Введение в Стандартный ML[1]

Роберт Харпер
School of Computer Science
Carnegie Mellon University
Pittsburg, PA 15213-3891

Перевод на русский язык М.Р.Ковтуна
под редакцией С.А.Романенко

[1] С упражнениями Кевина Митчела, Laboratory for Foundations of Computer Science, Edinburgh University, Edinburgh, United Kingdom.

Оглавление

Благодарности

Некоторые из использованных примеров были заимствованы из работ Луки Карделли [3], Робина Милнера [5], Дэйва Мак-Квина [6] и Абельсона и Зуссмана [1]. Иоахим Парроу, Дон Санёлла и Дэвид Уокер внесли ряд ценных предложений.

Глава 1

Введение

Эти лекции являются вводным курсом в язык программирования Стандартный ML. Перечислим наиболее важные черты Стандартного ML:

- ML является *функциональным* языком программирования. Функции являются полноправными объектами: они могут передаваться в качестве аргументов, вырабатываться в качестве результата и храниться в переменных. Основной способ организации управления в программах на ML — рекурсивное применение функций.

- ML является *интерактивным* языком. Каждое введенное предложение анализируется, компилируется и исполняется, и значение, полученное в результате исполнения предложения, вместе с его типом выдается пользователю.

- ML является *строго типизированным* языком. Каждое допустимое выражение имеет тип, который автоматически определяется компилятором. Строгая типизация гарантирует, что в период исполнения не может возникнуть ошибок в согласовании типов.[1]

- ML имеет *полиморфную* систему типов. Каждое допустимое предложение языка обладает однозначно определяемой *наиболее общей типизацией*, которая определяет контексты, в которых предложение может быть использовано.

- ML поддерживает *абстрактные типы данных.* Абстрактные типы — весьма полезный механизм в модульном программировании. Новые типы данных могут быть определены вместе с набором операций над значениями этих типов. Детали внутренней структуры

[1] Заметим, что ошибки такого рода — весьма распространенная вещь, и диагностика их на этапе компиляции заметно облегчает отладку программы. *(Прим. перев.)*

значений и реализации операций скрыты от программиста, использующего данные абстрактного типа; высокая степень изоляции дает существенные преимущества при разработке и сопровождении больших программ.

- В ML области действия идентификаторов определяются *статически*. Смысл всех идентификаторов в программе определяется статически, что позволяет создавать более модульные и более эффективные программы.

- ML содержит надежно типизированный механизм обработки *исключительных событий*. Механизм обработки исключительных событий является удобным средством описания действий в ненормальных ситуациях, возникающих в период исполнения программы.

- ML содержит *средства разбиения программ на модули*, обеспечивающие возможность разработки больших программ по частям. Программа на ML строится как набор взаимозависимых *структур*, связываемых друг с другом с помощью *функторов*. Раздельная компиляция осуществляется посредством экспорта и импорта функторов.

Стандартный ML является новейшим из семейства языков, берущих свое начало от языка ML, разработанного в Эдинбурге Майком Гордоном, Робином Милнером и Крисом Уодсвортом в середине 70-х годов [4]. С тех пор возникло много диалектов и реализаций, как в самом Эдинбурге, так и в других местах. Стандартный ML основывается на синтезе многих идей, воплощенных в различных вариантах языка (из которых в первую очередь следует отметить диалект, разработанный Лукой Карделли [3]), и на идеях функционального языка HOPE, разработанного Родом Берстелом, Дэйвом Мак-Квином и Доном Санёлла [2]. Последнее добавление в язык — модульная структура, предложенная Дэйвом Мак-Квином в [6].

Эти лекции являются неформальным введением в язык, описывающим его возможности и способы использования, и не должны рассматриваться как формальное описание Стандартного ML. Они писалась и переписывалась в течение нескольких лет, и сейчас нуждаются в некоторой переработке с целью отразить изменения, произошедшие в языке, и опыт его использования. Поэтому автор будет рад любым предложениям от читателей по этому вопросу.

Формальное определение Стандартного ML имеется в [7]. Несколько менее формальное (и несколько устаревшее) описание имеется в Отчете Эдинбургского университета [5]. Для детального знакомства с языком читателю следует обратиться к его формальному определению.

Глава 2

Ядро языка

2.1 Работа с ML в режиме диалога

Практически все реализации ML являются интерактивными; они основываются на диалоге типа “прочесть-вычислить-напечатать” (хорошо знакомому пользователям LISP). В процессе такого диалога пользователь вводит выражение, ML-система его анализирует, компилирует, выполняет и выдает результат на терминал[1].

Вот образец диалога:

```
- 3+2;
> 5 : int
```

ML запрашивает ввод с помощью “`- `” и предваряет вывод “`> `”. Пользователь вводит фразу “`3+2`”. ML вычисляет выражение и печатает его значение, “`5`”, вместе с типом значения, “`int`”.

В процессе диалога могут возникать различные виды ошибок. В основном они распадаются на три категории: синтаксические ошибки, ошибки в согласовании типов и ошибки времени исполнения. Вероятно, вы знакомы с синтаксическими ошибками и ошибками времени исполнения по вашему опыту работы с другими языками программирования. Приведем пример того, что происходит, когда введена синтаксически неправильная фраза:

```
- let x=3 in x end;
Parse error: Was expecting "in" in ... let <?> x ...
```

Ошибки времени исполнения (как, например, деление на 0) являются одним из видов *исключительных событий* (подробно они будут рассмот-

[1] Детали интерактивной работы с ML-системой меняются от одной реализации к другой, но “дух” общения одинаков во всех известных автору системах. Примеры в настоящей книге подготовлены с использованием Эдинбургского компилятора 1988 года.

рены далее). Сейчас мы только приведем пример того, что вы можете получить при возникновении ошибки времени исполнения:

```
-3 div 0;
Failure:   Div
```

Ошибки в согласовании типов не так привычны. Мы подробно рассмотрим типы и связанные с ними ошибки позднее; пока отметим, что ошибки в согласовании типов возникают при некорректном использовании значений, например, при попытке прибавить 3 к true:

```
- 3+true;
Type clash in:   3+true
Looking for a:   int
I have found a:  bool
```

Одной из весьма неприятных ошибок, не распознаваемых ML-системой, является бесконечный цикл. Если вы подозреваете, что ваша программа зациклилась, вы можете прекратить ее выполнение, нажав клавишу прерывания (обычно `Ctrl-C`). ML выдаст сообщение, говорящее о том, что возникло исключительное событие "interrupt", и вернется на верхний уровень выполнения программы. Некоторые реализации содержат средства отладки, которые могут помочь в определении причины зацикливания.

Бывают и другие виды ошибок, но они встречаются значительно реже, и возможные причины их трудно объяснить в общем случае. Если вы встретитесь с сообщением об ошибке, смысл которого вы не можете понять, постарайтесь найти кого-нибудь, кто имеет больше опыта в работе с ML, чтобы он помог вам разобраться в ситуации.

Детали интерфейса пользователя меняются от реализации к реализации, особенно в части формата вывода и сообщений об ошибках. Приводимые в настоящей книге примеры основываются на Эдинбургском ML; мы полагаем, что после знакомства с этими примерами у вас не должно возникнуть трудностей в понимании выдач других ML-систем.

2.2 Первичные выражения, значения и типы

Мы начнем наше введение в ML с описания множества *первичных* типов. В ML тип есть множество значений. Например, целые числа образуют тип; множество всех символьных строк и множество логических значений (“истина” и “ложь”) также образуют типы. Если имеются два типа σ и τ, то множество всех упорядоченных пар, первый член которых имеет тип σ, а второй — тип τ, является типом. Более того, множество всех функций, отображающих значения одного типа в значения другого типа, также образуют тип. В дополнение к этим и другим определенным в языке типам ML позволяет программисту определить свои собственные типы; к более подробному обсуждению этой возможности мы обратимся позднее.

Выражения в ML изображают значения точно так же, как последовательности цифр изображают числа. Тип выражения определяется системой правил, которые гарантируют, что если выражение имеет значение, то значение выражения принадлежит типу выражения (ну как, понятно?). Например, каждая последовательность цифр имеет тип `int`, поскольку значением последовательности цифр является целое число. Мы проиллюстрируем систему типов ML примерами.

2.2.1 Тип `unit`

Тип `unit` состоит из единственного значения, записываемого как `()`. Этот тип используется в тех случаях, когда выражение не имеет какого-либо осмысленного значения, а также вместо аргумента функции в тех случаях, когда функция не должна иметь аргументов[2].

2.2.2 Логические значения (тип `bool`)

Тип `bool` состоит из значений `true` и `false`. Для работы со значениями этого типа имеются одноместная операция `not` и две двухместных операции `andalso` и `orelse` (соответствующие обычным логическим операциям "не", "и" и "или")[3].

Условное выражение `if` e `then` e_1 `else` e_2 мы также рассмотрим здесь, поскольку его первый аргумент, e, должен иметь тип `bool`. Обратите внимание, что часть `else` должна присутствовать обязательно. Причина этого в том, что `if` в ML является условным *выражением*, а не условным *утверждением*, как, например, в Pascal'е. Если бы часть

[2] Иногда возникает необходимость рассматривать функции, принимающие постоянные (не зависящие от аргумента) значения. Поскольку зачастую в таких случаях область определения несущественна, уместно в качестве нее использовать `unit`. При этом в ML важно отличать саму функцию (что записывается как `f`) от результата ее применения (что записывается как `f()`). *(Прим. перев.)*

[3] Имеются веские причины, по которым вместо традиционных названий `and` и `or` для бинарных логических операций используются неожиданные `andalso` и `orelse`. Причины этого следующие. ML является *строгим* языком; это означает, что все аргументы функции вычисляются до вычисления значения функции. Если при этом окажется, что какой-либо аргумент не может быть вычислен (например, в процессе его вычисления возникает зацикливание), то и само значение функции не сможет быть вычислено. Операции же `andalso` и `orelse` являются исключениями из этого правила: при вычислении, например, e_1 `andalso` e_2 сначала вычисляется e_1, и если его значение есть `false`, то e_2 не вычисляется, а значением всего выражения будет `false`. Таким образом, выражение `false andalso` e_2 будет всегда определено (и будет иметь значение `false`) — в отличие, например, от выражения `0 *` e_2, значение которого будет не определено, если не определено e_2. *(Прим. перев.)*

`else` отсутствовала, то при значении e равном `false` выражение не имело бы значения. Кроме того, типы выражений e_1 и e_2 должны совпадать. Выражение

```
if true then true else ()
```

является *некорректным с точки зрения типизации* (или, короче, *нетипизируемым*), поскольку тип выражения в части `then` есть `bool`, а тип выражения в части `else` — `unit`.

```
- not true;
> false : bool
- false andalso true;
> false: booi
- false orelse true;
> true : bool
- if false then false else true;
> true: bool
- if true then false else true;
> false: bool
```

2.2.3 Целые числа (тип int)

Тип `int` является множеством (положительных и отрицательных) целых чисел. Целые числа записываются обычным способом, за исключением того, что для отрицательных чисел знак записывается тильдой `~` вместо традиционного минуса `-`.

```
- 75;
> 75 : itn
- ~24;
> ~24 : int
- (3+2) div 2;
> 2 : int
- (3+2) mod 2;
> 1 : int
```

Выражения могут содержать обычные знаки арифметических операций `+`, `-`, `*`, `div` и `mod` (`div` и `mod` обозначают соответственно целочисленное деление и получение остатка от деления нацело). Имеются и знаки для операций сравнения `<`, `<=`, `>`, `>=`, `=` и `<>`. Все эти операции требуют двух аргументов типа `int` и возвращают результат типа `bool` (`true` или `false` в соответствии с тем, выполнено условие или нет).

```
- 3<2
> false : bool
```

```
- 3*2 >= 12 div 6
> true: bool
- if 4*5 mod 3 = 1 then 17 else 51;
> 51 : int
```

Обратите внимание на то, что операции сравнения, примененные к двум целочисленным выражениям, вырабатывают `true` или `false`, и поэтому их результат имеет тип `bool`.

2.2.4 Строки (тип `string`)

Тип `string` состоит из всех конечных последовательностей литер. Строки записываются обычным способом, как последовательность литер между двойными кавычками. Если двойная кавычка должна войти в состав строки, она записывается как “`\"`”.

```
- "Fish knuckles";
> "Fish knuckles" : string
- "\"";
> """ : string
```

Специальные литеры также могут быть вставлены в строки, но мы не будем останавливаться на способе сделать это, т.к. в наших примерах это не понадобится. За более подробной информацией обратитесь к формальному описанию языка [7].

Функция `size` возвращает длину строки в литерах. Функция `^` является инфиксной двухместной операцией конкатенации строк[4].

```
- "Rhinocerous " ^ "Party";
> "Rhinocerous Party" : string
- size "Walrus whistle";
> 14 : int
```

2.2.5 Действительные числа (тип `real`)

Тип чисел с плавающей точкой называется в ML `real`. Действительные числа записываются более-менее обычным для языков программирования способом: целое число, за которым следует десятичная точка, за которой следует последовательность из одной или более десятичных цифр, за которыми следует знак экспоненты, `E`, за которым следует другое число. Экспонента может быть опущена, если присутствует десятичная точка; десятичная точка может быть опущена, если присутствует экспонента. Все это нужно для того, чтобы отличать целые числа от

[4] *Инфиксной* мы называем функцию, имеющую два аргумента, знак которой записывает между аргументами — как, например, обычно записывается в математике операция сложения.

действительных (ML не обеспечивает включения одного типа в другой ни в какой форме; при необходимости целое число должно быть явным образом преобразовано в действительное).

```
- 3.14159;
> 3.14159: real
- 3E2;
> 300.0 : real
- 3.14159E2;
> 314.159 : real
```

Для действительных чисел имеются обычные арифметические операции `-`, `+`, `-`, `*`, `/` (“/” обозначает операцию деления действительных чисел; операций `div` и `mod` для действительных чисел нет) и операции сравнения `<`, `<=`, `>`, `>=`, `=`, `<>`. Не допускается, чтобы один из операндов этих операций был действительным, а другой — целым. Также имеются функции `sin`, `sqrt`, `exp` и т.д., соответствующие обычным математическим функциям. Функция `real` преобразует целый аргумент в соответствующее действительное число, а функция `floor` преобразует действительный аргумент в наибольшее целое число, не превосходящее аргумент.

```
- 3.0 + 2.0;
> 5.0 : real
- (3.0 + 2.0) = real(3 + 2);
> true : bool
- floor(~3.2);
> ~4 : int
- cos(0.0);
> 1.0: real
- cos(0);
Type clash in:  (cos 0)
Looking for a:  real
I have found a: int
```

Этим завершается обзор *атомарных* типов ML. Теперь мы переходим к составным типам, которые строятся из других типов.

2.2.6 Упорядоченные энки

Тип σ`*`τ, где σ и τ — произвольные типы, является типом упорядоченной пары, где первый член имеет тип σ, а второй — тип τ. Упорядоченная пара записывается как `(`e_1`,`e_2`)`, где e_1 и e_2 — выражения. Аналогично образуются упорядоченные энки[5]. А именно, последовательность выражений, отделенных друг от друга запятыми, заключается в

[5]Мы решились ввести термин *энка* как “обобщение” терминов *двойка*, *тройка* и т.д. (вместо выглядящего несколько странно n-ка) по аналогии с получившим широкое распространение английским термином *tuple*. *(Прим. перев.)*

круглые скобки. Если выражения e_1, e_2, ..., e_n имеют типы σ_1, σ_2, ..., σ_n соответственно, то типом упорядоченной энки (e_1,e_2,...,e_n) будет σ_1*σ_2*...*σ_n.

```
- (true, ());
> (true, ()): bool * unit
- (1, false, 17, "Blubbet");
> (1,false, 17,"Blubbet"): int * bool * int * string
- (if 3=5 then "Yes" else "No", 14 mod 3 );
> ("No",2): string * int
```

Равенство между энками — покомпонентное: две энки равны, если их соответствующие компоненты равны. Если две энки имеют различные типы, то при попытке их сравнения возникает ошибка в согласовании типов: бессмысленно спрашивать, равны ли `("abc", ())` и `(true, 7)`, поскольку соответствующие компоненты этих пар имеют разные типы.

```
- (14 mod 3, not false ) = ( 1+1, true );
> true : bool
- ( "abc", (5*4) div 2 ) = ("a"^"bc", 11);
> false : bool
- (true, 7) = ("abc", ());
Type clash in:   (true,7) = ("abc",())
Looking for a:    bool * int
I have found a: string * unit
```

2.2.7 Списки

Тип τ `list` состоит из конечных последовательностей, или *списков*, значений типа τ. Например, тип `int list` состоит из списков целых чисел, а тип `bool list` состоит из списков логических значений. Имеется два способа записи списков, основной и сокращенный. Первый основывается на следующем определении списка: список значений типа τ либо пуст, либо состоит из значения типа τ, за которым следует список значений типа τ. В соответствии с этим определением пустой список записывается как `nil`, а непустой список записывается как e `::` l, где e — выражение типа τ, а l — выражение типа τ `list`. Название операции `::` произносится как "конс" ("cons") — в память о соответствующей функции языка LISP.

Поразмыслив некоторое время, вы поймете, что любой непустой список может быть записан в следующем виде:

e_1 `::` e_2 `::` ... `::` e_n `::` `nil` где каждое e_i является выражением типа τ и $n \geq 1$. Это соответствует интуитивному понятию списка значений данного типа, при этом `nil` играет роль завершителя списка.

Такой метод определения называется *индуктивным* (или *рекурсивным*) определением. Индуктивное определение состоит из одного или более *начальных случаев*, описывающих определяемые предметы непосредственно, и одного или более *индуктивных шагов*, позволяющих из уже построенных предметов строить новые[6].

В случае списков, начальным случаем является `nil`, а индуктивным шагом является применение операции `::`. Индуктивные определения типов играют важную роль в функциональном программировании, поскольку организация функциональной программы во многом определяется структурой обрабатываемых данных.

Ниже приводятся примеры использования `nil` и `::` для построения списков:

```
- nil;
> [] : 'a list
- 3 :: 4 :: nil;
> [3,4] : int list
- (3:: nil):: (4 :: 5 :: nil) :: nil;
> [[3],[4,5]]: int list list
- ["This", "is", "it" ];
> ["This","is","it"]: string list
```

Обратите внимание на то, что ML выводит списки в сокращенном формате: последовательность разделенных запятыми элементов списка, заключенная в квадратные скобки. Этот формат может использоваться и для ввода списков — но помните, что это лишь сокращение для полного формата.

Тип списка `nil` — особый: он включает *переменную типа* (выводится как `'a` и произносится как “альфа”). Причиной этого является то, что в пустом списке нет никакой информации о том, списком элементов какого типа он является. Было бы глупо требовать, чтобы имелись отдельные константы, обозначающие пустой список целых, пустой список логических и т.д. Поэтому ML трактует `nil` как *полиморфный* объект, т.е. как такой объект, который может принадлежать любому типу из некоторого класса. Константа `nil` может рассматриваться как `int list`, как `bool list` или как `int list list` в зависимости от контекста, в котором она

[6] Индуктивное определение задает определяемый класс предметов в соответствии со следующими тремя пунктами: (1) всякий предмет, описанный одним из начальных случаев, является определяемым предметом: (2) всякий предмет, полученный из уже построенных определяемых предметов путем применения какого-либо индуктивного шага, является определяемым предметом; (3) никаких других определяемых предметов, кроме полученных с помощью пунктов (1) и (2), нет. *(Прим. перев.)*

находится. Это выражается тем, что константе `nil` приписывается тип `'a list`, где `'a` является переменной, пробегающей множество типов. Частные случаи типа, включающего переменные типа (такой тип мы будем также называть *полиморфным типом*, или, короче, *политипом*), получаются путем замены всех вхождений данной переменной типа на какой-либо тип (при этом *все* вхождения данной переменной заменяются на *один и тот же* тип); подставляемый вместо переменной тип сам может быть полиморфным. Например, типы `int list` и `(int*'b) list` являются частными случаями политипа `'a list`. Типы, которые не содержат переменных типа, называются *мономорфными типами* (или, короче, *монотипами*).

Равенство для списков является поэлементным: два списка равны, если они состоят из одного и того же числа элементов и соответствующие элементы равны между собой. Как и в случае упорядоченных энок, невозможно сравнение двух списков с элементами разных типов, и попытка выполнить такую операцию приведет к ошибке в согласовании типов.

```
- [1,2,3] = 1 :: 2 :: 3 :: nil;
> true : bool
- [[1], [2,4] ] = [ [2 div 2], [1+1, 9 div 3] ];
> false : bool
```

2.2.8 Записи

Последний составной тип, который мы рассмотрим в настоящем разделе, есть *тип записи*. Записи в ML аналогичны записям в Pascal'е, структурам в C и т.д. Запись состоит из конечного множества помеченных полей, каждое из которых является значением некоторого типа; как и в случае упорядоченных энок, различные поля могут иметь различные типы. Запись задается последовательностью равенств в форме l=e (где l есть метка и e — выражение), заключенной в фигурные скобки. Каждое равенство l=e устанавливает значение поля, помеченного меткой l=e, равным значению выражения e. Типом записи является последовательность пар вида $l:\tau$ (где l есть метка, и τ — тип), разделенных запятыми и заключенных в фигурные скобки. Порядок равенств, задающих запись, не имеет значения: компоненты записи определяются своими метками, а не положением в записи. Равенство для записей — покомпонентное: две записи равны, если они имеют одно и то же множество меток полей, и поля, помеченные одинаковыми метками, имеют равные значения.

```
- {name="Foo", used=true };
> {name="Foo", used=true }: {name:string, used:bool}
- {name="Foo", used=true } = { used=not false, name="Foo"};
> true : bool
```

```
- {name="Bar", used=true} = {name="Foo", used=true};
> false : bool
```

Упорядоченные энки являются частным случаем записей: упорядоченная энка типа σ_1`*`σ_2`*...*`σ_n является сокращенным обозначением для записи типа `{ 1:`σ_1`, 2:`σ_2`,...,n:`σ_n` }`. Например, выражения `(3,4)` и `{1=3, 2=4}` обозначают в точности одно и то же.

На этом завершается наше введение в первичные типы, значения и выражения языка ML. Мы хотим обратить ваше внимание на то, что значения всех рассмотренных типов строятся некоторым единообразным методом. Для каждого типа имеется свой набор форматов выражений, задающих значения этого типа. Для атомарных типов такими выражениями являются *константы* этих типов. Например, константами типа `int` являются последовательности цифр со знаком, а константами типа `string` являются последовательности литер, заключенных в двойные кавычки. Для составных типов значения строятся с помощью *конструкторов значений*, функцией которых является построение значений составного типа из значений компонент. Например, конструктор пары, записываемый как `( , )`, берет два значения и формирует из них значение типа "упорядоченная пара". Аналогично, `nil` и `::` являются конструкторами, которые формируют значение типа "список". Синтаксис записи может также рассматриваться как конструктор. Такой взгляд на данные, как на формируемые из констант с помощью конструкторов, является основополагающим в ML и будет играть важную роль в последующих рассмотрениях.

В ML имеется еще один очень важный тип, тип функций. Однако перед тем, как приступить к рассмотрению функций, мы рассмотрим различные формы *объявлений* и основные виды выражений в ML. Это облегчит последующее изучение функций.

2.3 Идентификаторы, привязки и объявления

В этом разделе мы введем понятие *объявления*; объявления используются в ML для введения идентификаторов. Каждый идентификатор перед использованием должен быть объявлен (имена встроенных функций — таких, как `+` или `size` — считаются объявленными заранее). Идентификаторы в ML могут использоваться многими различными способами, и в соответствии с этими способами существуют различные формы объявлений. В настоящем разделе мы рассмотрим *идентификаторы значений*, или *переменные*. Переменная вводится путем *привязки* идентификатора к значению. Например:

```
- val x = 4*5;
> val x = 20 : int
```

```
- val s = "Abc"^"def";
> val s = "Abcdef" : string
- val pair = (x, s);
> val pair = (20, "Abcdef"): int * string
```

Фраза `val x = 4*5` называется *привязкой к значению*. Чтобы выполнить привязку, ML вычисляет значение выражения справа от знака равенства и устанавливает значение переменной, указанной слева от знака равенства, равным вычисленному значению. В приведенном выше примере x привязывается к целому числу 20. После этого идентификатор x навсегда остается привязанным к этому значению; так, значение `(x,s)` образуется из значений `x` и `s`. Обратите внимание на то, что выдача ML теперь слегка отличается от приводившихся ранее: перед значением переменной печатается "`x =` ". Причиной этого является то, что сразу после того, как идентификатор объявлен, ML печатает его определение (форма определения зависит от сорта идентификатора; пока что мы видели только переменные, для которых определением является значение переменной). Напомним, что когда на верхнем уровне диалога (в ответ на запрос ввода) вводится выражение, то оно вычисляется, и затем печатается его значение вместе с типом. На самом деле здесь ML привязывает к этому значению идентификатор `it`, так что в дальнейших фразах верхнего уровня диалога идентификатор `it` может использоваться для доступа к этому значению[7].

Важно отметить отличие понятия переменной в ML от понятия переменной в других языках программирования. Объявление переменной в ML более похоже на описание `const` в Pascal'е, нежели на описание `var`; в частности, привязка *не является* присваиванием. Когда некоторый идентификатор привязывается к значению, "создается" *новый* идентификатор — и он не имеет никакого отношения к (возможно) ранее объявленному такому же идентификатору. Более того, после того, как идентификатор привязан к значению, нет никакого способа изменить это значение: его значение всегда остается тем, к которому мы привязали его в момент объявления. Если вы незнакомы с функциональными языками программирования, это может показаться несколько странным; подождите, пока мы не рассмотрим примеры программ и не покажем, как это может использоваться. Так как ранее объявленные идентификаторы могут быть привязаны к новым значениям, необходимо уточнить, какая из привязок должна быть использована в данной точке программы. Давайте рассмотрим следующую последовательность привязок:

```
- val x = 17;
```

[7] Таким образом, можно считать, что ввод на верхнем уровне диалога выражения e является сокращенной формой для "`val it =` e". В некоторых реализациях ML этот факт явно отражен — в ответ на ввод выражения печатается "`val it = значение : тип`" (вместо приводимой здесь в примерах выдачи "`значение : тип`"). *(Прим. перев.)*

```
> val x = 17 : int
- val y = x;
> val y = 17 : int
- val x = true;
> val x = true : bool
- val z = x;
> val z = true : bool
```

Вторая привязка `x` “заслоняет” первую (но не оказывает никакого влияния на значение переменной `y`). Всякий раз, когда идентификатор используется в выражении, он считается ссылающимся на текстуально ближайшую объемлющую привязку этого идентификатора. Таким образом, использование `x` в правой части привязки идентификатора `z` ссылается на вторую привязку идентификатора `x`, и поэтому значение `z` есть `true`, а не `17`. Это правило аналогично правилам видимости в языках с блочной структурой. С помощью ключевого слова `and`, используемого в качестве разделителя, можно привязать к значениям сразу несколько идентификаторов:

```
- val x = 17;
> val x = 17 : int
- val x = true and y = x;
> val x = true : bool
  val y = 17 : int
```

Обратите внимание на то, что `y` получает значение `17`, а не `true`! Несколько привязок, соединенных словом `and`, вычисляются параллельно — сначала вычисляются их правые части, а затем идентификаторы, указанные в левых частях, привязываются к полученным значениям.

Для того чтобы продолжить изложение, нам нужно ввести несколько определений. Мы будем говорить, что роль объявления состоит в определении идентификатора для дальнейшего использования в программе. Существуют различные возможности использования идентификатора в программе; одним из них является использование в качестве переменной. Чтобы определить идентификатор для какой-нибудь цели, необходимо использовать привязку, соответствующую этой цели. Например, чтобы определить идентификатор как переменную, следует использовать привязку к значению (которая привязывает переменную к значению и устанавливает ее тип). Другие формы привязок будут введены позже. В общем случае, роль объявления состоит в построении некоторой *среды*, которая определяет смысл всех объявленных идентификаторов. Например, после выполнения приведенных выше привязок к значениям среда содержит информацию о том, что значение `x` есть `true` и значение `y` есть `17`. Вычисление выражения всегда происходит в некоторой среде; в упомянутой среде значением выражения `x` будет `true`.

Точно так же, как выражения могут быть соединены друг с другом операциями (как, например, сложение или образование упорядоченной пары) с целью получить новые выражения, объявления могут быть соединены с другими объявлениями. Результатом составного объявления является среда, которая определяется средами, порождаемыми компонентами составного объявления. Один из способов соединения объявлений мы уже видели: точка с запятой позволяет строить *последовательную композицию сред*[8].

```
- val x = 17; val x = true and y = x;
> val x = 17: int
> val x = true : bool
  val y = 17: int
```

Когда два объявления соединяются точкой с запятой, ML сначала вычисляет левое объявление, порождая среду $\mathcal{E}$, а затем вычисляет правое объявление (в среде $\mathcal{E}$), порождая среду $\mathcal{E}'$. Второе объявление может скрыть какие-то идентификаторы из первого объявления (как показано в примере).

Также полезно иметь *локальные* объявления, роль которых состоит только в том, чтобы облегчить построение других объявлений. Это может быть сделано, например, следующим способом:

```
- local
    val x = 10
  in
    val u = x*x + x*x
    val v = 2*x + (x div 5)
  end;
> val u = 200 : int
  val v = 22 : int
```

Привязка идентификатора `x` является локальной по отношению к привязкам идентификаторов `u` и `v`; `x` доступно в процессе привязки `u` и `v`, но не дальше. Это отражено и в результате объявления: объявлены только `u` и `v`. Имеется также возможность локализовать объявление, используемое при вычисления выражения:

```
- let
    val x = 10
  in
    x*x + 2*x + 1
  end;
> 121: int
```

[8] Точка с запятой по синтаксису необязательна: две последовательные привязки к значению считаются разделенными точкой с запятой

Объявление `x` является локальным и поэтому невидимо за пределами приведенной конструкции. Тело конструкции `let` (располагающееся между ключевыми словами `in` и `end`) вычисляется в среде, полученной в результате вычисления объявлений, расположенных перед `in`. В приведенном примере, расположенное там объявление привязывает идентификатор `x` к значению `10`. В полученной среде значение выражения `x*x+2*x+1` есть `121`; это значение и будет значением всего выражения.

Упражнение 2.3.1 *Какой результат будет напечатан ML-системой в ответ на ввод следующих объявлений (предполагается, что нет никаких других привязок для* `x`*,* `y` *и* `z`*):*

1. `val x=2 and y=x+1;`
2. `val x=1; local val x=2 in val y=x+1 end; val z=x+1;`
3. `let val x=1 in let val x=2 and y=x in x+y end end;`

2.4 Образцы

Как вы, вероятно, заметили, пока что у нас не имеется средств для выделения, например, первого члена упорядоченной пары. Выделение частей составных значений выполняется с помощью *сопоставления с образцом.* Значения составных типов сами являются составными; они строятся из составляющих их значений с помощью конструкторов. Естественно использовать аналогичную конструкцию для разложения их на составляющие значения.

Предположим, что `x` имеет тип `int*bool`. Тогда x является парой, левая компонента которой является целым, а правая — логическим. Мы можем получить значения левой и правой компонент, используя следующую обобщенную форму привязки к значению:

```
- val x = (17, true);
> val x = (17,true) : int*bool
- val (left, right) = x;
> val left = 17 : int
  val right = true : bool
```

Левая часть второй привязки является *образцом.* В общем случае образец строится из переменных и констант с помощью конструкторов. Таким образом, образец есть выражение, возможно, включающее переменные. Отличие от ранее рассматривавшихся выражений состоит в том, что в образце переменные не изображают значения определенные предшествующими привязками, а являются переменными, которые должны быть привязаны к значениям в процессе сопоставления с образцом. В приведенном выше примере `left` и `right` суть новые идентификаторы, которые должны быть привязаны к значениям. Сопоставление с образцом состоит в параллельном разложении на составные части значения `x` и образца, и сопоставлении компонент значения с соответствующими им частями образца. Переменная может быть сопоставлена с любым значением, и тогда идентификатор привязывается к этому значению. Если же образец содержит константу, то она должна совпадать с соответствующей частью значения, с которым выполняется сопоставление. В приведенном выше примере, поскольку `x` является упорядоченной парой, сопоставление завершается успешно; при этом левая компонента `x` привязывается к `left`, а правая — к `right`.

Обратите внимание на то, что простейшим случаем образца является переменная. Таким образом, рассмотренная ранее привязка является частным случаем сопоставления с образцом.

Бессмысленно пытаться сопоставить, например, целое с упорядоченной парой или список с записью. Поэтому любая такая попытка рассматривается как ошибка в согласовании типов, и обнаруживается статически, т.е. в период компиляции. Однако, сопоставление с образцом может потерпеть неудачу и динамически, т.е. во время исполнения программы:

```
- val x = (false, 17);
> val x = (false,17) : bool*int
- val (false, w) = x;
> val w = 17 : int
- val (true, w) = x;
Failure:   Match
```

Обратите внимание на то, что во второй и в третьей привязке образец содержит константу в качестве левого члена упорядоченной пары. Только пара с таким же значением левого члена может быть успешно сопоставлена с таким образцом. Во втором случае это условие выполнено, и сопоставление завершается успешно, привязывая идентификатор `w` к значению `17`. В последнем же случае условие не выполнено; сообщение `Failure: Match` говорит о том, что в период исполнения возникла ошибка при сопоставлении с образцом.

Сопоставление с образцом может быть выполнено для значений имеющих любой из введенных ранее типов. Например, мы можем получить компоненты трехэлементного списка следующим образом:

```
- val lst = [ "Lo", "and", "behold" ];
> val lst = [ "Lo", "and", "behold" ]: string list
- val [x1,x2,x3] = lst
> val x1 = "Lo" : string
  val x2 = "and" : string
  val x3 = "behold" : string
```

Это работает прекрасно — до тех пор, пока мы знаем длину списка. Но как быть в случае непустого списка `lst` произвольной длины? Ясно, что его невозможно разложить на компоненты с помощью одного фиксированного образца! Тем не менее, мы можем разложить `lst`, опираясь *на индуктивное определение списка.*

```
- val lst = [ "Lo", "and", "behold" ];
> val lst = [ "Lo", "and", "behold" ]: string list
- val hd :: tl = lst;
> val hd = "Lo": string
  val tl = ["and","behold"]: string list
```

Здесь `hd` привязывается к значению первого элемента списка `lst` (называемого *головой (head)* списка `lst`) и `tl` привязывается к списку, получающемуся после удаления первого элемента из `lst` (этот список называется *хвостом (tail)* списка `lst`). Типом `hd` является `string`, а типом `tl` — `string list`. Причина этого в том, что конструктор `::` требует в качестве левого аргумента элемент списка, а в качестве правого — список.

Упражнение 2.4.1 *Что произойдет, если мы напишем* `[hd,tl]=lst` *вместо того, что было написание выше? (Подсказка: Замените сокращение* `[hd,tl]` *полной записью.)*

Предположим, что нас интересует только голова списка. Тогда нам незачем приписывать имя хвосту (с единственной целью немедленно его забыть). Чтобы избавиться от необходимости выдумывать имена, которые в дальнейшем все равно не будут использоваться, ML позволяет писать вместо них *универсальный образец*, или *джокер* (обозначаемый знаком "`_`" — подчеркивание), который может быть сопоставлен с любым значением без привязки к нему идентификатора.

```
- val lst = [ "Lo", "and", "behold" ];
> val lst = [ "Lo", "and", "behold" ] : string list
- val hd :: _ = lst;
> val hd = "Lo" : string
```

Сопоставление с образцом может быть применено и для записей, и, как вы можете предположить, делается это с помощью помеченных полей. Следующий пример иллюстрирует сопоставление с образцом для записей:

```
- val r = {name="Foo", used=true};
> val r = {name="Foo",used=true} : {name:string,used:bool}
- val { used=u, name=n } = r;
> val n = "Foo" : string
  val u = true : bool
```

Иногда удобно выполнить частичное сопоставление записи с образцом. Это может быть выполнено с помощью *универсального образца для записей*, как в следующем примере:

```
- val {used=u,...} = r;
> val u = true : bool
```

Существенным ограничением при использовании универсального образца для записей является следующее: полный тип записи должен определяться на этапе компиляции (т.е. полный список имен полей записи и их типов должен определяться по контексту, в который входит образец).

Поскольку выделение одного поля из записи является широко распространенной операцией, для нее предусмотрено специальное обозначение: поле name записи `r` может быть обозначено как `#name r`. На самом деле `#name` является не более чем сокращенным обозначением для функции `fn {name=n,...} => n`, выделяющей поле name из записи. Поэтому, в частности, тип записи должен определяться из контекста, в котором эта функция используется. Например, `fn x => #name x` будет ошибкой, поскольку тип записи `x` не определяется однозначно контекстом. Поскольку упорядоченные энки являются частным случаем записи (именами полей у них являются целые числа от 1 до n), i-тая компонента упорядоченной энки может быть выделена с помощью функции `#i`.

Образцы могут вкладываться друг в друга, как в приводимом ниже примере:

```
- val x = (("foo",true), 17);
> val x = (("foo",true), 17) : (string*bool)*int
- val ((ll,lr), r) = x;
> val ll = "foo" : string
  val lr = true : bool
  val r = 17 : int
```

Иногда бывает удобно ввести “промежуточные” переменные в образце. Например, нам может понадобиться привязать к паре `(ll,rr)` идентификатор l. Это выполняется с помощью *многоуровневых образцов*. Многоуровневый образец получается путем приписывания образца к переменной внутри другого образца, как в следующем примере:

```
- val x = (("foo", true), 17);
> val x = (("foo", true), 17): (string*bool)*int
- val (l as (ll,lr), r) = x;
> val l = ("foo", true): string*bool
  val ll = "foo": string
  val lr = true : bool
  val r = 17 : int
```

Здесь сопоставление с образцом выполняется обычным способом: `l` и `r` привязываются к значениям левой и правой компонент `x`, но дополнительно производится сопоставление привязанного к `l` значения с образцом `(ll,lr)`. Результат выводится как обычно.

Имеется еще одно важное ограничение: любая переменная может входить в образец только один раз. В частности, нельзя задать образец вроде `(x,x)` — который должен был бы быть сопоставим только с симметричными парами. Это ограничение на практике не вызывает трудностей, но упомянуть его необходимо.

Упражнение 2.4.2 *Постройте образец, который привязывал бы переменную* `x` *к значению* `0` *при сопоставлении со следующим выражением (например, если дано выражение* `(true,"hello",0)`*, образцом должно быть* `(_,_,x)`*:*

1. `{a=1, b=0, c=true}`
2. `[~2, ~1, 0, 1, 2]`
3. `[(1,2), (0,1)]`

2.5 Определения функций

Мы уже использовали предопределенные функции, такие, как арифметические операции и операции сравнения. В этом разделе мы рассмотрим *привязки к функциональным значениям*, посредством которых в ML определяются новые функции.

Начнем с нескольких общих замечаний, касающихся понятия функции в ML. Функции используются путем *применения* их к аргументам (мы будем также использовать термин *аппликация*). Синтаксически это записывается как два выражения одно за другим (значением первого выражения должна являться функция, а значением второго — ее аргумент) — как, например, `size "abc"` для вызова функции `size` с аргументом `"abc"`. Все функции являются функциями одного аргумента; при необходимости использовать функции (содержательно) нескольких аргументов, n аргументов функции "упаковываются" в один — упорядоченную энку. Так, например, если функция `append` должна получать два аргумента-списка и возвращать результат-список, применение ее будет иметь вид `append(l1,l2)`: формально функция применяется к одному аргументу, который является упорядоченной парой `(l1,l2)`. Для некоторых функций от двух аргументов (обычно, встроенных) используется специальный синтаксис — так называемая *инфиксная запись*, в которой знак функции записывается между двумя ее аргументами. Например, запись e_1 `+` e_2 в действительности означает "применить функцию `+` к упорядоченной паре (e_1, e_2)". Можно использовать инфиксную запись и для функций, определяемых пользователем, однако мы не будем здесь на этом останавливаться.

Аппликация в ML может иметь более сложную форму, чем в других языках программирования. Причиной этого является следующее: в большинстве языков программирования функция может обозначаться только идентификатором, и поэтому вызов функции всегда имеет вид `f(`e_1`,...,`e_n`)`, где `f` — идентификатор. В ML нет такого ограничения: функция является обычным значением, и может быть получена в результате вычисления выражения. Поэтому в общем случае аппликация в ML имеет вид $e\,e'$. Вычисление такого выражения выполняется следующим образом: сначала вычисляется выражение e, в результате чего получается некоторая функция f; затем вычисляется выражение e', в результате чего получается некоторое значение ν; после этого функция f применяется к значению ν. В простейшем случае, когда выражение e является идентификатором (как, например, `size`), вычисление e является крайне простой операцией: нужно просто взять значение, к которому привязан идентификатор (оно должно быть функцией). Но в общем случае процесс вычисления e может быть сколь угодно сложным. Обратите внимание на то, что правила вычисления аппликации предполагают передачу аргумента *по значению*, поскольку аргумент вычисляется до применения функции.

Каким способом можно гарантировать, что в аппликации $e\,e'$ результатом вычисления выражения e будет функция (а, например, не целое число)? Чтобы ответить на этот вопрос, конечно, следует посмотреть, какой тип имеет e. Функции являются значениями, а каждое значение в ML имеет тип. *Функциональные типы* являются составными типами, членами которых являются функции. Функциональные типы записываются как σ `->` τ (произносится "σ в τ"), где σ и τ — типы. Выражение такого типа имеет в качестве значения функцию, которая может быть применена к аргументу типа σ (и только к аргументу такого типа), и, если ее вычисление завершается успешно, возвращает результат типа τ (к сожалению, в общем случае невозможно определить, завершается ли вычисление функции для любого аргумента или нет). Тип σ называется *типом области определения* функции, а тип τ — *типом области значений*. Аппликация $e\,e'$ допустима тогда и только тогда, когда тип e есть σ`->`τ, а тип e' — σ; тип всего выражения есть τ.

Например:

```
- size;
> size = fn : string -> int
- not;
> not = fn : bool -> bool
- not 3;
Type clash in: not 3
```

```
Looking for a: bool
I have found a: int
```

Тип `size` показывает, что она получает аргумент типа `string` и возвращает результат типа `int` (как мы и могли предполагать). Аналогично, `not` является функцией, получающей логический аргумент и возвращающей логический результат. Поскольку функции не имеют никакого внешнего представления, все они печатаются как `fn`. Применение `not` к `3` вызывает ошибку, поскольку тип области определения `not` есть `bool`, в то время как тип `3` есть `int`.

Поскольку функции являются значениями, мы можем привязать идентификатор к функции, используя обычный механизм привязки идентификатора к значению, введенный в предыдущем разделе. Например:

```
- val len = size;
> vai len = fn : string -> int
- len "abc";
> 3 : int
```

Идентификатор `size` привязан к некоторой (предопределенной) функции типа `string->int`. Приведенная выше привязка выполняется так: извлекается значение идентификатора `size` (которое является функцией), и затем к этому значению привязывается идентификатор `len`. Аппликация `len "abc"` вычисляется путем извлечения функции, к которой привязан идентификатор `len`, вычисления значения `"abc"` (которым является сама эта строка) и применения функции к строке. Результатом является `3`, поскольку функция, к которой в ML привязан идентификатор `size`, возвращает количество литер в строке-аргументе.

Функции являются составными объектами; однако они не являются построенными путем соединения других объектов в том смысле, в каком, например, упорядоченная пара построена из своих компонент. Поэтому их структура недоступна программисту, и, в частности, к функциям неприменимо сопоставление с образцом. Кроме того, невозможно проверить экстенсиональное равенство функций (т.е. выдают ли две функции равные результаты при равных аргументах), поскольку это является алгоритмически неразрешимой задачей. Заметим, что для всех других ранее введенных типов равенство имелось. В дальнейшем мы будем говорить, что тип *допускает проверку на равенство*, если мы можем для любых двух значений этого типа проверить, равны они или нет. Никакой функциональный тип не допускает проверки на равенство, а любой атомарный тип — допускает. Что же можно сказать относительно других составных типов? Напомним, что равенство упорядоченных пар было определено как “покомпонентное”: две упорядоченных пары равны тогда и только тогда, когда равны их левые компоненты и равны их правые компоненты. Таким образом, тип $\sigma*\tau$ допускает проверку на равенство тогда и только тогда, когда оба типа σ и τ допускают проверку на равенство. Аналогичные правила применимы и к типам, построенным другими способами. Грубое, но часто дающее правильный ответ правило состоит в том, что тип, при построении которого использовались функциональные типы, не допускает проверки на равенство (это не всегда верно; когда вы лучше познакомитесь с ML, изучите точное определение допустимости проверки на равенство в [7]).

После приведенных выше замечаний мы готовы перейти к обсуждению способов определения функций пользователем. Синтаксис определения функции во многом похож на используемый в других языках. Приведем несколько примеров:

```
- fun twice x = 2*x;
> val twice = fn : int->int
- twice 4;
> 8 : int
- fun fact x = if x=0 then 1 else x*fact(x-1);
> val fact = fn : int->int
- fact 5;
> 120 : int
- fun plus(x,y) : int = x+y
> val plus = fn : int*int->int
- plus(4,5);
> 9 : int
```

Функции определяются путем *привязки идентификатора к функциональному значению*; эта конструкция начинается ключевым словом `fun`. За именем функции следуют ее параметры, которые задаются образцом. В первых двух примерах параметр является простым образцом, состоящим из одного идентификатора; в третьем примере образец является упорядоченной парой, левая компонента которой есть `x` и правая `y`. Когда выполняется применение определенной пользователем функции, значение аргумента сопоставляется с параметром-образцом — точно так же, как и при привязке к значению; результатом этого сопоставления является некоторая среда, в которой и выполняется вычисление тела функции. Например, в случае функции `twice`, `x` привязывается к аргументу (который должен быть целым числом, поскольку тип функции `twice` есть `int->int`) и затем вычисляется тело функции `twice` (т.е. `2*x`); результатом является `8`. В случае функции `plus` сопоставление с образцом несколько более сложное, поскольку аргумент является упорядоченной парой, — однако это сопоставление ничем не отличается от привязки к значению, рассмотренной в предыдущем разделе: значение аргумента сопоставляется с образцом `(x,y)`, в результате чего `x` и `y` привязываются к значениям. Затем в полученной среде вычисляется значение тела функции, и результат определяется по обычным правилам. "`: int`" в определение `plus` называется *явным ограничением типа*, оно здесь необходимо, поскольку из контекста невозможно определить, о сложении значений какого типа — `int` или `real` — идет речь. Позже мы обсудим явные ограничения типа более подробно.

Упражнение 2.5.1 *Запишите функции* `circumference` *и* `area`*, вычисляющие соответственно длину окружности и площадь круга по радиусу.*

Упражнение 2.5.2 *Запишите функцию, вычисляющую модуль действительного числа.*

Определение функции `fact` иллюстрирует важную особенность определения функций в ML: функции, определяемые с помощью конструкции `fun`, являются *рекурсивными*; это означает, что вхождения идентификатора `fact` в правую часть определения функции `fact` ссылаются на определяемую функцию (а не какое-либо другое значение, к которому мог бы быть привязан идентификатор `fact` в окружающей среде). Таким образом, функция `fact` в процессе вычисления ее тела вызывает саму себя. Обратите внимание на то, что при каждом следующем рекурсивном вызове аргумент функции `fact` становится меньше, что гарантирует, что процесс вычисления функции завершится (если исходное значение аргумента было неотрицательным). Конечно, возможны и определения функций, вычисление которых никогда не завершается. Например:

```
- fun f(x) = f(x);
> val f = fn : 'a->'b
```

Любой вызов `f` приведет к возникновению бесконечного цикла, в котором `f` вызывает себя снова и снова.

Упражнение 2.5.3 *Альтернативный синтаксис для условного выражения может быть определен как:*

```
fun new_if ( A, B, C ) = if A then B else C
```

Объясните, что станет неправильным в определении функции `fact`*, если в нем использовать этот новый вариант условного выражения.*

Теперь мы готовы продолжить построение новых интересных функций и примеров программирования на ML. Рекурсия является ключевым моментом функционального программирования, и поэтому, если вы еще не очень хорошо овладели этим приемом, мы советуем вам внимательно разбирать все приводимые примеры и проводить вычисление рекурсивных функций "вручную".

Пока что мы рассмотрели функции, аргумент-образец которых является простой переменной или упорядоченной парой. Давайте рассмотрим, как мы можем определить функции на списках — и для начала попробуем построить функцию `is_nil`, которая определяет, является ли аргумент пустым списком или нет. Списковый тип имеет два конструктора: `nil` и `::`. Функция, определенная на списках, должна работать независимо от того, является ли список пустым или нет, и поэтому должна быть определена разбором случаев, один из которых есть `nil`, а другой есть `::`. Вот определение функции `is_nil`:

```
- fun is_nil (nil) = true
    | is_nil (_::_) = false;
> is_nil = fn : 'a list -> bool
- is_nil nil;
> true: bool
- is_nil [2,3];
> false : bool
```

Определение `is_nil` отражает структуру списков: функция определяется с помощью двух *предложений* — одно предложение для `nil`, а другое — для `hd::tl`. В определении функции предложения отделяются друг от друга вертикальной чертой.

В общем случае, если тип аргумента определяемой функции имеет более одного конструктора, то определение функции должно содержать по одному предложению на каждый конструктор. Это гарантирует то, что функция может принять любой аргумент данного типа. Такой способ определения функций называется *определением с помощью разбора случаев*, поскольку определение содержит по одному предложению для каждого случая формы аргумента.

Разумеется, определение функции с помощью разбора случаев применимо и для рекурсивных функций. Предположим, мы хотим определить функцию `append`, которая получает в качестве аргумента два списка и возвращает результат, который является списком, полученным путем приписывания второго списка в конец первого. Вот ее определение:

```
- fun append (nil, lst) = lst
    | append (hd :: tl, lst) = hd :: append (tl, lst);
> val append = fn : ('a list * 'a list) -> 'a list
```

Определение рассматривает два случая, один — для пустого списка, и второй — для непустого. Добавление списка `lst` к пустому списку выполняется крайне просто: результатом является список `lst`. В случае непустого первого списка (т.е. имеющего вид `hd::tl`) мы должны добавить список `lst` к `tl`, и результат соединить в список с `hd`.

Упражнение 2.5.4 *Вычислите выражение* `append([1,2],[3])` *вручную, чтобы убедиться в правильности определения append.*

Упражнение 2.5.5 *Что делает следующая функция:*

```
fun r [] = [] | r (h :: t) = append (r(t), [h])
```

Типом функции `append` является полиморфный тип, т.е. тип, чье определение включает переменную типа `'a`. Причиной этого является то, что функция `append` может быть применена к спискам элементов любого типа; единственным ограничением является то, что типы элементов обоих списков-аргументов должны совпадать (и это отражено в структуре типа функции `append`), `append` является примером *полиморфной функции*. Рассмотрим несколько примеров применения `append`:

```
- append ([], [1,2,3]);
> [1,2,3] : int list
- append ([1,2,3], [4,5,6]);
> [1,2,3,4,5,6]: int list
- append ([ "Bowl", "of" ], [ "soup" ]);
> ["Bowl","of","soup"]: string list
```

Обратите внимание на то, что мы применили `append` и к спискам типа `int list`, и к спискам типа `string list`.

В общем случае, ML присваивает выражению наиболее общий тип из возможных. Под "наиболее общим типом" понимается такой тип, в котором отражены все ограничения, вытекающие из внутренней структуры выражения, но не более того. Например, в определении функции `append` первый аргумент сопоставляется с образцами `nil` и `::`, из чего следует, что он должен иметь списковый тип. Тип второго аргумента должен быть также списковым типом с тем же типом элементов списка, поскольку в него могут заноситься элементы из первого списка. Из этих двух условий следует, что тип результата должен совпадать с типом обоих аргументов, и, таким образом, тип функции `append` есть `('a list * 'a list) -> 'a list`.

Вернемся к приведенному выше примеру функции `f(x)`, которая была определена как выдающая результат `f(x)`. Мы видим, что ее тип есть `'a->'b`: поскольку тело функции не накладывает никаких ограничений на аргумент, типом аргумента будет `'a` (что означает произвольный тип). Аналогично нет никаких ограничений на тип результата, и поэтому он есть `'b`. Убедитесь, что не может возникнуть никакой ошибки в согласовании типов при каком угодно использовании `f`, несмотря на то, что `f` имеет самый универсальный тип `'a->'b`.

Привязки к функциональным значениям являются одной из форм объявления, аналогичной привязкам к значениям, рассмотренным в предыдущем разделе (фактически привязка к функциональному значению является специальной формой привязки к значению). Таким образом, к настоящему моменту мы имеем два способа построения объявлений: привязка к значению и привязка к функциональному значению. Из этого, в частности, следует, что функции могут быть определены везде, где может быть определено значение; например, возможны локальные объявления функций. Ниже приводится пример эффективной функции инвертирования списков:

```
- fun reverse lst =
    let fun rev(nil, y) = y
          | rev(hd::tl, y) = rev(tl, hd::y)
    in
      rev(lst, nil)
    end;
> val reverse = fn : 'a list -> 'a list
```

Функция `rev` является локальной; она доступна только внутри конструкции `let`. Обратите внимание на то, что `rev` определена рекурсией по первому аргументу, а `reverse` просто вызывает `rev`, и поэтому для reverse нет необходимости анализировать аргумент.

В пределах объявления функции могут использоваться не только ее параметры и локальные переменные, но и любые переменные, которые доступны в точке объявления функции. Рассмотрим следующий пример:

```
- fun pairwith (x, lst) =
    let fun p y = (x, y)
    in map p lst end;
> val pairwith = fn : 'a * 'b list -> ('a * 'b) list
- val lst = [1,2,3];
> val lst = [1,2,3]: int list
- pairwith ("a", lst);
> [("a", 1), ("a", 2), ("a", 3)] : (string * int) list
```

Локальная функция `p` использует нелокальную (относительно нее) привязку идентификатора `x` — параметр функции `pairwith`. Здесь применяется обычное правило: при ссылке на нелокальный идентификатор используется наиболее близкая объемлющая привязка его к значению. Это в точности то же правило, что и используемое в других языках с блочной структурой, например, в Pascal'е (но оно отличается от правил, применяемых в большинстве реализаций LISP'а).

Упражнение 2.5.6 *Совершенным называется число, которое равно сумме всех своих делителей, включая* 1*, но исключая само число; например,* 6 *— совершенное число, так как* $6 = 1 + 2 + 3$*. Определите предикат (функцию типа* `int->bool`*)* `isperfect`*, проверяющий, является ли его аргумент совершенным числом.*

Выше было подчеркнуто, что функции в ML являются полноправными значениями; они имеют те же права и те же привилегии, что и любые другие значения. В частности, это означает, что функции могут быть переданы в качестве аргумента другим функциям и могут быть выработаны функциями в качестве результата. Функции, какие-либо аргументы или результат которых являются функциями, иногда называют *функциями высших порядков*. Эта терминология подчеркивает, что функции являются существенно более сложными объектами — в отличие, например, от целых чисел (которые называют "объектами первого порядка"). Однако, обращаем ваше внимание на то, что в ML нет никакой принципиальной разницы между, например, функциями, получающими в качестве аргумента число, и функциями, получающими в качестве аргумента другую функцию; поэтому упомянутая терминология может указывать разве что на содержательный способ использования функции.

Рассмотрим сначала функции, которые вырабатывают функции в качестве результата. Пусть `f` — такая функция. Что тогда можно сказать о ее типе? Пусть она имеет один аргумент типа τ и вырабатывает результат типа σ`->`ρ. Тогда тип функции `f` есть τ`->(`σ`->`ρ`)`. Результат применения функции `f` к аргументу типа τ есть функция типа σ`->`ρ, которая может быть применена к аргументу типа σ и выработать результат типа ρ. Такое последовательное применение записывается как `f(`e_1`)(`e_2`)`, или просто `f`$e_1 e_2$. Заметьте, что это не то же самое, что `f(`e_1`,`e_2`)`! `(`e_1`,`e_2`)` есть *один* объект — упорядоченная пара, и `f(`e_1`,`e_2`)` означает "применить функцию `f` к упорядоченной паре `(`e_1`,`e_2`)`", в то время как `f`$e_1 e_2$ означает "применить `f` к e_1, получить функцию и применить ее к e_2". Теперь становится понятным, почему ранее при объяснении понятия применения функции к аргументу мы подчеркивали, что функция *вычисляется*: здесь мы получили пример того, что функция задана не идентификатором, а сложным выражением.

Приведем несколько примеров, проясняющих сказанное:

```
- fun times (x:int) (y:int) = x*y;
> val times = fn : int->(int->int)
- val twice = times 2;
> val twice = fn: int->int
- twice 4;
> 8 : int
- times 3 4;
> 12: int
```

Функция `times` определена как функция, берущая в качестве аргумента целое число и вырабатывающая функцию, берущую в качестве аргумента целое число и вырабатывающую целое число[9]. Идентификатор `twice`

[9]Необходимость ": int" при `x` и `y` будет объяснена далее в разделе 2.6.

привязывается к значению `times 2`. Поскольку `2` имеет тип `int`, функция `times` может быть применена к `2`, и результатом будет объект типа `int->int` — как это и видно из сообщения о типе `twice`. Так как `twice` есть функция, она может быть применена к аргументу — и в нашем примере результат вычисления `twice 4` есть `8` (разумеется!). Наконец, `times` применяется к `3`, и затем результат этого применения применяется к `4`, в результате чего получается `12`. В этом последнем выражении подразумевается следующая расстановка скобок: `(times 3) 4`.

Столь же свободно могут использоваться функции, получающие другие функции в качестве аргументов. Такие функции часто называют *функционалами* или *операторами* (но, опять подчеркиваем, в ML такая терминология может указывать только на содержательное использование функций, а не на какие-то особые их языковые свойства). Классическим примером функции такого рода является функция `map`. Она получает в качестве аргументов функцию и список, и возвращает в качестве результата список, полученный из исходного применением функции-аргумента к каждому его элементу. Тип области определения функции-аргумента должен совпадать с типом элементов списка, но тип ее области значений произволен. Вот ее определение на ML:

```
- fun map f nil = nil
    | map f (hd::tl) = f(hd) :: map f tl;
> val map = fn : ('a->'b) -> ('a list) -> ('b list)
```

Обратите внимание на то, что тип `map` отражает связь между типом области определения функции-аргумента и типом элементов списка-аргумента, а также между типом области значений функции-аргумента и типом элементов списка-результата.

Вот несколько примеров использования функции `map`:

```
- val lst = [1,2,3,4,5];
> val lst = [1,2,3,4,5] : int list
- map twice lst;
> [2,4,6,8,10] : int list
- fun listify x = [x];
> val listify = fn : 'a -> 'a list
- map listify lst;
> [[1], [2], [3], [4], [5]] : int list list
```

Упражнение 2.5.7 *Определите функцию* `powerset`*, которая получает в качестве аргумента множество (представленное списком) и возвращает в качестве результата множество всех его подмножеств.*

Сочетая возможность рассмотрения функций как значений и возможность возвращать в качестве результата функцию, мы можем определить функцию, которая строит композицию двух других функций:

```
- fun compose (f, g) (x) = f(g(x));
> val compose = fn : ('a->'b * 'c->'a) -> ('c->'b)
- val fourtimes = compose (twice, twice);
> val fourtimes = fn : int -> int
- fourtimes 5;
> 20 : int
```

Давайте рассмотрим этот пример внимательно. Функция `compose` получает в качестве аргумента пару функций `(f,g)` и возвращает в качестве результата функцию; эта функция, будучи применена к аргументу `x`, возвращает в качестве результата `f(g(x))`. Поскольку результат есть `f(g(x))`, тип `x` должен быть типом области определения `g`; поскольку `f` применяется к `g(x)`, тип области определения `f` должен совпадать с типом области значений g. Таким образом мы получаем тип `compose`, который был сообщен ML-системой. Функция `fourtimes` получается путем применения `compose` к паре функций `(twice,twice)`. Результатом будет функция, которая, будучи применена к `x`, возвратит `twice(twice(x))`; в нашем случае, когда `x` есть `5`, результатом является `20`.

Теперь, когда вы ближе познакомились с функциями в ML, вы можете заметить на данном этапе определенную асимметрию между функциональными значениями и значениями других типов: у нас пока нет никаких способов записи выражений, вырабатывающих функции непосредственно; единственный способ получения функции — это привязка идентификатора к функциональному значению. Но почему должно требоваться, чтобы *каждая* функция имела имя? В определенных случаях это удобно, но есть также ситуации, в которых удобно использовать *безымянные* функции, или *лямбда-выражения* (последний термин восходит к LISP'у и λ-исчислению). Такие средства имеются в ML, и ниже приводятся примеры их использования:

```
- fun listify x = [x];
> val listify = fn : 'a -> 'a list
- val listify2 = fn x => [x];
> listify2 = fn : 'a -> 'a list
- listify 7;
> [7] : int list
- listify2 7;
> [7] : int list
- (fn x => [x]) (7);
> [7] : int list
- val lst = [1, 2, 3];
> val lst = [1, 2, 3] : int list
- map (fn x => [x], lst );
> [[1], [2], [3]]: int list list
```

Мы начали с определения очень простой функции `listify`, которая преобразует аргумент в одноэлементный список. Функция `listify2` полностью эквивалентна функции `listify` за исключением способа ее определения. Результатом вычисления выражения `fn x => [x]` является функция, которая преобразует аргумент `x` в список `[x]` — точно так же, как это делает функция `listify`. Такое выражение, вырабатывающее функцию, мы можем применять к аргументу непосредственно (как в предпоследнем случае) или передать в качестве аргумента другой функции (как в последнем случае).

Точно так же, как и при использовании `fun`, при использовании `fn` можно применять сопоставление с образцом:

```
- ( fn nil => nil | hd::tl => tl ) ([1,2,3]);
> [2,3] : int list
- (fn nil => nil | hd::tl=>tl)([]);
> nil : int list
```

Описание каждого случая здесь называется *правилом*, а такой способ определения функции — *определением с помощью набора правил*.

Заметим, что безымянная функция не может быть рекурсивной, поскольку нет никакого способа сослаться на нее в процессе определения. Это одна из причин того, почему функции в ML тесно связаны с объявлениями: одна из целей привязки к функциональному значению состоит в том, чтобы ввести имя функции с тем, чтобы это имя могло быть использовано в определении функции.

Упражнение 2.5.8 *Рассмотрим следующую задачу: сколькими способами можно разменять сумму в £*1 *монетами достоинством в* 1, 2, 5, 10, 20 *и* 50 *пенсов. Предположим, что мы ввели некоторый порядок на множестве достоинств монет. Очевидно, что тогда выполняется следующее соотношение:*

Количество способов разменять сумму a используя n типов монет	=	*Количество способов разменять сумму a, используя все типы монет, кроме первого*	+	*Количество способов разменять сумму $a-d$, используя все n типов монет (где d есть достоинство монеты первого типа)*

Это соотношение может быть преобразовано в рекурсивное определение функции, если добавить случаи, описывающие завершение рекурсии. Именно, если $a = 0$, имеется ровно 1 *способ размена; если $a < 0$ или $n = 0$, способов размена нет. Эти замечания позволяют записать следующее определение функции:*

```
fun first_denom 1 = 1
  | first_denom 2 = 2
  | first_denom 3 = 5
```

```
  | first_denom 4 = 10
  | first_denom 5 = 20
  | first_denom 6 = 50;

fun cc(0,_) = 1
  | cc(_,0) = 0
  | cc(amount, kinds) =
      if amount < 0 then 0
      else cc( amount-(first_denom kinds), kinds)
         + cc( amount, (kinds-1));

fun count_change amount = cc(amount,6);
```

Измените этот пример так, чтобы в нем использовался список достоинств монет (вместо функции `first_denom`*).*

Упражнение 2.5.9 *Приведенный выше алгоритм плох в том смысле, что в нем выполняется много излишних вычислений. Можете ли вы предложить более быстрый алгоритм? (Это непростая задача, и вы можете пропустить ее при первом чтении).*

Упражнение 2.5.10 *(Ханойские башни) Имеется три стержня (обозначим их A, B и C). На стержень A надето n дисков разного размера так, что внизу находится наибольший, над ним — чуть меньший, и т.д.; наверху находится самый маленький диск. Требуется перенести диски со стержня A на стержень C по следующим правилам: за один ход можно переложить с одного из стержней верхний диск на другой стержень при условии, что перекладываемый диск меньше диска, на который он кладется. Определите функцию, решающую эту задачу.*

2.6 Полиморфизм и перегрузка

Имеется одна тонкая, но важная деталь, которую нужно знать для понимания реализации полиморфизма в ML. Напомним, что полиморфным типом мы называем тип, в который входят переменные типа (в противоположность мономорфным типам, в которые такие переменные не входят). В предыдущем разделе мы определили полиморфную функцию как функцию, работающую с аргументами различных типов (из некоторого класса) единообразным способом. Ключевая идея состоит в том, что такую функцию “не интересуют” типы значений (или компонент значений); поэтому она работает независимо от этих значений, и, таким образом, допускает различные типы значений. Например, тип функции `append` есть `'a list * 'a list -> 'a list`, что отражает тот факт, что функция `append` не интересуется типом элементов списка; единственное, что требуется, это чтобы элементы обоих списков-аргументов имели одинаковый тип. Тип полиморфной функции всегда есть полиморфный тип; он задает бесконечное семейство типов, состоящее из типов, являющих частными случаями полиморфного типа (т.е. получаемых в результате замены переменных типа на какие-либо конкретные типы). Например, `append` может работать с аргументами типа `int list`, `bool list`, `int*bool list` и так далее до бесконечности. Обратите внимание на то, что полиморфизм не ограничивается функциями: пустой список `nil` является списком элементов любого типа, и поэтому типом `nil` является `'a list`.

Полиморфизм отличается от другого понятия — *перегрузки* (хотя, на первый взгляд, они схожи). Перегрузка связана со способом записи, а не со способом определения функции. Примером перегрузки может служить функция сложения, обозначаемая `+`. Мы записываем сложение целых чисел `2` и `3` как `2+3`, а сложение действительных чисел `2.0` и `3.0` — как `2.0+3.0`. Это может показаться похожим на случаи применения функции `append` к двум спискам целых чисел и двум спискам действительных чисел. Однако схожесть здесь лишь частичная: *одна и та же* функция `append` применяется для соединения списков любого типа, но *алгоритм сложения целых чисел отличается от алгоритма сложения действительных чисел.* (Если вы знакомы с обычным представлением чисел в компьютере, у вас это не вызовет сомнения). Таким образом, один и тот же символ `+` используется для обозначения двух различных функций — а не одной полиморфной функции. В каждом конкретном случае выбор функции, которую следует использовать, зависит от типа аргументов.

В этом причина того, что нельзя просто написать `fun plus(x,y) = x+y`: компилятор должен знать типы `x` и `y`, чтобы определить, какая из двух функций сложения должна быть использована — и поэтому он не допускает такого определения. Способ решения этой проблемы состоит в явном указании типа аргументов функции `plus`; это записывается как `fun plus(x:int, y:int) = x+y`. Интересный факт состоит в том, что, если бы не было перегруженных функций, то никогда бы не возникала необходимость явно указывать типы[10]. Но для того, чтобы обеспечить возможность перегрузки, и для того, чтобы повысить надежность программы, ML позволяет вам явно указывать тип конструкции путем приписывания к нему *ти́пового выражения.* Ниже мы приводим примеры использования явного указания типа:

```
- fun plus(x,y) = x+y;
Unresolvable overloaded identifier: +
- fun plus(x:int,y:int) = x+y;
```

[10]За исключением случая использования частичных образцов, как. например, `fun f{x,...} = x`.

```
> val plus = fn : int*int -> int
- 3 : bool
Type clach in: 3 : bool
Looking for a: bool
I have found a: int
- (plus, true) : (int*int -> int) * bool
> (fn, true): (int*int -> int) * bool
- fun id ( x : 'a ) = x;
> val id = fn : 'a->'a
```

Заметьте, что в программе переменные типа записываются точно так же, как их выводит ML: апостроф, за которым следует идентификатор.

Равенство является интересным промежуточным случаем. Равенство не является полиморфной функцией в том смысле, в каком таковой является функция `append`; однако, в противоположность сложению, равенство определено почти для всех типов. Как упоминалось выше, не все типы допускают проверку на равенство, однако для всех типов, допускающих проверку на равенство, существует функция `=`, которая возвращает `true` или `false` в соответствии с тем, равны или нет сравниваемые значения. Поскольку ML может сам определить, допускает тип проверку на равенство или нет, он позволяет использовать равенство "квазиполиморфным" путем. Для этого вводится новый сорт переменных типа, записываемых как `''a`, которые пробегают множество всех типов, допускающих проверку на равенство. Далее ML следит за тем, требуется или нет, чтобы какой-либо тип допускал проверку на равенство, и отражает этот факт в выведенных типах. Например:

```
- fun member (x, nil) = false
    | member (x, hd::tl) = if x=h then true else member(x,tl);
> val member = fn : ''a * ''a list -> bool
```

Вхождения `''a` в тип функции `member` указывают, что `member` может применяться только к аргументам, допускающим проверку на равенство.

2.7 Определения новых типов

ML обладает расширяемой системой типов. Имеется три формы объявления идентификатора как нового конструктора типа.

Простейшей формой является *прозрачная привязка к типу*, или *введение сокращенного наименования типа*. Конструктор типа (возможно, с параметрами) определяется как сокращение для некоторого (сложного) типового выражения. Все использования такого конструктора типа полностью эквивалентны использованию вместо него исходного выражения.

```
- type intpair = int*int;
> type intpair = int*int
- fun f (x : intpair) = let val (l,r)=x in l end;
> val f = fn : intpair -> int
- f(3,2);
> 3 : int
- type 'a pair = 'a*'a;
> type 'a pair = 'a*'a
- type boolpair = bool pair;
> type boolpair = bool pair
```

Нет никакой разницы между `intpair` и `int*int`, поскольку тип `intpair` определен равным типу `int*int`. Единственная причина, по которой ML печатает `intpair` в типе функции `f` — это то, что этот тип явно был указан при объявлении функции.

Система типов ML может быть также расширена с помощью *рекурсивно определяемых типов* (или, короче, *рекурсивных типов*)[11]. В этой конструкции указывается имя нового типа (возможно, с параметрами) и набор конструкторов значений для построения объектов этого типа. В простейшем случае рекурсивное объявление типа выглядит так:

```
- datatype color = Red | Blue | Yellow;
> type color
  con Red : color
  con Blue : color
  con Yellow : color
- Red;
> Red : color
```

Приведенное объявление привязывает идентификатор `color` к новому типу данных, имеющему конструкторы `Red`, `Blue` и `Yellow`[12]. Этот случай рекурсивного объявления напоминает перечислимые типы в Pascal'е.

Обратите внимание на то, что ML выводит слово `color` без последующего знака равенства, подчеркивая этим, что `color` является новым типом данных: он не совпадает ни с каким ранее определенным типом данных, и поэтому равенство здесь было бы неуместным. Кроме объявления нового типа, приведенная выше конструкция `datatype` объявляет

[11]Этот способ определения новых типов предоставляет многообразные возможности и является, пожалуй, наиболее важным в системе типов ML (да и других функциональных языков). Название *рекурсивный* отражает лишь одну (хотя и наиболее существенную) черту этого способа определения типов. Это название не является повсеместно принятым: так, например, в Haskell'е используется термин *алгебраический*: чаще же всего в англоязычной литературе используется термин *datatype* (просто копирующий вводящее определение ключевое слово) — однако дословный перевод этого термина (*тип данных*) в русском языке не отражает особенностей этого понятия. *(Прим. перев.)*

[12]Конструкторы без аргументов иногда называют *константами*.

также три новых *конструктора значений.* Эти конструкторы выводятся с предшествующим ключевым словом `con` (а не `val`), подчеркивая тем самым, что это конструкторы. Введенные конструкторы могут быть использованы для построения образцов при определении функции разбором случаев. Таким образом, рекурсивное определение типа является достаточно сложным: оно одновременно вводит и новый конструктор типа, и новые конструкторы значений.

Рекурсивные определения типов широко используются в ML. Например, можно считать, что встроенный тип `bool` объявлен как:

```
- datatype bool = true | false;
> type bool
  con true : bool
  con false: bool
```

Функции, аргументы которых имеют типы, введенные пользователем, могут быть определены путем разбора случаев так же, как и в случае исходных типов языка. При этом конструкторы значений могут быть использованы в образцах точно так же, как ранее мы использовали `nil` и `::` при определении функций, работающих со списками. Например:

```
-fun favorite Red = true
   | favorite Blue = false
   | favorite Yellow = false;
> val favorite = fn : color->bool
- val color = Red;
> val color = Red : color
- favorite color;
> true : bool
```

Этот пример также иллюстрирует возможность использования одного и того же идентификатора для нескольких различных целей. Идентификатор `color` используется и как имя определенного выше типа, и как переменная, привязанная к `Red`. Такое двойное использование не запрещено (хотя и не рекомендуется, поскольку может привести к затемнению смысла программы), поскольку компилятор всегда может понять из контекста, должно ли в данном месте находится имя типа или имя переменной.

Определяемые пользователем конструкторы значений могут иметь аргументы:

```
- datatype money = nomoney | coin of int | note of int |
                   check of string*int;
> type money
  con nomoney : money
  con coin : int->money
```

```
  con note: int->money
  con check: string*int->money
- fun amount(nomoney) = 0
    | amount(coin(pence)) = pence
    | amount(note(pounds)) = 100*pounds
    | amount(check(bank,pence)) = pence;
> val amount = fn : money -> int
```

Тип money имеет четыре конструктора, первый из которых является константой, а три остальных имеют аргументы. Функция `amount`, определенная разбором случаев с использованием этих конструкторов, возвращает сумму в пенсах, представленную объектом типа `money`.

А что можно сказать по поводу равенства для определенных пользователем рекурсивных типов? Напомним определение равенства для списков: два списка равны тогда и только тогда, когда они либо оба есть `nil`, либо имеют форму `h::t` и `h'::t'` соответственно, и `h` равно `h'` и `t` равно `t'`. В общем случае, два значения некоторого рекурсивного типа равны, если они "построены одним и тем же способом" (т.е. на верхнем уровне использованы один и тот же конструктор) и соответствующие компоненты равны между собой. Как следствие такого определения равенства, мы получаем, что определенный пользователем рекурсивный тип данных допускает проверку на равенство тогда и только тогда, когда все аргументы всех конструкторов значений имеют типы, допускающие проверку на равенство. В рассмотренном примере тип `money` допускает проверку на равенство, поскольку типы `int` и `string` допускают ее.

```
- nomoney = nomoney;
> true : bool
- nomoney = coin(5);
> false : bool
- coin(5) = coin(2+3);
> true : bool
- check("TSB",500) <> check("Clydesdale",500);
> true : bool
```

В рекурсивном определении типа допускается использование рекурсии[13]. Предположим, что мы хотим определить тип двоичных деревьев. Двоичное дерево есть либо лист, либо вершина, имеющая два двоичных дерева в качестве сыновей. В соответствии с этим записывается тип:

```
- datatype btree = empty | leaf | node of btree*btree;
> type btree
  con empty: btree
  con leaf: btree
```

[13] В конце концов, не зря же мы его так назвали! *(Прим. перев.)*

```
  con node : btree*btree->btree
- fun countleaves (empty) = 0
    | countleaves (leaf) = 1
    | countleaves (node(tree1,tree2)) =
        countleaves(tree1) + countleaves(tree2);
> val countleaves = fn : btree->int
```

Обратите внимание на то, что рекурсивное определение типа `btree` следует неформальному определению двоичного дерева. Функция `countleaves` является рекурсивной функцией, определенной на двоичных деревьях; она подсчитывает количество листьев дерева. Важно отметить, что функция, определенная на рекурсивных данных, является рекурсивной[14]. Определение функции `append`, рассмотренное ранее, демонстрирует этот факт. Это не случайно, поскольку можно считать, что предопределенный тип τ `list` объявлен следующим образом[15]:

```
- datatype 'a list = nil | :: of 'a * 'a list;
> type 'a list
  con nil : 'a list
  con :: : ('a * ('a list)) -> ('a list)
```

Этот пример иллюстрирует заодно использование *параметров* в определении типа: тип `list` получает в качестве параметра другой тип, определяющий тип элементов списка. Этот тип представлен переменной типа `'a`. Мы используем термин "конструктор типа", поскольку `list` строит новый тип из другого типа подобно тому, как конструктор значения строит новое значение из других значений.

Приведем другой пример рекурсивного типа с параметром:

```
- datatype 'a tree = empty | leaf of 'a |
                     node of 'a tree * 'a tree;
> type 'a tree
  con empty : 'a tree
  con leaf: 'a -> 'a tree
  con node : 'a tree * 'a tree -> 'a tree
- fun frontier( empty ) = []
    | frontier( leaf(x) ) = [x]
    | frontier ( node(t1,t2) ) =
        append(frontier(tl), frontier(t2));
```

[14]Это не формальное, а содержательное утверждение. Конечно, мы можем определить функцию с аргументом типа `btree` как `fun f(empty) = 1 | f(leaf) = 2 | f(node(_,_)) = 3`, и здесь нет никакой рекурсии. Однако все содержательно интересные функции, определенные на рекурсивных типах данных, за редчайшим исключением будут рекурсивными. *(Прим. перев.)*

[15]В этом примере мы игнорируем тот факт, что для конструктора `::` используется инфиксная форма записи: это не касается сути рассматриваемого вопроса.

```
> val frontier = fn : 'a tree -> 'a list
- val tree = node(leaf("a"), node(leaf("b"), leaf("c")));
> val tree = node(leaf("a"), node (leaf("b"), leaf("c")))
           : string tree
- frontier tree;
> ["a","b","c"] : string list
```

Функция `frontier` получает в качестве аргумента дерево и возвращает список значений, приписанных листьям дерева-аргумента.

Упражнение 2.7.1 *Определите функцию* `samefrontier(x,y)`*, которая возвращает* `true`*, если деревья* `x` *и* `y` *имеют одни и те же листья и в одном и том же порядке, независимо от внутренней структуры дерева, и* `false` *в противном случае. Правильное, но неудовлетворительное решение будет таким:*

```
fun samefrontier (x, y) = (frontier x) = (frontier y)
```

Это довольно трудное упражнение; основная проблема состоит в том, чтобы избежать полной развертки дерева в случае, когда функция должна выдать `false`*.*

ML также обеспечивает механизм определения *абстрактных типов*[16]. Они вводятся с помощью конструкции `abstype`. Абстрактный тип данных представляет собой некоторый рекурсивный тип данных и набор функций для работы с данными этого типа. Рекурсивный тип данных называется типом реализации абстрактного типа, а набор функций называется его интерфейсом. Тип, определенный с помощью конструкции `abstype`, является абстрактным в том смысле, что конструкторы его типа реализации недоступны программам, использующим этот тип: доступным является только интерфейс. Поскольку программа, использующая абстрактный тип данных, ничего не знает о его типе реализации, она пользуется для работы с данными абстрактного типа только функциями интерфейса. Поэтому реализация абстрактного типа данных может быть изменена в любой момент, и это никак не скажется на программе, использующей абстрактный тип данных. Использование абстрактных типов данных — важный элемент структурного программирования, поскольку он позволяет избежать ненужных связей между компонентами большой программы.

Приведем пример объявления абстрактного типа:

```
- abstype color = blend of int*int*int
  with val white = blend(0,0,0)
```

[16]Более мощным инструментом, однако, оказываются модули, рассматриваемые в следующей главе.

```
       and red = blend(15,0,0)
       and blue = blend(0,15,0)
       and yellow = blend(0,0,15)
       fun mix (parts:int, blend(r,b,y),
                parts':int, blend(x',y',z')) =
           if parts<0 orelse parts'<0 then white
           else let val tp = parts+parts'
                    and rp = (parts*r+parts'*r') div tp
                    and bp = (parts*b+parts'*b') div tp
                    and yp = (parts*y+parts'*y') div tp
                in blend(rp,bp,yp) end;
  end;
> type color
  val white = - : color
  val red = - : color
  val blue = - : color
  val yellow = - : color
  val mix = fn : int*color*int*color->color
- val green = mix(2, yellow, 1, blue);
> val green = - : color
- val black = mix (1, red, 2, mix(1, blue, 1, yellow));
> val black = - : color
```

Мы должны сделать несколько замечаний относительно приведенного примера. То, что идет после слова `abstype`, является рекурсивным определением типа (и здесь используется тот же самый синтаксис). Эта часть есть определение типа реализации. Далее идет описание интерфейса, заключенное в синтаксические скобки `with` и `end`. В выдаче ML мы видим, что после слова `color` нет знака равенства; это отражает тот факт, что `color` — это новый тип, не совпадающий ни с каким другим. Но в отличие от рекурсивного определения типа, в результате выполнения `abstype` не создано никаких конструкторов: это делает невозможным создание новых значений типа `color` иначе, как путем использования значений и функций интерфейса. Это обеспечивает высокую степень изоляции программы, использующей абстрактный тип данных, от его определения. Заметьте то, что функции, определенные внутри `with`, все же *имеют* доступ к типу реализации и его конструкторам — иначе бы этот тип был бесполезным!

Обратите внимание на то, что внешняя программа не может определять функции с аргументами абстрактного типа путем разложения аргумента на составные части (с помощью сопоставления с образцом). Это, в частности, означает, что абстрактный тип не допускает проверку на равенство. Если для абстрактного типа должно быть равенство, оно должно быть явно определено как одна из интерфейсных функций.

Итак, имеется три способа определения новых типов в ML. Прозрачные определения предназначены для сокращенной записи сложных типовых выражений; их задача — повысить читабельность текста программы, а не создание новых типов как таковых. Рекурсивные типы обеспечивают расширение системы типов ML. Объявление рекурсивного типа вводит новый конструктор типа и некоторый набор конструкторов значений этого типа. Рекурсивное определение типа подходит для сложных структур данных (как, например, деревья), внутренняя структура которых должна быть доступна программе. Если же важным является только поведение значений нового типа (как, например, в случае стека или очереди), более подходящим является определение абстрактного типа, структура реализации которого невидима для использующей абстрактный тип программы, и может использоваться только функциями, определяющими поведение данных.

Упражнение 2.7.2 *Абстрактный тип* `set` *может быть определен как:*

```
abstype 'a set = set of 'a list
with val emptyset: 'a set = ...
     fun singleton(e: 'a ): 'a set = ...
     fun union(s1: 'a set, s2: 'a set): 'a set = ...
     fun member(e: 'a, s: 'a set) : bool = ...
       | member(e, set(h::t)) = (e=h)
                                orelse member(e, set t)
     fun intersection (s1: 'a set, s2: 'a set): 'a set = ...
end;
```

Завершите определение этого абстрактного типа данных.

Упражнение 2.7.3 *Модифицируйте свое решение так, чтобы элементы множества хранились в виде упорядоченного списка. (Подсказка: Один из путей решения состоит в том, чтобы передавать отношение порядка (т.е. функцию типа* `'a*'a->bool` *в качестве дополнительного параметра каждой функции. Другой путь состоит в том, чтобы отношение порядка передавать только тем функциям, которые строят множества из исходных элементов — с тем, чтобы они вставляли его в представление множества. Тогда, например, функция построения объединения могла бы извлечь отношение порядка из своих аргументов и использовать его для построения результата. Мы вернемся к этой проблеме позже, когда будем в состоянии предложить более элегантный механизм параметризации такого сорта).*

2.8 Исключения

Предположим, что мы хотим написать функцию `head`, вырабатывающую первый элемент списка-аргумента. Первый элемент непустого списка легко получить путем сопоставления с образцом, но что должна возвращать `head`, если ее аргумент есть `nil`? Ясно, что что-то нужно сделать, — ведь `head` должна быть определена для всех значений аргумента, в том числе и для `nil`, — но что именно нужно делать, не очень понятно. Возвращать какое-либо стандартное значение не очень хорошо по двум причинам: во-первых, непонятно, какое же значение выбрать стандартным, а, во-вторых, это ограничит область определения функции: например, если мы положим `head(nil)` равным `nil`, то функция `head` будет применима только к спискам списков).

Для того, чтобы можно было описать разумное поведение программы в подобных случаях, ML предлагает механизм обработки *исключительных событий*. Целью этого механизма является предоставление функции возможности завершить работу изящным и не нарушающим правила согласования типов способом в тех ситуациях, когда она не может или не хочет возвратить результат. Например, функцию head можно было бы записать следующим способом:

```
- exception Head;
> exception Head
- fun head(nil) = raise Head
    | head(x::y) = x;
> val head = fn : 'a list -> 'a
- head [1,2,3];
> 1 : int
- head nil;
Failure:   Head
```

Первая строка является привязкой к *исключению* (исключительному значению): она объявляет, что идентификатор `Head` является именем исключения. Функция `head` определяется обычным способом путем разбора случаев. В случае непустого списка значение функции `head` есть просто первый элемент списка. Но в случае `nil` функция `head` не в состоянии вернуть какое-либо разумное значение, поэтому она *возбуждает исключение*. Результат этого виден в следующих за определением head строках: в ответ на применение `head` к `nil` выводится сообщение `Failure: Head`, показывающее, что вычисление выражения `head(nil)` привело к тому, что было возбуждено исключение `Head`. Напомним, что попытка деления на `0` приводит к аналогичному результату, и это не случайно: во встроенной функции `div` при попытке деления на `0` возбуждается исключение `Div`.

С помощью конструкций `exception` и `raise` мы можем определять функции, которые сигнализируют о возникновении нежелательных ситуаций путем возбуждения исключений. Но, с другой стороны, нужны еще и какие-то средства для обработки этих исключений. Такая возможность, естественно, есть в ML; соответствующая конструкция называется обработчиком исключений (или просто обработчиком). Мы проиллюстрируем ее использование на следующем простом примере:

```
- fun head2 lst = head(lst) handle Head => 0;
> val head = fn : int list -> int
- head2([1,2,3]);
> 1 : int
- head2(nil);
> 0 : int
```

Выражение вида e `handle` exn `=>` e' вычисляется следующим способом: сначала вычисляется e; если результатом является некоторое значение v, то значением всего выражения является v; если в процессе вычисления e возбуждается исключение exn, то вычисляется выражение e', и его значение будет значением всего выражения; если же возбуждается какое-либо другое исключение, то и все выражение возбуждает это исключение. Заметьте, что типы выражений e и e' должны совпадать — иначе тип всего выражения зависел бы от того, возбудилось ли в процессе вычисления e исключение или нет. Этим объясняется то, что тип функции `head2` есть `int list -> int`, хотя из анализа аргумента `lst` никак не следует, что этот список является списком целых чисел. В приведенном выше примере `head2` пытается применить `head` к `lst`; если эта попытка завершается успешно, т.е. если `head` возвращает значение, то это значение и становится значением `head2`; иначе, т.е. если в процессе вычисления `head(lst)` возбуждается исключение `Head`, в качестве результата возвращается `0`.

Если процессе вычисления выражения может быть возбуждено несколько различных исключений, то их можно перехватить с помощью одного обработчика:

```
- exception Odd;
> exception Odd
- fun foo n = if n mod 2 <> 0 then
                raise Odd
              else 17 div n;
> val foo = fn : int -> int
- fun bar m = foo(m) handle Odd => 0
                          | Div => 9999;
> val bar = fn : int -> int
- foo 0;
```

```
Failure: Div
- bar 0;
> 9999 : int;
-foo 3;
Failure: Odd
- bar 3;
> 0 : int
- foo 20;
> 1 : int
- bar 20;
> 1 : int
```

При вычислении функции `foo` может произойти два исключительных события: деление на `0`, в результате чего возбуждается исключение `Div`, или передача ей нечетного аргумента, в результате чего возбуждается исключена `Odd`. Функция `bar` обрабатывает оба этих исключения: если при вычислении `foo(m)` возбуждается исключение `Odd`, то `bar(m)` возвращает 0; если при вычислении `foo(m)` возбуждается исключение `Div`. то `bar(m)` возвращает `9999`; в противном случае `bar(m)` возвращает значение `foo(m)`.

Обратите внимание на то, что синтаксис обработчика нескольких исключений очень похож на синтаксис определения безымянной функции с помощью набора правил. Действительно, можно рассматривать обработчик исключений как безымянную функцию, тип области определения которой есть `exn` (тип исключительных значений), а тип области значений которой есть тип выражения слева от обработчика. С точки зрения проверки соответствия типов идентификаторы исключений суть не что иное, как конструкторы *исключительных значений* типа `exn` — точно так же, как `nil` и `::` являются конструкторами типа `'a list`.

Более того, исключения могут содержать внутри себя другие значения — для этого достаточно просто объявить их с аргументом подходящего типа. Присоединенное к исключению значение может быть использовано обработчиком исключений. Следующий пример иллюстрирует эту возможность:

```
- exception oddlist of int list and oddstring of string;
> exception oddlist of int list
  exception oddstring of string
- ... handle oddlist(nil) => 0
           | oddlist(h::t) => 17
           | oddstring("") => 0
           | oddstring(s) => size(s)-1
```

Здесь объявление exception вводит два исключения: `oddlist` с аргументом типа `int list`, и `oddstring` с аргументом типа `string`. Обработчик выполняет разбор случаев — сначала определяет, какое исключение было возбуждено, а затем анализирует его аргумент. В этом отношении обработчик устроен так же, как и определение функции, заданной набором правил.

Что произойдет, если обозначенное в примере выше многоточием выражение возбудит исключение, отличное от `oddlist` и `oddstring`? Здесь аналогия с функциями заканчивается: в случае функции, если аргумент не отождествится ни с одним из образцов, возбуждается исключение `Match`; в случае же обработчика исключений, если встречается исключение, не предусмотренное в обработчике, это исключение *возбуждается снова* — в надежде, что какой-либо объемлющий обработчик исключений все же обработает его. Например:

```
- exception Theirs and Mine;
> exception Theirs
  exception Mine
- fun f(x) = if x=0 then raise Mine else raise Theirs;
> val f = fn : int -> 'a
- f(0) handle Mine => 7;
> 7 : int
- f(1) handle Mine => 7;
Failure: Theirs
- (f(1) handle Mine => 7) handle Theirs => 8;
> 8 : int
```

Поскольку исключения в действительности являются значениями типа `exn`, аргументом конструкции `raise` может быть не только идентификатор, но и произвольное выражение (вырабатывающее значение типа `exn`). Например, функция `f` из приведенного выше примера может быть записана как:

```
- fun f(x) = raise (if x=0 then Mine else Theirs);
> val f = fn : int -> 'a
```

Как и при определении функции, в образце в обработчике исключений могут быть использованы *универсальные образцы* (или *джокеры*), которые могут быть сопоставлены с любым исключением. Например, следующее выражение перехватывает любые исключения, и, в этом случае, вырабатывает `0`:

```
... handle _ => 0;
```

Объявление исключения является обычным объявлением, и поэтому может быть локальным. Если обработчик перехватывает некоторое исключение, он должен находиться в области действия соответствующего объявления. Неправильный учет этого обстоятельства может привести к странным (на первый взгляд) сообщениям об ошибках, как, например:

```
- exception Exc;
> exception Exc
- (let exception Exc in raise Exc end) handle Exc => 0;
Failure:    Exc
```

Несмотря на то, что идентификатор `Exc` объявлен на внешнем уровне, внешний обработчик не может распознать исключение, возникшее внутри конструкции `let`, поскольку область видимости этого исключения, лежит внутри конструкции `let`. (Однако обработчик с образцом "`_`" смог бы обработать это исключение).

Упражнение 2.8.1 *Объясните, что неправильного в следующих двух программах:*

1.
```
exception Exn of bool;
fun f x =
  let exception Exn of int
  in if x> 100 then raise Exn x else x+1 end;
f(200) handle Exn true => 500 | Exn false => 1000;
```

2.
```
fun f x =
  let exception Exn in
    if p x then a x else if q x
    then f(b x) handle Exn => c x
    else raise Exn
  end;
f v;
```

Упражнение 2.8.2 *Напишите программу, размещающую n ферзей на шахматной доске размером n∗n так, что ни одна из фигур не нападает на другую.*

Упражнение 2.8.3 *Модифицируйте предыдущую программу так, чтобы она возвращала все решения задачи.*

2.9 Императивные возможности языка

ML разрешает использовать ссылки и присваивания. Ссылки являются указателями в кучу, для которых выполняется проверка согласования типов. Присваивания позволяют изменять значение, на которое указывает ссылка. Тип τ `ref` является типом ссылок на значения типа τ[17]. Функция `ref` типа `'a -> 'a ref` выделяет в куче место для аргумента,

[17] В настоящее время τ может быть только мономорфным типом: однако предполагается в будущем включить в язык один из рассматриваемых ныне способов работы с полиморфными ссылками.

копирует туда аргумент и возвращает в качестве результата ссылку на это место. Функция `!` типа `'a ref -> 'a` вырабатывает то значение, на которое указывает ссылка. Функция `:=` типа `'a ref * 'a -> unit` выполняет операцию присваивания.

```
- val x = ref 0;
> val x = ref(0): int ref
- !x;
> 0 : int
- x := 3;
> (): unit;
- !x;
> 3 : int
```

Все ссылочные типы допускают проверку на равенство. Объекты типа τ `ref` являются адресами в куче, и два таких объекта равны тогда и только тогда, когда они совпадают. Заметьте, что хотя равные ссылки заведомо указывают на одно и то же значение, обратное не всегда верно: на одно и то же значение могут указывать и несколько неравных ссылок!

```
- val x = ref 0;
> val x = ref 0 : int ref
- val y = ref 0;
> val y = ref 0 : int ref
- x = y
> false : bool
- !x = !y;
> true : bool
```

Это соответствует ситуации в языках типа Pascal'я, где можно иметь две различные переменные, содержащие одинаковые значения (в данный момент). Для тех, кто знаком с LISP'ом, заметим, что равенство ссылок в ML соответствует функции `eq`, а не `equal`.

Вместе со ссылками в языке появляются обычные для императивного языка конструкции: композиция последовательных утверждений и итеративное выполнение. В ML утверждения представляются выражениями типа `unit` (тип их выражает идею того, что эти выражения вычисляются ради их побочного эффекта, а не ради значения выражения). С помощью инфиксной операции "`;`" осуществляется последовательное выполнение утверждений, а конструкция `while` e `do` e' обеспечивает итеративное выполнение.

Упражнение 2.9.1 *Следующий абстрактный тип данных может быть использован для создания бесконечного потока значений:*

```
abstype 'a stream = stream of unit -> ('a * 'a stream)
```

```
  with fun next(stream f) = f()
       val mkstream = stream
  end;
```

Если дан некоторый поток `S`*, то* `next S` *возвращает первое значение из* `S` *и поток, состоящий из остальных значений. Это иллюстрируется следующим примером:*

```
- fun natural n = mkstream(fn () => (n, natural(n+1)));
> val natural = fn : int -> int stream
- val s = natural 0;
> val s = - : int stream
- val (first, rest) = next s;
> val first = 0 : int
  val rest = - : int stream
- val (next,_) = next rest;
> val next = 1 : int
```

Напишите функцию, которая возвращает бесконечный список простых чисел в виде потока.

Упражнение 2.9.2 *Приведенная выше реализация потока как абстрактного типа данных может оказаться крайне неэффективной, если одни и те же элементы потока используются многократно. Это происходит потому, что функция* `next` *вычисляет очередной элемент потока всякий раз, когда она вызывается. Повторное вычисление будет бесполезной тратой времени для таких потоков, как, например, простые числа, где возвращаемое значение будет всегда одним и тем же. Измените объявление типа* `Stream` *с использование ссылок так, чтобы устранить эту неэффективность.*

Упражнение 2.9.3 *Измените объявление типа* `stream` *так, чтобы он допускал как конечные, так и бесконечные потоки, используя предикат* `endofstream` *для определения конца потока.*

Глава 3

Модули

3.1 Обзор

Возможность разложения большой программы на некоторое количество относительно независимых модулей с четко определенным интерфейсом является крайне важной при разработке крупных программных продуктов. Модули в ML предназначены для решения этой проблемы.

Многие из современных языков программирования обладают подобными свойствами. К сожалению, в этой области не сложилось единой терминологии. Программные компоненты называются "модулями", "пакетами", "кластерами" и т.д., и т.п. В ML используется термин "структура", как сокращение для "структура среды". Такой выбор терминологии говорит о том, что в ML программный модуль есть инструмент построения среды. Напомним, что среда есть хранилище информации о смысле идентификаторов, объявленных в программе. Например, в результате выполнения объявления `val x=3` в среде появляется запись о том, что идентификатор `x` привязан к значению `3` типа `int`. Таким образом, разбиение программы на модули в ML понимается как разбиение среды на отдельные части, которыми можно манипулировать относительно независимо друг от друга. Мы говорим "относительно", поскольку, если несколько модулей объединяются в одну программу, они должны хоть как-то взаимодействовать между собой. Следовательно, должна быть какая-то возможность описания и организации этого взаимодействия. Это называется проблемой соиспользования (sharing).

То, какой набор операций над программным модулем доступен, и то, какие есть средства управления соиспользованием, являются основными характеристиками любого механизма разбиения на модули. По меньшей мере, должна быть возможность независимой компиляции отдельного программного модуля, возможность связывания скомпилированных модулей в единую программу и возможность изоляции одного модуля от другого с целью избежать нежелательной зависимости между модулями, основанной на “случайных” свойствах модуля — таких, как детали внутренней реализации. Соотношение и взаимосвязь между изоляцией (абстракцией) и соиспользованием есть ключевой момент, определяющий решение других проблем в системе разделения на модули.

Точно так же, как тип идентификатора является посредником, описывающим возможности использования идентификатора в программе, так и структура обладает некоторого рода типом, называемым “сигнатурой”, который описывает то, как видна структура во внешнем мире. В литературе тип программного модуля иногда называется “интерфейсом” или “описанием пакета”. Терминология ML возникает из аналогии между структурой среды и алгебраической структурой — а “тип” последней обычно называется сигнатурой. Точно так же, как тип является “краткой сводкой” свойств объекта, используемых в период компиляции, так и сигнатура является краткой сводкой свойств структуры, используемых в период компиляции. Однако, в противоположность ядру языка, явное приписывание сигнатуры какой-либо структуре влияет на ее свойства и в период компиляции, и в период исполнения, поскольку сигнатура накладывает ограничения на “видимость” содержимого данной структуры.

Функтором называется функция, преобразующая структуры в структуры. Идея функтора состоит в следующем: если какая-либо структура S зависит от структуры T только в той части, которая описана в сигнатуре T, то структура S может быть изолирована от деталей реализации структуры T путем определения некоторой функции, которая по структуре T с заданной сигнатурой вырабатывает структуру S, в которую “вмонтирована” структура T. В литературе это называется “параметризованными модулями” или “генерическими пакетами”. В ML выбран термин “функтор” по той причине, что он, с одной стороны, больше подходит для функционального языка, а, с другой стороны, принят в математической терминологии, использующей термины “структура” и “сигнатура”. Объявление функтора определяет изолированную структуру S, а применение функтора к структуре соответствует операции связывания модулей в единое целое. Функторы являются также основой для элегантного механизма сокрытия информации, называемого *абстракцией*. Во многих случаях абстракция является удачной заменой абстрактных типов данных.

Мы начнем наше введение в модульную структуру ML с рассмотрения структур и сигнатур.

3.2 Структуры и сигнатуры

Структура есть не что иное, как “материализованная” среда, превращенная в объект, которым можно манипулировать. Основным средством определения структур являются *инкапсулированные объявления*, состоящие из последовательности объявлений, заключенных в синтаксические скобки `struct` и `end`. Вот простой пример инкапсулированного объявления:

```
struct
  type t = int;
  val x = 3;
  fun f(x) = if x=0 then 1 else x*f(x-1)
end
```

"Значением" этого инкапсулированного объявления является структура, в которой идентификатор типа `t` привязан к типу `int`, а идентификаторы значений `x` и `f` привязаны соответственно к числу `3` и к функции вычисления факториала. Хотя мы и склонны рассматривать структуры как некоторый сорт значений, их статус отличается от статуса обычных значений. В частности, инкапсулированное объявление нельзя просто написать на верхнем уровне диалога - как, например, можно написать арифметическое выражение. Однако к нему можно привязать идентификатор — но и эта форма объявления, называемая *привязкой к структуре*, может появиться только либо на верхнем уровне диалога, либо внутри инкапсулированного объявления. На время мы ограничим наше внимание только привязками к структурам на верхнем уровне, а привязки внутри инкапсулированного объявления рассмотрим позже. К приведенному выше инкапсулированному объявлению может быть привязан идентификатор следующим способом:

```
- structure S =
    struct
      type t = int;
      val x = 3;
      fun f(x) = if x=0 then 1 else x*f(x-1)
    end;
> structure S =
    struct
      type t = int
      val f = fn : int -> int
      val x = 3 : int
    end
```

Обратите внимание на то, что результат привязки к структуре является средой[1]. ML печатает среду, получающуюся в результате выполнения объявлений между `struct` и `end` почти в той же форме, в какой выводятся сообщения после объявлений на верхнем уровне. Конечно, структура

[1]По техническим причинам некоторые реализации ML переупорядочивают компоненты среды перед выводом на экран.

задает независимую среду в том смысле, что объявления, появившиеся внутри инкапсулированного объявления, никак не влияют на другие объявления, выполненные на верхнем уровне. Так, например, после приведенного выше примера ни `t`, ни `f` недоступны на верхнем уровне.

Однако к идентификаторам, объявленным внутри структуры, может быть получен доступ с помощью *составных (qualified) имен*. Составное имя состоит из *пути доступа к структуре*, точки и самого идентификатора. В настоящий момент путь доступа к структуре будет состоять просто из идентификатора структуры; позже нам придется обобщить понятие пути до последовательности идентификаторов структур. Мы можем использовать компоненты структуры `S` с помощью составных имен следующим образом:

```
- x;
Type checking error in: x
Unbound value identifier: x
- S.x;
> 3 : int
- S.f(S.x);
> 6 : int
- S.x : S.t;
> 3 : S.t
```

Выражение `S.x` является составным именем, которое ссылается на значение, к которому привязан идентификатор `x` в структуре `S`. Значение выражения `S.x`, как вы и можете предположить, есть `3`. Аналогично, `S.f` обозначает функцию `f`, объявленную в структуре `S` (функцию вычисления факториала). Когда она применяется к `S.x` (т.е. к `3`), результатом будет `6`. Использование идентификаторов, объявленных в `S`, не ограничивается значениями: последний пример показывает возможность использования идентификатора типа `S.t`, объявленного в `S` как `int`.

Если в какой-то части программы упоминаются несколько компонент одной и той же структуры, выписывание составных имен превращается в утомительное занятие. Чтобы облегчить жизнь в таких ситуациях, ML предоставляет возможность “открыть” структуру с тем, чтобы к определенным внутри нее идентификаторам стал возможен непосредственный доступ.

```
- let open S in f(x) end;
> 6 : int
- open S;
> val x = 3 : int
  val f = fn : int -> int
  type t = int
```

В первом примере мы локально открываем структуру `S` внутри выражения `let`, что позволяет писать `f(x)` вместо громоздкого `S.f(S.x)`. Во втором примере мы открываем структуру `S` на верхнем уровне, и тем самым добавляем к среде верхнего уровня новые привязки идентификаторов — как это и видно из выдачи ML.

Часто бывает полезно рассматривать структуры как значения некоторого особого рода, поскольку, во-первых, это соответствует представлению о среде как о некотором объекте, а, во-вторых, имеется некоторый набор операций над структурами. В ядре языка каждое значение имеет тип; аналогично, структуры также имеют типы: это сигнатуры. Сигнатуры характеризуют структуры во многом подобно тому, как обычные типы характеризуют обычные значения, описывая способы, которыми значение может быть использовано в процессе вычислений. Хотя, строго говоря, сигнатуры не являются типами, но, тем не менее, аналогия между сигнатурами и типами должна помочь вам понять, для чего нужны сигнатуры.

Если мы рассмотрим выдачу ML в приведенных выше примерах, то увидим некоторые различия между выдачей в случае обычной привязки к значению и в случае привязки к структуре. А именно, при обычной привязке к значению, ML в ответ печатает и значение, и его тип. В случае же привязки к структуре печатается только значение. Давайте рассмотрим, что бы произошло, если бы ML твердо придерживался принципов работы с обычными значениями и в случае структурных значений:

```
- structure S =
    struct
      val x = 2+2;
      val b = (x=4)
    end;
> structure S =
    struct
      val x = 4
      val b = true
    end
  :
    sig
      val x : int
      val b : bool
    end
```

В этом причудливом примере информация о типах переменных появляется внутри сигнатуры, в то время как значения появляются внутри структуры. Это соответствует нашему интуитивному пониманию сигнатуры как характеристики значения (именно, структуры). Ясно, что если бы результат привязки к таким "толстым" объектам как структуры печатался в том же виде, как и результат привязки к обычным значениям, результат печати получался бы слишком громоздким, и поэтому ML-системы печатают результат привязки к структуре как некоторую “смесь” из структуры и ее сигнатуры.

Выражение, заключенное в скобки `sig` и `end` в приведенном выше примере, называется *сигнатурой*; его тело называется *спецификацией*. Спецификация во многом подобна объявлению, с тем отличием, что она только описывает идентификатор, связывая с ним тип, а не придает идентификатору значение (из которого в случае обычного объявления выводится тип). Пока что мы рассмотрели только `val`-спецификации; позднее мы рассмотрим и другие. В приведенном выше примере указывается, что `x` имеет тип `int`, а `b` имеет тип `bool`.

Сигнатуры не только выводятся ML-компилятором. Они играют важную роль в использовании модульной системы, в частности, при объявлении функторов, и поэтому они часто пишутся пользователем. Идентификатор может быть привязан к сигнатуре путем *привязки к сигнатуре* — подобно привязке к типу. Привязка к сигнатуре обозначается ключевым словом `signature`, и может появляться только на верхнем уровне.

```
- signature SIG =
    sig
      val x : int
      val b : bool
    end;
> signature SIG =
    sig
      val x : int
      val b : bool
    end
```

Поскольку информация выдаваемая ML в ответ на объявление сигнатуры не содержит ничего интересного, в последующих примерах она будет опускаться.

Основная польза сигнатур заключается в том, что структуры могут *сопоставляться с сигнатурами*. Структура называется сопоставимой с сигнатурой, если, грубо говоря, структура соответствует спецификациям сигнатуры. Поскольку спецификации подобны типам, идея сопоставления с сигнатурой близка проверке соответствия типов в ядре языка, хотя некоторые детали здесь довольно сложны. Один из способов использования сигнатуры состоит в указании ее после имени структуры при привязке идентификатора к структуре (что очень похоже на указание типа при привязке идентификатора к обычному значению) с целью обеспечения дополнительной проверки в период компиляции:

```
- structure S : SIG =
```

```
    struct
      val x = 2+1
      val b = (x=7)
    end;
> structure S =
    struct
      val x = 3 : int
      val b = false : bool
    end
```

“:SIG” показывает, что инкапсулированное объявление справа от знака равенства должно быть сопоставимо с сигнатурой SIG.

Поскольку, по мнению ML, приведенное выше объявление является приемлемым, указанная структура сопоставима с указанной сигнатурой. Почему это так? Данная структура сопоставима с сигнатурой SIG, поскольку:

1. S.x привязан к 3, что имеет тип int, как и требует сигнатура SIG.
2. S.b привязан к false, что имеет тип bool, как и требует сигнатура SIG.

Короче говоря, если в сигнатуре для переменной x указан тип τ, то в структуре идентификатор x должен привязываться к значению типа τ.

Сигнатура может описывать меньше идентификаторов, чем их представлено в структуре. Например:

```
- structure S : SIG =
    struct
      val x = 2+1
      val b = (x=7)
      val s = "Garbage"
    end;
> structure S =
    struct
      val x = 3 : int
      val b = false : bool
    end
```

Здесь в структуре S объявляются переменные x, b и s, в то время как сигнатура SIG описывает только переменные x и b. В результате не только тип переменной s является несущественным, но и вся переменная *удаляется* из структуры в процессе сопоставления с сигнатурой. Смысл этого состоит в том, что сигнатура SIG определяет *проекцию (view)* структуры, т.е. то, что должно быть видно в структуре; именно, она говорит, что должны быть видны только переменные x и b. Другие сигнатуры могут быть использованы для получения других проекций той же структуры. Например:

```
- structure S =
    struct
      val x = 2+1
      val b = false
      val s = "String"
    end;
> structure S =
    struct
      val x = 3 : int
      val b = false : bool
      val s = "String": string
    end
- signature SIG' =
    sig
      val x : int
      val b : bool
    end
  and SIG" =
    sig
      val b : bool
      val s : string
    end;
- structure S': SIG' = S and S'' : SIG'' = S; >
> structure S' =
    struct
      val x = 3 : int
      val b = false : bool
    end
  structure S'' =
    struct
      val b = false : bool
      val s = "String": string
    end
```

Упражнение 3.2.1 *Сигнатура для структур, которые содержат предикат упорядочения для некоторого типа, может быть записана так:*

```
signature ORD =
  sig
    type t
    val le : t*t -> bool
  end
```

Создайте сопоставимые с этой сигнатурой структуры, которые содержат предикат упорядочения для типов `int` *и* `real*string`.

Если значение в структуре имеет полиморфный тип, то оно будет удовлетворять любой спецификации, которая задает частный случай этого полиморфного типа. Так, например, если `x` привязано в структуре к `nil`, то `x` будет иметь тип `'a list`, и поэтому будет удовлетворять спецификациям, например, `int list` и `bool list list`. Но что будет, если спецификация содержит полиморфный тип? Пусть, например, в спецификации указано, что идентификатор `f` должен иметь тип `'a list -> 'a list`. Чтобы соответствовать такой спецификации, в структуре идентификатор `f` должен быть привязан к функции, получающей в качестве аргумента список любого типа и возвращающей в качестве результата список того же типа. Недостаточно, чтобы `f` имела тип, скажем, `int list -> int list`, поскольку спецификация требует, чтобы `f` была применима к `bool list` и т.д. Общий принцип состоит в том, что структура должна быть *по крайней мере столь же общей*, сколь общей является сигнатура. Таким образом, если в некоторой структуре имя `f` привязано к тождественной функции, которая имеет тип `'a->'a`, то `f` удовлетворяет спецификации, требующей функции типа `'a list -> 'a list`. Причина состоит в том, что `f` может получить аргумент любого и типа и выработать результат того же типа, и *тем более* `f` может получить список некоторого типа и выработать в результате список того же типа. Вот несколько примеров:

```
- signature SIG =
    sig
      val n : 'a list
      val l: int list
      val f: 'a list -> 'a list
    end;
- structure S : SIG =
    struct
      val n = nil    (* : 'a list *)
      val l = nil    (* : 'a list *)
      fun f(x) = x   (* : 'a->'a *)
    end
```

Упражнение 3.2.2 *Что неправильного в следующем объявлении?*

```
structure S : SIG =
  struct
    val n = [3,4]
    val l = nil
    fun f(x) = x
  end
```

Привязка идентификаторов к исключениям в структурах подчинена тем же ограничениям, что и в ядре языка: они должны использовать только мономорфные типы. Спецификация исключения задает только его тип. Правила сопоставления с сигнатурой такие же, как и в случае переменных (за исключением того, что сложностей, связанных с полиморфными типами, здесь не возникает).

```
- structure S =
    struct
      exception Barf
      exception Crap = Barf
      fun f(x) = if x=0 then raise Barf
                 else if x=1 then raise Crap
                 else 7
    end;
> structure S =
    struct
      exception Barf
      exception Crap
      val f = fn : int->int
    end
- S.f(0);
Failure:  Barf
- S.f(4);
> 7 : int
```

Некоторые интересные дополнительные вопросы возникают в связи с объявлениями и спецификациями типов. Во-первых, рассмотрим прозрачное объявление типа в структуре — как, например, в первом примере этого раздела, где идентификатор `t` привязывался к типу `int`. Какова будет сигнатура этой структуры? Давайте рассмотрим, что получилось бы при гипотетическом режиме печати, упоминавшемся ранее:

```
- structure S =
    struct
      type t = int
      val x = 3
      fun f(x) = if x=0 then 1 else x*f(x-1)
    end;
> structure S =
    struct
      type t = int
      val f = fn
      val x = 3
    end
```

```
  :
    sig
      type t
      val f : int->int
      val x : int
    end
```

Спецификация для идентификатора `t` в структуре, к которой привязан `S`, есть просто `type t`, что говорит о том. что “значением” `t` является тип.

Если тип имеет параметр, его спецификация также очевидна:

```
- structure S =
    struct
      type 'a t = 'a * int
      val x = (true,3)
    end;
> structure S =
    struct
      type 'a t = 'a * int
      val x = (true,3)
    end
  :
    sig
      type 'a t
      val x : bool * int
    end
```

Обратите внимание на форму спецификации для `t`.

Обе приведенные выше спецификации допустимы при записи сигнатуры. Но что произойдет при сопоставлении с такой сигнатурой? Рассмотрим следующий пример:

```
- signature SIG =
    sig
      type 'a t
      val x : int * bool
    end;
- structure S : SIG =
    struct
      type 'a t = 'a * bool
      val x = (3,true)
    end;
> structure S =
    struct
```

```
    type 'a t = 'a * bool
    val x = (3,true): int * bool
  end
```

Структура, к которой привязывается `S`, сопоставима с сигнатурой `SIG`, поскольку `S.t` есть унарный (т.е. имеющий один аргумент) конструктор типа — как этого и требует `SIG`.

Если сигнатура задает некоторый конструктор типа, то этот конструктор может использоваться далее в сигнатуре. Например:

```
- signature SIG =
    sig
      type 'a t
      val x : int t
    end;
```

Эта сигнатура определяет класс структур, которые объявляют унарный конструктор типа `t` и переменную типа `int t` (для этого конструктора `t`).

Теперь давайте вернемся к приведенной выше структуре `S` и рассмотрим вопрос, сопоставима она или нет с сигнатурой `SIG`. В соответствии с неформальным описанием `SIG`, которое мы только что дали, она будет сопоставимой. Более строго, `S` будет сопоставима с `SIG`, потому что:

1. `S.t`, как и требуется, есть унарный конструктор типа.

2. Тип `S.x` есть `int*bool`. Поскольку, по определению `S.t`, `int t` равно `int*bool`, `S.x` соответствует спецификации `int t`.

Важно осознавать, что в процессе сопоставления с сигнатурой все идентификаторы типа в сигнатуре заменяются на соответствующие идентификаторы из структуры, и поэтому спецификация `int t` превращается в `int S.t`.

Упражнение 3.2.3 *Какая сигнатура может быть сопоставлена со следующей структурой?*

```
structure S =
  struct
    type 'a t = 'a * int
    val x = (true, 3)
  end
```

При разработке программ полезно придерживаться *правила замкнутости сигнатуры*, которое требует, чтобы свободными идентификатора-ми[2] в сигнатуре были только идентификаторы других сигнатур и идентификаторы встроенных функций (такие, как + или ::)[3].

Упражнение 3.2.4 *Пусть даны структуры:*

```
structure A = struct datatype 'a D = d of 'a end
structure B =
  struct
    type t = int A.D
    fun f(A.d(x)) = A.d(x+1)
  end
```

С какими из следующих сигнатур будет сопоставима структура B*?*

1. `sig type t val f: int A.D -> int A.D end`
2. `sig type t val f: t -> int A.D end`
3. `sig type t val f: t -> t end`

Использование рекурсивных определений типов в структурах не вызывает особых трудностей. Рассмотрим следующий пример:

```
- signature SIG =
    sig
      type 'a List
      val Append : 'a List * 'a List -> 'a List
    end;
- structure S : SIG =
    struct
      datatype 'a List = Nil | Cons of 'a * 'a List
      fun Append (x,Nil) = x
        | Append (x,Cons(h,t)) = Cons(h,Append(x,t))
    end;
> structure S =
    struct
      type 'a List
      val Append = fn : 'a List * 'a List -> 'a List
    end
```

[2] *Свободным* называется идентификатор, который не объявлен в сигнатуре. *(Прим. перев.)*

[3] Подчеркнем, что правило замкнутости сигнатуры является не формальным требованием языка, а рекомендацией, следование которой облегчает построение сложных программ. *(Прим. перев.)*

В качестве упражнения убедитесь, что структура `S` действительно сопоставима с сигнатурой `SIG` (следуйте тем же путем, каким мы шли в приводимых ранее примерах).

В приведенном выше примере сигнатура `SIG`, приписанная структуре `S`, не содержит упоминаний о конструкторах значений типа `List`. Имеется два способа включить эти конструкторы в сигнатуру. Один из них состоит в том, что конструкторы трактуются как обычные значения, как показывает следующий пример:

```
- signature SIG =
    sig
      type 'a List
      val Nil: 'a List
      val Cons ; 'a * 'a List -> 'a List
      val Append : 'a List * 'a List -> 'a List
    end;
- structure S : SIG =
    struct
      datatype 'a List = Nil | Cons of 'a * 'a List
      fun Append(x,Nil) = x
        | Append(x,Cons(h,t)) = Cons(h,Append(x,t))
    end;
> structure S =
    struct
      type 'a List
      val Nil: 'a List
      val Cons : 'a * 'a List -> 'a List
      val Append = fn : 'a List * 'a List -> 'a List
    end
```

Обратите внимание на то, что `'a List` больше не является рекурсивным типом, и что `Nil` и `Cons` являются обычными переменными, а не конструкторами значений.

Другой способ состоит в том, чтобы объявить конструкторы конструкторами, и тем самым сделать видимым устройство типа. Форма спецификации, которая позволяет это сделать, синтаксически идентична объявлению рекурсивного типа:

```
- signature SIG =
    sig
      datatype 'a List = Nil | Cons of 'a * 'a List
      val Append : 'a List * 'a List -> 'a List
    end;
- structure T : SIG = S;
> structure T =
```

```
    struct
      type 'a List
      con Nil : 'a List
      con Cons : 'a * 'a List -> 'a List
      val Append = fn : 'a List * 'a List -> 'a List
    end
```

Полезность этого подхода, при котором мы специфицируем конструкторы, будет прояснена позже, когда мы введем функторы.

Объявления абстрактных типов данных в структурах не привносят ничего нового в сопоставление с сигнатурой, поскольку такое объявление вводит только новый тип и некоторые связанные с ним идентификаторы. Спецификации абстрактных типов не используются, потому что, как мы увидим далее, имеется другое средство абстрагирования типов, и поэтому в таких спецификациях нет нужды.

Упражнение 3.2.5 *Реализуйте стек, используя структуры и сигнатуры.*

На практике структуры обычно строятся на основе других структур в соответствии с принципами, определяемыми решаемой задачей. Если структура `S` построена на основе другой структуры `T`, то говорят, что `S` *зависит от* `T`. Мак-Квин рассматривал две классификации зависимости. Во-первых, зависимость `S` от `T` может быть *существенной* или *несущественной*. Существенная зависимость имеет место тогда, когда `S` не может быть использована без `T` — связь между структурами настолько тесная, что раздельное их использование является бессмысленным. Любые другие формы зависимости являются несущественными. Во-вторых, зависимость `S` от `T` может быть *явной* или *неявной*. Зависимость `S` от `T` будет явной, если сигнатура `S` может быть записана только со ссылками на сигнатуру `T`; в противном случае зависимость является неявной. Заметьте, что явная зависимость всегда является существенной.

Простейший случай несущественной зависимости возникает, если `S` импортирует некоторые значения из `T`, как в следующем примере:

```
- structure T =
    struct
      val x = 7
    end;
> structure T =
    struct
      val x = 7 : int
    end
- structure S =
    struct
```

```
      val y = T.x+1
    end;
> structure S =
    struct
      val y = 8 : int
    end
```

Ясно, что S может быть использована независимо от T, хотя S и определена с помощью ссылки на T. Эта форма зависимости иногда называется *зависимостью по построению.*

Существенная зависимость является гораздо более важной. Одна из форм существенной зависимости возникает тогда, когда T объявляет исключения, которые могут возбуждаться функциями, принадлежащими S. Например:

```
- structure T =
    struct
      exception Barf
      fun foo(x) = if x=0 then raise Barf else 3 div x
    end;
> structure T =
    struct
      exception Barf
      val foo(x) = fn :int -> int
    end
- structure S =
    struct
      fun g(x) = T.foo(x)+1
    end
```

Поскольку вычисление S.g(0) возбуждает исключение Barf, использование S возможно только в том контексте, в котором доступна T — иначе исключение не сможет быть обработано. Поэтому S существенно зависит от T, и должна использоваться только вместе с T. Заметьте, однако, что зависимость является неявной, поскольку сигнатура S не содержит ссылок на T.

Существенная и явная зависимость возникает тогда, когда S открыто использует рекурсивный тип, определенный в T, как например:

```
- structure T =
    struct
      datatype 'a List = Nil | Cons of 'a * 'a List
      fun len(Nil) = 0
        | len(Cons(h,t)) = 1 + len(t)
    end;
```

```
> structure T =
    struct
      type 'a List
      con Nil : 'a List
      con Cons : 'a * 'a List -> 'a List
      val len = fn : 'a List -> int
    end;
- structure S =
    struct
      val len = T.len
    end;
> structure S =
    struct
      val len = fn : 'a T.List -> int
    end
```

Заметьте, что сигнатура структуры `S` содержит ссылку на структуру `T`, отражая тот факт, что `len` может быть применена только к значениям типа, определенного в `T`. Заметьте, что правило замкнутости сигнатуры не позволяет в приведенном выше примере приписать структуре `S` какую-либо нетривиальную сигнатуру, поскольку сигнатура не может содержать свободный идентификатор структуры `T`. Это на первый взгляд может показаться неоправданным ограничением; однако этом позволяет привлечь внимание к тому факту, что `S` и `T` тесно связаны между собой, и поэтому должны быть объединены в один модуль. Такое объединение может быть выполнено путем включения структуры `T` в структуру `S` в качестве *подструктуры*; последнее достигается с помощью включения объявления `T` в набор инкапсулированных объявлений `S`, как это показано в следующем примере:

```
- structure S =
    struct
      structure T =
        struct
          datatype 'a List = Nil | Cons of 'a * 'a List
          fun len(Nil) = 0 | len (Cons(h,t)) = 1 + len(t)
        end
      val len = T.len
    end
> structure S =
    struct
      structure T =
        struct
          type 'a List
          con Nil : 'a List
```

```
        con Cons : 'a * 'a List -> 'a List
        val len = fn : 'a List -> int
      end
    val len = fn : 'a T.List -> int
  end
```

Таким путем можно иерархически организовать взаимосвязанные структуры, и объединить набор связанных структур в модуль.

Появление подструктур требует обобщения введенного ранее понятия пути доступа к структуре. В общем случае путь доступа к структуре записывается как разделенная точками последовательность имен структур, в которой каждая последующая структура является подструктурой предыдущей. Например, `S.T` является путем доступа к структуре, а `S.T.len` является составным именем, которое выбирает функцию `len` подструктуры `T` структуры `S`.

Если `T` является подструктурой структуры `S`, то сигнатура `S` будет выглядеть следующим образом:

```
- signature SIGT =
    sig
      datatype 'a List = Nil | Cons of 'a * 'a List
      val len : 'a List -> int
    end;
- signature SIGS =
    sig
      structure T: SIGT
      val len : 'a T.List -> int
    end;
```

Обратите внимание на спецификацию в `SIGS`, которая указывает, что подструктура `T` должна быть сопоставима с сигнатурой `SIGT`. Заметьте также то, что спецификация для `len` в `SIGS` содержит `T.List`; `T.List` является локальным в `SIGS` благодаря тому, что `T` есть подструктура `S`.

Упражнение 3.2.6 *Определите структуру* `Exp`*, которая реализует некоторый рекурсивный тип выражений и набор связанных с ними операций. Эта структура должна быть сопоставимой с сигнатурой*

```
signature EXP =
  sig
    datatype id = Id of string
    datatype exp = Var of id
                 | App of id * (exp list)
  end
```

Определите другую сигнатуру `SUBST` *и структуру* `Subst`*, которая реализует операцию подстановки для этих выражений (т. е. определите тип* `Subst` *как список пар "идентификатор/выражение", и функцию подстановки, которая по выражению и подстановке (значению типа* `subst`*) строит выражение, получающееся из исходного путем замены описанных в подстановке идентификаторов на соответствующие выражения).*

3.3 Абстракция

Ранее мы отметили, что процесс сопоставления структуры с сигнатурой "обрезает" структуру так, что в ней остаются только компоненты, присутствующие в сигнатуре. Приписывание сигнатуры структуре создает некоторую "проекцию" этой структуры, и, таким образом, сопоставление с сигнатурой обеспечивает некоторый ограниченный способ "сокрытия информации", ограничивая доступ к структуре только теми компонентами, которые имеются в сигнатуре. Одна из причин формирования таких ограничений состоит в том, что при построении сложных программных систем полезно иметь возможность точного описания интерфейса каждого программного модуля. То же самое может быть указано как одна из причин использования абстрактных типов данных в ядре языка: это позволяет сделать всех пользователей данного абстрактного типа данных независимыми от деталей его реализации. Сопоставление с сигнатурой может обеспечить некоторые из возможностей, предоставляемых абстрактными типами данных, поскольку с помощью него возможно "убрать" конструкторы рекурсивных типов, и, таким образом, спрятать внутреннее представление. Но это является одним из частных случаев более общего способа сокрытия информации в ML, называемого *абстракцией.*

Фундаментальная идея состоит в том, что при некоторых обстоятельствах нам хотелось бы ограничить то, что видно из структуры, в точности тем, что указано в сигнатуре. Это может быть проиллюстрировано следующим примером:

```
- signature SIG =
    sig
      type t
      val x : t -> t
    end;
- structure S : SIG =
    struct
      type t = int
      val x = fn x => x
    end;
```

```
> structure S =
    struct
      type t = int
      val x = fn : t -> t
    end
- S.x(3);
> 3 : int
- S.x(3) : S.t;
> 3 : int : S.t
```

Обратите внимание на то, что `S.t` есть `int`, хотя сигнатура `SIG` ни о чем таком не говорит.

Цель абстракции состоит в том, чтобы скрыть всю информацию о структуре, которая не упоминается явно в сигнатуре.

```
- abstraction S : SIG =
    struct
      type t = int
      val x = fn x => x
  end;
> abstraction S : SIG
- S.x(3);
> 3 : int
- S.x(3) : S.t;
Type error in: S.x(3) : S.t
Looking for a: int
I have found a: S.t
```

Эффект объявления абстракции состоит в ограничении всей доступной об `S` информации только той информацией, которая указана в `SIG`.

Имеется тесная связь между абстракцией и абстрактными типами данных. Рассмотрим следующий абстрактный тип:

```
- abstype 'a set = set of 'a list
  with
    val empty_set = set([])
    fun union(set(l1),set(l2)) = set(l1@l2)
  end;
> type 'a set
  val empty_set = - : 'a set
  val union = fn : 'a set * 'a set -> 'a set
- empty_set;
> - : 'a set
```

Это объявление определяет тип `'a set` с операциями `empty_set` и `union`. Конструктор `set` спрятан для того, чтобы можно было ручаться, что тип является абстрактным (т.е. что ни одна программа, использующая этот тип, не окажется зависимой от его представления).

В общем случае объявление `abstype` определяет тип и набор операций над данными этого типа, скрывая при этом тип реализации. Абстракция предлагает другой путь решения этой же задачи, что можно увидеть из следующего примера:

```
- signature SET =
    sig
      type 'a set
      val emty_set: 'a set
      val union : 'a set * 'a set -> 'a set
    end;
- abstraction Set: SET =
    struct
      datatype 'a set = set of 'a list
      val empty_set = set([])
      fun union(set(l1),set(l2)) = set(l1@l2)
    end;
> abstraction Set : SET
- Set.set;
Undefined variable Set.set
- S.empty_set;
> - : 'a S.set
```

Упражнение 3.3.1 *Определите абстракцию для комплексных чисел, используя следующую сигнатуру:*

```
signature COMPLEX =
  sig
    type complex
    exception divide: unit
    val rectangular: {real: real, imag : real} -> complex
    val plus : complex * complex -> complex
    val minus : complex * complex -> complex
    val times : complex * complex -> complex
    val divide : complex * complex -> complex
    val eq : complex * complex -> bool
    val real_part : complex -> real
    val imag_part: complex -> real
  end
```

Подсказка: используйте следующие формулы для реализации операций над комплексными числами:

$$(a + ib) + (c + id) = (a + c) + i(b + d)$$

$$(a+ib)-(c+id)=(a-c)+i(b-d)$$
$$(a+ib)*(c+id)=(ac-bd)+i(ad+bc)$$
$$(a+ib)/(c+id)=\frac{(ac+bd)+i(bc-ad)}{c^2+d^2}$$

Абстракция является более гибкой по сравнению с абстрактными типами данных в одних случаях и менее гибкой в других. Большая гибкость проистекает из того, что абстракция не обязана быть “типом данных с операциями”, как это вытекает из самой формы абстрактных типов. Например, может быть не объявлено ни одного типа вообще, а объявленные типы не обязаны быть рекурсивными типами. Абстрактные типы данных несколько более гибки потому, что они, являясь обычной формой объявления, могут появляться везде, где могут появляться объявления, — в то время как абстракции могут появляться лишь там, где могут появляться структуры: на верхнем уровне или в инкапсулированных объявлениях. Это ограничение не кажется существенным, поскольку обычно типы определяются на верхнем уровне[4].

3.4 Функторы

ML-программа является иерархически организованным собранием взаимосвязанных структур. Функторы (которые являются функциями над структурами) используются для упорядочения процесса разработки программ. В некотором смысле роль функторов аналогична роли связывающего загрузчика в других языках программирования: они являются инструментом, позволяющим из отдельных частей собрать готовую программу. Функторы определяются путем *привязки к функторам*; эта конструкция может появляться только на верхнем уровне. Синтаксис привязки к функтору похож на привязку к функции в ядре языка. Приведем пример:

```
- signature SIG =
    sig
      type t
      val eq : t*t -> bool
    end;
- functor F( P : SIG ) : SIG =
    struct
      type t = P.t * P.t
      fun eq((x,y),(u,v)) = P.eq(x,u) andalso P.eq(y,v)
```

[4] Мы рекомендуем при программировании на ML избегать абстрактных типов данных, поскольку в дальнейшем они могут быть устранены из языка (так как абстракция достаточна для их замены).

```
    end;
> functor F( P : SIG ) : SIG
```

Сигнатура SIG определяет тип t и бинарное отношение eq. Функтор F определяет функцию, которая, получив структуру, сопоставимую с сигнатурой SIG, вырабатывает другую структуру (которая в данном примере также должна быть сопоставима с сигнатурой SIG — однако, разумеется, в других случаях сигнатура структуры-результата не обязана совпадать с сигнатурой структуры-параметра).

Функторы применяются к структурам и вырабатывают другие структуры.

```
- structure S : SIG =
    struct
      type t = int
      val eq : t*t->bool = op =
    end;
> structure S =
    struct
      type t = int
      val eq = fn : t*t->bool
    end
- structure SS : SIG = F(S);
> structure SS =
    struct
      type t = int*int
      val eq = fn : t*t->bool
    end
```

Здесь мы создали структуру S, сопоставимую с сигнатурой SIG. Функтор F, применяемый к структуре S, строит новую структуру с той же сигнатурой — но в которой t уже является типом упорядоченных пар целых чисел, а функция равенства определена на этих парах. Обратите внимание на то, как SS строится из S с помощью функтора F.

Функторы в высокой степени обладают полиморфизмом, что объясняется тем, что сопоставляемая с сигнатурой структура может содержать больше информации, чем этого требует сигнатура. Например:

```
- structure T : SIG =
    struct
      type t = string * int
      val eq : t*t->bool = op =
      fun f(x:t) = (x,x)
    end;
> structure T =
```

```
    struct
      type t = string * int
      val eq : t*t->bool
  end
- structure TT : SIG = F(T);
> structure TT =
    struct
      type t = (string*int)*(string*int)
      val eq : t*t->bool
    end
```

Хотя и имеется ограничение, что функтор должен иметь ровно один аргумент, оно не является существенным: при необходимости несколько структур могут быть включены в одну как подструктуры, и затем эта структура может быть передана функтору. На практике это обычно не создает никаких неудобств, поскольку в тех случаях, когда несколько структур должны быть переданы в качестве аргумента функтору, они, как правило, настолько тесно связаны между собой, что имеется много других причин, по которым их разумно объединить в одну структуру. Функторы должны подчиняться тому же (рекомендательному) правилу замкнутости, что и сигнатуры: они не должны содержать открытых ссылок на значения, типы и исключения во внешней среде (за исключением предопределенных системных примитивов). В теле функтора без всяких ограничений могут использоваться ссылки на параметр и его компоненты (с использованием уточняющих имен), на локальные идентификаторы и на ранее определенные функторы и сигнатуры.

Тело функтора не обязано представлять из себя инкапсулированные объявления (хотя, вероятно, это наиболее распространенный случай). В теле функтора могут свободно использоваться составные имена и аппликации функторов (однако важный момент: функтор *не может* быть рекурсивным!). Приведем примеры:

```
- functor G( P : SIG ) : SIG = F(F(P));
> functor G( P : SIG ) : SIG
- functor I( P : SIG ) : SIG = P;
> functor I( P : SIG ) : SIG
```

Нужно отметить, что функтор `I` не является тождественным: если `S` есть структура, сопоставимая с сигнатурой `SIG`, но с большим количеством компонент, чем упомянуто в сигнатуре, то `F(S)` будет "урезанной" по сравнению с `S`. Например:

```
- structure S =
    struct
      type t = int
```

```
      val eq = op =
      fun f(x) = x
  end;
> structure S =
    struct
      type t = int
      val eq = fn : int*int -> bool
      val f = fn : 'a -> 'a
  end
- structure S' = I(S);
> structure S' =
    struct
      type t = int
      val eq = fn : int*int -> bool
    end
```

Обратите внимание на то, что компонента `f` структуры `S` отсутствует в `I(S)`.

Упражнение 3.4.1 *Преобразуйте вашу реализацию множеств на основе упорядоченных списков в форму, в которой функции равенства и порядка передаются в качестве аргументов функтору, строящему структуру для работы с множествами.*

Этим завершается наше введение в модульную систему ML. Нам осталось обсудить одну важную идею — соиспользование. Это мы обсудим несколько позже, после того, как рассмотрим примеры использования модулей в программировании.

3.5 Модульная система в реальной практике

В этом разделе мы проиллюстрируем возможности использования модульной системы для построения программ. Мы рассмотрим (в общих чертах) разработку синтаксического анализатора, преобразующего входную строку в абстрактное дерево синтаксического разбора и заносящего некоторую информацию о символах во входной строке в таблицу символов. Программа будет состоять из четырех модулей: один будет содержать собственно синтаксический анализатор, второй — процедуры работы с деревом синтаксического разбора, третий — процедуры работы с таблицей символов и четвертый — сканер. Вот сигнатуры этих модулей:

```
signature SYMBOL =
  sig
    type symbol
    val mksymbol : string -> symbol
```

```
    val eqsymbol : symbol*symbol -> bool
  end;
signature ABSTSYNTAX =
  sig
    structure Symbol : SYMBOL
    type term
    val idname : term -> Symbol.symbol
  end;
signature SYMBOLTABLE =
  sig
    structure Symbol : SYMBOL
    type entry type table
    val mktable : unit -> table
    val lookup : Symbol.symbol * table -> entry
  end;
signature PARSER =
  sig
    structure AbstSyntax : ABSTSYNTAX
    structure SimbolTable : SYMBOLTABLE
    val symtable : Symboltable.table
    val parse : string -> AbstSyntax.term
  end;
```

Разумеется, эти сигнатуры идеализированы и предельно сокращены, но все же они достаточно правдоподобны, чтобы служить в качестве убедительного и информативного примера. Обратите внимание на иерархическую организацию этих структур. Поскольку собственно синтаксический анализатор существенно использует функции работы и с деревом синтаксического разбора, и с таблицей символов, соответствующие структуры должны быть явно включены в него как подструктуры. Аналогично, и модуль работы с деревом синтаксического разбора, и модуль доступа к таблице символов используют сканер как подструктуру.

Теперь давайте посмотрим, как на основе этого мы можем построить синтаксический анализатор. Отложив на время вопросы выбора алгоритма и представления, мы можем написать следующие структуры:

```
structure Symbol: SYMBOL =
  struct
    datatype symbol = symbol of string * ...
    fun mksymbol(s) = symbol(s, ...)
    fun eqsymbol(syml,sym2) = ...
  end;
structure AbstSyntax : ABSTSYNTAX =
  struct
    structure Symbol : SYMBOL = Symbol
```

```
    datatype term = ...
    fun idname(term) = ...
  end;
structure SymbolTable : SYMBOLTABLE =
  struct
    structure Symbol: SYMBOL = Symbol
    type entry = ...
    type table = ...
    fun mktable() = ...
    fun lookup(sym, table) = ...
  end;
structure Parser: PARSER =
  struct
    structure AbstSyntax : ABSTSYNTAX = AbstSyntax
    structure SymbolTable : SYMBOLTABLE = SymbolTable
    val symtable = SymbolTable.mktable();
    fun parse(str) =
      ... SymbolTable.lookup(AbstSyntax.idname(t),symtable)...
end;
```

Обратите внимание на то, что в последней строке структуры `Parser` мы записали применение функции `SymbolTable.lookup` к результату функции `AbstSyntax.idname`. Это применение корректно с точки зрения согласования типов только благодаря тому, что структуры `AbstSyntax` и `SymbolTable` включают *одну и ту же* структуру `Symbol`. Если бы было две структуры, сопоставимые с сигнатурой `SYMBOL`, и одна из них была использована в структуре `SymbolTable`, а другая — в структуре `AbstSyntax`, в упомянутой строке возникла бы ошибка несоответствия типов. Имейте это в виду при чтении дальнейшего.

Организация нашего синтаксического анализатора пока кажется вполне удовлетворительной — по крайней мере до тех пор, пока мы рассматриваем статическую структуру программы. Но если мы предположим, что имеются еще многочисленные структуры, и каждая из них содержит несколько тысяч строк кода, то тогда предложенная структура окажется не совсем удобной. Предположим, например, что мы нашли ошибку в структуре `SymbolTable` и исправили ее. Теперь нам нужно собрать синтаксический анализатор заново. Это потребует перекомпиляции всех приведенных выше структур (а также, возможно, и других связанных с ними). Ясно, что нужна какая-то возможность раздельной компиляции и последующего связывания скомпилированных модулей. То, что нам нужно — это возможность отдельно скомпилировать один модуль и затем связать модули в единую программу. Эта идея, разумеется, не нова; новым, однако, является то, как эта проблема решается в ML.

Ключевая идея состоит в том, чтобы никогда не записывать ссылки на структуры явно, а вместо этого организовывать программу в виде набора функторов, каждый из которых получает в качестве аргумента структуры, от которых он зависит (или ничего, если функтор ни от чего не зависит). После этого редактирование связей будет состоять в применении функторов. В нашем случае функторы могут выглядеть следующим образом:

```
functor SymbolFun() : SYMBOL =
  struct
    datatype symbol = symbol of string * ...
    fun mksymbol(s) = symbol(s,...)
    fun eqsymbol(syml,sym2) = ...
  end;
functor AbstSyntaxFun( Symbol: SYMBOL ): ABSTSYNTAX =
  struct
    structure Symbol: SYMBOL = Symbol
    datatype term =... fun idname(term) = ...
  end;
functor SymbolTableFun( Symbol : SYMBOL ): SYMBOLTABLE =
  struct
    structure Symbol: SYMBOL = Symbol
    type entry = ...
    type table = ...
    fun mktable() = ...
    fun lookup(sym,table) = ...
  end;
signature PARSER_PIECES =
  sig
    structure SymbolTable : SYMBOLTABLE
    structure AbstSyntax : ABSTSYNTAX
  end;
functor ParserFun( Pieces : PARSER_PIECES ): PARSER =
  struct
    structure AbstSyntax : ABSTSYNTAX = Pieces.AbstSyntax
    structure SymbolTable : SYMBOLTALBLE = Pieces.SymbolTable
    val symtable = SymbolTable.mktable();
    fun parse(str) =
    ... SymbolTable.lookup( AbstSyntax.idname(t), symtable )...
  end;
```

Сигнатура PARSER_PIECES содержит две компоненты, от которых зависит синтаксический анализатор, — таблицу символов и дерево синтаксического разбора. Функтор ParserFun использует эту пару для построения синтаксического анализатора. Функтор SymbolFun не имеет аргументов, поскольку он не зависит ни от чего.

Программа строится из этих функторов с помощью следующей последовательности объявлений. Убедитесь, что результат будет тем же, что и ранее.

```
structure Symbol: SYMBOL = SymbolFun();
structure Pieces: PARCER_PIECES =
  struct
    structure SymbolTable: SYMBOLTABLE = SymbolTableFun(Symbol)
    structure AbstSyntax: ABSTSYNTAX = AbstSyntaxFun(Symbol)
  end;
structure Parser: PARSER = ParserFun( Pieces );
```

Однако мы умолчали о проблеме с ParserFun. Напомним, что функция parse, определенная в Parser, является корректной с точки зрения согласованности типов только потому, что структуры SymbolTable и AbstSyntax включают одну и ту же подструктуру Symbol, и благодаря этому используют один и тот же тип символов. Но теперь в ParserFun функция parse знает только сигнатуры этих двух структур, и ничего не знает о том, как они реализованы. Поэтому ML-компилятор укажет на ошибку в ParserFun, — и наша идея использования функторов для обеспечения модульного стиля программирования оказывается под угрозой.

Спасти дело помогают *спецификации соиспользования*. Идея состоит в том, чтобы включить в сигнатуру PARSER_PIECES набор равенств, гарантирующих тот факт, что только пара совместимых вариантов структур (для работы с таблицей символов и для работы с деревом синтаксического разбора) может быть передана функтору ParserFun. Новый вариант сигнатуры PARSER_PIECES будет выглядеть следующим образом:

```
signature PARSER_PIECES =
  sig
    structure SymboiTable : SYMBOLTABLE
    structure AbstSyntax: ABSTSYNTAX
    sharing SymbolTable.Symbol = AbstSyntax.Symbol
  end;
```

Фраза sharing гарантирует то, что может быть использована только совместимая пара структур SymboiTable и AbstSyntax (где “совместимая” означает “использующая одну и ту же структуру Symbol”). Если теперь мы используем эту модифицированную сигнатуру, то объявление функтора ParserFun становится допустимым.

В общем случае имеется две формы спецификации соиспользования — одна для типов, а другая для структур. В последнем примере мы использовали форму для структур — и обеспечили с помощью нее то, что обе компоненты параметра используют равные подструктуры. Две структуры равны тогда и только тогда, когда они получены в результате вычисления одного того же объявления структуры или применения одного и того же функтора к равным аргументам. Например, следующая попытка сформировать аргументы для функтора `ParserFun` будет отвергнута, поскольку она не удовлетворяет спецификации `sharing`:

```
structure Pieces : PARSER_PIECES =
  struct
    structure SymboiTable = SymbolTableFun( SymbolFun() )
    structure AbstSyntax = AbstSyntaxFun( SymbolFun() )
  end;
```

Причина здесь в том, что каждое применение `SymbolFun` создает новую структуру, отличную от всех других.

Другая форма спецификации соиспользования имеет дело с типами. Например, следующая версия `PARSER_PIECES` будет вполне подходящей, если единственное, в чем должны совпадать структуры `SymboiTable` и `AbstSyntax` — это используемый ими тип `symbol`:

```
signature PARSER_PIECES =
  sig
    structure SymboiTable : SYMBOLTABLE
    structure AbstSyntax : ABSTSYNTAX
    sharing SymboiTable.Symbol.symbol=AbstSyntax.Symbol.symbol
  end;
```

Равенство типов подобно равенству структур: два типа равны, если они получены в результате вычисления одного и того же объявления. Так, например, если два типа заданы идентичными объявлениями `datatype`, они будут различными.

Возвращаясь к нашему примеру, посмотрим, что произойдет, если мы нашли и исправили ошибку в функторе `SymbolFun`. Что мы должны сделать после этого? Во-первых, конечно, необходимо перекомпилировать `SymbolFun`. А затем достаточно повторить приведенную выше последовательность применений функторов — но повторной компиляции всех других функторов не требуется.

Глава 4

Ввод-вывод

ML обеспечивает только небольшое количество примитивов ввода-вывода, выполняющих посимвольный обмен с терминалом и файлами. Фундаментальным понятием в системе ввода-вывода ML является *поток литер* — конечная или бесконечная последовательность литер. Имеется два потоковых типа: `instream` — для потоков ввода, и `outstream` — для потоков вывода. Поток ввода получает свои литеры от *источника* (обычно это терминал или дисковый файл), а поток вывода посылает свои литеры *получателю* (также обычно терминалу или дисковому файлу). Поток инициализируется путем подключения его к источнику или получателю. Входной поток может иметь или не иметь конец; в том случае, когда конец имеется, ML обеспечивает возможность проверки условия достижения конца входного потока.

Основные примитивы ввода-вывода находятся в структуре `BasicIO`; ее сигнатура `BASICIO` имеет следующий вид:

```
signature BASICIO =
  sig
    (* Types and exceptions *)
    type instream
    type outstream
    exception io_failure: string

    (* Standard input and output streams *)
    val std_in : instream
    val std_out: outstream

    (* Stream creation *)
    val open_in : string -> instream
    val open_out: string -> outstream

    (* Operations on input streams *)
```

```
    val input: instream * int -> string
    val lookahead : instream -> string
    val close_in : instream -> unit
    val end_of_stream : instream -> bool

    (* Operations on output streams *)
    val output: outstream * string -> unit
    val close_out: outstream -> unit
  end;
```

Структура BasicIO автоматически открывается при запуске ML-системы, поэтому все упомянутые идентификаторы могут использоваться в программе без упоминания имени структуры.

Типы instream и outstream являются типами соответственно входных и выходных потоков. Исключение io_failure используется для информирования программы о любых ошибках, возникших в процессе ввода-вывода. Значение типа string, связанное с этим исключением, содержит информацию о возникшей ошибке (обычно в форме сообщения об ошибке).

Потоки std_in и std_out автоматически связываются с терминалом[1] (и не должны открываться или закрываться прикладной программой).

Примитивы open_in и open_out используются для связывания потока с дисковым файлом. В результате вычисления выражения open_in(s) создается новый входной поток, чьим источником является файл с именем s, и этот поток является результатом этого выражения. Если файла с именем s не существует, то возбуждается исключение io_failure, а параметром ее становится строка "Cannot open "^s. Аналогично, выражение open_out(s) создает новый выходной поток, чьим получателем является файл с именем s, и возвращает этот поток в качестве результата.

Примитив ввода input используется для чтения последовательности литер из входного потока. В результате вычисления input(s,n), n литер удаляются из входного потока, и сформированная из них строка возвращается в качестве результата. Если в данный момент в потоке доступно менее n литер, то программа ожидает, пока все необходимые литеры не станут доступными[2]. Если в процессе ввода достигается конец потока, то возвращаемая строка может состоять менее чем из n литер. В частности, результатом чтения из закрытого потока будет пустая строка. Функция

[1] В операционной системе UNIX эти потоки связываются с файлами стандартного ввода и вывода, которые могут быть подключены либо к терминалу, либо еще куда-то.

[2] Точный смысл слова “доступные” зависит от реализации. Например, операционная система обычно буферизует ввод с терминала, и передает программе строку целиком: в этом случае литеры становятся “доступными” программе после нажатия клавиши конца строки.

`lookahead(s)` возвращает очередную литеру входного потока `s` без удаления ее из потока. Работа с входным потоком завершается с помощью функции `close_in`. Обычно нет необходимости закрывать входные потоки, однако рекомендуется все-таки закрывать входной поток после того, как чтение из него завершено — иначе могут возникнуть неприятности, связанные с операционной системой. Конец входного потока может быть определен с помощью предиката `end_of_stream`, который описан как:

```
val end_of_stream(s) = (lookahead(s)="")
```

Примитив `output` используется для записи литер в поток (именно, записываются литеры из строки-аргумента). Функция `close_out` завершает вывод. После того, как выходной поток закрыт, любая попытка вывести в него что-то возбуждает исключение `io_failure` с параметром `"Output stream is closed"`.

В дополнение к базисному набору примитивов ввода-вывода ML также обеспечивает несколько дополнительных функций. Одна из них — `input_line` (типа `instream -> string`) — выполняет чтение строки из входного потока. Строка определяется как последовательность литер, завершающаяся признаком конца строки `\n`. Другая — функция `use` типа `string list -> unit` — получает в качестве аргумента список имен файлов; в результате ее вычисления содержимое указанных файлов прочитывается ML-системой так, как будто бы оно было введено на верхнем уровне диалога. Часто этот примитив используется при интерактивной работе для загрузки уже готовой части программы.

Упражнение 4.0.1 *Модифицируйте вашу программу решения задачи с ханойскими башнями так, чтобы она выводила найденную последовательность ходов на терминал.*

Упражнение 4.0.2 *Напишите функцию, печатающую решение задачи о ферзях в форме шахматной доски.*

Литература

[1] Harold Abelson and Gerald Sussman, *Structure and Interpetation of Computer Programs*, The MIT Press, 1985.

[2] Rod Burstall, David MacQueen, and Donald Sannella, *HOPE: An Experimental Applicative Language*, Edinburgh University Interyal Report CSR-62-80, 1980.

[3] Luca Cardelli, *ML under UNIX*, AT&T Bell Laboratories, 1984.

[4] Michael Gordon. Robin Milner, and Christopher Wadsworth, *Edinburgh LCF*, Springer-Verlag Lecture Notes in Computer Science, vol. 78, 1979.

[5] Robert Harper. David MacQueen, and Robin Milner, *Standard ML*, Edinburgh University Internal Report ECS-LFCS-86-2, March, 1986.

[6] David MacQueen. *Modules for Standard ML*, in [5].

[7] Robin Milner, Mads Tofte, and Robert Harper, *The Definition of Standard ML*, MIT Press, 1990.

Приложение A

Ответы

Ответ 2.3.1:

1. `Unbound value identifier: x`
2. ```
   > val x = 1 : int
   > val y = 3 : int
   > val z = 2 : int
   ```
3. `> 3:int`

Ответ 2.4.1:

ML-система будет пытаться сопоставить `hd::tl::nil` с `"Eat"::"the"::"walnut"::nil`. Поскольку эти списки имеют разную длину, сопоставление потерпит неудачу.

Ответ 2.4.2:

1. `{ b=x, ... }`
2. `_::_::x::_` или `[ _, _, x, _, _]`
3. `[_, (x, _)]`

Ответ 2.5.1:

```
local val pi=3.141592654
in fun circumference r = 2.0 * pi * r
   fun area r = pi * r * r
end
```
```

Ответ 2.5.2:

```
fun abs x = if x < 0.0 then -x else x
```

Ответ 2.5.3:

Для вычисления `fact(n)` потребуется вычислить выражение `new_if(n=0,1,fact(n-1))`. Аргументы функции вычисляются до вычисления самой функции; поэтому потребуется вычислить `fact(n-1)` (даже когда $n = 0!$); вычисление `fact(n-1)` по тем же причинам потребует вычисления `fact(n-2)` — и т.д.; возникнет бесконечный цикл.

Ответ 2.5.5:

Эта функция неэффективным способом выполняет перестановку элементов списка в обратном порядке.

Ответ 2.5.6:

```
fun isperfect n =
  let fun addfactors(1) = 1
        | addfactors(m) =
            if n mod m = 0 then m + addfactors(m-1)
                           else addfactors(m-1)
  in (n<2) orelse (addfactors(n div 2) = n) end
```

Ответ 2.5.7:

```
fun cons h t = h::t
fun powerset [] = [[]]
  | powerset(h::t) =
        let val pst = powerset t
        in (map (cons h) pst) @ pst end;
```

Ответ 2.5.8:

```
fun cc(0, _) = 1
  | cc(_,[]) = 0
  | cc(amount, kinds as (h::t)) =
      if amount < 0 then 0
      else cc(amount-h,kinds) + cc(amount,t);
fun count_change coins amount = cc(amount, coins);
```

Ответ 2.5.9:

```
fun nth(0,lst) = lst
  | nth(n,h::t) = nth(n-1,t);
fun count_change coins sum =
  let fun initial_table [] = [[0]]
        | initial_table (h::t) = [] :: (initial_table t)
      fun count(amount,table) =
        let fun count_using([],lst) = lst
              | count_using(h::t,h1::t1) =
                  let val t1' as ((c::_)::_) =
                                     count_using(t,t1)
                      val diff = amount - h
                      val cnt = c + if diff<0 then 0
                                    else if diff=0 then 1
                                    else hd(nth(h-1,h1))
                  in (cnt::h1)::t1' end
        in
          if amount > sum
          then hd(hd table)
          else count(amount+1, count_using(coins,table))
        end
  in count (0, initiaHable coins) end
```

Ответ 2.5.10:

```
local
  fun move_disk(from, to) = (from, to);
  fun transfer(from, to, spare, 1) = [move_disk(from,to)]
    | transfer(from, to, spare, n ) =
        transfer(from, spare, to, n-1) @
        [move_disk(from,to)] @
        transfer(spare, to, from, n-1)
in fun tower_of_hanoi(n) = transfer("A","B","C",n) end
```

Альтернативное решение, которое явно моделирует диски и проверяет допустимость ходов, может быть записано следующим образом:

```
local
  fun incl(m,n) = if m>n then 0 else m::incl(m+1,n)
  fun move_disk((f,fh::fl), (t,[]), spare) =
        ((f,fl), (t,[fh]), spare)
    | move_disk((f,fh::fl), (t,tl as (th::tt)), spare) =
        if (fh:int) > th then error "Illegal move"
        else ((f,fl), (t,fh::tl), spare)
```

```
  fun transfer(from, to, spare, 1) =
        move_disk(from, to, spare)
    | transfer(from, to, spare, n) =
        let val (f1,s1,t1) = transfer(from,spare,to,n-1)
            val (f2,t2,s2) = move_disk(f1,t1,s1)
            val (s3,t3,f3) = transfer(s2,t2,f2,n-1)
        in (f3,t3,s3) end
in
  fun tower_of_hanoi(n) =
       transfer(("A", incl(1,n)), ("B", []), ("C", []), n)
end
```

Ответ 2.7.1:

```
fun samefrontier(empty,empty) = true
  | samefrontier(leaf x, leaf y) = x=y
  | samefrontier(node(empty,t1),node(empty,t2)) =
      samefrontier(t1,t2)
  | samefrontier(node(leaf x,t1), node(leaf y,t2)) =
      x=y andalso samefrontier(t1,t2)
  | samefrontier(t1 as node _, t2 as node _) =
      samefrontier(adjust t1, adjust t2)
  | samefrontier(_,_) = false
and adjust(x as node(empty,_)) = x
  | adjust(x as node(leaf _,_)) = x
  | adjust(node(node(t1,t2),t3)) =
      adjust(node(t1,node(t2,t3)))
```

Приведем и другое решение, использующее исключения (см. раздел 2.8):

```
fun sameftrontier(treel,tree2) =
  let
    exception samefringe : unit
    fun check_el(empty,empty,rest_t2) = rest_t2
      | check_el(leaf x, leaf y, rest_t2 ) =
          if x=y then rest_t2 else raise samefringe
      | check_el(el,node(l,r), rest_t2) =
          check_el(el, l, r::rest_t2)
      | check_el(_,_,_) =
          raise samefringe
    fun check(_, []) =
          raise samefringe
      | check(empty,tree2) =
          check_el(empty, hd tree2, tl tree2)
```

```
    | check(l as leaf(el),tree2) =
        check_el(l, hd tree2, tl tree2)
    | check(node(t1,t2),tree2) =
        check(t2, check(t1,tree2))
in
  null (check(tree1,[tree2])) handle samefringe => false
end
```

Ответ 2.7.2:

```
abstype 'a set = set of 'a list
  with
    val emptyset = set []
    fun singleton e = set []
    fun union(set l1, set l2) = set (l1 @ l2)
    fun member(e, set []) = false
      | member(e, set(h::t)) =
          (e=h) orelse member(e, set t)
    fun intersection(set [], s2) = set []
      | intersection(set(h::t), s2) =
          let val tset as (set tl) =
                intersection(set t, s2)
          in if member(h,s2) then set(h::tl) else tset end
   end
```

Ответ 2.7.3:

```
abstype 'a set = set of ('a list *
                   {eq : 'a*'a->bool, lt : 'a*'a->bool})
  with
    fun emptyset ops = set([],ops)
    fun singleton(e,ops) = set([e],ops)
    fun member(e,set(l,{eq,lt}) =
      let fun find [] = false
            | find(h::t) = if eq(e,h) then true
                           else if lt(e,h) then false
                           else find(t)
      in find l end
    fun union(set(lst, ops as {eq,lt}), set(lst',_)) =
      let fun merge([],lst) = lst
            | merge(lst,[]) = lst
            | merge(l1 as (h1::t1), l2 as (h2::t2)) =
                if eq(h1,h2) then h1::merge(t1,t2)
                else if lt(h1,h2) then h1::merge(t1,t2)
                else h2::merge(t1,t2)
```

```
      in set(merge(lst,lst'),ops) end
    fun intersect(set(lst, ops as {eq,lt}), set(lst',_)) =
      let fun inter([],lst) = []
            | inter(lst, []) = []
            | inter(l1 as (h1::t1), l2 as (h2::t2)) =
                if eq(h1,h2) then h1::inter(t1,t2)
                else if lt(h1,h2) then inter(t1,l2)
                else merge(l1,t2)
      in set(inter(lst,lst'), ops) end
  end
```

Ответ 2.8.1:

1. Исключение, привязанное к `Exn` вне `let`, отличается от привязанного к `Exn` внутри `let`. Поэтому исключение, возникающее в процессе вычисления `f(200)` (имеющее целочисленный параметр `200`), может быть обработано только обработчиком исключений внутри `let` — и не может быть обработано внешним обработчиком (который, к тому же, ожидает параметра типа `bool`). В результате сообщение о возникновении необработанного исключения будет выдано на верхнем уровне диалога. Ситуация не изменится и в том случае, если внешнее исключение будет объявлено как получающее параметр типа `int` — оба исключения по-прежнему останутся различными.
2. Если при вычислении `f(v)` окажется, что `p(v)` ложно, а `q(v)` истинно, то будет рекурсивно вызвана функция `f` с параметром `b(v)`. Далее, если `p(b(v))` и `q(b(v))` окажутся ложными, будет выработано исключение `Exn`. Но это исключение не будет обработано приведенным в примере обработчиком, поскольку указанное в нем исключение `Exn` есть конструктор, созданный при вычислении `f(v)` — а выработанное `Exn` есть конструктор, созданный при вычислении `f(b(v))` — и это две разные вещи. (Еще раз обращаем внимание на то, что все ообъявления внутри объявления функции исполняются каждый раз при рекурсивном вызове, и при этом выполняется новая привязка идентификаторов).

Ответ 2.8.2:

```
fun threat((x::int,y), (x',y')) =
            (x=x')
     orelse (y=y')
     orelse (x+y=x'+y')
```

```
     orelse (x-y=x'-y')
fun conflict(pos, []) = false
  | conflict(pos, h::t) =
      threat(pos, h) orelse conflict(pos, t)

exception Conflict;

fun addqueen(l,n,place) =
  let fun tryqueen(j) =
    ( if conflict((i,j), place) then raise Conflict
      else if i=n then (i,j)::place
      else addqueen(l+1, n, (i,j)::place) )
    handle Conflict =>
      if j=n then raise Conflict else tryqueen(j+1)
  in tryqueen(1) end

fun queens(n) = addqueen(1, n, [])
```

Ответ 2.8.3:

```
exception Conflict of ((int*int) list) list

fun addqueen(l,n,place,places) =
  let fun tryqueen(j, places) =
        ( if conflict ((i,j), place)
          then raise Conflict(((i,j)::place)::places)
          else addqueen (l+1, n, (ij)::place, places) )
        handle Conflict newplace =>
          if j=n then raise Conflict(newplace)
          else tryqueen(j+1,newplaces)
  in tryqueen(1,places) end

fun allqueens(n) = addqueen(1, n, [], [])
                     handle Conflict(places) => places
```

Ответ 2.9.1:

```
val primes =
  let fun nextprime(n,l) =
        let fun check(n,[]) = n
              | check(n,h::t) = if (n mod h) = 0
                                then check(n+1,l)
                                else check(n,t)
        in check(n,l) end
      fun primstream(n,l) =
```

```
        mkstream (fn () =>
          let val n'=nextprime(n,l)
          in (n', primstream(n'+1,n'::l)) end)
  in primstream(2, []) end
```

Ответ 2.9.2:

```
abstype 'a stream =
                stream of ( unit -> ('a * 'a stream)) ref
  with fun next(stream f) =
                let val res = (!f)()
                in (f := fn() => res; res) end
       fun mkstream f = stream(ref f)
  end
```

Приведем другое решение — более многословное, но, возможно, более понятное:

```
abstype 'a stream = stream of 'a streamelmt ref
    and 'a streamelmt = uneval of (unit -> ('a*'a stream))
                      | eval of 'a*'a stream
  with fun next(stream(r as ref(uneval(f)))) =
             let val res=f() in (r := eval res; res ) end
         | next ( stream (ref(eval(r)))) = r
       fun mkstream f = stream(ref(uneval f))
  end
```

Ответ 2.9.3:

```
abstype 'a stream =
    stream of (unit -> ('a * 'a stream)) ref
    with local exception EndOfStream in
           fun next(stream f) =
             let val res = (!f)()
             in (f := fn () => res; res)
             end
           fun mkstream f = stream (ref f)
           fun emptystream() =
                stream(ref (fn () => raise EndOfStream))
           fun endofstream(s) =
                (next s; false) handle endofstream => true
         end
    end
```

Ответ 3.2.1:

```
structure INTORD : ORD =
  struct
    type t = int
    val le : int*int -> bool = op <
  end

structure RSORD : ORD =
  struct
    type t = real*string
    fun le((r1:real, s1:string), (r2,s2)) =
      (r1 < r2) oresle ((r1 = r2) andalso (s1 < s2)) end
```

Ответ 3.2.2:

Сигнатура указывает, что тип `n` есть `'a list`; отсюда, в частности, следует, что если структура `T` сопоставима с сигнатурой `SIG`, то выражение `true::(T,n)` должно быть допустимым. Это условие не будет выполнено в нашем примере, поскольку `n` имеет более конкретный тип, а именно `int list`. Поэтому данное определение компилятор признает ошибочным.

Ответ 3.2.3:

```
sig type 'a t val x : bool*int end
```
и
```
sig type 'a t vai x : bool t end
```

Ответ 3.2.4:

Только сигнатура `B` удовлетворяет правилу замкнутости сигнатуры (все остальные содержат открытые ссылки на структуру `A`).

Ответ 3.2.5:

```
signature STACK =
  sig
    datatype 'a stack = nilstack | push of 'a * 'a stack
    exception pop and top
    val empty : 'a stack -> bool
    and pop : 'a stack -> 'a stack
    and top : 'a stack -> 'a
  end
```

```
structure Stack : STACK =
  struct
    datatype 'a stack = nilstack | push of 'a * 'a stack
    exception pop and top
    fun empty(nilstack) = true
      | empty _ = false
    fun pop(push(_,s)) = s
      | pop _ = raise pop
    fun top(push(x, _)) = x
      | top _ = raise top
  end
```

Ответ 3.2.6:

```
structure Exp : EXP =
  struct
    datatype id = Id of string
    datatype exp = Var of id
                 | App of id * (exp list)
  end

signature SUBST =
  sig
    structure E : EXP
    type subst
    val subst: (E.id * E.exp) list -> subst
    val lookup : E.id * subst -> E.exp
    val substitute : subst -> E.exp -> E.exp
  end

structure Subst: SUBST =
  struct
    structure E = Exp
    type subst = (E.id * E.exp) list
    fun subst(x) = x
    fun lookup(id, Q) = E.Var id
      | lookup (id, (id',e)::l) =
          if id=id' then e else lookup(id,l)
    fun substitute s (E.Var id) = lookup(id.s)
      | substitute s (E.App(id.args)) =
          E.App(id, map(substitute s) args)
  end
```

Ответ 3.3.1:

```
abstraction Rect: COMPLEX =
```

```
  struct
    datatype complex = rect of real*real
    exception divide
    fun rectangular{real,imag} =
      rect(real,imag)
    fun plus(rect(a,b), rect(c,d)) = rect(a+c, b+d)
    fun minus(rect(a,b), rect(c,d)) = rect(a-c, b-d)
    fun times(rect(a,b), rect(c,d)) =
                                rect(a*c-b*d, a*d+b*c)
    fun divide(rect(a,b), rect(c,d)) =
      let val cd2 = c*c+d*d in
        if cd2=0.0 then raise divide
        else rect ((a*c+b*d)/cd2, (b*c-a*d)/cd2 )
      end
    fun eq(rect(a,b), rect(c,d)) = (a=c) andalso (b=d)
    fun real_part(rect(a,_)) = a
    fun imag_part(rect(_,b)) = b
  end
```

Ответ 3.4.1:

```
signature ORD =
  sig
    type elem
    val eq : elem*elem -> bool
    val le : elem*elem -> bool
  end

signature SET =
  sig
    type set
    structure O : ORD
    val emptyset: set
    val singleton : O.elem -> set
    val member: O.elem * set -> bool
    val union : set * set -> set
    val intersect: set * set -> set
  end

functor Set(O : ORD) : SET =
  struct
    datatype set = set of O.elem list
    structure O = O
    val emptyset = set []
    fun singleton e = set [e]
```

```
    fun member(e, set l) =
      let fun find [] = false
            | find (h::t) = if O.eq(e,h) then true
                            else if O.lt(e,h) then false
                            else find(t)
      in find l end
    fun union(set l, set l') =
      let fun merge([],l) = l
            | merge(l,[]) = l
            | merge(l1 as (h1::t1), l2 as (h2,t2)) =
                if O.eq(h1,h2) then h1::merge(t1,t2)
                else if O.lt(h1,h2) then h1::merge(t1,l2)
                else h2::merge(l1,t2)
      in set(merge(l,l')) end
    fun intersect(set l, set l') =
      let fun inter([],l) = []
            | inter(l, []) = []
            | inter(l1 as (h1::t1), l2 as (h2::t2)) =
                if O.eq(h1,h2) then h1::inter(t1,t2)
                else if O.lt(h1,h2) then inter(t1,l2)
                else inter(l1,t2))
      in set(inter(l,l')) end
  end
```

Ответ 4.0.1:

```
local
  fun incl(m,n) = if m>n then [] else m::incl(m+1,n)
  fun move_disk ((f, fh::fl), (t, tl), spare) =
        if not(null tl) andalso (fh:int) > hd tl
        then error "Illegal move"
        else (output (std_out,
                      "Move " ^ (makestring fh) ^
                      " from " ^ f ^ " to " ^ t ^ "\n" );
              ((f,fl), (t,fh::tl), spare))
  fun transfer(from, to, spare, 1) =
        move_disk (from, to, spare)
    | transfer(from, to, spare, n) =
        let val (f1,s1,t1) =
                        transfer (from, spare, to, n-1)
            val (f2,t2,s2) = move_disk(f1, t1, s1)
            val (s3,t3,f3) = transfer(s2, t2, f2, n-1)
        in (f3,t3,s3)
        end
in
```

```
  fun tower_of_hanoi(n) =
    (transfer(("A",incl(1,n)), ("B",[]), ("C",[]), n); ())
end
```

Ответ 4.0.2:

```
fun printboard(place,n,s) =
  let fun present(pos : (int*int), []) = false
        | present(pos, h::t) =
           (pos=h) orelse present(pos,t)
      fun printcolumn(i, j) =
            if j>n then ()
            else ( output (s, if present((i,j),place)
                              then " Q "
                              else " . ");
                   printcolumn(i,j+1) )
      fun printrow(i) =
            if i>n then ()
            else ( printcolumn(ij);
                   output(s,"\n");
                   printfow(i+1) )
  in ( printrow(1); output(s,"\n") )
  end
```