

Символьная алгебра

[45] 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

[46]

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.



[47]

Элементы абстрактной и компьютерной алгебры

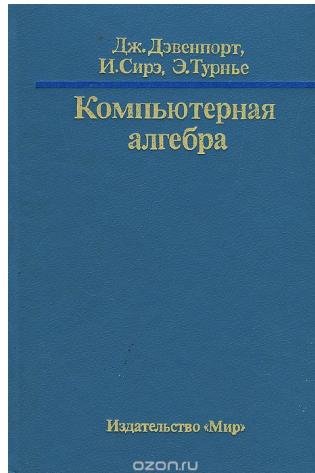
Дмитрий Матрос, Галина Поднебесова

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.



[48]

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

[49] **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.

[50] **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 [49]. Настоящее, третье издание отличается от предыдущего более доходчивым изложением. Добавлены новые примеры.

Теория игр

[51] 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.

[52] Математическая теория игр и приложения

Мазалов В.В.

Санкт-Петербург - Москва - Краснодар: Лань, 2010.

ISBN 978-5-8114-1025-5.

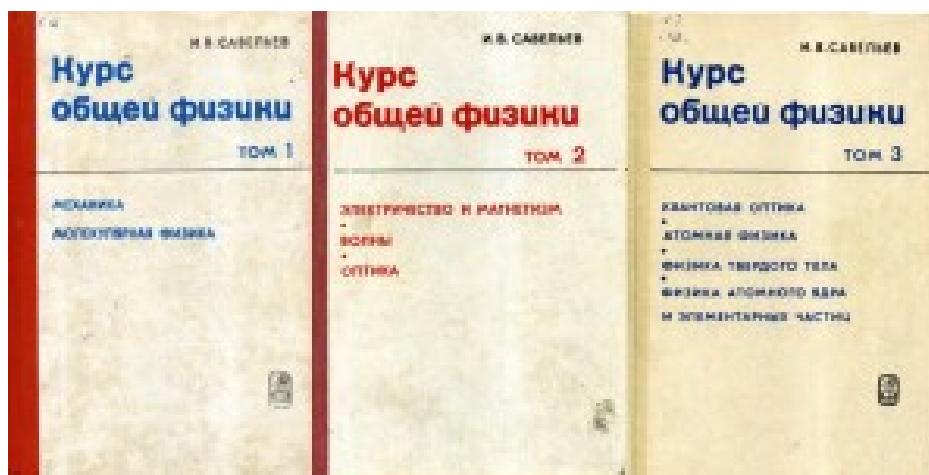
[53] Теория игр

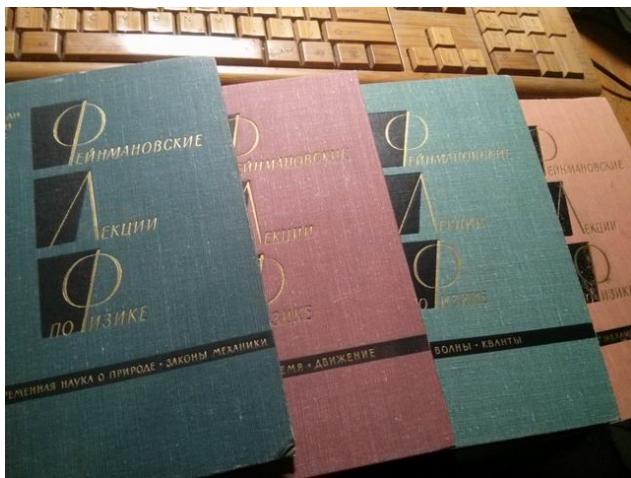
Оуэн Г.

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





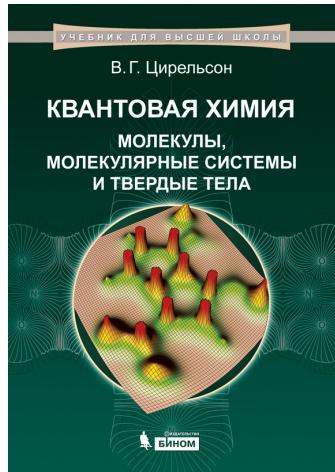
Фейнмановские лекции

по физике

Ричард Фейнман, Роберт Лейтон, Мэттью Сэндс

- [55] Современная наука о природе. Законы механики.
- [56] Пространство. Время. Движение.
- [57] Излучение. Волны. Кванты.
- [58] Кинетика. Теплота. Звук.
- [59] Электричество и магнетизм.
- [60] Электродинамика.
- [61] Физика сплошных сред.
- [62] Квантовая механика 1.
- [63] Квантовая механика 2.
- [64] Основы квантовой механики Д.И. Блохинцев

Химия



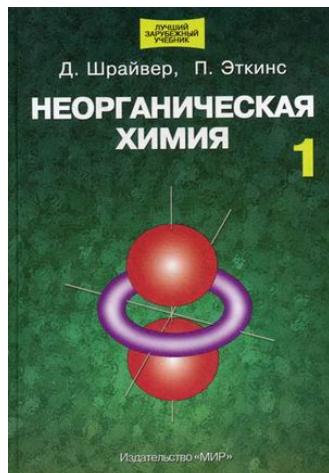
[65]

Квантовая химия. Молекулы, молекулярные системы и твердые тела. Учебное пособие Владимир Цирельсон



[66]

Вычислительные методы для инженеров-химиков X. Розенброк, С. Стори



[67]

Неорганическая химия В 2 томах
Д. Шрайвер, П. Эткинс

Задачники

Математика



[68]

Высшая математика в упражнениях и задачах
П.Е. Данко, А.Г.Попов, Т.Я. Кожевникова, С.П. Данко

[69] **Сборник задач по математической физике** Будак Б.М., Самарский А.А., Тихонов А.Н.

[70] **Сборник задач по математике для втузов. В 4 частях. Часть 1. Линейная алгебра и основы математического анализа**
Демидович

Стандарты и ГОСТы

- [71] 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

Предметный указатель

- Prolog,* unification, 34
 IeC функтор 139, variable, 36
 IeC тэг 140
арность, 139
константа, 139
куча, 140
несвязанная переменная, 140
переменная, 139
регистр, 142
сплющенная форма, 142
структурная ячейка, 140
субтерм, 139
терм, 139
терм программы, 139
терм запроса, 139
ячейка функтора, 140
ячейка переменной, 140
arity, 27
backtracking, 38
binding, 36
call, 29, 34
clause, 27
consult, 34
DCG, 240
definite clause grammar, 240
fact, 27
goal, 34
port, 38
predicate, 27
procedure, 27
program, 27
query, 29, 34
record, 27
relation, 27
rule, 27
„, 296
:, 261
абсцисса, 370
адрес хранения, 317
адрес размещения, 317
анонимная переменная, 82
базовый адрес, 298
бинарный формат, 298
биндинг логической переменной, 82
цель (Пролог), 88
дерево вывода, 90, 100
дерево заключений, 86
факт, 85, 88
граф смежности, 83
грамматика, 262
инкрементная компоновка, 306
канадский крест, 339
компоновка, 306
конъюнктивная цель, 82
консеквенция, 86
координата точки, 371
линкер, 297
линковка, 298
логическая переменная, 82
монитор **Qemu**, 300
назначение адресов, 298
низкоуровневое программирование, 293
объектный код, 296
оператор, 255
переменная цели, 88
правило, 88
привязка логической переменной, 82
разрешение символов, 307

релокация символов, 308
секционирование, 309
семантическое дерево, 83
символ, 255, 304
символьный тип, 254
синтаксическое дерево, 262
скрипт линкера, 313
состояние лексера, 264
спецификация MultiBoot, 349, 351
строчный комментарий, 263
таблица символов, 304
токен, 262
трассировка, 90
указатель адреса размещения, 313
унификация, 93, 100
вывод *Prolog*-программы, 86
заголовок правила, 88
загрузчик, 349

ABI, 315
application term, 122

backtracking, 101
bare metal, 293

closure, 123
cut, 103

ELF, 298

lambda term, 122
LMA, 317

make-правило, 330

standalone, 293
startup код, 317

VMA, 317