

Разработка программ с помощью Objective
Caml
Developing Applications With Objective Caml

Emmanuel CHAILLOUX Pascal MANOURY
Bruno PAGANO

18 мая 2007 г.

Оглавление

I	Основы языка	13
1	Функциональное программирование	15
1.1	Функциональное ядро Objective CAML	17
1.1.1	Значения, функции и базовые типы	17
1.1.2	Структуры условного контроля	23
1.1.3	Объявление значений	24
1.1.4	Функциональное выражение, функции	26
1.1.5	Полиморфизм и ограничение типа	34
1.1.6	Примеры	38
1.2	Объявления типов и сопоставление с образцом	40
1.2.1	Сопоставление с образцом	41
1.2.2	Декларация типов	49
1.2.3	Записи	51
1.2.4	Тип сумма (sum)	53
1.2.5	Рекурсивный тип	56
1.2.6	Параметризованный тип	57
1.2.7	Видимость описания	57
1.2.8	Функциональные типы	58
1.2.9	Пример: реализация деревьев	59
1.2.10	Не функциональные рекурсивные значения	61
1.3	Типизация, область определения и исключения	63
1.3.1	Частичные функции и исключения	63
1.3.2	Определения исключения	64
1.3.3	Возбуждение исключения	65
1.3.4	Перехват исключения	66
1.4	Полиморфизм и значения возвращаемые функциями	67
1.5	Калькулятор	69
1.6	Резюме	72

2	Императивное программирование	73
2.1	Физически изменяемые структуры данных	75
2.1.1	Векторы	75
2.1.2	Строки	79
2.1.3	Изменяемые поля записей	80
2.1.4	Указатели	81
2.1.5	Полиморфизм и изменяемые значения	82
2.2	Ввод/Вывод	84
2.2.1	Каналы	84
2.2.2	Чтение/запись	85
2.2.3	Пример Больше/Меньше	86
2.3	Структуры контроля	87
2.3.1	Последовательность	87
2.3.2	Циклы	89
2.3.3	Пример: реализация стека	91
2.3.4	Пример: расчет матриц	93
2.4	Порядок вычисления аргументов	94
2.5	Калькулятор с памятью	95
2.6	Резюме	98
3	Функциональный и императивный стиль	99
3.1	Сравнение между функциональным и императивным сти- лями	101
3.1.1	С функциональной стороны	101
3.1.2	Императивная сторона	102
3.1.3	Рекурсия или итерация	104
3.2	Какой стиль выбрать?	105
3.2.1	Последовательность или композиция функций . . .	106
3.2.2	Общее использование или копии значений	108
3.2.3	Критерии выбора	111
3.3	Смесь стилей	113
3.3.1	Замыкание и побочные эффекты	113
3.3.2	Физическое изменение и исключения	115
3.3.3	Изменяемые функциональные структуры данных . .	116
3.3.4	Пассивные изменяемые структуры	118
3.4	Поток данных	122
3.4.1	Конструкция	122
3.4.2	Деструкция и сопоставление потока	123
3.5	Резюме	127

4	Графический интерфейс	129
4.1	Использование библиотеки Graphics	130
4.2	Основные понятия	130
4.3	Графический вывод	131
4.3.1	Ориентир и графический контекст	132
4.3.2	Цвета	132
4.3.3	Рисунок и заливка	133
4.3.4	Текст	136
4.3.5	Bitmaps	138
4.3.6	Рисование рельефных блоков	139
4.4	Анимация	143
4.5	Обработка событий	145
4.5.1	Тип и функции событий	145
4.5.2	Скелет программы	146
4.5.3	Пример: telecran	148
4.6	Графический калькулятор	150
4.7	Резюме	155
5	Программы	157
5.1	Запросы базы данных	158
5.1.1	Формат данных	158
5.1.2	Чтение базы из файла	160
5.1.3	Общие принципы работы с базой данных	162
5.1.4	Критерии выборки	164
5.1.5	Обработка и вычисление	167
5.1.6	Пример	169
5.1.7	Дополнительные возможности	171
5.2	Интерпретатор языка BASIC	171
5.2.1	Абстрактный синтаксис	173
5.2.2	Вывод программы на экран	175
5.2.3	Лексический анализ	177
5.2.4	Синтаксический анализ	179
5.2.5	Вычисление	184
5.2.6	Последние штрихи	189
5.2.7	Что дальше?	191
5.3	Minesweeper	192
5.3.1	Абстрактное минное поле	192
5.3.2	Игровой интерфейс	200
5.3.3	Взаимодействие между программой и игроком	207

II Средства разработки	213
6 Компиляция и переносимость	219
6.1 Этапы компиляции	220
6.1.1 Компиляторы Objective CAML	221
6.1.2 Описание байт-код компилятора	222
6.2 Типы компиляции	224
6.2.1 Названия команд	224
6.2.2 Элементы компиляции	224
6.2.3 Расширения файлов Objective CAML	225
6.2.4 Байт-код компилятор	225
6.2.5 Нативный компилятор	227
6.2.6 Интерактивный интерпретатор	228
6.2.7 Создание интерпретатора	229
6.3 Автономный исполняемый файл	230
6.4 Переносимость и эффективность	231
6.4.1 Автономность и переносимость	232
6.4.2 Эффективность выполнения	232
7 Библиотеки	235
7.1 Классификация и использование библиотек	236
7.2 Автоматически загруженные библиотеки	237
7.3 Стандартная библиотека	238
7.3.1 Утилиты	239
7.3.2 Линейные структуры данных	240
7.3.3 Ввод/Вывод	247
7.3.4 Persistence	252
7.3.5 Системный интерфейс	260
7.4 Другие библиотеки дистрибутива	266
7.4.1 Точная арифметика	267
7.4.2 Динамическая загрузка кода	269
7.5 Упражнения	272
7.6 Резюме	272
8 Автоматический сборщик мусора	273
8.1 Введение	273
8.2 План главы	274
8.3 Программа и память	275
8.4 Выделение и освобождение памяти	275
8.4.1 Явное выделение памяти	275
8.4.2 Явное освобождение памяти	277

8.4.3	Неявное освобождение памяти	278
8.5	Автоматическая сборка памяти	279
8.5.1	Счетчики ссылок	279
8.5.2	Алгоритм-разведчик	280
8.5.3	Mark&Sweep	282
8.5.4	Stop&Copy	284
8.5.5	Другие GC	287
8.6	Управление памятью в Objective Caml	291
8.7	Модуль GC	292
8.8	Модуль Weak	295
8.9	Резюме	299
9	Средства анализа программ	301
9.1	Введение	301
9.2	План главы	302
9.3	Средства отладки	303
9.3.1	Trace	303
9.3.2	Отладка программ	309
9.3.3	Контроль над выполнением программы	310
9.4	Профайлер	312
9.4.1	Команды компиляции	313
9.4.2	Выполнение программы	313
9.4.3	Представление результата	315
9.5	Резюме	317
10	Средства лексического и синтаксического анализа	319
10.1	Введение	319
10.2	План главы	320
10.3	Лексика	320
10.3.1	Модуль Genlex	321
10.3.2	Использование потоков	322
10.3.3	Рациональные выражения	323
10.3.4	Библиотека Str	325
10.3.5	Инструмент Ocamllex	326
10.4	Синтаксис	328
10.4.1	Грамматика	329
10.4.2	Порождение и распознавание	330
10.4.3	Нисходящий анализ	331
10.4.4	Восходящий анализ	334
10.4.5	Инструмент ocamlyacc	338
10.4.6	Контекстная грамматика	341

10.5	Пересмотренный Basic	342
10.5.1	Файл <code>basic_parser.mly</code>	343
10.5.2	Файл <code>basic_lexer.mll</code>	346
10.5.3	Компиляция, компоновка	347
10.6	Резюме	348
11	Взаимодействие с языком C	349
11.1	Введение	349
11.2	План главы	350
11.3	Передача информации между Objective CAML и C	351
11.3.1	Внешние декларации	352
11.3.2	Декларация C функций	353
11.3.3	Редактирование связей с C	355
11.3.4	Смешивание операций ввода/вывода в C и Objective CAML	357
11.4	Анализ значений Objective CAML в C	358
11.4.1	Классификация значений <code>value</code>	359
11.4.2	Доступ к непосредственным значениям	359
11.4.3	Дискриминация структурных значений	361
11.4.4	Вызов замыкания Objective CAML в C	370
12	Программы	371
III	Устройство программы	373
13	Модульное программирование	375
14	Объектно-ориентированное программирование	377
15	Сравнение моделей устройств программ	379
16	Программы	381
IV	Параллелизм и распределение	383
17	Процессы и связь между процессами	385
18	Программирование одновременно—выполняемых задач	387
19	Программирование распределенных задач	389

<i>Оглавление</i>	9
20 Программы	391
21 Разработка программ с помощью Objective CAML	393
V Annexes	395

Список иллюстраций

2.1	Представление в памяти вектора с разделением элементов	77
2.2	Изменение элемента разделяемого вектора	77
3.1	Представление в памяти замыканий	115
4.1	Сеть эстафетное кольцо	135
4.2	Легенда координатных осей	137
4.3	Инверсия Jussieu	139
4.4	Вывод блоков с текстом	143
4.5	Перемещение объекта	145
4.6	Telecran	150
4.7	Графический калькулятор	150
5.1	Этапы запроса	163
5.2	Деревья абстрактного синтаксиса	179
5.3	Basic: Пример создания дерева абстрактного синтаксиса . .	180
5.4	Копия экрана	193
5.5	Основное окно игры	201
6.1	Виртуальная машина Zinc	223
6.2	Символы интерпретаторов и компиляторов.	223
6.3	Пример работы с байт-код компилятором	227
6.4	Сеанс работы в интерпретаторе.	230
8.1	Представление значения в памяти	276
8.2	Представление в памяти циклического списка	280
8.3	Высвобождение памяти сборщиком	281
8.4	Состояние кучи	281
8.5	Этап маркировки	283
8.6	Этап восстановления	284
8.7	Начало Stop&Copy	285
8.8	Копирование из from-space в to-space	285

8.9	Новые корни	286
8.10	Разделение значения	286
8.11	Смена ролей зон памяти	287
8.12	Выделение во время высвобождения памяти	291
11.1	Передача информации между Objective CAML и C	352
11.2	Смешанный исполняемый файл	355
11.3	Структура блока в куче Objective CAML	361
11.4	Представление замыкания	369

Часть I

Основы языка

Глава 1

Функциональное программирование

Введение

Первый язык функционального программирования LISP появился в конце 1950, в тот же момент, что и Fortran – один из первых императивных языков. Оба этих языка существуют и по сей день, хотя они немало изменились. Область их применения: вычислительные задачи для Фортрана и символьные (*symbolic*) для Lisp. Интерес к функциональному программированию состоит в простоте написания программ, где под программой подразумевается функция, примененная к аргументам. Она вычисляет результат, который возвращается как вывод программы. Таким образом можно с легкостью комбинировать программы: вывод одной, становится входным аргументом для другой.

Функциональное программирование основывается на простой модели вычислений, состоящей из трех конструкций: переменные, определение функции и ее применение к какому-либо аргументу. Эта модель, называемая λ -исчисление, была введена Alonzo CHURCH в 1932, еще до появления первых компьютеров. В λ -исчислении любая функция является переменной, так что она может быть использована как входной параметр другой функции, или возвращена как результат другой. Теория λ -исчисления утверждает что все то, что вычисляемо может быть представлено этим формализмом. Однако, синтаксис этой теории слишком ограничен, чтобы его можно было использовать его как язык программирования. В связи с этим к λ -исчислению были добавлены базовые типы (например, целые числа или строки символов), операторы для этих типов, управляющие структуры и объявление позволяющие именовать

переменные или функции, и в частности рекурсивные функции.

Существует разные классификации языков функционального программирования. Мы будем различать их по двум характеристикам, которые нам кажутся наиболее важными:

- без побочных эффектов (чистые, или правильные), или с побочным эффектом (соответственно, не чистые): чистый язык — это тот, в котором не существует изменения состояния. Все есть вычисление, и как оно происходит, нас не интересует. Не чистые языки, такие как *Caml* или *ML*, имеют императивные особенности, такие как изменение состояния. Они позволяют писать программы в стиле близкому к Фортрану, в котором важен порядок вычисления выражений.
- язык типизирован динамически или статически: типизация необходима для проверки соответствия аргумента, переданного функции, типу формального параметра. Это проверка может быть выполнена во время выполнения программы. В этом случае типизация называется динамической. В случае ошибки программа будет остановлена, как это происходит в *Lisp*. В случае статической типизации проверка осуществляется во время компиляции, то есть до выполнения программы. Таким образом она (проверка) не замедлит программу во время выполнения. Эта типизация используется в *ML* и в его диалектах, таких как *Objective CAML*. Только правильно типизированные программы, то есть успешно прошедшие проверку типов, могут быть скомпилированы и затем выполнены.

План главы

В этой главе представлены базовые элементы функциональной части языка *Objective CAML*, а именно: синтаксис, типы и механизм исключений. После этого вводного курса мы сможем написать нашу первую программу.

В первом разделе описаны основы языка, начиная с базовых типов и функций. Затем мы рассмотрим структурные и функциональные типы. После этого мы рассмотрим управляющие структуры, а также локальные и глобальные объявления. Во втором разделе мы обсудим определение типов для создания структур и механизм сопоставления с образцом, который используется для доступа к этим структурам. В третьем разделе рассматривается выводимый тип функций и область их применения, после чего следует описание механизма исключений. Четвертый

раздел объединяет введенные понятия в простой пример программы-калькулятора.

1.1 Функциональное ядро Objective CAML

Как любой другой язык функционального программирования, Objective CAML — это язык выражений, состоящий в основном из создания функций и их применения. Результатом вычисления одного из таких выражений является значение данного языка (*value in the language*) и выполнение программы заключается в вычислении всех выражений из которых она состоит.

1.1.1 Значения, функции и базовые типы

В Objective CAML определены следующие типы: целые числа, числа с плавающей запятой, символьный, строковый и логический.

Числа

Различают целые `int` и числа с плавающей запятой `float`. Objective CAML следует спецификации *IEEE 754* для представления чисел с плавающей запятой двойной точности. Операции с этими числами описаны в таблице 1.1. Если результат целочисленной операции выходит за интервал значений типа `int`, то это не приведет к ошибке. Результат будет находиться в интервале целых чисел, то есть все действия над целыми ограничены операцией `modulo` с границами интервала.

целые	плавающие
+ сложение	+. сложение
- вычитание и унарный минус	-. вычитание и унарный минус
* умножение	*. умножение
/ деление	/. деление
<code>mod</code> остаток целочисленного деления	** exponentiation

Разница между целыми числами и числами с плавающей запятой. Значения разных типов, таких как `float` и `int`, не могут сравниваться между собой напрямую. Для этого существует функции перевода одного типа в другой (`float_of_int` и `int_of_float`).

# 1 ;; - : int = 1 # 1 + 2 ;; - : int = 3 # 9 / 2 ;; - : int = 4 # 11 mod 3 ;; - : int = 2 (* лимит представления *) (* целых чисел *) # 2147483650 ;; - : int = 2	# 2.0 ;; - : float = 2 # 1.1 +. 2.2 ;; - : float = 3.3 # 9.1 /. 2.2 ;; - : float = 4.13636363636 # 1. /. 0. ;; - : float = Inf (* лимит представления *) (* с плавающей запятой *) # 22222222222.11111 ;; - : float = 22222222222
---	--

Таблица 1.1: Операции над числами

```
# 2=2.0;;
This expression has type float but is here used with type int
# 3.0=float_of_int 3;;
- : bool = true
```

Аналогично, операции над целыми числами и числами с плавающей запятой различны:

```
# 3+2;;
- : int = 5
# 3.0+.2.0;;
- : float = 5
# 3.0+2.0;;
Characters 0-3:
This expression has type float but is here used with type int
# sin 3.14159;;
- : float = 2.65358979335e-06
```

Неопределенный результат, например получаемый при делении на ноль, приведет к возникновению исключения (см. 1.3, стр. 63), которое остановит вычисление. Числа с плавающей запятой имеют специальные значения для бесконечных величин (**Inf**) и для не определенного результата (**NaN**¹). Основные операции над этими числами приведены в таблице 1.2

¹Not a Number

Ф-ии над числами с плавающей запятой	тригонометрические функции:
<code>ceil</code>	<code>cos</code> косинус
<code>floor</code>	<code>sin</code> синус
<code>sqrt</code> - квадратный корень	<code>tan</code> тангенс
<code>exp</code> - экспонента	<code>acos</code> арккосинус
<code>log</code> - натуральный логарифм	<code>asin</code> арксинус
<code>log10</code> - логарифм по базе 10	<code>atan</code> арктангенс

<code># ceil 3.4 ;;</code>	<code># sin 1.57078 ;;</code>
<code>- : float = 4</code>	<code>- : float = 0.999999999867</code>
<code># floor 3.4 ;;</code>	<code># sin (asin 0.707) ;;</code>
<code>- : float = 3</code>	<code>- : float = 0.707</code>
<code># ceil (-3.4) ;;</code>	<code># acos 0.0 ;;</code>
<code>- : float = -3</code>	<code>- : float = 1.57079632679</code>
<code># floor (-3.4) ;;</code>	<code># asin 3.14 ;;</code>
<code>- : float = -4</code>	<code>- : float = NaN</code>

Таблица 1.2: Функции над числами с плавающей запятой

Символы и строки

Символы, тип `char`, соответствуют целым числам в интервале от 0 до 255, первые 128 значений соответствуют кодам *ASCII*. Функция `char_of_int` и `int_of_char` преобразуют один тип в другой. Строки, тип `string` — это последовательность символов определенной длины (не длиннее $2^{24} - 6$). Оператором объединения строк (конкатенации) является шапка “`^`”. Следующие функции необходимы для перевода типов `int_of_string`, `string_of_int`, `string_of_float` и `float_of_string`.

```
# 'B' ;;
- : char = 'B'
# int_of_char 'B' ;;
- : int = 66
# "est une chaîne" ;;
- : string = "est une cha\238ne"
# (string_of_int 1987) ^ "est l'année de la création de CAML" ;;
- : string = "1987 est l'ann\233e de la cr\233ation de CAML"
```

Если строка состоит из цифр, то мы не сможем использовать ее в численных операциях, не выполнив явного преобразования.

```
# "1999" + 1 ;;
Characters 1-7:
```

This expression has type `string` but is here used with type `int`

```
# (int_of_string "1999") + 1 ;;
```

```
- : int = 2000
```

В модуле `String` собрано много функций для работы со строками (стр. 7.3.2)

Булевый тип

Значение типа `boolean` принадлежит множеству состоящему из двух элементов: `true` и `false`. Основные операторы описаны в таблице 1.3. По историческим причинам операторы `and` и `or` имеют две формы.

<code>not</code> - отрицание	
<code>&&</code> последовательное и	<code>&</code> синоним <code>&&</code>
<code> </code> последовательное или	<code>or</code> синоним <code> </code>

Таблица 1.3: Булевы операторы

```
# true ;;
- : bool = true
# not true ;;
- : bool = false
# true && false ;;
- : bool = false
```

Операторы `&&` и `||` или их синонимы, вычисляют аргумент слева и в зависимости от его значения, вычисляют правый аргумент. Они могут быть переписаны в виде условной структуры (см. 1.1.2 стр. 23).

<code>=</code> равенство структурное	<code><</code> меньше
<code>==</code> равенство физическое	<code>></code> больше
<code><></code> отрицание <code>=</code>	<code><=</code> меньше или равно
<code>!=</code> отрицание <code>==</code>	<code>>=</code> больше или равно

Таблица 1.4: Операторы сравнения и равенства

Операторы сравнения и равенства описаны в таблице 1.4. Это полиморфные операторы, то есть они применимы как для сравнения двух целых, так и двух строк. Единственным ограничением это то что операнды должны быть одного типа (см. 1.1.5 стр. 34).

```
# 1 <= 118 && (1 = 2 || not(1 = 2)) ;;
- : bool = true
# 1.0 <= 118.0 && (1.0 = 2.0 || not (1.0 = 2.0)) ;;
```

```

- : bool = true
# "un" < "deux" ;;
- : bool = false
# 0 < '0' ;;
Characters 4-7:

```

This expression has type `char` but is here used with type `int`

Структурное равенство, при проверке двух переменных, сравнивает значение полей структуры, тогда как физическое равенство проверяет занимают ли эти переменные одно и то же место в памяти. Оба оператора возвращают одинаковый результат для простых типов: `boolean`, `char`, `int` и константные конструкторы (см. 1.2.4 стр. 53).

Warning

числа с плавающей запятой и строки рассматриваются как структурные типы.

Unit

Тип `unit` определяет множество из всего одного элемента, значение которого: `()`

```

# () ;;
- : unit = ()

```

Это значение будет широко использоваться в императивных программах (см. 2 стр. 73), в функциях с побочным эффектом. Функции, результат которых равен `()`, соответствуют понятию процедуры, которое отсутствует в *Objective CAML*, так же как и аналог типа `void` в языке C.

Декартово произведение, кортежи

Значения разных типов могут быть сгруппированы в кортежи. Значения из которых состоит кортеж разделяются запятой. Для конструкции кортежа, используется символ `*`. `int*string` есть кортеж, в котором первый элемент целое число и второй строка.

```

v
# ( 12 , "octobre" ) ;;
- : int * string = 12, "octobre"

```

Иногда мы можем использовать более простую форму записи.

```
# 12 , "octobre" ;;
- : int * string = 12, "octobre"
```

Функции `fst` и `snd` дают доступ первому и второму элементу соответственно.

```
# fst ( 12 , "octobre" ) ;;
- : int = 12
# snd ( 12 , "octobre" ) ;;
- : string = "octobre"
```

Эти обе функции полиморфные, входной аргумент может быть любого типа.

```
# fst;;
- : 'a * 'b -> 'a = <fun>
# fst ( "octobre", 12 ) ;;
- : string = "octobre"
```

Тип `int*char*string` — это триплет, в котором первый элемент типа `int`, второй `char`, а третий — `string`.

```
# ( 65 , 'B' , "ascii" ) ;;
- : int * char * string = 65, 'B', "ascii"
```

Warning

Если аргумент функций `fst` и `snd` не пара, а другой n -кортеж, то мы получим ошибку.

```
# snd ( 65 , 'B' , "ascii" ) ;;
Characters 7-25:
```

This expression has type `int * char * string` but is here used with type `'a * 'b`

Существует разница между парой и триплетом: тип `int*int*int` отличен от `(int*int)*int` и `int*(int*int)`. Методы доступа к элементам триплета (и других кортежей) не определены в стандартной библиотеке. В случае необходимости мы используем сопоставление с образцом (см. 1.2.1).

Списки

Значения одного и того же типа могут быть объединены в списки. Список может быть либо пустым, либо содержать однотипные элементы.

```
# [] ;;
- : 'a list = []
# [ 1 ; 2 ; 3 ] ;;
- : int list = [1; 2; 3]
# [ 1 ; "deux" ; 3 ] ;;
Characters 15-18:
```

This expression has type `int list` but is here used with type `string list`

Для того чтобы добавить элемент в начало списка существует следующая функция в виде инфиксного оператора `::` — аналог `cons` в *Caml*.

```
# 1 :: 2 :: 3 :: [] ;;
- : int list = [1; 2; 3]
```

Для объединения (конкатенации) списков существует инфиксный оператор: `@`.

```
# [ 1 ] @ [ 2 ; 3 ] ;;
- : int list = [1; 2; 3]
# [ 1 ; 2 ] @ [ 3 ] ;;
- : int list = [1; 2; 3]
```

Остальные функции манипуляции списками определены в библиотеке `List`. Функции `hd` и `tl` дают доступ к первому и последнему элементу списка.

```
# List.hd [ 1 ; 2 ; 3 ] ;;
- : int = 1
# List.hd [] ;;
Uncaught exception: Failure("hd")
```

В последнем примере получить первый элемент пустого списка действительно “сложно”, поэтому возбуждается исключение (см. 1.3.1).

1.1.2 Структуры условного контроля

Одна из структур контроля необходимая в каждом языке программирования — условный оператор.

Синтаксис:

```
if expr1 then expr2 else expr3
```

Тип выражения `expr1` равен `bool`. Выражения `expr2` и `expr3` должны быть одного и того же типа.

```
# if 3=4 then 0 else 4 ;;
- : int = 4
```

```
# if 3=4 then "0" else "4" ;;
- : string = "4"
# if 3=4 then 0 else "4" ;;
```

Characters 20-23:

This expression has type string but is here used with type int

Условное выражение, как и любое другое, возвращает значение.

```
# (if 3=5 then 8 else 10) + 5 ;;
- : int = 15
```

Замечание

Ветка **else** может быть опущена, в этом случае будет подставлено значение по умолчанию равное **else ()**, в соответствии с этим выражение **expr2** должно быть типа **unit** (см. 2.3 стр. 87).

1.1.3 Объявление значений

Определение связывает имя со значением. Различают глобальные и локальные определения. В первом случае, объявленные имена видны во всех выражениях, следуют за ним, во втором — имена доступны только в текущем выражении. Мы также можем *одновременно* объявить несколько пар имя-значение.

Глобальные объявления

Синтаксис:

```
let nom = expr;
```

Глобальное объявление определяет связь имени **nom** со значением выражения **expr**, которое будет доступно всем следующим выражениям.

```
# let an = "1999" ;;
val an : string = "1999"
# let x = int_of_string(an) ;;
val x : int = 1999
# x ;;
- : int = 1999
# x + 1 ;;
- : int = 2000
# let nouvel_an = string_of_int (x + 1) ;;
val nouvel_an : string = "2000"
```


Одновременное глобальное объявление

Синтаксис:

```
let nom1 = expr1
and nom2 = expr2
:
and nomn = exprn}
```

При одновременном объявлении переменные будут известны только к концу всех объявлений.

```
# let x = 1 and y = 2 ;;
val x : int = 1
val y : int = 2
# x + y ;;
- : int = 3
# let z = 3 and t = z + 2 ;;
```

Characters 18-19:

Unbound value z

Можно сгруппировать несколько глобальных объявлений в одной фразе, вывод типов и значений произойдет к концу фразы, отмеченной “;;”. В данном случае объявления будут вычислены по порядку.

```
# let x = 2
  let y = x + 3 ;;
val x : int = 2
val y : int = 5
```

Глобальное объявление может быть скрыто локальным с тем же именем (см. 1.1.4 стр. 32).

Локальное объявление

Синтаксис:

```
let nom = expr1 in expr2};;
```

Имя **nom** связанное с выражением **expr1** известно только для вычисления **expr2**.

```
# let x1 = 3 in x1 * x1 ;;
- : int = 9
```

Локальное объявление, которое связывает **x1** со значением 3, существует только в ходе вычисления **x1*x1**.

```
# x1 ;;
Characters 1-3:
Unbound value x1
```

Локальное объявление скрывает любое глобальное с тем же именем, но как только мы выходим из блока в котором была оно определено, мы находим старое значение связанное с этим именем.

```
# let x = 2 ;;
val x : int = 2
# let x = 3 in x * x ;;
- : int = 9
# x * x ;;
- : int = 4
```

Локальное объявление — это обычное выражение, соответственно оно может быть использовано для построения других выражений.

```
# (let x = 3 in x * x) + 1 ;;
- : int = 10
```

Локальные объявления так же могут быть одновременными.

```
let  nom1 = expr1
and  nom2 = expr2
:
and  nomn = exprn
in   expr ;;

# let a = 3.0 and b = 4.0 in sqrt (a*.a +. b*.b) ;;
- : float = 5
# b ;;
Characters 0-1:
Unbound value b
```

1.1.4 Функциональное выражение, функции

Функциональное выражение состоит из *параметра* и *тела*. Формальный параметр — это имя переменной, а тело — это выражение. Обычно говорят что формальный параметр является абстрактным, по этой причине функциональное выражение тоже называется абстракцией.

Синтаксис:

```
function p -> expr
```

Таким образом функция возведения в квадрат будет выглядеть так:

```
# function x -> x*x ;;
- : int -> int = <fun>
```

Objective CAML сам определяет тип. Функциональный тип `int->int` это функция с параметром типа `int`, возвращающая значение типа `int`.

Функция с одним аргументом пишется как функция и аргумент следующий за ней.

```
# (function x -> x * x) 5 ;;
- : int = 25
```

Вычисление функции состоит в вычисление ее тела, в данном случае `x*x`, где формальный параметр `x`, заменен значением аргумента (эффективным параметром), здесь он равен 5.

При конструкции функционального выражения `expr` может быть любым выражением, в частности функциональным.

```
# function x -> (function y -> 3*x + y) ;;
- : int -> int -> int = <fun>
```

Скобки не обязательны, мы можем писать просто:

```
# function x -> function y -> 3*x + y ;;
- : int -> int -> int = <fun>
```

В простом случае мы скажем, что функция ожидает два аргумента целого типа на входе и возвращает значение целого типа. Но когда речь идет о функциональном языке, таком как Objective CAML, то скорее это соответствует типу функции с входным аргументом типа `int` и возвращающая функциональное значение типа `int->int`:

```
# (function x -> function y -> 3*x + y) 5 ;;
- : int -> int = <fun>
```

Естественно, мы можем применить это функциональное выражение к двум аргументам. Для этого напишем:

```
# (function x -> function y -> 3*x + y) 4 5 ;;
- : int = 17
```

Когда мы пишем `f a b`, подразумевается применение `(f a)` к `b`.

Давайте подробно рассмотрим выражение

```
(function x -> function y -> 3*x + y) 4 5
```

Для того чтобы вычислить это выражение, необходимо сначала вычислить значение

```
(function x -> function y -> 3*x + y) 4
```

что есть *функциональное выражение* равное

```
function y -> 3*4 + y
```

в котором **x** заменен на 4 в выражении $3*x+y$. Применяя это значение (являющееся функцией) к 5, мы получаем конечное значение $3*4+5=17$:

```
# (function x -> function y -> 3*x + y) 4 5 ;;
- : int = 17
```

Аргументность функции

Аргументностью функции называется число аргументов функции. По правилам, унаследованным из математики, аргументы функции задаются в скобках после имени функции. Мы пишем: $f(4,5)$. Как было указано ранее, в Objective CAML мы чаще используем следующий синтаксис: **f** 4 5. Естественно, можно написать функциональное выражение примененное к (4,5):

```
# function (x,y) -> 3*x + y ;;
- : int * int -> int = <fun>
```

Но в данном случае, функция ожидает не два аргумента, а один; тип которого пара целых. Попытка применить два аргумента к функции ожидающей пару или наоборот, передать пару функции для двух аргументов приведет к ошибке:

```
# (function (x,y) -> 3*x + y) 4 5 ;;
Characters 29-30:
```

This expression has type int but is here used with type int * int

```
# (function x -> function y -> 3*x + y) (4, 5) ;;
Characters 39-43:
```

This expression has type int * int but is here used with type int

Альтернативный синтаксис

Существует более компактная форма записи функций с несколькими аргументами, которая дошла к нам из старых версий *Caml*. Выглядит она так:

Синтаксис

```
fun p1 ... pn -> expr
```

Это позволяет не повторять слово **function** и стрелки. Данная запись эквивалентна

```
function pl -> -> function pn -> expr
```

```
# fun x y -> 3*x + y ;;
- : int -> int -> int = <fun>
# (fun x y -> 3*x + y) 4 5 ;;
- : int = 17
```

Эту форму можно часто встретить в библиотеках идущих в поставку с Objective CAML.

Замыкание

Objective CAML рассматривает функциональное выражение также как любое другое. Значение возвращаемое функциональным выражением называется замыканием. Каждое выражение Objective CAML вычисляется в окружении состоящем из соответствий имя–значение, которые были объявлены до вычисляемого выражения. Замыкание может быть описано как триплет, состоящий из имени формального параметра, тела функции и окружения выражения. Нам необходимо хранить это окружение, поскольку в теле функции кроме формальных параметров могут использоваться другие переменные. В функциональном выражении эти переменные называются свободными, нам понадобится их значение в момент применения функционального выражения.

```
# let m = 3 ;;
val m : int = 3
# function x -> x + m ;;
- : int -> int = <fun>
# (function x -> x + m) 5 ;;
- : int = 8
```

В случае когда применение замыкания к аргументу возвращает новое замыкание, оно (новое замыкание) получает в свое окружение все необходимые связи для следующего применения. Раздел 1.1.4 подробно рассматривает эти понятия. В главе 3, а так же в главе 11 мы вернемся к тому как замыкание представляется в памяти.

До сих пор рассмотренные функциональные выражения были *анонимными*, однако мы можем дать им имя.

Объявление функциональных значений

Функциональное значение объявляется так же как и другие, при помощи конструктора **let**

```
# let succ = function x -> x + 1 ;;
val succ : int -> int = <fun>
# succ 420 ;;
- : int = 421
# let g = function x -> function y -> 2*x + 3*y ;;
val g : int -> int -> int = <fun>
# g 1 2;;
- : int = 8
```

Для упрощения записи, можно использовать следующий синтаксис:
Синтаксис:

```
let nom p1...pn=expr
```

что эквивалентно:

```
let nom=function p1->->function pn-> expr
```

Следующие объявления `succ` и `g` эквивалентны предыдущим:

```
# let succ x = x + 1 ;;
val succ : int -> int = <fun>
# let g x y = 2*x + 3*y ;;
val g : int -> int -> int = <fun>
```

В следующем примере демонстрируется функциональная сторона Objective CAML, где функция `h1` получена применением `g` к аргументу. В данном случае мы имеем дело с частичным применением.

```
# let h1 = g 1 ;;
val h1 : int -> int = <fun>
# h1 2 ;;
- : int = 8
```

С помощью `g` мы можем определить другую функцию `h2` фиксируя значение второго параметра (`y`):

```
# let h2 = function x -> g x 2 ;;
val h2 : int -> int = <fun>
# h2 1 ;;
- : int = 8
```

Объявление инфиксных функций

Некоторые бинарные функции могут быть использованы в инфиксной форме. Например при сложении двух целых мы пишем `3+5` для применения `+` к `3` и `5`. Для того чтобы использовать символ `+` как классиче-

ское функциональное значение, необходимо указать это, окружая символ скобками (op).

В следующем примере определяется функция `succ` используя (+):

```
# (+) ;;
- : int -> int -> int = <fun>
# let succ = (+) 1 ;;
val succ : int -> int = <fun>
# succ 3 ;;
- : int = 4
```

Таким образом мы можем определить новые операторы, что мы и сделаем определив ++ для сложения двух пар целых.

```
# let (++) c1 c2 = (fst c1)+(fst c2), (snd c1)+(snd c2) ;;
val ++ : int * int -> int * int -> int * int = <fun>
# let c = (2,3) ;;
val c : int * int = 2, 3
# c ++ c ;;
- : int * int = 4, 6
```

Существуют однако ограничения на определение новых операторов, они должны содержать только *символы* (такие как *, +, \$, etc.), исключая буквы и цифры. Следующие функции являются исключением:

or mod land lor lxor lsr asr

Функции высшего порядка

Функциональное значение (замыкание) может быть возвращено как результат, а так же передано как аргумент функции. Такие функции, берущие на входе или возвращающие функциональные значения, называются функциями высшего порядка.

```
# let h = function f -> function y -> (f y) + y ;;
val h : (int -> int) -> int -> int = <fun>
```

Замечание

Выражения группируются справа налево, но функциональные типы объединяются слева направо. Таким образом тип функции `h` может быть написан:

```
(int -> int) -> int -> int
```

или

```
(int -> int) -> (int -> int)
```

При помощи функций высшего порядка можно элегантно обрабатывать списки. К примеру функция `List.map` применяет какую-нибудь функцию ко всем элементам списка и возвращает список результатов.

```
# List.map ;;
- : ('a -> 'b) -> 'a list -> 'b list = <fun>
# let square x = string_of_int (x*x) ;;
val square : int -> string = <fun>
# List.map square [1; 2; 3; 4] ;;
- : string list = ["1"; "4"; "9"; "16"]
```

Другой пример — функция `List.for_all` проверяет соответствуют ли элементы списка определенному критерию.

```
# List.for_all ;;
- : ('a -> bool) -> 'a list -> bool = <fun>
# List.for_all (function n -> n<>0) [-3; -2; -1; 1; 2; 3] ;;
- : bool = true
# List.for_all (function n -> n<>0) [-3; -2; 0; 1; 2; 3] ;;
- : bool = false
```

Видимость переменных

Для вычисления выражения, необходимо чтобы все используемые им переменные были определены, как, например, выражение `e` в определении

```
let p=e
```

Переменная `p` не известна в этом выражении, она может быть использована только в случае если `p` была объявлена ранее.

```
# let p = p ^ "-suffixe" ;;
Characters 9-10:
Unbound value p
# let p = "préfixe" ;;
val p : string = "pr\233fixe"
# let p = p ^ "-suffixe" ;;
val p : string = "pr\233fixe-suffixe"
```

В Objective CAML переменные связаны статически. При применении замыкания используется окружение в момент ее (замыкания) объявления (статическая видимость), а не в момент ее применения (динамическая видимость)


```

# let p = 10 ;;
val p : int = 10
# let k x = (x, p, x+p) ;;
val k : int -> int * int * int = <fun>
# k p ;;
- : int * int * int = 10, 10, 20
# let p = 1000 ;;
val p : int = 1000
# k p ;;
- : int * int * int = 1000, 10, 1010

```

В функции **k** имеется свободная переменная **p**, которая была определена в глобальном окружении, поэтому определение **k** принято. Связь между именем **p** и значением 10 в окружении замыкания **k** статическая, то есть не зависит от последнего определения **p**.

Рекурсивное объявление

Объявление переменной называется рекурсивным, если оно использует свой собственный идентификатор в своем определении. Эта возможность часто используется для определения рекурсивных функций. Как мы видели ранее, **let** не позволяет делать это, поэтому необходимо использовать специальный синтаксис:

```
let rec nom = expr ;;
```

Другой способ записи для функции с аргументами:

```
let rec nom p1 ... pn = expr ;;
```

Определим функцию **sigma** вычисляющую сумму целых от 0 до значения указанного аргументом:

```

# let rec sigma x = if x = 0 then 0 else x + sigma (x-1) ;;
val sigma : int -> int = <fun>
# sigma 10 ;;
- : int = 55

```

Как заметил читатель, эта функция рискует “не закончиться” если входной аргумент меньше 0.

Обычно, рекурсивное значение — это функция, компилятор не принимает некоторые рекурсивные объявления, значения которых не функциональные.

```
# let rec x = x + 1 ;;
```

Characters 13-18:

This kind of expression is not allowed as right-hand side of 'let rec'

Как мы увидим позднее, такие определения все таки возможны в некоторых случаях (см. 1.2.10 стр. 61).

Объявление **let rec** может быть скомбинировано с **and**. В этом случае функции определенные на одном и том же уровне, видны всем остальным. Это может быть полезно при декларации взаимно рекурсивных функций.

```
# let rec pair  n = (n<>1) && ((n=0) or (impair (n-1)))
and impair n = (n<>0) && ((n=1) or (pair (n-1)));
val pair : int -> bool = <fun>
val impair : int -> bool = <fun>
# pair 4 ;;
- : bool = true
# impair 5 ;;
- : bool = true
```

По тому же принципу, локальные функции могут быть рекурсивными. Новое определение функции **sigma** проверяет корректность входного аргумента, перед тем как посчитать сумму локальной функцией **sigma_rec**.

```
# let sigma x =
  let rec sigma_rec x =
    if x = 0 then 0 else x + sigma_rec (x-1) in
    if (x<0) then "erreur : argument negatif"
    else "sigma = " ^ (string_of_int (sigma_rec x)) ;;
val sigma : int -> string = <fun>
```

Замечание

Мы вынуждены были определить возвращаемый тип как **string**, поскольку необходимо чтобы он был один и тот же, независимо от входного аргумента, отрицательного или положительного. Какое значение должна вернуть **sigma** если аргумент больше нуля? Далее, мы увидим правильный способ решения этой проблемы (см. 1.3 стр. 63).

1.1.5 Полиморфизм и ограничение типа

Некоторые функции выполняют одни и те же инструкции независимо от типа аргументов. К примеру, для создание пары из двух значений нет смысла определять функции для каждого известного типа. Другой

пример, доступ к первому полю пары не зависит от того, какого типа это поле.

```
# let make_pair a b = (a,b) ;;
val make_pair : 'a -> 'b -> 'a * 'b = <fun>
# let p = make_pair "papier" 451 ;;
val p : string * int = "papier", 451
# let a = make_pair 'B' 65 ;;
val a : char * int = 'B', 65
# fst p ;;
- : string = "papier"
# fst a ;;
- : char = 'B'
```

Функция, для которой не нужно указывать тип входного аргумента или возвращаемого значения называется полиморфной. Синтезатор типов, включенный в компилятор Objective CAML находит наиболее общий тип для каждого выражения. В этом случае Objective CAML использует переменные, здесь они обозначены как `'a` и `'b`, для указания общих типов. Эти переменные конкретизируются типом аргумента в момент применения функции.

При помощи полиморфных функций, мы получаем возможность написания универсального кода для любого типа переменных, сохраняя при этом надежность статической типизации. Действительно, несмотря на то что `make_paire` полиморфная, значение созданное (`make_paire 'b' 65`) имеет строго определенный тип, который отличен от (`make_paire "paire"451`). Проверка типов реализуется в момент компиляции, таким образом универсальность кода никак не сказывается на эффективности программы.

Пример функций и полиморфных значений

В следующем примере приведена полиморфная функция с входным параметром функционального типа.

```
# let app = function f -> function x -> f x ;;
val app : ('a -> 'b) -> 'a -> 'b = <fun>
```

Мы можем применить ее к функции `impaire`, которая была определена ранее.

```
# app impair 2;;
- : bool = false
```

Функция тождества (`id`) возвращает полученный аргумент.

```
# let id x = x ;;
val id : 'a -> 'a = <fun>
# app id 1 ;;
- : int = 1
```

Следующая функция, `compose` принимает на входе две функции и еще один аргумент, к которому применяет две первые.

```
# let compose f g x = f (g x) ;;
val compose :
('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
# let add1 x = x+1 and mul5 x = x*5 in
compose mul5 add1 9 ;;
- : int = 50
```

Как мы видим, тип результата возвращаемого `g` должен быть таким же как и тип входного аргумента `f`.

Не только функциональные значения могут быть полиморфными, проиллюстрируем это на примере пустого списка.

```
# let l = [] ;;
val l : 'a list = []
```

Следующий пример иллюстрирует тот факт, что создание типов основывается на разрешении ограничений, накладываемых применением функций, а вовсе не на значениях, полученных в процессе выполнения.

```
# let q = List.tl [2] ;;
val q : int list = []
```

Здесь тип равен `List.tl 'a list -> 'a list`, то есть эта функция применяется к списку целых и возвращает список целых. Тот факт что в момент выполнения возвращен пустой список, не влияет на его тип.

Objective CAML создает параметризованные типы для каждой функции, которые не зависят от ее аргументов. Этот вид полиморфизма называется *параметрическим полиморфизмом*.²

Ограничение типа

Синтезатор типа Objective CAML образует самый общий тип и иногда бывает необходимо уточнить тип выражения.

Синтаксис ограничения типа следующий:

²Некоторые встроенные функции не подчиняются этому правилу, особенно функция структурного равенства (`=`), которая является полиморфной (ее тип `'a -> 'a -> bool`), но она исследует структуру своих аргументов для проверки равенства.

(expr:t)

В этом случае синтезатор типа воспользуется этим ограничением при конструкции типа выражения. Использование ограничения типа позволяет

- сделать видимым тип параметра функции
- запретить использование функции вне своей области применения
- уточнить тип выражения, это окажется необходимым в случае физически изменяемых значений (см. 2.1 стр. 75).

Рассмотрим использование ограничения типа.

```
# let add (x:int) (y:int) = x + y ;;
val add : int -> int -> int = <fun>
# let make_pair_int (x:int) (y:int) = x,y;;
val make_pair_int :
int -> int -> int * int = <fun>
# let compose_fn_int (f : int -> int)
  (g : int -> int) (x:int) = compose f g x;;
val compose_fn_int : (int -> int) ->
  (int -> int) -> int -> int = <fun>
# let nil = ([] : string list) ;;
val nil : string list = []
# 'H'::nil ;;
```

Characters 5-8:

This expression has type string list but is here used with type char list

Это ограничение полиморфизма позволяет лучше контролировать тип выражений, тем самым ограничивая тип определенный системой. В таких выражениях можно использовать любой определенный тип.

```
# let llnil = ([] : 'a list list) ;;
val llnil : 'a list list = []
# [1;2;3]:: llnil ;;
- : int list list = [[1; 2; 3]]
```

`llint` является списком списков любого типа.

В данном случае подразумевается ограничение типа, а не явная типизация заменяющая тип, который определил Objective CAML. В частности, мы не можем обобщить тип, более чем это позволяет вывод типов Objective CAML.

```
# let add_general (x:'a) (y:'b) = add x y ;;
val add_general : int -> int -> int = <fun>
```

Ограничения типа будут использованы в интерфейсах модулей 13, а так же в декларации классов 14

1.1.6 Примеры

В этом параграфе мы приведем несколько примеров функций. Большинство из них уже определены в Objective CAML, мы делаем это только в “педагогических” целях.

Тест остановки рекурсивных функций реализован при помощи проверки, имеющей стиль более близкий к Lisp. Мы увидим как это сделать в стиле *ML* (см. 1.2.1 стр. 41).

Размер списка

Начнем с функции проверяющей пустой список или нет.

```
# let null l = (l=[]) ;;
val null : 'a list -> bool = <fun>
```

Определим функцию **size** вычисления размера списка (т.е. число элементов).

```
# let rec size l =
  if null l then 0
  else 1+(size(List.tl l)) ;;
val size : 'a list -> int = <fun>
# size [] ;;
- : int = 0
# size [1;2;18;22] ;;
- : int = 4
```

Функция **size** проверяет список: если он пуст возвращает 0, иначе прибавляет 1 к длине остатка списка.

Итерация композиций (Iteration of composition)

Выражение **iterate n f** вычисляет f^n , соответствующее применение функции **f** **n** раз.

```
# let iterate n f =
  if n=0 then (function x->x)
  else compose f (iterate (n-1) f) ;;
val iterate : int -> ('a -> 'a) -> 'a -> 'a = <fun>
```

Функция `iterate` проверяет аргумент `n` на равенство нулю, если аргумент равен, то возвращаем функцию идентичности, иначе возвращаем композицию `f` с итерацией `f` `n-1` раз.

Используя `iterate` можно определить операцию возведения в степень, как итерацию умножения.

```
# let power i n =
let i_times = (*) i in
iterate n i_times 1;;
val power : int -> int -> int = <fun>
# power 2 8 ;;
- : int = 256
```

Функция `power` повторяет `n` раз функциональное выражение `i_times`, затем применяет этот результат к 1, таким образом мы получаем `n`-ю степень целого числа.

Таблица умножения

Напишем функцию `multab`, которая вычисляет ряд таблицы умножения соответствующую целому числу переданному в аргументе.

Для начала определим функцию `apply_fun_list`. Пусть `f_list` список функций, тогда вызов `apply_fun_list x f_list` возвращает список результатов применения каждого элемента списка `f_list` к `x`.

```
# let rec apply_fun_list x f_list =
  if null f_list then []
  else ((List.hd f_list) x) :: (apply_fun_list x (List.tl f_list)) ;;
val apply_fun_list : 'a -> ('a -> 'b) list -> 'b list = <fun>
# apply_fun_list 1 [(+) 1; (+) 2; (+) 3] ;;
- : int list = [2; 3; 4]
```

Функция `mk_mult_fun_list` возвращает список функций умножающих их аргумент на `i`, $0 \leq i \leq n$.

```
# let mk_mult_fun_list n =
  let rec mmfl_aux p =
    if p = n then [ ( * ) n ]
    else (( * ) p) :: (mmfl_aux (p+1))
  in (mmfl_aux 1) ;;
val mk_mult_fun_list : int -> (int -> int) list = <fun>
```

Подсчитаем ряд для 7

```
# let multab n = apply_fun_list n (mk_mult_fun_list 10) ;;
val multab : int -> int list = <fun>
# multab 7 ;;
- : int list = [7; 14; 21; 28; 35; 42; 49; 56; 63; 70]
```

Итерация в списке

Вызов функции `fold_left f a [e1; e2; ...; en]` возвращает `f... (f (f a e1) e2) ... en`, значит получаем `n` применений.

```
# let rec fold_left f a l =
  if null l then a
  else fold_left f (f a (List.hd l)) (List.tl l) ;;
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
```

С помощью функции `fold_left` можно компактно определить функцию вычисления суммы элементов списка целых чисел.

```
# let sum_list = fold_left (+) 0 ;;
val sum_list : int list -> int = <fun>
# sum_list [2;4;7] ;;
- : int = 13
```

Или, например конкатенация элементов списка строк.

```
# let concat_list = fold_left (^) "" ;;
val concat_list : string list -> string = <fun>
# concat_list ["Hello "; "world" ; " "; "!"] ;;
- : string = "Hello world !"
```

1.2 Объявления типов и сопоставление с образцом

С помощью типов определенных в Objective CAML мы можем определять структурные типы, при помощи кортежей или списков. Но в некоторых случаях бывает необходимо определить новые типы для описания специальных структур. В Objective CAML, объявления типов рекурсивны и могут быть параметризованы переменными типа, как в случае `'a list` который мы уже обсуждали. Вывод типов принимает во внимание новые объявления для определения типов выражений. Конструкция значений новых типов использует конструктор описанный определением типа.

Особой особенностью языков семейства *ML* является сопоставление с образцом, которое обеспечивает простой метод доступа к компонентам (полям) структур данных. Определение функции соответствует чаще всего сопоставлению с образцом по одному из ее параметров, что позволяет определять функции для различных случаев.

Для начала, продемонстрируем механизм сопоставления на существующих типах и затем опишем различные объявления структурных типов, конструкций таких переменных и доступ к компонентам по сопоставлению с образцом.

1.2.1 Сопоставление с образцом

Образец (по английски *pattern*) строго говоря не совсем выражение *Objective CAML*. Речь скорее идет о корректной компоновке (синтаксической и с точки зрения типов) элементов, таких как константы базовых типов (`int`, `bool`, `char...`), переменные, конструкторы и символ `_`, называемый универсальным образцом. Другие символы служат для записи шаблона, которые мы опишем по ходу.

Сопоставление с образцом применяется к значениям, оно служит для распознавания формы этих значений и позволяет определять порядок вычислений. С каждым образцом связано выражение для вычисления.

Синтаксис:

```
match expr with
| p1 -> expr1
:
| pn -> exprn
```

Выражение *expr* последовательно сопоставляется (фильтруется) с разными образцами *p*₁ . . . , *p*_{*n*}. Если один из образцов (например *p*_{*i*}) соответствует значению *expr*, то соответствующее ответвление будет вычислено (*expr*_{*i*}). Образцы *p*_{*i*} одинакового типа, так же как и *expr*_{*i*}. Вертикальная черта перед первым образцом не обязательна.

Пример

Приведем два способа, при помощи сопоставления с образцом, определения функции `imply` типа `(bool * bool) -> bool`, реализующую логическую импликацию. Образец для пар — `(,)`.

Первая версия перечисляет все возможности, как таблица истинности.

```
# let imply v = match v with
```

```

      (true,true)  -> true
    | (true,false) -> false
    | (false,true) -> true
    | (false,false) -> true;;
val imply : bool * bool -> bool = <fun>

```

Используя переменные группирующие несколько случаев, мы получаем более компактное определение.

```

# let imply v = match v with
  (true,x)  -> x
  | (false,x) -> true;;
val imply : bool * bool -> bool = <fun>

```

Обе версии `imply` выполняют одну и ту же функцию, то есть они возвращают одинаковые значения, для одинаковых входных аргументов.

Линейный образец

Образец обязательно должен быть линейным, то есть определенная переменная не может быть использована дважды. Мы могли бы написать:

```

# let equal c = match c with
  (x,x) -> true
  | (x,y) -> false;;

```

Characters 35-36:

This variable is bound several times in this matching

Но для этого компилятор должен уметь реализовывать тесты на равенство, что приводит к множеству проблем. Если мы ограничимся физическим значением переменных, то мы получим слишком “слабую” систему, не в состоянии, например, распознать равенство между двумя списками [1; 2]. Если же мы будем проверять на структурное равенство, то рискуем бесконечно проверять циклические структуры. Рекурсивные функции, например, это циклические структуры, но мы так же можем определить рекурсивные значения не являющиеся функциями (см. 1.2.10 стр. 61).

Универсальный образец

Символ `_` совпадает со всеми возможными значениями — он называется универсальным. Этот образец может быть использован для сопоставления сложных типов. Воспользуемся им для определения еще одной версии функции `imply`:

```
# let imply v = match v with
    (true,false) -> false
    | _          -> true;;
val imply : bool * bool -> bool = <fun>
```

Сопоставление должно обрабатывать все случаи, иначе компилятор выводит сообщение:

```
# let is_zero n = match n with 0 -> true ;;
Characters 17-40:
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
1
val is_zero : int -> bool = <fun>
```

В случае если аргумент не равен 0, функция не знает какое значение должно быть возвращено. Для исправления добавим универсальный образец:

```
# let is_zero n = match n with
    0 -> true
    | _ -> false ;;
val is_zero : int -> bool = <fun>
```

Если во время выполнения ни один образец не выбран, возбуждается исключение:

```
# let f x = match x with 1 -> 3 ;;
Characters 11-30:
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
0
val f : int -> int = <fun>
```

```
# f 1 ;;
- : int = 3
```

```
# f 4 ;;
```

```
Uncaught exception: Match_failure(, 11, 30)
```

Исключение `Match_Failure` возбуждено при вызове `f 4` и если оно не обработано, то вычисление останавливается (см. 1.3 стр. 63).

Комбинация образцов

Комбинация образцов позволяет получить новый образец, который может разложить значение в соответствии с одним или другим из начальных шаблонов.

Синтаксис

$p_1 \mid \dots \mid p_n$

Эта форма создает новый образец из комбинации мотивов p_1, \dots, p_n , с единственным ограничением — каждый из этих образцов должен содержать константные значения либо универсальный образец.

Следующий пример показывает как проверить является ли входной символ гласной буквой.

```
# let est_une_voyelle c = match c with
  'a' | 'e' | 'i' | 'o' | 'u' | 'y' -> true
  | _ -> false ;;
val est_une_voyelle : char -> bool = <fun>
# est_une_voyelle 'i' ;;
- : bool = true
# est_une_voyelle 'j' ;;
- : bool = false
```

Параметризованное сопоставление

Параметризованное сопоставление используется в основном для определения функций выбора. Для облегчения записи подобных определений, конструктор `function` разрешает следующее сопоставление одного параметра.

Синтаксис

```
function | p1 -> expr1
         | p_2 -> expr2
         :
         | p_n -> exprn
```

Вертикальная черта перед первым образцом не обязательна. Каждый раз при определении функции, мы используем сопоставление с образцом. Действительно, конструкция `function x -> expression` является определением сопоставления по уникальному образцу одной переменной. Можно использовать эту особенность в простых шаблонах:

```
# let f = function (x,y) -> 2*x + 3*y + 4 ;;
val f : int * int -> int = <fun>
```

Форма

```
function p1 -> expr1 | ... | pn -> exprn
```

эквивалентна

```
function expr -> match expr with p1 -> expr1 | ... | pn -> exprn
```

Используя схожесть определения на (см. 1.1.4 стр. 30), мы бы написали:

```
# let f (x,y) = 2*x + 3*y + 4 ;;
val f : int * int -> int = <fun>
```

Но подобная запись возможна лишь в случае если фильтруемое значение принадлежит к типу с единственным конструктором, иначе сопоставление не является исчерпывающим:

```
# let is_zero 0 = true ;;
Characters 13-21:
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
1
val is_zero : int -> bool = <fun>
```

Присвоение имени фильтруемому значению

Иногда при сопоставлении с образцом, бывает необходимо дать имя всему образцу или лишь его части. Следующая синтаксическая форма вводит ключевое слово **as**, которое ассоциирует имя образцу.

Синтаксис

```
(p as nom)
```

Это бывает полезно когда необходимо разбить значение на части, сохраняя при этом его целостность. В следующем примере функция возвращает наименьшее рациональное число из пары. Каждое рациональное число представлено числителем и знаменателем.

```
# let min_rat cr = match cr with
  ((_,0),c2) -> c2
  |(c1,(_,0)) -> c1
  |(((n1,d1) as r1), ((n2,d2) as r2)) ->
    if (n1 * d2) < (n2 * d1) then r1 else r2;;
val min_rat : (int * int) * (int * int) -> int * int = <fun>
```

Для сравнения двух рациональных чисел, необходимо разбить их для именования числителя и знаменателя (**n1**, **n2**, **d1** и **d2**), но так же для

собираения в одно целое начальные пары (**r1** или **r2**). Конструкция позволяет дать имена частям значения — это избавляет от реконструкции рационального числа при возвращении результата.

Сопоставление с ограничением

Сопоставление с ограничением соответствует вычислению условного выражения сразу после сопоставления с образцом. Если это выражение возвращает значение **true**, то выражение соответствующее этому образцу будет вычислено, иначе сопоставление будет продолжаться дальше.

Синтаксис:

```
match expr with
:
| pi when condi -> expri
:
```

В следующем примере используется два ограничения для проверки равенства двух рациональных чисел.

```
# let eq_rat cr = match cr with
  ((_,0),(_,0)) -> true
  | ((_,0),_) -> false
  | (_,(_,0)) -> false
  | ((n1,1), (n2,1)) when n1 = n2 -> true
  | ((n1,d1), (n2,d2)) when ((n1 * d2) = (n2 * d1)) -> true
  | _ -> false;;
val eq_rat : (int * int) * (int * int) -> bool = <fun>
```

Если при фильтрации четвертого образца ограничение не выполнится, то сопоставление продолжится по пятому образцу.

Warning

При проверке исчерпываемости сопоставления Objective CAML предполагает что условное выражение может быть ложным. В следствии, верификатор не учитывает этот образец, так как не возможно знать до выполнения сработает ограничение или нет. Исчерпываемость следующего фильтра не может быть определена.

```
# let f = function x when x = x -> true;;
```

Characters 10-40:

Warning: this pattern-matching is not exhaustive.

Here is an example of a value that is not matched:

```
val f : 'a -> bool = <fun>
```

Образец интервала символов

При сопоставлении символов, неудобно описывать комбинацию всех образцов, которая соответствует интервалу символов. Действительно, для того чтобы проверить является ли символ буквой, необходимо написать как минимум 26 образцов и скомбинировать их. Для символьного типа разрешается запись следующего шаблона:

Синтаксис

```
'c1' .. 'cn'
```

что соответствует 'c₁' | 'c₂' | ... | 'c_n'

К примеру образец

```
'0' .. '9'
```

соответствует образу

```
'0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'.
```

Первая форма быстрее и проще воспринимается при чтении.

Warning

Это особенность является расширением языка и может измениться в следующих версиях.

Используя комбинированные образцы и интервалы, напомним функцию по определению символа по нескольким критериям.

```
# let char_discriminate c = match c with
  | 'a' | 'e' | 'i' | 'o' | 'u' | 'y'
  | 'A' | 'E' | 'I' | 'O' | 'U' | 'Y' -> "Voyelle"
  | 'a'..'z' | 'A'..'Z' -> "Consonne"
  | '0'..'9' -> "Chiffre"
  | _ -> "Autre" ;;
val char_discriminate : char -> string = <fun>
```

Заметим, что порядок образцов важен, второе множество содержит первое, но оно проверяется только после неудачи первого теста.

Образцы списков

Как мы уже видели список может быть:

- либо пуст (в форме `[]`)
- либо состоять из первого элемента (заголовок) и под-списка (хвост).
В этом случае он имеет форму `t::q`.

Обе эти записи могут быть использованы в образце при фильтрации списка.

```
# let rec size x = match x with
| [] -> 0
| _::queue_x -> 1 + (size queue_x) ;;
val size : 'a list -> int = <fun>
# size [];
- : int = 0
# size [7;9;2;6];;
- : int = 4
```

Перепишем ранее показанный пример (см. 1.1.6 стр. 38) используя сопоставление с образцом для итерации списков.

```
# let rec fold_left f a = function
| [] -> a
| tete::queue -> fold_left f (f a tete) queue ;;
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
# fold_left (+) 0 [8;4;10];;
- : int = 22
```

Объявление значения через сопоставлением с образцом

Объявление значений само по себе использует сопоставление. `let x=18` сопоставляет образцы `x` со значением 18. Любой образец принимается как фильтр объявления, переменные образца ассоциируются со значениями которые они сопоставляют.

```
# let (a,b,c) = (1, true, 'A');;
val a : int = 1
val b : bool = true
val c : char = 'A'
# let (d,c) = 8, 3 in d + c;;
- : int = 11
```

Видимость переменных фильтра та же самая что у локальных переменных. Здесь с ассоциировано с 'A'.

```
# a + (int_of_char c);;
- : int = 66
```


Как и любой фильтр, определение значения может быть не исчерпывающим.

```
# let [x;y;z] = [1;2;3];;
```

Characters 5-12:

Warning: this pattern-matching is not exhaustive.

Here is an example of a value that is not matched:

```
[]  
val x : int = 1  
val y : int = 2  
val z : int = 3  
# let [x;y;z] = [1;2;3;4];;
```

Characters 4-11:

Warning: this pattern-matching is not exhaustive.

Here is an example of a value that is not matched:

```
[]  
Uncaught exception: Match_failure(, 4, 11)
```

Принимается любой образец, включая конструктор, универсальный и комбинированный.

```
# let tete :: 2 :: _ = [1; 2; 3] ;;
```

Characters 5-19:

Warning: this pattern-matching is not exhaustive.

Here is an example of a value that is not matched:

```
[]  
val tete : int = 1  
# let _ = 3. +. 0.14 in "PI" ;;  
- : string = "PI"
```

Последний пример мало интересен для функционального программирования, поскольку значение 3.14 не именовано, и соответственно будет потерянно.

1.2.2 Декларация типов

Объявление типа — это одна из других возможных фраз синтаксиса Objective CAML. Она позволяет определить новые типы соответствующие обычным структурам данных используемым в программах. Существует два больших семейства типов: тип-произведение для кортежей или записей или тип-сумма для `union`.

Синтаксис:

```
type nom = typedef ;;
```

В отличие от объявления переменных, объявление типов по умолчанию рекурсивно. То есть объявления типов, когда они скомбинированы, могут объявлять взаимно рекурсивные типы.

Синтаксис:

```
type nom1 = typedef1
and nom2 = typedef2
:
and nomn = typedefn ;;
```

Объявления типов могут быть параметризованы переменной типа. Имя переменной типа начинается всегда с апострофа (символ `'`).

Синтаксис

```
type 'a nom = typedef ;;
```

В случае когда таких переменных несколько, параметры типа декларируются как кортеж перед именем типа.

Синтаксис

```
type ('a1 ...' an) nom = typedef ;;
```

Только те параметры, которые определены в левой части декларации, могут появиться в ее правой части.

Замечание

Во время вывода на экран Objective CAML переименует параметры типа, первый в `'a`, второй в `'b` и так далее.

Мы всегда можем объявить новый тип используя ранее объявленные типы.

Синтаксис:

```
type name = type expression
```

Это полезно в том случае, когда мы хотим ограничить слишком общий тип:

```
# type 'param couple_entier = int * 'param ;;
type 'a couple_entier = int * 'a
# type couple_precis = float couple_entier ;;
type couple_precis = float couple_entier
```

Без ограничения, выводится наиболее общий тип:

```
# let x = (3, 3.14) ;;
val x : int * float = 3, 3.14
```

Но мы можем использовать ограничитель типа для того, чтобы получить желаемое имя:

```
# let (x:couple_precis) = (3, 3.14) ;;
val x : couple_precis = 3, 3.14
```

1.2.3 Записи

Запись — это кортеж, в котором каждому полю присваивается имя на подобии `record` в *Pascal* или `struct` в *C*. Запись всегда соответствует объявлению нового типа. Для определения записи необходимо указать ее имя, а так же имя и тип для каждого поля записи.

Синтаксис:

```
type nom = { nom1 : t1; ...; nomn : tn } ;;
```

Определим тип комплексного числа следующим образом.

```
# type complex = { re:float; im:float } ;;
type complex = { re: float; im: float }
```

Для того чтобы создать значение типа записи, нужно присвоить каждому полю значение (в каком угодно порядке).

Синтаксис:

```
{ nom1 = expr1; ...; nomn = exprn } ;;
```

В следующем примере мы создадим комплексное число, в котором реальная часть равна 2 и мнимая 3:

```
# let c = {re=2.;im=3.} ;;
val c : complex = {re=2; im=3}
# c = {im=3.;re=2.} ;;
- : bool = true
```

В случае если не хватает некоторых полей, произойдет следующая ошибка:

```
# let d = { im=4. } ;;
Characters 9-18:
Some labels are undefined
```

Доступ к полям возможен двумя способами: синтаксис с точкой, либо сопоставлением некоторых полей.

Синтаксис с точкой заключается в следующем:

Синтаксис:

```
expr.nom
```

Выражение `expr` должно иметь тип “запись с полем `nom`”.

Сопоставление записи позволяет получить значение связанное с определенным полем. Синтаксис образца для записи следующий.

Синтаксис

```
{ nomi = pi ; ...; nomj = pj }
```

Образцы находятся справа от символа `=` (`pi`, ..., `pj`). Необязательно перечислять все поля записи в образце.

Функция `add_complex` обращается к полям с помощью “точки”, тогда как функция `mult_complex` обращается при помощи фильтра.

```
# let add_complex c1 c2 = {re=c1.re+c2.re; im=c1.im+c2.im};;
val add_complex : complex -> complex -> complex = <fun>
# add_complex c c ;;
- : complex = {re=4; im=6}
# let mult_complex c1 c2 = match (c1,c2) with
  ({re=x1;im=y1},{re=x2;im=y2}) -> {re=x1*.x2-.y1*.y2;im=x1*.y2
    +.x2*.y1} ;;
val mult_complex : complex -> complex -> complex = <fun>
# mult_complex c c ;;
- : complex = {re=-5; im=12}
```

Преимущество записей, по сравнению с кортежами как минимум двойное:

- более информативное описание, благодаря именам полей — это в частности позволяет облегчить образцы фильтра;
- идентичное обращение по имени, порядок значения не имеет — главное указать имя.

```
# let a = (1,2,3) ;;
val a : int * int * int = 1, 2, 3
# let f tr = match tr with x,_,_ -> x ;;
val f : 'a * 'b * 'c -> 'a = <fun>
# f a ;;
- : int = 1
# type triplet = {x1:int; x2:int; x3:int} ;;
type triplet = { x1: int; x2: int; x3: int }
# let b = {x1=1; x2=2; x3=3} ;;
val b : triplet = {x1=1; x2=2; x3=3}
# let g tr = tr.x1 ;;
val g : triplet -> int = <fun>
```

```
# g b ;;
- : int = 1
```

При сопоставлении с образцом не обязательно перечислять все поля записи, тип будет вычислен по последней описанной записи, которая содержит указанные поля.

```
# let h tr = match tr with {x1=x} -> x;;
val h : triplet -> int = <fun>
# h b;;
- : int = 1
```

Существует конструкция позволяющая создать идентичную запись, с разницей в несколько полей, что удобно для записей с большим числом полей.

Синтаксис:

```
{ name with namei= expri ; ...; namej=expr _j}

# let c = {b with x1=0} ;;
val c : triplet = {x1=0; x2=2; x3=3}
```

Новая копия значения **b** отличается только лишь значением поля **x1**.

Warning

Это особенность является расширением языка и может измениться в следующих версиях.

1.2.4 Тип сумма (sum)

В отличие от кортежей или записей, которые соответствуют декартовому произведению, тип сумма соответствует объединению множеств (**union**). В один тип мы группируем несколько разных типов (например, целые числа и строки). Различные члены суммы определяются конструкторами, которые с одной стороны позволяют сконструировать значение этого типа и с другой получить доступ к компонентам этих значений при помощи сопоставления с образцом. Применить конструктор к аргументу, значит указать что возвращаемое значение будет этого нового типа.

Тип сумма описывается указыванием имени конструкторов и типа их возможного аргумента.

Синтаксис:

```
type nom = ...
| Nom1 ...
```

```
| Nomj of tj ...
| Nomk of tk * ...* tl ...;;
```

Имя конструктора это специальный идентификатор.

Замечание

Имя конструктора должна всегда начинаться с заглавной буквы.

Константный конструктор

Мы называем константным, конструктор без аргументов. Такие конструктора могут затем использоваться как значения языка, в качестве констант.

(Орел Решка)

```
# type piece = Pile | Face;;
type piece = | Pile | Face
# Pile;;
- : piece = Pile
```

Тип может быть определен подобным образом.

Конструктор с аргументами

Конструктора могут иметь аргументы, ключевое слово **of** указывает тип аргумента. Это позволяет сгруппировать под одним типом объекты разных типов, имеющих разные конструктора. Определим типы **couleur** и **carte** следующим образом.

N.B.

- тип **couleur** определяет масть карты — пика, сердце, кирпич и трефа
- тип **carte** определяет достоинство карты (король, дама...)

```
# type couleur = Pique | Coeur | Carreau | Trefle ;;
# type carte =
  Roi of couleur
  | Dame of couleur
  | Cavalier of couleur
  | Valet of couleur
  | Petite_carte of couleur * int
  | Atout of int
  | Excuse ;;
```

Создание значения типа `carte` получается применением конструктора к значению с нужным типом.

```
# Roi Pique ;;
- : carte = Roi Pique
# Petite_carte(Coeur, 10) ;;
- : carte = Petite_carte (Coeur, 10)
# Atout 21 ;;
- : carte = Atout 21
```

Следующая функция, `toutes_les_cartes`, создает список всех карт цвета указанного аргументом.

```
# let rec interval a b =
  if a = b then [b] else a::(interval (a+1) b) ;;
val interval : int -> int -> int list = <fun>
# let toutes_les_cartes s =
  let les_figures = [ Valet s; Cavalier s; Dame s; Roi s ]
  and les_autres = List.map (function n -> Petite_carte(s,n)) (
    interval 1 10)
  in les_figures @ les_autres ;;
val toutes_les_cartes : couleur -> carte list = <fun>
# toutes_les_cartes Coeur ;;
- : carte list =
[Valet Coeur; Cavalier Coeur; Dame Coeur; Roi Coeur; Petite_carte (
  Coeur, 1);
 Petite_carte (Coeur, 2); Petite_carte (Coeur, 3); Petite_carte (Coeur,
  ...) ; ...]
```

Для манипуляции значений типа `сумма`, мы используем сопоставление с образцом. В следующем примере опишем функцию преобразующую значения типа `couleur` и типа `carte` в строку (тип `string`):

```
# let string_of_couleur = function
  Pique -> "pique"
  | Carreau -> "carreau"
  | Coeur -> "coeur"
  | Trefle -> "trefle" ;;
val string_of_couleur : couleur -> string = <fun>
# let string_of_carte = function
  Roi c -> "roi de " ^ (string_of_couleur c)
  | Dame c -> "dame de " ^ (string_of_couleur c)
  | Valet c -> "valet de " ^ (string_of_couleur c)
  | Cavalier c -> "cavalier de " ^ (string_of_couleur c)
```

```

    | Petite_carte (c, n) -> (string_of_int n) ^ " de " ^ (
        string_of_couleur c)
    | Atout n      -> (string_of_int n) ^ " d'atout"
    | Excuse       -> "excuse" ;;
val string_of_carte : carte -> string = <fun>

```

Конструктор `Petite_carte` бинарный, для сопоставления такого значения мы должны дать имя обоим компонентам.

```

# let est_petite_carte c = match c with
    Petite_carte v -> true
    | _ -> false;;
%Characters 45–59:%
%The constructor Petite_carte expects 2 argument(s),%
%but is here applied to 1 argument(s)%

```

Чтобы не давать имя каждой компоненте конструктора, объявим его с одним аргументом, заключив в скобки тип ассоциированного кортежа. Сопоставление двух следующих конструкторов отличается.

```

# type t =
    C of int * bool
    | D of (int * bool) ;;
# let acces v = match v with
    C (i, b) -> i,b
    | D x -> x;;
val acces : t -> int * bool = <fun>

```

1.2.5 Рекурсивный тип

Определение рекурсивного типа необходимо в любом языке программирования, с его помощью мы можем определить такие типы как список, стек, дерево и так далее. В отличие от объявления `let`, `type` всегда рекурсивный в Objective CAML.

Типу список, определенному в Objective CAML, необходим один единственный аргумент. Для того чтобы хранить значения двух разных типов, как например `int` или `char` мы напишем:

```

# type int_or_char_list =
    Nil
    | Int_cons of int * int_or_char_list
    | Char_cons of char * int_or_char_list ;;

```



```
# let l1 = Char_cons ( '=', Int_cons(5, Nil) )
  in      Int_cons ( 2, Char_cons ( '+', Int_cons(3, l1) ) )  ;;
- : int_or_char_list =
Int_cons (2, Char_cons ( '+', Int_cons (3, Char_cons ( '=', Int_cons (...))
)))
```

1.2.6 Параметризованный тип

Мы также можем определить тип с параметрами, что позволит нам обобщить предыдущий список для двух любых типов.

```
# type ('a, 'b) list2 =
  Nil
  | Acons of 'a * ('a, 'b) list2
  | Bcons of 'b * ('a, 'b) list2 ;;

# Acons(2, Bcons('+', Acons(3, Bcons('=', Acons(5, Nil))))) ;;
- : (int, char) list2 =
Acons (2, Bcons ( '+', Acons (3, Bcons ( '=', Acons (...)))))
```

Естественно, оба параметра `'a` и `'b` могут быть одного типа:

```
# Acons(1, Bcons(2, Acons(3, Bcons(4, Nil))))) ;;
- : (int, int) list2 = Acons (1, Bcons (2, Acons (3, Bcons (4, Nil)))))
```

Как в предыдущем примере, мы можем использовать тип `list2` чтобы пометить четные и нечетные числа. Что бы создать обычный список, достаточно извлечь под-список четных чисел.

```
# let rec extract_odd = function
  Nil -> []
  | Acons(_, x) -> extract_odd x
  | Bcons(n, x) -> n::(extract_odd x) ;;
val extract_odd : ('a, 'b) list2 -> 'b list = <fun>
```

Определение этой функции ничего не говорит о свойстве значений хранящихся в структуре, по этой причине ее тип параметризован.

1.2.7 Видимость описания

На имена конструкторов распространяются те же правила, что и на глобальные объявления. Переопределение скрывает своего предшественника. Скрытые значения всегда существуют, они никуда не делись. Однако

цикл не сможет различить эти оба типа, поэтому мы получим не совсем понятное сообщение об ошибке.

В следующем примере, константный конструктор `Nil` типа `int_of_char` скрывается объявлением конструктора `('a, 'b) list2`.

```
# Int_cons(0, Nil) ;;
Characters 13-16:
This expression has type ('a, 'b) list2 but is here used with type
int_or_char_list
```

Во втором примере мы получим совсем бестолковое сообщение, по крайней мере на первый взгляд. Пусть мы имеем следующую программу:

```
# type t1 = Vide | Plein;;
type t1 = | Vide | Plein
# let vide_t1 x = match x with
  Vide -> true
  | Plein -> false ;;
val vide_t1 : t1 -> bool = <fun>
# vide_t1 Vide;;
- : bool = true
```

Затем переопределим тип `t1`:

```
# type t1 = { u : int; v : int } ;;
type t1 = { u: int; v: int }
# let y = { u=2; v=3 } ;;
val y : t1 = {u=2; v=3}
```

Теперь если мы применим функцию `empty_t1` к значению с новым типом `t1`, то получим следующее сообщение.

```
# empty_t1 y;;
Characters 9-10:
This expression has type t1 but is here used with type t1
```

Первое упоминание типа `t1` соответствует ранее определенному типу, тогда как второе — последнему.

1.2.8 Функциональные типы

Тип аргумента конструктора может быть каким угодно, и в частности он может быть функциональным. Следующий тип создает список состоящий из функциональных значений, кроме последнего элемента.

```
# type 'a listf =
```

```

    Val of 'a
  | Fun of ('a -> 'a) * 'a listf ;;
type 'a listf = | Val of 'a | Fun of ('a -> 'a) * 'a listf

```

Поскольку функциональные значения входят во множество значений которые обрабатываются языком, то мы можем создать значения типа `listf`:

```

# let huit_div = (/) 8 ;;
val huit_div : int -> int = <fun>
# let gl = Fun (succ, (Fun (huit_div, Val 4))) ;;
val gl : int listf = Fun (<fun>, Fun (<fun>, Val 4))

```

функция сопоставляющая подобные значения:

```

# let rec compute = function
    Val v -> v
  | Fun(f, x) -> f (compute x) ;;
val compute : 'a listf -> 'a = <fun>
# compute gl;;
- : int = 3

```

1.2.9 Пример: реализация деревьев

Деревья — одна из часто встречающихся структур в программировании. Рекурсивные типы позволяют с легкостью определять подобные вещи. В этой части мы приведем два примера с такими структурами.

Бинарные деревья Определим дерево, в котором узлы обозначены значениями одного и того же типа:

```

# type 'a arbre_bin =
    Empty
  | Node of 'a arbre_bin * 'a * 'a arbre_bin ;;

```

Этой структурой мы воспользуемся в программе сортировки бинарных деревьев. Бинарное дерево имеет следующее свойство: значение левого дочернего узла меньше корневого и всех значений правых дочерних узлов.

Извлечем значения узлов в виде отсортированного списка при помощи инфиксного просмотра следующей функцией:

```

# let rec list_of_arbre = function
    Empty -> []

```

```
| Node(fg, r, fd) -> (list_of_arbre fg) @ (r :: (list_of_arbre fd)) ;;
val list_of_arbre : 'a arbre_bin -> 'a list = <fun>
```

Для того чтобы получить из списка бинарное дерево, определим функцию:

```
# let rec ajout x = function
  Empty -> Node(Empty, x, Empty)
  | Node(fg, r, fd) -> if x < r then Node(ajout x fg, r, fd)
                        else Node(fg, r, ajout x fd) ;;
val ajout : 'a -> 'a arbre_bin -> 'a arbre_bin = <fun>
```

Функция трансформирующая список в бинарное дерево может быть получена использованием функции `ajout`

```
# let rec arbre_of_list = function
  [] -> Empty
  | t::q -> ajout t (arbre_of_list q) ;;
val arbre_of_list : 'a list -> 'a arbre_bin = <fun>
```

Тогда функция сортировки это всего-навсего композиция функций `arbre_of_list` и `list_of_arbre`.

```
# let tri x = list_of_arbre (arbre_of_list x) ;;
val tri : 'a list -> 'a list = <fun>
# tri [5; 8; 2; 7; 1; 0; 3; 6; 9; 4] ;;
- : int list = [0; 1; 2; 3; 4; 5; 6; 7; 8; 9]
```

3

Общие планарные деревья Воспользуемся функцией определенной в модуле `List` (см. 7 стр. 235):

- `List.map`: которая применяет одну функцию ко всем элементам списка и возвращает список результатов.
- `List.fold_left`: эквивалент функции `fold_left` определенной на стр. 40
- `List.exists` применяет функцию булевого значения ко всем элементам и если одно из применений возвращает `true`, то результат `true`, иначе `false`

Общее планарное дерево — это дерево в котором нет ограничение на число дочерних узлов: каждому узлу ассоциируем список дочерних узлов и длина этого списка не ограничена.

```
# type 'a arbre =
  Empty
  | Node of 'a * 'a arbre list ;;
```

Пустое дерево представлено значением `Empty`, лист — это узел без дочерних узлов формы `Node(x, [])` или `Node(x, [Empty; Empty; ...])`. Таким образом достаточно просто написать функции, манипулирующие подобными деревьями такие как принадлежность элемента дереву или вычисление высоты дерева.

Для проверки принадлежности элемента `e` воспользуемся следующим алгоритмом: если дерево пустое, тогда элемент `e` не принадлежит этому дереву, в противном случае `e` принадлежит дереву только если он равен значению корня дерева или принадлежит дочерним узлам.

```
# let rec appartient e = function
  Empty -> false
  | Node(v, fs) -> (e=v) or (List.exists (appartient e) fs) ;;
val appartient : 'a -> 'a arbre -> bool = <fun>
```

Для вычисления высоты дерева, напомним следующую функцию: высота пустого дерева равна 0, в противном случае его высота равна высоте самого большого под-дерева плюс 1.

```
# let rec hauteur =
  let max_list l = List.fold_left max 0 l in
  function
    Empty -> 0
    | Node (_, fs) -> 1 + (max_list (List.map hauteur fs)) ;;
val hauteur : 'a arbre -> int = <fun>
```

1.2.10 Не функциональные рекурсивные значения

Рекурсивное объявление не функциональных значений позволяет конструировать циклические структуры данных.

Следующее объявление создает циклический список одного элемента.

```
# let rec l = 1::l ;;
val l : int list =
  [1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; ...]
```

Применение рекурсивной функции к подобному списку приведет к переполнению памяти.

```
# size l ;;
Stack overflow during evaluation (looping recursion?).
```

Структурное равенство таких списков может быть проверенно лишь в случае, когда физическое равенство верно.

```
# l=l ;;
- : bool = true
```

В случае, если мы определим такой же новый список, не стоит использовать проверку на структурное равенство, под угрозой заикливания программы. Таким образом следующая инструкция не рекомендуется.

```
let rec l2 = 1::l2 in l=l2 ;;
```

Однако, физическое равенство остается возможным.

```
# let rec l2 = 1::l2 in l==l2 ;;
- : bool = false
```

Предикат == сравнивает непосредственно значение или разделения структурного объекта (одним словом равенство по адресу). Мы воспользуемся этим тестом для того чтобы при просмотре списка, не проверять уже просмотренные под-списки. Сначала, определим функцию `memq` которая проверяет присутствие элемента в списке при помощи физического равенства. В это же время функция `mem` проверяет равенство структурное. Обе функции принадлежат модулю `List`.

```
# let rec memq a l = match l with
  [] -> false
  | b::l -> (a==b) or (memq a l) ;;
val memq : 'a -> 'a list -> bool = <fun>
```

Функция вычисляющая размера списка определена при помощи списка уже проверенных списков и останавливается при встрече списка второй раз.

```
# let special_size l =
  let rec size_aux previous l = match l with
    [] -> 0
    | _::l1 -> if memq l previous then 0
               else 1 + (size_aux (l::previous) l1)
  in size_aux [] l ;;
val special_size : 'a list -> int = <fun>
# special_size [1;2;3;4] ;;
- : int = 4
# special_size l ;;
- : int = 1
# let rec l1 = 1::2::l2 and l2 = 1::2::l1 in special_size l1 ;;
- : int = 4
```

1.3 Типизация, область определения и исключения

Вычисленный тип функции принадлежит подмножеству множества в котором она определена. Если входной аргумент функции типа `int`, это не значит что она сможет быть выполнена для любого переданного ей целого числа. Таких случаях мы используем механизм исключений Objective CAML. Запуск исключения провоцирует остановку вычислений, которое может быть выявлено и обработано. Для этого необходимо установить обработчик исключения, до того как исключение может быть возбуждено.

1.3.1 Частичные функции и исключения

Область определения функции соответствует множеству значений которыми функция манипулирует. Существует множество математических частичных функций, таких как например, деление или натуральный логарифм. Проблемы возникают в том случае, когда функция манипулирует более сложными структурами данных. Действительно, каков будет результат функции определяющей первый элемент списка, если список пуст. Аналогично, вычисление факториала для отрицательного числа приводит к бесконечному вычислению.

Определенное число исключительных ситуаций может произойти во время выполнения программы, как например деление на ноль. Попытка деления на ноль может привести в лучшем случае к остановке программы и в худшем к неправильному (некоэррантному) состоянию машины. Безопасность языка программирования заключается в гарантии того, что такие случаи не возникнут. Исключение есть один из способов такой гарантии.

Деление 1 на 0 провоцирует запуск специального исключения:

```
# 1/0;;  
Uncaught exception: Division_by_zero
```

Сообщение `Uncaught exception: Division_by_zero` указывает во первых на то, что исключение `Division_by_zero` возбуждено и во вторых, что оно не было обработано.

Часто, тип функции не соответствует области ее определения в том случае когда сопоставление с образцом не является полным, то есть когда он не отслеживает все возможные случаи. Чтобы избежать такой ситуации, Objective CAML выводит следующее сообщение.

```
# let tete l = match l with t::q -> t ;;
Characters 14-36:
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
val tete : 'a list -> 'a = <fun>
```

Однако, если программист все таки настаивает на своей версии, то Objective CAML воспользуется механизмом исключения в случае неправильного вызова частичной функции.

```
# tete [] ;;
Uncaught exception: Match_failure("", 14, 36)
```

Мы уже встречали другое исключение определенное в Objective CAML: **Failure**, с входным аргументом типа **string**. Запустить это исключение можно функцией **failwith**. Воспользуемся этим и дополним определение функции **tete**.

```
# let tete = function
  [] -> failwith "List is empty"
  | h::t -> h;;
val tete : 'a list -> 'a = <fun>
# tete [] ;;
Uncaught exception: Failure("List is empty")
```

1.3.2 Определения исключения

В Objective CAML, тип исключений — **exn**. Это особенный тип — *расширяемая* сумма: мы можем увеличить множество значений этого типа объявляя новые конструкторы. Такая особенность позволяет программисту определять свои собственные исключения.

Синтаксис объявления следующий:

Синтаксис:

```
exception Nom ;;
```

Синтаксис:

```
exception Nom of t ;;
```

Вот несколько примеров объявления исключений.

```
# exception A_MOI;;
exception A_MOI
# A_MOI;;
```



```

- : exn = A_MOI
# exception Depth of int;;
exception Depth of int
# Depth 4;;
- : exn = Depth(4)

```

Имена исключений являются конструкторами — это значит они должны начинаться с заглавной буквы.

```

# exception minuscule ;;
Characters 11-20:
Syntax error

```

Warning

Исключения мономорфны, при объявлении типа аргумента мы не можем указать параметризующий тип.

```

# exception Value of 'a ;;
Characters 20-22:
Unbound type parameter 'a

```

Полиморфное исключение позволило бы определение функций, возвращающих результат любого типа. Подобный пример мы рассмотрим далее на стр. 67.

1.3.3 Возбуждение исключения

raise — функциональный примитив языка Objective CAML, ее входной аргумент есть исключение, а возвращает он полиморфный тип.

```

# raise ;;
- : exn -> 'a = <fun>
# raise A_MOI;;
Uncaught exception: A_MOI
# 1+(raise IS_Mine);;
Uncaught exception: A_MOI
# raise (Depth 4);;
Uncaught exception: Depth(4)

```

В Objective CAML нет возможности написать функцию **raise**, она должна быть определена системой.

1.3.4 Перехват исключения

Весь интерес возбуждения исключений содержится в возможности их обработать и изменить выполнение программы в зависимости от этого исключения. В этом случае, порядок вычисления выражения имеет значение для того чтобы определить какое исключение было запущено. Таким образом мы выходим из рамок чисто функционального программирования, потому как порядок вычисления аргументов может изменить результат. Об этом мы поговорим в следующей главе (стр. 94)

Следующая конструкция, вычисляющая значение какого-то выражения, отлавливает исключение, возбужденного во время этого вычисления.

Синтаксис:

```
try expr with
| p_1 -> expr_1
:
| pn -> expr_n
```

Если вычисление **expr** не возбудит исключения, то тип результата будет типом подсчитываемым этим выражением. Иначе, в приведенном выше сопоставлении будет искаться ответвление соответствующее этому исключению и будет возвращено значение следующего за ним выражения.

Если ни одно ответвление не соответствует исключению, то оно будет переправлено до следующего обработчика **try-with** установленного в программе. Таким образом сопоставление исключения всегда исчерпывающий. Подразумевается что последний фильтр **|e->raise e**. Если ни одного обработчика не найдено в программе, система сама займется обработкой этого исключения и остановит программу с выводом сообщения об ошибке.

Важно понимать разницу между вычислением исключения (то есть значение типа **exn**) и его обработкой, которое приостанавливает вычисления. Исключение, как и любой другой тип, может быть возвращено функцией.

```
# let rendre x = Failure x ;;
val rendre : string -> exn = <fun>
# rendre "test" ;;
- : exn = Failure("test")
# let declencher x = raise (Failure x) ;;
val declencher : string -> 'a = <fun>
# declencher "test" ;;
Uncaught exception: Failure("test")
```

Как вы наверно заметили, функция `declencher` (*запустить*) не возвращает значение типа `exp`, тогда как `rendre` (*вернуть*) возвращает.

Вычисления с исключениями

Кроме обработки не желаемых значений, исключения можно использовать для оптимизации. Следующий пример реализует произведение всех элементов списка целых чисел. Мы воспользуемся исключением для остановки просмотра списка если обнаружим 0, который мы вернем как результат функции.

```
# exception Found_zero ;;
exception Found_zero
# let rec mult_rec l = match l with
  | [] -> 1
  | 0 :: _ -> raise Found_zero
  | n :: x -> n * (mult_rec x) ;;
val mult_rec : int list -> int = <fun>
# let mult_list l =
  try mult_rec l with Found_zero -> 0 ;;
val mult_list : int list -> int = <fun>
# mult_list [1;2;3;0;5;6] ;;
- : int = 0
```

Оставшиеся вычисления, то есть произведения элементов после встреченного нуля, не выполняются. После встречи `raise`, вычисление вновь продолжится с `with`.

1.4 Полиморфизм и значения возвращаемые функциями

Полиморфизм Objective CAML позволяет определить функции, тип возвращаемого выражения которых конкретно не определен. Например:

```
# let id x = x ;;
val id : 'a -> 'a = <fun>
```

Однако, возвращаемый тип `id` зависит от типа аргумента. Таким образом, если мы применим функцию `id` к какому-нибудь аргументу, то механизм вывода (определения) типа сможет реализовать переменную типа `'a`. Для каждого частного использования тип функции `id` может быть определен.

Иначе, использование жесткой статической типизации, которая обеспечивает правильность выполнения, не будет иметь смысла. Функция с полностью неопределенным типом как `'a->'b`, разрешит какое попало преобразование типа, что привело бы к ошибке выполнения, так как физическое представление значений не одно и то же.

Очевидное противоречие

Мы можем определить функции, которые возвращают значение, тип которого не соответствует типу аргументов. Рассмотрим несколько примеров и проанализируем почему это не противоречит жесткой статической типизации.

Вот первый пример:

```
# let f x = [] ;;
val f : 'a -> 'b list = <fun>
```

Эта функция конструирует полиморфные значения из любого типа.

```
# f () ;;
- : '_a list = []
# f "n'importe quoi" ;;
- : '_a list = []
```

Однако, полученное значение не совсем является не определенным — подразумевается список. Таким образом она не может использоваться где попало.

Вот три примера функций типа `'a -> 'b`:

```
# let rec f1 x = f1 x ;;
val f1 : 'a -> 'b = <fun>
# let f2 x = failwith "n'importe quoi" ;;
val f2 : 'a -> 'b = <fun>
# let f3 x = List.hd [] ;;
val f3 : 'a -> 'b = <fun>
```

Эти функции не являются “опасными” в отношении правильности выполнения программы, так как их невозможно использовать для конструкции значений: первая заиклиивается, две последние запускают исключение, которые останавливают выполнение программы.

Чтобы не было возможности определить функции типа `'a -> 'b`, новые конструкторы исключений не должны иметь аргументы типы которых содержат переменную.

Если бы могли объявить полиморфное исключение `Poly_exn` типа `'a -> exn`, то тогда следующая функция была бы возможна:

```
let f = function
  0 -> raise (Poly_exn false)
  | n -> n+1 ;;
```

Таким образом, тип функции `f` это `int->int` и тип `Poly_exn` это `'a -> exn`, теперь напишем следующее:

```
let g n = try f n with Poly_exn x -> x+1 ;;
```

Это правильно типизированная функция (поскольку аргумент `Poly_exn` может быть каким угодно) и вызов `g 0` попытается сложить целое с булевым значением!

1.5 Калькулятор

Для того чтобы понять как организуются программы на Objective CAML, необходимо самим реализовать такую программу. Напишем калькулятор манипулирующий целыми числами при помощи 4 стандартных операций.

Для начала, определим тип `key`, для представления кнопки калькулятора. Всего таких кнопок будет 15: по одной для каждой операции, для каждой цифры и для кнопки равно.

```
# type key = Plus | Minus | Times | Div | Equals | Digit of int ;;
```

Заметим, что цифровые кнопки сгруппированы под одним конструктором `Digit` с аргументом целого типа. Таким образом, некоторые значения типа `key` не являются на самом деле кнопкой. Например, `(Digit 32)` есть значение типа `key`, но не представляет ни одну кнопку калькулятора.

Напишем функцию `valid` которая проверяет входной аргумент на принадлежность к типу `key`. Тип функции `key->bool`.

На первый этапе определим функцию проверяющую что число принадлежит интервалу 0-9. Объявим ее с именем `is_digit`.

```
# let is_digit = function x -> (x>=0) && (x<=9) ;;
val is_digit : int -> bool = <fun>
```

Теперь функция `valid` фильтрующая свой аргумент тип которого `key`:

```
# let valid ky = match ky with
  Digit n -> is_digit n
  | _ -> true ;;
val valid : key -> bool = <fun>
```

Первый фильтр применяется в случае если входной аргумент построен конструктором `Digit`, в этом случае он проверяется функцией `is_digit`. Второй фильтр применяется в любом другом случае. Напомним, что благодаря фильтру, тип фильтруемого значения обязательно будет `key`.

Перед тем как начать реализовывать алгоритм калькулятора, определим модель формально описывающую реакцию на нажатие кнопок. Мы подразумеваем что калькулятор располагает памятью в которой хранится результат последней операции, последняя нажатая кнопка, последний использованный оператор и число выведенное на экран. Все эти 4 значения назовем состоянием калькулятора, оно изменяется каждый раз при нажатии на одну из кнопок. Это изменение называется транзицией, а теория управляющая подобным механизмом есть теория автоматов.

Состояние представлено следующим типом:

```
# type state = {
  lcd : int; (* last computation done *)
  lka : key; (* last key activated *)
  loa : key; (* last operator activated *)
  vpr : int (* value printed *)
} ;;
```

В таблице 1.5 приведен пример состояний калькулятора.

	state	key
	(0,=,=,0)	3
3/4	(0,3,=,3)	+
3/4	(3,+,+,3)	2
3/4	(3,2,+,2)	1
3/4	(3,1,+,21)	x
3/4	(24,*,*,24)	2
3/4	(24,2,*,2)	=
3/4	(48,=,=,48)	

Таблица 1.5: Транзиции для $3+21*2=$

Нам нужна функция `evaluate` с тремя аргументами: двумя операндами и оператором, она возвращает результат операции. Для этого функция фильтрует входной аргумент типа `key`:

```
# let evaluate x y ky = match ky with
  Plus   -> x + y
| Minus  -> x - y
```

```

| Times -> x * y
| Div   -> x / y
| Equals -> y
| Digit _ -> failwith "evaluate : no op";;
val evaluate : int -> int -> key -> int = <fun>

```

Дадим определение транзиций, перечисляя все возможные случаи. Предположим, что текущее состояние есть квадри-плет (a, b, A, d) :

- кнопка с цифрой x нажата, есть два возможных случая:
 - * предыдущая нажатая кнопка тоже цифра, значит пользователь продолжает набирать число и необходимо добавить x к значению выводимому на экран то есть изменить его на $d*10+x$. Новое состояние будет: $(a, (Digit\ x), A, d*10+x)$
 - * предыдущая нажатая кнопка не цифра, тогда мы только начинаем писать новое число. Новое состояние: $(a, (Digit\ x), A, x)$
- кнопка с оператором B была нажата, значит второй операнд был полностью введен и теперь мы хотим "что-то" сделать с двумя операндами. Для этого сохранена последняя операция (A) . Новое состояние: $(A\ d, B, B, a\ A\ d)$

Для того чтобы написать функцию `transition`, достаточно дословно перевести в Objective CAML предыдущее описание при помощи сопоставления входного аргумента. Кнопку с двумя значениями обработаем локальной функцией `digit_transition` при помощи сопоставления.

```

# let transition st ky =
  let digit_transition n = function
    Digit _ -> { st with lka=ky; vpr=st.vpr*10+n }
    | _      -> { st with lka=ky; vpr=n }
  in
    match ky with
      Digit p -> digit_transition p st.lka
      | _      -> let res = evaluate st.lcd st.vpr st.loa
                    in { lcd=res; lka=ky; loa=ky; vpr=res } ;;
val transition : state -> key -> state = <fun>

```

Эта функция из `state` и `key` считает новое состояние `state`.

Протестируем теперь программу:

```

# let initial_state = { lcd=0; lka=Equals; loa=Equals; vpr=0 } ;;
val initial_state : state = {lcd=0; lka=Equals; loa=Equals; vpr=0}

```

```

# let state2 = transition initial_state (Digit 3) ;;
val state2 : state = {lcd=0; lka=Digit 3; loa=Equals; vpr=3}
# let state3 = transition state2 Plus ;;
val state3 : state = {lcd=3; lka=Plus; loa=Plus; vpr=3}
# let state4 = transition state3 (Digit 2) ;;
val state4 : state = {lcd=3; lka=Digit 2; loa=Plus; vpr=2}
# let state5 = transition state4 (Digit 1) ;;
val state5 : state = {lcd=3; lka=Digit 1; loa=Plus; vpr=21}
# let state6 = transition state5 Times ;;
val state6 : state = {lcd=24; lka=Times; loa=Times; vpr=24}
# let state7 = transition state6 (Digit 2) ;;
val state7 : state = {lcd=24; lka=Digit 2; loa=Times; vpr=2}
# let state8 = transition state7 Equals ;;
val state8 : state = {lcd=48; lka=Equals; loa=Equals; vpr=48}

```

Мы можем сделать тоже самое, передав список транзаций.

```

# let transition_list st ls = List.fold_left transition st ls ;;
val transition_list : state -> key list -> state = <fun>
# let example = [ Digit 3; Plus; Digit 2; Digit 1; Times; Digit 2; Equals
  ]
  in transition_list initial_state example ;;
- : state = {lcd=48; lka=Equals; loa=Equals; vpr=48}

```

1.6 Резюме

В этой главе мы изучили основы функционального программирования и параметризованный полиморфизм, что вместе являются основными характеристиками Objective CAML. Также мы описали синтаксис функциональной части ядра и типов. Это дало нам возможность написать первые программы. Так же мы подчеркнули глубокую разницу между типом функции и областью ее применения. Механизм исключений позволяет решить эту проблему, и в то же время вводит новый стиль программирования, где мы явно указываем манеру вычисления.

Глава 2

Императивное программирование

Введение

В отличие от функционального программирования, где значение вычисляется посредством применения функции к аргументам, не заботясь о том как это происходит, императивное программирование ближе к машинному представлению, так как оно вводит понятие состояния памяти, которое изменяется под воздействием программы. Каждое такое воздействие называется инструкцией и императивная программа есть набор упорядоченных инструкций. Состояние памяти может быть изменено при выполнении каждой инструкции. Операции ввода/вывода можно рассматривать как изменение оперативной памяти, видео памяти или файлов.

Подобный стиль программирования напрямую происходит от программирования на ассемблере. Мы встречаем этот стиль в языках программирования первого поколения (*Fortran*, *C*, *Pascal*, etc). Следующие элементы Objective CAML соответствуют приведенной модели:

- физически изменяемые¹ структуры данных, такие как массив или запись с изменяемыми (mutable) полями
- операции ввода/вывода
- структуры контроля выполнения программы как цикл и исключения.

¹в оригинальном издании (французском) используется выражение *физически изменяемые*, тогда как в английском переводе лишь *изменяемые*

Некоторые алгоритмы реализуются проще таким стилем программирования. Примером может послужить произведение двух матриц. Несмотря на то что эту операцию можно реализовать чисто функциональным способом, при этом списки заменят массивы, это не будет ни эффективнее, ни естественней по отношению к императивному стилю.

Интерес интеграции императивной модели в функциональный язык состоит в написании при необходимости подобных алгоритмов в этом стиле программирования. Два главных недостатка императивного программирования по отношению к функциональному это:

- усложнение системы типов языка и отклонение (rejecting) некоторых программ, которые бы без этого рассматривались бы как “корректные”,
- необходимость учитывать состоянием памяти и порядок вычисления.

Однако, при соблюдении некоторых правил написания программ, выбор стиля программирования предоставляет большие возможности в написании алгоритмов, что является главной целью языков программирования. К тому же, программа написанная в стиле близкому к стилю алгоритма, имеет больше шансов быть корректной (или по крайней мере быстрее реализована).

По этим причинам в Objective CAML имеются типы данных, значения которых физически изменяемые, структуры контроля выполнения программ и библиотека ввода/вывода в императивном стиле.

План главы

В данной главе продолжается представление базовых элементов Objective CAML затронутых в предыдущей главе, но теперь мы заинтересуемся императивными конструкциями. Глава разбита на 5 разделов. Первый, наиболее важный, раскрывает различные физически изменяемые структуры данных и описывает их представление в памяти. Во втором разделе кратко излагаются базовые операции ввода/вывода. Третий знакомит с новыми итеративными структурами контроля. В четвертом разделе рассказывается об особенностях выполнения императивных программ, в частности о порядке вычисления аргументов функции. В последнем разделе мы переделаем калькулятор предыдущей главы в калькулятор с памятью.

2.1 Физически изменяемые структуры данных

Значения следующих типов: массивы, строки, записи с модифицируемыми полями и указатели есть структурированные значения, компоненты которых могут быть физически изменены.

Мы уже видели, что переменная в Objective CAML привязана к значению и хранит эту связку до конца ее существования. Мы можем изменить связку лишь переопределением, но и в этом случае речь не будет идти о той же самой переменной. Новая переменная с тем же именем скроет предыдущую, которая перестанет быть доступной напрямую и останется неизменной. В случае с изменяемыми переменными, мы можем ассоциировать новое значение переменной без ее переопределения. Значение переменной доступно в режиме чтение/запись.

2.1.1 Векторы

Векторы или одномерные массивы группируют определенное число однотипных элементов. Существует несколько способов создания векторов, первый — перечисление элементов, разделенных запятой и заключенных между символами `[|` и `|]`.

```
# let v = [| 3.14; 6.28; 9.42 |] ;;
val v : float array = [|3.14; 6.28; 9.42|]
```

Функция `Array.create` с двумя аргументами на входе: размер вектора и начальное значение, возвращает новый вектор.

```
# let v = Array.create 3 3.14;;
val v : float array = [|3.14; 3.14; 3.14|]
```

Для того чтобы изменить или просто посмотреть значение элемента необходимо указать его номер.

Синтаксис

```
expr1 . ( expr2 )}
```

Синтаксис

```
expr1 . ( expr2 ) <- expr3
```

`expr1` должен быть вектором (тип `array`) с элементами типа `expr3`. Конечно, выражение `expr2` должно быть типа `int`. Результат модификации есть выражение типа `unit`. Номер первого элемента вектора 0 и последнего — размер вектора минус 1. Скобки вокруг выражения обязательны.

```
# v.(1) ;;
- : float = 3.14
# v.(0) <- 100.0 ;;
- : unit = ()
# v ;;
- : float array = [|100; 3.14; 3.14|]
```

Если индекс элемента не принадлежит интервалу индексов вектора, то возбуждается исключение в момент доступа.

```
# v.(-1) +. 4.0;;
Uncaught exception: Invalid_argument("Array.get")
```

Эта проверка осуществляется в момент выполнения программы, что может сказаться на ее скорости. Однако, это необходимо для избежания заполнения зоны памяти вне вектора, что может привести к серьезным ошибкам.

Функции манипуляции векторами являются частью модуля **Array** стандартной библиотеки, описание которого дано в главе 7 на стр. 235. В следующих примерах мы воспользуемся тремя функциями из этого модуля:

create создающая вектор заданного размера и начальным значением

length возвращающая размер вектора

append для конкатенации двух векторов

Совместное использование значений вектора

Все элементы вектора содержат значение, переданное во время создания вектора. В случае, если это значение структурное, оно будет совместно использоваться (sharing). Создадим, для примера, матрицу как вектор векторов при помощи функции **Array.create**:

```
# let v = Array.create 3 0;;
val v : int array = [|0; 0; 0|]
# let m = Array.create 3 v;;
val m : int array array = [| [|0; 0; 0|]; [|0; 0; 0|]; [|0; 0; 0|] |]
```

Если поменять одно из полей вектора **v**, который мы использовали для создания **m**, мы изменим все *строки* матрицы (см. рисунки 2.1 и 2.2).

```
# v.(0) <- 1;;
- : unit = ()
```

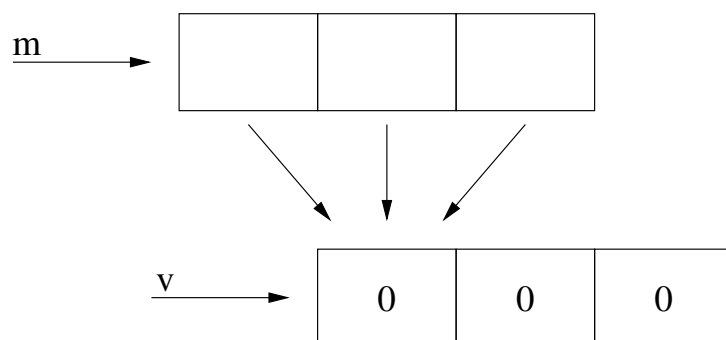


Рис. 2.1: Представление в памяти вектора с разделением элементов

```
# m;;
- : int array array = [[|1; 0; 0|]; [|1; 0; 0|]; [|1; 0; 0|]]
```

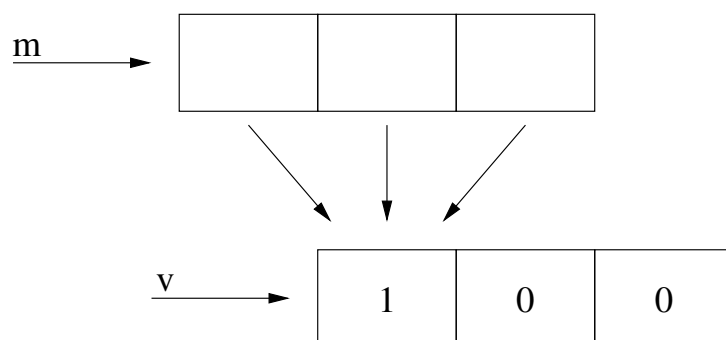


Рис. 2.2: Изменение элемента разделяемого вектора

Если значение инициализации вектора (второй аргумент функции `Array.create`) простое, атомарное, то оно копируется каждый раз и совместно используется если это значение структурное.

Мы называем атомарным значением то, размер которого не превышает стандартный размер значений Objective CAML, то есть **memory word** — целое, символ, булевый и конструкторы констант. Другие значения, называемые структурными, представлены указателем на зону памяти. Эта разница объяснена более детально в главе 8 стр. 273.

Векторы чисел с плавающей запятой есть особый случай. Несмотря на то что эти числа (`float`) — структурные значения, при создании вектора начальное значение копируется. Делается это для оптимизации. В главе 11 стр. 349, в которой обсуждается интерфейс с языком C мы обсудим эту проблему.

Не квадратная матрица

Матрица, вектор векторов, не обязательно должна быть квадратной. Действительно, мы можем заменить вектор на другой вектор с иным размером, что удобно для ограничения размера матрицы. Следующее значение `t` создает треугольную матрицу для коэффициентов треугольника Паскаля.

```
# let t = [|
  [|1|];
  [|1; 1|];
  [|1; 2; 1|];
  [|1; 3; 3; 1|];
  [|1; 4; 6; 4; 1|];
  [|1; 5; 10; 10; 5; 1|]
|] ;;

val t : int array array =
  [| [|1|]; [|1; 1|]; [|1; 2; 1|];
    [|1; 3; 3; 1|]; [|1; 4; 6; 4; ...|]; ... |]
# t.(3) ;;
- : int array = [|1; 3; 3; 1|]
```

В этом примере элемент вектора `t` по индексу `i` есть вектор целых чисел размером `i+1`. Для манипуляции такой матрицей необходимо знать размер каждого элемента вектора.

Копия векторов

При копировании вектора или конкатенации двух векторов мы получаем новый вектор. Модификация исходных векторов не повлияет на значение копий, кроме случая разделения значений рассмотренного ранее.

```
# let v2 = Array.copy v ;;
val v2 : int array = [|1; 0; 0|]
# let m2 = Array.copy m ;;
val m2 : int array array = [| [|1; 0; 0|]; [|1; 0; 0|]; [|1; 0; 0|] |]
# v.(1) <- 352;;
- : unit = ()
# v2;; - : int array = [|1; 0; 0|]
# m2 ;;
- : int array array = [| [|1; 352; 0|]; [|1; 352; 0|]; [|1; 352; 0|] |]
```

В приведенном примере видно что копия `m` содержит лишь указатели на `v`, если один из элементов `v` изменен, то `m2` тоже будет изменен. Кон-

катенация создает новый вектор длина которого равна сумме длин двух других.

```
# let mm = Array.append m m ;;
val mm : int array array =
  [[|1; 352; 0|]; [|1; 352; 0|]; [|1; 352; 0|]; [|1; 352; 0|];
   [|1; 352; ...|]; ...]
# Array.length mm ;;
- : int = 6
# m.(0) <- Array.create 3 0 ;;
- : unit = ()
# m ;;
- : int array array =
  [[|0; 0; 0|]; [|1; 352; 0|]; [|1; 352; 0|]]
# mm ;;
- : int array array = [[|1; 352; 0|]; [|1; 352; 0|]; [|1; 352; 0|]; [|1;
  352; 0|];
  [|1; 352; ...|]; ...]
```

Однако, изменение `v`, разделяемого `m` и `mm`, приведет к изменению обеих матриц:

```
# v.(1) <- 18 ;;
- : unit = ()
# mm;;
- : int array array =
  [[|1; 18; 0|]; [|1; 18; 0|]; [|1; 18; 0|];
  [|1; 18; 0|]; [|1; 18; ...|]; ...]
```

2.1.2 Строки

Строки можно рассматривать как частный случай массива символов. Однако, по причинам использования памяти этот тип специализирован и синтаксис доступа к элементам изменен.

Синтаксис

```
expr1 . [expr2]
```

Элементы строки могут быть физически изменены.

Синтаксис

```
expr1 . [expr2] <- expr3
```

```
# let s = "hello" ;;
val s : string = "hello"
# s.[2];;
- : char = 'l'
# s.[2] <- 'Z';;
- : unit = ()
# s;;
- : string = "heZlo"
```

2.1.3 Изменяемые поля записей

Поля записи могут быть объявлены изменяемыми. Для этого достаточно указать при декларации типа поля ключевое слово **mutable**

Синтаксис

```
type name = { ...; mutable name : t ; ... }
```

Пример определения точки плоскости.

```
# type point = { mutable xc : float; mutable yc : float } ;;
type point = { mutable xc: float; mutable yc: float }
# let p = { xc = 1.0; yc = 0.0 } ;;
val p : point = {xc=1; yc=0}
```

Для изменения значения поля объявленного **mutable** используйте следующий синтаксис.

Синтаксис

```
expr1.nom <- expr2
```

Выражение **expr1** должно быть типа запись содержащее поле **nom**. Операция модификации возвращает значение типа **unit**.

```
# p.xc <- 3.0 ;;
- : unit = ()
# p ;;
- : point = {xc=3; yc=0}
```

Напишем функцию перемещения точки, которая изменяет ее координаты. Здесь мы используем локальную декларацию с фильтражом, для упорядочения побочных эффектов.

```
# let moveto p dx dy =
  let () = p.xc <- p.xc +. dx
  in p.yc <- p.yc +. dy ;;
val moveto : point -> float -> float -> unit = <fun>
```



```
# moveto p 1.1 2.2 ;;
- : unit = ()
# p ;;
- : point = {xc=4.1; yc=2.2}
```

Мы можем определить изменяемые или не изменяемые поля, только поля, помеченные ключевым словом **mutable**, будут изменяемые.

```
# type t = { c1 : int; mutable c2 : int } ;;
type t = { c1: int; mutable c2: int }
# let r = { c1 = 0; c2 = 0 } ;;
val r : t = {c1=0; c2=0}
# r.c1 <- 1 ;;
Characters 0-9:
The label c1 is not mutable
# r.c2 <- 1 ;;
- : unit = ()
# r ;;
- : t = {c1=0; c2=1}
```

Далее, мы приводим пример использования записей с изменяемыми полями и массивов для того чтобы реализовать структуру стека (см 2.3.3 стр. 91).

2.1.4 Указатели

Objective CAML предлагает полиморфный тип **ref**, рассматриваемый как указатель на любое значение. В Objective CAML указатель точнее будет называть указатель на значение. Указываемое значение может быть изменено. Тип **ref** определен как запись с изменяемым полем.

```
type 'a ref = {mutable contents:'a}
```

Создать ссылку на значение можно функцией **ref**. Указываемое значение может быть получено использованием префикса **!**. Для модификации значения используем инфиксную функцию **(:=)**.

```
# let x = ref 3 ;;
val x : int ref = {contents=3}
# x ;;
- : int ref = {contents=3}
# !x ;;
- : int = 3
# x := 4 ;;
```

```

- : unit = ()
# !x ;;
- : int = 4
# x := !x+1 ;;
- : unit = ()
# !x ;;
- : int = 5

```

2.1.5 Полиморфизм и изменяемые значения

Тип **ref** параметризованный (1.2.6), что позволяет создавать указатели на любой тип. Однако, существует несколько ограничений. Представим, что нет никаких ограничений и мы можем объявить

```
let x = ref [] ;;
```

В этом случае тип переменной **x** будет **'a list ref** и можем изменить значение способом не соответствующим статической типизации Objective CAML.

```

x := 1 :: !x ;;
x := true :: !x ;;

```

При этом получаем одну и ту же переменную типа **int list** в один момент и **bool list** в другой.

Во избежании подобной ситуации, система типов Objective CAML использует новую категорию типа переменной — слабо типизированные переменные. Синтаксически, они отличаются предшествующим символом подчеркивания.

```

# let x = ref [] ;;
val x : 'a list ref = {contents=[]}

```

Переменная типа 'a не является параметром типа, то есть ее тип не известен до момента ее реализации. Лишь в момент первого использования **x**, после объявления, тип будет окончательно зафиксирован.

```

# x := 0::!x ;;
- : unit = ()
# x ;;
- : int list ref = {contents=[0]}

```

Таким образом тип переменной **x** **int list ref**.

Тип, содержащий неизвестную, является мономорфным, не смотря на то что его тип еще не был указан и невозможно реализовать то что неизвестно полиморфным типом.

```
# let x = ref [] ;;
val x : 'a list ref = {contents=[]}
# x := (function y -> ()):!x ;;
- : unit = ()
# x ;;
- : ('a -> unit) list ref = {contents=[<fun>]}
```

В этом примере, не смотря на то что мы реализуем неизвестную типом *a priori* полиморфным (`'a->unit`), тип остался мономорфным с новой неизвестной типа.

Это ограничение полиморфизма распространяется не только на указатели, но и на любое значение содержащее изменяемую часть: массивы, записи с полями объявленные `mutable`, и так далее. Все параметры типа, даже не имеющие никакого отношения к модифицируемым атрибутам, есть переменные слабого (`weak`) типа.

```
# type ('a,'b) t = { ch1 : 'a list ; mutable ch2 : 'b list } ;;
type ('a, 'b) t = { ch1: 'a list ; mutable ch2: 'b list }
# let x = { ch1 = [] ; ch2 = [] } ;;
val x : ('a, 'b) t = {ch1=[]; ch2=[]}
```

Warning

Эта модификация типа повлияет на чисто функциональные программы.

Если к полиморфной переменной применена полиморфная функция, то в результате мы получим переменную слабого типа, так как нельзя не учитывать что функция может создать физически изменяемое значение. Иными словами, результат будет всегда мономорфный.

```
# (function x -> x) [] ;;
- : 'a list = []
```

Тот же результат мы получим для частичных функций.

```
# let f a b = a ;;
val f : 'a -> 'b -> 'a = <fun>
# let g = f 1 ;;
val g : 'a -> int = <fun>
```

Для получения полиморфного типа необходимо вычесть второй аргумент `f` и применить ее:

```
# let h x = f 1 x ;;
val h : 'a -> int = <fun>
```

Действительно, выражения определяющее `x` есть функциональное выражение `function x -> f 1 x`. Его вычисление даст замыкание, которое не рискует произвести побочный эффект, так как тело функции не вычислено.

В общем случае, мы различаем выражение *не расширяемое*, расчет которого не приведет к побочным эффектам и другим *расширяемым*. Система типов Objective CAML классифицирует выражения в соответствии с синтаксисом:

- выражения *не расширяемые* состоят в основном из переменных, конструкторов не изменяемых значений и абстракций
- выражения *расширяемые* состоят из применений, конструкторов изменяемых значений. Так же к ним необходимо добавить структуры контроля: условный (*conditional*) или фильтр.

2.2 Ввод/Вывод

Функции в/в вычисляют значение (чаще всего типа `unit`), однако в ходе вычисления, они изменяют состояние устройств в/в: изменение буфера клавиатуры, перерисовка экрана, запись в файл и так далее. Для каналов ввода и вывода определены для типа: `in_channel` и `out_channel`. При обнаружении конца файла возбуждается исключение `End_of_file`. Следующие константы соответствуют стандартным каналам ввода, вывода и ошибок *a la Unix*: `stdin`, `stdout` и `stderr`.

2.2.1 Каналы

Функции ввода/вывода стандартной библиотеки Objective CAML, манипулируют каналами типа `in_channel` и `out_channel` соответственно. Для создания подобных каналов используется функция:

```
# open_in;;
- : string -> in_channel = <fun>
# open_out;;
- : string -> out_channel = <fun>
```

`open_in` открывает файл², если он не существует, возбуждается исключение `Sys_error`.

`open_out` создает указанный файл если такой не существует, если же он существует то его содержимое уничтожается.

²с правом на чтение, при необходимости

```
# let ic = open_in "koala";
val ic : in_channel = <abstr>
# let oc = open_out "koala";
val oc : out_channel = <abstr>
```

Функции закрывания каналов:

```
# close_in ;;
- : in_channel -> unit = <fun>
# close_out ;;
- : out_channel -> unit = <fun>
```

2.2.2 Чтение/запись

Стандартные функции чтения/записи:

```
# input_line ;;
- : in_channel -> string = <fun>
# input ;;
- : in_channel -> string -> int -> int -> int = <fun>
# output ;;
- : out_channel -> string -> int -> int -> unit = <fun>
```

- `input_line ic`: читает из входного канала `ic` символы до обнаружения возврата каретки или конца файла и возвращает в форме строки (не включая возврат каретки).
- `input ic s p l`: считывает `l` символов из канала `ic` и помещает их в строку `s` со смещением `p`. Функция возвращает число прочитанных символов.
- `output oc s p l`: записывает в выходной канал `oc` часть строки `s` начиная с символа `p` длиной `l`.

Следующие функции чтения (записи) из (в) стандартного входа:

```
# read_line ;;
- : unit -> string = <fun>
# print_string ;;
- : string -> unit = <fun>
# print_newline ;;
- : unit -> unit = <fun>
```

Другие значения простых типов данных могут быть прочтены/записаны, значения таких типов могут быть переведены в тип список символов.

Локальное объявление и порядок выполнения Попробуем вывести на экран выражение формы `let x=e1 in e2`. Зная, что в общем случае локальная переменная `x` может быть использована в `e2`, выражение `e1` вычислено первым и только затем `e2`. Если эти оба выражения есть императивные функции с побочным эффектом результат которых `()`, мы выполнили их в правильном порядке. В частности, так как нам известен тип возвращаемого результата `e1`: константа `()` типа `unit`, мы получим последовательный вывод на экран используя сопоставление с образцом `()`.

```
# let () = print_string "and one," in
  let () = print_string " and two," in
  let () = print_string " and three" in
    print_string " zero" ;;
and one, and two, and three zero— : unit = ()
```

2.2.3 Пример Больше/Меньше

В следующем примере мы приводим игру Больше/Меньше, состоящую в поиске числа, после каждого ответа программа выводит на экран сообщение в соответствии с тем что предложенное число больше или меньше загаданного.

```
# let rec hilo n =
  let () = print_string "type a number: " in
  let i = read_int ()
  in
    if i = n then
      let () = print_string "BRAVO" in
      let () = print_newline ()
      in print_newline ()
    else
      let () =
        if i < n then
          let () = print_string "Higher"
          in print_newline ()
        else
          let () = print_string "Lower"
          in print_newline ()
      in hilo n ;;
val hilo : int -> unit = <fun>
```

Результат запуска

```
# hilo 64;;
type a number: 88
Lower
type a number: 44
Higher
type a number: 64
BRAVO

— : unit = ()
```

2.3 Структуры контроля

Операции ввода/вывода и изменяемые значения имеют побочный эффект. Их использование упрощено императивным стилем программирования с новыми структурами контроля. В этом параграфе мы поговорим о последовательных и итеративных структурах.

Нам уже приходилось встречать (см. 1.1.2 стр. 23) условную структуру контроля **if then** смысл которой такой же как и в императивных языках программирования.

Пример

```
# let n = ref 1 ;;
val n : int ref = {contents=1}
# if !n > 0 then n := n-1 ;;
Characters 20-21:
```

This expression has type int ref but is here used with type int

2.3.1 Последовательность

Самая типичная императивная структура — последовательность, которая позволяет вычислять выражения разделенные точкой с запятой в порядке слева направо.

Синтаксис

```
expr1 ;...; exprn
```

Последовательность — это выражение, результат которого есть результат последнего выражения (здесь **exprn**). Все выражения вычислены и побочный эффект каждой учтен.

```
# print_string "2 = "; 1+1 ;;
2 = - : int = 2
```

С побочным эффектом, мы получаем обычные конструкции императивного программирования.

```
# let x = ref 1 ;;
val x : int ref = {contents=1}
# x:=!x+1 ; x:=!x*4 ; !x ;;
- : int = 8
```

В связи с тем что значения предшествующие точке запятой не сохранены, Objective CAML выводит предупреждение в случае если их тип отличен от типа `unit`.

```
# print_int 1; 2 ; 3 ;;
Characters 14-15:
Warning: this expression should have type unit.
1- : int = 3
```

Чтобы избежать этого сообщения, можно воспользоваться функцией `ignore`.

```
# print_int 1; ignore 2; 3 ;;
1- : int = 3
```

Другое сообщение будет выведено в случае если забыт аргумент функции

```
# let g x y = x := y ;;
val g : 'a ref -> 'a -> unit = <fun>
# let a = ref 10;;
val a : int ref = {contents=10}
# let u = 1 in g a ; g a u ;;
Characters 13-16:
Warning: this function application is partial,
maybe some arguments are missing.
- : unit = ()
# let u = !a in ignore (g a) ; g a u ;;
- : unit = ()
```

Чаще всего мы используем скобки для определения зоны видимости.
Синтаксис

(expr)

Синтаксис

begin expr **end**

Теперь мы можем написать Больше/Меньше простым стилем (стр. 86).

```
# let rec hilo n =
  print_string "type a number: ";
  let i = read_int () in
  if i = n then print_string "BRAVO\n\n"
  else
    begin
      if i < n then print_string "Higher\n" else print_string "Lower\n"
    end ;
    hilo n
  end ;;
val hilo : int -> unit = <fun>
```

2.3.2 Циклы

Структуры итеративного контроля тоже не принадлежат “миру” функционального программирования. Условие повторения цикла или выхода из него имеет смысл в случае когда есть физическое изменение памяти. Существует две структуры итеративного контроля: цикл **for** для ограниченного количества итераций и **while** для неограниченного. Структуры цикла возвращают константу () типа **unit**.

Цикл **for** может быть возрастающим (**to**) или убывающим (**downto**) с шагом в одну единицу.

Синтаксис

```
for nom = expr1 to expr2 do expr3 done
for nom = expr1 downto expr2 do expr3 done
```

Тип выражений expr1 и expr2 — **int**. Если тип expr3 не **unit**, компилятор выдаст предупреждение.

```
# for i=1 to 10 do
  print_int i;
  print_string " "
done;
print_newline() ;;
1 2 3 4 5 6 7 8 9 10
- : unit = ()
# for i=10 downto 1 do
```

```

    print_int i;
    print_string " "
  done;
  print_newline() ;;
10 9 8 7 6 5 4 3 2 1
- : unit = ()

```

Неограниченный цикл **while** имеет следующий синтаксис

```
while expr1 do expr2 done
```

Тип выражения **expr1** должен быть **bool**. И, как в случае **for**, если тип **expr2** не **unit**, компилятор выдаст предупреждение.

```

# let r = ref 1
in while !r < 11 do
  print_int !r ;
  print_string " " ;
  r := !r+1
done ;;
1 2 3 4 5 6 7 8 9 10 - : unit = ()

```

Необходимо помнить, что циклы — это такие же выражения как и любые другие, они вычисляют значение **()** типа **unit**.

```

# let f () = print_string "-- end\n" ;;
val f : unit -> unit = <fun>
# f (for i=1 to 10 do print_int i ; print_string " " done) ;;
1 2 3 4 5 6 7 8 9 10 -- end
- : unit = ()

```

Обратите внимание на то, что строка “- end\n” выводится после цифр 1...10: подтверждение того что аргументы (в данном случае цикл) вычисляются перед передачей функции.

В императивном стиле тело цикла (выражение **expr**) не возвращает результата, а производит побочный эффект. В Objective CAML, если тип тела цикла не **unit**, то компилятор выдает сообщение :

```

# let s = [5; 4; 3; 2; 1; 0] ;;
val s : int list = [5; 4; 3; 2; 1; 0]
# for i=0 to 5 do List.tl s done ;;

```

Characters 17-26:

Warning: this expression should have type unit.

```
- : unit = ()
```

2.3.3 Пример: реализация стека

В нашем примере структура данных `stack` будет запись с двумя полями: массив элементов и индекс первой свободной позиции в массиве.

```
# type 'a stack = { mutable ind:int; size:int; mutable elts : 'a array }
;;
```

В поле `size` будет храниться максимальный размер стека.

Операции над стеком будут `init_stack` для инициализации, `push` для добавления и `pop` для удаления элемента.

```
# let init_stack n = {ind=0; size=n; elts =[]} ;;
val init_stack : int -> 'a stack = <fun>
```

Эта функция не может создавать не пустой массив, так для создания массива необходимо передать элемент. Поэтому поле `elts` получает пустой массив.

Два исключения добавлены на случай попытки удалить элемент из пустого стека и добавить элемент в полный. Они используются в функциях `pop` и `push`.

```
# exception Stack_empty ;;
# exception Stack_full ;;

# let pop p =
  if p.ind = 0 then raise Stack_empty
  else (p.ind <- p.ind - 1; p.elts.(p.ind)) ;;
val pop : 'a stack -> 'a = <fun>
# let push e p =
  if p.elts = [] then
    (p.elts <- Array.create p.size e;
     p.ind <- 1)
  else if p.ind >= p.size then raise Stack_full
  else (p.elts.(p.ind) <- e; p.ind <- p.ind + 1) ;;
val push : 'a -> 'a stack -> unit = <fun>
```

Небольшой пример использования:

```
# let p = init_stack 4 ;;
val p : '_a stack = {ind=0; size=4; elts=[]}
# push 1 p ;;
- : unit = ()
# for i = 2 to 5 do push i p done ;;
Uncaught exception: Stack_full
```

```

# p ;;
- : int stack = {ind=4; size=4; elts=[1; 2; 3; 4]}
# pop p ;;
- : int = 4
# pop p ;;
- : int = 3

```

Чтобы избежать исключения `Stack_full` при переполнении стека, мы можем каждый раз увеличивать его размер. Для этого нужно чтобы поле `size` было модифицируемым.

```

# type 'a stack =
  {mutable ind:int ; mutable size:int ; mutable elts : 'a array} ;;
# let init_stack n = {ind=0; size=max n 1; elts = []} ;;
# let n_push e p =
  if p.elts = []
  then
    begin
      p.elts <- Array.create p.size e;
      p.ind <- 1
    end
  else if p.ind >= p.size then
    begin
      let nt = 2 * p.size in
      let nv = Array.create nt e in
      for j=0 to p.size-1 do nv.(j) <- p.elts.(j) done ;
      p.elts <- nv;
      p.size <- nt;
      p.ind <- p.ind + 1
    end
  else
    begin
      p.elts.(p.ind) <- e ;
      p.ind <- p.ind + 1
    end ;;
val n_push : 'a -> 'a stack -> unit = <fun>

```

Однако необходимо быть осторожным со структурами которые могут беспредельно увеличиваться. Вот небольшой пример стека, увеличивающегося по необходимости.

```

# let p = init_stack 4 ;;
val p : 'a stack = {ind=0; size=4; elts=[]}

```

```

# for i = 1 to 5 do n_push i p done ;;
- : unit = ()
# p ;;
- : int stack = {ind=5; size=8; elts=[[1; 2; 3; 4; 5; 5; 5; 5]]}
# p.size ;;
- : int = 8

```

В функции `pop` желательно добавить возможность уменьшения размера стека, это позволит сэкономить память.

2.3.4 Пример: расчет матриц

В этом примере мы определим тип “матрица” — двумерный массив чисел с плавающей запятой и несколько функций. Мономорфный тип `mat` это запись, состоящая из размеров и элементов матрицы. Функции `creat_mat`, `access_mat` и `mod_mat` служат для создания, доступа и изменения элементов матрицы.

```

# type mat = { n:int; m:int; t: float array array };;
type mat = { n: int; m: int; t: float array array }
# let create_mat n m = { n=n; m=m; t = Array.create_matrix n m 0.0
  } ;;
val create_mat : int -> int -> mat = <fun>
# let access_mat m i j = m.t.(i).(j) ;;
val access_mat : mat -> int -> int -> float = <fun>
# let mod_mat m i j e = m.t.(i).(j) <- e ;;
val mod_mat : mat -> int -> int -> float -> unit = <fun>
# let a = create_mat 3 3 ;;
val a : mat = {n=3; m=3; t=[[[0; 0; 0]; [0; 0; 0]; [0; 0; 0]]}
# mod_mat a 1 1 2.0; mod_mat a 1 2 1.0; mod_mat a 2 1 1.0 ;;
- : unit = ()
# a ;;
- : mat = {n=3; m=3; t=[[[0; 0; 0]; [0; 2; 1]; [0; 1; 0]]}

```

Сумма двух матриц `a` и `b` есть матрица `c`, такая что

$$c_{ij} = a_{ij} + b_{ij}$$

```

# let add_mat p q =
  if p.n = q.n && p.m = q.m then
    let r = create_mat p.n p.m in
    for i = 0 to p.n-1 do
      for j = 0 to p.m-1 do
        mod_mat r i j (p.t.(i).(j) +. q.t.(i).(j))

```

```

    done
  done ;
  r
  else failwith "add_mat : dimensions incompatible";;
val add_mat : mat -> mat -> mat = <fun>
# add_mat a a ;;
- : mat = {n=3; m=3; t=[[[0; 0; 0]]; [[0; 4; 2]]; [[0; 2; 0]]]}

```

Произведение двух матриц **a** и **b** есть матрица **c**, такая что $c_{ij} = \sum_{k=1}^m a_{ik} * b_{ki}$

```

# let mul_mat p q =
  if p.m = q.n then
    let r = create_mat p.n q.m in
    for i = 0 to p.n-1 do
      for j = 0 to q.m-1 do
        let c = ref 0.0 in
        for k = 0 to p.m-1 do
          c := !c +. (p.t.(i).(k) *. q.t.(k).(j))
        done;
        mod_mat r i j !c
      done
    done;
  r
  else failwith "mul_mat : dimensions incompatible" ;;
val mul_mat : mat -> mat -> mat = <fun>
# mul_mat a a;;
- : mat = {n=3; m=3; t=[[[0; 0; 0]]; [[0; 5; 2]]; [[0; 2; 1]]]}

```

2.4 Порядок вычисления аргументов

В функциональном языке программирования порядок вычисления аргументов не имеет значения. Из-за того что нет ни изменения памяти, ни приостановки вычисления, расчет одного аргумента не влияет на вычисление другого. Objective CAML поддерживает физически изменяемые значения и исключения, поэтому пренебречь порядком вычисления аргументов нельзя. Следующий пример специфичен для Objective CAML 2.04 ОС Linux на платформе Intel:

```

# let new_print_string s = print_string s; String.length s ;;
val new_print_string : string -> int = <fun>
# (+) (new_print_string "Hello ") (new_print_string "World!") ;;

```

```
World!Hello — : int = 12
```

По выводу на экран мы видим что вторая строка печатается после первой.

Таков же результат для исключений:

```
# try (failwith "function") ( failwith "argument") with Failure s -> s;;
— : string = "argument"
```

Если необходимо указать порядок вычисления аргументов, необходимо использовать локальные декларации, форсируя таким образом порядок перед вызовом функции. Предыдущий пример может быть переписан следующим способом:

```
# let e1 = (new_print_string "Hello ")
  in let e2 = (new_print_string "World!")
  in (+) e1 e2 ;;
Hello World!— : int = 12
```

В Objective CAML порядок вычисления не указан, на сегодняшний день все реализации caml делают это слева направо. Однако рассчитывать на это может быть рискованно, в случае если в будущем язык будет реализован иначе.

Это вечный сюжет дебатов при концепции языка. Нужно ли специально не указывать некоторые особенности языка и предложить программистам не пользоваться ими, иначе они рискуют получить разные результаты для разных компиляторов. Или же необходимо их указать и, следовательно, разрешить программистам ими пользоваться, что усложнит компилятор и сделает невозможным некоторые оптимизации?

2.5 Калькулятор с памятью

Вернемся к нашему примеру с калькулятором описанным в предыдущей главе и добавим ему пользовательский интерфейс. Теперь мы будем вводить операции напрямую и получать результат сразу без вызова функции транзиций (из одного состояния в другое) при каждом нажатии на кнопку.

Добавим 4 новые кнопки: **C** которая очищает экран, **M** для сохранения результата в памяти, **m** для его вызова из памяти и **OFF** для выключения калькулятора. Что соответствует следующему типу:

```
# type key = Plus | Minus | Times | Div | Equals | Digit of int
           | Store | Recall | Clear | Off ;;
```

Теперь определим функцию, переводящую введенные символы в значение типа `key`. Исключение `Invalid_key` отлавливает все символы, которые не соответствуют кнопкам калькулятора. Функция `code` модуля `Char` переводит символ в код *ASCII*.

```
# exception Invalid_key ;;
exception Invalid_key
# let translation c = match c with
  | '+' -> Plus
  | '-' -> Minus
  | '*' -> Times
  | '/' -> Div
  | '=' -> Equals
  | 'C' | 'c' -> Clear
  | 'M' -> Store
  | 'm' -> Recall
  | 'o' | 'O' -> Off
  | '0'..'9' as c -> Digit ((Char.code c) - (Char.code '0'))
  | _ -> raise Invalid_key ;;
val translation : char -> key = <fun>
```

В императивном стиле, функция перехода (`transition`) не приведет к новому состоянию, а физически изменит текущее состояние калькулятора. Таким образом необходимо изменить тип `state` так, чтобы его поля были модифицируемые. Наконец, определим исключение `Key_off` для обработки нажатия на кнопку `OFF`.

```
# type state = {
  mutable lcd : int; (* last computation done *)
  mutable lka : bool; (* last key activated *)
  mutable loa : key; (* last operator activated *)
  mutable vpr : int; (* value printed *)
  mutable mem : int (* memory of calculator *)
};;

# exception Key_off ;;
exception Key_off
# let transition s key = match key with
  | Clear -> s.vpr <- 0
  | Digit n -> s.vpr <- ( if s.lka then s.vpr*10+n else n );
                    s.lka <- true
  | Store -> s.lka <- false ;
                    s.mem <- s.vpr
```



```

| Recall -> s.lka <- false ;
              s.vpr <- s.mem
| Off -> raise Key_off
| _ -> let lcd = match s.loa with
              Plus -> s.lcd + s.vpr
              | Minus -> s.lcd - s.vpr
              | Times -> s.lcd * s.vpr
              | Div -> s.lcd / s.vpr
              | Equals -> s.vpr
              | _ -> failwith "transition: impossible match"
in
  s.lcd <- lcd ;
  s.lka <- false ;
  s.loa <- key ;
  s.vpr <- s.lcd;;
val transition : state -> key -> unit = <fun>

```

Определим функцию запуска калькулятора **go**, которая возвращает **()**, так как нас интересует только ее эффект на окружение (ввод/вывод, изменение состояния). Ее аргумент есть константа **()**, так как наш калькулятор автономен (он сам определяет свое начальное состояние) и интерактивен (данные необходимые для расчета вводятся с клавиатуры по мере необходимости). Транзиции реализуются в бесконечном цикле (**while true do**) из которого мы можем выйти при помощи исключения **Key_off**.

```

# let go () =
  let state = { lcd=0; lka=false; loa=Equals; vpr=0; mem=0 }
  in try
    while true do
      try
        let input = translation (input_char stdin)
        in transition state input ;
        print_newline () ;
        print_string "result: " ;
        print_int state.vpr ;
        print_newline ()
      with
        Invalid_key -> () (* no effect *)
    done
  with
    Key_off -> () ;;

```

val go : unit -> unit = <bfun>

Заметим, что начальное состояние должно быть либо передано в параметре, либо объявлено локально внутри функции `go`, для того чтобы оно каждый раз инициализировалось при запуске этой функции. Если бы мы использовали значение `initial_state` в функциональном стиле, калькулятор начинал бы работать со старым состоянием, которое он имел перед выключением. Таким образом было бы не просто использовать два калькулятора в одной программе.

2.6 Резюме

В этой главе вы видели применение стилей императивного программирования (физически изменяемые значения, ввод-вывод, структуры итеративного контроля) в функциональном языке. Только **mutable** значения, такие как строки, векторы и записи с изменяемыми полями могут быть физически изменены. Другие значения не могут меняться после их создания. Таким образом мы имеем значения *read-only* для функциональной части и значения *read-write* для императивной.

Нужно отметить, что в случае отказа от использования императивных возможностей языка, это расширение функционального “ядра” языка не меняют его возможностей функционального программирования, за исключением некоторых, легко решаемых проблем с типами.

Глава 3

Функциональный и императивный стиль

Введение

Языки функционального и императивного программирования различаются контролем выполнения программы и управлением памятью.

- функциональная программа вычисляет выражение, в следствии чего мы получаем некоторый результат. Порядок в котором выполнены операции расчета и физическое представление данных не влияют на результат, он одинаков во всех случаях. При таком порядке вычислений, сбор памяти в Objective CAML неявно осуществляется вызовом автоматического сборщика мусора или Garbage Collector (GC) (см. главу 8).
- императивная программа это список инструкций изменяющих состояние памяти. Каждый этап выполнения строго определен структурами контроля, указывающими на следующую инструкцию. Такие программы чаще всего манипулируют указателями на значение, чем самими значениями. Отсюда необходимость явного выделения и освобождения памяти, что может порой приводить к ошибкам доступа к памяти, однако ничто не запрещает использование GC.

Императивные языки предоставляют больший контроль над выполнением программы и памятью. Находясь ближе к реальной машине, такой код может быть эффективнее, но при этом код теряет в устойчивости выполнения. Функциональное программирование предоставляет более высокий уровень абстракции и таким образом лучший уровень

устойчивости выполнения: типизация (динамическая или статическая) помогает избежать некорректных значений, автоматическая сборка мусора, хотя и замедляет скорость выполнения, обеспечивает правильное манипулирование областями памяти.

Исторически обе эти парадигмы программирования существовали в разных сферах: символьные программы для первого случая и числовые для второго. Однако, с тех времен кое-что изменилось, в частности появилась техника компиляции функциональных языков и выросла эффективность GC. С другой стороны, устойчивость выполнения стала важным критерием, иногда даже важнейшим критерием качества программного обеспечения. В подтверждение этому “коммерческий аргумент” языка Java: эффективность не должна преобладать над правильностью, оставаясь при этом благоразумно хорошей. Эта идея приобретает с каждым днем новых сторонников в мире производителей программного обеспечения.

Objective CAML придерживается этой позиции: он объединяет обе парадигмы, расширяя таким образом область своего применения и облегчая написание алгоритмов в том или ином стиле. Он сохраняет, однако, хорошие свойства правильности выполнения благодаря статической типизации, GC и механизму исключений. Исключения — это первая структура контроля выполнения позволяющая приостановить/продолжить расчет при возникновении определенных условий. Эта особенность находится на границе двух стилей программирования, хоть она и не изменяет результат, но может изменить порядок вычислений. Введение физически изменяемых значений может повлиять на чисто функциональную часть языка. Порядок вычисления аргументов функции становится определяемым если это вычисление производит побочный эффект. По этим причинам подобные языки называются “не чистые функциональные языки”. Мы теряем часть абстракции, так как программист должен учитывать модель памяти и выполнение программы. Это не всегда плохо, в частности для читаемости кода. Однако, императивные особенности изменяют систему типов языка: некоторые функциональные программы, с теоретически правильными типами, не являются правильными на практике из-за введения ссылок (reference). Хотя такие программы могут быть легко переписаны.

План главы

В этой главе мы сравним функциональную и императивную модель Objective CAML по критерию контроля выполнения программы и пред-

ставления значений в памяти. Смесь обоих стилей позволяет конструировать новые структуры данных. Это будет рассмотрено в первом разделе. Во втором разделе мы обсудим выбор между композицией функций (composition of functions) или последовательности (sequencing) с одной стороны и разделением (sharing) или копирование значений с другой. Третий раздел выявляет интерес смешивание двух стилей для создания функциональных изменяемых данных (mutable functional data), что позволит создавать не полностью вычисленные (evaluated) данные. В четвертом разделе рассмотрены streams, потенциально бесконечные потоки данных и их интеграция, посредством сопоставления с образцом.

3.1 Сравнение между функциональным и императивным стилями

Воспользуемся строками (`string`) и списками (`'a list`) для иллюстрации разницы между двумя стилями.

3.1.1 С функциональной стороны

`map` это одна из классических функций в среде функциональных языков. В чистом функциональном стиле она пишется так:

```
# let rec map f l = match l with
| [] -> []
| t::q -> (f t) :: (map f q) ;;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

Конечный список состоит из результатов применения функции `f` к элементам списка переданного в аргументе. Он рекурсивно создается указывая начальный элемент (заголовок) (`f t`) и последующую часть (хвост) (`map f q`). В частности, программа не указывает какой из них будет вычислен первым.

При этом, для написания такой функции программисту не нужно знать физическое представление данных. Проблемы выделения памяти и разделения данных решаются Objective CAML без внешнего вмешательства программиста. Следующий пример иллюстрирует это:

```
# let example = [ "one" ; "two" ; "three" ] ;;
val example : string list = ["one"; "two"; "three"]
# let result = map (function x -> x) example ;;
val result : string list = ["one"; "two"; "three"]
```

Оба списка `example` и `result` содержат одинаковые значения:

```
# example = result ;;
- : bool = true
```

Оба значения имеют одинаковую структуру, хоть они и представлены в памяти по разному. Убедимся в этом при помощи проверки на физическое равенство:

```
# example == result ;;
- : bool = false
# (List.tl example) == (List.tl result) ;;
- : bool = false
```

3.1.2 Императивная сторона

Вернемся к предыдущему примеру и изменим строку списка `result`.

```
# (List.hd result) .[1] <- 's' ;;
- : unit = ()
# result ;;
- : string list = ["ose"; "two"; "three"]
# example ;;
- : string list = ["ose"; "two"; "three"]
```

Определено, изменив список `result`, мы изменили список `example`. То есть знание физической структуры необходимо как только мы пользуемся императивными особенностями.

Рассмотрим теперь как порядок вычисления аргументов функции может стать ловушкой в императивном программировании. Определим структуру изменяемого списка, а так же функции создания, добавления и доступа:

```
# type 'a ilist = { mutable c : 'a list } ;;
type 'a ilist = { mutable c: 'a list }
# let icreate () = { c = [] }
  let iempty l = (l.c = [])
  let icons x y = y.c <- x::y.c ; y
  let ihd x = List.hd x.c
  let itl x = x.c <- List.tl x.c ; x ;;
val icreate : unit -> 'a ilist = <fun>
val iempty : 'a ilist -> bool = <fun>
val icons : 'a -> 'a ilist -> 'a ilist = <fun>
val ihd : 'a ilist -> 'a = <fun>
```

```

val itl : 'a ilist -> 'a ilist = <fun>
# let rec imap f l =
    if iempty l then icreate()
    else icons (f (ihd l)) (imap f (itl l)) ;;
val imap : ('a -> 'b) -> 'a ilist -> 'b ilist = <fun>

```

Несмотря на то что мы переняли общую структуру функции `map` предыдущего параграфа, с `imap` мы получим другой результат:

```

# let example = icons "one" (icons "two" (icons "three" (icreate())))) ;;
val example : string ilist = {c=["one"; "two"; "three"]}
# imap (function x -> x) example ;;
Uncaught exception: Failure("hd")

```

В чем тут дело? Вычисление `(itl l)` произошло раньше чем `(ihd l)` и в последней итерации `imap`, список `l` пустой в момент обращения к его заголовку. Список `example` теперь пуст, хоть мы и не получили никакого результата:

```

# example ;;
- : string ilist = {c=[]}

```

Проблема функции `imap` в недостаточном контроле смеси стилей программирования: мы предоставили системе выбор порядка вычисления. Переформулируем функцию `imap` явно указав порядок при помощи конструкции `let .. in ..`

```

# let rec imap f l =
    if iempty l then icreate()
    else let h = ihd l in icons (f h) (imap f (itl l)) ;;
val imap : ('a -> 'b) -> 'a ilist -> 'b ilist = <fun>
# let example = icons "one" (icons "two" (icons "three" (icreate())))) ;;
val example : string ilist = {c=["one"; "two"; "three"]}
# imap (function x -> x) example ;;
- : string ilist = {c=["one"; "two"; "three"]}

```

Однако, начальный список опять утерян:

```

# example ;;
- : string ilist = {c=[]}

```

Использование оператора последовательности (sequencing operator) и цикла есть другой способ явного указания порядка.

```

# let imap f l =
    let l_res = icreate ()
    in while not (iempty l) do

```

```

        ignore (icons (f (ihd l)) l_res) ;
        ignore (itl l)
    done ;
    { l_res with c = List.rev l_res.c } ;;
val imap : ('a -> 'b) -> 'a ilist -> 'b ilist = <fun>
# let example = icons "one" (icons "two" (icons "three" (icreate()))) ;;
val example : string ilist = {c=["one"; "two"; "three"]}
# imap (function x -> x) example ;;
- : string ilist = {c=["one"; "two"; "three"]}

```

Присутствие `ignore` — это факт того что нас интересует побочный эффект функций над ее аргументами, а не их (функций) результат. Так же было необходимо выстроить в правильном порядке элементы результата (функцией `List.rev`).

3.1.3 Рекурсия или итерация

Часто ошибочно ассоциируют рекурсию с функциональным стилем и императивный с итерацией. Чисто функциональная программа не может быть итеративной, так как значение условия цикла не меняется никогда. Тогда как императивная программа может быть рекурсивной: функция `imap` тому пример.

Значения аргументов функции хранятся во время ее выполнения. Если она (функция) вызовет другую функцию, то эта последняя сохранит и свои аргументы. Эти значения хранятся в стеке выполнения. При возврате из функции эти значения извлечены из стека. Зная что пространство памяти ограничено, мы можем дойти до ее предела, используя функцию с очень большой глубиной рекурсии. В подобных случаях Objective CAML возбуждает исключение `Stack_overflow`.

```

# let rec succ n = if n = 0 then 1 else 1 + succ (n-1) ;;
val succ : int -> int = <fun>
# succ 100000 ;;
Stack overflow during evaluation (looping recursion?).

```

В итеративной версии место занимаемое функцией `succ_iter` в стеке не зависит от величины аргумента.

```

# let succ_iter n =
    let i = ref 0 in
    for j=0 to n do incr i done ;    !i ;;
val succ_iter : int -> int = <fun>
# succ_iter 100000 ;;

```



```
— : int = 100001
```

У следующей версии функции *a priori* аналогичная глубина рекурсии, однако она успешно выполняется с тем же аргументом.

```
# let succ_rt n =
  let rec succ_aux n accu =
    if n = 0 then accu else succ_aux (n-1) (accu+1)
  in
    succ_aux 1 n ;;
val succ_rt : int -> int = <fun>
# succ_rt 100000 ;;
— : int = 100001
```

Данная функция имеет специальную форму рекурсивного вызова, так называемую конечную рекурсию при которой результат вызова функции будет результатом функции без дополнительных вычислений. Таким образом отпадает необходимость хранить аргументы функции во время вычисления рекурсивного вызова. Если Objective CAML распознал конечную рекурсивную функцию, то перед ее вызовом аргументы извлекаются из стека.

Распознавание конечной рекурсии существует во многих языках, однако это свойство просто необходимо функциональным языкам, где она естественно много используется.

3.2 Какой стиль выбрать?

Естественно, речь не идет о “священных” либо эстетических для каждого понятиях, *a priori* не существует стиля красивее или лучше чем другой. Однако, в зависимости от решаемой проблемы, один стиль может быть более подходящим или более адаптированным чем другой.

Первое правило — простота. Желаемый алгоритм (будь то в книге или лишь в голове программиста) уже определен в каком-то стиле, вполне естественно его и использовать при реализации.

Второй критерий — эффективность программы. Можно сказать что императивная программа (хорошо написанная) более эффективна чем ее функциональный аналог, но в очень многих случаях разница между ними не достаточно существенна для оправдания сложности кода императивного стиля, там где функциональный был бы естественен. Функция `map` есть хороший пример натурально выраженной в функциональном стиле проблемы и мы ни в чем не выиграем написав ее в императивном стиле.

3.2.1 Последовательность или композиция функций

Мы уже видели, что как только в программе появляются побочные эффекты, необходимо явно указывать порядок выполнения элементов программы. Это может быть сделано двумя способами:

функциональным опираясь на то что Objective CAML строгий язык, то есть аргумент вычислен до того как он будет передан функции. В выражение `(f (g x))` сначала вычисляется `(g x)` и затем передача этого результат функции `f`. В более сложных выражениях, промежуточный результат может быть именован конструкцией `let in`, но принцип остается тем же: `let aux=(g x) in (f aux)`.

императивный используя последовательность или другую структуру контроля (цикл). В этом случае, результатом будет побочный эффект над памятью, а не значение возвращенное функцией: `aux:=(g x) ; (f!aux)`.

Давайте рассмотрим данную проблему выбора стиля на следующем примере. Рекурсивный алгоритм быстрой сортировки вектора описывается следующим образом:

1. выбрать опорную точку: выбрать индекс элемента в векторе
2. переставить элементы вокруг этой точки: переставить элементы вектора так чтобы значения меньше значения в опорной точке были слева от нее, а большие значения справа
3. отсортировать (тем же алгоритмом) два полученных вектора: элементы после опорной точки и до нее

Сортировка вектора — означает изменение его состояния, поэтому мы должны использовать императивный стиль, как минимум для манипуляции данными.

Начнем с определения функции переставляющей элементы вектора.

```
# let permute_element vec n p =
  let aux = vec.(n) in vec.(n) <- vec.(p) ; vec.(p) <- aux ;;
val permute_element : 'a array -> int -> int -> unit = <fun>
```

Выбор правильной точки опоры важен для эффективности алгоритма, но мы ограничимся самым простым способом: вернем индекс первого элемента вектора.

```
# let choose_pivot vec start finish = start ;;
val choose_pivot : 'a -> 'b -> 'c -> 'b = <fun>
```

Напишем теперь желаемый алгоритм переставляющий элементы вектора вокруг выбранной точки.

- установить опорную точку в начало вектора
- i индекс второго элемента вектора
- j индекс последнего элемента вектора
- если элемент с индексом j больше чем значение в опорной точке, поменяем местами их значения и увеличим i на единицу, иначе уменьшим j на единицу
- до тех пор пока i меньше чем j повторить предыдущую операцию
- к этому этапу каждый элемент с индексом меньшим чем i (или j) меньше значения в опорной точке, а остальные элементы больше: если элемент с индексом i меньше чем опорное значение мы меняем их местами, иначе меняем с предыдущим элементом.

При реализации алгоритма мы естественно воспользуемся императивными управляющими структурами.

```
# let permute_pivot vec start finish ind_pivot =
  permute_element vec start ind_pivot ;
  let i = ref (start+1) and j = ref finish and pivot = vec.(start) in
  while !i < !j do
    if vec.(!j) >= pivot then decr j
    else
      begin
        permute_element vec !i !j ;
        incr i
      end
    done ;
    if vec.(!i) > pivot then decr i ;
    permute_element vec start !i ;
    !i
  ;;
val permute_pivot : 'a array -> int -> int -> int -> int = <fun>
```

Кроме эффекта над вектором, функция возвращает индекс опорной точки.

Нам остается лишь собрать воедино различные этапы и написать рекурсивный вызов для подвекторов.

```

# let rec quick vec start finish =
  if start < finish
  then
    let pivot = choose_pivot vec start finish in
    let place_pivot = permute_pivot vec start finish pivot in
    quick (quick vec start (place_pivot-1)) (place_pivot+1) finish
  else vec ;;
val quick : 'a array -> int -> int -> 'a array = <fun>

```

Здесь мы воспользовались двумя стилями. Выбранная опорная точка служит аргументом при перестановке вокруг нее и ее порядок в векторе после этой процедуры служит аргументом рекурсивного вызова. Зато, полученный после перестановки вектор мы получаем в результате побочного эффекта, а не как возвращенное значение функции `permute_pivot`. Тогда как функция `quick` возвращает вектор и сортировка подвекторов реализуется композицией рекурсивных вызовов.

Теперь главная функция:

```

# let quicksort tab = quick tab 0 ((Array.length tab)-1) ;;
val quicksort : 'a array -> 'a array = <fun>

```

Это полиморфная функция, так как отношение порядка < полиморфное.

```

# let t1 = [|4;8;1;12;7;3;1;9|] ;;
val t1 : int array = [|4; 8; 1; 12; 7; 3; 1; 9|]
# quicksort t1 ;;
- : int array = [|1; 1; 3; 4; 7; 8; 9; 12|]
# t1 ;;
- : int array = [|1; 1; 3; 4; 7; 8; 9; 12|]
# let t2 = ["the"; " little "; "cat"; "is"; "dead"] ;;
val t2 : string array = ["the"; " little "; "cat"; "is"; "dead"]
# quicksort t2 ;;
- : string array = ["cat"; "dead"; "is"; " little "; "the"]
# t2 ;;
- : string array = ["cat"; "dead"; "is"; " little "; "the"]

```

3.2.2 Общее использование или копии значений

До тех пор пока наши данные не изменяемые, нет необходимости знать используются они совместно или нет.

```

# let id x = x ;;

```

```

val id : 'a -> 'a = <fun>
# let a = [ 1; 2; 3 ] ;;
val a : int list = [1; 2; 3]
# let b = id a ;;
val b : int list = [1; 2; 3]

```

Является ли **b** копией **a** или же это один и тот же список не имеет значения, так как по любому это “неосязаемые” значения. Однако, если на место целых чисел, мы поместим изменяемые значения, необходимо будет знать скажется ли изменение одного значения на другое.

Реализация полиморфизма Objective CAML вызывает (causes) копию непосредственных значений (immediate values) и разделение (sharing) структурных значений. Хотя передача аргументов осуществляется копированием, в случае структурных значений передается лишь указатель. Как в случае с функцией **id**.

```

# let a = [| 1 ; 2 ; 3 |] ;;
val a : int array = [|1; 2; 3|]
# let b = id a ;;
val b : int array = [|1; 2; 3|]
# a.(1) <- 4 ;;
- : unit = ()
# a ;;
- : int array = [|1; 4; 3|]
# b ;;
- : int array = [|1; 4; 3|]

```

Здесь мы действительно имеем случай выбора стиля программирования, от которого зависит эффективность представления данных. С одной стороны, использование изменяемых значений позволяет немедленное манипулирование данными (без дополнительного выделения памяти). Однако это вынуждает, в некоторых случаях, делать копии там, где использование неизменяемого значения позволило бы разделение. Проиллюстрируем это двумя способами реализации списков.

```

# type 'a list_immutable = LNil | LCons of 'a * 'a list_immutable ;;
# type 'a list_mutable = LMNil | LMcons of 'a * 'a list_mutable ref ;;

```

Фиксированные списки строго эквивалентны спискам Objective CAML, тогда как изменяемые больше в стиле C, где ячейка состоит из значения и ссылки на следующую ячейку.

С фиксированными списками существует единственный способ реализовать конкатенацию и он вынуждает копирование структуры первого

списка, тогда как второй список может быть разделен с конечным списком.

```
# let rec concat l1 l2 = match l1 with
  | LNil -> l2
  | LIcons (a,l11) -> LIcons(a, (concat l11 l2)) ;;
val concat : 'a list_immutable -> 'a list_immutable -> 'a
list_immutable = <fun>

# let li1 = LIcons(1, LIcons(2, LNil))
  and li2 = LIcons(3, LIcons(4, LNil)) ;;
val li1 : int list_immutable = LIcons (1, LIcons (2, LNil))
val li2 : int list_immutable = LIcons (3, LIcons (4, LNil))
# let li3 = concat li1 li2 ;;
val li3 : int list_immutable =
  LIcons (1, LIcons (2, LIcons (3, LIcons (4, LNil))))
# li1==li3 ;;
- : bool = false
# let tLI l = match l with
  | LNil -> failwith "Liste vide"
  | LIcons(_,x) -> x ;;
val tLI : 'a list_immutable -> 'a list_immutable = <fun>
# tLI(tLI(li3)) == li2 ;;
- : bool = true
```

В этом примере мы видим что первые ячейки `li1` и `li3` различны, тогда как вторая часть `li3` есть именно `li2`.

С изменяемыми списками можно изменить аргументы (функция `concat_share`) или создать новое значение (функция `concat_copy`)

```
# let rec concat_copy l1 l2 = match l1 with
  | LNil -> l2
  | LMcons (x,l11) -> LMcons(x, ref (concat_copy !l11 l2)) ;;
val concat_copy : 'a list_mutable -> 'a list_mutable -> 'a list_mutable
= <fun>
```

Это решение, `concat_copy`, аналогично предыдущей функции `concat`. Вот второй вариант, с общим использованием.

```
# let concat_share l1 l2 =
  match l1 with
  | LNil -> l2
  | _ -> let rec set_last = function
```

```

        LMnil      -> failwith "concat_share : impossible case!!
        "
    | LMcons(_,l) -> if !=LMnil then l:=l2 else set_last !!
in
    set_last l1 ;
    l1 ;;
val concat_share : 'a list_mutable -> 'a list_mutable -> 'a
    list_mutable =
    <fun>

```

Конкатенация с общим использованием не нуждается в выделении памяти (мы не используем конструктор `LMcons`), мы лишь ограничиваемся тем что последняя ячейка первого списка теперь указывает на второй список. При этом, конкатенация способна изменить аргументы переданные функции.

```

# let lm1 = LMcons(1, ref (LMcons(2, ref LMnil)))
  and lm2 = LMcons(3, ref (LMcons(4, ref LMnil))) ;;
val lm1 : int list_mutable =
    LMcons (1, {contents=LMcons (2, {contents=LMnil}}))
val lm2 : int list_mutable =
    LMcons (3, {contents=LMcons (4, {contents=LMnil}}))
# let lm3 = concat_share lm1 lm2 ;;
val lm3 : int list_mutable =
    LMcons (1, {contents=LMcons (2, {contents=LMcons (...)}}))

```

Мы получили ожидаемый результат для `lm3`, однако значение `lm1` изменено.

```

# lm1 ;;
- : int list_mutable =
LMcons (1, {contents=LMcons (2, {contents=LMcons (...)}}))

```

Таким образом это может повлиять на оставшуюся часть программы.

3.2.3 Критерии выбора

В чисто функциональной программе побочных эффектов не существует, это свойство лишает нас операций ввода/вывода, исключений и изменяемых структур данных. Наше определение функционального стиля является менее ограниченным, то есть функция не изменяющая свое глобальное окружение может быть использована в функциональном стиле. Подобная функция может иметь локальные изменяемые значения (и значит

следовать императивному стилю), но не должна изменять ни глобальные переменные ни свои аргументы. С внешней стороны, такие функции можно рассматривать как “черный ящик”, чье поведение сравнимо с поведением чисто функциональной функции, с разницей что выполнение первой может быть приостановлен возбуждением исключения. В том же духе, изменяемое значение, которое больше не изменяется после инициализации, может быть использовано в функциональном стиле.

С другой стороны, программа написанная в императивном стиле, унаследует следующие достоинства Objective CAML: правильность статической типизации, автоматическое управление памятью, механизм исключений, параметризованный полиморфизм и вывод типов.

Выбор между стилем функциональным и императивным зависит от программного обеспечения которое вы хотите реализовать. Выбор может быть сделан в соответствии со следующими характеристиками:

выбор структур данных: от использования или нет изменяемых структур данных будет зависеть выбор стиля программирования. Действительно, функциональный стиль по своей природе не совместим с изменением значений. Однако, создание и просмотр значения не зависят от ее свойств. Таким образом мы возвращаемся к дискуссии “изменение на месте vs копия” в 3.2.2 на стр. 108, к которой мы вернемся для обсуждения критерия эффективности.

структура данных существует: если в программе необходимо менять изменяемые структуры данных (*modify mutable data structures*), то императивный стиль является единственно возможным. Однако, если необходимо лишь читать значения, то применение функционального стиля гарантирует целостность данных. Использование рекурсивных структур данных подразумевает рекурсивные функции, которые могут быть определены используя особенности того или иного стиля программирования. Однако в общем случае бывает проще истолковывать создание значения следуя рекуррентному определению. Этот подход более близок к функциональному стилю, чем повторяющаяся обработка этого значения рекурсией.

критерий эффективности: немедленное изменение без сомнения лучше чем создание значения. В случае когда эффективность кода является главенствующим критерием, чаша весов перевешивает в сторону императивного стиля. Однако отметим что совместное использование значений может оказаться нелегкой задачей и в конце концов более дорогостоящей, чем копирование значений с самого

начала. Чистая функциональность имеет определенную цену: частичное применение (partial application) и использование функций переданных в виде аргумента другой функции несет более серьезные накладные расходы в процессе выполнения программы, чем явное применение видимой функции. Стоит избегать использования этой функциональной особенности в тех местах, где критерий производительности является решающим.

критерий разработки: более высокий уровень абстракции функционального стиля программирования позволяет более быстрое написание компактного кода, содержащего меньше ошибок чем императивный вариант, который по своей природе является более “многословным”. Таким образом, функциональный стиль является выгодным при разработке значительных программ. Независимость функции к своему контексту окружения позволяет разделить код на более маленькие части и тем самым облегчить его проверку и читаемость. Более высокая модульность функционального стиля плюс возможность передавать функции (а значит и обработку) в аргумент другим функциям повышает вероятность повторного использования программ.

Эти несколько замечаний подтверждают тот факт, что часто смешанное использование обоих стилей является рациональным выбором. Функциональный стиль быстрее в разработке и обеспечивает более простую организацию программы, однако части кода где необходимо повысить скорость выполнения лучше написать в императивном стиле.

3.3 Смесь стилей

Как мы уже заметили, язык программирования имеющий функциональные и императивные особенности дает свободу выбора наиболее подходящего стиля для реализации того или иного алгоритма. Конечно, мы можем использовать оба аспекта Objective CAML совместно в одной и той же функции. Именно этим мы сейчас и займемся.

3.3.1 Замыкание и побочные эффекты

Обычно функция с побочным эффектом рассматривается как процедура и она возвращает значение () типа `unit`. Однако, иногда бывает полезно произвести побочный эффект внутри функции и вернуть определенное

значение. Мы уже использовали подобный коктейль стилей в функции `permute_pivot` в быстрой сортировке.

В следующем примере реализуем генератор символов, который создает новый символ при каждом вызове функции. Мы используем счетчик, значение которого увеличивается с каждым вызовом.

```
# let c = ref 0;;
val c : int ref = {contents=0}
# let reset_symb = function () -> c:=0 ;;
val reset_symb : unit -> unit = <fun>
# let new_symb = function s -> c:=!c+1 ; s^(string_of_int !c) ;;
val new_symb : string -> string = <fun>
# new_symb "VAR" ;;
- : string = "VAR1"
# new_symb "VAR" ;;
- : string = "VAR2"
# reset_symb () ;;
- : unit = ()
# new_symb "WAR" ;;
- : string = "WAR1"
# new_symb "WAR" ;;
- : string = "WAR2"
```

А теперь спрячем ссылку на `c` от всей программы следующим образом:

```
# let (reset_s , new_s) =
  let c = ref 0
  in let f1 () = c := 0
     and f2 s = c := !c+1 ; s^(string_of_int !c)
     in (f1,f2) ;;
val reset_s : unit -> unit = <fun>
val new_s : string -> string = <fun>
```

Таким образом мы объявили пару функций, разделяющие локальную (для декларации) переменную `c`. Использование обеих функций производит тот же результат что и раньше.

```
# new_s "VAR" ;;
- : string = "VAR1"
# new_s "VAR" ;;
- : string = "VAR2"
# reset_s();;
- : unit = ()
```

```
# new_s "WAR";
- : string = "WAR1"
# new_s "WAR";
- : string = "WAR2"
```

Этот пример иллюстрирует способ представления замыкания. Замыкание можно рассматривать как пару, состоящую из кода (то есть часть **function**) и локальное окружение, содержащее значения свободных переменных замыкания с другой стороны. На диаграмме 3.1 мы можем увидеть представление в памяти замыканий **reset_s** и **newt_s**.

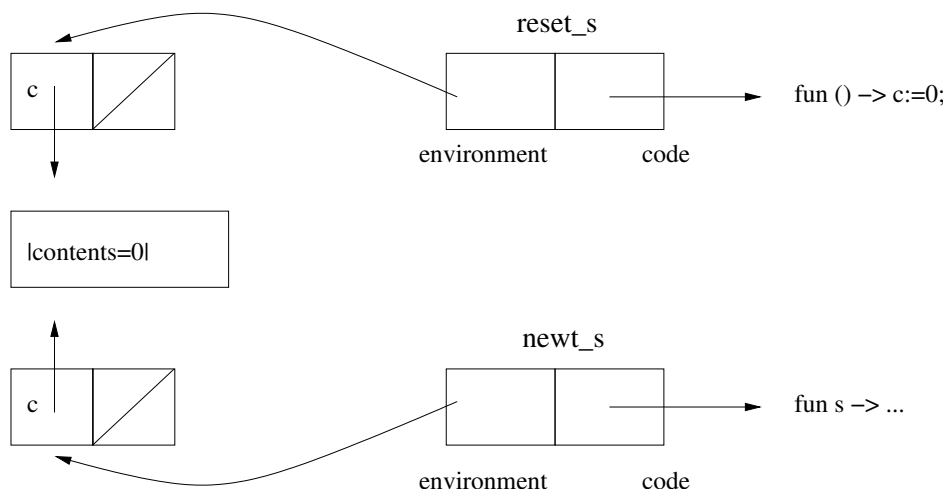


Рис. 3.1: Представление в памяти замыканий

Оба эти замыкания разделяют одно и то же окружение (значение **c**). Когда одно из них меняет ссылку **c**, то оно меняет содержимое памяти разделяемое с другим окружением.

3.3.2 Физическое изменение и исключения

Исключения позволяет отслеживать ситуацию, при которой дальнейшее продолжение программы невозможно. В подобном случае сборщик исключений позволит продолжить вычисление, зная что произошла ошибка. В момент возбуждения исключения может возникнуть проблема побочного эффекта с состоянием модифицируемых данных. Это состояние не может быть гарантировано если произошли физические изменения в ответвлении неудавшегося вычисления.

Определим функцию увеличения (**++**), с аналогичным результатом что и в **C**:

```
# let (++) x = x:=!x+1; x;;
val ++ : int ref -> int ref = <fun>
```

Следующий пример, иллюстрирует небольшой расчет в котором деление на ноль совпадает с побочным эффектом:

```
# let x = ref 2;;
val x : int ref = {contents=2}
(* 1 *)
# !((++) x) * (1/0) ;;
Uncaught exception: Division_by_zero
# x;;
- : int ref = {contents=2}
(* 2 *)
# (1/0) * !((++) x) ;;
Uncaught exception: Division_by_zero
# x;;
- : int ref = {contents=3}
```

Переменная `x` не изменяется во время вычисления выражения `(*1*)`, тогда как она изменяется во время `(*2*)`. Если заранее не сохранить начальные значения, конструкция `try .. with ..` не должна (в части `with ..`) зависеть от изменяемых переменных, которые участвуют в вычислении возбуждившем исключение.

3.3.3 Изменяемые функциональные структуры данных

В функциональном программировании программа, как функциональное выражение, является в то же время данными — чтобы уяснить этот момент напишем список ассоциированных значений в виде функционального выражения. Этот список ассоциаций `('a * 'b) list` можно рассматривать как частичную функцию из `'a` (множество ключей) в `'b` (множество соответствующих значений). Другими словами, функция `'a -> 'b`.

Пустой список это неопределенная функция, которую мы будем моделировать возбуждением исключения:

```
# let nil_assoc = function x -> raise Not_found ;;
val nil_assoc : 'a -> 'b = <fun>
```

Теперь напишем функцию `add_assoc` добавляющую элемент в список или другими словами расширим функцию новыми значениями.

```
# let add_assoc (k,v) l = function x -> if x = k then v else l x ;;
val add_assoc : 'a * 'b -> ('a -> 'b) -> 'a -> 'b = <fun>
# let l = add_assoc ('1', 1) (add_assoc ('2', 2) nil_assoc) ;;
val l : char -> int = <fun>
# l '2' ;;
- : int = 2
# l 'x' ;;
Uncaught exception: Not_found
```

Перепишем функцию `mem_assoc`:

```
# let mem_assoc k l = try (l k) ; true with Not_found -> false ;;
val mem_assoc : 'a -> ('a -> 'b) -> bool = <fun>
# mem_assoc '2' l ;;
- : bool = true
# mem_assoc 'x' l ;;
- : bool = false
```

Однако, написание функции удаляющей элементы из списка, не совсем простое дело. У нас больше нет доступа к значениям входящим в замыкание. Для этого мы спрячем старое значение возбуждив исключения `Not_found`.

```
# let rem_assoc k l = function x -> if x=k then raise Not_found else
  l x ;;
val rem_assoc : 'a -> ('a -> 'b) -> 'a -> 'b = <fun>
# let l = rem_assoc '2' l ;;
val l : char -> int = <fun>
# l '2' ;;
Uncaught exception: Not_found
```

Естественно, можно воспользоваться ссылками и побочным эффектом для использования таких значений, однако существует несколько предостережений.

```
# let add_assoc_again (k,v) l = l := (function x -> if x=k then v
  else !l x) ;;
val add_assoc_again : 'a * 'b -> ('a -> 'b) ref -> unit = <fun>
```

Мы получили функцию `l` которая указывает сама на саму себя и значит будет зацикливаться. Этот досадный побочный эффект есть результат того что разыменованное `!l` находится внутри замыкания `function x ->`. Значение `!l` вычислено при выполнении, а не компиляции. В этот момент, `l` указывает на значение измененное `add_assoc`. Необходимо исправить наше определение используя замыкание полученное определением `add_assoc`:

```

# let add_assoc_again (k, v) l = l := add_assoc (k, v) !l ;;
val add_assoc_again : 'a * 'b -> ('a -> 'b) ref -> unit = <fun>
# let l = ref nil_assoc ;;
val l : ('_a -> '_b) ref = {contents=<fun>}
# add_assoc_again ('1',1) l ;;
- : unit = ()
# add_assoc_again ('2',2) l ;;
- : unit = ()
# ! '1' ;;
- : int = 1
# ! 'x' ;;
Uncaught exception: Not_found

```

3.3.4 Пассивные изменяемые структуры

Смесь императивных особенностей с функциональным языком образует хорошие средства реализации языков программирования. В данном параграфе мы проиллюстрируем эту особенность в реализации структур данных с отсроченным вычислением. Такая структура данных не вычисляется полностью, вычисление продвигается в зависимости от использования структуры.

Отложенное вычисление, часто используемое в чистых функциональных языках, можно моделировать использованием функциональных значений (возможно, изменяемых). Выгода от использования данных с отложенным выполнением двойная; во первых вычисляется лишь то что необходимо для расчета, во вторых использование потенциально бесконечных данных.

Определим тип `vm` элементы которого либо уже вычисленное значение (конструктор `Imm`) либо значение которое будет вычислено (конструктор `Deferred`).

```

# type 'a v =
  Imm of 'a
  | Deferred of (unit -> 'a);;
# type 'a vm = {mutable c : 'a v };;

```

Откладывание вычислений может быть получено инкапсуляцией в замыкание. Функция вычисляющая такое значение должна или вернуть значение если оно уже вычислено или, в противном случае, вычислить и сохранить результат.

```

# let eval e = match e.c with

```

```

Imm a -> a
| Deferred f -> let u = f () in e.c <- Imm u ; u ;;
val eval : 'a vm -> 'a = <fun>

```

Операции задержки и активации вычисления также называют замораживанием и размораживанием значения.

Напишем условный контроль в виде функции:

```

# let if_deferred c e1 e2 =
  if eval c then eval e1 else eval e2 ;;
val if_deferred : bool vm -> 'a vm -> 'a vm -> 'a = <fun>

```

А теперь воспользуемся этим в рекурсивной функции подсчета факториала.

```

# let rec facr n =
  si_ret {c=Ret(fun () -> n = 0)}
        {c=Ret(fun () -> 1)}
        {c=Ret(fun () -> n*(facr(n-1)))} ;;
val facr : int -> int = <fun>
# facr 5;;
- : int = 120

```

Заметим, что классический `if` не может быть записан в виде функции. Действительно, определим функцию `if_function` следующим образом:

```

# let if_function c e1 e2 = if c then e1 else e2 ;;
val if_function : bool -> 'a -> 'a -> 'a = <fun>

```

Дело в том что все три аргумента вычисляются, это приводит к зацикливанию, так как рекурсивный вызов `fact(n-1)` всегда вычисляется, даже в случае когда `n=0`.

```

# let rec fact n = if_function (n=0) 1 (n*fact(n-1)) ;;
val fact : int -> int = <fun>
# fact 5 ;;
Stack overflow during evaluation (looping recursion?).

```

Модуль Lazy

Трудности во внедрении замороженных значений происходят от конструкции выражений с отложенным вычислением в контексте немедленного вычисления Objective CAML. Мы это увидели при попытке переопределить условное выражение. Нельзя написать функцию замораживающую значение при конструкции объекта типа `vm`.

```
# let gele e = { c = Ret (fun () -> e) };;
val gele : 'a -> 'a vm = <fun>
```

Эта функция следует стратегии вычисления Objective CAML, то есть выражение **e** вычисляется перед тем как создать замыкание **fun () -> e**. Проиллюстрируем это в следующем примере:

```
# freeze (print_string "trace"; print_newline(); 4*5);;
trace
- : int vm = {c=Deferred <fun>}
```

По этой причине была введена следующая синтаксическая форма:
Синтаксис

lazy expr

Warning

Это особенность является расширением языка и может измениться в следующих версиях.

Когда к выражению применяется ключевое слово **lazy** то создается значение особого типа, который определен в модуле **Lazy**:

```
# let x = lazy (print_string "Hello"; 3*4) ;;
val x : int Lazy.status ref = {contents=Lazy.Delayed <fun>}
```

Выражение (**print_string** “Hello”) не вычислено, так как не было никакого вывода на экран. При помощи функции **Lazy.force** мы можем вынудить вычисление выражения.

```
# Lazy.force x ;;
Hello- : int = 12
```

Тут мы замечаем, что значение **x** изменилось:

```
# x ;;
- : int Lazy.t = {contents=Lazy.Value 12}
```

Теперь это значение замороженного выражения, в данном случае 12.

Новый вызов функции **force** просто возвращает вычисленное значение:

```
# Lazy.force x ;;
- : int = 12
```

Строка “Hello” больше не выводится на экран.

“Бесконечные” структуры данных

Другой интерес использования отложенного вычисления состоит в возможности построения потенциально бесконечных структур данных, как на пример множество натуральных чисел. Вместо того чтобы конструировать каждое число, мы определим лишь первый элемент и способ получения следующего.

Определим настраиваемую (generic) структуру `'a enum` с помощью которой мы будем определять элементы множества.

```
# type 'a enum = { mutable i : 'a; f : 'a -> 'a } ;;
type 'a enum = { mutable i: 'a; f: 'a -> 'a }
# let next e = let x = e.i in e.i <- (e.f e.i) ; x ;;
val next : 'a enum -> 'a = <fun>
```

Для того, чтобы получить множество натуральных достаточно конкретизировать (instanciating) поля этой структуры.

```
# let nat = { i=0; f=fun x -> x + 1 };;
val nat : int enum = {i=0; f=<fun>}
# next nat;;
- : int = 0
# next nat;;
- : int = 1
# next nat;;
- : int = 2
```

Другой пример — ряд чисел Фибоначчи, который определен как:

$$\begin{cases} u_0 = 1 \\ u_1 = 1 \\ u_{n+2} = u_n + u_{n+1}; \end{cases}$$

Функция, вычисляющая текущее значение, должна использовать значения u_{n-1} и u_{n-2} . Для этого воспользуемся состоянием `s` в следующем замыкании.

```
# let fib = let fx = let c = ref 0 in fun v -> let r = !c + v in c:=v ; r
              in { i=1 ; f=fx } ;;
val fib : int enum = {i=1; f=<fun>}
# for i=0 to 10 do print_int (next fib); print_string " " done ;;
1 1 2 3 5 8 13 21 34 55 89 - : unit = ()
```

3.4 Поток данных

Потоки это потенциально бесконечная последовательность данных определенного рода. Вычисление части потока выполняется по необходимости для текущего вычисления — пассивные структуры данных.

stream абстрактный тип данных, реализация которого неизвестна. Мы манипулируем объектами этого типа при помощи функций конструкции и деструкции. Для удобства пользователя, Objective CAML предоставляет пользователю простые синтаксические конструкции для создания и доступа к элементам потока.

Warning

Это особенность является расширением языка и может измениться в следующих версиях.

3.4.1 Конструкция

Упрощенный синтаксис конструкции потоков похож на синтаксис конструкции списков или векторов. Пустой поток создается следующим способом:

```
# [< >] ;;
- : 'a Stream.t = <abstr>
```

Другой способ конструкции потока состоит в перечислении элементов этого потока, где перед каждым из них ставится апостроф.

```
# [< '0; '2; '4 >] ;;
- : int Stream.t = <abstr>
```

Выражения, перед которыми не стоит апостроф, рассматриваются как под-потоки.

```
# [< '0; [< '1; '2; '3 >]; '4 >] ;;
- : int Stream.t = <abstr>
# let s1 = [< '1; '2; '3 >] in [< s1; '4 >] ;;
- : int Stream.t = <abstr>
# let concat_stream a b = [< a ; b >] ;;
val concat_stream : 'a Stream.t -> 'a Stream.t -> 'a Stream.t = <fun>
# concat_stream [< "if"; "c"; "then"; "1" >] [< "else"; "2" >] ;;
- : string Stream.t = <abstr>
```

Остальные функции сгруппированы в модуле `Stream`. На пример, функции `of_channel` и `of_string` возвращают поток состоящий из символов полученных из входного потока или строки символов.

```
# Stream.of_channel ;;
- : in_channel -> char Stream.t = <fun>
# Stream.of_string ;;
- : string -> char Stream.t = <fun>
```

Отложенное вычисление потоков позволяет использовать бесконечные структуры данных, как это было в случае с типом `'a enum` на стр. 121. Определим поток натуральных чисел при помощи первого элемента и функции вычисляющей поток из следующих элементов.

```
# let rec nat_stream n = [< 'n ; nat_stream (n+1) >] ;;
val nat_stream : int -> int Stream.t = <fun>
# let nat = nat_stream 0 ;;
val nat : int Stream.t = <abstr>
```

3.4.2 Деструкция и сопоставление потока

Элементарная операция `next` одновременно вычисляет, возвращает и извлекает первый элемент потока.

```
# for i=0 to 10 do
  print_int (Stream.next nat) ;
  print_string " "
done ;;
0 1 2 3 4 5 6 7 8 9 10 - : unit = ()
# Stream.next nat ;;
- : int = 11
```

Когда данные в потоке закончились возбуждается исключение.

```
# Stream.next [< >] ;;
Uncaught exception: Stream.Failure
```

Для использования потоков Objective CAML предоставляет специальное сопоставление для потоков — уничтожающее сопоставление (`destructive matching`). Сопоставляемое значение вычисляется и удаляется из потока. Понятие исчерпываемости сопоставления (`exhaustive match`) не существует для потоков, так как мы используем пассивные и потенциально бесконечные структуры данных. Синтаксис сопоставления следующий:

Синтаксис

```
match expr with parser [< 'p1 ...>] -> expr1 | ...
```

Функция **next** может быть переписана в следующей форме:

```
# let next s = match s with parser [< 'x >] -> x ;;
val next : 'a Stream.t -> 'a = <fun>
# next nat;;
- : int = 12
```

Заметьте, что перечисление чисел началось с того места где мы остановились.

Существует другая форма синтаксиса для фильтрации функционального параметра типа **Stream.t**.

Синтаксис

```
parser p -> ...
```

Перепишем функцию **next** используя новый синтаксис.

```
# let next = parser [<'x>] -> x ;;
val next : 'a Stream.t -> 'a = <fun>
# next nat ;;
- : int = 13
```

Мы можем сопоставлять пустой поток, но необходимо помнить о следующем: образец потока [**<>**] сопоставляется с каким угодно потоком. То есть, поток **s** всегда равен потоку [**<** [**<>**]; **s** **>**]. Поэтому нужно поменять обычный порядок сопоставления.

```
# let rec it_stream f s =
  match s with parser
    | [< 'x ; ss >] -> f x ; it_stream f ss
    | [<>] -> () ;;
val it_stream : ('a -> 'b) -> 'a Stream.t -> unit = <fun>
# let print_int1 n = print_int n ; print_string" " ;;
val print_int1 : int -> unit = <fun>
# it_stream print_int1 [<'1; '2; '3>] ;;
1 2 3 - : unit = ()
```

Используя тот факт что сопоставление уничтожающее, перепишем предыдущую функцию.

```
# let rec it_stream f s =
  match s with parser
    | [< 'x >] -> f x ; it_stream f s
    | [<>] -> () ;;
val it_stream : ('a -> 'b) -> 'a Stream.t -> unit = <fun>
```

```
# it_stream print_int1 [<'1; '2; '3>] ;;
1 2 3 - : unit = ()
```

Несмотря на то что потоки пассивные, они добровольно и с “восторгом” отдадут свой первый элемент, который после этого будет утерян для потока. Эта особенность отображается на сопоставлении. Следующая функция есть попытка (обреченная на неудачу) вывода на экран двух чисел из потока, в конце потока может остаться один элемент.

```
# let print_int2 n1
  n2 =
    print_string "(" ; print_int n1 ; print_string "," ;
    print_int n2 ; print_string ")" ;;
val print_int2 : int -> int -> unit = <fun>
# let rec print_stream s =
  match s with parser
  | [<'x; 'y >] -> print_int2 x y; print_stream s
  | [<'z >] -> print_int1 z; print_stream s
  | [<>] -> print_newline() ;;
val print_stream : int Stream.t -> unit = <fun>
# print_stream [<'1; '2; '3>];;
(1,2)Uncaught exception: Stream.Error("")
```

Два первых элемента потока были выведены на экран без проблем, однако во время вычисления рекурсивного вызова (`print_stream [< 3 >]`) образец обнаружил `x`, который был “употреблен”, но для `y` в потоке ничего нет. Это и привело к ошибке. Дело в том что второй образец абсолютно бесполезный, если поток не пустой то первый образец всегда совпадет.

Для того, чтобы получить ожидаемый результат необходимо упорядочить сопоставление.

```
# let rec print_stream s =
  match s with parser
  | [<'x >]
    -> (match s with parser
        | [<'y >] -> print_int2 x y; print_stream s
        | [<>] -> print_int1 x; print_stream s)
  | [<>] -> print_newline() ;;
val print_stream : int Stream.t -> unit = <fun>
# print_stream [<'1; '2; '3>];;
(1,2)3
- : unit = ()
```

Если сопоставление не срабатывает на первом элементе образца, то фильтр работает как обычно.

```
# let rec print_stream s =
  match s with parser
  | [< '1; 'y >] -> print_int2 1 y; print_stream s
  | [< 'z >] -> print_int1 z; print_stream s
  | [<>] -> print_newline() ;;
val print_stream : int Stream.t -> unit = <fun>
# print_stream [<'1; '2; '3>] ;;
(1,2)3
- : unit = ()
```

Пределы сопоставления

Из-за своего свойства уничтожения сопоставление потоков отличается от сопоставления типов сумма. Давайте рассмотрим на сколько глубоко они отличаются.

Вот простейшая функция складывающая два элемента потока.

```
# let rec sum s =
  match s with parser
  | [< 'n; ss >] -> n+(sum ss)
  | [<>] -> 0 ;;
val sum : int Stream.t -> int = <fun>
# sum [<'1; '2; '3; '4>] ;;
- : int = 10
```

Однако, мы можем поглотить поток “изнутри” придав имя полученному результату.

```
# let rec sum s =
  match s with parser
  | [< 'n; r = sum >] -> n+r
  | [<>] -> 0 ;;
val sum : int Stream.t -> int = <fun>
# sum [<'1; '2; '3; '4>] ;;
- : int = 10
```

В главе 10, посвященной лексическому и синтаксическому анализу, мы рассмотрим другие примеры использования потоков. В частности, мы увидим как “поглощение” потока изнутри можно с выгодой использовать.

3.5 Резюме

В этой главе мы сравнили функциональный и императивный стили программирования. Основные различия состоят в контроле выполнения (неявный в функциональном и явный в императивном стиле) и представлении в памяти данных (явное разделение или копирования в императивном стиле, не имеющее такой важности в функциональном). Реализация алгоритмов в функциональном и императивном стилях подразумевает эти различия. Выбор между обоими стилями на самом деле приводит к их одновременному использованию. Это позволяет явно выразить представление замыкания, оптимизировать критические части программы и создать изменяемые функциональные данные. Физическое изменение значений в окружении замыкания помогает нам лучше понять что такое функциональное значение. Одновременное использование обоих стилей предоставляет мощные средства для реализации. Мы воспользовались этим при создании потенциально бесконечных данных.

Глава 4

Графический интерфейс

Введение

В этой главе представлена библиотека Graphics, она входит является частью дистрибутива языка Objective CAML. Эта библиотека одинаково работает в графических интерфейсах основных платформ: Windows, MacOS, Unix с оболочкой X-Windows. С помощью Graphics мы можем реализовать графические рисунки с текстом, картинками, управлять различными базовыми событиями как нажатие на кнопку мыши или клавиатуры.

Модель программирования графических рисунков — “модель художника”: последний нарисованный слой стирает предыдущий. Это императивная модель, графическое окно в ней представляется как вектор пикселей, которые физически изменяются различными графическими функциями. Взаимодействие с мышью и клавиатурой подчиняется модели программирования событиями: главная функция программы это бесконечный цикл, в котором мы ожидаем действие пользователя.

Несмотря на простоту библиотеки Graphics, она вполне достаточна для введения концепций программирования графических интерфейсов с одной стороны и с другой стороны содержит базовые элементы простые в использовании программистом, при помощи которых мы можем реализовать более сложные интерфейсы.

План главы

В первом разделе мы объясним как пользоваться этой библиотекой на различных системах. Во втором изучим основы графического программирования: точка, рисунок, заполнение, цвет, bitmap. Для того чтобы

проиллюстрировать эти концепции, в третьем разделе опишем и реализуем функции рисования “блоков” (boxes). В четвертом разделе увидим анимацию графических объектов и их взаимодействие с фоном экрана или другими анимационными объектами. В пятом разделе представлен стиль программирования событиями, а так же скелет любого графического приложения. И наконец, в последнем разделе мы воспользуемся библиотекой Graphics для реализации интерфейса калькулятора приведенного на стр. 95.

4.1 Использование библиотеки Graphics

Использование библиотеки зависит от операционной системы и способа компиляции. Здесь мы рассмотрим только программы используемые в интерактивном цикле Objective CAML. В операционных системах Windows и MacOS интерактивное рабочее окружение само загружает библиотеку. Для систем Unix необходимо создать новый toplevel¹, который зависит от того где находится библиотека X11. Если она находится в одном из каталогов по умолчанию, где ищутся библиотеки C, то командная строка будет выглядеть так:

```
ocamlmktop -custom -o mytoplevel graphics.cma -cclib -lX11
```

Этим мы создаем команду mytoplevel с включенной библиотекой X11. Запуск этой команды осуществляется так:

```
./mytoplevel
```

Если же библиотека расположена в другом каталоге, как в Linux, необходимо его явно указать:

```
ocamlmktop -custom -o montoplevel graphics.cma -cclib \  
-L/usr/X11/lib -cclib -lX11
```

В этом примере файл libX11.a ищется в каталоге /usr/X11/lib.

Более подробный пример команды ocamlmktop приведен в главе 6.

4.2 Основные понятия

Программирование графических интерфейсов изначально связано с развитием аппаратных средств, в частности мониторов и графических карт. Для того чтобы изображение имело хорошее качество, необходимо чтобы оно регулярно и часто обновлялось (перерисовывалось), как в кино. Для

¹в оригинале на английском, прим. пер.

рисования на экране существует два основных метода: либо используя список видимых сегментов, при этом только нужная часть изображения рисуется на экране, либо рисуя все точки экрана (экран bitmap). На обычных компьютерах используется последний способ.

Экран bitmap можно рассматривать как прямоугольник точек, называемые пиксел от английского picture element. Они являются базовыми элементами изображения. Высота и ширина экрана называется разрешением основного bitmap. Размер этого bitmap зависит от памяти занимаемой пикселом. Для черно-белого изображения пиксел может занимать один бит. Для цветных или серых изображений размер пиксела зависит от числа оттенков ассоциированных пикселу. Для bitmap 320x640 пикселей по 256 цветов в каждом, нам понадобится 8 битов на каждый пиксел, соответственно размер видео-памяти должен быть 480×640 байтов = 307200 байтов $\simeq 300$ ко. Это разрешения до сих пор используется некоторыми программами в MS-DOS.

В следующем списке приведены базовые операции над bitmap из библиотеки Graphics:

- закрашивание пиксела
- рисование сегмента
- рисование формы: прямоугольник, эллипс
- заливка замкнутой формы: прямоугольник, эллипс, многоугольник
- рисование текста
- манипуляция и перемещение части изображения

Каждая из этих операций использует координаты точки изображения для задания места рисования. Некоторые характеристики графических операций образуют графический контекст: толщина линии, соединение линий, выбор шрифта и его размер, стиль заливки. Графическая операция всегда выполняется в определенном контексте и ее результат зависит от этого контекста. Графический контекст библиотеки Graphics содержит лишь текущие точку, цвет, шрифт и его размер.

4.3 Графический вывод

При графическом выводе следующие элементы являются базовыми: ориентир (начальная точка?) (reference point), графический контекст, цвет, изображение, заполнение замкнутых фигур, тексты и bitmap-ы.

4.3.1 Ориентир и графический контекст

Библиотека Graphics управляет единственным и главным окном. Координаты ориентира окна могут меняться от нижней левой точки (0,0) до верхнего правого угла. Вот основные операции над этим окном:

- `open_graph`, тип которой `string -> unit`, открывает окно
- `close_graph`, тип которой `unit -> unit`, закрывает
- `clear_graph`, тип которой `unit -> unit`, и очищает

Размер окна определяется функциями `size_x` и `size_y`.

Строка, которую мы передаем функции `open_graph`, зависит от графической оболочки машины где запускается программа, то есть она платформенно-зависимая. Если строка пустая, то полученное окно будет иметь свойства по умолчанию. Мы можем явно указать размер окна; в X-Windows строка “ 200x300” создаст окно размером 200 пискелей по ширине и 300 по высоте. Заметьте, пробел — первый символ строки “ 200x300”, обязателен.

В графическом контексте есть несколько компонентов которые можно просмотреть/изменить:

```

текущая точка:  current_point : unit -> int * int
                  moveto : int -> int -> unit
текущий цвет :  set_color : color -> unit
ширина текущей линии : set_line_width : int -> unit
текущий шрифт :  set_font : string -> unit
размер этого шрифта : set_text_size : int -> unit
```

4.3.2 Цвета

Цвет представлен тремя байтами, каждый из которых хранит яркость основного цвета в модели RGB (red, green, blue) в интервале от 0 до 255. Функция `rgb` (с типом `int -> int -> int -> color`) создает цвет с тремя указанными значениями. Если все три значения равны, то мы получим серый цвет более или менее светлый, в зависимости от значений. Черный соответствует минимуму яркости для каждой компоненты цвета `rgb(0,0,0)`, белый цвет — `rgb(255,255,255)`. Есть некоторые предопределенные цвета: `black`, `white`, `red`, `green`, `blue`, `yellow`, `cyan`, `magenta`.

Переменные `foreground` и `background` соответствуют текущему цвету и цвету фона. При стирании экрана, осуществляется заполнение цветом фона.

Цвет (значение типа `color`) это на самом деле целое число, которое мы можем разбить на три части (`from_rgb`) или инвертировать (`inv_color`).

```
(* color == R * 256 * 256 + G * 256 + B *)
# let from_rgb (c : Graphics.color) =
  let r = c / 65536 and g = c / 256 mod 256 and b = c mod 256
  in (r,g,b) ;;
val from_rgb : Graphics.color -> int * int * int = <fun>
# let inv_color (c : Graphics.color) =
  let (r,g,b) = from_rgb c
  in Graphics.rgb (255-r) (255-g) (255-b) ;;
val inv_color : Graphics.color -> Graphics.color = <fun>
```

Функция `point_color`, типа `int -> int -> color`, возвращает цвет точки, координаты которой указаны на входе.

4.3.3 Рисунок и заливка

Функция рисования выводит линию на экран, при этом для толщины и цвета используются текущие значения. Функция заливки закрашивает замкнутую форму текущим цветом. Различные функции рисования и заливки приведены в таблице 4.1.

рисунок	заполнение	тип
<code>plot</code>		<code>int -> int -> unit</code>
<code>lineto</code>		<code>int -> int -> unit</code>
	<code>fill_rect</code>	<code>int -> int -> int -> int -> unit</code>
	<code>fill_poly</code>	<code>(int * int) array -> unit</code>
<code>draw_arc</code>	<code>fill_arc</code>	<code>int -> int -> int -> int -> int -> unit</code>
<code>draw_ellipse</code>	<code>fill_ellipse</code>	<code>int -> int -> int -> int -> unit</code>
<code>draw_circle</code>	<code>fill_circle</code>	<code>int -> int -> int -> unit</code>

Таблица 4.1: Функции рисования и заливки

Заметьте, что функция `lineto` имеет побочный эффект: она изменяет положение текущей точки.

Рисование многоугольников В следующем примере, мы добавим некоторые функции рисования, которые не существуют в библиотеке. Многоугольник определяется вектором его вершин.

```
# let draw_rect x0 y0 w h =
  let (a,b) = Graphics.current_point()
```

```

and x1 = x0+w and y1 = y0+h
in
  Graphics.moveto x0 y0;
  Graphics.lineto x0 y1; Graphics.lineto x1 y1;
  Graphics.lineto x1 y0; Graphics.lineto x0 y0;
  Graphics.moveto a b;;
val draw_rect : int -> int -> int -> int -> unit = <fun>

# let draw_poly r =
  let (a,b) = Graphics.current_point () in
  let (x0,y0) = r.(0) in Graphics.moveto x0 y0;
  for i = 1 to (Array.length r)-1 do
    let (x,y) = r.(i) in Graphics.lineto x y
  done;
  Graphics.lineto x0 y0;
  Graphics.moveto a b;;
val draw_poly : (int * int) array -> unit = <fun>

```

Обратите внимание на то что эти функции берут такие же аргументы что и существующие функции заливки. Так они не изменяют состояние текущей точки.

Модель художника Следующий пример иллюстрирует сеть эстафетного кольца (token ring) (рис. 4.1). Каждый компьютер представлен в виде небольшого круга, все компьютеры соединены между собой и сеть образует кольцо. Текущее положение маркера в сети изображено черным кругом.

Функция net_points создает координаты всех компьютеров сети, эти данные будут храниться в векторе.

```

# let pi = 3.1415927;;
val pi : float = 3.1415927
# let net_points (x,y) l n =
  let a = 2. *. pi /. (float n) in
  let rec aux (xa,ya) i =
    if i > n then []
  else
    let na = (float i) *. a in
    let x1 = xa + (int_of_float (cos(na) *. l))
    and y1 = ya + (int_of_float (sin(na) *. l)) in
    let np = (x1,y1) in
    np::(aux np (i+1))

```

```

    in Array.of_list (aux (x,y) 1) ;;
val net_points : int * int -> float -> int -> (int * int) array = <fun
>

```

Функция `draw_net` выводит соединения, компьютеры и маркер.

```

# let draw_net (x,y) l n sc st =
  let r = net_points (x,y) l n in
  draw_poly r;
  let draw_machine (x,y) =
    Graphics.set_color Graphics.background;
    Graphics.fill_circle x y sc;
    Graphics.set_color Graphics.foreground;
    Graphics.draw_circle x y sc
  in
    Array.iter draw_machine r;
    Graphics.fill_circle x y st ;;
val draw_net : int * int -> float -> int -> int -> int -> unit = <
fun>

```

При следующем вызове получим левую картинку на рисунке 4.1.

```

# draw_net (140,20) 60.0 10 10 3;;
- : unit = ()

# save_screen "IMAGES/tokenring.caa";;
- : unit = ()

```

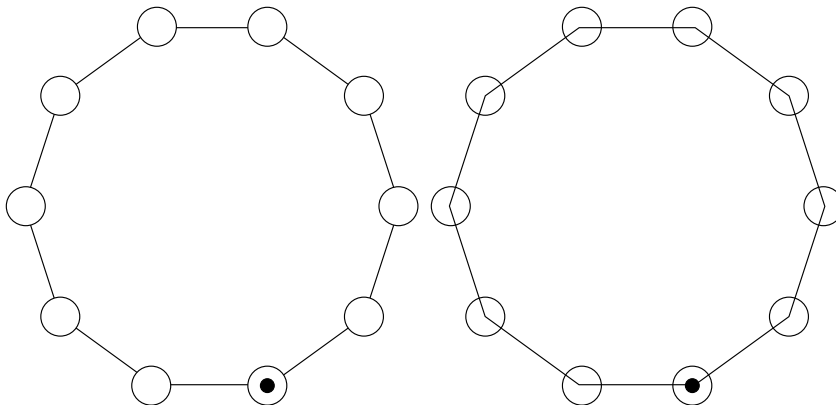


Рис. 4.1: Сеть эстафетное кольцо

Заметим что порядок вывода на экран важен — сначала соединения, а затем узлы. Изображение узлов сети скрывает (в тексте стирает, прим.

пер.) часть отрезков соединений. Таким образом нет надобности вычислять точки пересечения отрезков и кругов. На правой картинке рисунка 4.1 результат обратного порядка вывода на экран, где мы видим пересечения отрезков внутри кругов, представляющих узлы.

4.3.4 Текст

Две следующие функции вывода текста на экран чрезвычайно просты: `draw_char` (с типом `char -> unit`) и `draw_string` (с типом `string -> unit`) выводят на экран один символ и строку соответственно. Вывод осуществляется на текущую позицию экрана, также эти обе функции учитывают значение текущего шрифта и его размер.

Замечание

Вывод текста на экран может зависеть от графической оболочки.

Для переданной в аргументе строки, функция `text_size` возвращает пару целых чисел, которые соответствуют размерам выведенной строки с текущим шрифтом и размером.

Вертикальный вывод текста В следующем примере функция `draw_string_v` вертикально выводит на экран строку начиная с текущей позиции. Результат изображен на рисунке 4.2, каждая буква выводится отдельно, меняя вертикальные координаты.

```
# let draw_string_v s =
  let (xi, yi) = Graphics.current_point()
  and l = String.length s
  and (_, h) = Graphics.text_size s
  in
    Graphics.draw_char s.[0];
    for i=1 to l-1 do
      let (_, b) = Graphics.current_point()
      in Graphics.moveto xi (b-h);
        Graphics.draw_char s.[i]
    done;
    let (a, _) = Graphics.current_point() in Graphics.moveto a yi;;
val draw_string_v : string -> unit = <fun>
```

Эта функция изменяет текущую позицию, после вывода позиция перемещается на расстояние равное ширине символа.

Следующая программа выводит легенду вдоль координатных осей (рис. 4.2).

```
#
Graphics.moveto 0 150 ; Graphics.lineto 300 150 ;
Graphics.moveto 2 130 ; Graphics.draw_string "absciss" ;
Graphics.moveto 150 0 ; Graphics.lineto 150 300 ;
Graphics.moveto 135 280 ; draw_string_v "ordinate" ;;
- : unit = ()
```

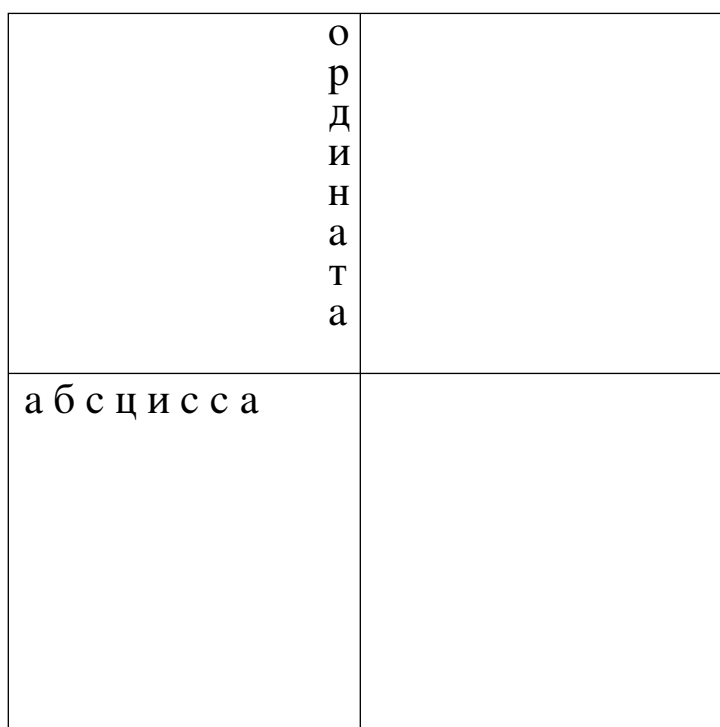


Рис. 4.2: Легенда координатных осей

Для того чтобы вывести вертикальный текст, необходимо помнить что функция `draw_string_v` изменяет текущую позицию. Для этого определим функцию `draw_text_v` с двумя аргументами: расстояние между столбцами и список слов.

```
# let draw_text_v n l =
  let f s = let (a,b) = Graphics.current_point()
             in draw_string_v s ;
               Graphics.moveto (a+n) b
  in List.iter f l ;;
```

```
val draw_text_v : int -> string list -> unit = <fun>
```

Если необходимо реализовать другие манипуляции с текстом, такие, как вращение, мы должны использовать bitmap каждой буквы и произвести вращение всех пикселей.

4.3.5 Bitmaps

Bitmap может быть представлен либо как матрица цветов (color array array), либо как значение абстрактного типа² image из библиотеки Graphics. Имена и типы функций приведены в таблице 4.2.

fonction	type
make_image	color array array -> image
dump_image	image -> color array array
draw_image	image -> int -> int -> unit
get_image	int -> int -> int -> int -> image
blit_image	image -> int -> int -> unit
create_image	int -> int -> image

Таблица 4.2: Функции манипуляции bitmaps

Функции make_image и dump_image конвертируют из одного типа в другой — image и color array array. Функция draw_image выводит на экран bitmap начиная с координат его нижнего левого угла.

При помощи функции get_image мы можем захватить прямоугольную часть экрана и создать таким образом изображение, для этого необходимо указать нижнюю левую и правую верхнюю углы зоны захвата. Функция blit_image захватывает экран часть экрана и сохраняет изображение которое мы передали в виде аргумента (тип image), второй аргумент указывает нижний левый угол части экрана, которую мы желаем захватить. Размер захватываемой части зависит от размеров изображения, переданного в аргументе. Функция create_image инициализирует изображение, для этого необходимо указать его будущий размер. Позднее, это изображение мы можем изменить функцией blit_image.

Предопределенный цвет transp позволяет сделать точки изображения прозрачными. Это позволяет нам вывести изображение лишь на части прямоугольника, прозрачные точки не изменяют начальный экран.

²абстрактным называется тип представление которого не известно. Объявление подобных типов рассматривается в главе 13

Поляризация jussieu ³

В следующем примере мы инвертируем цвет точек bitmap-а, для чего будем использовать функцию, которая была представлена на 132, для каждого пиксела bitmap-а.

```
# let inv_image i =
  let inv_vec = Array.map (fun c -> inv_color c) in
  let inv_mat = Array.map inv_vec in
  let matrice_inversee = inv_mat (Graphics.dump_image i) in
  Graphics.make_image matrice_inversee ;;
val inv_image : Graphics.image -> Graphics.image = <fun>
```

На рисунке 4.3, картинка слева — начальное изображение и правое — новое, “освещенное солнцем”, после использование функции `inv_image`.

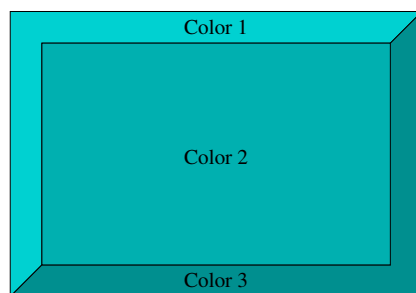
```
# let f_jussieu2 () = inv_image jussieu1;;
val f_jussieu2 : unit -> Graphics.image = <fun>
```

Рис. 4.3: Инверсия Jussieu

4.3.6 Рисование рельефных блоков

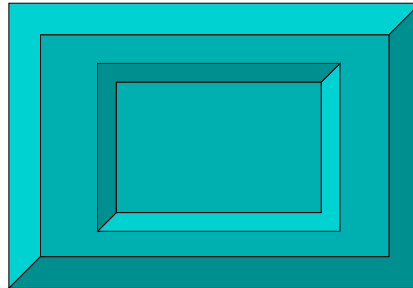
В этом примере мы попытаемся определить несколько полезных функций вывода рельефных блоков. Блоком мы называем общий объект, который послужит в дальнейшем. Блок вписывается в прямоугольник, который задан начальной точкой, высотой и шириной.

Для того чтобы придать блоку рельефный вид, достаточно добавить две трапеции светлых тонов и две более темных.



Инвертируя цвета можно создать впечатление вогнутого или выпуклого блока.

³Jussieu - здание парижского университета. Там в частности находится известная Лаборатория Информатики Париж-6 (lir6), где работают авторы книги, прим. пер.



Реализация Добавим новые свойства блока: толщина окантовки, тип вывода на экран (выпуклый, вогнутый или плоский), цвета окантовки и фона. Вся эта информация сгруппирована в записи.

```
# type relief = Top | Bot | Flat ;;
# type box_config =
  { x:int; y:int; w:int; h:int; bw:int; mutable r:relief;
    b1_col : Graphics.color;
    b2_col : Graphics.color;
    b_col : Graphics.color };;
```

Только поле `r` может быть изменено. Для вывода на экран мы используем функцию рисующую прямоугольник `draw_rect`, которую мы определили на 133.

Для удобства, определим функцию рисующую контур блока.

```
# let draw_box_outline bcf col =
  Graphics.set_color col;
  draw_rect bcf.x bcf.y bcf.w bcf.h;;
val draw_box_outline : box_config -> Graphics.color -> unit = <fun>
```

Функция вывода на экран состоит из трех частей: рисование первой окантовки, затем второй и внутренней части блока.

```
# let draw_box bcf =
  let x1 = bcf.x and y1 = bcf.y in
  let x2 = x1+bcf.w and y2 = y1+bcf.h in
  let ix1 = x1+bcf.bw and ix2 = x2-bcf.bw
  and iy1 = y1+bcf.bw and iy2 = y2-bcf.bw in
  let border1 g =
    Graphics.set_color g;
    Graphics.fill_poly
      [| (x1,y1);(ix1,iy1);(ix2,iy1);(ix2,iy2);(x2,y2);(x2,y1) |]
  in
  let border2 g =
```

```

Graphics.set_color g;
Graphics.fill_poly
  [| (x1,y1);(ix1, iy1);(ix1, iy2);(ix2, iy2);(x2,y2);(x1,y2) |]
in
Graphics.set_color bcf.b_col;
( match bcf.r with
  Top ->
    Graphics.fill_rect ix1 iy1 (ix2-ix1) (iy2-iy1);
    border1 bcf.b1_col;
    border2 bcf.b2_col
  | Bot ->
    Graphics.fill_rect ix1 iy1 (ix2-ix1) (iy2-iy1);
    border1 bcf.b2_col;
    border2 bcf.b1_col
  | Flat ->
    Graphics.fill_rect x1 y1 bcf.w bcf.h );
draw_box_outline bcf Graphics.black;;
val draw_box : box_config -> unit = <fun>

```

Контур блока подсвечен черным цветом. Для того чтобы стереть блок, достаточно заполнить пространство фоновым цветом.

```

# let erase_box bcf =
  Graphics.set_color bcf.b_col;
  Graphics.fill_rect (bcf.x+bcf.bw) (bcf.y+bcf.bw)
    (bcf.w-(2*bcf.bw)) (bcf.h-(2*bcf.bw));;
val erase_box : box_config -> unit = <fun>

```

И наконец, определим функцию выводящую текст выровненный слева, справа или по центру блока. Для описания позиции текста в блоке определим тип `position`.

```

# type position = Left | Center | Right;;
type position = | Left | Center | Right
# let draw_string_in_box pos str bcf col =
  let (w, h) = Graphics.text_size str in
  let ty = bcf.y + (bcf.h-h)/2 in
  ( match pos with
    Center -> Graphics.moveto (bcf.x + (bcf.w-w)/2) ty
  | Right -> let tx = bcf.x + bcf.w - w - bcf.bw - 1 in
    Graphics.moveto tx ty
  | Left -> let tx = bcf.x + bcf.bw + 1 in Graphics.moveto tx
    ty );

```

```

    Graphics.set_color col;
    Graphics.draw_string str;;
val draw_string_in_box :
  position -> string -> box_config -> Graphics.color -> unit = <fun
    >

```

Пример: рисование игры Чтобы продемонстрировать использование блоков, выведем на экран поле игры “крестики-нолики”, как на рисунке 4.4. Для упрощения задачи, определим цвета для игры.

```

# let set_gray x = (Graphics.rgb x x x);;
val set_gray : int -> Graphics.color = <fun>
# let gray1= set_gray 100 and gray2= set_gray 170 and gray3=
  set_gray 240;;
val gray1 : Graphics.color = 6579300
val gray2 : Graphics.color = 11184810
val gray3 : Graphics.color = 15790320

```

Теперь, напомним функцию рисующую матрицу блоков одного размера.

```

# let rec create_grid nb_col n sep b =
  if n < 0 then []
  else
    let px = n mod nb_col and py = n / nb_col in
    let nx = b.x + sep + px*(b.w+sep)
    and ny = b.y + sep + py*(b.h+sep) in
    let b1 = {b with x=nx; y=ny} in
    b1::(create_grid nb_col (n-1) sep b);;
val create_grid : int -> int -> int -> box_config -> box_config list
  = <fun>

```

Создадим список блоков.

```

# let vb =
  let b = {x=0; y=0; w=20; h=20; bw=2;
    b1_col=grey1; b2_col=grey3; b_col=grey2; r=Top} in
  Array.of_list (create_grid 5 24 2 b) ;;
val vb : box_config array =
  [|{ x=90; y=90; w=20; h=20; bw=2; r=Top; b1_col=6579300; b2_col=
    15790320;
    b_col=11184810};
    {x=68; y=90; w=20; h=20; bw=2; r=Top; b1_col=6579300; b2_col=
    15790320;

```

```
b_col=...};
...||
```

Изображение 4.4 соответствует результату следующих вызовов:

```
# Array.iter draw_box vb ;
draw_string_in_box Center "X" vb.(5) Graphics.black ;
draw_string_in_box Center "X" vb.(8) Graphics.black ;
draw_string_in_box Center "O" vb.(12) Graphics.yellow ;
draw_string_in_box Center "O" vb.(11) Graphics.yellow ;;
- : unit = ()
```

Рис. 4.4: Вывод блоков с текстом

4.4 Анимация

Анимация на экране компьютера использует ту же технику что и мультипликационные фильмы. Большая часть изображения не меняется, только *подвижная часть* изменяет цвет своих пикселей. При работе с анимацией, сразу же возникает проблема скорости рисования. Она зависит от сложности вычислений и производительности процессора. Поэтому, чтобы анимация была переносима и имела одинаковый эффект, необходимо учитывать производительность процессора. Чтобы получить плавную анимацию, желательно вывести на экран на новое положение анимационный объект, затем стереть старый, учитывая при этом пересечение областей обоих объектов.

Перемещение объекта Для упрощения задачи, мы будем перемещать объекты простой формы, как прямоугольник. Остается проблема заполнения экрана за перемещенным объектом.

Мы хотим перемещать прямоугольник в замкнутом пространстве. Объект перемещается с определенной скоростью в направлении X и Y. Если он дойдет до границы графического окна, объект *отскочит* он него на определенный угол отражения. Объект перемещается из одного положения в другое без перекрывания между новой и старой позицией. В зависимости от текущего положения (x,y), размера объекта (sx,sy) и его скорости (dx,dy) функция `calc_pv` вычисляет, учитывая границы окна, новое положение и новую скорость.

```
# let calc_pv (x,y) (sx,sy) (dx,dy) =
```

```

let nx1 = x+dx and ny1 = y + dy
and nx2 = x+sx+dx and ny2 = y+sy+dy
and ndx = ref dx and ndy = ref dy
in
  ( if (nx1 < 0) || (nx2 >= Graphics.size_x()) then ndx := -dx );
  ( if (ny1 < 0) || (ny2 >= Graphics.size_y()) then ndy := -dy );
  ((x+ !ndx, y+ !ndy), (!ndx, !ndy));;
val calc_pv :
  int * int -> int * int -> int * int -> (int * int) * (int * int) = <
    fun>

```

Функция перемещая прямоугольник, указанный положением pos и размером size, n раз, полученная траектория учитывает скорость speed и границы окна. Путь перемещения, изображенный на рисунок 4.5, получен инверсией bitmap-a, который соответствует перемещенному прямоугольнику.

```

# let move_rect pos size speed n =
  let (x, y) = pos and (sx,sy) = size in
  let mem = ref (Graphics.get_image x y sx sy) in
  let rec move_aux x y speed n =
    if n = 0 then Graphics.moveto x y
  else
    let ((nx,ny),n_speed) = calc_pv (x,y) (sx,sy) speed
    and old_mem = !mem in
      mem := Graphics.get_image nx ny sx sy;
      Graphics.set_color Graphics.blue;
      Graphics.fill_rect nx ny sx sy;
      Graphics.draw_image (inv_image old_mem) x y;
      move_aux nx ny n_speed (n-1)
  in move_aux x y speed n;;
val move_rect : int * int -> int * int -> int * int -> int -> unit = <
  fun>

```

Следующий код соответствует изображениям на рисунке 4.5, первое получено на красном фоне, второе — перемещением объекта по картинке Jussieu.

```

# let anim_rect () =
  Graphics.moveto 105 120;
  Graphics.set_color Graphics.white;
  Graphics.draw_string "Start";
  move_rect (140,120) (8,8) (8,4) 150;

```



```

let (x,y) = Graphics.current_point() in
  Graphics.moveto (x+13) y;
  Graphics.set_color Graphics.white;
  Graphics.draw_string "End";;
val anim_rect : unit -> unit = <fun>
# anim_rect();;
- : unit = ()

```

Рис. 4.5: Перемещение объекта

Наша задача была упрощена тем, что не было пересечений между новым и старым положением объекта. В противном случае, необходимо написать функцию вычисляющую это пересечение, что может быть более или менее сложно в зависимости от формы объекта. В настоящем примере с квадратом, при пересечении квадратов получается прямоугольник, который необходимо стереть.

4.5 Обработка событий

Для того чтобы создать программу взаимодействующую с пользователем необходимо обрабатывать события произошедшие в графическом окне. Следующие события обрабатываются библиотекой Graphics: нажатие на клавишу клавиатуры, на кнопку мыши и ее перемещение.

Стиль и устройство программы изменяются: программа превращается в бесконечный цикл ожидающий события. После обработки нового события, программа вновь возвращается в бесконечный цикл, если только это он не было предвиденно для остановки программы.

4.5.1 Тип и функции событий

Существует следующая главная функция ожидания события: `wait_next_event` с типом `list -> status`.

Различные события определены типом `summa event`:

```

type event = Button_down | Button_up | Key_pressed | Mouse_motion
  | Poll ;;

```

Первые четыре значения соответствуют нажатию и отпусканию кнопки мыши, нажатие на клавишу клавиатуры и перемещение мыши. Если конструктор `Poll` добавить в список событий, то ожидание не будет блокирующим. Функция возвращает значение типа `status`.

```
type status =
  { mouse_x : int;
    mouse_y : int;
    button : bool;
    keypressed : bool;
    key : char };;
```

Поля этой записи содержат координаты курсора мыши, булево значение равное истине если кнопка мыши была нажата, такое же значение для клавиатуры и последнее значение содержит символ нажатой клавиши. Следующие функции используют значения этой записи.

- `mouse_pos : unit -> int * int` : возвращает положение курсора на экране, если курсор вне графического окна, то координаты находятся вне окна.
- `button_down : unit -> bool` : указывает на нажатие кнопки мыши
- `read_key : unit -> char` : возвращает символ нажатой клавиши
- `key_pressed : unit -> bool` : указывает на нажатие клавиши клавиатуры, ожидание не блокирующее

Библиотека `Graphics` обрабатывает события на минимальном уровне, однако полученный код переносим на различные платформы: `Windows`, `MacOS` или `X-Windows`. Именно по этой причине данная библиотека не различает кнопки мыши. На компьютерах `Mac` существует всего одна кнопка. Другие события, как `exposing a window` или изменение размеров окна не доступны и оставлены на обработку библиотекой.

4.5.2 Скелет программы

Все программы с пользовательским интерфейсом содержат цикл, потенциально бесконечный, осуществляющий ожидание событий от пользователя. Как только оно возникает, программа выполняет связанное с ним действие. У следующей функции 5 функциональных аргументов. Первый и второй служат для запуска и остановки программы, два следующие это функции обрабатывающие события связанные с клавиатурой и мышью. Последний аргумент служит для управления исключениями, которые могут возникнуть во время вычислений. События, заканчивающие программу, возбуждают исключение `End`.

```

# exception End;;
exception End
# let skel f_init f_end f_key f_mouse f_except =
  f_init ();
  try
    while true do
      try
        let s = Graphics.wait_next_event
          [Graphics.Button_down; Graphics.Key_pressed]
        in if s.Graphics.keypressed then f_key s.Graphics.key
          else if s.Graphics.button
            then f_mouse s.Graphics.mouse_x s.Graphics.mouse_y
          with
            End -> raise End
            | e -> f_except e
        done
      with
        End -> f_end ();;
    val skel :
      (unit -> 'a) ->
      (unit -> unit) ->
      (char -> unit) -> (int -> int -> unit) -> (exn -> unit) -> unit =
      <fun>

```

Этот скелет мы используем в программе моделирующей печатную мини-машину. Нажатие на клавишу выводит соответствующий символ, нажатие на кнопку мыши меняет текущую позицию, а клавиша с символом § заканчивает программу. Единственная трудность заключается в переходе на новую строку. Для упрощения, предположим что высота символа не превышает 12 пикселей.

```

# let next_line () =
  let (x,y) = Graphics.current_point()
  in if y>12 then Graphics.moveto 0 (y-12)
    else Graphics.moveto 0 y;;
val next_line : unit -> unit = <fun>
# let handle_char c = match c with
  '&' -> raise End
  | '\n' -> next_line ()
  | '\r' -> next_line ()
  | _ -> Graphics.draw_char c;;
val handle_char : char -> unit = <fun>

```

```
# let go () = skel
  (fun () -> Graphics.clear_graph ();
    Graphics.moveto 0 (Graphics.size_y() -12) )
  (fun () -> Graphics.clear_graph())
  handle_char
  (fun x y -> Graphics.moveto x y)
  (fun e -> ());;
val go : unit -> unit = <fun>
```

Клавиша DEL, стирающая предыдущий символ, не обрабатывается этой программой.

4.5.3 Пример: telecran

Telecran небольшая игра рисования для тренировки координации. При помощи клавиш контроля точка на планшете может передвигаться по X и Y не отрывая пера от планшета. При помощи данной модели, мы проиллюстрируем взаимодействие программы и пользователя. Для этого, применим предыдущий скелет, а также некоторые клавиши клавиатуры для указания движения.

Определим тип `state` — запись в которой будет храниться размер планшета выраженный в количестве позиций по X и Y, текущая позиция пера, масштаб для просмотра, цвет которым рисует перо, цвет экрана и цвет для определения положения пера на экране.

```
# type state = {maxx:int; maxy:int; mutable x : int; mutable y :int;
  scale :int;
  bc : Graphics.color;
  fc : Graphics.color; pc : Graphics.color};;
```

Функция `draw_point` рисует точку по указанным координатам, масштабу и цвету.

```
# let draw_point x y s c =
  Graphics.set_color c;
  Graphics.fill_rect (s*x) (s*y) s s;;
val draw_point : int -> int -> int -> Graphics.color -> unit = <fun>
```

Все функции инициализации, обработки событий и выхода из программы получают параметр соответствующий состоянию (`state`). Вот как определены первые четыре функции.

```
# let t_init s () =
  Graphics.open_graph (" " ^ (string_of_int (s.scale*s.maxx)) ^
```

```

    "x" ^ (string_of_int (s.scale*s.maxy)));
    Graphics.set_color s.bc;
    Graphics.fill_rect 0 0 (s.scale*s.maxx+1) (s.scale*s.maxy+1);
    draw_point s.x s.y s.scale s.pc;;
val t_init : state -> unit -> unit = <fun>
# let t_end s () =
    Graphics.close_graph();
    print_string "Good bye..."; print_newline();;
val t_end : 'a -> unit -> unit = <fun>
# let t_mouse s x y = ();;
val t_mouse : 'a -> 'b -> 'c -> unit = <fun>
# let t_except s ex = ();;
val t_except : 'a -> 'b -> unit = <fun>

```

Функция `t_init` открывает окно и выводит перо на текущую позицию, `t_end` закрывает это окно и выводит сообщение, `t_mouse` и `t_except` ничего не делают. В этой программе действия мыши, так же как и исключения, не обрабатываются. Последние могут возникнуть во время запуска программы. Функция `t_key` — главная, она отвечает за обработку нажатий на клавиатуру.

```

# let t_key s c =
    draw_point s.x s.y s.scale s.fc;
    (match c with
      | '8' -> if s.y < s.maxy then s.y <- s.y + 1;
      | '2' -> if s.y > 0 then s.y <- s.y - 1
      | '4' -> if s.x > 0 then s.x <- s.x - 1
      | '6' -> if s.x < s.maxx then s.x <- s.x + 1
      | 'c' -> Graphics.set_color s.bc;
        Graphics.fill_rect 0 0 (s.scale*s.maxx+1) (s.scale*s.maxy
          +1);
        Graphics.clear_graph()
      | 'e' -> raise End
      | _ -> ());
    draw_point s.x s.y s.scale s.pc;;
val t_key : state -> char -> unit = <fun>

```

Она закрашивает текущую точку цветом пера. В зависимости от переданного символа изменяет, если возможно, текущую позицию пера (символы '2', '4', '6', '8'), очищает экран (символ 'c') или возбуждает исключение `End` (символ 'e'), затем выводит новое положение пера. Все остальные символы проигнорированы. Выбор клавиш для перемещение пера основывается на расположении клавиш малой цифровой клавиатуры.

И наконец, определим состояние, а также воспользуемся скелетом следующим образом:

```
# let stel = {maxx=120; maxy=120; x=60; y=60;
              scale=4; bc=Graphics.rgb 130 130 130;
              fc=Graphics.black; pc=Graphics.red};;
val stel : state =
  {maxx=120; maxy=120; x=60; y=60; scale=4; bc=8553090; fc=0; pc
   =16711680}
# let slate () =
  skel (t_init stel) (t_end stel) (t_key stel)
      (t_mouse stel) (t_except stel) ;;
val slate : unit -> unit = <fun>
```

Вызов функции `slate` выводит на экран окно и ожидает действий с клавиатуры. На рисунке 4.6 приведено изображение, реализованное этой программой.

Рис. 4.6: Telecran

4.6 Графический калькулятор

Вернемся к нашему примеру с калькулятором, описанным в предыдущей главе о императивном программировании (95). Создадим для него графический интерфейс, облегчающий использование.

Интерфейс реализует кнопки (цифры и функции) и экран для просмотра результата. Кнопка может быть нажата либо с использованием мыши, либо с клавиатуры. На рисунке 4.7 изображен будущий интерфейс.

Рис. 4.7: Графический калькулятор

Здесь мы вновь воспользуемся функцией рисования блоков, описанной на странице 139. Определим следующий тип:

```
# type calc_state =
  { s : state; k : (box_config * key * string ) list ; v : box_config } ;;
```

В нем хранится состояние калькулятора, список блоков для каждой кнопки и блок для вывода результата. Так как мы хотим построить легко изменяемый калькулятор, поэтому создание интерфейса параметризовано списком из ассоциаций:

```
# let descr_calc =
  [ (Digit 0,"0"); (Digit 1,"1"); (Digit 2,"2"); (Equals, "=");
    (Digit 3,"3"); (Digit 4,"4"); (Digit 5,"5"); (Plus, "+");
    (Digit 6,"6"); (Digit 7,"7"); (Digit 8,"8"); (Minus, "-");
    (Digit 9,"9"); (Recall,"RCL"); (Div, "/"); (Times, "*");
    (Off,"AC"); (Store, "STO"); (Clear,"CE/C")
  ] ;;
```

Создание блоков кнопок С помощью этого описания мы создаем список блоков. У функции `gen_boxes` 5 параметров: описание (`descr`), число колонок (`n`), расстояние между кнопками (`wsep`), расстояние между текстом и границами блока (`wsepint`), размер окантовки (`wbord`). Она возвращает список блоков кнопок и блок для вывода результата. Для вычисления расположения элементов, воспользуемся функцией `max_xy` вычисляющей максимальные размеры списка пар целых чисел, а также функцию `max_box` вычисляющую максимальное положение списка блоков.

```
# let gen_xy vals comp o =
  List.fold_left (fun a (x,y) -> comp (fst a) x,comp (snd a) y) o vals
  ;;
val gen_xy : ('a * 'a) list -> ('b -> 'a -> 'b) -> 'b * 'b -> 'b * 'b
  = <fun>
# let max_xy vals = gen_xy vals max (min_int,min_int);;
val max_xy : (int * int) list -> int * int = <fun>
# let max_boxl l =
  let bmax (mx,my) b = max mx b.x, max my b.y
  in List.fold_left bmax (min_int,min_int) l ;;
val max_boxl : box_config list -> int * int = <fun>
```

Ниже представлена главная функция `gen_boxes`, создающая интерфейс.

```
# let gen_boxes descr n wsep wsepint wbord =
  let l_l = List.length descr in
  let nb_lig = if l_l mod n = 0 then l_l / n else l_l / n + 1 in
  let ls = List.map (fun (x,y) -> Graphics.text_size y) descr in
  let sx,sy = max_xy ls in
  let sx,sy = sx+wsepint ,sy+wsepint in
  let r = ref [] in
  for i=0 to l_l-1 do
    let px = i mod n and py = i / n in
```

```

    let b = { x = wsep * (px+1) + (sx+2*wbord) * px ;
              y = wsep * (py+1) + (sy+2*wbord) * py ;
              w = sx; h = sy ; bw = wbord;
              r=Top;
              b1_col = gray1; b2_col = gray3; b_col =gray2}
    in r:= b ::! r
done;
let mpx,mpy = max_boxl !r in
let upx,upy = mpx+sx+wbord+wsep,mpy+sy+wbord+wsep in
let (wa,ha) = Graphics.text_size "      0" in
let v = { x=(upx-(wa+wsepint +wbord))/2 ; y= upy+ wsep;
          w=wa+wsepint; h = ha +wsepint; bw = wbord *2; r=
          Flat ;
          b1_col = gray1; b2_col = gray3; b_col =Graphics.
          black}
in
    upx,(upy+wsep+ha+wsepint+wsep+2*wbord),v,
    List.map2 (fun b (x,y) -> b,x,y ) (List.rev !r) descr;;
val gen_boxes :
  ('a * string) list ->
  int ->
  int ->
  int -> int -> int * int * box_config * (box_config * 'a * string) list
  =
  <fun>

```

Взаимодействие Мы хотим воспользоваться скелетом определенным на странице 146, для этого определим функции обработки клавиатуры и мыши. Функция обработки нажатий на клавиши клавиатуры очень проста; она передает символ, переведенный в тип `key`, функции калькулятора `transition`, затем выводит текст о состоянии калькулятора.

```

# let f_key cs c =
  transition cs.s (translation c);
  erase_box cs.v;
  draw_string_in_box Right (string_of_int cs.s.vpr) cs.v Graphics.white
  ;;
val f_key : calc_state -> char -> unit = <fun>

```

Обработка мышинных событий немного сложнее. Необходимо удостовериться, что нажатие было произведено на одну из кнопок калькулято-

ра. Для этого, определим функцию, которая проверяющую положение на принадлежность блоку.

```
# let mem (x,y) (x0,y0,w,h) =
    (x >= x0) && (x < x0+w) && (y >= y0) && (y < y0+h);;
val mem : int * int -> int * int * int * int -> bool = <fun>
# let f_mouse cs x y =
    try
        let b,t,s =
            List.find (fun (b,_,_) ->
                mem (x,y) (b.x+b.bw,b.y+b.bw,b.w,b.h)) cs.k
        in
            transition cs.s t;
            erase_box cs.v;
            draw_string_in_box Right (string_of_int cs.s.vpr) cs.v Graphics.
            white
    with Not_found -> ();;
val f_mouse : calc_state -> int -> int -> unit = <fun>
```

Если функция `f_mouse` нашла блок, которому соответствуют координаты нажатия, она передает соответствующую кнопку функции транзисии, затем выводит результат. В противном случае эта функция ничего не делает.

Функция `f_exc` обрабатывает исключения которые могут возникнуть во время выполнения программы.

```
# let f_exc cs ex =
    match ex with
    | Division_by_zero ->
        transition cs.s Clear;
        erase_box cs.v;
        draw_string_in_box Right "Div 0" cs.v (Graphics.red)
    | Invalid_key -> ()
    | Key_off -> raise End
    | _ -> raise ex;;
val f_exc : calc_state -> exn -> unit = <fun>
```

В случае деления на ноль, эта функция обнуляет состояние калькулятора и выводит сообщение об ошибке. Неправильная клавиша просто-напросто игнорируется. И наконец, исключение `Key_off` возбуждает исключение `End` выхода из цикла скелета программы.

Инициализация и выход При инициализации калькулятора необходимо вычислить размер окна. Следующая функция генерирует нужную графическую информацию, для этого она использует список кнопок-текст и возвращает размер главного окна.

```
# let create_e k =
  Graphics.close_graph ();
  Graphics.open_graph " 10x10";
  let mx,my,v,lb = gen_boxes k 4 4 5 2 in
  let s = {lcd=0; lka = false; loa = Equals; vpr = 0; mem = 0} in
  mx,my,{s=s; k=lb;v=v};;
val create_e : (key * string) list -> int * int * calc_state = <fun>
```

Функция инициализации использует результат предыдущей функцию.

```
# let f_init mx my cs () =
  Graphics.close_graph();
  Graphics.open_graph (" "^(string_of_int mx)^"x"^(string_of_int
    my));
  Graphics.set_color gray2;
  Graphics.fill_rect 0 0 (mx+1) (my+1);
  List.iter (fun (b,_,_) -> draw_box b) cs.k;
  List.iter
    (fun (b,_,s) -> draw_string_in_box Center s b Graphics.black
      ) cs.k ;
  draw_box cs.v;
  erase_box cs.v;
  draw_string_in_box Right "hello" cs.v (Graphics.white);;
val f_init : int -> int -> calc_state -> unit -> unit = <fun>
```

Функция выхода закрывает окно.

```
# let f_end e () = Graphics.close_graph();;
val f_end : 'a -> unit -> unit = <fun>
```

Функция go, параметризованная описанием, запускает цикл ожидания событий.

```
# let go descr =
  let mx,my,e = create_e descr in
  skel (f_init mx my e) (f_end e) (f_key e) (f_mouse e) (f_exc e);;
val go : (key * string) list -> unit = <fun>
```

Вызов go descr_calc соответствует изображению 4.7.

4.7 Резюме

В этой главе мы представили основы графического программирования и программирование событиями, используя библиотеку Graphics дистрибутива Objective CAML. Сначала мы рассмотрели базовые элементы (цвет, рисунок, заливка, текст и bitmap), затем изучили анимацию этих элементов. После введения механизма обработки событий библиотеки Graphics, мы увидели общий метод управления действиями пользователя используя программирование событиями. Для того чтобы улучшить интерактивность и предложить разработчику интерактивные графические компоненты была разработана библиотека, упрощающая создание графических интерфейсов, Awt. Это библиотека была использована при написании интерфейса императивного калькулятора.

Глава 5

Программы

Введение

Преимущество одного языка программирования над другим заключается в простоте разработки качественных программ и легком сопровождении программного обеспечения. Первая часть книги, посвященная представлению языка Objective CAML, вполне естественно завершится реализацией нескольких программ.

В первой программе мы реализуем несколько функций запрашивающих информацию из базы данных. Наше внимание будет акцентировано на функциональном стиле программирования и использование списков. Таким образом пользователь будет иметь набор функций для формулирования и выполнения запросов прямо в языке Objective CAML. В этом примере мы покажем разработчику как он может с легкостью предоставить набор функций необходимых пользователю.

Вторая программа это интерпретатор BASIC¹. Напомним, что подобные императивные языки принесли немалый успех первым микрокомпьютерам. Двадцать лет спустя, реализация таких языков является простой задачей. Несмотря на то что BASIC императивный язык, для написания интерпретатора мы воспользуемся функциональной частью Objective CAML, в частности для вычисления инструкций. Однако, для лексического и синтаксического анализа мы используем физически изменяемую структуру данных.

Третья программа — всем известная игра Minesweeper, которая входит в стандартный дистрибутив Windows. Цель игры — найти все спрятанные мины, исследуя рядом расположенные ячейки. Для реализации мы воспользуемся императивной частью языка, так как поле игры пред-

¹сокращение Beginner's All purpose Symbolic Instruction Code

ставлено в виде матрицы, которая будет изменяться после каждого хода игрока. Мы, также используем модуль Graphics для реализации интерфейса игры и обработки событий. Вычисление автоматически открывающихся ячеек будет сделано в функциональном стиле.

Данная программа использует модуль Graphics описанный в 4 главе (см. стр. 129) и несколько функций из модулей Random и Sys из главы 7 (см. стр. 239 и 261).

5.1 Запросы базы данных

Реализация базы данных, ее интерфейса и языка запросов — слишком амбициозный проект для данной книги и для знаний читателя в Objective CAML на данный момент. Тем не менее, ограничив задачу и используя лучшие возможности функционального программирования, можно реализовать достаточно интересное средство для обработки запросов. Мы изучим как использовать итераторы и частичное применение для написания и выполнения запросов. Также, мы увидим использование типа данных инкапсулирующих функциональные значения.

В этом примере мы будем работать над базой данных содержащей информацию о членах ассоциации. База храниться в файле `association.dat`.

5.1.1 Формат данных

Для хранения данных, большинство баз данных используют свой собственный, так называемый “проприетарный” формат. Чаще всего, есть возможность экспортировать эти данные в текстовый формат. Вот одна из возможных структур:

- база данных есть набор *карточек*, разделенных возвратом каретки;
- каждая карточка есть набор *полей*, разделенных определенным сепаратором, ':' в данном случае;
- поле есть строка, не содержащая ни сепаратор ':', ни возврат каретки;
- первая карточка есть имена полей, разделенные символом '|'.

Файл ассоциации начинается так:

```
Num|Lastname|Firstname|Address|Tel|Email|Pref|Date|Amount
0:Chailloux:Emmanuel:Universite P6:0144274427:ec@lip6.fr:email:25.12.1998:100
```

```
1:Manoury:Pascal:Laboratoire PPS::pm@lip6.fr:mail:03.03.1997:150.00
2:Pagano:Bruno:Cristal:0139633963::mail:25.12.1998:150.00
3:Baro:Sylvain::0144274427:baro@pps.fr:email:01.03.1999:50.00
```

- Num номер члена ассоциации
- Lastname, Firstname, Address, Tel и Email говорят сами за себя
- Pref указывает как член предпочитает получать информацию: по почте (mail), по emailу (email) или по телефону (tel).
- Date и Amount соответственно дата и сумма последнего взноса.

Теперь необходимо выбрать формат в котором программа будет хранить данные базы. У нас есть выбор: список или вектор карточек. Списком легче манипулировать; добавление и удаление карточек являются простыми операциями. Зато векторы предоставляют одинаковое время доступа к любой карточке. Так как мы желаем использовать все карточки, а не какие-то конкретно, каждый запрос обрабатывает все множество карточек. По этой причине мы выбираем списки. Для карточек у нас тот же самый выбор: список или вектор строк? В этом случае ситуация обратная; с одной стороны формат карточки зафиксирован для всей базы данных и мы не можем добавить новые поля. С другой стороны, в зависимости от будущих операций, мы используем лишь некоторые поля карточек, соответственно необходимо быстро получить к ним доступ.

Вполне естественным решением данной задачи будет использование вектора проиндексированного именами полей. Однако подобный тип не возможен в Objective CAML, мы воспользуемся обычным вектором (проиндексированный целыми числами) и функцией, которая возвращает имя поля в зависимости от индекса.

```
# type data_card = string array ;;
# type data_base = { card_index : string -> int ; data : data_card list
  } ;;
```

Реализуем доступ к полю по имени n карточки dc базы данных db при помощи следующей функции:

```
# let field db n (dc:data_card) = dc.(db.card_index n) ;;
val field : data_base -> string -> data_card -> string = <fun>
```

Мы принудительно привели тип переменной dc к data_card, тем самым наша функция field принимает лишь вектор строк, а не какой попало.

Проиллюстрируем на небольшом примере.

```
# let base_ex =
  { data = [ [| "Chailloux"; "Emmanuel" |] ; [| "Manoury"; "Pascal" |] ] ;
    card_index = function "Lastname" -> 0 | "Firstname" -> 1
                  | _ -> raise Not_found } ;;
val base_ex : data_base =
  { card_index = <fun>;
    data = [| [| "Chailloux"; "Emmanuel" |] ; [| "Manoury"; "Pascal" |] |] }
# List.map (field base_ex "Lastname") base_ex.data ;;
- : string list = ["Chailloux"; "Manoury"]
```

Выражение `field base_ex "Lastname"` вычисляется как функция, которая берет на вход карточку и возвращает поле `"Lastname"`. Используя `List.map`, мы применяем эту функцию к каждой карточке базы данных `base_ex` и в результате получаем список полей `"Lastname"`.

На этом примере показано, как мы собираемся использовать функциональный стиль программирования. В данном случае частичное применение функции `field` определяет функцию доступа к конкретному полю, независимо от числа карточек в базе данных. В то же время, в реализации функции `field` есть недостаток: если мы обращаемся каждый раз к одному и тому же полю, индекс вычисляется каждый раз. Мы предпочитаем следующую реализацию.

```
# let field base name =
  let i = base.card_index name in fun (card : data_card) -> card.(i) ;;
val field : data_base -> string -> data_card -> string = <fun>
```

Здесь, после применения функции к аргументу, вычисляется индекс поля и используется в последующих вычислениях.

5.1.2 Чтение базы из файла

Для Objective CAML, файл с базой данных это множество линий. Наша задача состоит в том чтобы прочитать каждую линию, как строку, затем в разбить ее на части при помощи сепараторов и таким образом извлечь данные, а также данные для функции индексации полей.

Утилита для обработки линий

Нам нужна функция `split`, которая будет разбивать строку в соответствии с определенным разделителем. Для этого мы воспользуемся функцией `suffix`, возвращающая суффикс строки с начиная с позиции `i`. В этом нам помогут три предопределенные функции:

- `String.length` возвращает длину строки;
- `String.sub` возвращает подстроку строки `s` начиная с позиции `i` и длиной `l`;
- `String.index_from` для строки `s` вычисляет начиная с позиции `n`, позицию первого встреченного символа `c`.

```
# let suffix s i = try String.sub s i ((String.length s)-i)
                    with Invalid_argument("String.sub") -> "" ;;
val suffix : string -> int -> string = <fun>
# let split c s =
  let rec split_from n =
    try let p = String.index_from s n c
        in (String.sub s n (p-n)) :: (split_from (p+1))
    with Not_found -> [ suffix s n ]
  in if s="" then [] else split_from 0 ;;
val split : char -> string -> string list = <fun>
```

Обратите внимание на обработку исключений в этой функции, в особенности исключение `Not_found`.

Вычисление структуры `data_base` Для того чтобы получить из списка вектор, достаточно воспользоваться соответствующей функцией из модуля `Array (of_list)`. Вычисление функции индекса из списка имени полей может показаться сложной задачей, но к счастью модуль `List` предоставляет нам все необходимые для этого средства.

У нас имеется список строк, значит нам нужна функция, которая ассоциирует строке индекс, то есть ее положение или номер в списке.

```
# let mk_index list_names =
  let rec make_enum a b = if a > b then [] else a::(make_enum (a+1)
    b) in
  let list_index = (make_enum 0 ((List.length list_names) - 1)) in
  let assoc_index_name = List.combine list_names list_index in
  function name -> List.assoc name assoc_index_name ;;
val mk_index : 'a list -> 'a -> int = <fun>
```

Для реализации этой функции, мы создаем список индексов, который мы комбинируем со списком имен полей. Таким образом мы получаем новый список ассоциаций с типом `string * int list`. Для того, чтобы найти индекс связанный с именем, воспользуемся специально созданной на подобный случай функцией `assoc` из библиотеки `List`. Функция `mk_index`

возвращает функцию которая берет на входе имя и вызывает `assoc` с этим именем и списком построенным ранее.

Теперь мы готовы, к тому чтобы написать функцию читающую файлы базы данных в указанном формате.

```
# let read_base filename =
  let channel = open_in filename in
  let split_line = split ':' in
  let list_names = split '|' (input_line channel) in
  let rec read_file () =
    try
      let data = Array.of_list (split_line (input_line channel)) in
      data :: (read_file ())
    with End_of_file -> close_in channel ; []
  in
  { card_index = mk_index list_names ; data = read_file () } ;;
val read_base : string -> data_base = <fun>
```

Считывание записей из файл осуществляется функцией `read_file`, которая рекурсивно оперирует входным каналом. Конец файла оповещается исключением `End_of_file`. В этом случае мы закроем канал и вернем пустой список.

Считаем файл ассоциации.

```
# let base_ex = read_base "association.dat" ;;
val base_ex : data_base =
  {card_index=<fun>;
  data=
    [| "0"; "Chailloux"; "Emmanuel"; "Universit\233 P6"; "0144274427";
      "ec@lip6.fr"; "email"; "25.12.1998"; "100.00" |];
    [| "1"; "Manoury"; "Pascal"; "Laboratoire PPS"; ...]; ...]}
```

5.1.3 Общие принципы работы с базой данных

Богатство и сложность обработки множества данных базы пропорциональны богатству и сложности используемого языка запросов. Так как в данном случае мы решили использовать Objective CAML в качестве языка запросов, *a priori* ограничений на выражение запросов нет! Мы так же хотим предоставить несколько простых средств манипуляции карточками и их данными. Для получения желанной простоты, необходимо ограничить мощь Objective CAML, для этого определим несколько целей и принципов обработки.

Цель обработки данных заключается в получении так называемого состояния базы. Создание такого состояния базы можно разбить на три этапа:

- выборка по какому-нибудь критерию множества карточек
- индивидуальная обработка каждой из выбранных карточек
- обработка множества данных полученных из этих карточек

Что мы и изображили на рисунке 5.1.

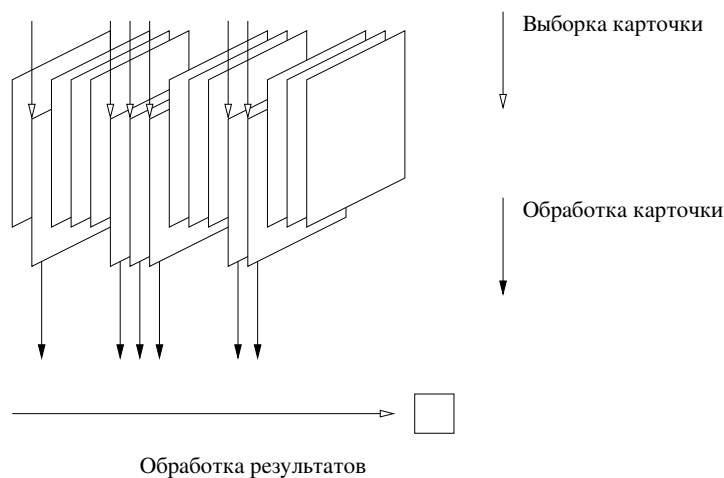


Рис. 5.1: Этапы запроса

В соответствии с этим, нам нужно 3 функции следующих типов:

- $(data_card \rightarrow bool) \rightarrow data_card\ list \rightarrow data_card\ list$
- $(data_card \rightarrow 'a) \rightarrow data_card\ list \rightarrow 'a\ list$
- $('a \rightarrow 'b \rightarrow 'b) \rightarrow 'a\ list \rightarrow 'b \rightarrow 'b$

Objective CAML предоставляет три функции высшего порядка, известные как итераторы, представленные на странице 242. Они соответствуют нашей спецификации.

```
# List.find_all ;;
- : ('a -> bool) -> 'a list -> 'a list = <fun>
# List.map ;;
- : ('a -> 'b) -> 'a list -> 'b list = <fun>
# List.fold_right ;;
- : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
```

После того как мы определим функциональные аргументы, мы сможем воспользоваться этими функциями для реализации состояния в три этапа.

Для некоторых запросов нам понадобится следующая функция.

```
# List.iter ;;
- : ('a -> unit) -> 'a list -> unit = <fun>
```

В случае, когда обработка данных ограничивается выводом на экран, вычислять нечего.

В следующих параграфах, мы увидим несколько простых методов обработки данных, а так же определение функций выражающих критерии выборки. Небольшой пример в заключении этой главы использует эти функции в соответствии с приведенными выше принципами.

5.1.4 Критерии выборки

В конкретном случае, логическая функция, которая определяет критерии выборки карточки, есть логическая комбинация свойств данных всех или части полей карточки. Каждое поле карточки, представленное строкой, может нести в себе информацию другого типа: целое число, число с плавающей запятой, etc.

Критерии выборки по одному полю

Выборка определенного поля по конкретному критерию будет осуществляться при помощи следующей функции `data_base -> 'a -> string -> data_card -> bool`. Параметр типом `'a` соответствует типу информации хранимой в поле. Имя этого поля указано аргументом с типом `string`.

Поля со строковым типом данных Определим два простых теста для работы с этим типом данных: тест на равенство с другой строкой и тест на “не пустоту” строки.

```
# let eq_sfield db s n dc = (s = (field db n dc)) ;;
val eq_sfield : data_base -> string -> string -> data_card -> bool
    = <fun>
# let nonempty_sfield db n dc = (" " <> (field db n dc)) ;;
val nonempty_sfield : data_base -> string -> data_card -> bool = <
    fun>
```

Поля со типом данных число с плавающей запятой Для проверки информации содержащей числа с плавающей запятой достаточно перевести значение строки содержащей десятичное число в тип `float`. Вот несколько примеров полученных использованием настраиваемой функции `tst_ffield` :

```
# let tst_ffield r db v n dc = r v (float_of_string (field db n dc)) ;;
val tst_ffield :
  ('a -> float -> 'b) -> data_base -> 'a -> string -> data_card
  -> 'b = <fun>
# let eq_ffield = tst_ffield (=) ;;
# let lt_ffield = tst_ffield (<) ;;
# let le_ffield = tst_ffield (<=) ;;
(* etc. *)
```

Тип у данных функций:

```
data_base -> float -> string -> data_card -> bool.
```

Этот тип информации немного сложнее, он зависит от представления даты в базе данных и требует определения способа сравнения дат.

Установим формат даты карточки как строку `дд.мм.гггг`. Для того чтобы получить дополнительные возможности сравнения, добавим в формат даты символ `'_'`, заменяющий день, месяц или год. Даты сравниваются в лексикографическом порядке в формате (год, месяц, день). Для того чтобы мы могли пользоваться выражениями как “до июля 1998”, будем использовать сопоставление с *образцом даты*: “`_.07.1998`”. Сравнение даты с образцом реализуется функцией `tst_dfield`, которая анализирует образец и создает *ad hoc* сравнивающую функцию. Для того чтобы определить эту универсальную функцию проверки даты, нам понадобятся несколько дополнительных функций.

Напишем две функции преобразующие дату (`ints_of_string`) и образцы даты (`ints_of_dpat`) в триплет целых чисел. Мы заменим символ `'_'` образца на целое число 0.

```
# let split_date = split '.' ;;
val split_date : string -> string list = <fun>
# let ints_of_string d =
  try match split_date d with
    | [d;m;y] -> [int_of_string y; int_of_string m; int_of_string d]
    | _ -> failwith "Bad date format"
  with Failure("int_of_string") -> failwith "Bad date format" ;;
val ints_of_string : string -> int list = <fun>
```

```
# let ints_of_dpat d =
  let int_of_stringpat = function "_" -> 0 | s -> int_of_string s
  in try match split_date d with
    [d;m;y] -> [ int_of_stringpat y; int_of_stringpat m;
                  int_of_stringpat d ]
    | _ -> failwith "Bad date format"
    with Failure("int_of_string") -> failwith "Bad date pattern" ;;
val ints_of_dpat : string -> int list = <fun>
```

Напишем функцию теста, которая использует отношение целых *r*. Здесь мы реализуем лексикографический порядок, при этом мы обрабатываем специальный случай с нулем.

```
# let rec app_dtst r d1 d2 = match d1, d2 with
  [] , [] -> false
| (0::d1) , (_::d2) -> app_dtst r d1 d2
| (n1::d1) , (n2::d2) -> (r n1 n2) || ((n1 = n2) && (app_dtst r d1 d2))
| _ , _ -> failwith "Bad date pattern or format" ;;
val app_dtst : (int -> int -> bool) -> int list -> int list -> bool = <fun>
```

Наконец, определим универсальную функцию *tst_dfield* со следующими аргументами: отношение *r*, база данных *db*, образец *dp*, имя поля *nm* и карточка *dc*. Эта функция проверяет, что образец и извлеченное поле удовлетворяют отношению.

```
# let tst_dfield r db dp nm dc =
  r (ints_of_dpat dp) (ints_of_string (field db nm dc)) ;;
val tst_dfield :
  (int list -> int list -> 'a) ->
  data_base -> string -> string -> data_card -> 'a = <fun>
```

Теперь применим функцию к трем отношениям.

```
# let eq_dfield = tst_dfield (=) ;;
# let le_dfield = tst_dfield (<=) ;;
# let ge_dfield = tst_dfield (>=) ;;
```

Тип этих функций следующий:

```
data_base -> string -> string -> data_card -> bool.
```

Композиция критериев

Три первые аргумента проверок, которые мы определили — база данных, значение и имя поля. Когда мы пишем запросы базы данных, значения этих аргументов известны. Для базы `base_ex` проверка «до июля 1998» пишется следующим образом.

```
# ge_dfield base_ex "_07.1998" "Date" ;;
- : data_card -> bool = <fun>
```

Получается, что проверка это функция имеющая тип `data_card -> bool`. Теперь нам нужно получить логические комбинации результатов подобных функций, примененных к одной и той же карточке. Для этого воспользуемся следующим итератором.

```
# let fold_funs b c fs dc =
  List.fold_right (fun f -> fun r -> c (f dc) r) fs b ;;
val fold_funs : 'a -> ('b -> 'a -> 'a) -> ('c -> 'b) list -> 'c -> 'a
  = <fun>
```

Здесь `b` — значение базы, функция `c` — логический оператор, `fs` — список функций проверки по полю и `dc` — карточка.

В следующем примере получаем конъюнкцию (логическое произведение) и дизъюнкцию (логическая сумма) списка проверок.

```
# let and_fold fs = fold_funs true (&) fs ;;
val and_fold : ('a -> bool) list -> 'a -> bool = <fun>
# let or_fold fs = fold_funs false (or) fs ;;
val or_fold : ('a -> bool) list -> 'a -> bool = <fun>
```

Для удобства определим отрицание функции проверки.

```
# let not_fun f dc = not (f dc) ;;
val not_fun : ('a -> bool) -> 'a -> bool = <fun>
```

Для того, чтобы выбрать карточку, дата которой находится в определенном интервале, воспользуемся комбинаторными операторами.

```
# let date_interval db d1 d2 =
  and_fold [(le_dfield db d1 "Date"); (ge_dfield db d2 "Date")] ;;
val date_interval : data_base -> string -> string -> data_card ->
  bool = <fun>
```

5.1.5 Обработка и вычисление

Трудно представить себе все возможные обработки карточек или множество данных полученных после этой обработки. Тем не менее можно

с уверенностью определить два класса таких обработок: численное вычисление и форматирование данных для печати. Рассмотрим каждый каждый случай на примере.

Форматирование

Подготовим к печати строку, содержащую имя члена ассоциации и кое-какую информацию.

Начнем с определения функции, которая из списка строк и разделителя создает строку состоящую из элемент списка разделенных сепаратором.

```
# let format_list c =
  let s = String.make 1 c in
  List.fold_left (fun x y -> if x="" then y else x^s^y) "" ;;
val format_list : char -> string list -> string = <fun>
```

Определим функцию `extract`, которая создает список из полей с информацией, она извлекает из каждой карточки данные полей, имена которых переданы в списке.

```
# let extract db ns dc =
  List.map (fun n -> field db n dc) ns ;;
val extract : data_base -> string list -> data_card -> string list = <fun>
```

Функция форматирования для печати выглядит следующим образом.

```
# let format_line db ns dc =
  (String.uppercase (field db "Lastname" dc))
  ^" "^(field db "Firstname" dc)
  ^"\t"^(format_list '\t' (extract db ns dc))
  ^"\n" ;;
val format_line : data_base -> string list -> data_card -> string = <fun>
```

Аргумент `ns` является списком с именами полей, которые нас интересуют. Поля разделены символом табуляции (`'\t'`), а строка заканчивается возвратом каретки.

Вот как можно вывести на экран имена и фамилии членов ассоциации.

```
# List.iter print_string (List.map (format_line base_ex []) base_ex.
  data) ;;
CHAILLOUX Emmanuel
```


MANOURY Pascal
 PAGANO Bruno
 BARO Sylvain
 — : unit = ()

Числовое вычисление

Давайте вычислим сумму членских взносов для определенного множества карточек. Для этого достаточно извлечь нужное поле, привести к целому типу и вычислить сумму. Нужный результат может быть получен композицией этих функций. Для упрощения записи, определим инфиксный оператор композиции.

```
# let (++) f g x = g (f x) ;;
val ++ : ('a -> 'b) -> ('b -> 'c) -> 'a -> 'c = <fun>
```

Воспользуемся этим оператором в следующем определении.

```
# let total db dcs =
  List.fold_right ((field db "Amount") ++ float_of_string ++ (+.))
    dcs 0.0 ;;
val total : data_base -> data_card list -> float = <fun>
```

Аналогичным образом можно применить эту функцию ко всей базе данных.

```
# total base_ex base_ex.data ;;
— : float = 450
```

5.1.6 Пример

В заключении, проиллюстрируем на примере принципы, которые мы представили ранее в этой главе.

Рассмотрим два типа запроса базы данных:

- запрос, возвращающий два списка, каждый из них содержит имя члена ассоциации, а так же электронный адрес для первого списка и почтовый адрес для второго, в зависимости от предпочтений.
- запрос, возвращающий состояние взносов на указанный период времени. Это состояние будет хранить имена, фамилии, дату, взнос и сумму всех взносов.

Списки адресов

Для создания этих списков мы сначала выбираем в соответствии со значением поля “Pref” релевантные карточки, затем используем функцию `format_line`.

```
# let mail_addresses db =
  let dcs = List.find_all (eq_sfield db "mail" "Pref") db.data in
  List.map (format_line db ["Mail"]) dcs ;;
val mail_addresses : data_base -> string list = <fun>

# let email_addresses db =
  let dcs = List.find_all (eq_sfield db "email" "Pref") db.data in
  List.map (format_line db ["Email"]) dcs ;;
val email_addresses : data_base -> string list = <fun>
```

Состояние взносов

Вычисление состояния взносов выполняется по обычному принципу: выборка, затем обработка. В данном случае обработка состоит из двух частей: форматирование и вычисление общей суммы взносов.

```
# let fees_state db d1 d2 =
  let dcs = List.find_all (date_interval db d1 d2) db.data in
  let ls = List.map (format_line db ["Date"; "Amount"]) dcs in
  let t = total db dcs in
  ls, t ;;
val fees_state : data_base -> string -> string -> string list * float =
  <fun>
```

В результате этой функции мы получим пару состоящую из списка строк с информацией и суммы взносов.

Основная программа

Основная программа состоит из интерактивного цикла, выводящего результат запроса, который пользователь выбрал из меню. Здесь мы используем императивный стиль программирования, за исключением вывода результата при помощи итератора.

```
# let main() =
  let db = read_base "association.dat" in
  let finished = ref false in
  while not !finished do
```

```

print_string" 1: List of mail addresses\n";
print_string" 2: List of email addresses\n";
print_string" 3: Received fees\n";
print_string" 0: Exit\n";
print_string"Your choice: ";
match read_int() with
  0 -> finished := true
| 1 -> (List.iter print_string (mail_addresses db))
| 2 -> (List.iter print_string (email_addresses db))
| 3
-> (let d1 = print_string"Start date: "; read_line() in
    let d2 = print_string"End date: "; read_line() in
    let ls, t = fees_state db d1 d2 in
      List.iter print_string ls;
      print_string"Total: "; print_float t; print_newline())
| _ -> ()
done;
print_string"bye\n" ;;
val main : unit -> unit = <fun>

```

Мы вернемся к этому примеру в главе 20, чтобы добавить к нему интерфейс при помощи web навигатора.

5.1.7 Дополнительные возможности

Вполне естественным будет добавить в базу данных нашего примера информацию о типе каждого поля. Эта информация пригодится в случае если мы хотим определить универсальные (generic) операторы сравнения со следующим типом `data_base -> 'a -> string -> data_card -> bool`. Имя поля (третий аргумент) позволяет передавать управление соответствующей функции сравнения и проверки.

5.2 Интерпретатор языка BASIC

В данном разделе мы рассмотрим интерпретатор языка Basic. Это программа, которая выполняет другие программы написанные на языке Basic. Конечно, мы ограничимся лишь частью команд этого языка, наш интерпретатор будет распознавать следующие команды:

- **PRINT** *выражение* Выводит результат вычисления выражения.

- **INPUT** *переменная* Выводит приглашение на экран (?), ожидает ввод целого числа с клавиатуры и затем присваивает это число переменной.
- **LET** *переменная=выражение* Присваивает переменной результат вычисления выражения.
- **GOTO** *номер строки* Выполнение продолжится с указанной строки.
- **IF** *условие* **THEN** *номер строки* Выполнение продолжится с указанной строки если *условие* верно.
- **REM** *строка символов* Комментарии.

Каждая строка программы Basic помечена номером и содержит лишь одну инструкцию. Программа, вычисляющая факториал для числа введенного с клавиатуры, выглядит следующим образом:

```

5  REM inputting the argument
10 PRINT " factorial of:"
20 INPUT A
30 LET B = 1
35 REM beginning of the loop
40 IF A <= 1 THEN 80
50 LET B = B * A
60 LET A = A - 1
70 GOTO 40
75 REM prints the result
80 PRINT B

```

Реализуем также мини-редактор по принципу интерактивного цикла. У нас должна быть возможность добавлять новые строки, вывод программы и ее вычисление. Запуск предыдущей программы осуществляется командой RUN. Пример вычисления этой программы.

```

> RUN
factorial of: ? 5
120

```

Это вычисление можно разбить на несколько разных этапов.

Описание абстрактного синтаксиса необходимо определить типы данных для описания программ на Basic, а так же другие компоненты (строки, комментарии, выражения, и т.д.)

Вывод программы эта часть состоит в переводе программы на Basic из внутреннего формата в строки, для того чтобы вывести ее на экран.

Лексический и синтаксический анализ обе эти части проделывают обратную операцию, то есть перевод строки во внутренний формат программы на Basic (абстрактный синтаксис)

Вычисление это основа интерпретатора. Далее мы увидим что функциональный язык как Objective CAML особенно хорошо подходит для решения подобных проблем.

Интерактивный цикл здесь реализуется все вышесказанное.

5.2.1 Абстрактный синтаксис

В таблице представлен конкретный синтаксис в форме BNF программы на Basic. Мы вернемся к данному способу описания языка в главе 10 на странице 319.

Unary_Op	::=	-		!								
Binary_Op	::=	+		-		*		/		%		
		=		<		>		<=		>=		<>
		&		,		,						
Expression	::=	integer										
		variable										
		"string"										
		Unary_Op	Expression									
		Expression	Binary_Op	Expression								
		(Expression)								
Command	::=	REM	string									
		GOTO	integer									
		LET	variable	=	Expression							
		PRINT	Expression									
		INPUT	variable									
		IF	Expression	THEN	integer							
Line	::=	integer	Command									
Program	::=	Line										
		Line	Program									

```
Phrase ::= Line | RUN | LIST | END
```

Заметим, что правильность выражения с точки зрения грамматики не означает возможность его вычисления. Например, `1+"hello"` есть выражение, однако его невозможно вычислить. Это сделано с целью облегчить абстрактный синтаксис языка Basic. Расплата за это — программы на Basic, синтаксически правильные, могут привести к ошибке из-за несоответствия типов.

Теперь определить типы данных Objective CAML просто, достаточно перевести абстрактный синтаксис в тип сумма.

```
# type unr_op = UMINUS | NOT ;;
# type bin_op = PLUS | MINUS | MULT | DIV | MOD | EQUAL |
  LESS | LESSEQ | GREAT | GREATEQ | DIFF | AND | OR ;;
# type expression =
  ExpInt of int
  | ExpVar of string
  | ExpStr of string
  | ExpUnr of unr_op * expression
  | ExpBin of expression * bin_op * expression ;;
# type command =
  Rem of string
  | Goto of int
  | Print of expression
  | Input of string
  | If of expression * int
  | Let of string * expression ;;
# type line = { num : int ; cmd : command } ;;
# type program = line list ;;
```

Определим синтаксис команд для мини-редактора.

```
# type phrase = Line of line | List | Run | PEnd ;;
```

Обычно, чтобы облегчить синтаксис, программистам разрешается не указывать все скобки. Например, под выражением `1+3*4` подразумевается `1+(3*4)`. Для этого, каждому оператору языка присваивается целое число — приоритет:

```
# let priority_uop = function NOT -> 1 | UMINUS -> 7
  let priority_binop = function
    MULT | DIV -> 6
    | PLUS | MINUS -> 5
    | MOD -> 4
```

```

    | EQUAL | LESS | LESSEQ | GREAT | GREATEQ | DIFF -> 3
    | AND | OR -> 2 ;;
val priority_uop : unr_op -> int = <fun>
val priority_binop : bin_op -> int = <fun>

```

Целые числа означают, так называемый, приоритет операторов. Далее мы увидим как они используются при синтаксическом анализе или выводе программ на экран.

5.2.2 Вывод программы на экран

Для того, чтобы вывести программу хранящуюся в памяти, необходимо уметь перевести строку программы из абстрактного синтаксиса в строку символов.

Перевод операторов может быть получен легко и просто:

```

# let pp_binop = function
    PLUS -> "+" | MULT -> "*" | MOD -> "%" | MINUS -> "-"
    | DIV -> "/" | EQUAL -> "=" | LESS -> "<"
    | LESSEQ -> "<=" | GREAT -> ">"
    | GREATEQ -> ">=" | DIFF -> "<>" | AND -> "&" | OR
    -> "|"
let pp_unrop = function UMINUS -> "-" | NOT -> "!" ;;
val pp_binop : bin_op -> string = <fun>
val pp_unrop : unr_op -> string = <fun>

```

Вывод выражений соблюдает приоритет операторов для того чтобы получить выражение с минимумом скобок. Мы используем скобки лишь в случае если оператор в под-выражении справа от оператора менее приоритетный всего оператор целого выражения. К тому же, арифметические операторы ассоциативные слева, это значит что выражение 1-2-3 эквивалентно (1-2)-3.

Чтобы получить данный результат, создадим две функции `prl` и `prg`, которые будут обрабатывать левые и правые под-деревья соответственно. У этих функций два аргумента: дерево выражений и приоритет оператора в корне дерева, основываясь на значении последнего мы решим нужны-ли скобки в выражении или нет. Чтобы учитывать ассоциативность операторов мы различаем левое под-дерево от правого. Если приоритет текущего оператора одинаков с корневым, то ставить скобки для левого под-дерева не нужно. Для правого под-дерева скобки могут понадобиться, как в следующих случаях: 1-(2-3) or 1-(2+3).

Начальное дерево рассматривается как левое под-дерево оператора с

минимальным приоритетом (0). Вот как работает функция вывода выражений `pp_expression`:

```
# let parenthesis x = "(" ^ x ^ ")" ;;
val parenthesis : string -> string = <fun>
# let pp_expression =
  let rec ppl pr = function
    | ExpInt n -> (string_of_int n)
    | ExpVar v -> v
    | ExpStr s -> "\"" ^ s ^ "\""
    | ExpUnr (op,e) ->
      let res = (pp_unrop op)^(ppl (priority_uop op) e)
      in if pr=0 then res else parenthesis res
    | ExpBin (e1,op,e2) ->
      let pr2 = priority_binop op
      in let res = (ppl pr2 e1)^(pp_binop op)^(ppl pr2 e2)
      (* parenthesis if priority is not greater *)
      in if pr2 >= pr then res else parenthesis res
  and ppr pr exp = match exp with
    (* right subtrees only differ for binary operators *)
    | ExpBin (e1,op,e2) ->
      let pr2 = priority_binop op
      in let res = (ppl pr2 e1)^(pp_binop op)^(ppr pr2 e2)
      in if pr2 > pr then res else parenthesis res
    | _ -> ppl pr exp
  in ppl 0 ;;
val pp_expression : expression -> string = <fun>
```

Для вывода инструкций, воспользуемся предыдущей функцией. При этом добавим номер перед каждой инструкцией.

```
# let pp_command = function
  | Rem s -> "REM " ^ s
  | Goto n -> "GOTO " ^ (string_of_int n)
  | Print e -> "PRINT " ^ (pp_expression e)
  | Input v -> "INPUT " ^ v
  | If (e,n) -> "IF "^(pp_expression e)^" THEN "^(string_of_int n)
  | Let (v,e) -> "LET " ^ v ^ " = " ^ (pp_expression e) ;;
val pp_command : command -> string = <fun>
# let pp_line l = (string_of_int l.num) ^ " " ^ (pp_command l.cmd) ;;
val pp_line : line -> string = <fun>
```


5.2.3 Лексический анализ

Синтаксический и лексический анализ реализуют противоположную выводу на экран операцию. Для полученной строки создается синтаксическое дерево. Лексический анализ разбивает строку инструкции на независимые лексические части, называемые лексемами. Для этого добавим следующий тип в Objective CAML:

```
# type lexeme = Lint of int
                | Lident of string
                | Lsymbol of string
                | Lstring of string
                | Lend ;;
```

Для обозначения конца выражения мы добавили специальную лексему **Lend**. Она не является частью анализируемой строки, а добавляется функцией лексического анализа (см. стр. 179).

Для анализа строки мы используем тип записи содержащую изменяемое поле, значение которого указывает на часть строки, которую осталось обработать. Размер строки будет необходим во многих случаях, поэтому мы храним это константное значение в записи.

```
# type string_lexer = {string:string; mutable current:int; size:int } ;;
```

Такой способ определения лексического анализа можно рассматривать как применение функции к значению с типом **string_lexer**, в результате чего получим значение типа **lexeme**. Изменение индекса строки, которую осталось проанализировать, получается в результате побочного эффекта.

```
# let init_lex s = { string=s; current=0 ; size=String.length s } ;;
val init_lex : string -> string_lexer = <fun>
# let forward cl = cl.current <- cl.current+1 ;;
val forward : string_lexer -> unit = <fun>
# let forward_n cl n = cl.current <- cl.current+n ;;
val forward_n : string_lexer -> int -> unit = <fun>
# let extract pred cl =
  let st = cl.string and pos = cl.current in
  let rec ext n = if n<cl.size && (pred st.[n]) then ext (n+1) else n
  in
  let res = ext pos
  in cl.current <- res ; String.sub cl.string pos (res-pos) ;;
val extract : (char -> bool) -> string_lexer -> string = <fun>
```

Следующие функции извлекают лексему из строки и изменяют маркер текущей позиции. Функции *extract_int* и *extract_ident* извлекают целое число и идентификатор соответственно.

```
# let extract_int =
  let is_int = function '0'..'9' -> true | _ -> false
  in function cl -> int_of_string (extract is_int cl)
let extract_ident =
  let is_alpha_num = function
    'a'..'z' | 'A'..'Z' | '0'..'9' | '_' -> true
  | _ -> false
  in extract is_alpha_num ;;
val extract_int : string_lexer -> int = <fun>
val extract_ident : string_lexer -> string = <fun>
```

Функция *lexer* использует обе предыдущие функции для извлечения лексем.

```
# exception LexerError ;;
exception LexerError
# let rec lexer cl =
  let
    , ,
    | '\t' -> forward cl ; lexer cl
    | 'a'..'z'
    | 'A'..'Z' -> Lident (extract_ident cl)
    | '0'..'9' -> Lint (extract_int cl)
    | '"' -> forward cl ;
      let res = Lstring (extract ((<>) "") cl)
      in forward cl ; res
    | '+' | '-' | '*' | '/' | '%' | '&' | '|' | '!' | '=' | '(' | ')'
      ->
        forward cl ; Lsymbol (String.make 1 c)
    | '<'
    | '>' -> forward cl;
      if cl.current >= cl.size then Lsymbol (String.make 1
        c)
      else let cs = cl.string.[cl.current]
        in ( match (c,cs) with
          ('<','=') -> forward cl; Lsymbol "<="
          | ('>','=') -> forward cl; Lsymbol ">="
          | ('<','>') -> forward cl; Lsymbol "<>"
          | _ -> Lsymbol (String.make 1 c) )
```

```

    | _ -> raise LexerError
  in
    if cl.current >= cl.size then Lend
    else lexer_char cl.string.[cl.current] ;;
  val lexer : string_lexer -> lexeme = <fun>

```

Принцип действия функции *lexer* очень простой: здесь анализируется текущий символ строки, в зависимости от его значения возвращается соответствующая лексема и текущая позиция перемещается на начало следующей лексемы. Это очень простой и эффективный подход, две лексемы могут различаться по первому же символу. Для символа ‘<’ необходимо проверить следующий символ, за ним может следовать ‘=’ или ‘>’ и является другой лексемой. То же самое касается символа ‘>’.

5.2.4 Синтаксический анализ

При анализе выражений языка возникают некоторые проблемы; знание начала выражения не достаточно для того, чтобы определить всю его структуру. Пусть мы анализируем часть строки $1+2+3$. В зависимости от того, что за этим следует $+4$ или $*4$, полученные деревья для части $1+2+3$ различаются (см. рис. 5.2).

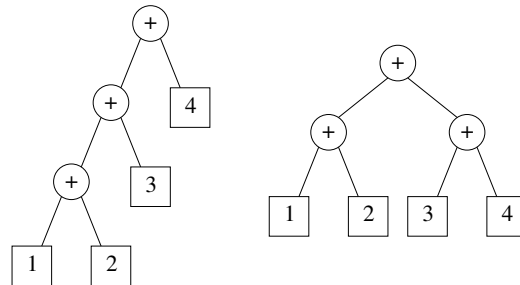


Рис. 5.2: Деревья абстрактного синтаксиса

Однако, структура дерева для $1+2$ одинакова в обоих случаях, поэтому мы можем его построить. В связи с тем что у нас отсутствует информация о части $+3$, мы временно сохраним эту информацию до нужного момента.

При построении дерева абстрактного синтаксиса мы воспользуемся стековым автоматом, схожий с тем, который используется *yacc* (см. стр. 338). Лексему читаются одна за другой и помещаются в стек до тех пор, пока у нас не будет достаточно информации, чтобы построить выражение. После этого, лексемы удаляются из стека и заменяются построенным выражением. Эта операция называется редукцией.

Тип помещаемых в стек элементов следующий:

```
# type exp_elem =
  Texp of expression (* expression *)
| Tbin of bin_op      (* binary operator *)
| Tunr of unr_op      (* unary operator *)
| Tlp                (* left parenthesis *) ;;
```

Заметим, что правые скобки не сохраняются, так как лишь левые скобки важны при операции редукции.

На рисунке 5.3 проиллюстрировано изменение стека при анализе выражения $(1 + 2 * 3) * 4$. Символ над стрелкой есть текущий символ строки.

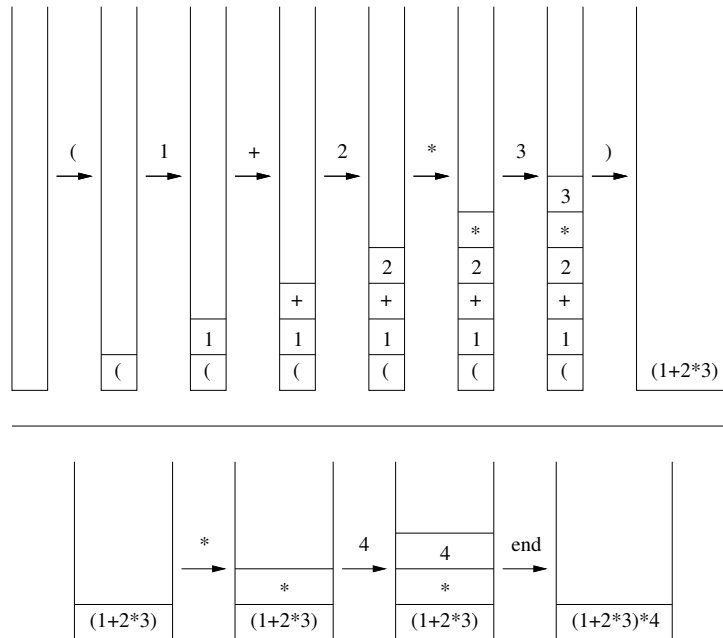


Рис. 5.3: Basic: Пример создания дерева абстрактного синтаксиса

Определим исключение для синтаксических ошибок.

```
# exception ParseError ;;
```

Сначала определим операторы при помощи символов.

```
# let unr_symb = function
  "!" -> NOT | "-" -> UMINUS | _ -> raise ParseError
let bin_symb = function
  "+" -> PLUS | "-" -> MINUS | "*" -> MULT | "/" -> DIV | "
    %" -> MOD
```

```

| "=" -> EQUAL | "<" -> LESS | "<=" -> LESSEQ | ">" ->
  GREAT
| ">=" -> GREATEQ | "<>" -> DIFF | "&" -> AND | "|" ->
  OR
| _ -> raise ParseError
let tsymb s = try Tbin (bin_symb s) with ParseError -> Tunr (
  unr_symb s) ;;
val unr_symb : string -> unr_op = <fun>
val bin_symb : string -> bin_op = <fun>
val tsymb : string -> exp_elem = <fun>

```

Функция **reduce** реализует редукцию стека. Существует два случая, в которых стек начинается:

- выражение, с последующим унарным оператором
- выражение, затем бинарный оператор и другое выражение.

Кроме того, другой аргумент функции *reduce* — это минимальный приоритет, который должен иметь оператор, чтобы редукция имела место. Для приведения без условий достаточно указать минимальный нулевой приоритет.

```

# let reduce pr = function
  (Texp e)::(Tunr op)::st when (priority_uop op) >= pr
    -> (Texp (ExpUnr (op,e)))::st
| (Texp e1)::(Tbin op)::(Texp e2)::st when (priority_binop op) >=
  pr
  -> (Texp (ExpBin (e2,op,e1)))::st
| _ -> raise ParseError ;;
val reduce : int -> exp_elem list -> exp_elem list = <fun>

```

Заметим, что элементы выражения помещаются в стек в порядке чтения, из-за чего необходимо поменять местами операнды бинарной операции.

stack_or_reduce это главная функция синтаксического анализа, в соответствии с переданной ей лексемой она либо помещает новый элемент в стек либо выполняет редукцию.

```

# let rec stack_or_reduce lex stack = match lex , stack with
  Lint n , _ -> (Texp (ExpInt n))::stack
| Lident v , _ -> (Texp (ExpVar v))::stack
| Lstring s , _ -> (Texp (ExpStr s))::stack
| Lsymbol "(" , _ -> Tlp::stack

```

```

| Lsymbol ")" , (Texp e)::Tlp::st -> (Texp e)::st
| Lsymbol ")" , _ -> stack_or_reduce lex (reduce 0 stack)
| Lsymbol s , _
  -> let symbol =
      if s<>"-" then tsymb s
      (* remove the ambiguity of the “-” symbol *)
      (* according to the last exp element put on the stack *)
    else match stack
      with (Texp _)::_ -> Tbin MINUS
           | _ -> Tunr UMINUS
    in ( match symbol with
        Tunr op -> (Tunr op)::stack
      | Tbin op ->
        ( try stack_or_reduce lex (reduce (priority_binop op
                                           )
                                           stack )
          with ParseError -> (Tbin op)::stack )
        | _ -> raise ParseError )
| _ , _ -> raise ParseError ;;
val stack_or_reduce : lexeme -> exp_elem list -> exp_elem list = <
fun>

```

После того как все лексемы извлечены и помещены в стек, дерево абстрактного синтаксиса может быть построено из элементов оставшихся в стеке — это задача функции *reduce_all*. Если анализируемое выражение было правильно сформировано, то в стеке должен остаться лишь один элемент, содержащий дерево этого выражения.

```

# let rec reduce_all = function
| [] -> raise ParseError
| [Texp x] -> x
| st -> reduce_all (reduce 0 st) ;;
val reduce_all : exp_elem list -> expression = <fun>

```

Основной функцией анализа выражений является *parse_exp*. Она просматривает строку, извлекает различные лексемы и передает их функции *stack_or_reduce*. Анализ прекращается, когда текущая лексема соответствует предикату переданному в аргументе.

```

# let parse_exp stop cl =
  let p = ref 0 in
  let rec parse_one stack =
    let l = ( p:=cl.current ; lexer cl)

```

```

    in if not (stop l) then parse_one (stack_or_reduce l stack)
      else ( cl.current <- !p ; reduce_all stack )
    in parse_one || ;;
val parse_exp : (lexeme -> bool) -> string_lexer -> expression = <
  fun>

```

Заметим, что лексема, которая определяет конец анализа, не используется при построении выражения. Для того чтобы проанализировать эту лексему позднее, необходимо установить текущую позицию на ее начало (переменная *p*).

Перейдем теперь к анализу строки с инструкцией.

```

# let parse_cmd cl = match lexer cl with
  Lident s -> ( match s with
    "REM" -> Rem (extract (fun _ -> true) cl)
  | "GOTO" -> Goto (match lexer cl with
      Lint p -> p
    | _ -> raise ParseError)
  | "INPUT" -> Input (match lexer cl with
      Lident v -> v
    | _ -> raise ParseError)
  | "PRINT" -> Print (parse_exp ((=) Lend) cl)
  | "LET" ->
      let l2 = lexer cl and l3 = lexer cl
      in ( match l2 ,l3 with
          (Lident v,Lsymbol "=") -> Let (v,parse_exp ((=)
              Lend) cl)
        | _ -> raise ParseError )
  | "IF" ->
      let test = parse_exp ((=) (Lident "THEN")) cl
      in ( match ignore (lexer cl) ; lexer cl with
          Lint n -> If (test,n)
        | _ -> raise ParseError )
  | _ -> raise ParseError )
val parse_cmd : string_lexer -> command = <fun>

```

И наконец, главная функция синтаксического анализа команд введенных пользователем в интерактивном цикле.

```

# let parse str =
  let cl = init_lex str
  in match lexer cl with

```

```

      Lint n -> Line { num=n ; cmd=parse_cmd cl }
    | Lident "LIST" -> List
    | Lident "RUN" -> Run
    | Lident "END" -> PEnd
    | _ -> raise ParseError ;;
val parse : string -> phrase = <fun>

```

5.2.5 Вычисление

Программа на Basic состоит из набора строк и выполнение начинается с первой строки. Интерпретация строки программы заключается в исполнении задачи инструкции, которая находится на этой строке. Существует три множества инструкций: ввод/вывод (**PRINT** и **INPUT**), декларация переменных или присвоение (**LET**) и переход (**GOTO** и **THEN**). Инструкции ввода/вывода реализуют взаимодействие с пользователем, для этого будут использованы соответствующие команды Objective CAML.

Для объявления и присвоения переменных, необходимо уметь вычислить значение арифметического выражение и знать расположение в памяти этой переменной. Результат вычисления выражения может быть либо целым числом, либо булевым значением, либо строкой. Сгруппируем их в типе `value`.

```
# type value = Vint of int | Vstr of string | Vbool of bool ;;
```

При объявлении переменной, нужно выделить память, чтобы хранить значение ассоциированное этой переменной. Для изменения переменной необходимо поменять значение связанно с именем переменной. Соответственно, программа Basic использует *окружение*, которое хранит связи имя переменной–значение. Данное окружение представлено в виде списка из пар (имя, значение).

```
# type environment = (string * value) list ;;
```

Для того чтобы получить содержимое переменной мы используем ее имя. При изменении значения переменной, меняется соответствующая пара.

В инструкциях перехода, условного или безусловного, указывается номер строки на которой должно продолжиться выполнение программы. По умолчанию — это следующая строка. В связи с этим, необходимо запомнить номер текущей строки.

Список инструкций из которых состоит программа, редактируемая в интерактивном цикле, не подходит для эффективного выполнения про-

граммы. Действительно, для того чтобы реализовать переход (If и Goto) необходимо пересмотреть весь список инструкций, чтобы найти строку с нужным номером. Для того чтобы можно было напрямую перейти на нужную строку, достаточно заменить структуру списка на вектор. В данном случае при переходе будет использоваться не номер строки, а ее индекс в векторе. В этом случае, перед запуском программы командой RUN, сделаем пре-обработку инструкций, называемую компоновкой (assembly). По некоторым причинам, которые будут объяснены в следующем параграфе, скомпонованная программа представлена вектором строк, а не инструкций.

```
# type code = line array ;;
```

Как и для калькулятора из прошлых глав, вычислитель использует состояние, которое изменяется при каждом вычислении. Информация, которую необходимо знать в каждый момент — это программа в целом, следующая строка на выполнение и значения переменных. Выполняемая программа отличается от программы набранной в интерактивном цикле. Вместо того списка инструкций, мы используем вектор инструкций. Таким образом состояние программы описывается следующим типом.

```
# type state_exec = { line:int ; xprog:code ; xenv:environment } ;;
```

Ошибки могут возникнуть в двух следующих случаях: вычисление выражения и переход на несуществующую строку. Соответственно, нужно обработать эти оба случая, чтобы интерпретатор корректно останавливался и выводил сообщение об ошибке. Определим исключение, а также функцию, которая будет его возбуждать и указывать номер строки на которой оно произошло.

```
# exception RunError of int
let runerr n = raise (RunError n) ;;
exception RunError of int
val runerr : int -> 'a = <fun>
```

Компоновка Компоновка программы, состоящей из списка пронумерованных строк (тип **program**), заключается в переводе списка в вектор и корректировки инструкций перехода. Эта корректировка реализуется связкой номера строки и соответствующего ей индекса вектора. Для облегчения задачи, мы создаем вектор пронумерованных строки. Данный вектор будет просматриваться каждый раз, когда необходимо найти индекс связанный со строкой. Если номер строки не найден, будет возвращено значение -1.

```

# exception Result_lookup_index of int ;;
exception Result_lookup_index of int
# let lookup_index tprog num_line =
  try
    for i=0 to (Array.length tprog)-1 do
      let num_i = tprog.(i).num
      in if num_i=num_line then raise (Result_lookup_index i)
         else if num_i>num_line then raise (Result_lookup_index
            (-1))
    done ;
    (-1)
  with Result_lookup_index i -> i ;;
val lookup_index : line array -> int -> int = <fun>

# let assemble prog =
  let tprog = Array.of_list prog in
  for i=0 to (Array.length tprog)-1 do
    match tprog.(i).cmd with
      Goto n -> let index = lookup_index tprog n
                 in tprog.(i) <- { tprog.(i) with cmd = Goto index
                                   }
    | If(c,n) -> let index = lookup_index tprog n
                 in tprog.(i) <- { tprog.(i) with cmd = If (c,index)
                                   }
    | _ -> ()
  done ;
  tprog ;;
val assemble : line list -> line array = <fun>

```

Вычисление выражений Функция вычисления выражений обходит дерево абстрактного синтаксиса и выполняет операции, указанные в каждом узле дерева.

В следующих случаях возбуждается исключение **RunError**: несоответствие типов, деление на ноль и необъявленная переменная.

```

# let rec eval_exp n envt expr = match expr with
  ExpInt p -> Vint p
  | ExpVar v -> ( try List.assoc v envt with Not_found -> runerr n )
  | ExpUnr (UMINUS,e) ->
    ( match eval_exp n envt e with
      Vint p -> Vint (-p)

```

```

    | _ -> runerr n )
| ExpUnr (NOT,e) ->
    ( match eval_exp n envt e with
      Vbool p -> Vbool (not p)
    | _ -> runerr n )
| ExpStr s -> Vstr s
| ExpBin (e1,op,e2)
  -> match eval_exp n envt e1 , op , eval_exp n envt e2 with
    Vint v1 , PLUS , Vint v2 -> Vint (v1 + v2)
    | Vint v1 , MINUS , Vint v2 -> Vint (v1 - v2)
    | Vint v1 , MULT , Vint v2 -> Vint (v1 * v2)
    | Vint v1 , DIV , Vint v2 when v2<>0 -> Vint (v1 /
      v2)
    | Vint v1 , MOD , Vint v2 when v2<>0 -> Vint (v1
      mod v2)

    | Vint v1 , EQUAL , Vint v2 -> Vbool (v1 = v2)
    | Vint v1 , DIFF , Vint v2 -> Vbool (v1 <> v2)
    | Vint v1 , LESS , Vint v2 -> Vbool (v1 < v2)
    | Vint v1 , GREAT , Vint v2 -> Vbool (v1 > v2)
    | Vint v1 , LESSEQ , Vint v2 -> Vbool (v1 <= v2)
    | Vint v1 , GREATEQ , Vint v2 -> Vbool (v1 >= v2)

    | Vbool v1 , AND , Vbool v2 -> Vbool (v1 && v2)
    | Vbool v1 , OR , Vbool v2 -> Vbool (v1 || v2)

    | Vstr v1 , PLUS , Vstr v2 -> Vstr (v1 ^ v2)
    | _ , _ , _ -> runerr n ;;
val eval_exp : int -> (string * value) list -> expression -> value = <
fun>

```

Вычисление инструкций Для того, чтобы реализовать вычисление строки инструкций, нам понадобятся несколько дополнительных функций.

Добавление новой связки (имя переменной–значение) в окружение, заменяет старую, с таким же именем, если она существует.

```

# let rec add v e env = match env with
  [] -> [v,e]
  | (w,f) :: l -> if w=v then (v,e)::l else (w,f)::(add v e l) ;;
val add : 'a -> 'b -> ('a * 'b) list -> ('a * 'b) list = <fun>

```

Другая функция, для вывода целых чисел или строк, пригодится при вычислении команды **PRINT**.

```
# let print_value v = match v with
  | Vint n -> print_int n
  | Vbool true -> print_string "true"
  | Vbool false -> print_string "false"
  | Vstr s -> print_string s ;;
val print_value : value -> unit = <fun>
```

Вычисление инструкции есть переход из одного состояния в другое. В частности, окружение будет изменено, если инструкция это присвоение. Значение следующей строки на выполнение изменяется каждый раз. Если строка не существует, вернем значение -1.

```
# let next_line state =
  let n = state.line + 1 in
  if n < Array.length state.xprog then n else -1 ;;
val next_line : state_exec -> int = <fun>
# let eval_cmd state =
  match state.xprog.(state.line).cmd with
  | Rem _ -> { state with line = next_line state }
  | Print e -> print_value (eval_exp state.line state.xenv e) ;
    print_newline () ;
    { state with line = next_line state }
  | Let(v,e) -> let ev = eval_exp state.line state.xenv e
    in { state with line = next_line state ;
      xenv = add v ev state.xenv }
  | Goto n -> { state with line = n }
  | Input v -> let x = try read_int ()
    with Failure "int_of_string" -> 0
    in { state with line = next_line state;
      xenv = add v (Vint x) state.xenv }
  | If (t,n) -> match eval_exp state.line state.xenv t with
    | Vbool true -> { state with line = n }
    | Vbool false -> { state with line = next_line state }
    | _ -> runerr state.line ;;
val eval_cmd : state_exec -> state_exec = <fun>
```

При каждом вызове функция перехода из одного состояние в другое `eval_cmd` ищет текущую строку, выполняет ее и затем устанавливает номер следующей строки как текущую строку. Если мы достигли последней строки программы, то номеру текущей строки присваивается значение

-1, что позволит нам остановить программу.

Вычисление программы Будет рекурсивно применять функцию перехода, до тех пор, пока не получим состояние, в котором номер текущей строки равен -1.

```
# let rec run state =
  if state.line = -1 then state else run (eval_cmd state) ;;
val run : state_exec -> state_exec = <fun>
```

5.2.6 Последние штрихи

Осталось лишь реализовать мини-редактор и собрать воедино все части программы, реализованные ранее.

Функция **insert** вставляет новую строку в соответствующее место в программе.

```
# let rec insert line p = match p with
  [] -> [line]
  | l :: prog ->
    if l.num < line.num then l::(insert line prog)
    else if l.num=line.num then line::prog
    else line :: l :: prog ;;
val insert : line -> line list -> line list = <fun>
```

Функция **print_prog** выводит на экран код программы.

```
# let print_prog prog =
  let print_line x = print_string (pp_line x) ; print_newline () in
  print_newline () ;
  List.iter print_line prog ;
  print_newline () ;;
val print_prog : line list -> unit = <fun>
```

Функция **one_command** либо добавляет строку, либо выполняет команду. Она управляет состоянием интерактивного цикла, состоящего из программы и окружения. Это состояние, представленное типом **loop_state**, отличается от состояния выполнения программы.

```
# type loop_state = { prog:program; env:environment } ;;
# exception End ;;
```

```
# let one_command state =
  print_string "> " ; flush stdout ;
```

```

try
  match parse (input_line stdin) with
    Line l -> { state with prog = insert l state.prog }
  | List -> (print_prog state.prog ; state )
  | Run
    -> let tprog = assemble state.prog in
      let xstate = run { line = 0; xprog = tprog; xenv = state.env }
      in
        {state with env = xstate.xenv }
  | PEnd -> raise End
with
  LexerError -> print_string "Illegal character\n"; state
  | ParseError -> print_string "syntax error\n"; state
  | RunError n ->
    print_string "runtime error at line ";
    print_int n ;
    print_string "\n";
    state ;;
val one_command : loop_state -> loop_state = <fun>

```

Главной функцией является `go`, она запускает интерактивный цикл Basic.

```

# let go () =
  try
    print_string "Mini-BASIC version 0.1\n\n";
    let rec loop state = loop (one_command state) in
      loop { prog = []; env = [] }
  with End -> print_string "See you later...\n";;
val go : unit -> unit = <fun>

```

Цикл реализуется локальной функцией `loop`. Цикл заканчивается при возбуждении исключения `End` функцией `one_command`.

Пример C+/C-

Вернемся к игре C+/C-, описанной в главе 2 на странице 86. Вот ее эквивалент, написанный на Basic.

```

10 PRINT "Give the hidden number: "
20 INPUT N
30 PRINT "Give a number: "
40 INPUT R

```

```
50 IF R = N THEN 110
60 IF R < N THEN 90
70 PRINT "C-"
80 GOTO 30
90 PRINT "C+"
100 GOTO 30
110 PRINT "CONGRATULATIONS"
```

Пример запуска данной программы.

```
> RUN
Give the hidden number:
64
Give a number:
88
C-
Give a number:
44
C+
Give a number:
64
CONGRATULATIONS
```

5.2.7 Что дальше?

Данный интерпретатор Basic обладает минимумом возможностей. Тем, кто желает его обогатить, мы предлагаем следующие расширения:

1. *числа с плавающей запятой*: наш интерпретатор распознает лишь целые числа, булевы значения и строки. Добавьте числа с плавающей запятой, а так же соответствующие операции в грамматику языка. Кроме лексического анализа, необходимо изменить вычисление с учетом приведения типов между целыми числами и числами с плавающей запятой.
2. *векторы*: то есть добавить к синтаксису инструкцию **DIM var[x]**, при помощи которой объявляется вектор **var** размером **x**. А так же выражение **var[i]**, которое ссылается на **i**-ый элемент вектора **var**.
3. *директивы*: добавить директивы **SAVE "file_name"** и **LOAD "file_name"** для записи файла на диск и загрузки с диска соответственно.

4. *подпрограммы*: вызов подпрограммы осуществляется инструкцией **GOSUB** номер строки. Эта инструкция реализует переход на этот номер и сохраняет при этом номер строки из которой произошел вызов. Инструкция **RETURN** продолжает выполнение программы со строки, которая следует за последним вызовом **GOSUB**, если она существует, или выходит из программы. Для этого, вычисление должно контролировать не только окружение, но и стек, в котором хранятся адреса возврата различных вызовов **GOSUB**. При помощи инструкции **GOSUB** можно объявлять рекурсивные подпрограммы.

5.3 Minesweeper

Напомним вкратце правила игры: необходимо исследовать минное поле, не попав при этом ни на одну из них. Минное поле — это двумерный массив, несколько элементов которого содержат скрытые мины, а остальные пусты. В начале игры, клетки поля закрыты и игрок должен их исследовать одну за одной. Игрок побеждает, если он исследовал все клетки поля, не содержащие мин.

На каждом этапе игры игрок может либо “открыть” клетку либо пометить ее как “заминированной”. Если он открыл клетку с миной, то игрок проигрывает. Иначе клетка меняет свой вид и в ней выводится число заминированных клеток вокруг (максимум 8). Если игрок пометил клетку как заминированную, то он не может ее открыть, не убрав метку.

Разделим реализацию программы на три части.

1. описание абстрактной игры, состоящей из внутреннего представления минного поля и функций управляющих этим представлением.
2. графическое описание игры с соответствующими функциями рисования клеток.
3. часть, отвечающая за взаимодействие между двумя предыдущими частями.

5.3.1 Абстрактное минное поле

В этой части мы рассмотрим минное поле как абстрактную сущность, не уделяя внимания способам вывода на экран.

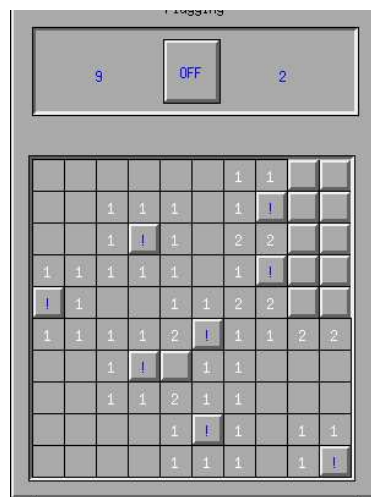


Рис. 5.4: Копия экрана

Конфигурация Минное поле характеризуется своими размерами и числом заминированных клеток. Сгруппируем эти три параметра в одной записи и определим конфигурацию по умолчанию: размер 10x10 и 15 мин.

```
# type config = {
  nbcols : int ;
  nbrows : int ;
  nbmines : int };;
# let default_config = { nbcols=10; nbrows=10; nbmines=15 } ;;
```

Минное поле Вполне естественно будет определить минное поле как двумерный массив. Так же нужно уточнить натуру элементов массива и информацию, которую необходимо знать для каждого из них. Состояние клетки может быть:

- клетка заминирована?
- клетка открыта?
- клетка помечена?
- сколько из окружающих ячеек заминировано?

Последняя информация не так важна, это значение можно вычислить в нужный момент. Но будет проще сделать данный расчет раз и навсегда в начале игры.

Клетка представлена записью с четырьмя полями, хранящими выше-указанную информацию.

```
# type cell = {
  mutable mined : bool ;
  mutable seen : bool ;
  mutable flag : bool ;
  mutable nbm : int
} ;;
```

Двумерный массив есть вектор векторов клеток.

```
# type board = cell array array ;;
```

Итератор Далее в программе нам понадобится применять функцию к каждой клетке поля. Реализуем универсальный итератор `iter_cells`, который применяет указанную функцию `f` к каждому элементу массива конфигурации `cf`.

```
# let iter_cells cf f =
  for i=0 to cf.nbcols-1 do for j=0 to cf.nbrows-1 do f (i,j) done
  done ;;
val iter_cells : config -> (int * int -> 'a) -> unit = <fun>
```

Здесь мы получили хорошее сочетание функционального и императивного стилей. Для итеративного применения функции с побочным эффектом (она не возвращает результат) ко всем элементам массива, используется функция высшего порядка (функция, аргумент которой есть другая функция).

Инициализация Расположение заминированных клеток будет определяться случайно. Для r и c , число линий и колонок заминированного поля, и m число мин, необходимо получить список состоящий из m чисел в интервале от 1 до $r * c$. В алгоритме подразумевается, что $m < r * c$, однако необходимо сделать эту проверку в программе.

Простым решением этой задачи будет создание пустого списка. Затем мы генерируем случайное число и размещаем его в список, если оно уже не принадлежит списку. Повторим эту операцию до тех пор, пока в списке не будет m чисел. Для этих целей, воспользуемся следующими функциями из модулей `Random` и `Sys`:

Random.int : int->int для входного аргумента n возвращает случайное число в диапазоне от 0 до $n - 1$.

Random.init : int->unit инициализация генератора случайных чисел.

Sys.time : unit->float возвращает время использования процессора в миллисекундах с начала запуска программы. Эта функция используется при инициализации генератора случайных чисел при каждой новой игре.

Модули, содержащие эти функции, описаны в главе 7 на страницах 239 и 261 соответственно.

У функции, случайно выбирающей заминированные клетки, два аргумента: общее число клеток (`cr`) и число мин(`m`). Она возвращает список из m линейных координат.

```
# let random_list_mines cr m =
  let cell_list = ref []
  in while (List.length !cell_list) < m do
    let n = Random.int cr in
    if not (List.mem n !cell_list) then cell_list := n :: !cell_list
  done ;
  !cell_list ;;
val random_list_mines : int -> int -> int list = <fun>
```

Мы не можем заявить что эта функция, так она написана, закончится через определенное число итераций. Если генератор случайных чисел достаточно хороший, то можно лишь с уверенностью сказать, что вероятность того что эта функция не закончится равна нулю. Откуда мы получаем парадоксальное суждение: «функция закончится если она выполняется бесконечно». Однако, на практике эта функция никогда нас не подводила, поэтому удовольствуемся данным негарантированным определением для генерации списка заминированных клеток.

Для того, чтобы при каждой новой игре получить разные заминированные клетки, нужно инициализировать генератор случайных чисел. Инициализировать будем при помощи процессорного времени в миллисекундах, которое истекло с момента запуска программы.

```
# let generate_seed () =
  let t = Sys.time () in
  let n = int_of_float (t*.1000.0)
  in Random.init(n mod 100000) ;;
val generate_seed : unit -> unit = <fun>
```

Практика показывает, что одна и та же программа затрачивает в среднем одинаковое время, из-за чего мы получаем схожий результат функции `generate_seed`. В связи с этим, функция `Unix.time` предпочтительней (см. гл. 17).

Во время инициализации минного поля, а так же в ходе игры, необходимо знать для данной клетки число окружающих заминированных клеток (функция `neighbors`). При вычислении множества соседних клеток, мы учитываем крайние клетки, у которых меньше соседей, чем у тех что находятся в середине поля (функция `valid`).

```
# let valid cf (i,j) = i>=0 && i<cf.nbcols && j>=0 && j<cf.nbrows ;;
val valid : config -> int * int -> bool = <fun>
# let neighbors cf (x,y) =
  let ngb = [x-1,y-1; x-1,y; x-1,y+1; x,y-1; x,y+1; x+1,y-1; x+1,y;
             x+1,y+1]
```

```

in List.filter (valid cf) ngb ;;
val neighbors : config -> int * int -> (int * int) list = <fun>

```

Инициализация минного поля реализуется функцией `initialize_board`, она выполняет четыре задачи:

1. генерация списка заминированных клеток
2. создание двумерного массива состоящего из разных клеток
3. пометка заминированных клеток
4. вычисление количества заминированных соседних клеток для каждой незаминированной клетки

В этой функции используется несколько локальных функций, которые мы вкратце опишем.

- `cell_init`: получить начальные значения клетки.
- `copy_cell_init`: инициализация клетки.
- `set_mined`: заминировать клетку.
- `count_mined_adj`: для конкретной клетки вычислить количество соседних заминированных клеток.
- `set_count`: для конкретной незаминированной клетки обновить количество заминированных соседних клеток.

```

# let initialize_board cf =
  let cell_init () = { mined=false; seen=false; flag=false; nbm=0 } in
  let copy_cell_init b (i,j) = b.(i).(j) <- cell_init() in
  let set_mined b n = b.(n / cf.nbrows).(n mod cf.nbrows).mined <-
    true
  in
  let count_mined_adj b (i,j) =
    let x = ref 0 in
    let inc_if_mined (i,j) = if b.(i).(j).mined then incr x
    in List.iter inc_if_mined (neighbors cf (i,j)) ;
    !x
  in
  let set_count b (i,j) =
    if not b.(i).(j).mined

```

```

    then b.(i).(j).nbm <- count_mined_adj b (i,j)
  in
  let list_mined = random_list_mines (cf.nbc*cf.nbr) cf.nbmines
    in
  let board = Array.make_matrix cf.nbc cf.nbr (cell_init ())
  in iter_cells cf (copy_cell_init board) ;
    List.iter (set_mined board) list_mined ;
    iter_cells cf (set_count board) ;
    board ;;
val initialize_board : config -> cell array array = <fun>

```

Открытие клетки Если во время игры игрок открывает клетку у которой нет ни одного заминированного соседа, он с уверенностью может открыть соседние клетки, до тех пор пока есть такие клетки. Для того, чтобы избавить игрока от этой нудного момента игры, не требующего размышления, игра сама откроет нужные клетки в этом случае. При открытии клетки функция `cells_to_see` возвращает список клеток которые можно открыть.

Идея алгоритма достаточно просто излагается: если у открытой клетки есть заминированные соседи, то список ограничивается лишь этой самой клеткой, иначе список состоит из ее соседей, а так же из соседей ее соседей. Трудность состоит в том, чтобы написать незацикливающуюся программу, так как клетка является соседом самой себе. Надо избежать проверки по несколько раз одной и той же клетки поля. Для того, чтобы знать какая клетка была открыта, создадим вектор `visited` булевых значений. Размер вектора соответствует количеству клеток. Если элемент вектора равен `true`, это значит что соответствующая клетка была исследована. Рекурсивный поиск клеток осуществляется только среди непомеченных клеток.

Используя список соседних клеток, функция `relevant`, вычисляет два под-списка. Каждый под-список состоит из незаминированных, неоткрытых, непомеченных игроком и непроверенных клеток (которым соответствует значение `false` в векторе `visited`, прим. пер.). Первый под-список включает соседей, у которых есть как минимум один заминированных сосед, второй состоит из соседних клеток без заминированных соседей. Эти клетки помечаются как проверенные. Заметим, что помеченные игроком клетки, даже если они на самом деле незаминированы, исключаются из списков. Смысл метки заключается как раз в том, чтобы избежать открытия клетки.

Функция `cells_to_see_rec` рекурсивно реализует цикл поиска. Ис-

ходя из обновляемого списка клеток, которые необходимо проверить, она возвращает список клеток, которые будут открыты. Начальный список содержит лишь последнюю открытую клетку, которая помечена как проверенная.

```
# let cells_to_see bd cf (i,j) =
  let visited = Array.make_matrix cf.nbcfs cf.nbrows false in
  let rec relevant = function
    [] -> ([],[])
  | ((x,y) as c) :: t ->
    let cell = bd.(x).(y)
    in if cell.mined || cell.flag || cell.seen || visited.(x).(y)
      then relevant t
      else let (l1,l2) = relevant t
            in visited.(x).(y) <- true ;
              if cell.nbm=0 then (l1,c::l2) else (c :: l1,l2)
  in
  let rec cells_to_see_rec = function
    [] -> []
  | ((x,y) as c) :: t ->
    if bd.(x).(y).nbm<>0 then c :: (cells_to_see_rec t)
    else let (l1,l2) = relevant (neighbors cf c)
          in (c :: l1) @ (cells_to_see_rec (l2 @ t))
  in visited.(i).(j) <- true ;
    cells_to_see_rec [(i,j)] ;;
val cells_to_see :
  cell array array -> config -> int * int -> (int * int) list = <fun>
```

С первого взгляда, аргумент `cells_to_see_rec` увеличивается между двумя последовательными вызовами функции, тогда как рекуррентное отношение основывается на этом аргументе. Соответственно, может возникнуть вопрос — заканчивается ли эта функция? Использование вектора `visited` гарантирует, что уже проверенная клетка не будет включена в результат `relevant`. В то же время, клетки, которые добавляются в список проверяемых клеток, происходят из `relevant`. Этим гарантируется, что определенная клетка будет возвращена `relevant` всего один раз и в следствии она будет представлена в единственном экземпляре в списке проверяемых клеток. Раз количество клеток ограничено, значит наша функция тоже закончится.

На этом неграфическая часть игры заканчивается. Рассмотрим стиль программирования, которым мы воспользовались. Выбор изменяемых структур данных вынуждает использовать императивный стиль с цик-

лами и присвоением. Однако, для решения дополнительных задач мы применили списки и функции обработки в функциональном стиле. Стиль программирования предписывается структурами данных, которыми мы манипулируем. Функция `cells_to_see` тому хороший пример: она использует списки и, вполне естественно, эта функция написана в функциональном стиле. Для хранения информации о проверенных клетках мы используем вектор, обновление вектора осуществляется императивно. Конечно, мы могли бы сделать тоже самое в чисто функциональном стиле, используя список. Однако, цена этого решения выше, чем для предыдущего (поиск элемента в списке напрямую зависит от размера списка, тогда как для вектора время поиск есть константная величина) и оно не является более простым.

5.3.2 Игровой интерфейс

Эта часть игры зависит от структур данных, которые представляют состояние игры (см. стр. 192). Цель этой части — изобразить на экране различные компоненты игры, как на рисунке 5.5. Для этого воспользуемся функциями рисования блоков, описанными ранее на стр. 139.

Свойства различных компонентов игры описываются следующими параметрами.

<code># let b0 = 3 ;;</code>	<code># let h1 = w1 ;;</code>
<code># let w1 = 15 ;;</code>	<code># let h2 = 30 ;;</code>
<code># let w2 = w1 ;;</code>	<code># let h3 = w5+20 + 2*b0 ;</code>
<code># let w4 = 20 + 2*b0 ;;</code>	<code># let h4 = h2 ;;</code>
<code># let w3 = w4*default_config.ncols + 2*b0 ;;</code>	<code># let h5 = 20 + 2*b0 ;;</code>
<code># let w5 = 40 + 2*b0 ;;</code>	<code># let h6 = w5 + 2*b0 ;;</code>

При помощи этих параметров мы расширим базовую конфигурацию игры (значения с типом `config`) и определим новую запись `window_config`. В поле `cf` содержится минимальная конфигурация. Каждой компоненте, изображенной на экране, ассоциируем блок: основное окно (поле `main_box`), минное поле (поле `field_box`), диалоговое окно (поле `dialog_box`) из двух блоков (поля `d1_box` и `d2_box`), кнопка для пометки (поле `flag_box`) и текущая клетка (поле `current_box`).

```
# type window_config = {
  cf : config ;
  main_box : box_config ;
  field_box : box_config ;
  dialog_box : box_config ;
```

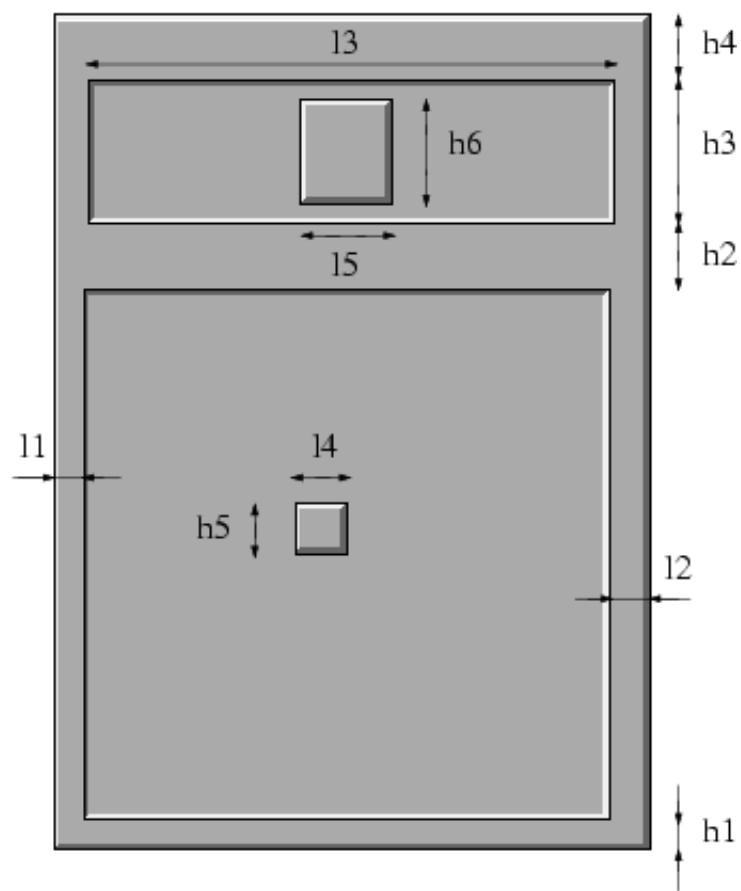



Рис. 5.5: Основное окно игры

```

d1_box : box_config ;
d2_box : box_config ;
flag_box : box_config ;
mutable current_box : box_config ;
cell : int*int -> (int*int) ;
coor : int*int -> (int*int)
} ;;

```

Кроме этого, значение с типом `window_config` содержит две функции:

- `cell`: для координат данной клетки возвращает координаты ее блока
- `coor`: для координат пикселя окна возвращает координаты соответствующей клетки.

Конфигурация Определим функцию, которая создает графическую конфигурацию (с типом `window_config`) в соответствии с минимальной конфигурацией (с типом `config`) и вышеописанных параметров. Значения параметров некоторых компонент зависят друг от друга. Например, ширина основного блока зависит от ширины блока минного поля, которых в свою очередь зависит от количества столбцов. Чтобы не вычислять одно и то же по несколько раз, мы будем постепенно инициализировать эти поля. В отсутствии специальных функций данный этап инициализации немного нудный.

```

# let make_box x y w h bw r =
  { x=x; y=y; w=w; h=h; bw=bw; r=r; b1_col=gray1; b2_col=gray3;
    b_col=gray2 } ;;
val make_box : int -> int -> int -> int -> int -> relief ->
  box_config =
  <fun>
# let make_wcf cf =
  let wcols = b0 + cf.nbcols*w4 + b0
  and hrows = b0 + cf.nbrows*h5 + b0 in
  let main_box = let gw = (b0 + w1 + wcols + w2 + b0)
    and gh = (b0 + h1 + hrows + h2 + h3 + h4 + b0)
    in make_box 0 0 gw gh b0 Top
  and field_box = make_box w1 h1 wcols hrows b0 Bot in
  let dialog_box = make_box ((main_box.w - w3) / 2)
    (b0+h1+hrows+h2)
    w3 h3 b0 Bot
in

```

```

let d1_box = make_box (dialog_box.x + b0) (b0 + h1 + hrows + h2)
                    ((w3-w5)/2-(2*b0)) (h3-(2*b0)) 5 Flat in
let flag_box = make_box (d1_box.x + d1_box.w)
                    (d1_box.y + (h3-h6) / 2) w5 h6 b0 Top in
let d2_box = make_box (flag_box.x + flag_box.w)
                    d1_box.y d1_box.w d1_box.h 5 Flat in
let current_box = make_box 0 0 w4 h5 b0 Top
in { cf = cf;
    main_box = main_box; field_box=field_box; dialog_box=
        dialog_box;
    d1_box=d1_box;
    flag_box=flag_box; d2_box=d2_box; current_box =
        current_box;
    cell = (fun (i,j) -> ( w1+b0+w4*i , h1+b0+h5*j)) ;
    coor = (fun (x,y) -> ( (x-w1)/w4 , (y-h1)/h5 )) } ;;
val make_wcf : config -> window_config = <fun>

```

Вывод клеток Теперь нам предстоит определить функции вывода клеток в различных случаях: клетка может быть открыта или закрыта, содержать или нет информацию. Вывод (блока) текущей клетки всегда будет осуществляться (поле `cc_bcf`).

Определим две функции изменяющие конфигурацию текущей клетки; одна закрывает клетку, другая открывает ее.

```

# let close_ccell wcf i j =
    let x,y = wcf.cell (i,j)
    in wcf.current_box <- {wcf.current_box with x=x; y=y; r=Top} ;;
val close_ccell : window_config -> int -> int -> unit = <fun>
# let open_ccell wcf i j =
    let x,y = wcf.cell (i,j)
    in wcf.current_box <- {wcf.current_box with x=x; y=y; r=Flat} ;;
val open_ccell : window_config -> int -> int -> unit = <fun>

```

В зависимости от ситуации, необходимо выводить информацию на клетках. Для каждого случая мы определяем функцию.

Вывод закрытой клетки:

```

# let draw_closed_cc wcf i j =
    close_ccell wcf i j;
    draw_box wcf.current_box ;;
val draw_closed_cc : window_config -> int -> int -> unit = <fun>

```

Вывести открытую клетку с числом мин:

```
# let draw_num_cc wcf i j n =
  open_ccell wcf i j ;
  draw_box wcf.current_box ;
  if n<>0 then draw_string_in_box Center (string_of_int n)
                                     wcf.current_box Graphics.white ;;
val draw_num_cc : window_config -> int -> int -> int -> unit = <fun>
```

Вывод клетки, содержащей мину:

```
# let draw_mine_cc wcf i j =
  open_ccell wcf i j ;
  let cc = wcf.current_box
  in draw_box wcf.current_box ;
    Graphics.set_color Graphics.black ;
    Graphics.fill_circle (cc.x+cc.w/2) (cc.y+cc.h/2) (cc.h/3) ;;
val draw_mine_cc : window_config -> int -> int -> unit = <fun>
```

Вывод заминированной и помеченной клетки:

```
# let draw_flag_cc wcf i j =
  close_ccell wcf i j ;
  draw_box wcf.current_box ;
  draw_string_in_box Center "!" wcf.current_box Graphics.blue ;;
val draw_flag_cc : window_config -> int -> int -> unit = <fun>
```

Вывод ошибочно помеченной клетки:

```
# let draw_cross_cc wcf i j =
  let x,y = wcf.cell (i,j)
  and w,h = wcf.current_box.w, wcf.current_box.h in
  let a=x+w/4 and b=x+3*w/4
  and c=y+h/4 and d=y+3*h/4
  in Graphics.set_color Graphics.red ;
    Graphics.set_line_width 3 ;
    Graphics.moveto a d ; Graphics.lineto b c ;
    Graphics.moveto a c ; Graphics.lineto b d ;
    Graphics.set_line_width 1 ;;
val draw_cross_cc : window_config -> int -> int -> unit = <fun>
```

В ходе игры, выбор соответствующей функции вывода клетки осуществляется следующим:

```
# let draw_cell wcf bd i j =
  let cell = bd.(i).(j)
  in match (cell.flag, cell.seen, cell.mined) with
```

```

        (true,_,_) -> draw_flag_cc wcf i j
    | (_,false,_) -> draw_closed_cc wcf i j
    | (_,_,true) -> draw_mine_cc wcf i j
    | _           -> draw_num_cc wcf i j cell.nbm ;;
val draw_cell : window_config -> cell array array -> int -> int ->
    unit = <fun>

```

Для вывода клеток в конце игры, воспользуемся специальной функцией. Она немного отличается от предыдущих тем, что к концу все клетки должны быть открыты. К тому же, на ошибочно помеченных клетках выводится красный крест.

```

# let draw_cell_end wcf bd i j =
    let cell = bd.(i).(j)
    in match (cell.flag, cell.mined) with
        (true,true) -> draw_flag_cc wcf i j
    | (true,false) -> draw_num_cc wcf i j cell.nbm; draw_cross_cc
        wcf i j
    | (false,true) -> draw_mine_cc wcf i j
    | (false,false) -> draw_num_cc wcf i j cell.nbm ;;
val draw_cell_end : window_config -> cell array array -> int -> int
    -> unit = <fun>

```

Вывод остальных компонентов Состояние режима отметки клеток отображается выпуклым или вогнутым блоком с надписью ON или OFF:

```

# let draw_flag_switch wcf on =
    if on then wcf.flag_box.r <- Bot else wcf.flag_box.r <- Top ;
    draw_box wcf.flag_box ;
    if on then draw_string_in_box Center "ON" wcf.flag_box Graphics.
        red
    else draw_string_in_box Center "OFF" wcf.flag_box Graphics.blue ;;
val draw_flag_switch : window_config -> bool -> unit = <fun>

```

Выведем надпись о предназначении помечающей кнопки.

```

# let draw_flag_title wcf =
    let m = "Flagging" in
    let w,h = Graphics.text_size m in
    let x = (wcf.main_box.w-w)/2
    and y0 = wcf.dialog_box.y+wcf.dialog_box.h in
    let y = y0+(wcf.main_box.h-(y0+h))/2
    in Graphics.moveto x y ;

```

```

    Graphics.draw_string m ;;
val draw_flag_title : window_config -> unit = <fun>

```

На протяжении всей игры число клеток, которые осталось открыть и число помеченных клеток выводится в диалоговом окне с обеих сторон помечающей кнопки.

```

# let print_score wcf nbcto nbfc =
    erase_box wcf.d1_box ;
    draw_string_in_box Center (string_of_int nbcto) wcf.d1_box
    Graphics.blue ;
    erase_box wcf.d2_box ;
    draw_string_in_box Center (string_of_int (wcf.cf.nbmines-nbfc)) wcf
    .d2_box
    ( if nbfc>wcf.cf.nbmines then Graphics.red else Graphics.blue ) ;;
val print_score : window_config -> int -> int -> unit = <fun>

```

Чтобы нарисовать начальное минное поле, нужно вывести (число линий)×(число столбцов) раз закрытую клетку. Хотя это и один и тот же рисунок каждый раз необходимо нарисовать и заполнить прямоугольный и четыре трапеции, на это может уйти немало времени. Для ускорения этого процесса, воспользуемся следующей техникой: нарисуем один раз клетку, захватим ее в виде растрового изображения (bitmap) и затем скопируем эту кнопку в нужных местах.

```

# let draw_field_initial wcf =
    draw_closed_cc wcf 0 0 ;
    let cc = wcf.current_box in
    let bitmap = draw_box cc ; Graphics.get_image cc.x cc.y cc.w cc.h in
    let draw_bitmap (i,j) = let x,y=wcf.cell (i,j)
    in Graphics.draw_image bitmap x y
    in iter_cells wcf.cf draw_bitmap ;;
val draw_field_initial : window_config -> unit = <fun>

```

В конце игры, все минное поле открывается и на неправильно помеченных клетках ставится красный крест.

```

# let draw_field_end wcf bd =
    iter_cells wcf.cf (fun (i,j) -> draw_cell_end wcf bd i j) ;;
val draw_field_end : window_config -> cell array array -> unit = <
fun>

```

И наконец, основная функция вывода на экран открывает графический контекст и выводит начальное состояние различных компонент.

```

# let open_wcf wcf =

```

```

Graphics.open_graph ( " " ^ (string_of_int wcf.main_box.w) ^ "x" ^
                      (string_of_int wcf.main_box.h) ) ;
draw_box wcf.main_box ;
draw_box wcf.dialog_box ;
draw_flag_switch wcf false ;
draw_box wcf.field_box ;
draw_field_initial wcf ;
draw_flag_title wcf ;
print_score wcf ((wcf.cf.nbrows*wcf.cf.nbcols)-wcf.cf.nbmines) 0 ;;
val open_wcf : window_config -> unit = <fun>

```

Заметим, что все графические функции используют конфигурацию с типом `window_config`. Это делает их независимыми от расположения компонент игры. Если мы пожелаем изменить это расположение, код функций останется неизменным, необходимо будет лишь обновить конфигурацию.

5.3.3 Взаимодействие между программой и игроком

Определим возможные действия игрока:

- выбрать, нажатием кнопки, режим пометки или открытия клетки.
- нажать на одну из клеток, для того чтобы ее открыть или пометить.
- нажать на клавишу 'q', для того чтобы покинуть игру.

Напомним, что событие `Graphics` должно быть связано с записью (`Graphics.status`), в которой хранится информация о текущем состоянии клавиатуры и мыши в момент возникновения события. Все события мыши, кроме нажатия на кнопку пометки или на клетку минного поля, игнорируются. Чтобы различать оба события, создадим соответствующий тип.

```
# type clickon = Out | Cell of (int*int) | SelectBox ;;
```

Действия нажатия и отпускания кнопки мыши соответствуют двум разным событиям. Если оба события произошли на одной и той же компоненте (клетка минного поля или кнопка пометки), то такой клик считается правильным и обрабатывается.

```
# let locate_click wcf st1 st2 =
  let clickon_of st =
    let x = st.Graphics.mouse_x and y = st.Graphics.mouse_y
```

```

in if x>=wcf.flag_box.x && x<=wcf.flag_box.x+wcf.flag_box.w
    &&
    y>=wcf.flag_box.y && y<=wcf.flag_box.y+wcf.flag_box.h
    then SelectBox
    else let (x2,y2) = wcf.coor (x,y)
        in if x2>=0 && x2<wcf.cf.nbcols && y2>=0 && y2<wcf.cf
            .nbrows
            then Cell (x2,y2) else Out
in
let r1=clickon_of st1 and r2=clickon_of st2
in if r1=r2 then r1 else Out ;;
val locate_click :
window_config -> Graphics.status -> Graphics.status -> clickon = <
fun>

```

Сердце программы находится в функции `loop` и заключается в ожидании и обработке событий. Эта функция похожа на `skel`, описанной на странице 146, однако здесь мы точнее определяем тип события мыши. Условия остановки цикла следующие:

- нажатие на клавишу `q` (или `Q`), что рассматривается как желание остановить программу.
- в случае если игрок открыл заминированную клетку, тогда партия проиграна.
- в случае если игрок открыл все незаминированные клетки, тогда партия выиграна.

Объединим данные, необходимые функциям обрабатывающим взаимодействие с игроком, в записи `minesw_cf`.

```

# type minesw_cf =
{ wcf : window_config; bd : cell array array;
  mutable nb_flagged_cells : int;
  mutable nb_hidden_cells : int;
  mutable flag_switch_on : bool } ;;

```

Эти поля соответствуют:

- `wcf`: графическая конфигурация.
- `bd`: матрица клеток.

- `nb_flagged_cells`: булево значение, означающее что игра находится или нет в режиме пометки.
- `nb_hidden_cells`: число помеченных клеток
- `flag_switch_on`: число незаминированных и неоткрытых клеток

Теперь мы готовы, к тому чтобы написать основной цикл.

```
# let loop d f_init f_key f_mouse f_end =
  f_init ();
  try
    while true do
      let st = Graphics.wait_next_event
        [Graphics.Button_down;Graphics.Key_pressed]
      in if st.Graphics.keypressed then f_key st.Graphics.key
        else let st2 = Graphics.wait_next_event [Graphics.Button_up
          |
        in f_mouse (locate_click d.wcf st st2)
    done
  with End -> f_end ();;
val loop :
```

```
minesw_cf ->
(unit -> 'a) -> (char -> 'b) -> (clickon -> 'b) -> (unit -> unit)
-> unit = <fun>
```

Функции инициализации, окончания и обработки клавиатуры достаточно банальны.

```
# let d_init d () = open_wcf d.wcf
let d_end () = Graphics.close_graph()
let d_key c = if c='q' || c='Q' then raise End;;
val d_init : minesw_cf -> unit -> unit = <fun>
val d_end : unit -> unit = <fun>
val d_key : char -> unit = <fun>
```

Для обработки событий мыши нам понадобится несколько дополнительных функций.

- `flag_cell`: реакция на клик на клетку в режиме пометки.
- `ending`: конец игры. В этом случае открыть все заминированные клетки, вывести сообщение о победе или проигрыше и ожидать действия с клавиатуры или мыши для того, чтобы закрыть программу.

- **reveal**: реакция на клик на клетку в режиме открытия (режим пометки отключен).

```

# let flag_cell d i j =
  if d.bd.(i).(j).flag
  then ( d.nb_flagged_cells <- d.nb_flagged_cells - 1;
         d.bd.(i).(j).flag <- false )
  else ( d.nb_flagged_cells <- d.nb_flagged_cells + 1;
         d.bd.(i).(j).flag <- true );
  draw_cell d.wcf d.bd i j;
  print_score d.wcf d.nb_hidden_cells d.nb_flagged_cells;;
val flag_cell : minesw_cf -> int -> int -> unit = <fun>

# let ending d str =
  draw_field_end d.wcf d.bd;
  erase_box d.wcf.flag_box;
  draw_string_in_box Center str d.wcf.flag_box Graphics.black;
  ignore(Graphics.wait_next_event
          [Graphics.Button_down;Graphics.Key_pressed]);
  raise End;;
val ending : minesw_cf -> string -> 'a = <fun>

# let reveal d i j =
  let reveal_cell (i,j) =
    d.bd.(i).(j).seen <- true;
    draw_cell d.wcf d.bd i j;
    d.nb_hidden_cells <- d.nb_hidden_cells - 1
  in
    List.iter reveal_cell (cells_to_see d.bd d.wcf.cf (i,j));
    print_score d.wcf d.nb_hidden_cells d.nb_flagged_cells;
    if d.nb_hidden_cells = 0 then ending d "WON";;
val reveal : minesw_cf -> int -> int -> unit = <fun>

Функция, обрабатывающая события мыши, сопоставляет значение
типа clickon.

# let d_mouse d click = match click with
  Cell (i,j) ->
    if d.bd.(i).(j).seen then ()
    else if d.flag_switch_on then flag_cell d i j
    else if d.bd.(i).(j).flag then ()
    else if d.bd.(i).(j).mined then ending d "LOST"

```

```

      else reveal d i j
    | SelectBox ->
      d.flag_switch_on <- not d.flag_switch_on;
      draw_flag_switch d.wcf d.flag_switch_on
    | Out -> () ;;
val d_mouse : minesw_cf -> clickon -> unit = <fun>

```

При создании конфигурации игры, нам необходимо три параметра: число линий, число колонок и число мин.

```

# let create_minesw nb_c nb_r nb_m =
  let nbc = max default_config.nbcols nb_c
  and nbr = max default_config.nbrows nb_r in
  let nbm = min (nbc*nbr) (max 1 nb_m) in
  let cf = { nbcols=nbc ; nbrows=nbr ; nbmines=nbm } in
  generate_seed () ;
  let wcf = make_wcf cf in
  { wcf = wcf ;
    bd = initialize_board wcf.cf ;
    nb_flagged_cells = 0 ;
    nb_hidden_cells = cf.nbrows*cf.nbcols-cf.nbmines ;
    flag_switch_on = false } ;;
val create_minesw : int -> int -> int -> minesw_cf = <fun>

```

Функция, запускающая игру, сначала создает конфигурацию игры с помощью трех выше указанных параметров и запускает цикл обработки событий.

```

# let go nbc nbr nbm =
  let d = create_minesw nbc nbr nbm in
  loop d (d_init d) d_key (d_mouse d) (d_end);;
val go : int -> int -> int -> unit = <fun>

```

Вызов `go 10 10 10` запускает игру, показную на изображении 5.4.

Что дальше?

Из данной программы можно сделать самостоятельный исполняемый файл. В главе 6 объяснено как это сделать. После этого, можно сделать игру более удобной, передавая размер минного поля в командной строке. Передача аргументов рассматривается в главе 7, где приводится пример для данной игры (см. стр. 263).

Другое интересное расширение — научить машину саму открывать клетки. Для этого необходимо уметь определять следующий правильный

ход и играть его первым в этом случае. Затем вычислить вероятность присутствия мины для каждой клетки и играть клетку, для которой это значение минимально.

Часть II

Средства разработки

Содержимое второй части

Представленные здесь компоненты окружения являются частью дистрибутива Objective CAML. Это различные компиляторы, многочисленные библиотеки, средства анализа программы, средства лексического и синтаксического анализа и интерфейса с языком C.

Objective CAML является компилируемым языком, с двумя способами генерации кода:

- byte-code выполняемый виртуальной машиной
- нативный код, выполняемый напрямую процессором

Интерактивный цикл Objective CAML использует byte-code для выполнения введенных фраз. Он является первейшим средством помощи при разработке. Интерактивный цикл позволяет типизировать, компилировать и быстро проверять определения функций. К тому же он выдает значения входных аргументов и возврата функций.

Другие привычные средства разработки входят в дистрибутив: вычисление зависимостей между файлами, отладка и анализ (profiling). При помощи средств отладки программы мы можем пошагово выполнять программы, устанавливать метки останова и просмотреть значения переменных. Средство анализа выдает число вызовов функций, время затраченное на вызов функции или на выполнение определенной части кода. Последние две тулзы доступны только на платформах Unix.

Богатство языка происходит не только из его core (?), но и из различных библиотек; множество повторноиспользуемых программ, входящих в дистрибутив языка. Objective CAML тому пример. Графическая библиотека, включенная в дистрибутив, уже не раз упоминалась. Далее в книге, мы опишем немало новых библиотек. Они вносят большое число новых возможностей, однако обратной стороной медали являются некоторые трудности, в особенности в строгости типов.

Какой бы не была богатой библиотека, всегда необходимо уметь связываться с другим языком программирования. Дистрибутив языка Objective CAML содержит все необходимое для интерфейса с языком C. То

есть мы можем вызывать функции С в коде Objective CAML и наоборот. Разница в управлении памятью в Objective CAML и С может стать препятствием в понимании и использовании этого интерфейса. Эта разница заключается в том, что в Objective CAML имеется автоматический сборщик мусора.

И в С и в Objective CAML мы можем динамически выделять память и таким образом достаточно гибко управлять пространством в соответствии с нуждами программы. Естественно, это имеет смысл лишь если мы можем затем высвободить память для других нужд выполнения программы. Автоматический сборщик освобождает программиста от этой задачи, которая является частым источником ошибок и берет на себя управление сборкой памяти. Эта особенность является одним из элементов надежности языка Objective CAML.

Однако, данный механизм отражается (влияет?) на представление данных. То есть для того, чтобы правильно связывать код на Objective CAML и С, необходимо четко представлять себе принципы управления памятью.

В 6 главе мы рассмотрим базовые элементы системы Objective CAML: виртуальная машина, компиляторы и библиотеки. Здесь мы так же затронем различные методы компиляции и сравним переносимость с эффективностью.

В 7 дается глобальный обзор множества типов, функций и исключений, поставляемых с дистрибутивом. Однако, она не заменяет официального руководства, в котором подробно описаны эти библиотеки. Но здесь мы рассматриваются новые возможности предоставляемые некоторыми модуля библиотек. В частности, можно привести пример форматированного вывода, системный интерфейс и persistence of values

В 8 представлены различные методы автоматической сборки мусора, а затем описан принцип, используемый сборщиком Objective CAML.

В 9 мы рассмотрим способы отладки программ Objective CAML. Несмотря на некоторые недостатки, при помощи этих средств можно легко обнаружить проблемное место программы.

В 10 описываются некоторые проблемы лексического и синтаксического анализа языка: библиотека регулярный выражений, средства `osamlex` и `osamlyacc`, использование потоков (`stream`).

Сюжет 11 главы — интерфейс с языком С. На сегодня, язык программирования не может быть полностью изолированным от других. При помощи данного интерфейса, программы на Objective CAML могут вызвать функцию на С и передать ей значения Objective CAML и наоборот. Основное затруднение интерфейса заключается в модели памяти. По этой причине, желательно прочитать главу 8.

В 12 представлены два приложения: обогащенная графическая библиотека, основанная на иерархической структуре, как AWT² JAVA, классическая программа поиска пути в графе с наименьшей стоимостью с новым графическим интерфейсом и кэш-памятью.

²Abstract Windowing Toolkit

Глава 6

Компиляция и переносимость

Введение

Для того, чтобы текст программы превратился в исполняемый модуль, необходимо выполнить несколько операций. Эти операции сгруппированы в процессе компиляции. В ходе этого процесса, строится абстрактное синтаксическое дерево (как для интерпретатора Basic, стр. 171), затем оно превращается в последовательность инструкций для реального процессора или для виртуальной машины. В последнем случае, для того чтобы выполнить программу, необходим интерпретатор инструкций виртуальной машины. В каждом случае, результат компиляции должен быть связан с библиотекой времени выполнения, входящей в дистрибутив. Она может меняться в зависимости от процессора и операционной системы. В нее входят примитивные функции (такие как операции над числами, системный интерфейс) и администратор памяти.

В Objective CAML существует два компилятора. Первый из них — это компилятор для виртуальной машины, в результате которого мы получаем байт-код. Второй компилятор создает программу, состоящую из машинного кода (native code), выполняемого реальным процессором, как Intel, Motorola, SPARC, PA-RISC, Power-PC или Alpha. Компилятор байт-кода отдает предпочтение переносимости кода, тогда как компилятор в машинный код увеличивает скорость выполнения. Интерактивный интерпретатор, который мы видели в первой части, использует байт-код. Каждая введенная фраза компилируется и затем выполняется в окружении символов, определенных в течении интерактивной сессии.

План главы

В этой главе представлены различные способы компиляции программ на Objective CAML и проводится сравнение по переносимости и эффективности этих способов. В первом разделе мы обсудим различные этапы компиляторов Objective CAML. Во втором — различные способы компиляции и синтаксис, необходимый для получения исполняемых файлов. В третьем разделе, мы увидим как создать автономный исполняемый файл, который может быть выполнен независимо от установленного дистрибутива Objective CAML. И, наконец, четвертый раздел сравнивает различные способы компиляции в ракурсе переносимости и эффективности выполнения.

6.1 Этапы компиляции

Исполняемый файл получается в результате этапов трансляции и компоновки, изображенных в таблице 6.1.

макроподстановка	Исходный текст программы
компиляция	Исходный текст программы
компоновка	Программа на ассемблере
редактирование связей	Машинные инструкции
	Исполняемый код

Таблица 6.1: Порядок создания исполняемого файла

Макроподстановка заключается в подстановке одних частей текста вместо других при помощи системы макросов. При компиляции исходный код программы транслируется в команды ассемблера. После компоновки мы получим файл, состоящий из машинных инструкций. И, наконец, при редактировании связей добавляется библиотека, в основном состоящая из администратора памяти, а так же связь с основными объектами операционной системы (такие как файлы, каталоги, процессы и т.д.)

6.1.1 Компиляторы Objective CAML

Детали различных этапов генерации кода компиляторов Objective CAML представлены таблице 6.2. Внутреннее представление кода программы, сгенерированного компилятором, называется промежуточным языком (ПЯ).

лексический анализ	Последовательность символов
синтаксический анализ	Последовательность лексем
семантический анализ	Синтаксическое дерево
генерация промежуточного кода	Маркированное синтаксическое дерево
оптимизация промежуточного кода	Последовательность кода в ПЯ
генерация псевдокода	Последовательность кода в ПЯ
	Программа на ассемблере

Таблица 6.2: Этапы компиляции

При помощи лексического анализа из последовательности символов, мы получаем последовательность лексических элементов (лексем). В основном лексемы соответствуют целым числам, числам с плавающей запятой, строкам и идентификаторам. Сообщение **Illegal character** генерируется на этом этапе анализа.

Синтаксический анализ создает синтаксическое дерево, проверяя при этом последовательность лексем с точки зрения правил грамматики языка. Сообщение **Syntax error** указывает на то, что анализируемая часть кода не соответствует правилам грамматики.

При семантическом анализе просматривается синтаксическое дерево, здесь нас интересует другой аспект корректности программы. На этом этапе, в Objective CAML происходит вывод типа и, если он прошел успешно, то выводимый тип является *самым общим типом* для выражения или объявления. Сообщения об ошибке типа генерируются на данном этапе. В этот же момент выявляются случаи, в которых тип члена последовательности отличен от **unit**. Здесь также генерируются другие предупреждения, возникшие на пример при сопоставлении (не исчерпаемость, неиспользуемые ветви сопоставления).

При генерации и оптимизации промежуточного кода не выводится

никаких сообщений об ошибках или предупреждений. Эти этапы манипуляции промежуточными структурами позволяют факторизовать разработку различных компиляторов Objective CAML.

Генерация исполняемого модуля — это финальный этап компиляции, который зависит от компилятора.

6.1.2 Описание байт-код компилятора

Виртуальная машина Objective CAML называется Zinc (от « Zinc Is Not Caml »). Она была создана Гзавье Леруа (Xavier Leroy) и описана в ([Ler90]). Это имя было выбрано, для того чтобы подчеркнуть разницу между первыми реализациями языка Caml, основанного на виртуальной машине CAM (от Categorical Abstract Machine, см. [CCM87]).

На рисунке 6.1 изображена виртуальная машина Zinc. В первой части представлен интерпретатор связанный с библиотекой. Вторая часть соответствует компилятору, который генерирует байт-код для машины Zinc. Третья часть содержит библиотеки, идущие вместе с компилятором. Они более подробно описаны в главе 7.

Графические символы, используемые на рисунке 6.1, являются стандартными для компиляции. Простой блок символизирует файл, написанный на языке, который указан внутри блока. Двойной блок представляет интерпретацию одного языка программой, написанной на другом языке. Тройной блок — исходный язык компилируется в машинный при помощи компилятора, написанном на третьем языке. Символы, соответствующие интерпретаторам и компиляторам, изображены на рисунке 6.2.

Пояснение к рисунку 6.1:

- BC: байт-код машины Zinc
- C: код C
- .o: объектный-файл, зависящий от используемой архитектуры
- μ : микропроцессор
- OC (v1 или v2): код Objective CAML

Замечание

Основная часть компилятора Objective CAML написана на языке Objective CAML. Переход с версии v1 на версию v2 изображен на второй части рисунка 6.1.

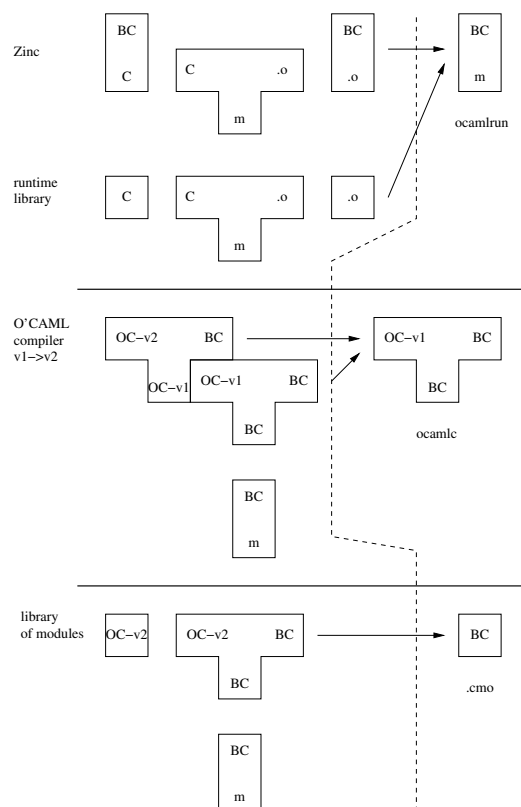


Рис. 6.1: Виртуальная машина Zinc

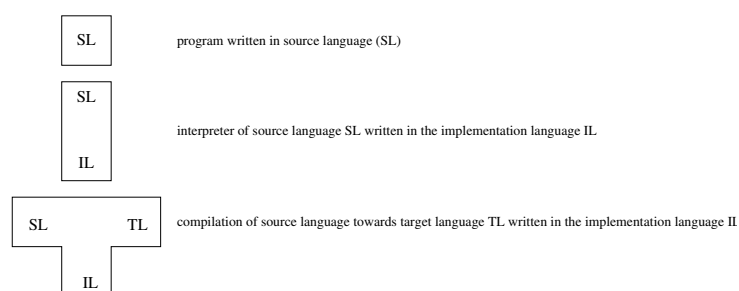


Рис. 6.2: Символы интерпретаторов и компиляторов.

6.2 Типы компиляции

Дистрибутив языка зависит от типа процессора и операционной системы. В дистрибутив Objective CAML для каждой архитектуры (пара: процессор, операционная система) входит интерактивная среда интерпретатора, байт-код компилятор, и, в большинстве случаев, компилятор машинного кода для данной архитектуры.

6.2.1 Названия команд

В таблице 6.3 приводятся названия различных компиляторов, входящих в дистрибутив Objective CAML. Первые четыре из них являются частью каждого дистрибутива.

ocaml	интерактивная среда интерпретатора
ocamlrun	интерпретатор байт-кода
ocamlc	компилятор в байт-кода
ocamlopt	компилятор машинного кода
ocamlc.opt	оптимизированный компилятор байт-кода
ocamlopt.opt	оптимизированный компилятор машинного кода
ocamlmktop	конструктор новых сред интерпретатора

Таблица 6.3: Команды компиляции

Оптимизированные компиляторы были сами скомпилированы компилятором машинного кода, при этом получается более быстрый исполняемый файл.

6.2.2 Элементы компиляции

Элемент компиляции соответствует наименьшей части программы на Objective CAML, которая может быть скомпилирована. Для интерактивного интерпретатора таким элементом является фраза на языке Objective CAML, тогда как для пакетных компиляторов таким элементом является пара файлов: исходный код и файл интерфейсов, где файл интерфейсов не обязателен. Если он отсутствует, то все глобальные объявления файла-исходника будут видны другим элементам компиляции. Создание интерфейсных файлов будет рассмотрено в главе 13 посвященной модульному программированию. Эти файлы отличаются друг от друга расширением имени файла.

6.2.3 Расширения файлов Objective CAML

В таблице 6.4 представлены расширения файлов используемые для программ Objective CAML и C.

расширение	значение
.ml	исходник
.mli	интерфейсный файл
.cmo	объектный файл (байт-код)
.cma	объектный файл библиотеки
.cmi	скомпилированный интерфейсный файл
.cmx	объектный файл (нативный)
.cmxa	объектный файл библиотеки (нативный)
.c	исходник на C
.o	объектный файл C (нативный)
.a	объектный файл библиотеки C (нативный)

Таблица 6.4: Расширения файлов

Файлы `example.ml` и `example.mli` формируют элемент компиляции. Скомпилированный интерфейсный файл (`example.cmi`) может быть использован нативным и байт-код компилятором. Файлы на C используются для интерфейса между Objective CAML и библиотеками, написанными на языке C (11).

6.2.4 Байт-код компилятор

Общая форма команды компиляции следующая:

```
command options file_name
```

Objective CAML подчиняется тому же правилу.

```
ocamlc -c example.ml
```

Перед параметрами компилятора ставится символ '-', как это принято в системе Unix. Расширения файлов интерпретируются в соответствии с таблицей 6.4. В приведенном примере, файл `example.ml` рассматривается как исходник на Objective CAML и после его компиляции будет сгенерировано два файла `example.cmo` и `example.cmi`. Опция '-c' указывает компилятору, что необходимо скомпилировать только объектный файл. Без этой опции компилятор `ocamlc` создаст исполняемый файл `a.out`, то есть в данном случае он реализует редактирование связей.

-a	создать библиотеку
-c	компиляция, без редактирования связей
-o имя_файла	указывает имя исполняемого файла
-linkall	связать со всеми используемыми библиотеками
-i	вывести все скомпилированные глобальные объявления
-pp команда	использовать <i>команду</i> как препроцессор
-unsafe	отключить проверку индексов
-v	вывести версию компилятора
-w список	выбрать, в соответствии со списком, уровень предупреждений
-impl файл	указывает, что <i>файл</i> это исходник на Caml (см. таб. 6.7)
-intf файл	указывает, что <i>файл</i> есть интерфейсный файл на Caml (.mli)
-I каталог	добавить <i>каталог</i> в список каталогов

Таблица 6.5: Основные опции байт-код компилятора

нити	-thread (18, стр. 387)
отладка	-g, -noassert (9, стр. 301)
автономный исполняемый файл	-custom, -cclib, -ccopt, -cc (см. стр. 6.3)
<i>runtime</i>	-make-runtime , -use-runtime
C interface	-output-obj (11, стр. 349)

Таблица 6.6: Другие опции байт-код компилятора

В таблице 6.5 описаны основные опции байт-код компилятора. Остальные опции приведены в таблице 6.6.

Чтобы получить возможные опции байт-код компилятора, используйте параметр `-help`.

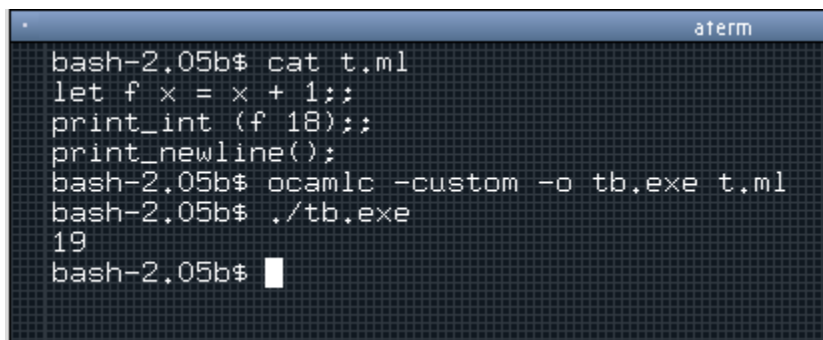
В таблице 6.7 описаны различные уровни предупреждений. Уровень — это переключатель (включено/выключено), который символизируется буквой. Заглавная буква включает данный уровень, а прописная выключает.

Principal levels	
A/a	включить/выключить все сообщения
F/f	частичное применение в последовательности
P/p	для неполного сопоставления
U/u	для лишних случаев в сопоставлении
X/x	включить/выключить все остальные сообщения
M/m и V/v	for hidden object (см. главу 14)

Таблица 6.7: Описание предупреждений компилятора

По умолчанию установлен максимальный уровень предупреждений (A).

Пример использования байт-кода компилятора изображен на рисунке 6.3.



```

bash-2.05b$ cat t.ml
let f x = x + 1;;
print_int (f 18);;
print_newline();
bash-2.05b$ ocamlc -custom -o tb.exe t.ml
bash-2.05b$ ./tb.exe
19
bash-2.05b$

```

Рис. 6.3: Пример работы с байт-код компилятором

6.2.5 Нативный компилятор

Принцип действия компилятора машинного кода такой же что и байт-код компилятора, с разницей в сгенерированных файлах. Большинство опции компиляции те же что проведены в таблицах 6.5 и 6.6. Однако, в

таблице 6.6 необходимо удалить опции связанные с *runtime*. Специфические опции для нативного компилятора приведены в таблице 6.8. Уровни предупреждений остаются теми же.

-compact	оптимизация размера кода
-S	сохранить код на ассемблере в файле
-inline уровень	установить уровень “агрессивности” разложения функций

Таблица 6.8: Специальные опции для нативного компилятора

Разложение есть расширенный случай макроподстановки. Для функций с фиксированным числом аргументов, на месте их вызова подставляется тело самих функций. Таким образом несколько вызовов функции соответствуют новым копиям ее тела. Разложением мы избавляемся от затрат, возникающих при инициализации вызова и возврата функции. Однако, расплатой за это будет увеличение размера кода. Основные уровни разложения функции следующие:

- 0: разложение разрешено, в случае если оно не увеличивает размер кода
- 1: значение по умолчанию, при этом допускается небольшое увеличение кода
- $n > 1$: увеличивает терпимость к увеличению кода

6.2.6 Интерактивный интерпретатор

У интерактивного интерпретатора всего две опции:

- *-I каталог*: добавить каталог в начало списка каталогов, где находятся скомпилированные файлы или исходники.
- *-unsafe*: не проверять выход индекса за пределы для векторов и строк

Интерактивный интерпретатор распознает несколько директив, позволяющих интерактивно изменить его функционирование. Все эти директивы описаны в таблице 6.9, они начинаются с символа ‘#’ и заканчиваются привычным ‘;’.

Директивы, относящиеся к каталогам, подчиняются правилам используемой операционной системы.

Директивы загрузки имеют отличное от других директив функционирование. Директива *#use* считывает указанный файл, как если бы он

#quit;;	выйти из интерпретатора
#directory каталог ;;	добавить каталог в список путей
#cd каталог ;;	сменить текущий каталог
#load объектный_файл ;;	загрузить файл .смо
#use исходный_файл ;;	скомпилировать и загрузить исходных файл
#print_depth глубина ;;	изменить глубину вывода на экран
#print_length ширина ;;	аналогично для ширины
#install_printer функция ;;	указать функцию вывода на экран
#remove_printer функция ;;	переключится на стандартный вывод на экран
#trace функция ;;	трассировка аргументов функции
#untrace функция ;;	отменить трассировку
#untrace_all ;;	отменить все трассировки

Таблица 6.9: Директивы интерпретатора.

был введен интерактивно. Директива **#load** загружает файл с расширением .смо. В этом последнем случае, глобальные декларации этого файла недоступны напрямую. Для этого необходимо использовать синтаксис с точкой. Если в файле `example.ml` содержится глобальное объявление `f`, то после загрузки байт-код (`#load "example.смо";;`), значение `f` доступно как `Example.f`. Обратите внимание, что первая буква файла — заглавная. Этот синтаксис происходит из модульной системы Objective CAML (см. главу 13, стр. 375). Директивы настройки глубины и ширины вывода на экран позволяют контролировать формат выводимых значений, что удобно, когда мы имеем дело с большими данными.

Директивы трассировки аргументов и результата функции особенно полезны при отладке программ. Этот вариант использования интерактивного интерпретатора рассматривается более подробно в главе об анализе программ (9).

На рисунке 6.4 изображен сеанс работы в интерпретаторе.

6.2.7 Создание интерпретатора

При помощи команды **ocamlmktop** мы можем создать новый цикл интерпретатора, в который загружены определенные библиотеки или модули. Кроме того, что мы избавляемся от **#load** в начале сеанса работы, это так же необходимо для подключения библиотек на C.

Опции этой команды являются частью опций байт-код компилятора (ocamlc):

-cclib имя_библиотеки, -ccopt опции, -custom, -I каталог -o имя_файла

В главе о графическом интерфейсе (4 стр. 129) используется эта ко-

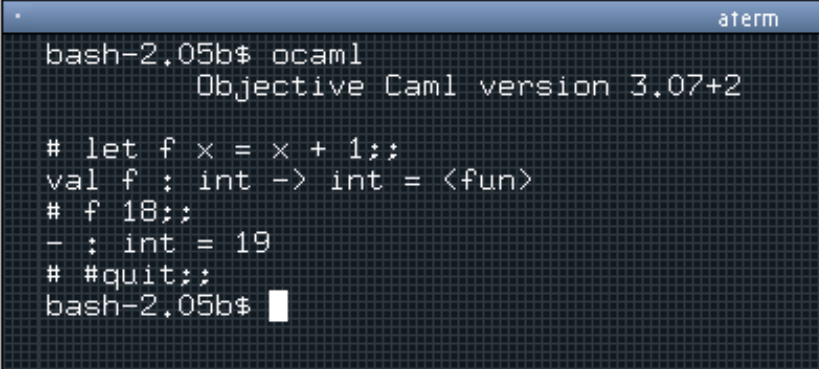
A screenshot of a terminal window titled 'aterm'. The prompt is 'bash-2.05b\$'. The user enters 'ocaml', and the prompt changes to 'Objective Caml version 3.07+2'. The user then enters several OCaml expressions: '# let f x = x + 1;;', 'val f : int -> int = <fun>', '# f 18;;', '- : int = 19', and '# #quit;;'. The prompt returns to 'bash-2.05b\$'.

Рис. 6.4: Сеанс работы в интерпретаторе.

манда для создания цикла содержащего библиотеку **Graphics** следующим способом:

```
ocamlmktop -custom -o mytoplevel graphics.cma -cclib \  
-I/usr/X11/lib -cclib -lX11
```

Эта команда создает исполняемый файл **mytoplevel**, содержащий байт-код библиотеки **graphics.cma**. Этот файл является автономным (**-custom**, смотрите следующий раздел) и он связан с библиотекой **X11** (**libX11.a**), которая располагается в каталоге **/usr/X11/lib**.

6.3 Автономный исполняемый файл

Автономный исполняемый файл (**standalone**) есть исполняемый файл, независящий от дистрибутива Objective CAML на используемой машине. С одной стороны, бинарный файл упрощает распространение программ, с другой стороны, он не зависит от расположения дистрибутива Objective CAML на компьютере.

Компилятор машинного кода всегда генерирует автономный исполняемый файл. Тогда как без опции **-custom**, байт-код компилятор создаст исполняемый файл, который нуждается в интерпретаторе **ocamlrun**. Пусть файл **example.ml** содержит следующее:

```
let f x = x + 1;;  
print_int (f 18);;  
print_newline();;
```

Следующая команда создает файл **example.exe**:

```
ocamlc -o example.exe example.ml
```

Полученный файл, размером около 8Kb, может быть выполнен при помощи следующей команды:

```
\$ ocamlrun example.exe  
19
```

Интерпретатор выполняет команды виртуальной машины Zinc, хранящиеся в файле `example.exe`

Для операционной системы Unix, в первой строке такого файла находится путь к байт-код интерпретатору, например:

```
#!/usr/local/bin/ocamlrun
```

При помощи этой строки, файл может быть напрямую запущен интерпретатором. Таким образом, выполнение файла `example.exe` запускает файл, путь к которому находится в первой строке. Если интерпретатор не найдет по этому пути программу-интерпретатор, сообщение `Command not found` будет выведено на экран и выполнение остановится.

Та же компиляция с опцией `-custom` создаст автономный файл с именем `exauto.exe`:

```
ocamlc -custom -o exauto.exe example.ml
```

Теперь размер файла около 85Kb, так как он содержит не только байт-код, но и интерпретатор. Этот файл может быть самостоятельно запущен или скомпилирован на другую машину (той же архитектуры и той же операционной системы) для запуска.

Warning

Размер исполняемого файла не соответствует размеру процесса во время выполнения. Размер статического файла на диске не учитывает размер динамически выделяемой памяти (8).

6.4 Переносимость и эффективность

Интерес компиляции для абстрактной машины заключается в том что, мы получим код который может быть запущен на какой угодно реальной архитектуре. Основным неудобством является интерпретация виртуальных инструкций. Нативный компилятор создает более эффективный код, однако он предназначен лишь для одной определенной архитектуры. Поэтому, желательно чтобы у нас был выбор типа компиляции в зависимости от разрабатываемого программного обеспечения. Автоно-

мий исполняемого файла, то есть его независимость от установленного дистрибутива Objective CAML, лишает приложение переносимости.

6.4.1 Автономность и переносимость

Для того чтобы получить автономный исполняемый файл, компилятор отредактировал связи байт-код файла `t.cmo` с runtime библиотекой, интерпретатором байт-кода и некоторым кодом на C. Предполагается, что в системе установлен компилятор C. Вставка машинного кода означает, что файл не может быть выполнен на другой архитектуре или системе.

Дело обстоит по другому в случае байт-код файла. Так как виртуальные инструкции одни и те же для всех архитектур, лишь присутствие интерпретатора имеет значение. Такой файл может быть запущен на любой системе, где присутствует команда `ocamlrun`, являющаяся частью дистрибутива Objective CAML для Sparc и Solaris, Intel и Windows и т.д. Всегда желательно использовать одни и те же версии компилятора и интерпретатора.

Переносимость файлов в байт-код формате позволяет напрямую распространять и сразу же использовать библиотеки в подобной форме, на машине с интерпретатором Objective CAML.

6.4.2 Эффективность выполнения

Компилятор байт-кода генерирует инструкции машины Zinc, они будут интерпретированы `ocamlrun`. Эта фаза интерпретации кода негативно влияет на скорость выполнения. Мы можем представить себе интерпретатор Zinc в действии как большое сопоставление с образцом (сопоставление `match ... with`), где каждая инструкция есть образец и ответвления расчета изменяю состояние стека и счетчика (адрес следующей инструкции). Несмотря на то, что данное сопоставление оптимизировано, оно все же снижает эффективность выполнения.

Следующий пример, хоть он и не тестирует все составные части языка, иллюстрирует разницу во времени выполнения между байт-код и автономным вариантом расчета чисел Fibonacci. Пусть у нас есть следующая программа `fib.ml`:

```
let rec fib n =  
  if n < 2 then 1  
  else (fib (n-1)) + (fib(n-2));;  
  
и главная программа main.ml:
```



```
for i = 1 to 10 do  
  print_int (Fib.fib 30);  
  print_newline()  
done;;
```

Откомпилируем их следующим образом:

```
$ ocamlc -o fib.exe fib.ml main.ml  
$ ocamlpt -o fibopt.exe fib.ml main.ml
```

Мы получим два файла `fib.exe` и `fibopt.exe`. При помощи команды `time` системы Unix на процессоре Pentium 350 с установленной операционной системой Linux, получим следующие значения:

fib.exe (байт-код)	fibopt.exe (машинный)
7 сек.	1 сек.

То есть для одной и той же программы скорость ее двух версий отличается в 7 раз. Данная программа не проверяет все свойства языка, полученная выгода значительно зависит от типа приложения.

Резюме

В этой главр мы увидели различные этапы компиляции Objective CAML. Byte-code компилятор создает переносимый двоичный код и таким образом позволяет распространять библиотеки в подобном формате. Автономные исполняемые файлы теряют эту особенность. Нативный компилятор отдает предпочтение эффективности полученного кода в ущерб переносимости.

Глава 7

Библиотеки

Введение

В наборе с каждым языком программирования идут программы, которые могут быть использованы программистом — они называются библиотеками. Количество и качество библиотек является одним из основных критериев удобства использования языка. Библиотеки можно разбить на два типа. В первый тип библиотек предоставляет типы и функции частого использования и которые могут быть определены языком. Второй — предоставляющий возможности, которые не могут быть определены языком. Иначе говоря, при помощи первого типа библиотек, мы избавляемся от переопределений таких вещей как стеки, очередь, и т.д., а второй тип расширяет возможности языка.

Большое количество библиотек поставляется в дистрибутиве Objective CAML. Они распространяются в виде скомпилированных файлов. Однако, любознательный читатель может найти исходные файлы библиотек в дистрибутиве исходников языка.

Библиотеки Objective CAML организованы по модулям, которые в свою очередь являются элементами компиляции. Каждый из них содержит глобальные определения типов, исключений и значений, которые могут быть использованы в программе. В данной главе, мы не будем рассматривать построение подобных модулей, а лишь использование существующих. В 13 главе мы вернемся к концепции модуля (логический элемент) и элемента компиляции и опишем язык описания модулей в Objective CAML. А в 11 мы обсудим включение кода написанного на другом языке в библиотеки Objective CAML, в частности интеграция кода на C в Objective CAML.

В дистрибутив Objective CAML входит preloaded библиотека (модуль

Pervasives), набор базовых модулей, называемых стандартной библиотекой и много других библиотек, добавляющих дополнительные возможности. Некоторые библиотеки лишь упоминаются в данной главе или они описываются в следующих главах.

План главы

В этой главе мы опишем библиотеки входящие в дистрибутив Objective CAML. Некоторые из них уже были представлены в предыдущих главах, например Graphics (см. главу 4) или лишь упомянуты, как библиотека Array. В первом разделе описана организация различных библиотек. Во втором разделе мы рассмотрим preloaded модуль Pervasives. В третьем разделе мы сделаем классификацию множества модулей, сгруппированных в стандартной библиотеки. И наконец в четвертом разделе подробно рассмотрим библиотеки точной арифметики и динамической загрузки кода.

7.1 Классификация и использование библиотек

Библиотеки дистрибутива Objective CAML могут быть разделены на три части. В первой содержатся preloaded глобальные объявления. Вторая, так называемая стандартная библиотека, имеет достаточно высокий уровень стабильности. Она разделена на 4 части:

- структуры данных
- input/output
- системный интерфейс
- синтаксический и лексический анализ

В третьей части библиотек находятся модули расширяющие возможности языка, как например библиотека Graphics (см. главу 4). В этой части мы можем найти библиотеки для обработки регулярных выражений (Str), точной арифметики (Num), системных вызовов Unix (Unix), нитей (Threads) и динамической загрузки byte-code (Dynlink).

Операции ввода/вывода и системный интерфейс стандартной библиотеки совместимы с различными операционными системами: Unix, Windows,

MacOS. Не все библиотеки третьей группы обладают такой особенностью. Также существует множество библиотек поставляемых независимо от дистрибутива Objective CAML.

Использование Для того, чтобы использовать модуль или библиотеку в программе, используют синтаксис с точкой, указав имя модуля, а затем имя объекта. К примеру, если необходим объект `f` из модуля `Name`, то мы пишем `Name.f`. Чтобы постоянно не добавлять префикс из имени модуля, можно открыть модуль и вызывать `f` напрямую.

Синтаксис `open Name`

С этого момента, все объявления модуля `Name` будут рассматриваться как глобальные в текущем окружении. Если какое-то объявление имеет одно и то же имя в двух библиотеках, то имя последней загруженной библиотеки перекрывает предыдущее объявление. Чтобы вызвать первое объявление, необходимо использовать синтаксис с точкой.

7.2 Автоматически загруженные библиотеки

Библиотека `Pervasives` всегда автоматически подгружается, как в интерактивной среде, так и при компиляции нативным компилятором. Эта библиотека всегда связывается (*linked*) и является начальным окружением языка. В ней содержатся следующие объявления:

типы : базовые типы (`int`, `char`, `string`, `float`, `bool`, `unit`, `exn`, `'a array`, `'a list`) и типы `'a option` (см. стр. 248) и (`'a`, `'b`, `'c`) `format` (см. стр. 295).

исключения : около десятка исключений, возбуждаемых `execution library`. Вот наиболее общие из них:

- `Failure of string`, возбуждаемое функцией `failwith`, примененной к строке.
- `Invalid_argument of string` указывающей что аргумент не может быть обработан функцией, которая возбудила исключение.
- `Sys_error of string` возбуждается в основном при операциях ввода/вывода, например при попытке открыть несуществующий файл.

- `End_of_file` возбуждается при достижении конца файла.
 - `Division_by_zero` возбуждается при целочисленном делении.
А так же системные исключения:
 - `Out_of_memory` и `Stack_overflow` возникающих при переполнении кучи и стека. Заметим, что исключение `Out_of_memory` не может быть отловлено программой, так как в этой ситуации уже невозможно выделить новое пространство памяти для продолжения расчета.
- Отслеживание исключения `Stack_overflow` меняется в зависимости от того как была скомпилирована программа: нативно или в `byte-code`. В первом случае это не возможно.

функции : около 140, половина из них соответствует функциям C из `execution library`. Среди них мы можем найти арифметические операторы и операторы сравнения, целочисленные функции и функции над числами с плавающей запятой, строковые функции, функции над указателями (`references`) и ввода/вывода. Необходимо заметить, что некоторые из этих объявлений на самом деле лишь синонимы объявлений из других модулей, но здесь они находятся по историческим и `implementation reasons`.

7.3 Стандартная библиотека

В стандартной библиотеке сгруппированы стабильные, платформонезависимые модули. На данный момент библиотека содержит 29 модулей, которые в свою очередь состоят из 400 функций, 30 типов, половина из которых абстрактные, 8 исключений, 10 под-модулей и 3 параметризованных модуля. Конечно, мы не будем детально рассматривать все объявления модулей, для этого существует справочное руководство [LRVD99]. Представим лишь модули вводящие новые концепции или сложные в использовании.

Стандартная библиотека может быть разбита на 4 большие части:

линейные структуры данных (15 модулей), некоторые из них мы уже видели в первой части.

ввод/вывод (4 модуля), для форматирования вывода, сохраняемость и создание криптографических ключей.

синтаксический и лексический анализ (4 модуля), их описание дано в главе 10 на стр. 319.

системный интерфейс при помощи которого мы можем передавать, анализировать параметры передане команде, перемещаться в каталогах и обращаться к файлам.

К этим четырем частям добавим пятую, в которой содержатся полезные функции манипуляции или создания структур данных, как например функция текстовой обработки или генерации псевдо-случайных чисел и т.д.

7.3.1 Утилиты

Модули, которые мы назовем «полезными», содержат:

- символы: модуль `Char` состоит в основном из функций перевода (conversion);
- клонирование объектов: ОП будет представлено в главе 14 на стр. 377;
- отложенное вычисление: модуль `Lazy` рассмотрен в первой части книги на стр. 118;
- генератор случайных чисел: модуль `Random`, описание которого проведено ниже.

Генератор случайных чисел

При помощи модуля `Random` можно генерировать случайные числа. Он реализует функцию генерации чисел, которая производит начальную установку счетчика при помощи списка чисел или одного числа. Для того чтобы эта функция не выдавала одно и тоже число, программист должен инициализировать ее разными числами. Основываясь на этом числе, функция вычисляет список случайных чисел. Однако функция, инициализированная одним и тем же числом, выдаст одинаковый список. Поэтому, для корректной установки счетчика, нам понадобится внешний источник, как например компьютерное время или время прошедшее с момента запуска прогаммы.

Ниже приведены функции модуля:

- инициализация: `init` с типом `int` -> `unit` и `full_init` с типом `int array` -> `unit` инициализирующие генератор. Вторая функция ожидает на входе вектор.

- генерировать случайное число: *bits* с типом `unit -> int` возвращает позитивное целое число, *int* с типом `int -> int` возвращает целое число между 0 и пределом, переданным в аргументе и *float*, которая возвращает число с плавающей запятой между 0 и пределом, переданным в аргументе.

7.3.2 Линейные структуры данных

В следующем списке представлены модули линейных структур данных:

- простые модули: `Array`, `String`, `List`, `Sort`, `Stack`, `Queue`, `Buffer`, `Hashtbl` (параметризован) и `Weak`;
- параметризованные модули: `Hashtbl` (с параметром `HashedType`), `Map` и `Set` (с параметром `OrderedType`).

Параметризованные модули построены на основе других модулей, благодаря чему они становятся еще более общими (*generic*). Создание параметризованных модулей будет представлено в главе 13 на стр. 375.

Линейные структуры простых данных

Имя модуля указывает на структуру данных, которыми он умеет манипулировать. Если тип абстрактный, то есть представление типа спрятано, то в соответствии с принятым соглашением, подобный тип именуется `t` внутри модуля. Такие модули реализуют следующие структуры:

- модуль `Array`: векторы
- модуль `List`: списки
- модуль `String`: символьные строки
- модуль `Hashtbl`: хэш-таблицы (абстрактный тип)
- модуль `Buffer`: расширяемые символьные строки (абстрактный тип)
- модуль `Stack`: стек (абстрактный тип)
- модуль `Queue`: очереди или FIFO (абстрактный тип)
- модуль `Weak`: вектор `weak` указателей.

Укажем так же последний модуль манипулирующий линейными структурами данных:

- модуль `Sort`: сортировка списков и векторов, объединение списков

Семейство общих функций За исключением модуля `Sort`, все остальные модули определяют структуры данных, функции создания таких структур и доступа к элементам этих структур, а так же функции манипуляции, включая функцию перевода в другие типы данных. Только модуль `List` обходится без физического изменения данных. Мы не станем давать полное описание всех функций, ограничимся лишь семейством функций, используемых этими модулями. Модули `List` и `Array` будут рассмотрены в деталях, так как это наиболее часто встречающиеся структуры в императивном и функциональном программировании.

Следующие функции можно найти во всех или почти всех модулях:

- функция `length`, которая вычисляет размер типа переданного в аргументе
- функция `clear` очищает структуры, если она изменяемая
- функция `add` для добавления элемента, она может иметь другое имя, в зависимости от “обычаев” (как например `push` для стека)
- функция доступа к *i*-тому элементу, обычно называемой `get`
- функция удаления элемента (часто первого) `remove` или `take`

По тому же принципу, в нескольких модулях можно встретить функции просмотра и обработки элементов:

- `map`: применяет функцию ко всем элементам структуры и возвращает новую структуру с результатами.
- `iter`: схожа с `map`, но возвращает `()`.

Для структур с индексированными элементами существуют следующие функции.

- `fill`: изменяет часть структуры на новое значение;
- `blit`: копирует часть структуры в другую того же типа;
- `sub`: копирует часть структуры и создает новую.

Модули `List` и `Array`

Описывая функции этих библиотек, мы сакцентируем внимание на особенностях и общие черты каждой из них. Для функций одинаковых для обоих модулей тип `t` означает `'a list` или `a' array`. Для функций специфичных одному модулю воспользуемся синтаксисом с точкой.

Схожие или аналогичные функции Первая из них - подсчет длины.

<code>length</code>	<code>:</code>	<code>'a t -> int</code>
---------------------	----------------	-----------------------------

Две функции для конкатенации двух структур или всех структур списка.

<code>append</code>	<code>:</code>	<code>'a t -> 'a t -> 'a t</code>
<code>concat</code>	<code>:</code>	<code>'a t list -> 'a t</code>

Каждый модуль включает функцию доступа к элементу структуры по позиции.

<code>List.nth</code>	<code>:</code>	<code>'a list -> int -> 'a</code>
<code>Array.get</code>	<code>:</code>	<code>'a array -> int -> 'a</code>

В связи с тем что Функция для доступа к *i*-тому элементу вектора *t* часто используется у нее есть укороченный синтаксис: *t.(i)*.

Две функции, позволяющие выполнить определенное действие (функцию) над всеми элементами структуры.

<code>iter</code>	<code>:</code>	<code>('a -> unit) -> 'a t -> unit</code>
<code>map</code>	<code>:</code>	<code>('a -> 'b) -> 'a t -> 'b t</code>

Чтобы вывести все элементы списка или вектора, воспользуемся *iter*.

```
# let print_content iter print_item xs =
  iter (fun x -> print_string("; print_item x; print_string")") xs;
  print_newline() ;;
val print_content : (('a -> unit) -> 'b -> 'c) -> ('a -> 'd) -> 'b
  -> unit =
  <fun>
# print_content List.iter print_int [1;2;3;4;5] ;;
(1)(2)(3)(4)(5)
- : unit = ()
# print_content Array.iter print_int [|1;2;3;4;5|] ;;
(1)(2)(3)(4)(5)
- : unit = ()
```

Функция *map* создает новую структуру, которая содержит результаты применения функции к элементам списка или вектора. Проверим это на векторе с изменяемым содержимым:

```
# let a = [|1;2;3;4|] ;;
val a : int array = [|1; 2; 3; 4|]
# let b = Array.map succ a ;;
val b : int array = [|2; 3; 4; 5|]
# a, b;;
- : int array * int array = [|1; 2; 3; 4|], [|2; 3; 4; 5|]
```

При помощи следующих итераторов можно создать частичное применение функции для каждого элемента вектора:

<code>fold_left</code>	:	<code>(('a -> 'b -> 'a) -> 'a -> 'b t -> 'a</code>
<code>fold_right</code>	:	<code>(('a -> 'b -> 'b) -> 'a t -> 'b -> 'b</code>

Этим итераторам необходимо указать базовый случай со значением по умолчанию, которое будет возвращено в случае если структура пустая.

```
fold_left f r [v1; v2; ...; vn] = f ... ( f (f r v1) v2 ) ... vn
fold_right f [v1; v2; ...; vn] r = f v1 ( f v2 ... (f vn r) ... )
```

При помощи этих функций можно легко превратить бинарную функцию в n-арную. Для коммутативной и ассоциативной операции итерация слева на право или справа налево одинакова:

```
# List.fold_left (+) 0 [|1;2;3;4|] ;;
- : int = 10
# List.fold_right (+) [|1;2;3;4|] 0 ;;
- : int = 10
# List.fold_left List.append [0] [|1|;|2|;|3|;|4|] ;;
- : int list = [0; 1; 2; 3; 4]
# List.fold_right List.append [|1|;|2|;|3|;|4|] [0] ;;
- : int list = [1; 2; 3; 4; 0]
```

Отметим, что пустой лист является нейтральным элементом слева и справа для двоичного сложения. Для этого частного случая следующие выражения эквивалентны:

```
# List.fold_left List.append [] [|1|;|2|;|3|;|4|] ;;
- : int list = [1; 2; 3; 4]
# List.fold_right List.append [|1|;|2|;|3|;|4|] [] ;;
- : int list = [1; 2; 3; 4]
```

Таким образом мы “создали” функцию *List.concat*.

Операции над списками Следующие полезные функции находятся в модуле *List*.

<code>List.hd</code>	:	<code>'a list -> 'a</code>
		первый элемент списка
<code>List.tl</code>	:	<code>'a list -> 'a</code>
		список, без первого элемента
<code>List.rev</code>	:	<code>'a list -> 'a list</code>
		список в обратном порядке
<code>List.mem</code>	:	<code>'a -> 'a list -> bool</code>
		тест на принадлежность списку
<code>List.flatten</code>	:	<code>'a list list -> 'a list</code>
		“разложить” список списков
<code>List.rev_append</code>	:	<code>'a list -> 'a list -> 'a list</code>
		то же что и <code>append (rev l1) l2</code>

Первые две функции являются частичными, если им передать пустой список, то исключение **Failure** будет возбуждено. Существует вариант *mem*: *memq*, использующий физическое равенство.

```
# let c = (1,2) ;;
val c : int * int = 1, 2
# let l = [c] ;;
val l : (int * int) list = [1, 2]
# List.memq (1,2) l ;;
- : bool = false
# List.memq c l ;;
- : bool = true
```

В модуле **List** имеется два особенных итератора, обобщающих булеву конъюнкцию и дизъюнкцию (*and/or*): *List.for_all* и *List.exists* определяются следующим образом.

```
# let for_all f xs = List.fold_right (fun x -> fun b -> (f x) & b) xs
  true ;;
val for_all : ('a -> bool) -> 'a list -> bool = <fun>
# let exists f xs = List.fold_right (fun x -> fun b -> (f x) or b) xs
  false ;;
val exists : ('a -> bool) -> 'a list -> bool = <fun>
```

В модуле **List** имеется различные варианты итераторов (*iter2*, *map2*, etc.), которые принимают на входе два листа и обходят их параллельно. В случае если листы имеют разную длину, возбуждается исключение **Invalid_argument**.

Следующие функции позволяют осуществлять поиск элемента в списке при помощи критерия в виде булевой функции:

<code>List.find</code>	:	<code>('a -> bool) -> 'a list -> 'a</code>
<code>List.find_all</code>	:	<code>('a -> bool) -> 'a list -> 'a list</code>

У функции `find_all` существует псевдоним `filter`.

A variant of the general search function is the partitioning of a list:

<code>List.partition</code>	:	<code>('a -> bool) -> 'a list -> 'a list * 'a list</code>
-----------------------------	---	--

Часто используемые функции из модуля `List` для разбиения или создания списка из списка пар:

<code>List.split</code>	:	<code>('a * 'b) list -> 'a list * 'b list</code>
<code>List.combine</code>	:	<code>'a list -> 'b list -> ('a * 'b) list</code>

И наконец, часто используемая структура, совмещающая список с парой: список ассоциаций. Такая структура удобна когда необходимо хранить данные ассоциированные ключу. Она состоит из списка пар, первый элемент которой ключ, а второй - связанное с этим ключом значение. Для подобных списков имеются следующие функции:

<code>List.assoc</code>	:	<code>'a -> ('a * 'b) list -> 'b</code>
<code>List.mem_assoc</code>	:	извлекает информацию связанную с ключом <code>'a -> ('a * 'b) list -> bool</code>
		проверяет существование ключа
<code>List.remove_assoc</code>	:	<code>'a -> ('a * 'b) list -> ('a * 'b) list</code>
		удаляет элемент связанный с ключом

У каждой из этих функций существует аналог, использующий физическое равенство вместо структурного: `List.assq`, `List.mem_assq` и `List.remove_assq`

Манипуляции над векторами Векторы, так часто используемые в императивном программировании, являются физически изменяемыми структурами. В модуле `Array` имеется функция для изменения значения элемента:

<code>Array.set</code>	:	<code>'a array -> int -> 'a -> unit</code>
------------------------	---	---

Как и функция `get`, у функции `set` имеется укороченная запись: `t.(i) <- a`

Существует 3 функции выделения памяти под вектор:

<code>Array.create</code>	:	<code>int -> 'a -> 'a array</code>
		создать вектор заданного размера, все элементы которого инициализированы
<code>Array.make</code>	:	<code>int -> 'a -> 'a array</code>
		укороченная запись для <code>create</code>
<code>Array.init</code>	:	<code>int -> (int -> 'a) -> 'a array</code>
		создать вектор заданного размера, все элементы которого инициализированы

Так как матрицы это часто встречающаяся структура, в модуле `Array` имеется две функции создания матриц:

<code>Array.create_matrix</code>	:	<code>int -> int -> 'a -> 'a array array</code>
<code>Array.make_matrix</code>	:	<code>int -> int -> 'a -> 'a array array</code>

Функция `set` превращается в функцию изменяющую значения интервала, который указан индексом началом и длиной:

<code>Array.fill</code>	:	<code>'a array -> int -> int -> 'a -> unit</code>
-------------------------	---	---

При помощи следующих функций можно скопировать целый вектор или часть вектора, указанного началом и длиной. При этом мы получаем новую структуру:

<code>Array.copy</code>	:	<code>'a array -> 'a array</code>
<code>Array.sub</code>	:	<code>'a array -> int -> int -> 'a array</code>

Копия или вырезка могут реализовываться и в существующий вектор:

<code>Array.blit</code>	:	<code>'a array -> int -> 'a array -> int -> int -> unit</code>
-------------------------	---	---

Первый целочисленный аргумент является индексом в первом векторе, второй аргумент - индекс второго вектора и третий число копируемых элементов. Функции `blit`, `sub` и `fill` возбуждают исключения `Invalid_argument`.

В связи с тем что к элементам массива обычно доступны при помощи индекса, были определены следующие два итератора:

<code>Array.iteri</code>	:	<code>(int -> 'a -> unit) -> 'a array -> unit</code>
<code>Array.mapi</code>	:	<code>(int -> 'a -> 'b) -> 'a array -> 'b array</code>

Они применяют функцию, первый аргумент которой является индексом желаемого элемента.

```
# let f i a = (string_of_int i) ^ ":" ^ (string_of_int a) in
  Array.mapi f [| 4; 3; 2; 1; 0 |] ;;
- : string array = [| "0:4"; "1:3"; "2:2"; "3:1"; "4:0" |]
```

В модуле **Array** не имеется функции-аналога *map* для списков, которая бы могла изменить содержимое каждой ячейки вектора на результат функции от значения этой ячейки. Но мы можем получить тоже самое при помощи функции **iteri**:

```
# let iter_and_set f t =
  Array.iteri (fun i -> fun x -> t.(i) <- f x) t ;;
val iter_and_set : ('a -> 'a) -> 'a array -> unit = <fun>
# let v = [|0;1;2;3;4|] ;;
val v : int array = [|0; 1; 2; 3; 4|]
# iter_and_set succ v ;;
- : unit = ()
# v ;;
- : int array = [|1; 2; 3; 4; 5|]
```

И наконец, модуль **Array** предоставляет две функции для создания списка из вектора и наоборот.

Array.of_list	:	a list -> 'a array
Array.to_list	:	a array -> 'a list

7.3.3 Ввод/Вывод

В стандартной библиотеке имеется четыре модуля ввода-вывода

- **Printf**: для форматированного вывода
- **Format**: для форматирования блоками с автоматической поддержкой переноса строки
- **Marshal**: реализует механизм постоянных (persistent) значений
- **Digest**: для создания уникальных ключей

Детали модуля **Marshal** будут приведены позже в этой главе, когда мы займемся обработкой постоянных данных (persistent data structures) (см. стр. 252).

Модуль **Printf**

При помощи модуля **Printf** мы можем форматировать текст на манер функции **printf** из стандартной библиотеки языка C. Желаемый формат представлен строкой, которая будет обработана в соответствии со стандартами функции **printf**, то есть используя символ **%**. Этот символ, если

Тип	Символ	Результат
целое	d или i	десятичное со знаком
символ	u	десятичное без знака
	x	шестнадцатеричное без знака в нижнем регистре
	X	то же самое в верхнем регистре
	c	символ
строка	s	строка
действительное	f	десятичное
	e или E	экспоненциальная запись
	g или G	то же самое
логический	b	true или false
специальный	a или t	функциональный параметр типа (out_channel -> 'a -> unit) -> 'a -> unit или out_channel -> unit

Таблица 7.1: Символы преобразования типов

за ним следует буква, определяет тип данных на данной позиции. Следующая запись “(x=%d, y=%d)” означает что вместо %d нужно вывести два целых числа.

Спецификация формата Формат указывает параметры для выводимой строки. Базовые типы: int, float, char и string будут преобразованы в строки и подставлены на их места в распечатываемой строке. В результате передачи значений 77 и 43 формату “(x=%d, y=%d)” получим строку “(x=77, y=43)”. Основные символы, определяющие тип преобразования приведены в таблице 7.1.

В формате можно так же установить выравнивание выводимых строк. Для этого необходимо указать размер в символах преобразования типа. Делается это так: между символом % и типом преобразования вставляется число, например %10d, после чего мы получаем выравнивание справа по десяти символам. Если размер результата превосходит размер ограничения, то это ограничение будет проигнорировано. Отрицательное значение реализует выравнивание слева. Для перевода действительных чисел стоит указать точность результата. В этом случае после % вставим точку и целое число. Следующая запись %.5f указывает, что мы желаем пять чисел после запятой.

Существует два специальных формата: **a** и **t**, они указывают функциональный аргумент. Typically, a print function defined by the user. В этом заключается специфичность Objective CAML.

Функции модуля Тип пяти функций модуля `Printf` приведен в таблице 7.2.

<code>fprintf</code>	:	<code>out_channel -> ('a, out_channel, unit) format -> 'a</code>
<code>printf</code>	:	<code>('a, out_channel, unit) format -> 'a</code>
<code>eprintf</code>	:	<code>('a, out_channel, unit) format -> 'a</code>
<code>sprintf</code>	:	<code>('a, unit, string) format -> 'a</code>
<code>bprintf</code>	:	<code>Buffer.t -> ('a, Buffer.t, string) format -> 'a</code>

Таблица 7.2: Функции форматирования `printf`

Функции *fprintf* ожидает три параметра: имя канала, формат и аргументы с типами, описанными форматом. Функции *eprintf* и *printf* есть специализированные версии для стандартного выхода и стандартного выхода ошибок. И наконец, *sprintf* и *bprintf* не выводят результат на экран, а записывают его в строку.

Вот несколько примеров использования различных форматов:

```
# Printf.printf "(x=%d, y=%d)" 34 78 ;;
(x=34, y=78)- : unit = ()
# Printf.printf "name = %s, age = %d" "Patricia" 18 ;;
name = Patricia, age = 18- : unit = ()
# let s = Printf.sprintf "%10.5f\n%10.5f\n" (-.12.24) (2.30000008) ;;
val s : string = " -12.24000\n 2.30000\n"
# print_string s ;;
-12.24000
 2.30000
- : unit = ()
```

В следующем примере, создадим функцию для распечатки матрицы вещественных чисел в заданном формате:

```
# let print_mat m =
  Printf.printf "\n" ;
  for i=0 to (Array.length m)-1 do
    for j=0 to (Array.length m.(0))-1 do
      Printf.printf "%10.3f" m.(i).(j)
    done ;
    Printf.printf "\n"
  done ;;
val print_mat : float array array -> unit = <fun>
# print_mat (Array.create 4 [| 1.2; -.44.22; 35.2 |]) ;;
```

```

1.200    -44.220    35.200
1.200    -44.220    35.200
1.200    -44.220    35.200
1.200    -44.220    35.200
- : unit = ()

```

Замечания по типу формата Описание формата задается в виде символьной строки, но оно не является типом **string**. При распознавании формата создается значение с типом **format**, в котором параметр 'а реализуется либо в **unit**, если формат не указывает параметр, либо в функциональный тип, который соответствует функции с нужным количеством аргументов и возвращающей значение **unit**.

Проиллюстрируем это на примере, частично применив функцию к формату.

```

# let p3 =
  Printf.printf "begin\n%d is val1\n%s is val2\n%f is val3\n" ;;
begin
val p3 : int -> string -> float -> unit = <fun>

```

Таким образом мы получили функцию ожидающую три аргумента. Заметьте, что слово “begin” уже было выведено. Другой формат приводит к созданию функции другого типа.

```

# let p2 =
  Printf.printf "begin\n%f is val1\n%s is val2\n" ;;
begin
val p2 : float -> string -> unit = <fun>

```

Передавая аргументы, одни за одним функции *p3* получим постепенный результат:

```

# let p31 = p3 45 ;;
45 est val1
val p31 : string -> float -> unit = <fun>
# let p32 = p31 "hello" ;;
hello est val2
val p32 : float -> unit = <fun>
# let p33 = p32 3.14 ;;
3.140000 est val3
val p33 : unit = ()
# p33 ;;
- : unit = ()

```

Последнее получено значение ничего не выводит, оно возвращает значение `()` типа `unit`.

Мы не сможем создать значение `format` используя значения типа `string`:

```
# let f d =
```

```
  Printf.printf (d^d);;
```

Characters 27–30:

This expression has **type** `string` but is here used **with type**

```
('a, out_channel, unit) format
```

Компилятор не знает значение строки, переданной в аргументе, в следствии чего он не знает тип который реализует параметр `'a` типа `format`.

С другой стороны, строки символов являются физически изменяемым типом, то есть мы можем поменять `%d` на другую букву, таким образом динамически изменить формат вывода. Это не совместимо со статической генерацией функции перевода.

Модуль Digest

Из символьной строки, какого угодно размера, хэш функции возвращает строку фиксированного размера, чаще всего меньшего размера. Такие функции возвращают отпечаток (digest).

Такие функции используются для создания хэш таблиц, как на пример модуль `Hashtbl`, позволяющий проверить присутствует ли определенный элемент в таблице прямым доступом (direct access) используя отпечаток. Например, функция `f_mod_n` подсчитывает остаток от деления суммы кодов сомволгов ASCII строки на `n` и является хэш функция. Если мы создадим таблицу размером `n`, то благодаря отпечаткам получим прямой доступ к элементам. Однако, две строки могут иметь одинаковый отпечаток. В случае подобных коллизий, в хэш таблицу добавляется дополнительное “хранилище” для хранения таких элементов. Когда таких коллизий становится много, доступ к таблице становится малоэффективным. Если `n` — размер отпечатка `a`, тогда вероятность возникновения коллизий между двумя разными строка равна $1/2^n$.

Вероятность возникновения коллизий у non-reversible хэш функции очень мала, таким образом трудно подобрать строку к конкретному отпечатку. Конечно, функция `f_mod_n` таковой не является. Таким образом non-reversible хэш функции позволяют идентифицировать (authentication) символьную строку полученную по Интернету, файл, и т.д.

Модуль `Digest` использует алгоритм MD5, как аббревиатура Message

Digest 5. Размер возвращаемого отпечатка 128 бит. Несмотря на то, что алгоритм свободно доступен, на сегодняшний день невозможно создать строку имея отпечаток. В этом модуле определен тип `Digest.t` как аббревиатура `string`. В таблице 7.3 приведены основные функции этого модуля.

<code>string</code>	:	<code>string -> t</code>
		возвращает отпечаток строки
<code>file</code>	:	<code>string -> t</code>
		возвращает отпечаток файла

Таблица 7.3: Функции модуля Digest

В следующем примере мы используем функцию `string` на небольшой строке и на большой строке, построенной из первой. Заметьте, что размер отпечатка остается такой же:

```
# let s = "The small cat is dead..." ;;
val s : string = "The small cat is dead..."
# Digest.string s ;;
- : Digest.t = "xr6\127\171(\134=\238'\252F\028\t\210$"
```

```
# let r = ref s in
  for i=1 to 100 do r:= s ^ !r done;
  Digest.string !r ;;
- : Digest.t = "\232\197[C]\137\180{>\224QX\155\131D\225"
```

Применяя отпечатки программ, мы тем самым гарантируем, что версия используемой программы правильная. К примеру, во время динамической загрузки кода (см. стр. 269), используются отпечатки для загрузки файла с байт-кодом.

```
# Digest.file "basic.ml" ;;
- : Digest.t = "\179\026\191\137\157Ly|^w7\183\164:\167q"
```

7.3.4 Persistence

Сохраняемость это способность сохранять значения переменных когда программа не выполняется. Примером может служить сохранение переменных в файле. Эта переменная доступна любой программе, которая может читать этот файл. Чтение и запись сохраняемого значения требует определенный формат данных. Необходимо определить способ хранения сложной структуры в памяти компьютера, например дерева,

в линейную структуру — последовательность байтов в файле. По этой причине кодировка сохраняемых значений называется линеаризацией¹.

Реализация и трудности линеаризации

При реализации механизма линеаризации данных необходимо сделать определенный выбор и встречаются следующие трудности:

чтение запись структур данных Память компьютера может быть рассмотрена как вектор байтов, тогда значение можно представить как часть памяти которое оно занимает. В этом случае мы можем сжать значение, сохранив лишь полезную часть.

разделение или копия При линеаризации данных нужно ли сохранять разделение (sharing). Типичный случай - бинарное дерево, у которого имеется два одинаковых (в физическом смысле) узла, здесь мы можем указать для второго узла, что он уже сохранен. Это свойство влияет на размер сохраняемых данных и на затраченное время. On the other hand, in the presence of physically modifiable values, this could change the behavior of this value after a recovery depending on whether or not sharing was conserved.

циклические структуры В подобном случае, линеаризация без разделения рискует заикнуться. Таким образом необходимо сохранить это разделение.

функциональные значения Функциональные значения, или замыкания, состоят из двух частей: окружения и кода. Часть кода состоит из адреса по которому находится код. Что в данном случае происходит с кодом? Мы конечно можем сохранить этот адрес, но в этом случае только та же программа сможет правильно распознать этот адрес. Мы так же можем сохранить последовательность инструкций этой функции, однако нам будет необходим механизм динамической загрузки кода.

гарантия типа при загрузке В этом заключается основная трудность данного механизма. Статическая типизация гарантирует что значения не создадут ошибок типа во время выполнения, но это касается лишь значений принадлежащих исполняющейся программе. Какой тип нужно дать внешнему, по отношению к программе, значению, которое не было проверено анализатором типов. Лишь только для

¹В JAVA используется термин сериализация (serialization)

того чтобы, прочитанное значение имело мономорфный тип сгенерированный компилятором, необходима передача этого значения в момент сохранения и проверка в момент чтения.

Модуль Marshal

Механизм линейаризации модуля Marshal предоставляет нам выбор: сохранить или нет разделение для обрабатываемых значений. Он так же умеет обращаться с замыканиями, но здесь сохраняется лишь указатель на адрес по которому находится код.

Этот модуль состоит в основном из функций линейаризации в какой-нибудь канал либо строку (string) и функции получающие данные из канала или строки. Функции линейаризации параметризуются. У следующего типа может быть два возможных случая:

```
type external_flag =  
  No_sharing  
| Closures;;
```

Константный конструктор `No_sharing` указывает что разделение значений не должно быть сохранено, так как по умолчанию разделение сохраняется. Конструктор `Closures` необходим когда мы имеем дело с замыканиями, в этом случае сохраняется указатель на код. Если этот конструктор отсутствует и мы попытаемся сохранить функциональное значение, то будет возбуждено исключение.

Warning

Конструктор `Closures` не может использоваться в интерактивном режиме, а лишь в командной строке.

Функции записи и чтения данного модуля приведенные в таблице 7.4.

<code>to_channel</code>	:	<code>out_channel -> 'a -> extern_flag list -> unit</code>
<code>to_string</code>	:	<code>'a -> extern_flag list -> string</code>
<code>to_buffer</code>	:	<code>string -> int -> int -> 'a -> extern_flag list -> unit</code>
<code>from_channel</code>	:	<code>in_channel -> 'a</code>
<code>from_string</code>	:	<code>string -> int -> 'a</code>

Таблица 7.4: Функции модуля Marshal

У функции `to_channel` три аргумента: выходной канал, значение и список опций, она записывает переданное значение в указанный канал.

Функция `to_string` создает строку, которая соответствует линейаризации указанного значения, тогда как `to_buffer` делает то же самое, изменяя часть строки переданной в аргументе. Функция `from_channel` возвращает прочитанное из канала линейаризованное значение. У функции `from_string` два аргумента: строка из которой необходимо прочитать значение и позиция в строке с которого начинается это значение. В файле или строке можно сохранить Несколько линейаризованных значений. В первом случае такие значения читаются последовательно, во втором достаточно указать правильный отступ от начала строки, чтобы прочитать необходимое значение.

```
# let s = Marshal.to_string [1;2;3;4] [] in String.sub s 0 10;;
- : string = "\132\149\166\190\000\000\000\t\000\000"
```

Warning

Использование данного модуля приводит к потере гарантии статической типизации (см. стр. 260)

При перечитывании сохраняемого объекта получаем значение неопределенного типа:

```
# let x = Marshal.from_string (Marshal.to_string [1; 2; 3; 4] []) 0;;
val x : 'a = <poly>
```

Эта неопределенность отмечена в Objective CAML переменной нестро-го типа `'a`. Поэтому желательно указать ожидаемый тип:

```
# let l =
  let s = (Marshal.to_string [1; 2; 3; 4] []) in
  (Marshal.from_string s 0 : int list) ;;
val l : int list = [1; 2; 3; 4]
```

На странице 260 мы вернемся к этому моменту.

Замечание

Функция `output_value` подгружаемой библиотеки соответствует вызову `to_channel` с пустым списком опций. Функция `input_value` модуля `Pervasives` вызывает функцию `from_channel`. Они сохранены для совместимости со старыми программами.

Пример: сохранение экрана

Наша задача — сохранить bitmap всего экрана в виде матрицы цветов. Функция `save_screen` забирает bitmap, конвертирует его в вектор цветов и сохраняет в файле, имя которого передано функции.

```
# let save_screen name =
  let i = Graphics.get_image 0 0 (Graphics.size_x ())
                                (Graphics.size_y ()) in
    let j = Graphics.dump_image i in
    let oc = open_out name in
      output_value oc j;
      close_out oc;;
val save_screen : string -> unit = <fun>
```

Функция `load_screen` выполняет обратную операцию; открывает файл, имя которого указано, считывает сохраненные данные, конвертирует полученную матрицу цветов в bitmap и затем рисует на экране.

```
# let load_screen name =
  let ic = open_in name in
  let image = ((input_value ic) : Graphics.color array array) in
    close_in ic;
    Graphics.close_graph();
    Graphics.open_graph (" "^(string_of_int(Array.length image.(0)))
                          ^"x"^(string_of_int(Array.length image)));
  let image2 = Graphics.make_image image in
    Graphics.draw_image image2 0 0; image2 ;;
val load_screen : string -> Graphics.image = <fun>
```

Warning

Значения абстрактных типов не могут быть сохраняемыми.

По этой причине в предыдущем примере не используется абстрактный тип `Graphics.image`, а конкретный тип `color array array`. Абстракция типов обсуждается в главе 13.

Разделение

Потеря данного свойства для значения может порой привести к тому что оно станет неиспользуемым. Вернемся к примеру генератора символов на стр. 113. Пусть нам необходимо сохранить функциональные зна-

чения `new_s` и `reset_s`, для того чтобы снова получить значения этих счетчиков. Напишем следующий код:

```
# let reset_s,new_s =
  let c = ref 0 in
    ( function () -> c := 0 ) ,
    ( function s -> c:=!c+1; s^(string_of_int !c) ) ;;

# let save =
  Marshal.to_string (new_s,reset_s) [Marshal.Closures;Marshal.
    No_sharing] ;;

# let (new_s1,reset_s1) =
  (Marshal.from_string save 0 : ((string -> string) * (unit -> unit)))
  ;;

# (* 1 *)
Printf.printf "new_s : %s\n" (new_s "X");
Printf.printf "new_s : %s\n" (new_s "X");
(* 2 *)
Printf.printf "new_s1 : %s\n" (new_s1 "X");
(* 3 *)
reset_s1();
Printf.printf "new_s1 (after reset_s1) : %s\n" (new_s1 "X") ;;
Characters 148-154:
Unbound value new_s1
```

Первые два вывода (* 1 *) выводят правильный результат. Вывод (* 2 *), после прочтения замыкания тоже кажется корректным (после X2 следует X3). Но на самом деле разделение счетчика с между функциями `new_s1` `reset_s1` утеряно. Об этом мы можем судить по выводу X4, несмотря на то что мы обнулили перед этим счетчик. Каждое из замыканий имеет свой собственный счетчик и вызов `reset_s1` не инициализирует счетчик `new_s1`. То есть мы не должны были указывать опцию `No_sharing` во время линеаризации.

Данный пример не работает, код соответствующий приведенному выше тексту следующий (прим. пер.):

```
let reset_s,new_s =
  let c = ref 0 in
    ( function () -> c := 0 ) ,
    ( function s -> c:=!c+1; s^(string_of_int !c) ) ;;
```

```

(* 1 *)
Printf.printf "new_s : %s\n" (new_s "X");;

(* 2 *)
Printf.printf "new_s : %s\n" (new_s "X");;

let save =
  Marshal.to_string (new_s,reset_s) [Marshal.Closures;Marshal.
    No_sharing] ;;

let (new_s1,reset_s1) =
  (Marshal.from_string save 0 : ((string -> string) * (unit -> unit)))
  ;;

(* 3 *)
Printf.printf "new_s1 : %s\n" (new_s1 "X");;
reset_s1();;
Printf.printf "new_s1 (after reset_s1) : %s\n" (new_s1 "X") ;;

```

Чаще всего необходимо сохранять разделение. Однако, там где важна скорость выполнения отсутствие разделения ускоряет сохранение значений. В следующем примере функция копирует матрицу. В данном случае желательно избавиться от разделения.

```

# let copy_mat_f (m : float array array) =
  let s = Marshal.to_string m [Marshal.No_sharing] in
  (Marshal.from_string s 0 : float array array) ;;
val copy_mat_f : float array array -> float array array = <fun>

```

Мы можем воспользоваться этой функцией для создания матрицы без разделения:

```

# let create_mat_f n m v =
  let m = Array.create n (Array.create m v) in
  copie_mat_f m;;
val create_mat_f : int -> int -> float -> float array array = <fun>
# let a = create_mat_f 3 4 3.14;;
val a : float array array =
  [|[|3.14; 3.14; 3.14; 3.14|]; [|3.14; 3.14; 3.14; 3.14|];
    [|3.14; 3.14; 3.14; 3.14|]|]
# a.(1).(2) <- 6.28;;
- : unit = ()
# a;;
- : float array array =

```

```
[[[3.14; 3.14; 3.14; 3.14]]; [[3.14; 3.14; 6.28; 3.14]];
[[3.14; 3.14; 3.14; 3.14]]]
```

Поведение этой функции больше сходит на `Array.create_matrix` чем `Array.create`.

Размер значений

Порой необходимо бывает знать размер сохраняемого значения. Если разделение сохранено, то указанный размер достаточно точно отображает информацию. Хотя кодировка порой оптимизирует размер непосредственных² (или элементарных) значений, информация о размере их кодировки позволяет нам сравнить различные реализации структур данных. С другой стороны, для программ, которые выполняются без остановки, как во встроенных системах или даже серверах, просмотр размер структур данных помогает отловить утечки памяти. Две функции для вычисления размера и константа модуля `Marshal` описаны в таблице 7.5.

<code>header_size</code>	:	<code>int</code>
<code>data_size</code>	:	<code>string -> int -> int</code>
<code>total_size</code>	:	<code>string -> int -> int</code>

Таблица 7.5: Функции модуля `Marshal`

Размер сохраняемого значения есть сумма размера его данных и размер заголовка.

В следующем примере мы приводим сравнение двух бинарных деревьев при помощи функции `MD5`:

```
# let size x = Marshal.data_size (Marshal.to_string x []) 0;;
val size : 'a -> int = <fun>
# type 'a bintree1 = Empty1 | Node1 of 'a * 'a bintree1 * 'a bintree1 ;;
type 'a bintree1 = | Empty1 | Node1 of 'a * 'a bintree1 * 'a bintree1
# let s1 =
  Node1(2, Node1(1, Node1(0, Empty1, Empty1), Empty1),
        Node1(3, Empty1, Empty1)) ;;
val s1 : int bintree1 =
  Node1
    (2, Node1 (1, Node1 (0, Empty1, Empty1), Empty1),
     Node1 (3, Empty1, Empty1))
# type 'a bintree2 =
```

²Например вектор символов

```

    Empty2 | Leaf2 of 'a | Node2 of 'a * 'a bintree2 * 'a bintree2 ;;
type 'a bintree2 =
    | Empty2
    | Leaf2 of 'a
    | Node2 of 'a * 'a bintree2 * 'a bintree2
# let s2 =
    Node2(2, Node2(1, Leaf2 0, Empty2), Leaf2 3) ;;
val s2 : int bintree2 = Node2 (2, Node2 (1, Leaf2 0, Empty2), Leaf2 3)
# let s1, s2 = size s1, size s2 ;;
val s1 : int = 13
val s2 : int = 9

```

Полученные при помощи функции `size` значения совпадают с предполагаемым размером деревьев `s1` и `s2`.

Проблемы типизации

Действительной проблемой сохраняемых значений является возможность нарушить механизм типизации Objective CAML. Функции записи создают правильный мономорфный тип: `unit` и `string`. Тогда как функции чтения возвращают полиморфный тип `'a`. С таким сохраняемым значением мы можем сделать что угодно. Вот самый худший вариант (см. гл. 1 на стр. 67): создание функции `magic_copy` с типом `'a -> 'b`.

```

# let magic_copy a =
    let s = Marshal.to_string a [Marshal.Closures] in
    Marshal.from_string s 0;;
val magic_copy : 'a -> 'b = <fun>

```

Использование подобной функции приведет к аварийной остановке программы:

```

# (magic_copy 3 : float) +. 3.1;;
Segmentation fault

```

В интерактивной среде (в Линуксе) текущая сессия заканчивается с сообщением об ошибке памяти.

7.3.5 Системный интерфейс

В стандартной библиотеке содержится 6 модулей системного интерфейса:

- модуль `Sys`: для взаимодействия программы с системой

- модуль **Arg**: для анализа переданных программе аргументов
- модуль **Filename**: для перемещения по каталогам (независимо от операционной системы)
- модуль **Printexc**: для перехвата и вывода исключений
- модуль **Gc**: для контроля автоматического сборщика памяти, который будет описан в главе 8
- модуль **Callback**: для вызова функций Objective CAML в программе на C, об этом читайте в главе 11.

Описание четырех первых модулей приведено ниже.

Модуль Sys

Данный модуль содержит полезные функции для взаимодействия с операционной системой, а так же для обработки сигналов, полученных программой. В таблице 7.6 приведены значения содержащие информацию о системе.

OS_type	: string : тип системы
interactive	: bool ref : истина, если интерактивный режим
word_size	: string : размер слова (32 или 64 бита)
max_string_length	: int : максимальный размер символьной строки
max_array_length	: int : максимальный размер вектора
time	: unit -> float : время в секундах, истекшее с начала выполнения программы

Таблица 7.6: Данные об операционной системе

Связь между программой и операционной системой осуществляется через командную строку, переменную окружения и запуском другой программы. Эти функции описаны в таблице 7.7.

При помощи Функции в таблице 7.8 можно “использовать” файловую систему.

<code>argv</code>	: <code>string array</code> : содержит вектор параметров
<code>getenv</code>	: <code>string -> string</code> : получить значение переменной окружения
<code>command</code>	: <code>string -> int</code> : выполнения команды с указанным именем

Таблица 7.7: Связь с операционной системой

<code>file_exists</code>	: <code>string -> bool</code> : возвращает истина, если файл существует
<code>remove</code>	: <code>string -> unit</code> : удалить файл
<code>rename</code>	: <code>string -> string -> unit</code> : переименовать файл
<code>chdir</code>	: <code>string -> unit</code> : сменить текущий каталог
<code>getcwd</code>	: <code>unit -> string</code> : вернуть имя текущего каталога

Таблица 7.8: Операции с файловой системой

Обработка сигналов будет описано в специальной главе о системном программировании (17).

Напишем следующую программу, в ней мы вновь вернемся к примеру с сохранением графического окна в виде вектора цветов. Функция `main` проверяет не запущена ли она в интерактивном режиме, затем считывает имена файлов в командной строке, проверяет существуют ли они и затем выводит их на экран (при помощи функции `load_screen`). Между двумя выводами, мы вставляем ожидание нажатия на клавишу.

```
# let main () =
  if not (!Sys.interactive) then
    for i = 0 to Array.length(Sys.argv) - 1 do
      let name = Sys.argv.(i) in
        if Sys.file_exists name then
          begin
            ignore(load_screen name);
            ignore(Graphics.read_key)
          end
        done;;
  val main : unit -> unit = <fun>
```

Модуль Arg

Модуль **Arg** определяет синтаксис для анализа аргументов командной строки. В нем имеется функция для синтаксического анализа, а так же определение действия, которое будет связано с анализируемым элементом.

Элементы командой строки разделены между собой одним или несколькими пробелами. Все они хранятся в векторе **Sys.argv**. В синтаксисе определенном **Sys.argv** некоторые элементы начинаются знаком минус ('-'). Такие элементы называются ключевыми словами командной строки. Ключевым словам можно привязать определенное действие, которое ожидает аргумент с типом **string**, **int** или **float**. Значения аргументов инициализируются значениями командной строки, которые непосредственно следуют за ключевым словом. В таком случае вызывается функция перевода значения из строки в ожидаемый тип данных. Другие аргументы командной строки называются анонимными. Им ассоциируется общее действие, которое принимает их значение в виде аргумента. Если указана неопределенная опция, то на экран выводится небольшая помощь, которая определяется пользователем.

Действия, связанные с ключевыми словами определены следующим типом:

```
type spec =
  | Unit of (unit -> unit)    (* Call the function with unit argument*)
  | Set of bool ref          (* Set the reference to true*)
  | Clear of bool ref        (* Set the reference to false*)
  | String of (string -> unit) (* Call the function with a string
                                argument *)
  | Int of (int -> unit)      (* Call the function with an int
                                argument *)
  | Float of (float -> unit)  (* Call the function with a float
                                argument *)
  | Rest of (string -> unit)  (* Stop interpreting keywords and call the
                                function with each remaining argument*)
```

Для анализа командной строки существует следующая функция:

```
# Arg.parse ;;
- : (string * Arg.spec * string) list -> (string -> unit) -> string ->
    unit =
<fun>
```

Первым аргументом является список кортежей формы (**key**, **spec**, **doc**), где:

- **key** есть строка соответствующая ключевому слову. Эта строка начинается со специального символа `'_'`
- **spec** есть значение типа **spec**, которое ассоциирует действие ключу **key**
- **doc** есть символьная строка, содержащая описание опции **key**. Она выводится в случае синтаксической ошибки.

вторым аргументом передается функция обработки анонимных аргументов командной строки. Последним аргументом является строка, которая выводится в заголовке помощи.

В модуле **Arg** имеется так же:

- **Bad**: исключение с входным аргументом типа строка. Оно может быть использовано обрабатывающими функциями.
- **usage** : типа `(string * Arg.spec * string) list -> string ->`. Эта функция выводит подсказку об использовании программы. Желательно ей передавать те же самые аргументы что и для **parse**
- **current**: типа `int ref`. Содержит ссылку на текущее значение индекса вектора **Sys.argv**. Это значение может быть изменено при необходимости.

Примера ради, напомним функцию **read_args**, которая инициализирует игру Minesweeper, представленную в главе 5, страница 192. Возможные опции будут следующими: **-col**, **-lin** и **-min**. За ними буду следовать целые числа, указывающие число столбцов, число линий и число мин соответственно. Эти значения не должны быть меньше значений по умолчанию: 10, 10 и 15.

Для этого создадим три функции обработки:

```
# let set_nbcolls cf n = cf := {!cf with nbcolls = n} ;;
# let set_nbrows cf n = cf := {!cf with nbrows = n} ;;
# let set_nbmines cf n = cf := {!cf with nbmines = n} ;;
```

У всех этих функций одинаковый тип: `config ref -> int -> unit`. Функции анализа командной строки можно написать следующим образом:

```
# let read_args() =
  let cf = ref default_config in
  let speclist =
    [("-col", Arg.Int (set_nbcolls cf), "number of columns (>=10))");
```



```

    ("—lin", Arg.Int (set_nbrows cf), "number of lines (>=10)");
    ("—min", Arg.Int (set_nbmines cf), "number of mines (>=15)")]
  in
  let usage_msg = "usage : minesweep [—col n] [—lin n] [—min n]" in
    Arg.parse speclist (fun s -> ()) usage_msg; !cf ;;
  val read_args : unit -> config = <fun>

```

При помощи переданных параметров эта функция вычисляет конфигурацию, которая затем будет передана функции открывающей графическое окно `open_wcf` при запуске игры. Каждая из опций, не является обязательной, как на то указывает само слово опция. Если одна из них отсутствует в командной строки, то будет использоваться значение по умолчанию. Порядок опций не имеет значений.

Модуль Filename

В модуле `Filename` реализованы операции по доступу к файловой системе, независимо от операционной системы. Правила по которым называются имена в Windows, Unix и MacOS сильно различаются.

Модуль Printexc

Этом совсем небольшой модуле из трех функций (таблица 7.9) предоставляет нам сборщик исключений. Он очень удобен, особенно для программ запускаемых с командной строки³, чтобы быть уверенным в том что ни одно исключение не будет упущено и это не приведет к остановке программы.

<code>catch</code>	: (<code>'a -> 'b</code>) -> <code>'a -> 'b</code> : сборщик исключений
<code>print</code>	: (<code>'a -> 'b</code>) -> <code>'a -> 'b</code> : вывести и пере-возбудить исключение
<code>to_string</code>	: <code>exn -> string</code> : перевести исключение в строку

Таблица 7.9: Сборщик исключений

Функция `catch` применяет свой первый аргумент ко второму, это приведет к запуску основной функции программы. Если исключение будет возбуждено на уровне `catch`, то есть не выловленное внутри программы,

³В интерактивном режиме есть свой сборщик, который выводит сообщение о неотловленном исключении

то `catch` выведет свое имя и прекратит программу. Функция `print` ведет себя схожим образом, с разницей в том что после вывода на экран она снова возбуждает исключение. И наконец функция `to_string` переводит исключение в строку символов. Она используется двумя предыдущими функциями. Если бы мы переписывали программу просмотра `bitmap`, то мы бы скрыли функцию `main` внутри функции `go`:

```
# let go () =
  Printexc.catch main ();;
val go : unit -> unit = <fun>
```

Это позволит нормально закончить программу и вывести при этом значение неотловленного исключения.

7.4 Другие библиотеки дистрибутива

Другие библиотеки, входящие в дистрибутив Objective CAML, затрагивают следующие расширения:

графика состоящий из независящего от платформы модуля `Graphics`, который был описан в главе 4.

точная арифметика несколько модулей для выполнения точных расчетов с целыми и рациональными числами. Числа представлены в виде целых Objective CAML когда это возможно.

фильтрация регулярных выражений облегчает анализ строк и текста. Модуль `Str` будет описан в главе 10.

системные вызовы Unix при помощи модуля `Unix` этой библиотеки можно вызывать из Objective CAML системные вызовы Unix. Большая часть этой библиотеки совместима с системой Windows. Мы воспользуемся этой библиотекой в главах 17 и 19

“легковесный” процесс несколько модулей, которые будут подробно описаны в главе 18.

доступ к базам данных NDBD работает только в Unix и не будет описан.

динамическая загрузка кода состоит из одного модуля `Dynlink`.

При помощи приведенных примеров мы опишем библиотеки для работы с большими числами и динамической загрузки.

7.4.1 Точная арифметика

Данная библиотека обеспечивает точную арифметику при работе с большими числами; целыми или рациональными. У значений типа `int` и `float` существует предел. Вычисления целых чисел реализуются по модулю самого большого положительного числа, что может привести к незамеченному переполнению. Результат вычисления чисел с плавающей запятой округляется, при распространении этот феномен также может привести к ошибкам. Данная библиотека смягчает выше указанные недостатки.

Эта библиотека написана на языке C, поэтому нам необходимо создать новую интерактивную среду, в которую включен следующий код:

```
ocamlmktop -custom -o top nums.cma -cclib -lnums
```

Библиотека состоит из нескольких модулей, два наиболее важных из них это `Num` для всех операций и `Arith_status` для контроля опций расчета. Общий тип `num` является тип сумма группирующая три следующих типа:

```
type num = Int of int
          | Big_int of big_int
          | Ratio of ratio
```

Типы `big_int` и `ratio` являются абстрактными.

После операций со значениями типа `num` ставится символ `'/'`. Например сложение двух значений `num` запишется как `+/`, тип этой операции `num -> num -> num`. Для операций сравнения применяется аналогичное правило. Вследующем примере, мы вычисляем факториал:

```
# let rec fact_num n =
  if Num.(<=/) n (Num.Int 0) then (Num.Int 1)
  else Num.( */ ) n (fact_num ( Num.( - / ) n (Num.Int 1) ));
val fact_num : Num.num -> Num.num = <fun>
# let r = fact_num (Num.Int 100);;
val r : Num.num = Num.Big_int <abstr>
# let n = Num.string_of_num r in (String.sub n 0 50) ^ "...";;
- : string = "93326215443944152681699238856266700490715968264381..."
```

Предварительная загрузка модуля `Num` облегчает читаемость кода:

```
# open Num ;;
# let rec fact_num n =
  if n <= / (Int 0) then (Int 1)
  else n */ (fact_num ( n - / (Int 1) )) ;;
val fact_num : Num.num -> Num.num = <fun>
```

Вычисление рациональных чисел так же является точным. Для того чтобы вычислить число e по следующей формуле:

$$e = \lim_{m \rightarrow \infty} \left(1 + \frac{1}{m}\right)^m$$

напишем такую функцию:

```
# let calc_e m =
  let a = Num.(+ /) (Num.Int 1) ( Num.( / ) (Num.Int 1) m) in
    Num.( ** / ) a m;;
val calc_e : Num.num -> Num.num = <fun>
# let r = calc_e (Num.Int 100);;
val r : Num.num = Ratio <abstr>
# let n = Num.string_of_num r in (String.sub n 0 50) ^ "...";;
- : string = "27048138294215260932671947108075308336779383827810..."
```

При помощи модуля **Arith_status** мы можем контролировать вычисление: округление результата для вывода, нормализация рациональных чисел и обработка нулевого знаменателя дроби. При помощи функции **arith_status** мы можем знать состояние указанных опций.

```
# Arith_status.arith_status();;
```

```
Normalization during computation --> OFF
  (returned by get_normalize_ratio ())
  (modifiable with set_normalize_ratio <your choice>)
```

```
Normalization when printing --> ON
  (returned by get_normalize_ratio_when_printing ())
  (modifiable with set_normalize_ratio_when_printing <your choice
    >)
```

```
Floating point approximation when printing rational numbers --> OFF
  (returned by get_approx_printing ())
  (modifiable with set_approx_printing <your choice>)
```

```
Error when a rational denominator is null --> ON
  (returned by get_error_when_null_denominator ())
  (modifiable with set_error_when_null_denominator <your choice
    >)
- : unit = ()
```

В зависимости от необходимости, эти опции могут быть изменены. К примеру если мы желаем выводить на экран приближенное значение предыдущего расчета:

```
# Arith_status.set_approx_printing true;;  
- : unit = ()  
# Num.string_of_num (calc_e (Num.Int 100));;  
- : string = "0.270481382942e1"
```

Вычисление больших чисел занимает больше времени чем для обычных целых, а так же значения занимают больше места в памяти. Однако, данная библиотека старается делать представление данных наиболее экономичным. В любом случае, наша цель, избежать распространение ошибок из-за округления и возможность проводить вычисления над большими числами, оправдывает потерю эффективности.

7.4.2 Динамическая загрузка кода

Для динамической загрузки программ, в виде байт-кода, существует библиотека `Dynlink`. Преимущества у динамической загрузки кода следующие:

- уменьшение разера кода программы. Если определенные модули не используются, то они не будут загружены.
- возможность выбора модуля для загрузки во время выполнения программы. В зависимости от условий выполнения программы, можно выбрать тот или иной модуль для загрузки.
- возможность изменить поведение модуля во время выполнения. Опять же, при определенных условиях программа может загрузить новый модуль и скрыть код предыдущего.

Интерактивная среда Objective CAML использует подобный механизм, почему бы и разработчик на Objective CAML не имел подобных удобств.

Во время загрузки объектного файла (с расширением `.cmo`), вычисляются различные выражения. Программа, которая загружает модуль, не имеет доступа к именам модуля, поэтому обновление таблицы имен основной программы лежит на ответственности загружаемого модуля.

Warning

Динамическая загрузка кода может быть действует лишь для байт-код

объектов.

Описание модуля

Для того чтобы загрузить файл `f.со`, содержащий байт-код, необходимо с одной стороны знать путь по которому находится данный файл, и имена модулей которые он использует с другой стороны. Динамические загружаемые байт-код файлы не знают путь по которому находятся модули стандартной библиотеки и имена модулей. Поэтому необходимо предоставить им эту информацию.

<code>init</code>	: <code>unit -> unit</code> : инициализация динамической загрузки
<code>add_interfaces</code>	: <code>string list -> string list -> unit</code> : добавление имен модулей и путь по которому находится с
<code>loadfile</code>	: <code>string -> unit</code> : загрузка байт-код файла
<code>clear_avalaible_units</code>	: <code>unit -> unit</code> : удаляет имена модулей и путь по которому находится ста
<code>add_avalaibl_units</code>	: <code>(string * Digest.t) list -> unit</code> : добавить имя модуля и отпечаток для загрузки, без инте
<code>allow_unsafe_modules</code>	: <code>bool -> unit</code> : загружать файлы содержащие внешние объявления
<code>loadfile_private</code>	: <code>string -> unit</code> : загруженный модуль не будет доступен для следующих з

Таблица 7.10: Функции модуля Dynlink

В момент динамической загрузки, может возникнуть множество ошибок. Кроме того, что загружаемый файл и интерфейсный файлы должны существовать по искомым каталогам, байт-код должен быть корректным и загружаемым. Подобные ошибки сгруппированы в типе `error`, который используется как аргумент исключения `Error` и функции `error` с типом `varnameerror -> string`. При помощи данной функции мы получаем понятные сообщения об ошибках.

Пример

Для того, чтобы проиллюстрировать динамическую загрузку, создадим три модуля:

- `F` в котором определена ссылка на функцию `g`
- `Mod1` и `Mod2` в которых изменяется поведение функции на которую указывает `F.f`

Модуль `F` определен в файле `f.ml`:

```
let g () =
  print_string "I am the 'f' function by default\n" ; flush stdout ;;
let f = ref g ;;
```

Модуль `Mod1` определен в файле `mod1.ml`:

```
print_string "The 'Mod1' module modifies the value of 'F.f'\n" ; flush
  stdout ;;
let g () =
  print_string "I am the 'f' function of module 'Mod1'\n" ;
  flush stdout ;;
F.f := g ;;
```

Модуль `Mod2` определен в файле `mod1.m2`:

```
print_string "The 'Mod2' module modifies the value of 'F.f'\n" ; flush
  stdout ;;
let g () =
  print_string "I am the 'f' function of module 'Mod2'\n" ;
  flush stdout ;;
F.f := g ;;
```

И наконец определим основную программу в файле `main.ml`, в которой вызовем функцию на которую указывает `F.f`, затем загрузим модуль `Mod1`, снова вызовем функцию `F.f`, загрузим модуль `Mod2` и вызовем в последний раз `F.f`.

```
let main () =
  try
    Dynlink.init () ;
    Dynlink.add_interfaces [ "Pervasives"; "F" ; "Mod1" ; "Mod2" ]
      [ Sys.getcwd() ; "/usr/local/lib/ocaml/" ] ;
    !(F.f) () ;
    Dynlink.loadfile "mod1.cmo" ; !(F.f) () ;
    Dynlink.loadfile "mod2.cmo" ; !(F.f) ()
  with
  Dynlink.Error e -> print_endline (Dynlink.error_message e) ; exit 1
  ;;
```

```
main () ;;
```

Кроме указанных действий, инициализации динамической загрузки, программа должна объявить используемые интерфейсы вызовом `Dynlink.add_interface`. Скомпилируем созданные файлы.

```
\begin{lstlisting}
$ ocamlc -c f.ml
$ ocamlc -o main dynlink.cma f.cmo main.ml
$ ocamlc -c f.cmo mod1.ml
$ ocamlc -c f.cmo mod2.ml
```

Вызов программы `main` даст следующий результат:

```
$ main
I am the 'f' function by default
The 'Mod1' module modifies the value of 'F.f'
I am the 'f' function of module 'Mod1'
The 'Mod2' module modifies the value of 'F.f'
I am the 'f' function of module 'Mod2'
```

Во время динамической загрузки, происходит выполнения кода. Это проиллюстрировано выводом на экран строк начинающихся на “The 'Mod...”. Возможные побочные эффекты данного модуля наследуются основной программой. По этой причине различные вызовы функции `F.f` приводят к разным вызовам функций.

Библиотека `Dynlink` предоставляет базовый механизм динамической загрузки байт-кода. Программист должен самостоятельно позаботиться об управлении таблицей, для того чтобы загрузка имела место.

7.5 Упражнения

7.6 Резюме

В этой главе мы ознакомились с различными библиотеками стандартной библиотеки Objective CAML. Библиотеки представляют из себя множество простых модулей (или отдельных элементов компиляции). Здесь мы детально рассмотрели модули форматированного вывода (`Printf`), сохраняемости значений (`Marshal`), системного интерфейса (`Sys`) и отлавливания исключений (`Printexc`). Модули синтаксического анализа, управления памятью, системное и сетевое программирование, легковесные процессы будут рассмотрены в следующих главах.

Глава 8

Автоматический сборщик мусора

8.1 Введение

Модель, по которой процессор выполняет программу, соответствует императивному программированию. Программа — это набор машинных инструкций, выполнение которых приводит к изменению памяти компьютера. Память состоит в основном из значений созданных или манипулируемых программой. Как и любой другой ресурс компьютера, память имеет свой предел. Поэтому программа, пытающаяся заполучить больше памяти, чем количество которое может ей выделить система, окажется в неверном состоянии. По этой причине необходимо оптимизировать использование памяти. Для этого нужно освободить в памяти место, занимаемое значениями, в которых программа больше не нуждается. Такое управление памятью серьезно сказывается на выполнении и эффективности программы.

Действие по резервированию памяти, для ее последующего использования, называется выделением памяти. Различают статическое выделение, которое реализуется в момент загрузки программы, до ее выполнения, и динамическое выделение, происходящее во время выполнения программы. Память, выделенная статически, остается зарезервированной до конца выполнения программы, а динамически выделенная память может быть освобождена и использована затем для других целей.

Подобное управление памятью может привести к следующим трудностям:

- если память была высвобождена, в то время как в ней находилось используемое программой значение, то оно (значение) может быть

утеряно. Подобные указатели на несуществующие данные называются “повисшими ссылка” (*dangling pointer*).

- если адрес выделенной памяти больше не известен (утерян), то мы не сможем ее высвободить до конца выполнения программы. Этот феномен называется утечкой памяти (*leak*).

Чтобы избежать подобные проблемы, управление памятью требует от программиста осторожности и внимания. Данная задача может оказаться нелегкой, если в программе имеются сложные структуры данных, в особенности если они совместно используют участки памяти.

На сегодняшний день во многих языках программирования существуют механизмы автоматической сборки памяти, облегчающие задачу программиста. Основная идея заключается в том, что программе необходимы только те динамически выделенные участки памяти, на которые она указывает напрямую или косвенно. Все другие значения в памяти, адрес которых больше не доступен, могут быть высвобождены. Подобное высвобождение памяти может произойти сразу после того как значение больше не используется или позже, когда программе понадобится больше памяти, чем она располагает.

В Objective CAML существует подобный механизм, называемый GC (от английского *Garbage Collector* или сборщик мусора). Память выделяется в момент создания значения (то есть при использовании конструктора) и не явно освобождается. Большинству программ нет необходимости даже знать о существовании GC. Однако подобный механизм может серьезно повлиять на эффективность программ, в которой очень часто создаются новые значения. В подобных случаях бывает необходимо знать параметры GC или даже явно вызывать сборщик мусора. Для того чтобы правильно создать интерфейс между Objective CAML и другим языком программирования (см. гл. 11), нужно конкретно представлять себе каким образом GC влияет на представление данных.

8.2 План главы

В данной главе мы рассмотрим динамическое выделение и различные алгоритмы освобождения памяти. В особенности будет рассмотрен сборщик мусора Objective CAML, который является сочетанием этих алгоритмов. В начале сделаем обзор различных классов памяти и их особенностей. Затем опишем выделение памяти, а так же сравним ее явное и не явное освобождение. В третьей части представлены основные алгоритмы высвобождения памяти, а в четвертой мы детально рассмотрим

сборщик Objective CAML. В следующей части будет приведен пример использования GC для контроля области динамически распределяемой памяти (куча). В последней части представляются достоинства *слабых указателей* из одноименного модуля **Weak** при создании кэша памяти.

8.3 Программа и память

Программа на машинном языке это последовательность инструкций, которые изменяют память. Память обычно состоит из следующих частей:

- регистры (зона памяти доступная напрямую и быстро)
- стек
- статически выделенная зона памяти (сегмент данных или data segment)
- динамически выделенная зона памяти (куча или heap)

Только стек и куча могут динамически менять свой размер во время исполнения программы. В зависимости от языка программирования, может существовать или нет определенный контроль указанных элементов памяти. При динамической загрузке (см. стр. 269) код размещается в динамической зоне памяти.

8.4 Выделение и освобождение памяти

Большинство языков позволяют динамически выделять память. Для примера можно привести C, Pascal, Lisp, ML, SmallTalk, C++, Java, Ada.

8.4.1 Явное выделение памяти

Различают два типа выделения памяти:

- простое, при котором мы получаем зону памяти указанного размера, не заботясь о содержимом этой зоны;
- выделение, при котором в дополнение к предыдущему процессу, реализуется инициализация.

Вызовы *new* в Pascal или *malloc* в C соответствуют первому случаю. Они возвращают указатель на зону памяти (то есть адрес), при помощи которого можно прочитать или изменить значение хранимое в этой памяти. Второй случай соответствует функциям создания (конструкции)

значений в Objective CAML или объектно-ориентированных языков. В ОО языках экземпляры класса создаются при помощи оператора **new**, которому передается конструктор класса. Конструктору в свою очередь можно передать параметры. В функциональных языках при определении структурных значений (пара, список, запись, вектор, замыкание) вызывается конструктор.

Рассмотрим конструкцию значения в Objective CAML на примере. На рисунке 8.1 изображено представление значения в памяти.

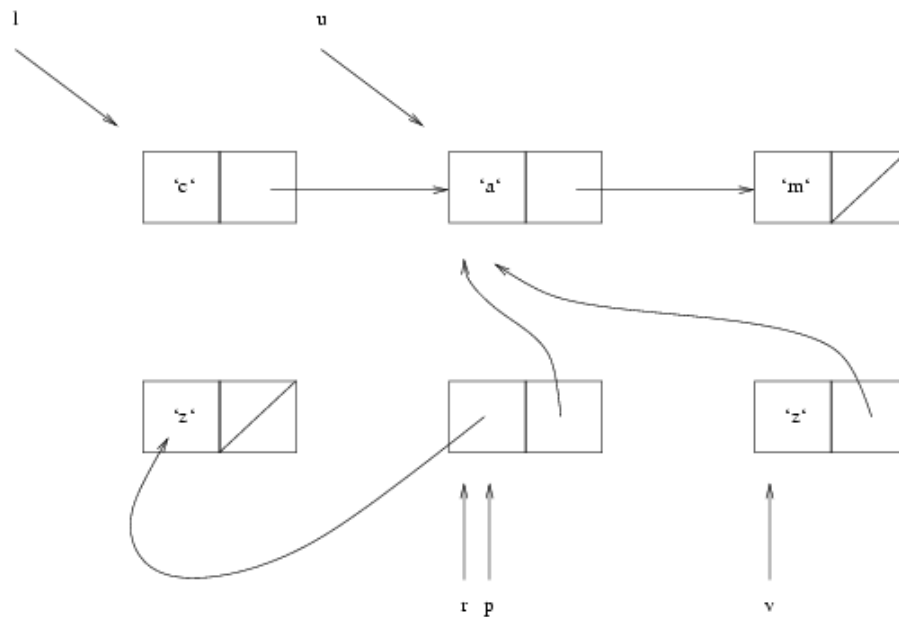


Рис. 8.1: Представление значения в памяти

```
# let u = let l = ['c'; 'a'; 'm'] in List.tl l ;;
val u : char list = ['a'; 'm']
# let v = let r = ( ['z'], u )
              in match r with p -> (fst p) @ (snd p) ;;
val v : char list = ['z'; 'a'; 'm']
```

На рисунке элемент списка представлен как кортеж из двух слов, в первом хранится символ, а во втором указатель на следующий элемент списка. На самом деле все обстоит немного иначе, детали будут рассмотрены в главе о связке с языком C (см. стр. 349).

Сначала создается значение **l**, для каждой ячейки списка `['c'; 'a'; 'm']` вызывается конструктор `(::)`. Глобальное объявление **u** соответствует хвосту списка **l**. В данном примере значения разделяются между **l** и **u**, то

есть между аргументом функции *List.tl* и ее результатом. После вычисления данного выражения лишь *u* продолжает существовать (1 прекращает существовать).

Затем создается список из одного элемента, потом пара *r* в которой содержится этот список и список *u*. Эта пара сопоставляется с образом, при этом она переименуется сопоставлением в *p*. После этого первый элемент *p* объединяется с его вторым элементом, откуда получаем список *['z'; 'a'; 'm']* связанный с глобальным идентификатором *v*. Заметим, что результат *snd* (список *['a'; 'm']*) разделяется с *p*, тогда как результат *fst* (символ *'z'*) копируется.

Во всех этих случаях, происходит явное выделение памяти, то есть запрошенное программистом (в виде инструкции или выражения языка).

Замечание

При выделении памяти, сохраняется информация о размере выделенного объекта. Это необходимо для последующего высвобождения.

8.4.2 Явное освобождение памяти

В языках программирования, с явным освобождением памяти, существует специальный оператор освобождения (*free* в C или *dispose* в Pascal), которому передается адрес (указатель) объекта, который нужно освободить. Путные синонимы. Используя сохраненную во время выделения памяти информацию, программа высвобождает участок памяти и он может быть использован в последствии.

Динамическое выделение памяти обычно используется для меняющихся структур данных, например списки, деревья и т.д. Освобождения памяти занимаемого такими структурами не делается “одним движением”, а при помощи функции, которая проходится по всем элементам структуры. Такие функции называются дескрукторами.

Не трудно корректно определить подобную функцию, однако ее использование может оказаться сложной задачей. Действительно, для того чтобы освободить память занимаемую структурой, нужно пройти по всем ее указателям и применить к ним оператор освобождения памяти. Может оказаться правильным предоставить эту задачу программисту, так как он лучше знает свои данные и как их освобождать. Однако неправильное использование указанных операторов можно привести к ошибке выполнения программы. Основные трудности заключаются в следующем:

- повисшие указатели: память уже была высвобождена, но существуют указатели которые продолжают ссылаться на эту область памяти. При попытке их использовать, мы рискуем получить некорректные данные.
- недоступные области памяти (утечки памяти или leak). Выделенный участок памяти, на который не ссылается ни один указатель. Таким образом, мы получаем утечки памяти, так как у нас нет возможности ее освободить.

Знать длительность существования всего множества значений программы есть основная трудность явного высвобождения памяти.

8.4.3 Неявное освобождение памяти

В языках программирования с неявным освобождением памяти не существует операторов освобождения памяти. Программист не может высвободить память занятую значением. Вместо этого, срабатывает сборщик памяти для неиспользуемого значения, то есть на которое больше никто не ссылается. Сборщик так может быть запущен в случае нехватки памяти.

Алгоритм автоматического высвобождения памяти является в какой-то мере глобальным деструктором. Из-за этой особенности концепция и реализация такого деструктора намного сложнее, чем деструктора специфичного для определенной структуры данных. Однако, если данная трудность преодолена, автоматическая сборка памяти значительно повышает надежность управления памятью. В частности, исчезает риск появления повисших указателей.

Автоматическая сборка памяти вносит следующие улучшения область динамически распределяемой памяти (куча):

- сжатие: вся собранная память состоит из одного блока, что позволяет избежать фрагментации и соответственно выделить память размером кучи
- локализация: различные составляющие одного значения близки друг к другу в адресном пространстве. Эта особенность позволяет остаться в одних и тех же страницах памяти (memory pages) при просмотре этого значения и избежать выхода из кэш.

При выборе архитектуры GC необходимо учитывать определенные критерии и ограничения:

- фактор сборки (*регенерации*) памяти: какой процент неиспользуемой памяти доступен?
- фрагментация памяти: можно ли выделить блок равный размеру свободной памяти?
- замедление выделения и сборки памяти
- compilation: представление значений действительно полностью свободно?

В реальности, главным критерием является надежность, в связи с чем GC находят компромисс между остальными ограничениями.

8.5 Автоматическая сборка памяти

Различают два класса алгоритмов автоматической сборки памяти:

- счетчики ссылок: каждый выделенный блок памяти знает количество ссылок, которые на него указывают. Блок высвобождается, когда счетчик становится равен нулю.
- алгоритмы-разведчики: начиная с определенного множества корней (указателей), просматривается множество доступных значений на подобие просмотра ориентированного графа.

Алгоритмы-разведчики более часто используются в языках программирования. Счетчик ссылок загружает систему (обновление ссылок) даже когда нечего высвобождать.

8.5.1 Счетчики ссылок

Каждой выделенной зоне памяти ассоциируется счетчик, который хранит число указателей на данный объект. Он увеличивается при каждом новом указателе и уменьшается когда указатель исчезает. Зона памяти высвобождается, когда счетчик равен 0.

Выгода такой системы заключается, в том что зона памяти высвобождается сразу же после того как она перестала использоваться. Кроме того, что такой сборщик загружает систему, он не умеет обращаться с круговыми (циклическими) объектами. Предположим что Objective CAML обладает таким сборщиком. В следующем примере создается временное значение 1 — список символов, где последнее звено указывает на ячейку с 'с'. Данный список является циклическим (изб. 8.2).

```
# let rec l = 'c'::'a'::'m'::l in List.hd l ;;
- : char = 'c'
```

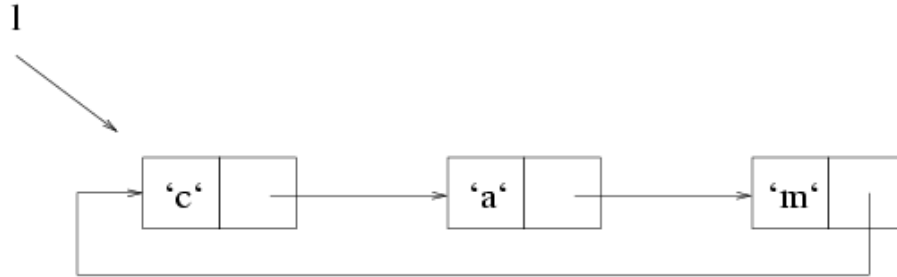


Рис. 8.2: Представление в памяти циклического списка

К концу вычисления данного выражения, у каждого элемента списка `l` счетчик будет равен 1 (даже у первого, так как список циклический). Однако данное значение больше не доступно и не сможет быть высвобождено, так как его счетчик не равен нулю. В языках программирования использующих сборщик с алгоритмом счетчика ссылок, как например Python, и разрешающих циклические значения необходимо предусмотреть дополнительный алгоритм-разведчик памяти.

8.5.2 Алгоритм-разведчик

Алгоритм-разведчик обследует граф доступных значений кучи. Для этого он использует множество корней, которые указывают на начало поиска. Данные корни находятся вне кучи, чаще всего они хранятся в стеке. Предположим, что в приведенном выше примере изб. 8.1, значения `u` и `v` являются частью корней. Если начать обход, начиная с этих значений, то получим граф значений, которые необходимо сохранить: звенья и указатели обведенные жирной линией на изб. 8.3.

При обходе графа необходимо уметь различать непосредственные значения от указателей. Если корень указывает на целое число, то не стоит его рассматривать как адрес другой ячейки в памяти. В функциональных языках для этого используются несколько битов каждой ячейки кучи — биты маркировки (*tag bits*). По этой причине в целых числах Objective CAML используется лишь 31 бит. Эта особенность обсуждается в главе 11, стр. 349. Существуют другие решения чтобы различить указатель от непосредственного значения, они будут описаны на стр. 288.

Приведем два следующих наиболее используемых алгоритма: *Mark&Sweep*

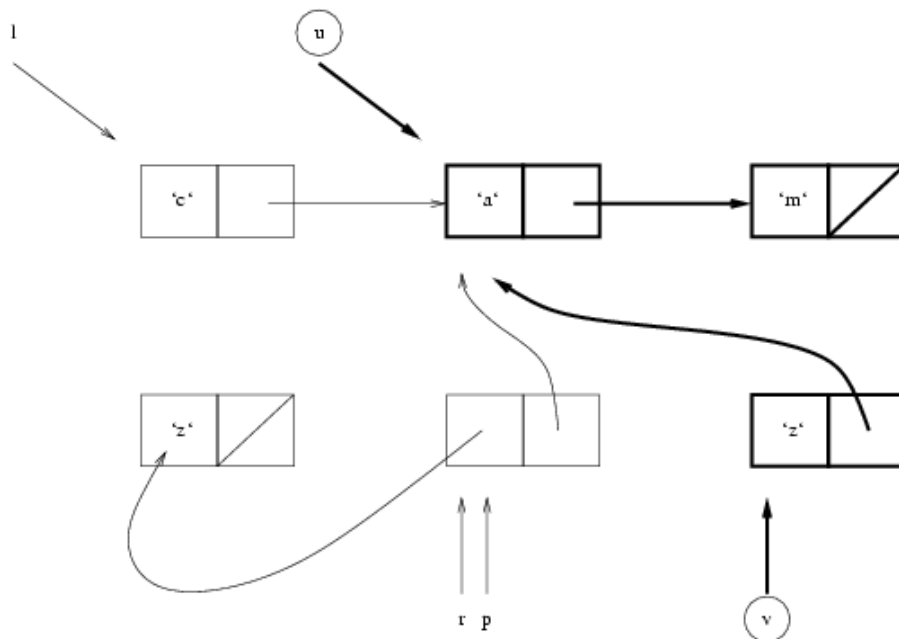


Рис. 8.3: Высвобождение памяти сборщиком

и Stop&Copy. Первый создает список свободных ячеек кучи, а второй копирует еще “живые” ячейки во вторичную зону памяти.

Необходимо представить себе кучу как вектор ячеек памяти. Состояние кучи примера 8.1 изображено на 8.4.

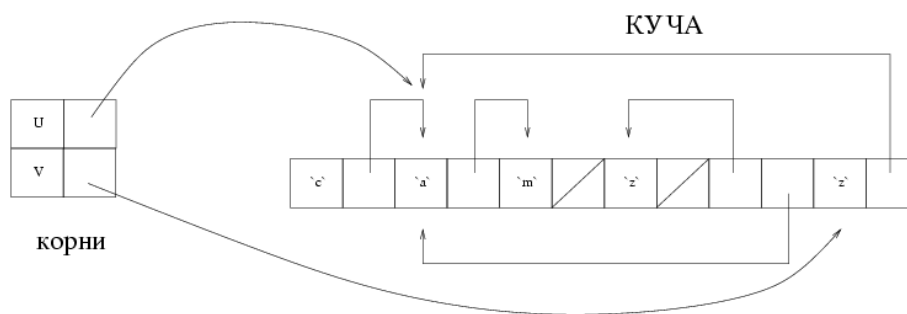


Рис. 8.4: Состояние кучи

Качества алгоритма-разведчика определяются следующими свойствами:

- эффективность: от чего зависит временная сложность: от размера кучи или размера активных ячеек?

- фактор сборки: доступна ли вся свободная память?
- компактность: вся свободная память доступна в виде одного блока?
- локализация: находятся ли рядом различные ячейки одной структуры?
- потребности в ресурсах: сколько памяти необходимо самому алгоритму для выполнения?
- перемещение значений: меняют ли значения свое расположение в памяти после “сборки мусора”?

Свойство локализации, при просмотре структурированного значения, позволяет избежать перехода на другую страницу памяти. Компактность предназначена для того чтобы избежать фрагментация памяти, и в то же время выделить блок памяти размером во всю доступную память. Эффективность, фактор сборки памяти и необходимость в дополнительной памяти тесно связаны с временной и пространственной сложностью алгоритма.

8.5.3 Mark&Sweep

Основная идея алгоритма Mark&Sweep заключается в том, что он поддерживает список свободных ячеек памяти кучи, называемый *freelist*. Этот список проверяется для каждого запроса выделения памяти. Если он пуст или количество свободных ячеек недостаточно, то выполняется Mark&Sweep.

Алгоритм действует в два этапа:

- начиная от корней пометить используемые ячейки памяти (*Mark*),
- последовательный просмотр всей кучи и восстановление не помеченных ячеек (*Sweep*).

Проиллюстрируем управление памятью алгоритмом Mark&Sweep изображением, на котором воспользуемся четырьмя цветами: белый, серый¹, черный и заштрихованный. Для этапа маркировки применяется черный и серый цвет, для восстановления заштрихованный и для выделения памяти белый.

Смысл цветов во время маркировки следующий:

- серый: ячейка помечена, ее потомки не помечены;

¹в on-line версии книги серый цвет имеет голубой оттенок

- **черный**: ячейка помечена, ее прямые потомки тоже помечены.

Необходимо хранить множество серых ячеек, для того чтобы удостовериться в том что все проверено. По окончании маркировки все ячейки либо белые либо черные. Ячейки, которые были достигнуты начиная от корней, помечены черным цветом. На рисунке 8.5 изображен промежуточный этап маркировки предыдущего примера 8.4: корень *u* был уже обработан, а *v* еще нет.

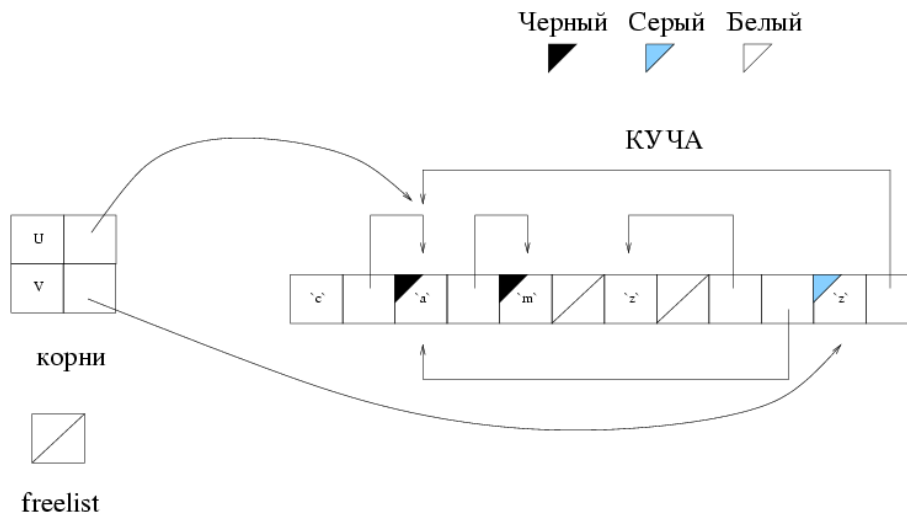


Рис. 8.5: Этап маркировки

Список свободных ячеек создается в течении этапа восстановления. Для того этапа используются следующие цвета:

- **черный** становится белым, ячейка используется
- **белый** становится заштрихованным, ячейка добавляется в *freelist*.

На рисунке 8.6 изображено изменение кучи и создание *freelist*.

Перечислим характеристики алгоритма Mark&Sweep:

- зависимость от размера кучи (этап Sweep)
- восстанавливает всю свободную память
- не сжимает память
- не гарантирует локализацию
- не перемещает данные

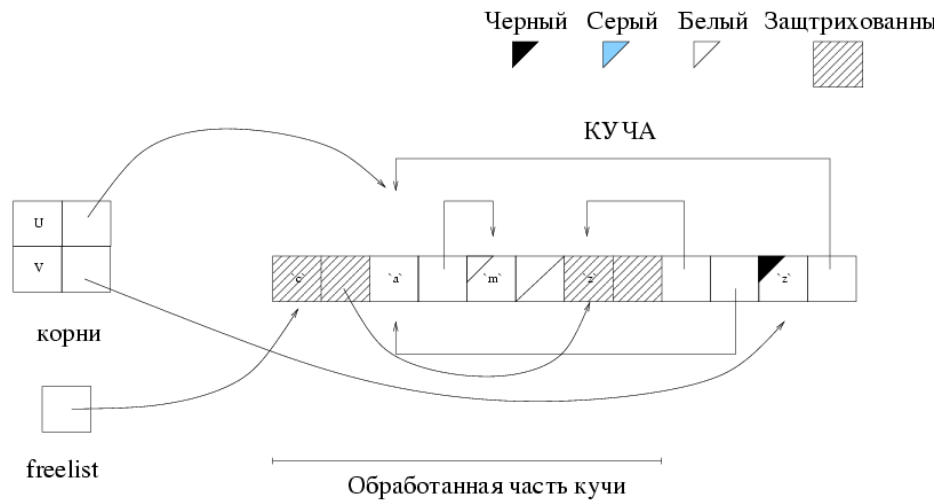


Рис. 8.6: Этап восстановления

Этап маркировки чаще всего реализован рекурсивной функцией и как следствие она использует часть стека выполнения. Мы могли бы провести полностью итеративную версию алгоритма Mark&Sweep, которая бы не использовала стек неизвестной глубины, но в итоге рекурсивная версия оказывается более эффективной.

Для функционирования алгоритму Mark&Sweep необходимо знать размер значений. Он хранится либо в самих значениях, либо вычтен из адресов памяти, разделением кучи на зону объектов с ограниченным размером. Этот алгоритм, используемый с первых версий Lisp, до сих пор широко применяется. Часть GC Objective CAML использует подобный алгоритм.

8.5.4 Stop&Copy

Основным принципом алгоритма Stop&Copy является использование вторичной памяти для копирования и сжатия данных. Куча разбита на две части: используемая зона (называемая from-space) и зона копирования (to-space).

Принцип действия следующий. Начиная от корней скопировать все значения из from-space в to-space. При этом сохраняем новый адрес перенесенного значения (чаще все новый адрес сохраняется на старом месте значения), для того чтобы обновить все другие значения указывающие на текущее.

Содержимое скопированных ячеек формирует новые корни. До тех

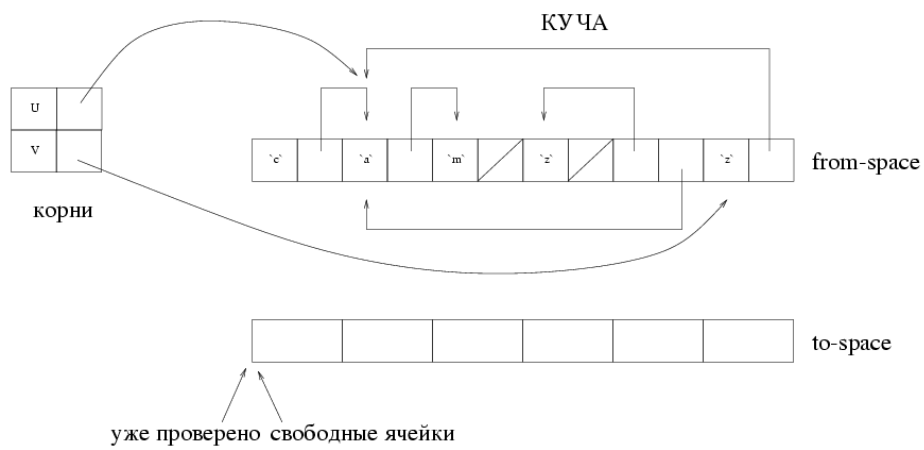


Рис. 8.7: Начало Stop&Copy

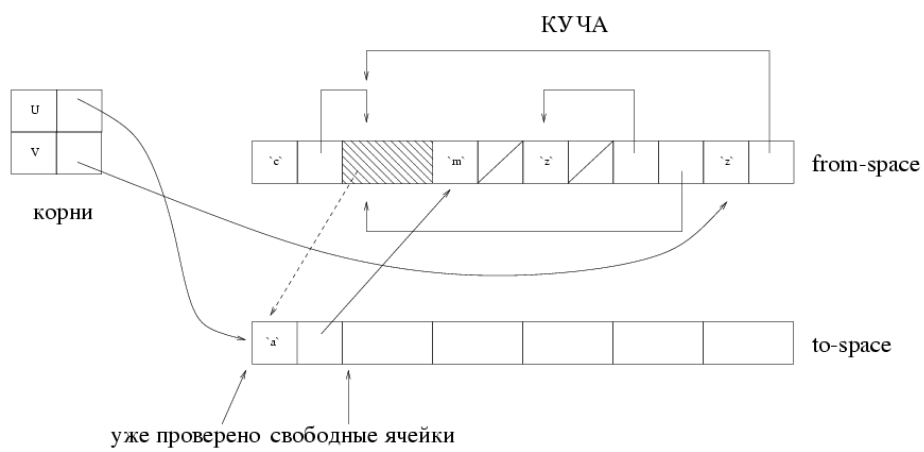


Рис. 8.8: Копирование из from-space в to-space

пор пока все корни не будут обработаны, алгоритм продолжает выполняться.

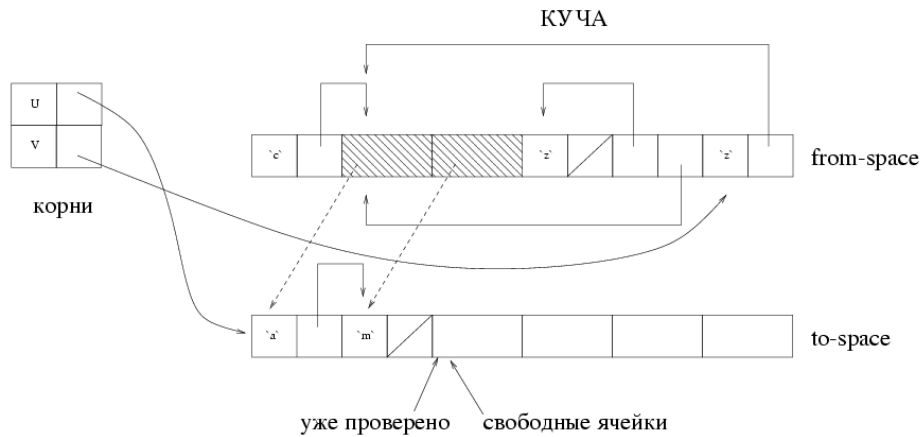


Рис. 8.9: Новые корни

В случае разделения, то есть когда копируемое значение уже скопировано, мы лишь указываем новый адрес значения.

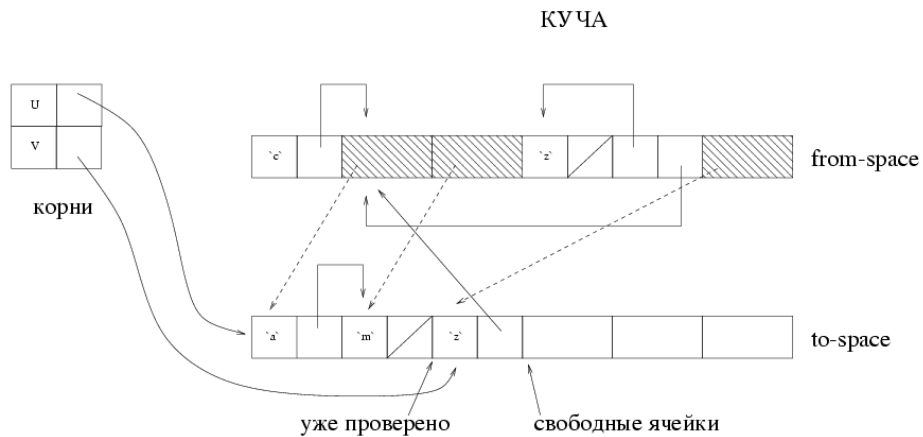


Рис. 8.10: Разделение значения

По окончании работы GC все корни обновлены и указывают на новые адреса. Обе зоны меняются ролями для будущего GC.

Перечислим основные характеристики алгоритма Stop&Copy:

- зависимость лишь от размера переносимых объектов
- только половина памяти доступна

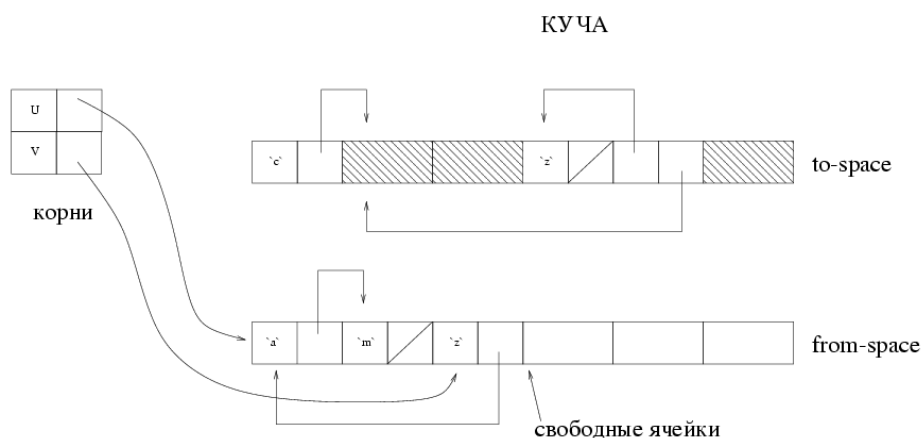


Рис. 8.11: Смена ролей зон памяти

- сжимает память
- возможность локализации (обход в ширину)
- не использует дополнительную память (from-space + to-space)
- не рекурсивный алгоритм
- перемещает данные на новое место

8.5.5 Другие GC

Существует немало других алгоритмов GC, многие из них основываются на двух предыдущих алгоритмах. Часто такие алгоритмы созданны специально для определенных проблем, как например для вычисления больших матриц, или же они зависят от типа компиляции. В Generational GC возможны оптимизации основанные на “возрасте” значения. Conservative GC используется в тех случаях, когда нет явной разницы между указателями и непосредственными значениями (например когда компилируем в код C). И наконец Incremental GC позволяет избежать замедления вызванного работой GC.

Generational Garbage Collection

Функциональные программы используют большое количество памяти и замечено, что очень много значений имеют очень короткое существование². С другой стороны, если какое-то значение выжило несколько

²Большинство не выживают после GC

GC, то вполне возможно что это значение будет еще долго существовать. Чтобы не обходить каждый раз всю кучу, как это делает алгоритм Mark&Sweep, во время высвобождения памяти, желательно чтобы GC проверял лишь значения “выжившие” после одного или нескольких GC (то есть новые значения). Чаще всего в этой области памяти мы высвобождаем больше места. С этой целью объекты помечаются либо временем либо числом GC которые они “пережили”. Для оптимизации GC используются различные алгоритмы в зависимости от возраста значений:

- GC новых значений должны быть быстрыми и проходить лишь “молодые” значения
- GC старых значений должны быть редкими и высвобождать всю занятую ими память.

Стареющее значение будет все меньше и меньше обследоваться частыми GC. Трудность заключается в том чтобы анализировать часть памяти содержащую лишь молодые объекты. В чистых функциональных языках, то есть где нет физического изменения, молодые объекты могут указывать на старые, но старые не могут указывать на молодые, так как те появились позже. Таким образом подобный механизм хорошо подходит функциональным языкам, за исключением тех, где имеет место отложенное вычисление, которое позволяет рассчитать элементы структуры, после расчета этой самой структуры. Однако в нечистых функциональных языках всегда можно изменить часть значения старого объекта, чтобы она ссылалась на более молодой. Теперь же проблема заключается в том, чтобы определить зону памяти только с молодыми объектами на которые указывают старые. Для этого нужно создать таблицу ссылок старых объектов на молодые, что позволит нам получить правильные GC. В следующей части, мы рассмотрим GC Objective CAML.

Conservative Garbage Collectors

До сих пор предполагалось что мы можем отличить указатель от немодифицируемого (атомарного) значения. Необходимо отметить, что в функциональных языках с параметризуемым полиморфизмом все значения занимают одно слово памяти³. Это свойство позволяет иметь общий (generic) код полиморфных функций.

³С единственным исключением для Objective CAML касающимся векторов чисел с плавающей запятой (см. гл. 11) на 367.

Однако, ограничение множества целых чисел может быть недопустимым. В этом случае, conservative garbage collectors позволяют избежать маркировки немедленных значений как целых и значения используют целое слово памяти без всяких меток. Чтобы не обходить зоны памяти с корня содержащего целое число, используется алгоритм различающий немедленные значения и корректные указатели. Этот алгоритм основывается на двух предположениях:

- любое значение выходящее за начальный и конечный адрес кучи считается немедленным;
- выделенные объекты выровнены по адресу памяти слова. Любое значение не соответствующее корректному выравниванию рассматривается как немедленное значение.

Таким образом любое правильное, по отношению к адресам кучи, значение будет рассмотрено как указатель и будет сохранено GC, даже если на самом деле это немедленное значение. С использованием специальных страниц памяти под объекты разного размера такие случаи могут стать очень редкими. Недостатком данного алгоритма является отсутствие гарантии, что вся неиспользуемая память будет восстановлена. Однако, мы уверены, что восстанавливается лишь неиспользуемая память.

Чаще всего, conservative garbage collectors не перемещают значения (отсюда название — консервативный). Действительно, в связи с тем что GC рассматривает некоторые немедленные значения как указатели, было бы ущемлено его переносить. Однако, в создание множества корней могут быть внесены особенности, позволяющие с уверенностью перенести часть памяти, которая соответствует “правильным” корням.

Такая схема действия GC часто используется при компиляции функционального языка в язык C, который рассматривается как переносимый ассемблер. При этом немедленные значения C занимают целое слово памяти.

Incremental Garbage Collection

Одним из часто упоминаемых недостатков GC является заметная пользователю остановка выполняющейся программы и при чем на неопределенное время. Первый случай может вызывать неудобство для определенного круга программ. Например, игры в стиле action. Ее остановка на несколько секунд будет плохо воспринята игроком, так как игра продолжится так же неожиданно как и была приостановлена. Второй случай приводит к потере контроля над программами, которые обрабатывают

определенные события в реальном времени. Типичный пример — встроенные программы контролирующие автомобиль или станок. Подобные программы реального времени, которые должны реагировать в ограниченный период времени, избегают использование GC.

Incremental Garbage Collection должны обладать способностью останавливаться в любой момент и затем продолжить свою работу, гарантируя безопасность высвобождения памяти. Такие сборщики дают неплохие результаты для первого случая. Они так же могут быть использованы во втором случае, если соблюдать следующую дисциплину программирования: четко разделять код на части которые используют динамическое выделения памяти и те которые не используют.

Вернемся к примеру *Mark&Sweep* и рассмотрим те изменения, которые необходимо сделать, что этот алгоритм стал инкрементальный. В принципе, таких изменений всего два:

- как удостоверится, что все что необходимо, было помечено во время этапа маркировки?
- как выделять память, во время процесса маркировки или сбора памяти?

Если алгоритм *Mark&Sweep* приостановлен во время процесса *Mark*, необходимо обеспечить чтобы память выделенная между остановкой маркировкой и затем возобновлением не была высвобождена во время *Sweep*. Для этого, ячейки памяти выделенные в подобный промежуток, заранее помечаются черным или серым цветом.

Если алгоритм *Mark&Sweep* приостановлен в момент *Sweep*, то он может быть затем продолжен как обычно “перекрашивая” выделенные ячейки памяти белым цветом. Действительно, во время этапа *Sweep* вся куча последовательно просматривается и память, выделенная во время приостановки сборщика, находится впереди точки, с которой алгоритм продолжит свою работу. Таким образом эта память не будет проверена до следующего цикла GC.

На рисунке 8.12 изображено выделение памяти во время этапа высвобождения памяти. Корень *w* создан следующим кодом:

```
# let w = 'f'::v;;
val w : char list = ['f'; 'z'; 'a'; 'm']
```

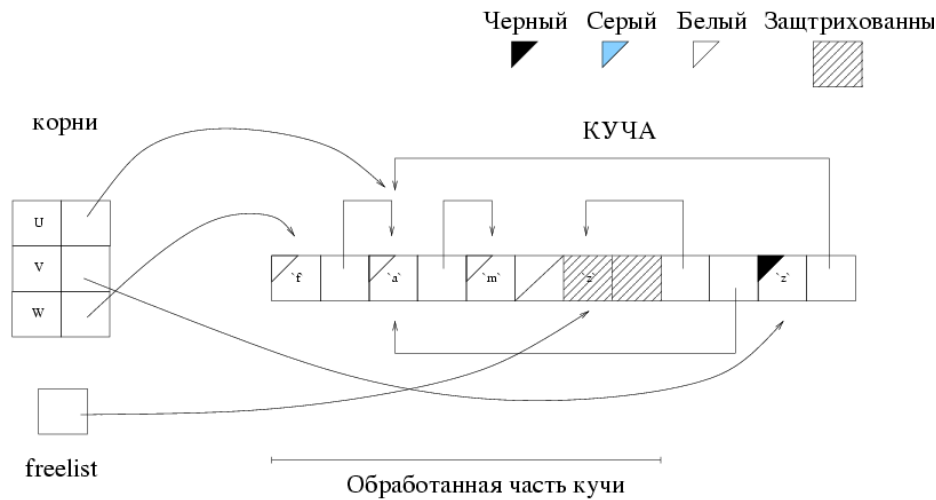


Рис. 8.12: Выделение во время высвобождения памяти

8.6 Управление памятью в Objective Caml

Сборщик мусора Objective CAML использует различные методы, которые мы привели в предыдущей части. Сборщик OCaml является сборщиком с двумя поколениями: новое и старое. Для молодых поколений используется в основном алгоритм *Stop&Copy* (minor garbage collection) и инкрементальный *Mark&Sweep* для старых поколений (major garbage collection).

Объект молодого поколения, который продолжает существовать после минорного GC перемещается в старое поколение. Пространство **to-space** используется алгоритмом *Stop&Copy* как пространство старого поколения. По окончании этого процесса, пространство **from-space** полностью свободно.

Когда мы затронули тему сборщиков с поколениями, мы отметили проблему нечистых функциональных языков: значение старого поколения может ссылаться на объект из нового поколения. Небольшой пример:

```
# let older = ref [1] ;;
val older : int list ref = {contents=[1]}
(* ... *)
# let newer = [2;5;8] in
  older := newer ;;
- : unit = ()
```

Предполагается что на месте комментариев имеется большая последовательность кода, во время которой значение **older** переходит в старое

поколение. Минорный GC должен учитывать некоторые значения старого поколения. То есть необходимо поддерживать таблицу ссылок объектов старого поколения на объекты нового, которые становятся частью корней минорного GC. Эта таблица почти не увеличивается и к концу минорного GC она становится пустой.

Напомним, что *Mark&Sweep* старого поколения — инкрементальный, то есть часть мажорного GC выполняется во время каждого минорного GC. Мажорный GC — это *Mark&Sweep*, который следует алгоритму описанному на стр. 287. Выгода в подобном подходе заключается в уменьшении времени ожидания мажорного GC, предварительном выполнении фазы маркировки во время каждого минорного GC. Когда запускается мажорный GC, процесс маркировки необработанных областей уже закончен и этап сборки памяти запущен. Так как алгоритм *Mark&Sweep* может привести к сильной фрагментации памяти, по этой причине процесс сжатия памяти может быть запущен после мажорного GC.

В сумме, мы получаем следующие этапы:

- минорный GC: выполняет *Stop&Copy* на молодом поколении, “устаревает” выжившие объекты, переместив их из одной зоны в другую и затем выполняет часть *Mark&Sweep* старого поколения. Если смена зоны памяти не удалась, то сам процесс не удался. В этом случае переходим к этапу 2.
- Конец цикла мажорного GC В случае ошибки переходим к этапу 3.
- Снова мажорный GC, чтобы проверить стали ли объекты отмеченные как используемые во время инкрементального этапа освобождены В случае ошибки переходим к этапу 4.
- Сжатие зоны старого поколения, с целью получить как можно большее непрерывное пространство памяти. В случае ошибки на последнем этапе программа останавливается.

Модуль GC позволяет запустить один из этапов GC.

Особенность сборщика Objective CAML в том что размер кучи не фиксирован в начале программы, он может увеличиваться или уменьшаться по мере необходимости программы.

8.7 Модуль GC

При помощи этого модуля мы можем контролировать кучу, получить различную информацию о ее использовании, а так же запустить различ-

stat	unit -> stat
print_stat	out_channel -> unit
get	unit control
set	control -> unit

Таблица 8.1: Функции контроля и информации о куче.

ные этапы сборщика. В данном модуле определены две записи: **stat** и **control**. Тип полей записи **control** модифицируемый и при их помощи можно контролировать поведение сборщика. Поля записи **stat** нельзя изменить, они лишь отражают состояние сборщика данный момент.

Поля записи **stat** содержат следующие счетчики:

- число GC: `minor_collections`, `major_collections` и `compactions`
- число слов, которые были перемещены с момента запуска программы `minor_words`, `promoted_words`, `major_words`.

В записи **control** имеются следующие записи:

- `minor_heap_size` определяет размер зоны для молодого поколения.
- `major_heap_increment` определяет прирост зоны памяти старого поколения.
- `space_overhead` определяет порог памяти в процентах. При его превышении запускается мажорный GC (по умолчанию это значение равно 42).
- `max_overhead` определяет отношение между размером занятой и свободной памяти, начиная с которого запускается сжатие памяти. При значении 0 сжатие запускается после каждого мажорного GC. Максимальное значение 1000000 подавляет запуск сжатия.

Приведем следующие функции манипулирующие типами **stat** и **control** в таблице 8.1.

При помощи следующих функций типа `unit -> unit` можно принудительно выполнить определенные этапы GC: `minor` (первый этап), `major` (первый и второй этапы), `full_major` (первый, второй и третий этапы), `compact` (первый, второй, третий и четвертый этапы).

Примеры

```
# Gc.stat();
- : Gc.stat =
{Gc.minor_words=549290; Gc.promoted_words=60620; Gc.major_words
  =204615;
  Gc.minor_collections=17; Gc.major_collections=2; Gc.heap_words
    =190464;
  Gc.heap_chunks=3; Gc.live_words=52727; Gc.live_blocks=12959;
  Gc.free_words=31155; Gc.free_blocks=84; Gc.largest_free=17899;
  Gc.fragments=3; Gc.compactions=0}
```

Здесь мы видим число запусков различных этапов: минорный GC, мажорный GC и сжатие, а так же число слов обработанных различными частями памяти. Вызов `compact` принудительно выполняет все четыре этапа GC и изменяет статистическую информацию о куче (см. вызов `Gc.stat`).

```
# Gc.compact();
- : unit = ()
# Gc.stat();
- : Gc.stat =
{Gc.minor_words=555758; Gc.promoted_words=61651; Gc.major_words
  =205646;
  Gc.minor_collections=18; Gc.major_collections=4; Gc.heap_words
    =190464;
  Gc.heap_chunks=3; Gc.live_words=130637; Gc.live_blocks=30770;
  Gc.free_words=59827; Gc.free_blocks=1; Gc.largest_free=59827;
  Gc.fragments=0; Gc.compactions=1}
```

Поля `GC.minor_collections` и `compactions` увеличены на 1, тогда как поле `Gc.major_collections` увеличено на 2. Все поля типа `GC.control` является изменяемыми. Для того чтобы изменения вошли в силу, необходимо использовать функцию `Gc.set`. Эта функция ожидает на входе значение типа `control` и изменяет поведение GC.

Поле `verbose` может иметь значения от 0 до 127, активируя 7 различных индикаторов.

```
# c.Gc.verbose <- 31;;
Characters 1–2:
This expression has type int * int but is here used with type Gc.control
# Gc.set c;;
Characters 7–8:
```

This expression has **type** `int * int` but is here used **with type** `Gc.control`
`# Gc.compact();;`
`- : unit = ()`

И получим следующий вывод на экране:

```
<>Starting new major GC cycle
allocated_words = 329
extra_heap_memory = 0u
amount of work to do = 3285u
Marking 1274 words
!Starting new major GC cycle
Compacting heap...
done.
```

Здесь выводятся различные этапы GC и число обработанных объектов.

8.8 Модуль Weak

Слабый указатель (weak pointer) это указатель, память которого может быть высвобождена GC в любой момент. Это может звучать странно, что значение может исчезнуть в любой момент. В действительности необходимо рассматривать эти указатели как хранилище еще пока доступных значений. Подобное свойство может оказаться очень полезным в случае если мы располагаем не большой по сравнению с сохраняемыми элементами памятью. Классический примером является управление кэшем: значение может быть утеряно, но оно остается доступным, до тех пока оно существует.

В Objective CAML нет возможности управлять единичным слабым указателем, а лишь вектором указателей. В модуле `Weak` определен абстрактный тип `'a Weak.t`, который соответствует абстрактному типу `'a option array`, вектор слабых указателей типа `'a`. Конкретный тип `'a option` определяется следующим образом:

```
type 'a option = None | Some of 'a;;
```

В таблице 8.2 представлены основные функции данного модуля.

Функция `create` выделяет пространство для вектора слабых указателей, каждый элемент вектора инициализирован значением `None`. Функция `set` устанавливает значение типа `'a Option` по указанному индексу вектора. Функция `get` возвращает значение расположенное по указанному индексу в векторе. Полученное значение вносится в множество корней GC и не может быть высвобождено до тех пор пока не него существуют

функция	тип
create	<code>int -> 'a t</code>
set	<code>'a t -> int -> 'a option -> unit</code>
get	<code>'a t -> int -> 'a option</code>
check	<code>'a t -> int -> bool</code>

Таблица 8.2: Основные функции модуля `Weak`.

ссылки. Для того чтобы проверить существует ли желаемое значение, можно использовать либо функцию `check`, либо сопоставлением с образцом типа `'a Option`. Первый вариант не зависит от представления слабых указателей в памяти.

Так же существуют обычные функции для линейных данных: `length` для определения длины, `fill` и `blit` для копирования частей вектора.

Пример: кэш изображений

Довольно часто в графических программах открыто сразу несколько изображений. Когда мы переключаемся с одного изображения на другое, первое сохраняется на диск, а второе загружается из другого файла. Обычно, программа хранит имена послених обработанных изображений. Во избежание постоянного доступа к диску и эффективного использования памяти, используется кэш хранящий последние загруженные изображения. Содержимое кэша изображений может быть очищенной в любой момент по необходимости. Такой кэш реализуется в виде вектора слабых указателей и управление кэшем доверяется GC. Во время загрузки изображения, сначала проверяется присутствует ли оно уже в кэше, если да, то оно становится текущим изображением, иначе оно загружается с диска.

Вектор изображений реализуется следующим способом:

```
# type table_of_images = {
  size : int;
  mutable ind : int;
  mutable name : string;
  mutable current : Graphics.color array array;
  cache : ( string * Graphics.color array array) Weak.t } ;;
```

В поле `size` хранится размер вектора, `ind` является индексом текущего изображения, в `name` хранится имя текущего изображения, в поле `current` содержится текущее изображение, а вектор `cache` хранит слабые указатели на изображения. В этом векторе содержатся последние

загруженные изображения и их имена.

Функция `init_table` инициализирует вектор начальными изображениями.

```
# let open_image filename =
  let ic = open_in filename
  in let i = ((input_value ic) : Graphics.color array array)
  in ( close_in ic ; i ) ;;
val open_image : string -> Graphics.color array array = <fun>

# let init_table n filename =
  let i = open_image filename
  in let c = Weak.create n
  in Weak.set c 0 (Some (filename,i)) ;
  { size=n; ind=0; name = filename; current = i; cache = c } ;;
val init_table : int -> string -> table_of_images = <fun>
```

При загрузке нового изображения, текущее сохраняется в векторе и затем загружается новое изображение. Перед этим, необходимо просмотреть кэш, на случай если искомое изображение уже загружено.

```
# exception Found of int * Graphics.color array array ;;
# let search_table filename table =
  try
    for i=0 to table.size-1 do
      if i<>table.ind then match Weak.get table.cache i with
        Some (n,img) when n=filename -> raise (Found (i,img))
        | _ -> ()
      done ;
    None
  with Found (i,img) -> Some (i,img) ;;

# let load_table filename table =
  if table.name = filename then () (* the image is the current image *)
  else
    match search_table filename table with
    Some (i,img) ->
      (* the image found becomes the current image *)
      table.current <- img ;
      table.name <- filename ;
      table.ind <- i
    | None ->
      (* the image isn't in the cache, need to load it *)
```

```

(* find an empty spot in the cache *)
let i = ref 0 in
  while (!i < table.size && Weak.check table.cache !i) do incr i
  done ;
(* if none are free, take a full slot *)
( if !i=table.size then i:=(table.ind+1) mod table.size ) ;
(* load the image here and make it the current one *)
table.current <- open_image filename ;
table.ind <- !i ;
table.name <- filename ;
Weak.set table.cache table.ind (Some (filename,table.current
)) ;;
val load_table : string -> table_of_images -> unit = <fun>

```

Функция `load_table` проверяет является ли запрашиваемое изображение текущим, если нет, то проверка продолжается в кэше и в конечном итоге загружается с диска если поиск в кэше ничего не дал. В последних двух случаях это изображение становится текущим.

Для того чтобы проверить эту функцию, воспользуемся следующим кодом, выводящий содержимое кэша:

```

# let print_table table =
  for i = 0 to table.size-1 do
    match Weak.get table.cache ((i+table.ind) mod table.size) with
    | None -> print_string "[ ] "
    | Some (n,_) -> print_string n ; print_string " "
  done ;;
val print_table : table_of_images -> unit = <fun>

```

Протестируем конечную программу:

```

# let t = init_table 10 "IMAGES/animfond.caa" ;;
val t : table_of_images =
{ size=10; ind=0; name="IMAGES/animfond.caa";
  current=
  [||7372452; 7372452; 7372452; 7372452; 7372452; 7372452; 7372452;
    7372452; 7372452; 7372452; 7372452; 7505571; 7505571;
    ...||];
  ...||;
  cache=...}
# load_table "IMAGES/anim.caa" t ;;
- : unit = ()
# print_table t ;;

```

IMAGES/anim.caa [] [] [] [] [] [] [] [] - : unit = ()

Этот механизм кэша может быть использован в различных случаях.

8.9 Резюме

В данной главе были приведены различные алгоритмы высвобождения памяти. Это было сделано с целью представить алгоритм используемый Objective CAML. GC Objective CAML является инкрементальным GC, с двумя поколениями значений. Он использует алгоритм **Mark&Sweep** для старого поколения и **Stop&Copy** для молодого. Два модуля, напрямую связанные с GC, позволяют контролировать состояние кучи. При помощи модуля **Gc** мы можем проанализировать поведение кучи и изменить некоторые ее параметры, чтобы оптимизировать ее для определенных приложений. При помощи модуля **Weak** мы можем сохранить в векторе потенциально высвобождаемые, но еще доступные значения. Это свойство бывает полезным для реализации кэша памяти.

Глава 9

Средства анализа программ

9.1 Введение

Средства анализа программ предоставляют дополнительную информацию, по сравнению с компилятором или компоновщиком. Часть этих средств реализуют статический анализ. То есть они анализируют (в виде исходного кода или синтаксического дерева) и определяют различные свойства кода, такие как зависимости между модулями или возможные исключения. Другие средства проводят динамический анализ, то есть изучают программы во время ее выполнения. Такие средства полезны, в случае когда необходимо знать сколько раз была вызвана определенная функция, какие аргументы были ей переданы или время, затраченное на выполнение части кода программы. Такие средства анализа могут быть интерактивными, как например отладчик программ. В этом случае выполнение программы изменяется, для того чтобы реагировать на команды пользователя, который может установить контрольные точки чтобы просмотреть значения или перезапустить программу с новыми аргументами.

В дистрибутив Objective CAML входят подобные средства. Некоторые из них имеют не часто встречающиеся характеристики, связанные со статической типизацией. Такая типизация гарантирует, что во время выполнения кода, не произойдет ошибок типа и используется компилятором для того чтобы генерировать эффективный код небольшого размера. Часть информации о типе созданных значений теряется и из-за этой особенности нельзя просмотреть аргументы полиморфных функций.

9.2 План главы

В этой короткой главе представляются средства анализа входящие в дистрибутив Objective CAML. В первой части описывается команда `ocamldep`, которая определяет зависимости множества файлов Objective CAML принадлежащих одной и той же программе. Во второй части представляются средства отладки, как например результаты применения функций или стандартное средство отладки — `ocamldebug` для операционных систем Unix. В третьей части мы рассмотрим профайлер. Он анализирует программы и результат этого анализа может быть использован для оптимизации этой программы.

Вычисление зависимостей между файлами интерфейса и реализации имеет двойную цель. Первая — получить глобальное видение зависимостей между модулями? Вторая — использовать данную информацию для того чтобы компилировать лишь необходимый минимум при изменении некоторых файлов.

Команда `ocamldep` анализирует файлы `.ml` и `.mli` и выводит зависимости файлов в формате `Makefile`¹.

Зависимости появляются при глобальной декларации других модулей, либо используя синтаксис с точкой (пример: `M1.f`), либо открывая модуль (пример: `open M1`).

Предположим, что имеются файлы `dp.ml`:

```
let print_vect v =
  for i = 0 to Array.length v do
    Printf.printf "%f " v.(i)
  done;
  print_newline();
и d1.ml:
let init n e =
  let v = Array.create 4 3.14 in
  Dp.print_vect v;
  v;;
```

При вызове команды `ocamldep` с указанием файлов, получим следующий результат:

```
$ ocamldep dp.ml d1.ml array.ml array.mli printf.ml printf.mli
dp.cmo: array.cmi printf.cmi
dp.cmx: array.cmx printf.cmx
```

¹Файлы `Makefile` используются командой `make` чтобы обновлять при необходимости файлы или программы

```
d1.cmo: array.cmi dp.cmo
d1.cmx: array.cmx dp.cmx
array.cmo: array.cmi
array.cmx: array.cmi
printf.cmo: printf.cmi
printf.cmx: printf.cmi
```

Зависимости вычисляются для байт-код и нативных компиляторов. Поясним полученный результат: компиляция файла `dp.cmo` зависит от файлов `array.cmi` и `printf.cmi`. Файлы с расширением `.cmi` зависят от файлов с таким же именем, но с расширением `.mli`. Аналогичное правило распространяется на файлы `.ml` с `.cmo` и `.cmx`.

Объектные файлы дистрибутива отсутствуют в списке зависимостей. В действительности, если команда `ocamldep` не найдет их в текущем каталоге, то они будут найдены в каталоге библиотек дистрибутива и выдаст следующий результат:

```
$ ocamldep dp.ml d1.ml
d1.cmo: dp.cmo
d1.cmx: dp.cmx
```

Для того, чтобы указать команде `ocamldep` дополнительные каталоги для поиска файлов, используйте опцию `-I каталог`.

9.3 Средства отладки

Существует два типа средств для отладки программ. Первый тип это трассировка глобальных функций, интерактивного интерпретатора. Второе средство это отладчик, который не используется в нормальном интерактивном интерпретаторе. После первичного выполнения программы, отладчик позволяет вернуться к контрольным точкам, просмотреть определенные значения или перезапустить функции с новыми параметрами. Данный отладчик доступен лишь в среде Unix, потому что он использует вызов `fork` для создания процесса программы.

9.3.1 Trace

Трассировкой называется вывод входных аргументов функции и ее результата в момент выполнения программы. Команды трассировки это директивы интерактивного интерпретатора. При помощи директив можно начать трассировку функции, остановить трассировку функции и

остановить все установленные трассировки. Эти три директивы представлены в следующей таблице:

<code>#trace имя_функции</code>	трассировать указанную функцию
<code>#untrace имя_функции</code>	прекратить трассировку указанной функции
<code>#untrace_all</code>	прекратить все трассировки

В следующем примере, определим функцию:

```
# let f x = x + 1;;
val f : int -> int = <fun>
# f 4;;
- : int = 5
```

Теперь зададим директиву трассировки данной функции, после чего переданные аргументы и результат будут выведены на экран.

```
# #trace f;;
f is now traced.
# f 4;;
f <-- 4
f --> 5
- : int = 5
```

Передача аргумента 4 функции `f` выводится на экран, функция реализует вычисление и затем результат так же выводится на экран. Значения параметров вызова функции указываются стрелкой влево, а возвращаемое значение стрелкой вправо.

Функции с несколькими параметрами

Таким же образом можно трассировать функции с несколькими аргументами (или возвращающими замыкание). Каждая передача аргумента выводится на экран. Для того чтобы различить замыкания, символом `*` помечается число уже переданных замыканию аргументов. Пусть имеется функция `verif_div` с четырьмя аргументами `a b c d`, соответствующие целочисленному делению: `a = bq + r`.

```
# let verif_div a b q r =
  a = b*q + r;;
val verif_div : int -> int -> int -> int -> bool = <fun>
# verif_div 11 5 2 1;;
- : bool = true
```

Трассировка данной функции выводит на экран передачу 4 аргумента:


```
# #trace verif_div;;
verif_div is now traced.
# verif_div 11 5 2 1;;
verif_div <-- 11
verif_div --> <fun>
verif_div* <-- 5
verif_div* --> <fun>
verif_div** <-- 2
verif_div** --> <fun>
verif_div*** <-- 1
verif_div*** --> true
- : bool = true
```

Рекурсивные функции

Трассировка дает ценную информацию о рекурсивных функциях. Можно с легкостью определить неверный критерий остановки рекурсии.

Определим функцию `belongs_to`, которая проверяет принадлежит ли число списку целых чисел:

```
# let rec belongs_to (e : int) l = match l with
| [] -> false
| t :: q -> (e = t) || belongs_to e q ;;
val belongs_to : int -> int list -> bool = <fun>
# belongs_to 4 [3;5;7] ;;
- : bool = false
# belongs_to 4 [1; 2; 3; 4; 5; 6; 7; 8] ;;
- : bool = true
```

Трассировка вызова функции `belongs_to 4 [3;5;7]` выведет на экран передачу четырех аргументов и возвращенные результаты.

```
# #trace belongs_to ;;
belongs_to is now traced.
# belongs_to 4 [3;5;7] ;;
belongs_to <-- 4
belongs_to --> <fun>
belongs_to* <-- [3; 5; 7]
belongs_to <-- 4
belongs_to --> <fun>
belongs_to* <-- [5; 7]
belongs_to <-- 4
```

```

belongs_to --> <fun>
belongs_to* <-- [7]
belongs_to <-- 4
belongs_to --> <fun>
belongs_to* <-- []
belongs_to* --> false
belongs_to* --> false
belongs_to* --> false
belongs_to* --> false
- : bool = false

```

При каждом вызове функции `belongs_to` ей передается аргумент 4 и тестируемый список. Когда передается пустой список, то функция возвращает **false**. Это значение передается каждому ожидающему рекурсивному вызову.

В следующем примере список пройден лишь частично, до того места где находится нужный элемент:

```

# belongs_to 4 [1; 2; 3; 4; 5; 6; 7; 8] ;;
belongs_to <-- 4
belongs_to --> <fun>
belongs_to* <-- [1; 2; 3; 4; 5; 6; 7; 8]
belongs_to <-- 4
belongs_to --> <fun>
belongs_to* <-- [2; 3; 4; 5; 6; 7; 8]
belongs_to <-- 4
belongs_to --> <fun>
belongs_to* <-- [3; 4; 5; 6; 7; 8]
belongs_to <-- 4
belongs_to --> <fun>
belongs_to* <-- [4; 5; 6; 7; 8]
belongs_to* --> true
belongs_to* --> true
belongs_to* --> true
belongs_to* --> true
- : bool = true

```

Как только 4 оказывается в заголовке списка, функция возвращает значение **true** и оно возвращается всем ожидающим рекурсивным вызовам.

Если в функции `belongs_to` поменять порядок `'||'`, то функция все так же будет возвращать правильный результат, но при этом она будет проверять список до конца.

```

# let rec belongs_to (e : int) = function
  || -> false
  | t::q -> belongs_to e q || (e = t) ;;
val belongs_to : int -> int list -> bool = <fun>
# #trace belongs_to ;;
belongs_to is now traced.
# belongs_to 3 [3;5;7] ;;
belongs_to <-- 3
belongs_to --> <fun>
belongs_to* <-- [3; 5; 7]
belongs_to <-- 3
belongs_to --> <fun>
belongs_to* <-- [5; 7]
belongs_to <-- 3
belongs_to --> <fun>
belongs_to* <-- [7]
belongs_to <-- 3
belongs_to --> <fun>
belongs_to* <-- []
belongs_to* --> false
belongs_to* --> false
belongs_to* --> false
belongs_to* --> true
- : bool = true

```

Несмотря на то, что число 3 находится в начале списка, он, список, проверяется до конца. Таким образом трассировка помогает анализировать эффективность рекурсивных функций.

Полиморфные функции

Трассировка аргументов полиморфной функции не выводит на экран значение параметризованного типа. Пусть функция `belongs_to` написана без явного указания типа.

```

# let rec belongs_to e l = match l with
  || -> false
  | t::q -> (e = t) || belongs_to e q ;;
val belongs_to : 'a -> 'a list -> bool = <fun>

```

Теперь, функция `belongs_to` стала полиморфная и в ее трассировке вывод значений аргументов заменен на `<poly>`.

```

# #trace belongs_to ;;

```

```

belongs_to is now traced.
# belongs_to 3 [2;3;4] ;;
belongs_to <-- <poly>
belongs_to --> <fun>
belongs_to* <-- [<poly>; <poly>; <poly>]
belongs_to <-- <poly>
belongs_to --> <fun>
belongs_to* <-- [<poly>; <poly>]
belongs_to* --> true
belongs_to* --> true
- : bool = true

```

Интерактивный интерпретатор умеет выводить на экран только значения мономорфных типов. К тому же он сохраняет лишь выведенный тип глобальных деклараций. То есть после компиляции выражения `belongs_to 3 [2;3;4]`, интерактивный интерпретатор не располагает другой информацией о типе, кроме того, что тип функции `belongs_to` есть `a -> 'a list -> bool`. Типы (мономорфные) `3` и `[2;3;4]` утеряны, так как при статической типизации значения не хранят информацию о типе. Поэтому, трассировка программы ассоциирует значениям функции `belongs_to` полиморфные типы `'a` и `'a list`.

Из-за того что значения не сохраняют информацию о типе, мы не можем создать общую функцию `print` с типом `a -> unit`.

Локальные функции

Локальные функции не трассируются по аналогичным причинам, которые связаны со статической типизацией. В окружении интерактивного интерпретатора сохраняются только типы глобальных деклараций. Однако, не смотря на это, следующий стиль программирования достаточно распространен:

```

# let belongs_to e l =
  let rec bel_aux l = match l with
    [] -> false
  | t :: q -> (e = t) || (bel_aux q)
  in
    bel_aux l;;
val belongs_to : 'a -> 'a list -> bool = <fun>

```

Глобальная функция лишь вызывает локальную, чтобы она выполнила необходимую задачу.

Замечания о трассировке

На данный момент, трассировка является единственным мультиплатформенным средством отладки. Его недостатками являются отсутствие вывода для локальных объявлений и полиморфных параметров функций. По этой причине трассировка мало используется, особенно на первых этапах знакомства с языком.

9.3.2 Отладка программ

Средством отладки программ или отладчик в Objective CAML является `ocamldeb`. Он позволяет пошаговое выполнение программы, вставлять контрольные точки, просмотреть или изменить значения окружения.

Сказать пошаговое выполнения, значить осознавать что такое в программе шаг. В императивном программировании это понятие соответствует инструкции языка. Однако подобное определение не имеет смысла в функциональном программировании. Здесь скорее говорят о событии (*event*) программы. Событием может быть применения, вход функции, сопоставление с образцом, условие, цикл, элемент последовательности.

Warning

Отладчик доступен только для систем Unix.

Компиляция для отладчика

При компиляции с опцией `-g` создается файл с расширением `.cmo`, что позволяет создать инструкции необходимые для отладки. Данная опция существует только для байт-код компилятора. При компиляции различных файлов программы, необходимо указать данную опцию. После того как получен исполняемый файл, необходимо запустить `ocamldebug` следующим образом:

```
ocamldebug [options] executable [arguments]
```

Пусть есть файл `fact.ml` со следующим кодом для вычисления факториала:

```
let fact n =  
  let rec fact_aux p q n =  
    if n = 0 then p  
    else fact_aux (p+q) p (n-1)  
  in  
  fact_aux 1 1 n;;
```

Основная программа находится в файле `main.ml`, она запускает долгую рекурсию вызовом `Fact.fact` на `-1`.

```
let x = ref 4;;
let go () =
  x := -1;
  Fact.fact !x;;
go();;
```

Оба файла компилируются с опцией `-g`:

```
$ ocamlc -g -i -o fact.exe fact.ml main.ml
val fact : int -> int
val x : int ref
val go : unit -> int
```

Запуск отладчика

После того как программа скомпилирована подобным образом, ее запуск в режиме отладчика происходит следующим образом:

```
$ ocamldebug fact.exe
Objective Caml Debugger version 3.00

(oed)
```

9.3.3 Контроль над выполнение программы

Контроль над выполнением реализуется при помощи событий программы. Можно либо продвинуться вперед или назад на n элементов программы, либо продвинуться вперед или назад до первой точки контроля (или на n точек). Точку контроля можно установить на функцию или элемент программы. Элемент языка можно выбрать по номеру линии, столбца или номера символа. Эти координаты могут быть определены по отношению к модулю.

В приведенном ниже примере, мы устанавливаем точку на четвертой строке модуля `Main`:

```
(oed) step 0
Loading program... done.
Time : 0
Beginning of program.
```

```
(ocd) break @ Main 4
Breakpoint 1 at 4964 : file Main, line 4 column 3
```

Инициализация модуля происходит до самой программы. И по этой причине контрольная точка 4 находится после 4964 инструкций.

Перемещаться по программе можно либо по отношению к элементам программы либо по отношению к контрольным точкам. **run** и **reverse** выполняют программу до первой встретившейся точки. В первом случае выполнение происходит в нормальном направлении, а во втором в обратном порядке. По команде **step** выполняется один или n элементов программы входя в функции, а **next** не входя в функции. Соответственно команды **backstep** и **previous** выполняют программу в обратном порядке. И наконец команда **finish** завершает вызов текущей функции, а **start** возвращается к элементу стоящему перед вызовом функции.

В продолжении примера, продвинемся до контрольной точки, а затем выполним три элемента программы.

```
(ocd) run
Time : 6 — pc : 4964 — module Main
Breakpoint : 1
4 <|b|>Fact.fact !x;;
(ocd) step
Time : 7 — pc : 4860 — module Fact
2 <|b|>let rec fact_aux p q n =
(ocd) step
Time : 8 — pc : 4876 — module Fact
6 <|b|>fact_aux 1 1 n;;
(ocd) step
Time : 9 — pc : 4788 — module Fact
3 <|b|>if n = 0 then p
```

Просмотр значений

В контрольной точке можно просмотреть значения связанные с активным блоком. Команды **print** и **display** выводят в зависимости от глубины значения.

Выведем значение **n**, затем вернемся на три элемента назад и выведем значение **x**.

```
(ocd) print n
n : int = -1
(ocd) backstep 3
```

```
Time : 6 — pc : 4964 — module Main
Breakpoint : 1
4  <|b|>Fact.fact !x;;
(oed) print x
x : int ref = {contents=-1}
```

Эти же команды позволяют вывести содержимое вектора или поле записи.

```
(oed) print x.contents
$1 : int = -1
```

Стек выполнения

Вложенные вызовы функций можно просмотреть в стеке выполнения. Для этого существует команда **backtrace** или **bt**, которая выводит порядок в котором вызывались функции. При помощи команд **up down** выбирается следующий или предыдущих активный блок. Для описания текущего блока используется команда **frame**.

9.4 Профайлер

С помощью профайлера можно получить определенную информацию о выполнении программы: сколько раз была выполнена функция или структура контроля (условный, цикл или сопоставление с образцом). Эта информация сохраняется в файле, анализируя который, можно обнаружить алгоритмические ошибки или критические моменты, которые можно оптимизировать.

Для того, чтобы профайлер смог реализовать анализ, необходимо скомпилировать код со специальными опциями. Эти опции добавляют в код инструкции, необходимые для сбора данных. Существует две методики профайлинга: одна — для байт-код компилятора и другая — для нативного компилятора, плюс две команды для представления результата. При анализе машинного кода мы получим так же время затрачено на выполнение каждой функции.

Профайлинг программы состоит из трех этапов:

- компиляция с опциями для профайлинга
- выполнение программы
- представление результатов

f	вызов функции
i	ответвления if
l	цикл while и for
m	ответвления match
t	ответвления try
a	все опции

Таблица 9.1: Опции команд профайлинга.

9.4.1 Команды компиляции

Ниже приводятся команды компиляции с опциями для профайлинга

- `ocamlcp -p` опции для байт-код компилятора
- `ocamlcpt -p` опции для нативного компилятора

Указанные компиляторы создают такие же типы файлов, что и обычные команды компиляции (6). В таблице 9.1 приведены различные опции компиляции.

Данные команды указывают, какие структуры контроля должны учитываться профайлером. По умолчанию используются опции `fm`.

9.4.2 Выполнение программы

Байт-код компилятор

При успешном выполнении программы, которая была скомпилирована со специальными для профайлера опциями, мы получим файл `ocamlprof.dump`, который содержит желаемую информацию.

Вернемся к нашему примеру произведения элементов целочисленного списка. Пусть файл `f1.ml` содержит следующий код:

```
let rec interval a b =
  if b < a then []
  else a :: (interval (a+1) b);;

exception Found_zero ;;

let mult_list l =
  let rec mult_rec l = match l with
    | [] -> 1
    | 0 :: _ -> raise Found_zero
```

```

    | n::x -> n * (mult_rec x)
  in
    try mult_rec l with Found_zero -> 0
;;

```

А так же файл `f2.ml`, в котором вызываются функции из `f1.ml`:

```

let l1 = F1.interval 1 30;;
let l2 = F1.interval 31 60;;
let l3 = l1 @ (0::l2) ;;

print_int (F1.mult_list l1) ;;
print_newline();;

print_int (F1.mult_list l3) ;;
print_newline();;

```

Компиляция файлов для профайлера осуществляется так:

```

ocamlcp -i -p a -c f1.ml
val profile_f1_ : int array
val interval : int -> int -> int list
exception Found_zero
val mult_list : int list -> int

```

С опцией `-p`, компилятор добавляет новую функцию (`profile_f1_`) для инициализации счетчиков модуля `F1`. Это касается и файла `f2.ml`:

```

ocamlcp -i -p a -o f2.exe f1.cmo f2.ml
val profile_f2_ : int array
val l1 : int list
val l2 : int list
val l3 : int list

```

Нативный компилятор

Компиляция в машинный код выглядит следующим образом:

```

$ ocamlcpt -i -p -c f1.ml
val interval : int -> int -> int list
exception Found_zero
val mult_list : int list -> int
$ ocamlcpt -i -p -o f2nat.exe f1.cmx f2.ml

```

Здесь используется лишь опция `-p` без аргументов. После выполнения `f2.exe` мы получим файл `gmon.out`. Формат данного файла распознается обычными средствами Unix (см. стр. 9.4.3).

9.4.3 Представление результата

Так как способы анализа программ отличаются в зависимости от вида компиляции, представление результата соответственно тоже отличается. В случае компиляции в байт-код в исходный код программы добавляется число выполнений для структур контроля. При компиляции в машинный код для каждой функции добавляется время затраченное на ее выполнение и число выполнений.

Байт-код компилятор

Команда `ocamlprof` представляет анализ результата измерений профайлера. Для этого используется информация содержащаяся в файле `camlprof.dump`. Данная команда читает приведенный файл и затем, при помощи исходного кода, создает новый исходный код программы, содержащий желаемые данные в виде комментариев.

Для нашего примера получим следующее:

```
$ ocamlprof fl.ml
```

```
let rec interval a b =
  (* 62 *) if b < a then (* 2 *) []
  else (* 60 *) a :: (interval (a+1) b);;

exception Found_zero ;;

let mult_list l =
  (* 2 *) let rec mult_rec l = (* 62 *) match l with
    [] -> (* 1 *) 1
  | 0::_ -> (* 1 *) raise Found_zero
  | n::x -> (* 60 *) n * (mult_rec x)
  in
  try mult_rec l with Found_zero -> (* 1 *) 0
;;
```

Полученные счетчики отражают запрошенные в модуле `F2` вычисления. Мы получили 2 вызова для `list_mult` и 62 для вспомогательной функции `mult_rec`. Анализ ответвлений `match` показывает что образец

по умолчанию выполнялся 60 раз, образец `[]` выполнялся один раз в самом начале и образец с нулем в начале списка так же выполнялся один раз, возбуждая исключение. В строке с `try` указывается значение образца вызвавшее исключение.

У команды `camlprof` имеется две опции. При помощи опции `-f` файл можно указать файл, в котором содержатся измерения. Опцией `-F` строка можно добавить строку в комментарии структурам контроля.

Нативный компилятор

Чтобы получить время, затраченное на вызов функции умножения элементов списка, напомним следующий файл `f3.ml`:

```
let l1 = F1.interval 1 30;;
let l2 = F1.interval 31 60;;
let l3 = l1 @ (0::l2);;

for i=0 to 100000 do
  F1.mult_list l1;
  F1.mult_list l3
done;;

print_int (F1.mult_list l1);;
print_newline();;

print_int (F1.mult_list l3);;
print_newline();;
```

Это такой же файл как и `f3.ml` с 100000 итерациями.

После выполнения программы получим файл `gmon.out`. Формат данного файла распознается командой `gprof`, присутствующей в системе Unix. Вызов этой команды выведет на экран затраченное время и граф вызовов. В связи с тем, что вывод получается достаточно большой, мы покажем лишь начало, где содержится имена функций, которые были вызваны минимум один раз и время затраченное на их исполнение.

```
$ gprof f3nat.exe
Flat profile :
```

Each sample counts as 0.01 seconds.

	%	cumulative	self		self	total	
time	seconds	seconds	calls	us/call	us/call		name
92.31	0.36	0.36	200004	1.80	1.80		F1_mult_rec_45

7.69	0.39	0.03	200004	0.15	1.95	F1_mult_list_43
0.00	0.39	0.00	2690	0.00	0.00	oldify
0.00	0.39	0.00	302	0.00	0.00	darken
0.00	0.39	0.00	188	0.00	0.00	gc_message
0.00	0.39	0.00	174	0.00	0.00	aligned_malloc
0.00	0.39	0.00	173	0.00	0.00	alloc_shr
0.00	0.39	0.00	173	0.00	0.00	fl_allocate
0.00	0.39	0.00	34	0.00	0.00	caml_alloc3
0.00	0.39	0.00	30	0.00	0.00	caml_call_gc
0.00	0.39	0.00	30	0.00	0.00	garbage_collectio

Здесь наглядно видно, что почти все время выполнения затрачено на исполнение функции `F1_mult_rec_45`, которая соответствует функции `F1.mult_rec` из файла `f1.ml`. Мы так же видим, что вызывается много других функций. Первые из них, являются функциями стандартной библиотеки по управлению памятью(8).

9.5 Резюме

В данной главе мы ознакомились с различными вспомогательными инструментами разработки программ, которые входят в дистрибутив Objective CAML.

Первое из этих средств реализует статический анализ и определяет зависимости между множеством элементов компиляции. Эта информация затем интегрируются в файл `Makefile`, что позволяет компилировать только необходимые файлы. То есть, если вы изменили какой-то файл исходник, то необходимо перекомпилировать лишь сам файл и зависящие от него файлы, а не всю программы.

Другие средства предоставляют информацию о выполнении программы. Интерактивная среда предлагает трассировку выполнения, но, как мы видели, полиморфизм устанавливает достаточно серьезные ограничения на исследуемые значения. Нам “видны” лишь мономорфные глобальные объявления, а так же моморфные параметры функции. С помощью этого мы все такие можем отслеживать выполнени рекурсивных функций.

И последний инструмент является традиционным в окружении Unix, *i.e.* отладчик и профайлер. При помощи отладчика можно пошагово выполнять программу, а профайлер выдает информацию о производительности програмы. Оба этих средства могут быть использованны в полной мере лишь в среде Unix. /

Глава 10

Средства лексического и синтаксического анализа

10.1 Введение

Определение и реализация средств лексического и синтаксического анализа являлись важным доменом исследования в информатике. Эта работа привела к созданию генераторов лексического и синтаксического анализа `lex` и `yacc`. Команды `camllex` `camlyacc`, которые мы представим в этой главе, являются их достойными наследниками. Два указанных инструмента стали *de-facto* стандартными, однако существуют другие средства, как например потоки или рациональные (регулярные) выражения из библиотеки `Str`, которые могут быть достаточны для простых случаев, там где не нужен мощный анализ.

Необходимость подобных инструментов особенно чувствовалась в таких доменах, как компиляция языков программирования. Однако и другие программы могут с успехом использовать данные средства: базы данных, позволяющие определять запросы или электронная таблица, где содержимое ячейки можно определить как результат какой-нибудь формулы. Проще говоря, любая программа, в которой взаимодействие с пользователем осуществляется при помощи языка, использует лексический и синтаксический анализ.

Возьмем простой случай. ASCII формат часто используется для хранения данных, будь то конфигурационный системный файл или данные табличного файла. Здесь, для использования данных, необходим лексический и синтаксический анализ.

Обобщая, скажем что лексический и синтаксический анализ преобразует линейный поток символов в данные с более богатой структурой:

последовательность слов, структура записи, абстрактное синтаксическое дерево программы и т.д.

У каждого языка есть словарный состав (лексика) и грамматика, которая описывает каким образом эти составные объединяются (синтаксис). Для того, чтобы машина или программа могли корректно обрабатывать язык, этот язык должен иметь точные лексические и синтаксические правила. У машины нет “тонкого чувства” для того чтобы правильно оценить двусмысленность натуральных языков. По этой причине, к машине необходимо обращаться в соответствии с четкими правилами, в которых нет исключений. В соответствии с этим, понятия лексики и семантики получили формальные определения, которые будут кратко представлены в данной главе.

10.2 План главы

Данная глава знакомит нас со средствами лексического и синтаксического анализа, которые входят в дистрибутив Objective CAML. Обычно, синтаксический анализ следует за лексическим. В первой части мы узнаем о простом инструменте лексического анализа из модуля `Genlex`. После этого ознакомимся с формализмом рациональных выражений и тем самым рассмотрим более детально определение множества лексических единиц. А так же проиллюстрируем их реализацию в модуле `Str` и инструменте `ocamllex`. Во второй части мы определим грамматику и рассмотрим правила создания фраз языка. После этого рассмотрим два анализа фраз: восходящий и нисходящий. Они будут проиллюстрированы использованием `Stream` и `ocamlyacc`. В приведенных примерах используется контекстно-независимая грамматика. Здесь мы узнаем как реализовать контекстный анализ при помощи `Stream`. В третьей части мы вернемся к интерпретатору BASIC (см. стр 171) и при помощи `ocamllex` и `ocamlyacc` добавим лексические и синтаксические функции анализа языка.

10.3 Лексика

Синтаксический анализ это предварительный и необходимый этап обработки последовательностей символов: он разделяет этот поток в последовательность слов, так называемых лексические единицы или лексемы.

10.3.1 Модуль Genlex

В данном модуле имеется элементарное средство для анализа символьного потока. Для этого используются несколько категорий предопределенных лексических единиц. Эти категории различаются по типу:

```
# type token =
  Kwd of string
| Ident of string
| Int of int
| Float of float
| String of string
| Char of char ;;
```

Таким образом мы можем разузнать в потоке символов целое число (конструктор `Int`) и получить его значение (аргумент конструктора `int`). Распознаваемые символы и строки подчиняются следующим общепринятым соглашениям: строка окружена символами ("), а символ окружен ('). Десятичное число представлено либо используя запись с точкой (например 0.01), либо с мантиссой и экспонентой (на пример 1E-2). Кроме этого остались конструкторы `Kwd` и `Ident`.

Конструктор `Ident` предназначен для определения идентификаторов. Идентификатором может быть имя переменной или функции языка программирования. Они состоят из какой угодно последовательности букв и цифр, могут включать символ подчеркивания (`_`) или апостроф (`'`). Данная последовательность не должна начинаться с цифры. Любая последовательность следующих операндов тоже будет считаться идентификатором: `+`, `*`, `>` или `-`. И наконец, конструктор `Kwd` определяет категорию специальных идентификаторов или символов.

Категория ключевых слова — единственная из этого множества, которую можно сконфигурировать. Для того, чтобы создать лексический анализатор, воспользуемся следующей конструкцией, которой необходимо передать список ключевых слов на место первого аргумента.

```
# Genlex.make_lexer ;;
- : string list -> char Stream.t -> Genlex.token Stream.t = <fun>
```

Тем самым получаем функцию, которая принимает на вход поток символов и возвращает поток лексических единиц (с типом `token`).

Таким образом, мы без труда реализуем лексический анализатор для интерпретатора BASIC. Объявим множество ключевых слов:

```
# let keywords =
  [ "REM"; "GOTO"; "LET"; "PRINT"; "INPUT"; "IF"; "THEN";
```

```
"-"; "!="; "+"; "-"; "*"; "/"; "%";
"="; "<"; ">"; "<="; ">="; "<>";
"&"; "|" ] ;;
```

При помощи данного множества, определим функцию лексического анализа:

```
# let line_lexer l = Genlex.make_lexer keywords (Stream.of_string l) ;;
val line_lexer : string -> Genlex.token Stream.t = <fun>
# line_lexer "LET x = x + y * 3" ;;
- : Genlex.token Stream.t = <abstr>
```

Приведенная функция `line_lexer`, из входящего потока символов создает поток соответствующих лексем.

10.3.2 Использование потоков

Мы также можем реализовать лексический анализ “в ручную” используя потоки.

В следующем примере определен лексический анализатор арифметических выражений. Функции `lexer` передается поток символов из которого она создает поток лексических единиц с типом `lexeme Stream.t`¹. Символы пробел, табуляция и переход на новую строку удаляются. Для упрощения, мы не будем обрабатывать переменные и отрицательные целые числа.

```
# let rec spaces s =
  match s with parser
  | [<" " ; rest >] -> spaces rest
  | [<"\t" ; rest >] -> spaces rest
  | [<"\n" ; rest >] -> spaces rest
  | [<>] -> ();;
val spaces : char Stream.t -> unit = <fun>
# let rec lexer s =
  spaces s;
  match s with parser
  | [<"(" >] -> [<'Lsymbol "(" ; lexer s >]
  | [<")" >] -> [<'Lsymbol ")" ; lexer s >]
  | [<"+>] -> [<'Lsymbol "+" ; lexer s >]
  | [<"->] -> [<'Lsymbol "-" ; lexer s >]
  | [<"*" >] -> [<'Lsymbol "*" ; lexer s >]
  | [<"/>] -> [<'Lsymbol "/" ; lexer s >]
```

¹тип `lexeme` определен на стр. 177.

```

| [< '0'..'9' as c;
  i,v = lexint (Char.code c - Char.code('0')) >]
  -> [<'Lint i ; lexer v>]
and lexint r s =
  match s with parser
  [< '0'..'9' as c >]
  -> let u = (Char.code c) - (Char.code '0') in lexint (10*r + u) s
| [<>] -> r,s
;;
val lexer : char Stream.t -> lexeme Stream.t = <fun>
val lexint : int -> char Stream.t -> int * char Stream.t = <fun>

```

Функция `lexint` предназначена для анализа той части потока символов, которая соответствует числовой постоянной. Она вызывается, когда функция `lexer` встречает цифры. В этом случае функция `lexint` поглощает все последовательные цифры и выдает соответствующее значение полученного числа.

10.3.3 Рациональные выражения

Оставим ненадолго практику и рассмотрим проблему лексических единиц с теоретической точки зрения.

Лексическая единица является словом. Слово образуется при конкатенации элементов алфавита. В нашем случае алфавитом является множество символов ASCII.

Теоретически, слово может вообще не содержать символов (пустое слово²) или состоять из одного символа.

Теоретические исследования конкатенации элементов алфавита для образования лексических элементов (лексем) привели к созданию простого формализма, известного как рациональные выражения.

Определение

Рациональные выражения позволяют определить множества слов. Пример такого множества: идентификаторы. Принцип определения основан на некоторых теоретико-множественных операциях. Пусть M и N два множества слов, тогда мы можем определить:

1. объединение M и N , записываемое $M \mid N$.

²традиционно, такое слово обозначается греческой буквой эpsilon: ε

2. дополнение , записываемое \hat{M} : множество всех слов, кроме тех, которые входят в M .
3. конкатенация M и N : множество всех слов созданных конкатенацией слова из M и слова из N . Записывается просто MN .
4. мы можем повторить операцию конкатенации слов множества M и тем самым получить множество слов образованных из конечной последовательности слов множеств M . Такое множество записывается $M+$. Он содержит все слова множества M , все слова полученные конкатенацией двух слов множества M , трех слов, и т.д. Если мы желаем чтобы данное множество содержало пустое слово, необходимо писать $M*$.
5. для удобства, существует дополнительная конструкция $M?$, которая включает все слова множества M , а так же пустое слово.

Один единственный символ ассоциируется с одноэлементным множеством. В таком случае выражение $a | b | c$ описывает множество состоящее из трех слов a , b и c . Существует более компактная запись: $[abc]$. Так как наш алфавит является упорядоченным (по порядку кодов ASCII), можно определить интервал. Например, множество цифр запишется как $[0 - 9]$. Для группировки выражений можно использовать скобки.

Для того, чтобы использовать в записи сами символы-операторы, как обычные символы, необходимо ставить перед ними обратную косую черту: \backslash . Например множество $(\backslash*)^*$ обозначает множество последовательностей звездочек.

Пример

Пусть существует множество из цифр $(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)$, символы плюс $(+)$ и минус $(-)$, точки $(.)$ и буквы E . Теперь мы можем определить множество чисел num . Назовем $integers$ множество определенное выражением $[0 - 9]^+$. Множество неотрицательных чисел $unum$ определяется так:

$$integers?(integers)?(E(\backslash + | -)?integers)?$$

Множество отрицательных и положительных чисел записывается:

$$unum | - unum \text{ или } -?unum$$

.	любой символ, кроме \
*	ноль или несколько экземпляров предыдущего выражения
+	хотя бы один экземпляр предыдущего выражения
?	ноль или один экземпляр предыдущего выражения
[..]	множество символов
	интервал записывается при помощи – (пример [0 – 9])
	дополнение записывается при помощи ^ (пример [^ A – Z])
^	начало строки
	не путать с дополнением ^
\$	конец строки
	вариант
(..)	группировка в одно выражение
	можно ссылаться на это выражение
i	числовая константа i ссылается на i-ый элемент группированного выражения
\	забой, используется для сопоставления зарезервированных символов в рациональных выражениях

Таблица 10.1: Рациональные выражения.

Распознавание

Теперь, после того как множество выражений определено, остается проблема распознавания принадлежности строки символов или одной из ее подстрок этому множеству. Для решения данной задачи необходимо реализовать программу обработки выражений, которая соответствует формальным определениям множества. Для рациональных выражений такая обработка может быть автоматизированна. Подобная автоматизация реализована в модуле **Genlex** из библиотеки **Str** и инструментом **ocamllex**, которые будут представлены в следующих двух параграфах.

10.3.4 Библиотека Str

В данном модуле имеется абстрактный тип **regexp** и функция **regexp**. Указанный тип представляет рациональные выражения, а функция трансформирует рациональное выражение, представленное в виде строки символов, в абстрактное выражение.

Модуль **Str** содержит несколько функций, которые используют рациональные выражения и манипулируют символьными строками. Синтаксис рациональных выражений библиотеки **Str** приведен в таблице 10.1.

Пример

В следующем примере напомним функцию, которая переводит дату из английского формата во французский. Предполагается, что входной файл состоит из строк, разбитых на поля данных и элементы даты разделяются точкой. Определим функцию, которая из полученной строки (линия файла), выделяет дату, разбивает ее на части, переводит во французских формат и тем самым заменяет старую дату на новую.

```
# let french_date_of d =
  match d with
  | mm; dd; yy -> dd ^ "/" ^ mm ^ "/" ^ yy
  | _ -> failwith "Bad date format" ;;
val french_date_of : string list -> string = <fun>

# let english_date_format = Str.regexp "[0-9]+\.[0-9]+\.[0-9]+" ;;
val english_date_format : Str.regexp = <abstr>

# let trans_date l =
  try
    let i = Str.search_forward english_date_format l 0 in
    let d1 = Str.matched_string l in
    let d2 = french_date_of (Str.split (Str.regexp "\.") d1) in
    Str.global_replace english_date_format d2 l
  with Not_found -> l ;;
val trans_date : string -> string = <fun>

# trans_date "      .....06.13.99....." ;;
- : string = "      .....13/06/99....."
```

10.3.5 Инструмент Ocamllex

ocamllex — это лексический генератор созданный для Objective CAML по модели **lex**, написанном на языке C. При помощи файла, описывающего элементы лексики в виде множества рациональных выражений, которые необходимо распознать, он создает файл-исходник на Objective CAML. К описанию каждого лексического элемента можно привязать какое-нибудь действие, называемое семантическое действие. В полученном коде используется абстрактный тип **lexbuf** из модуля **Lexing**. В данном модуле также имеется несколько функций управления лексическими буферами, которые могут быть использованы программистом для того чтобы определить необходимые действия.

Обычно, файлы, описывающие лексику, имеют расширение `.mll`. Для того, чтобы из файла `lex_file.mll` получить файл на Objective CAML, необходимо выполнить следующую команду:

```
ocamllex lex_file.mll
```

После этого, мы получим файл `lex_file.ml`, содержащий код лексического анализатора. Теперь, данный файл можно использовать в программе на Objective CAML. Каждому множеству правил анализа соответствует функция, которая принимает лексический буфер (типа `Lexing.lexbuf`) и затем возвращает значение, определенное семантическим действием. Значит, все действия для определенного правила должны создавать значение одного и того же типа.

Формат у файла для `ocamllex` следующий:

```
{
    header
}
let ident = regexp
    ...
rule ruleset1 = parse
                regexp { action }
                | ...
                | regexp { action }
and ruleset2 = parse
                ...
and ...
{
    trailer —and—end
}
```

Части “заголовок” и “продолжение–и–конец” не являются обязательными. Здесь вставляется код Objective CAML, определяющий типы данных, функции и т.д. необходимые для обработки данных. В последней части используются функции, которые используют правила анализа множества лексических данных из средней части. Серия объявлений, которая предшествует определению правил, позволяет дать имя некоторым рациональным выражениям. Эти имена будут использоваться в определении правил.

Пример

Вернемся к нашему интерпретатору BASIC и усовершенствуем тип возвращаемых лексических единиц. Таким образом, мы можем воспользо-

ваться функцией `lexer`, (см. стр. 177) которая возвращает такой же тип результата (`lexeme`), но на входе она получает буфер типа `Lexing.lexbuf`.

```

{
  let string_chars s =
    String.sub s 1 ((String.length s)-2) ;;
}

let op_ar = ['-','+', '*', '\%', '/']
let op_bool = ['!', '&', '|']
let rel = ['=', '<', '>']

rule lexer = parse
  [' '] { lexer lexbuf }
| op_ar { Lsymbol (Lexing.lexeme lexbuf) }
| op_bool { Lsymbol (Lexing.lexeme lexbuf) }
| "<=" { Lsymbol (Lexing.lexeme lexbuf) }
| ">=" { Lsymbol (Lexing.lexeme lexbuf) }
| "<>" { Lsymbol (Lexing.lexeme lexbuf) }
| rel { Lsymbol (Lexing.lexeme lexbuf) }
| "REM" { Lsymbol (Lexing.lexeme lexbuf) }
| "LET" { Lsymbol (Lexing.lexeme lexbuf) }
| "PRINT" { Lsymbol (Lexing.lexeme lexbuf) }
| "INPUT" { Lsymbol (Lexing.lexeme lexbuf) }
| "IF" { Lsymbol (Lexing.lexeme lexbuf) }
| "THEN" { Lsymbol (Lexing.lexeme lexbuf) }
| '-'? ['0'-'9']+ { Lint (int_of_string (Lexing.lexeme lexbuf)) }
| ['A'-'z']+ { Lident (Lexing.lexeme lexbuf) }
| '"' ['^' '"']* '"' { Lstring (string_chars (Lexing.lexeme lexbuf)) }

```

После обработки данного файла командой `ocamllex` получим функцию `lexer` типа `Lexing.lexbuf -> lexeme`. Далее, мы рассмотрим каким образом подобные функции используются в синтаксическом анализе (см. стр. 341).

10.4 Синтаксис

Благодаря лексическому анализу, мы в состоянии разбить поток символов на более структурированные элементы: лексические элементы. Теперь необходимо знать как правильно объединять эти элементы в синтаксически корректные фразы какого-нибудь языка. Правила синтакси-

ческой группировки определены посредством грамматических правил. Формализм, произошедший из лингвистики, был с успехом перенят математиками, занимающимися теориями языков, и специалистами по информатике. На странице 173 мы уже видели пример грамматики для языка BASIC. Здесь мы снова вернемся к этому примеру, чтобы более углубленно ознакомиться с базовыми концепциями грамматики.

10.4.1 Грамматика

Говоря формальным языком, грамматика основывается на четырех элементах:

1. Множество символов, называемых терминалами. Эти символы являются лексическими элементами языка. В Бэйсике к ним относятся символы операторов, арифметических отношений и логические (+, &, <, <=, ...), ключевые слова языка (**GOTO**, **PRINT**, **IF**, **THEN**, ...), целые числа (элемент *integer*) и переменные (элемент *variable*).
2. Множество не терминальных символов, которые представляют синтаксические компоненты языка. Например, программа на языке Бэйсик состоит из линий. Таким образом мы имеем компоненту *LINE*, которая в свою очередь состоит из выражений (*EXPRESSION*), и т.д.
3. Множество так называемых порождающих правил. Они описывают каким образом комбинируются терминальные и не терминальные символы, чтобы создать синтаксическую компоненту. Строка в Бэйсике начинается с номера, за которой следует инструкция. Смысл правила следующий:

$$\text{LINE} ::= \textit{integer} \text{ INSTRUCTION}$$

Одна и та же компонента может быть порождена несколькими способами. В этом случае варианты разделяются символом |:

$$\begin{array}{lcl} \text{INSTRUCTION} & ::= & \text{LET } \textit{variable} = \text{INSTRUCTION} \\ & & | \quad \text{GOTO } \textit{integer} \\ & & | \quad \text{PRINT } \text{EXPRESSION} \\ & & \text{etc} \end{array}$$

4. Среди всех нетерминальных символов различают один, называемый аксиомой. Правило, которое порождает эту аксиому является порождающим правилом всего языка.

10.4.2 Порождение и распознавание

С помощью порождающих правил можно определить принадлежит ли последовательность лексем языку.

Рассмотрим простой язык, описывающий арифметические выражения:

$$\begin{array}{lll} \text{EXP} & ::= & \text{integer} \quad (R1) \\ & | & \text{EXP} + \text{EXP} \quad (R2) \\ & | & \text{EXP} * \text{EXP} \quad (R3) \\ & | & (\text{EXP}) \quad (R4) \end{array}$$

здесь $(R1)$ $(R2)$ $(R3)$ $(R4)$ имена правил. По окончании лексического анализа выражение $1 * (2 + 3)$ становится последовательностью следующих лексем:

$$\text{integer} * (\text{integer} + \text{integer})$$

Для того, чтобы проанализировать эту фразу и убедиться в том что она принадлежит языку арифметических выражений, воспользуемся правилами справа налево: если часть выражения соответствует правому члену какого-нибудь правила, мы заменяем это выражение соответствующим левым членом. Этот процесс продолжается до тех пор, пока выражение не будет редуцировано до аксиомы. Ниже представлен результат такого анализа³:

$$\begin{array}{ll} \underline{\text{integer}} * (\text{integer} + \text{integer}) & \xleftarrow{(R1)} \text{EXP} * (\underline{\text{integer}} + \text{integer}) \\ & \xleftarrow{(R1)} \text{EXP} * (\text{EXP} + \underline{\text{integer}}) \\ & \xleftarrow{(R1)} \text{EXP} * (\underline{\text{EXP} + \text{EXP}}) \\ & \xleftarrow{(R2)} \text{EXP} * (\underline{\text{EXP}}) \\ & \xleftarrow{(R4)} \underline{\text{EXP} * \text{EXP}} \\ & \xleftarrow{(R3)} \text{EXP} \end{array}$$

Начиная с последней линии, которая содержит лишь EXP и следуя стрелкам, можно определить каким образом было полученное выражение исходя из аксиомы EXP. Соответственно данная фраза является правильно сформированной фразой языка арифметических выражений, определенного грамматикой.

³анализируемая часть выражения подчеркнута, а так же указано используемое правило

Преобразование грамматики в программу, способную распознать принадлежность последовательности лексем языку, который определен грамматикой, является более сложной проблемой, чем проблема использования рациональных приложений. Действительно, в соответствии с математическим результатом любое множество (слов), определенное формализмом рациональных выражений, может быть определено другим формализмом: детерминированные конечные автоматы. Такие автоматы легко реализуются программами, принимающими поток символов. Подобный результат для грамматик в общем не существует. Однако, имеются пробные (?) (weaker) результаты устанавливающие эквивалентность между определенными классами грамматик и более богатыми автоматами: стековый автомат (pushdown automata). Здесь мы не станем вдаваться в детали этих результатов, ни в точное определение таких автоматов. Однако, мы можем определить какие классы грамматики могут использоваться в средствах генерации синтаксических анализаторов или для реализации напрямую анализатора.

10.4.3 Нисходящий анализ

Расчленение выражения $1 * (2 + 3)$ в предыдущем параграфе не является единственным: мы с таким же успехом могли бы начать редуцирование *integer*, то есть воспользоваться правилом (R2) редуцирования $2 + 3$. Эти два способа распознавания являются двумя типами анализа: восходящий анализ (справа налево) и нисходящий слева направо. Последний анализ легко реализуется при помощи потоков лексем модуля **Stream**. Средство **ocaml** использует восходящий анализ, при котором применяется стек, как это уже было проиллюстрировано синтаксическим анализатором программ на Бэйсике. Выбор анализа не просто дело вкуса, в зависимости от используемой для спецификации языка формы грамматики, можно или нет применять нисходящий анализ.

Простой случай

Каноническим примером нисходящего анализа является префиксная запись арифметических выражений, определяемая как:

$$\begin{array}{lcl} \text{EXP} & ::= & \text{integer} \\ & | & + \text{EXP EXP} \\ & | & * \text{EXP EXP} \end{array}$$

В данном случае достаточно знать первую лексему, для того чтобы определить какое правило может быть использовано. При помощи по-

добной предсказуемости нет необходимости явно управлять стеком, достаточно положиться на рекурсивный вызов анализатора. И тогда при помощи **Genlex** и **Stream** очень просто написать программу реализующую нисходящий анализ. Функция **infix_of** из полученного префиксного выражения возвращает его инфиксный эквивалент:

```
# let lexer s =
  let ll = Genlex.make_lexer ["+";"*"]
  in ll (Stream.of_string s) ;;
val lexer : string -> Genlex.token Stream.t = <fun>
# let rec stream_parse s =
  match s with parser
  | [<'Genlex.Ident x>] -> x
  | [<'Genlex.Int n>] -> string_of_int n
  | [<'Genlex.Kwd "+"; e1=stream_parse; e2=stream_parse>] -> "(" ^
    e1 ^ "+" ^ e2 ^ ")"
  | [<'Genlex.Kwd "*"; e1=stream_parse; e2=stream_parse>] -> "(" ^
    e1 ^ "*" ^ e2 ^ ")"
  | [<>] -> failwith "Parse error"
;;
val stream_parse : Genlex.token Stream.t -> string = <fun>
# let infix_of s = stream_parse (lexer s) ;;
val infix_of : string -> string = <fun>
# infix_of "*+3 11 22";;
- : string = "((3+11)*22)"
```

Однако не стоит забывать о некоторой примитивности лексического анализа. Советуем периодически добавлять пробелы между различными лексическими элементами.

```
# infix_of "*+3 11 22";;
- : string = "*+ "
```

Случай посложней

Синтаксический анализ при помощи потоков предсказуем, он обладает грамматикой двумя условиями:

1. В правилах грамматики не должно быть левой рекурсии. Правило называется рекурсивным слева, если его правый член начинается с нетерминального символа, который является левой частью правила. Например: $EXP ::= EXP + EXP$

2. Не должно существовать правил начинающихся одним и тем же выражением.

Грамматика арифметических выражений, приведенная на стр. 330, не подходит для нисходящего анализа: они не удовлетворяют ни одному из условий. Для того, чтобы применить нисходящий анализ необходимо переформулировать грамматику таким образом, чтобы удалить левую рекурсию и неопределенность правил. Вот полученный результат:

EXPR	::=	АТОМ	NEXTEXPR
NEXTEXPR	::=	+	АТОМ
		+	АТОМ
		-	АТОМ
		*	АТОМ
		/	АТОМ
		ε	
АТОМ	::=	<i>integer</i>	
		(EXPR)	

Заметьте использование пустого слова ε в определении NEXTEXPR. Оно необходимо, если мы хотим чтобы просто целое число являлось выражением.

Следующий анализатор есть просто перевод вышеуказанной грамматики в код. Он реализует абстрактное синтаксическое дерево арифметических выражений.

```
# let rec rest = parser
  | [< 'Lsymbol "+" ; e2 = atom >] -> Some (PLUS,e2)
  | [< 'Lsymbol "-" ; e2 = atom >] -> Some (MINUS,e2)
  | [< 'Lsymbol "*" ; e2 = atom >] -> Some (MULT,e2)
  | [< 'Lsymbol "/" ; e2 = atom >] -> Some (DIV,e2)
  | [< >] -> None
and atom = parser
  | [< 'Lint i >] -> ExpInt i
  | [< 'Lsymbol "(" ; e = expr ; 'Lsymbol ")" >] -> e
and expr s =
  match s with parser
  | [< e1 = atom >] ->
    match rest s with
    | None -> e1
    | Some (op,e2) -> ExpBin(e1,op,e2) ;;
val rest : lexeme Stream.t -> (bin_op * expression) option = <fun>
```

```

val atom : lexeme Stream.t -> expression = <fun>
val expr : lexeme Stream.t -> expression = <fun>

```

Сложность использования нисходящего анализа заключается в том, что грамматика должна быть очень ограниченной формы. Если язык выражен естественно с использованием левой рекурсии (как в инфиксных выражениях), то не всегда легко определить эквивалентную грамматику, то есть определяющую такой же язык, которая бы удовлетворяла требованиям нисходящего анализа. По этой причине, средства `uacc` и `osamluacc` реализуют восходящий анализ, который разрешает определение более естественных грамматик. Однако, мы увидим, что даже в этом случае существуют ограничения.

10.4.4 Восходящий анализ

Мы уже вкратце представили на странице 179 принципы восходящего анализа: продвинутся (`shift`) и сократить (`reduce`). После каждого подобного действия, состояние стека изменяется. Из этой последовательности можно вывести правила грамматики, в случае если грамматика это позволяет, как в примере с нисходящим анализом. Опять же, сложности возникают из-за неопределенности правил, когда невозможно выбрать между продвинутся или сократить. Проиллюстрируем действие восходящего анализа и его недостатки на все тех же арифметических выражениях в постфиксном и инфиксном написании.

Положительная сторона Упрощенная постфиксная грамматика арифметических выражений выглядит так:

$$\begin{array}{ll}
 \text{EXP} ::= & \text{integer} \quad (R1) \\
 & | \quad \text{EXP EXP} + \quad (R2) \\
 & | \quad \text{EXP EXP} * \quad (R3)
 \end{array}$$

Данная грамматика является двойственной по отношению к префиксной: для того чтобы точно знать какое правило следует применить, необходимо дождаться окончания анализа. В действительности анализ подобных выражений схож с вычислением при помощи стека. Только вместо проталкивания результата вычисления, проталкиваются грамматические символы. Если в начале стек пустой, то после того как ввод закончен, необходимо получить стек содержащий лишь нетерминальную аксиому. Приведем изменение стека: если мы продвигаемся, то проталкивается текущий нетерминальный символ; если сокращаем, то первые

Действие	Вход	Стек
	<u>1</u> 2 + 3 * 4 + \$	[]
Сдвиг		
	2 + <u>3</u> * 4 + \$	[1]
Сократить (<i>R1</i>)		
	2 + 3 * <u>4</u> + \$	[EXP]
Сдвиг		
	2 + 3 * 4 + <u>\$</u>	[2 EXP]
Сократить (<i>R1</i>)		
	2 + <u>3</u> * 4 + \$	[EXP EXP]
Сдвиг, Сократить (<i>R2</i>)		
	3 * 4 + \$	[EXP]
Сдвиг, Сократить (<i>R1</i>)		
	*4 + \$	[EXP EXP]
Сдвиг, Сократить (<i>R3</i>)		
	4 + \$	[EXP]
Сдвиг, Сократить (<i>R1</i>)		
	+ \$	[EXP EXP]
Сдвиг, Сократить (<i>R2</i>)		
	\$	[EXP]

Таблица 10.2: Восходящий анализ.

символы стека соответствуют правому члену (в обратном порядке) правила и тогда мы заменяем эти элементы соответствующими нетерминальными элементами.

В таблице 10.2 приведен восходящий анализ выражения $1\ 2 + 3 * 4 +$. Считываемая лексическая единица подчеркивается для более удобного чтения. Конец потока помечается символом $\$$.

Отрицательная сторона Вся трудность перехода от грамматики к программе распознающей язык заключается в определении действия, которое необходимо применить. Проиллюстрируем эту проблему на трех примерах, приводящих к трем неопределенностям.

Первый пример есть грамматика выражений использующих операцию сложения:

$$\begin{array}{ll}
 E0 & ::= \textit{integer} \quad (R1) \\
 & | \quad E0 + E0 \quad (R2)
 \end{array}$$

Неопределенность данной грамматики проявляется при использовании правила R2. Предположим следующую ситуацию:

Действие	Вход	Стек
:		
	<u>+</u>	E0 + E0...
:		

В подобном случае невозможно определить необходимо сдвинуть и протолкнуть в стек + или сократить в соответствии с правилом (R2) оба E0 и присутствующий в стеке +. Подобная ситуация называется конфликтом сдвиг–сокращение (shift/reduce). Она является следствием того, что выражение *integer + integer + integer* может быть получено деривацией справа двумя способами.

Первый вариант:

$$\begin{array}{lcl}
 E0 & \xrightarrow{(R2)} & E0 + \underline{E0} \\
 & \xrightarrow{(R1)} & \underline{E0} + integer \\
 & \xrightarrow{(R2)} & E0 + \underline{E0} + integer
 \end{array}$$

Второй вариант:

$$\begin{array}{lcl}
 E0 & \xrightarrow{(R2)} & E0 + \underline{E0} \\
 & \xrightarrow{(R2)} & E0 + E0 + \underline{E0} \\
 & \xrightarrow{(R1)} & E0 + \underline{E0} + integer
 \end{array}$$

Выражения, полученные двумя деривациями, могут показаться одинаковыми с точки зрения вычисления выражения,

$$(integer + integer) + integer \text{ и } integer + (integer + integer)$$

но разными для конструкции синтаксического дерева (см. рис. 5.2 на стр. 179)

Второй пример грамматики, порождающей конфликт между сдвиг–сокращение, содержит такую же неопределенность: явное заключение в скобки. Но в отличие от предыдущего случая, выбор сдвиг–сокращение изменяет смысл выражения. Пусть есть грамматика:

$$\begin{array}{lcl}
 E1 & ::= & integer \quad (R1) \\
 & | & E1 + E1 \quad (R2) \\
 & | & E1 * E1 \quad (R3)
 \end{array}$$

Здесь мы снова получаем предыдущий конфликт как в случае с $+$ так и для $*$, но к этому добавляется другой, между $+$ и $*$. Опять же, одно и то же выражение может быть получено двумя способами, так как у него существует две деривации справа:

$integer + integer * integer$

Первый вариант:

$$\begin{array}{lcl} E1 & \xrightarrow{(R3)} & E1 * \underline{E1} \\ & \xrightarrow{(R1)} & \underline{E1} * integer \\ & \xrightarrow{(R2)} & E1 + \underline{E1} * integer \end{array}$$

Второй вариант:

$$\begin{array}{lcl} E1 & \xrightarrow{(R2)} & E1 + \underline{E1} \\ & \xrightarrow{(R2)} & E1 + E1 * \underline{E1} \\ & \xrightarrow{(R1)} & E1 + \underline{E1} * integer \end{array}$$

В данном выражении обе пары скобок имеют разный смысл:

$(integer + integer) * integer \neq integer + (integer * integer)$

Подобную проблему, мы уже встречали в выражениях Basic (см. стр. 179). Она была разрешена при помощи приоритетов, которые присваиваются операторам: сначала редуцируется правило (R3), затем (R2), что соответствует заключению в скобки произведения.

Данную проблему выбора между $+$ и $*$ можно решить изменив грамматику. Для того, введем два новых терминальных символа: член T (*term*) и множитель F (*factor*). Отсюда получаем:

$$\begin{array}{lll} E & ::= & E + T \quad (R1) \\ & | & T \quad (R2) \\ T & ::= & T * F \quad (R3) \\ & | & F \quad (R4) \\ F & ::= & integer \quad (R5) \end{array}$$

После этого, единственный способ получить $integer + integer * integer$: посредством правила (R1).

Третий и последний случай касается условных конструкций языка программирования. На пример в Pascal существует две конструкции: `if .. then` и `if .. then .. else`. Пусть существует следующая грамматика:

Действие	Вход	Стек
:	<u>else</u> ...	[INSTR then EXP if...]
:		

INSTR ::= if EXP then INSTR (R1)
 — if EXP then INSTR else INSTR (R2)
 — etc...

И в следующей ситуации:

Невозможно определить соответствуют ли элементы стека правила (R1) и в этом случае необходимо сократить или соответствуют первому INSTR правила (R2) и тогда необходимо сдвинуть.

Кроме конфликтов сдвиг–сокращение, восходящий анализ вызывает конфликт сокращение–сокращение.

Мы представим здесь инструмент `osamlyacc`, который использует подобную технику может встретить указанные конфликты.

10.4.5 Инструмент `osamlyacc`

Принцип действия `osamlyacc` схож с `osamlex`: из входного файла, содержащего грамматику, в которой каждое правило связано с семантическим действием, генерируется два файла Objective CAML с расширением `.ml` и `.mli`. Эти файлы содержат функции и интерфейс синтаксического анализа.

Общий формат Файл для `osamlyacc` с синтаксическим описанием заканчивается расширением `.mly` и имеет следующий формат:

```
%{
    заголовок
}%
    декларации
%%
    правила
%%
    продолжение–и–конец
```

Формат правил следующий:

```

нетерминальный символ : символ...символ {семантическое действие}
                        | ...
нетерминальный символ : символ...символ {семантическое действие}
                        ;

```

Символ может быть либо терминальным либо нет. Роль частей “заголовков” и “продолжение-и-конец” такая же как и у `osamllex`, с разницей в том, что заголовок не виден лишь правилам, но не декларациям. В частности, это означает что загрузка модулей (**open**) не учитывается в декларациях и следовательно типы должны быть известны.

Семантическое действие Семантическое действие это куски кода на Objective CAML, которые выполняются при редуцировании связанного с ним правила. В теле действия можно ссылаться на компоненты правого члена правила. Они нумеруются справа налево начиная с 1. Для того, чтобы ссылаться на первый компонент нужно использовать `$1`, на второй `$2` и так далее.

Аксиомы Нетерминальные аксиомы грамматики объявляются в декларативной части входного файла:

```
\%start non-terminal .. non-terminal
```

Для каждой из них будет создана своя функция анализа. В декларативной части всегда необходимо указывать возвращаемый тип функции.

```
\%type <output-type> non-terminal
```

Тип `output-type` должен быть известен.

Warning

Нетерминальные символы становятся именем функций анализа, поэтому они не должны начинаться с заглавной буквы, которая зарезервирована для конструкторов.

Лексические единицы Грамматические правила ссылаются на лексические единицы, которые являются терминальными символами правил.

Лексема(ы) объявляется следующим способом:

```
\%token PLUS MINUS MULT DIV MOD
```

Некоторые лексические единицы, например идентификаторы, являются множеством символьных строк. Когда обнаруживается подобный идентификатор, мы бы хотели получить эту символьную строку. Для этого необходимо указать лексическому анализатору, что эти лексемы имеют привязанное значение, окружив тип значения `<` и `>`.

```
\%token <string> IDENT
```

Warning

После того, как данные объявления обработаны `osamlyacc`, они превращаются в конструкторы типа `token`. Они должны обязательно начинаться с заглавной буквы.

У нас есть возможность использовать строки символов как неявные терминальные символы.

```
expr : expr "+" expr { ... }
      | expr "*" expr { ... }
      | ...
      ;
```

И в данном случае нет надобности объявлять необходимый символ: они напрямую обрабатываются синтаксическим анализатором, без прохода через лексический. Однако, мы не советуем подобный метод, из соображений единообразия объявлений.

Приоритет, ассоциативность Мы уже видели, что немало конфликтов, возникающих при нисходящем анализе, появляются из неявных правил ассоциативности какого-нибудь оператора или из-за конфликта приоритетов между операторами. Для разрешения подобных конфликтов, можно объявить правила ассоциативности по умолчанию (слева направо, справа налево или никакое) для операторов, а так же приоритет. В приведенном ниже объявлении, указано, что операторы `+` (лексема `PLUS`) и `*` (`MULT`) ассоциативные справа и что у оператора `*` приоритет выше, чем у `+`, потому что `MULT` объявлен после `PLUS`.

```
%left PLUS
```

```
%left MULT
```

Операторы объявленные на одной линии, имеют одинаковые приоритеты.

Опции командной строки Команда `osamlyacc` распознает две следующие опции:

- **-b name:** будут созданы два файла `name.ml` `name.mli`
- **-v:** будет создан файл с расширением `.output`, который будет содержать нумерацию правил, состояние автомата, распознающего грамматику и конфликтные ситуации

Совместное использование с `ocamllex`

Можно связать оба инструмента `ocamllex` и `ocamlyacc` таким образом, чтобы поток символов, превращенный в поток лексем стал входными данными для синтаксического анализатора. Для этого, тип `lexeme` должен быть известен обоим инструментам. Этот тип определен в файлах с расширением `.mli` и `.ml`, созданных `ocamlyacc` при помощи деклараций токенов из соответствующего файла с расширением `.mly`. Файл `.mli` вводит (*import*) указанный тип, `ocamllex` переводит данный файл в функцию Objective CAML с типом `Lexing.lexbuf -> lexeme`. В примере на стр. 342 иллюстрируется указанное взаимодействие и описываются различные этапы компиляции.

10.4.6 Контекстная грамматика

Анализаторы, генерируемые `ocamlyacc`, обрабатывают языки, порождаемые контекстно-независимой грамматикой. Анализ потока лексем не зависит от синтаксических значений, которые уже были проанализированы. Данное утверждение неверно для языка L , представленного следующей формулой:

$$L ::= wCw \mid w, w \in (A \mid B)^*$$

здесь A, B и C терминальные символы. Мы записали wCw (где $w \in (A \mid B)^*$), а не просто $(A \mid B)^*C(A \mid B)^*$, для того чтобы получить *такую же* слово слева и справа от C .

Для анализа слова из языка L , необходимо хранить в памяти, то что было до символа C , чтобы убедиться, что после следует тоже самое. Далее представим решение данной проблемы. Идея алгоритма состоит в создании функции, которая анализирует поток и точно распознает часть слова, которое находится до возможного нахождения C .

Создадим тип:

```
# type token = A | B | C ;;
```

Функция `parse_w1` создает функцию запоминающую первое w в виде списка функций анализирующих неделимый поток (то есть для одного токена):

```
# let rec parse_w1 s =
  match s with parser
    | [<'A; l = parse_w1 >] -> (parser [<'A >] -> "a")::l
    | [<'B; l = parse_w1 >] -> (parser [<'B >] -> "b")::l
    | [< >] -> [] ;;
val parse_w1 : token Stream.t -> (token Stream.t -> string) list = <
  fun>
```

Результатом функций, созданных `parse_w1`, является строка символов, содержащая проанализированную синтаксическую единицу.

Из списка, созданного предыдущей функцией, функция `parse_w2` формирует в одну функцию анализа:

```
# let rec parse_w1 s =
  match s with parser
    | [<'A; l = parse_w1 >] -> (parser [<'A >] -> "a")::l
    | [<'B; l = parse_w1 >] -> (parser [<'B >] -> "b")::l
    | [< >] -> [] ;;
val parse_w1 : token Stream.t -> (token Stream.t -> string) list = <
  fun>
```

Результатом применения `parse_w2` будет подслово w . По своей конструкции, функция `parse_w2` распознает лишь подслова пройденные функцией `parse_w1`.

Воспользуемся возможностью называть промежуточные результаты потока и напомним функцию распознавания слов языка L :

```
# let parse_L = parser [< l = parse_w1 ; 'C; r = (parse_w2 l) >] -> r
;;
val parse_L : token Stream.t -> string = <fun>
```

Ниже представлены два примера, в первом мы получаем слово вокруг C , а во втором получаем ошибку, так как слова вокруг C разные.

```
# parse_L [< 'A; 'B; 'B; 'C; 'A; 'B; 'B >];;
- : string = "abb"
# parse_L [< 'A; 'B; 'C; 'B; 'A >];;
Uncaught exception: Stream.Error("")
```

10.5 Пересмотренный Basic

Теперь, используя совместно `osamllex` и `osamlyacc`, заменим функцию `parse` для Бэйсика, приведенную на странице 183, на функции полученные при помощи файлов спецификации лексики и синтаксиса языка.

Для этого, мы не сможем воспользоваться типами лексических единиц, в таком виде как они были определены. Необходимо определить более точные типы, чтобы различать операторы, команды и ключевые слова.

Так же, нам понадобится изолировать в отдельном файле `basic_types.mli` декларации типов, относящиеся к абстрактному синтаксису. В нем будут содержаться декларации типа `sentences`, а так же других типы необходимые этому.

10.5.1 Файл `basic_parser.mly`

Заголовок Данный файл содержит вызовы деклараций типов абстрактного синтаксиса и две функции перевода строк символов в их эквивалент абстрактного синтаксиса.

```
%{
open Basic_types ;;

let phrase_of_cmd c =
  match c with
  | "RUN" -> Run
  | "LIST" -> List
  | "END" -> End
  | _ -> failwith "line : unexpected command"
;;

let bin_op_of_rel r =
  match r with
  | "=" -> EQUAL
  | "<" -> INF
  | "<=" -> INFEQ
  | ">" -> SUP
  | ">=" -> SUPEQ
  | "<>" -> DIFF
  | _ -> failwith "line : unexpected relation symbol"
;;

%}
```

Декларации Здесь содержится три части: декларация лексем, декларация правил ассоциативности и приоритетов, декларация аксиомы `line`,

которая соответствует анализу линии программы или команды.

Ниже представлены лексические единицы:

```
%token <int> Lint
%token <string> Lident
%token <string> Lstring
%token <string> Lcmd
%token Lplus Lminus Lmult Ldiv Lmod
%token <string> Lrel
%token Land Lor Lneg
%token Lpar Rpar
%token <string> Lrem
%token Lrem Llet Lprint Linput Lif Lthen Lgoto
%token Lequal
%token Leol
```

Имена деклараций говорят сами за себя и они описаны в файле `basic_lexer.mll` (см. стр. 346).

Правила приоритета операторов схожи со значениями, которые определяются функциями `priority_uop` и `priority_binop`, которые были определены грамматикой Бейсика (см. стр. 173).

```
%right Lneg
%left Land Lor
%left Lequal Lrel
%left Lmod
%left Lplus Lminus
%left Lmult Ldiv
%nonassoc Lop
```

Символ `Lop` необходим для обработки унарных минусов. Он не является терминальным символом, а “псевдо-терминальным”. Благодаря этому, получаем перегрузку операторов, когда в двух случаях использования одного и того же оператора, приоритет меняется в зависимости от контекста. Мы вернемся к этому случаю, когда будем рассматривать правила грамматики.

Здесь нетерминальной аксиомой является `line`. Полученная функция возвращает дерево абстрактного синтаксиса, которое соответствует проанализированной линии.

```
%start line
%type <Basic_types.phrase> line
```


Правила грамматики Грамматика делится на 3 нетерминальных элемента: **line** для линии, **inst** для инструкции и **exp** для выражений. Действия, которые привязаны к каждому правилу лишь создают соответствующую часть абстрактного синтаксиса.

```
%%
line :
    Lint inst Leol                { Line {num=$1; inst=$2} }
  | Lcmd Leol                    { phrase_of_cmd $1 }
  ;

inst :
    Lrem                        { Rem $1 }
  | Lgoto Lint                  { Goto $2 }
  | Lprint exp                  { Print $2 }
  | Linput Lident               { Input $2 }
  | Lif exp Lthen Lint          { If ($2, $4) }
  | Llet Lident Lequal exp      { Let ($2, $4) }
  ;

exp :
    Lint                        { ExpInt $1 }
  | Lident                      { ExpVar $1 }
  | Lstring                     { ExpStr $1 }
  | Lneg exp                    { ExpUnr (NOT, $2) }
  | exp Lplus exp               { ExpBin ($1, PLUS, $3) }
  | exp Lminus exp              { ExpBin ($1, MINUS, $3) }
  | exp Lmult exp               { ExpBin ($1, MULT, $3) }
  | exp Ldiv exp                { ExpBin ($1, DIV, $3) }
  | exp Lmod exp                { ExpBin ($1, MOD, $3) }
  | exp Lequal exp              { ExpBin ($1, EQUAL, $3) }
  | exp Lrel exp                { ExpBin ($1, (bin_op_of_rel $2), $3) }
  | exp Land exp                { ExpBin ($1, AND, $3) }
  | exp Lor exp                 { ExpBin ($1, OR, $3) }
  | Lminus exp %prec Lop        { ExpUnr(OPPOSITE, $2) }
  | Lpar exp Rpar               { $2 }
  ;
%%
```

Данные правила не нуждаются в особых комментариях, кроме следующего:

exp :

```
...
| Lminus exp %prec Lor { ExpUnr(OPPOSITE, $2) }
```

Это правило касается использования унарного минуса -. Ключевое слово `%prec` означает, что указанная конструкция получает приоритет от `Lor` (в данном случае наивысший).

10.5.2 Файл `basic_lexer.mll`

Лексический анализ содержит лишь одно множество: `lexer`, которое точно соответствует старой функции `lexer` (см. стр. 179).

Семантическое действие, которое связано с распознаванием лексических единиц, возвращает результат соответствующего конструктора. Необходимо загрузить файл синтаксических правил, так как в нем декларируется тип лексических единиц. Добавим так же функцию, которая удаляет кавычки вокруг строк.

```
{
  open Basic_parser ;;

  let string_chars s = String.sub s 1 ((String.length s)-2) ;;
}

rule lexer = parse
  [' ' '\t']          { lexer lexbuf }

  | '\n'              { Leol }

  | '!'               { Lneg }
  | '&'               { Land }
  | '|'               { Lor }
  | '='               { Lequal }
  | '%'               { Lmod }
  | '+'               { Lplus }
  | '-'               { Lminus }
  | '*'               { Lmult }
  | '/'               { Ldiv }

  | ['<' '>']         { Lrel (Lexing.lexeme lexbuf) }
  | "<="               { Lrel (Lexing.lexeme lexbuf) }
  | ">="               { Lrel (Lexing.lexeme lexbuf) }
```

```

| "REM" [^ '\n']*      { Lrem (Lexing.lexeme lexbuf) }
| "LET"                { Llet  }
| "PRINT"              { Lprint }
| "INPUT"              { Linput }
| "IF"                 { Lif  }
| "THEN"               { Lthen }
| "GOTO"               { Lgoto }

| "RUN"                { Lcmd (Lexing.lexeme lexbuf) }
| "LIST"               { Lcmd (Lexing.lexeme lexbuf) }
| "END"                { Lcmd (Lexing.lexeme lexbuf) }

| ['0'-'9']+           { Lint (int_of_string (Lexing.lexeme lexbuf)) }
| ['A'-'Z']+           { Lident (Lexing.lexeme lexbuf) }
| '"' [^ '"'']* '"'    { Lstring (string_chars (Lexing.lexeme lexbuf)) }

```

Заметьте, что нам пришлось изолировать символ `=`, который используется одновременно в выражениях и приравниваниях.

Для двух рациональных выражений необходимо привести определенные объяснения. Линия комментариев соответствует выражению `("REM" [^ '\n']*)`, где за ключевым словом `REM` следует какое угодно количество символов и затем перевод строки. Правило, которое соответствует символьным строкам, `('\" [^ '\"']* '\')`, подразумевает последовательность символов, отличных от `"` и заключенных в кавычки `"`.

10.5.3 Компиляция, компоновка

Компиляция должна быть реализована в определенном порядке. Это связано с взаимозависимостью деклараций лексем. Поэтому в нашем случае, необходимо выполнить команды в следующем порядке:

```

ocamlc -c basic_types.mli
ocamlyacc basic_parser.mly
ocamllex basic_lexer.mll
ocamlc -c basic_parser.mli
ocamlc -c basic_lexer.ml
ocamlc -c basic_parser.ml

```

После чего получим файлы `basic_lexer.cmo` и `basic_parser.cmo`, которые можно использовать в нашей программе.

Теперь, у нас есть весь необходимый арсенал, для того чтобы переделать программу.

Удалим все типы и функции параграфов “лексический анализ” (стр. 177) и “синтаксический анализ” (стр. 179) для программы Бэйсик. Также в функции `one_command` заменим выражение:

```
match parse (input_line stdin) with
```

```
на
```

```
match line lexer (Lexing.from_string ((input_line stdin) ^ "\n")) with
```

Заметьте, что необходимо поместить в конце линии символ конца `'n'`, который был удален функцией `input_line`. Это необходимо, потому что данный символ используется для указания конца командной линии (`Leol`).

10.6 Резюме

В данной главе были описаны различные средства лексического и синтаксического анализа Objective CAML. По порядку описания:

- модуль `Str` для фильтрации рациональных выражений
- модуль `Genlex` для быстрого создания простых лексических анализаторов
- `ocamllex` представитель семейства `lex`
- `ocamlyacc` представитель семейства `yacc`
- потоки, для построения нисходящих анализаторов, в том числе и контекстных

При помощи инструментов `ocamllex` и `ocamlyacc` мы переделали синтаксический анализ Бэйсика, который проще поддерживать, чем анализатор представленный на стр. 171.

Глава 11

Взаимодействие с языком С

11.1 Введение

Часто, при разработке программного обеспечения на каком-нибудь языке, бывает необходимо интегрировать библиотеки, которые были написанные на других языках программирования по следующим двум причинам:

- использовать библиотеки, которые не возможно написать на данном языке, то есть таким образом расширить возможности языка
- использовать высокоэффективные библиотеки, написанные на другом языке

В данном случае получаем программу, которая собрана из кусков на различных языках и каждый из этих специализированных кусков написан на наиболее подходящем для решения данной проблемы языке. Такие части программы взаимодействуют между собой, передавая данные и выполняя расчет.

В Objective CAML существует механизм взаимодействия с языком С. Из кода Objective CAML можно вызвать функцию на С, передать аргументы и получить результат расчета. Обратное так же возможно: программа на С может вызвать расчет на Objective CAML и затем продолжить обрабатывать полученный результат.

Выбор языка С оправдан следующими причинами:

- С нормализованный язык (C ANSI)
- он используется для написания операционных систем (Unix, Windows, MacOS, и т.д.)

- большое количество библиотек написано на C
- большинство языков программирования имеют механизм взаимодействия с C, что позволяет установить взаимосвязь с этими языками посредством C.

В последнем случае C выступает в роли эсперанто языков программирования.

Однако, взаимодействие между Objective CAML и C создает определенные трудности, которые представлены ниже:

представление значений в памяти Например основные значения (*int*, *char*, *float*) в обоих языках различаются представлением в памяти. Необходимо переводить их при обмене данными в обе стороны. То же самое касается структурированных значений: записи, тип сумма или массивы.

сборщик мусора Objective CAML Реализовать сборщик мусора в C можно, но эта особенность не определена нормой языка. Поэтому, нельзя допустить, чтобы вызов функции на C произвел несовместимые с GC изменения в памяти.

остановка расчета Механизм исключений в обоих языках различается, в следствии чего возникает проблема перехвата исключений.

разделение общих ресурсов Буферы Ввода/Вывода, например, не разделяются и не отображают последовательность операций Ввода/-Вывода двух программ.

Программы на Objective CAML наследуют надежность статической типизации и автоматическое управление памятью. Корректное использование библиотек на C или взаимодействие посредством C не должно подвергнуть риску эту надежность. Поэтому, для гармоничного взаимодействия обоих языков, необходимо следовать четким правилам.

11.2 План главы

В данной главе, мы рассмотрим средства, входящие в дистрибутив Objective CAML, для взаимодействия с C, которые позволят создать исполняемые файлы, состоящих из частей на обоих языках программирования. Эти средства предоставляют функции, при помощи которых можно перевести значение из одного языка в другой, надежно выделить память в

C, используя кучу Objective CAML и GC, а так же возбудить исключения Objective CAML в *C*.

В первой части главы мы рассмотрим использование *C* функции в Objective CAML, как создать исполнимый файл и интерактивную среду, которые содержат эту функцию. Во второй части мы изучим каким образом значения Objective CAML представлены в *C*. В следующей части уточняется как создавать и менять значения Objective CAML в *C*. В ней так же рассматриваются проблемы, возникающие при выделении памяти в *C* в “присутствии” GC Objective CAML, а так же способы надежного выделения памяти в *C*. В пятой главе описано управление исключениями, как они возбуждаются и отлавливаются в зависимости от места остановки расчета. В последней части мы рассмотрим как использовать код Objective CAML в *C*.

Замечание

Для изучения следующего материала, необходимы знания языка *C*. Так же, желательно прочесть главу 8, для четкого понимания проблем связанных с автоматической сборкой памяти.

11.3 Передача информации между Objective CAML и *C*

Передача информации между Objective CAML и *C* осуществляется созданием исполняемого файла (или интерактивной среды), состоящей из двух частей, которые могут быть скомпилированы отдельно. И тогда, до создания исполнимого файла, компоновщик должен установить связь между именами Objective CAML и *C*. Для этого программа на Objective CAML должна содержать внешние декларации.

На рисунке 11.1 изображена программа, состоящая из одного куска на *C* и другого на Objective CAML.

Можно представить каждую часть, как содержащую код, инструкции, соответствующие определению функций и вычислению выражений Objective CAML, а так же зоны динамического выделения памяти. Применение функции f к трем целым числам Objective CAML провоцирует вычисление функции f_c на *C*. Тело функции переводит целые числа Objective CAML в целые числа *C*, вычисляет сумму и затем возвращает результат переведенный в целое число Objective CAML.

Далее будут представлены первичные элементы взаимодействия Об-

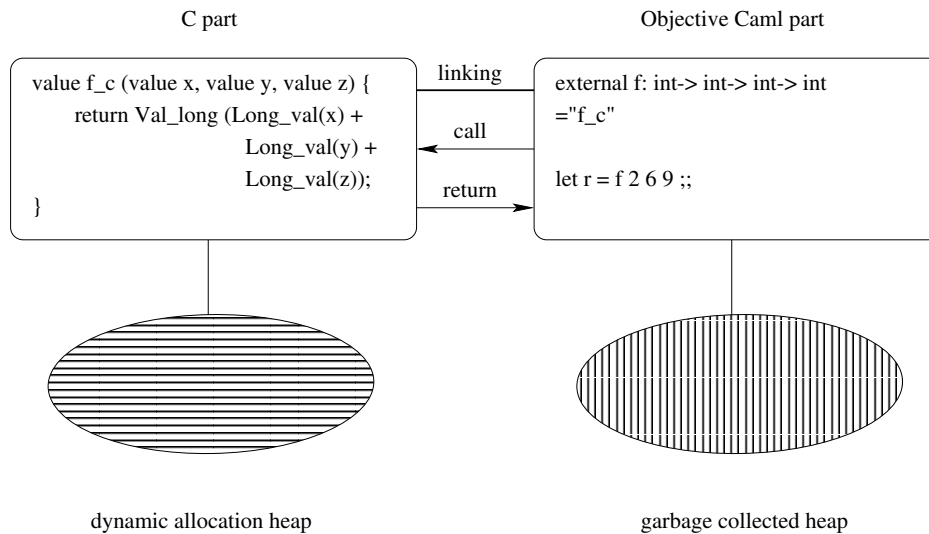


Рис. 11.1: Передача информации между Objective CAML и C

jective CAML и C: внешние декларации, ограничения накладываемые на C функции, которые можно вызвать в Objective CAML и опции редактора связей. Затем мы приведем пример использования ввода/вывода.

11.3.1 Внешние декларации

Внешние декларации функций в Objective CAML необходимы для установления связи между декларацией функции C и именем в Objective CAML, а так же для указания типа функции.

Синтаксис декларации следующий:

Синтаксис `external caml_name : type = "C_name"`

Эта запись означает, что вызов функции `caml_name` в Objective CAML спровоцирует вызов функции C `C_name` с соответствующими аргументами. В примере на изображении 11.1 декларируется функция `f`, вызов которой соответствует выполнению функции `f_c`.

Можно объявить функцию, как внешнюю в интерфейсе (т.е. в файле `.mli`), либо явно указывая на то, что она внешняя, либо как обычное значение:

Синтаксис `external caml_name : type = "C_name" val caml_name : type`

Во втором случае, вызов C функции проходит через общий механизм функций Objective CAML. Подобный подход менее эффективен, но он скрывает реализацию функции как C функции.

11.3.2 Декларация C функций

Число аргументов C функций, вызываемых в Objective CAML, должно быть одинаковым с внешним объявлением функции. Тип значений Objective CAML в C: `value`. Так как представление этих значений является однородным (273), то их можно представить одним единственным типом C. На стр. 358 мы познакомимся со средствами перевода значений и продемонстрируем функцию экспорта значений Objective CAML.

Функция на изображении 11.1 соответствует приведенным ограничениям. Функция `f_c` связана с функцией Objective CAML, тип которой `int -> int -> int -> int` и является функцией с тремя аргументами и результатом типа `value`.

Интерпретатор байт-кода по разному вычисляет вызовы функции в зависимости от числа аргументов¹. Если число аргументов меньше или равно пяти, то они помещаются в стек. Если же число аргументов больше пяти, то в C функцию передается вектор, содержащий все входные аргументы и затем число аргументов. Необходимо разделять оба случая, так как C функция может быть вызвана интерпретатором байт-кода. С другой стороны, нативный компилятор всегда вызывает внешнюю функцию передавая ей напрямую все входные аргументы.

Функция с более 5 аргументов

В данном случае нужно написать две функции, одну для байт-код интерпретатора, а другую для нативного. Синтаксис внешних деклараций позволяет использовать одну декларацию для обеих функций C:

Синтаксис	<pre>external caml_name : type = "C_name_bytecode C_name_native"</pre>
-----------	--

У функции `C_name_bytecode` два аргумента: вектор значений с типом `value` (C указатель на тип `*value`) и целое число, которое указывает размер вектора.

¹Напомним, что в функции `fst` с типом `'a * 'b -> 'a` имеет один аргумент — пара.

Пример

Следующая программа на C определяет две разные функции для сложения шести целых чисел: *plus_native* для нативного кода и *plus_bytecode* для вызова из байт-код интерпретатора. Необходимо указать файл `mlvalues.h`, в котором содержится определение C типов значений Objective CAML и макросы преобразования.

```
#include <stdio.h>
#include <caml/mlvalues.h>

value plus_native (value x1,value x2,value x3,value x4,value x5,value x6)
{
    printf("<< NATIVE PLUS >>\n") ; fflush(stdout) ;
    return Val_long ( Long_val(x1) + Long_val(x2) + Long_val(x3)
                     + Long_val(x4) + Long_val(x5) + Long_val(x6)) ;
}

value plus_bytecode (value * tab_val, int num_val)
{
    int i;
    long res;
    printf("<< BYTECODED PLUS >> : ") ; fflush(stdout) ;
    for (i=0,res=0;i<num_val;i++) res += Long_val(tab_val[i]) ;
    return Val_long(res) ;
}
```

Программа на Objective CAML `exOCAML.ml` вызывает обе C функции.

```
external plus : int -> int -> int -> int -> int -> int -> int
           = "plus_bytecode" "plus_native" ;;
print_int (plus 1 2 3 4 5 6) ;;
print_newline () ;;
```

Теперь скомпилируем обе программы двумя компиляторами Objective CAML и компилятором C, который назовем `cc`.

```
$ cc -c -I/usr/local/lib/ocaml exC.c

$ ocamlc -custom exC.o exOCAML.ml -o ex_byte_code.exe
$ ex_byte_code.exe
<< BYTECODED PLUS >> : 21

$ ocamlopt exC.o exOCAML.ml -o ex_native.exe
```

```
$ ex_native.exe
<< NATIVE PLUS >> :
21
```

Для того, чтобы не переписывать два раза функцию, простейшим решением может быть использование нативной функции в теле функции для байт-код интерпретатора, как указано в примере:

```
value prim_nat (value x1, ..., value xn) { ... }
value prim_bc (value *tab, int n)
{ return prim_nat(tab[0], tab[1], ..., tab[n-1]) ; }
```

11.3.3 Редактирование связей с C

Из скомпилированных файлов на Objective CAML и C, компоновщик создает исполняемый файл. Результат, полученный нативным компилятором, изображен на рисунке 11.2.

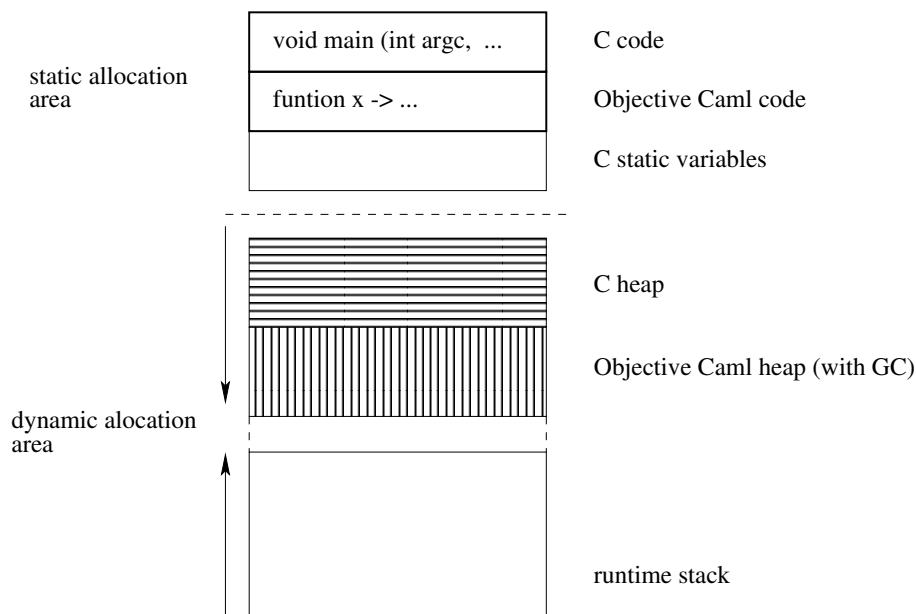


Рис. 11.2: Смешанный исполняемый файл

Инструкции программ на Objective CAML и C размещены в статической зоне памяти. В динамической зоне находится стек выполнения (где располагаются текущие вызовы) и кучи Objective CAML и C.

Runtime библиотека

Функции на C, которые могут быть вызваны в программе и использующие лишь стандартные библиотеки, содержатся в `execution` библиотеке (см. рис. 6.1 на стр. 223). То есть нет необходимости что-либо дополнительно указывать. Однако, в случае если мы используем библиотеки `Graphics`, `Num` и `Str` необходимо указать компоновщику связей библиотеки, соответствующие приведенным модулям. Для этого существует опция компилятора `-custom`. Тот же принцип применяется и к функциям C, которые мы желаем использовать: необходимо указать объектный файл содержащий эти функции при редактировании связей компоновщиком. Мы продемонстрируем это на примере.

Несколько случаев редактирования связей

Необходимо различать компиляцию нативного компилятора, байт-код компилятор и создание интерактивной среды. Опции для различных типов компиляций описаны в главе 6.

Вернемся к изображению 11.1, чтобы проиллюстрировать три режима компиляции. Для этого переименуем файл Objective CAML в `progocaml.ml`. В этом файле вызывается внешняя функция из C файла `progC.c`, который в свою очередь использует C библиотеку `libC`. После того, как эти файлы скомпилированы независимо друг от друга, редакция связей осуществляется следующими командами:

- байт-код:

```
ocamlc -custom -o vbc.exe progC.o a_C_library.a progocaml.cmo
```

- нативный код:

```
ocamlopt progC.o -o vn.exe a_C_library.a progocaml.cmx
```

После этого получим два исполняемых файла: `vbc.exe` для байт-код версии и для нативной `vn.exe`.

Создание новой абстрактной машины

Мы можем обогатить `runtime` библиотеку, включив в нее новые C функции. Для этого существуют следующие команды:

```
ocamlc -make-runtime -o new_ocamlrun progC.o a_C_library.a
```

Теперь можно создать байт-код файл `vbcnam.exe` используя новую абстрактную машину:

```
ocamlc -o vbcnam.exe -use-runtime new_ocamlrun progocaml.cmo
```

Данный код можно запустить в виде аргумента новой абстрактной машине командой `new_ocaml vbcnam.exe` или напрямую `vbcnam.exe`.

Замечание

Редактирование связей в режиме `-custom` вынуждает компилятор сканировать объектные файлы `.cmo` для создания таблицы используемых внешних функций.

Создание интерактивного цикла

Для того чтобы использовать внешнюю функцию в интерактивном цикле, необходимо создать новый цикл, содержащий код C функции и программу на Objective CAML с ее декларацией.

Пусть есть скомпилированный файл `progC.o` с функцией `f_c`. Создадим новый интерактивный цикл `ftop`:

```
ocamlmktop -custom -o ftop progC.o a_C_library.a ex.ml
```

В файле `ex.ml` содержится декларация внешней функции `f`. Благодаря этому указанная функция известна интерактивному циклу `ftop`, а код предоставляется объектом `progC.o`.

11.3.4 Смешивание операций ввода/вывода в C и Objective CAML

Функции C и Objective CAML не разделяют буфера файлового ввода и вывода. Пусть есть программа на C:

```
#include <stdio.h>
#include <caml/mlvalues.h>
value hello_world (value v)
{ printf("Hello World !!"); fflush(stdout); return v; }
```

Для того, чтобы запись на стандартный выход проходила в правильном порядке, необходимо явно сбрасывать содержимое файловых буферов (`fflush`).

```
# external caml_hello_world : unit -> unit = "hello_world" ;;
external caml_hello_world : unit -> unit = "hello_world"
# print_string "<< " ;
  caml_hello_world () ;
```

<code>mlvalues.h</code>	определение типа <code>value</code> и макросов перевода типов
<code>alloc.h</code>	функции выделения памяти для значений Objective CAML
<code>memory.h</code>	макросы, более высоко уровня, для перевода типов

Таблица 11.1: Файлы, используемые для интерфейса с C

```
print_string " >>\n" ;
flush stdout ;;
Hello World !!<< >>
- : unit = ()
```

Полученный вывод не совсем удовлетворительный. Перепишем программу Objective CAML следующим образом:

```
# print_string "<< " ; flush stdout ;
caml_hello_world () ;
print_string " >>\n" ; flush stdout ;;
<< Hello World !! >>
- : unit = ()
```

Систематическое освобождение буфера перед каждой командой записи позволяет соблюсти порядок вывода на экран между двумя языками.

11.4 Анализ значений Objective CAML в C

Машинное представление значений в Objective CAML отличается от C, даже для таких простых типов как целое число. Причиной этому является необходимость в сохранении дополнительной информации GC при сборке памяти. Так как представление значений Objective CAML в памяти однородно, они видны в C как единый тип `value`.

Каждый раз как в Objective CAML вызывается C функция с аргументами, они должны быть переведены в соответствующий тип. То же самое касается результата C функции, вызываемой из Objective CAML.

Для данной цели существует несколько макросов и функций C. Они находятся в файлах, приведенных в таблице 11.1, которые предоставлены дистрибутивом Objective CAML. Они находятся в папке `LIBOCAML/caml`, где `LIBOCAML` является папкой, в которой установлены библиотеки Objective CAML².

²По умолчанию в Unix это `/usr/local/lib/ocaml`. В Windows C:
OCAML
LIB

<i>Is_long</i> (<i>v</i>)	<i>v</i> целое число Objective CAML?
<i>Is_block</i> (<i>v</i>)	<i>v</i> указатель Objective CAML?
<i>Long_val</i> (<i>v</i>)	возвращает целое число C long
<i>Int_val</i> (<i>v</i>)	возвращает целое число C
<i>Bool_val</i> (<i>v</i>)	возвращает "булево" значение C (значение 0 для false)

Таблица 11.2: Перевод непосредственных значений

11.4.1 Классификация значений value

Значение, тип которого `value`, может быть:

- целое число
- указатель на кучу Objective CAML
- указатель на область памяти вне кучи Objective CAML

Куча является зоной памяти, выделяемая программе для динамически размещаемых структур данных. В программе на Objective CAML GC управляет кучей. Программа на C может в свою очередь создать данные в собственной куче и затем передать указатель на выделенное значение Objective CAML.

В таблице 11.2 проведены макросы проверки и перевода типов:

Напомним, что в C существует несколько типов для целого числа: `short`, `int` и `long`, тогда как в Objective CAML существует один единственный тип: `int`.

11.4.2 Доступ к непосредственным значениям

Для представления непосредственных значений Objective CAML используются целые числа:

- целые представлены своим значением
- символы представлены кодом ASCII
- constant constructors are represented by an integer corresponding to their position in the datatype declaration: the *n*th constant constructor of a datatype is represented by the integer *n*-1.

В следующей C программе определена функция *inspect*, которая проверяет значение типа `value`:

```

#include <stdio.h>
#include <caml/mlvalues.h>
value inspect (value v)
{
  if (Is_long(v))
    printf ("v is an integer (%ld) : %ld", (long) v, Long_val(v));
  else if (Is_block(v))
    printf ("v is a pointer");
  else
    printf ("v is neither an integer nor a pointer (???)");
  printf(" ");
  fflush(stdout) ;
  return v ;
}

```

Данная функция проверяет является ли аргумент целым числом. Если да, то сначала на экран выводится значение в виде `int C`, а затем значение, переведенное макросом `Long_val` в целое `C (long)`.

Как видно из следующего примера, представление целых числе в Objective CAML отличается от целых C:

```

# external inspect : 'a -> 'a = "inspect" ;;
external inspect : 'a -> 'a = "inspect"
# inspect 123 ;;
v is an integer (247) : 123 - : int = 123
# inspect max_int;;
v is an integer (2147483647) : 1073741823 - : int = 1073741823

```

Другие стандартные типы, как например `char` и `char` представленные непосредственными значениями.

```

# inspect 'A' ;;
v is an integer (131) : 65 - : char = 'A'
# inspect true ;;
v is an integer (3) : 1 - : bool = true
# inspect false ;;
v is an integer (1) : 0 - : bool = false
# inspect [] ;;
v is an integer (1) : 0 - : '_a list = []

```

Определим тип `foo` в Objective CAML:

```

# type foo = C1 | C2 of int | C3 | C4 ;;

```


Функция *inspect* выводит различный результат в зависимости от конструктора, функционального или константного,

```
# inspect C1 ;;
v is an integer (1) : 0    - : foo = C1
# inspect C4 ;;
v is an integer (5) : 2    - : foo = C4
# inspect (C2 1) ;;
v is a pointer      - : foo = C2 1
```

Когда функция распознает непосредственное значение, она выводит "физическое" содержимое этого значения в скобках (т.е. значение в виде целого числа со знаком, занимающего одно машинное слово – `int` в *C*), а затем выводит "логическое" содержимое (значение, полученное макросами перевода типов).

В приведенных примерах, мы видим разницу между "физическим" и "логическим" содержимым. Она обусловлена бит-тегом³, который используется GC для того, чтобы различить непосредственное значение от указателя (см. главу 8 на стр. 273).

11.4.3 Дискриминация структурных значений

Кроме непосредственных, все остальные значения являются структурными значениями Objective CAML: n-уплеты, непустые списки, конструкторы с одним и более аргументов, векторы, записи, замыкания и абстрактные значения. Память для них выделяется в куче в виде блоков. У каждого блока имеется заголовок, содержащий информацию о содержимом блока и размере блока в машинных словах. На изображении 11.3 нарисована структура блока для компьютера со словом длиной в 32 бита.

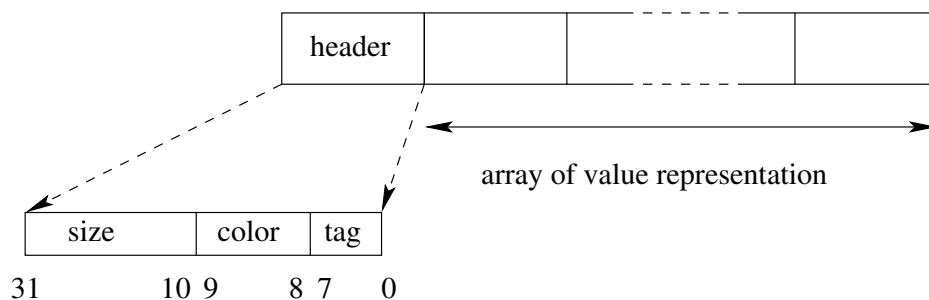


Рис. 11.3: Структура блока в куче Objective CAML

³В приведенных примерах бит-тег является младшим битом.

<code>Wosize_val(v)</code>	возвращает размер блока (без заголовка)
<code>Tag_val(v)</code>	возвращает тег блока

Таблица 11.3: Информация о блоках памяти

from 0 to <code>No_scan_tag</code> -1	вектор значений <code>value</code> Objective CAML
<code>Closure_tag</code>	замыкание
<code>String_tag</code>	строка символов
<code>Double_tag</code>	вещественное число двойной точности
<code>Double_array_tag</code>	вектор вещественных чисел
<code>Abstract_tag</code>	абстрактный тип данных
<code>Final_tag</code>	абстрактный тип данных с завершающей функцией

Таблица 11.4: Описание тегов блоков памяти

Два "цветных" бита используются GC при прохождении графа значений в куче (см. 8, стр 273). Поле "тег" указывает на "тип" значения, которое содержится в блоке. В таблице 11.3 перечисляются функции, возвращающие эту информацию.

В таблице 11.4 указаны различные значения тегов.

В зависимости от значения, которое возвращается функцией `Tag_val`, используются разные макросы для доступа к блоку памяти. Если оно меньше чем `No_scan_tag`, то структура блока памяти это массив значений `value`. Макросы, для получения значений из блока памяти описаны в таблице 11.5. В соответствии с языком C и Objective CAML, первый элемент массива находится по индексу 0.

Как и для непосредственных значений, определим функцию просмотра блока памяти. С функция `print_block` проверяет входной аргумент (тип `value`) является ли он непосредственным значением или блоком памяти. Если он оказывается блоком памяти, то функция выведет тип бло-

<code>Field(v,n)</code>	возвращает n-ое значение вектора
<code>Code_val(v)</code>	возвращает code pointer для замыкания
<code>string_length(v)</code>	возвращает длину строки
<code>Byte(v,n)</code>	возвращает n-ый символ строки, в виде типа <code>char</code>
<code>Byte_u(v,n)</code>	то же самое, но возвращает тип <code>unsigned char</code>
<code>String_val(v)</code>	возвращает строку с типом C (<code>char *</code>)
<code>Double_val(v)</code>	возвращает вещественное
<code>Double_field(v,n)</code>	возвращает n-ый элемент вектора вещественных

Таблица 11.5: Доступ к элементам блока

ка и его значение. Данная функция вызывается из функции *inspect_block*, которая будет вызвана из программы Objective CAML.

```
#include <stdio.h>
#include <caml/mlvalues.h>

void margin (int n)
{ while (n-- > 0) printf("."); return; }

void print_block (value v,int m)
{
    int size, i;
    margin(m);
    if (Is_long(v))
        { printf("immediate value (%d)\n", Long_val(v)); return; };
    printf ("memory block: size=%d - ", size=Wosize_val(v));
    switch (Tag_val(v))
    {
        case Closure_tag :
            printf("closure with %d free variables\n", size-1);
            margin(m+4); printf("code pointer: %p\n",Code_val(v)) ;
            for (i=1;i<size;i++) print_block(Field(v,i), m+4);
            break;
        case String_tag :
            printf("string: %s (%s)\n", String_val(v),(char *) v);
            break;
        case Double_tag:
            printf("float: %g\n", Double_val(v));
            break;
        case Double_array_tag :
            printf ("float array: ");
            for (i=0;i<size/Double_wosize;i++) printf(" %g", Double_field(
                v,i));
            printf("\n");
            break;
        case Abstract_tag : printf("abstract type\n"); break;
        case Final_tag : printf("abstract finalized type\n"); break;
        default:
            if (Tag_val(v)>=No_scan_tag) { printf("unknown tag"); break;
            };
            printf("structured block (tag=%d):\n",Tag_val(v));
```

```

        for (i=0;i<size;i++) print_block(Field(v,i),m+4);
    }
    return ;
}

```

```

value inspect_block (value v)
{ print_block(v,4); fflush(stdout); return v; }

```

Различные значения типов блока обрабатываются оператором **switch**. Определим функцию *inspect*.

```

# external inspect : 'a -> 'a = "inspect_block" ;;
external inspect : 'a -> 'a = "inspect_block"

```

Эта функция используется для описания структурных значений Objective CAML. Входной аргумент не должен быть в виде замкнутой (кольцевой) структуры, иначе функция заиклится.

Массивы, n-уплеты и записи

Массивы и n-уплеты представлены в виде массивов **value**.

```

# inspect [| 1; 2; 3 |] ;;
.... memory block: size=3 — structured block (tag=0):
..... immediate value (1)
..... immediate value (2)
..... immediate value (3)
- : int array = [|1; 2; 3|]
# inspect ( 10 , true , () ) ;;
.... memory block: size=3 — structured block (tag=0):
..... immediate value (10)
..... immediate value (1)
..... immediate value (0)
- : int * bool * unit = 10, true, ()

```

Массивы типов **value** используются для представления записей Objective CAML в том же порядке что и декларация типов. Тот факт, что поле является изменяемым или нет, не влияет на его физическое представление.

```

# type foo = { fld1: int ; mutable fld2: int } ;;
type foo = { fld1: int ; mutable fld2: int }
# inspect { fld1=10 ; fld2=20 } ;;
.... memory block: size=2 — structured block (tag=0):
..... immediate value (10)

```

```

..... immediate value (20)
- : foo = {fld1=10; fld2=20}

```

Warning

C функции могут беспрепятственно физически изменить неизменяемые значения Objective CAML. Контроль за соответствием использования функций C лежит плечах разработчика.

Тип сумма

Как мы уже видели, константные конструкторы представлены целыми числами. Для представления других конструкторов используется вектор, в котором содержатся аргументы конструктора. Конструктор распознается по значению *тега*. Этот тег соответствует порядку определения конструктора в типе: 0 соответствует первому конструктору, 1 второму и т.д.

```

# type foo = C1 of int * int * int | C2 of int | C3 | C4 of int * int ;;
type foo = | C1 of int * int * int | C2 of int | C3 | C4 of int * int
# inspect (C1 (1,2,3)) ;;
.... memory block: size=3 - structured block (tag=0):
..... immediate value (1)
..... immediate value (2)
..... immediate value (3)
- : foo = C1 (1, 2, 3)
# inspect (C4 (1,2)) ;;
.... memory block: size=2 - structured block (tag=2):
..... immediate value (1)
..... immediate value (2)
- : foo = C4 (1, 2)

```

Тип `list` это тип сумма, которая определена следующим образом:

```
type 'a list = [] | :: of 'a * 'a list
```

У данного типа всего один неконстантный конструктор (`::`) с единственным тегом 0.

Строка символов

Каждый символ строки занимает один байт. Таким образом блок памяти, представляющий строку, хранит 4 символа в одном машинном слове (на 32 битной архитектуре).

Warning

В строках Objective CAML может содержаться символ с кодом ASCII 0, что соответствует символу конца строки в C.

```
#include <stdio.h>
#include <caml/mlvalues.h>

value explore_string (value v)
{
    char *s;
    int i, size;
    s = (char *) v;
    size = Wosize_val(v) * sizeof(value);
    for (i=0;i<size;i++)
    {
        int p = (unsigned int) s[i];
        if ((p>31) && (p<128)) printf("%c",s[i]); else printf("(#%u)",p);
    }
    printf("\n");
    fflush(stdout);
    return v;
}
```

Конец строки в Objective CAML определяется с помощью размера блока, из которого она состоит, а также с помощью последнего байта последнего слова блока, в котором указывается число *неиспользованных* байтов в последнем слове. На следующем примере станет понятней роль последнего байта.

```
# external explore : string -> string = "explore_string" ;;
external explore : string -> string = "explore_string"
# ignore(explore "");
ignore(explore "a");
ignore(explore "ab");
ignore(explore "abc");
ignore(explore "abcd");
ignore(explore "abcd\000") ;;
(#0)(#0)(#0)(#3)
a(#0)(#0)(#2)
ab(#0)(#1)
abc(#0)
```

```
abcd(#0)(#0)(#0)(#3)
abcd(#0)(#0)(#0)(#2)
- : unit = ()
```

В двух последних примерах (“abcd” et “abcd 000”) длина строк соответственно 4 и 5 символов. По этой причине последний байт меняет свое значение.

Вещественные и вектор вещественных значений

В Objective CAML имеется всего один тип для чисел с плавающей запятой: `float`. Значения для типа `float` выделяются в куче и имеют размер в 2 машинных слова.

```
# inspect 1.5 ;;
.... memory block: size=2 - float : 1.5
- : float = 1.5
# inspect 0.0;;
.... memory block: size=2 - float : 0
- : float = 0
```

Вектор вещественных значений имеет специальную структуру, которая позволяет оптимально использовать память. По этой причине у такого вектора имеется специальный, отличный от других тегов макрос доступа.

```
# inspect [| 1.5 ; 2.5 ; 3.5 |] ;;
.... memory block: size=6 - float array: 1.5 2.5 3.5
- : float array = [|1.5; 2.5; 3.5|]
```

Благодаря данной оптимизации Objective CAML может выполнять числовые расчеты с подобным типом данных намного быстрее, чем при использовании указателя на кучу.

Warning

Когда в *C* необходимо выделить вектор для вещественных чисел Objective CAML, размер вектора вычисляется по следующей формуле: число элементов помноженное на `Double_wosize`. Данный макрос указывает число слов необходимых для вещественных чисел с двойной точностью.

За исключением `float array`, вещественные содержащиеся в других структурах данных сохраняют их обычную представление, то есть как структурное значение, выделенное в куче. В следующем примере анализируется список вещественных.

```
# inspect [ 3.14; 1.2; 7.6];;
.... memory block: size=2 — structured block (tag=0):
..... memory block: size=2 — float : 3.14
..... memory block: size=2 — structured block (tag=0):
..... memory block: size=2 — float : 1.2
..... memory block: size=2 — structured block (tag=0):
..... memory block: size=2 — float : 7.6
..... immediate value (0)
— : float list = [3.14; 1.2; 7.6]
```

Список виден как блок размером в два слова, в которых содержится заголовок и хвост списка. Заголовок списка это вещественное, также размером в два слова.

Замыкание

Функциональное значение характеризуется кодом с одной стороны и окружением с другой (см. 1 на стр. 15). Существует два способа, с помощью которых можно получить функциональное значение: явно использовать абстракцию (как например в `fun x -> x+1`) или частично применить функцию (`(fun x -> fun y -> x+y) 1`).

В окружении замыкания может находиться 3 типа переменных: глобальные, локальные и унаследованные частичным применением. Реализация все трех категорий переменных различна. Глобальные переменные являются частью глобального окружения и явно не видны в окружении замыкания. Как мы увидим далее, локальные и унаследованные частичным применением параметры могут находиться в замыкании. Таким образом, с точки зрения реализации, окружение замыкания касается лишь локальных и абстрактных параметров.

Если окружение замыкания пусто, то в нем имеется лишь указатель на код.

```
# let f = fun x y z -> x+y+z ;;
val f : int -> int -> int -> int = <fun>
# inspect f ;;
.... memory block: size=1 — closure with 0 free variables
..... code pointer: 0x807308c
— : int -> int -> int -> int = <fun>
```

Если она получена частичным применением какой-нибудь абстракции, без локального объявления, это замыкание будет содержать лишь значения для каждого параметра и замыкание без окружения.


```

# let a1 = f 1 ;;
val a1 : int -> int -> int = <fun>
# inspect (a1) ;;
... memory block: size=3 — closure with 2 free variables
..... code pointer: 0x8073088
..... memory block: size=1 — closure with 0 free variables
..... code pointer: 0x807308c
..... immediate value (1)
- : int -> int -> int = <fun>
# let a2 = a1 2 ;;
val a2 : int -> int = <fun>
# inspect (a2) ;;
... memory block: size=4 — closure with 3 free variables
..... code pointer: 0x8073088
..... memory block: size=1 — closure with 0 free variables
..... code pointer: 0x807308c
..... immediate value (1)
..... immediate value (2)
- : int -> int = <fun>

```

На рис. 11.4 графически изображен предыдущий вывод.

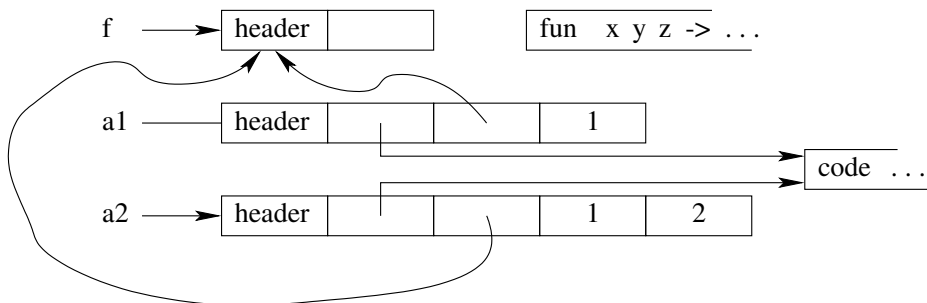


Рис. 11.4: Представление замыкания

В функции `f` отсутствуют свободные переменные. Для замыкания без окружения, указатель кода указывает на код, который необходимо применить, когда все необходимые аргументы будут указаны, то есть в данном случае это `x+y+z`. Замыкание с окружением указывает на один и тот же общий код (здесь один и тот же код для `a1` и `a2`). Данный код проверяет что все необходимые аргументы на месте и в этом случае они проталкиваются и код выполняется. В противном случае будет создано новое замыкание, с окружением в котором как всегда на первом месте указатель на замыкание без окружения из которого частичное замыка-

ние получено. Это позволяет запустить реальный код с одной стороны и хранить указатель на самого себя для рекурсивных функций.

В случае когда используется локальная декларация, частичное применение выглядит так:

```
# let g x = let y=2 in fun z -> x+y+z ;;
val g : int -> int -> int = <fun>
# let a1 = g 1 ;;
val a1 : int -> int = <fun>
# inspect a1 ;;
.... memory block: size=3 - closure with 2 free variables
..... code pointer: 0x8086548
..... immediate value (1)
..... immediate value (2)
- : int -> int = <fun>
```

Вызов замыкания Objective CAML из C рассмотрен в гл. 11.4.4.

Абстрактные типы

Значение абстрактного типа представлено вектором значений `value`. На самом деле, информация о типе нужна лишь синтезатору типов. Во время выполнения программы информация о типе не требуется, лишь только GC необходимо знать представление в памяти и размер значений.

11.4.4 Вызов замыкания Objective CAML в C

Глава 12

Программы

Часть III

Устройство программы

Глава 13

Модульное программирование

Глава 14

Объектно-ориентированное программирование

Глава 15

Сравнение моделей устройств программ

Глава 16

Программы

Часть IV

Параллелизм и распределение

Глава 17

Процессы и связь между процессами

Глава 18

Программирование одновременно–выполняемых задач

Глава 19

Программирование распределенных задач

Глава 20

Программы

Глава 21

Разработка программ с помощью Objective CAML

Часть V

Annexes

