Министерство образования и науки
Российской Федерации

Федеральное агентство по образованию

Ярославский государственный университет
им. П.Г. Демидова

В.А. Башкин

Функциональное программирование на языке SML

Методические указания

Рекомендовано Научно-методическим советом университета для студентов, обучающихся по специальности Математическое обеспечение и администрирование информационных систем

Ярославль 2007

УДК 004.4:004.7
ББК 3973.2–018.1я73
Б 33

Рекомендовано
Редакционно-издательским советом университета
в качестве учебного издания. План 2007 года

Б 33 **Башкин, В.А.** Функциональное программирование на языке SML: метод. указания / В.А. Башкин; Яросл. гос. ун-т. — Ярославль: ЯрГУ, 2007. — 40 с.

В методических указаниях излагаются основные принципы создания программ на языке SML. Описываются средства языка, приводятся примеры использования приемов функционального программирования.

Предназначено для студентов IV курса факультета информатики и вычислительной техники ЯрГУ, обучающихся по специальности 010503 Математическое обеспечение и администрирование информационных систем, очной формы обучения (дисциплина «Функциональное программирование», блок ДС).

Библиогр.: 11 назв.

УДК 004.4:004.7
ББК 3973.2–018.1я73

Оглавление

Предисловие

Методические указания предназначены для студентов 4 курса факультета информатики и вычислительной техники ЯрГУ, обучающихся по специальности 010503 Математическое обеспечение и администрирование информационных систем.

В методических указаниях излагаются основы программирования на языке `SML/NJ` (Standard ML of New Jersey). Этот диалект языка `Standard ML` является одним из самых популярных современных языков функционального программирования.

В первом разделе приводятся общие сведения о функциональном стиле программирования, а также описываются некоторые особенности работы в системе `SML/NJ`.

Во втором разделе вводятся основные понятия языка, перечисляются базовые типы данных и стандартные операторы; рассказывается о способах работы с ключевыми компонентами функциональной программы — функциями и динамическими структурами данных; приводятся примеры важнейших приёмов программирования: сопоставления с образцом, передачи функционального параметра, а также различных видов рекурсии.

В третьем разделе описываются дополнительные возможности языка: параметрический полиморфизм, работа с типами данных (в том числе рекурсивными), обработка исключений, а также императивные обращения к памяти и файлам. Кроме того, рассматриваются методы модульного программирования на `SML` с использованием структур, сигнатур и функторов.

Общие сведения о функциональном стиле программирования можно получить из книг [2, 4]. Кроме того, в сети доступно довольно много англоязычных электронных руководств по программированию на языке `SML` (наиболее полное — [7]).

1. Предварительные сведения

1.1. Функциональное программирование

Когда программист пишет программу на каком-нибудь "обычном" языке программирования (например, `C++` или `Pascal`), он формирует (сначала в голове, а потом и в коде программы) последовательность выполнения тех или иных операций над набором переменных. Фактически, компьютер получает *указания*, что и как ему делать. Такой стиль программирования называется *императивным*, а реализующие его языки — *императивными языками программирования*.

Однако это не единственный возможный способ управления компьютером. Существуют языки и подходы, реализующие так называемый *декларативный стиль программирования*. Программы на таких языках представляют собой не последовательности действий, а исчерпывающие описания взаимосвязей результатов вычислений и входных данных.

Основное принципиальное различие императивного и декларативного стилей программирования можно сформулировать следующим образом: если в императивной программе указано, *как делать*, то в декларативной — *что делать*.

В декларативном программировании также есть своё разделение. Основными его видами являются *логическое* и *функциональное* программирование. Логическая программа представляет собой набор правил, связывающих различные факты предметной области. Функциональная программа — это набор функций, связанных между собой аргументами, которые, в свою очередь, тоже могут быть функциями. Атомарный акт вычислений — это подготовка аргументов для использующей их функции. Готовность аргументов трактуется как разрешение вычисления функции.

Функциональные конструкции можно найти и в "обычных" программах. Простейший пример — вычисление арифметического

выражения $x = a*b+c*d$. Здесь не указано явно, какое из действий должно быть выполнено первым — $a * b$ или $c * d$. Таким образом, задан не алгоритм, а *правило* вычисления. Конечно, при компиляции программы получится именно последовательность машинных команд (причем порядок действий выберет сам компилятор), однако на этапе написания текста программы программисту удалось уйти от слишком подробной детализации процесса вычисления.

Оказывается, перейти от последовательного алгоритма к правилам преобразования аргументов можно не только в таких простых ситуациях, как арифметические вычисления. Существуют функциональные способы описания циклов, рекурсии и выбора. Математическим доказательством реализуемости любого алгоритма при помощи одних только функций является фундаментальный результат Чёрча об универсальности λ-исчисления. Фактически, λ-функции Чёрча появились даже раньше, чем машины Тьюринга, так что функциональное программирование получило математическое обоснование раньше императивного.

Основной принцип функционального программирования — это сам принцип функциональности: при вызове одной и той же функции с одними и теми же значениями аргументов всегда возвращается один и тот же результат. В "обычных" языках программирования этот принцип нарушен, так как там функция может обращаться к глобальным переменным или общей памяти.

Второй важнейший принцип — отсутствие искусственного управления процессом вычислений. Конечно, если аргументом функции является другая функция, то она будет вычислена раньше, чем вызвавшая её функция. Однако других способов задания последовательности действий (например, циклов) в чистой функциональной программе быть не может.

Следование этим принципам приводит к довольно интересным результатам. Например, в чистых функциональных языках нет понятия переменной, следовательно, отсутствует и оператор присваивания. Нет операторных скобок, операторов цикла, механиз-

мов работы с памятью. Синтаксис очень беден, однако, и программа получается, как правило, более красивая и компактная.

К плюсам функционального подхода можно отнести:

- близость кода программы её спецификации (по сути дела программа представляет собой уточнённую спецификацию);
- отсутствие побочных эффектов от использования оператора присваивания и работы с памятью;
- отсутствие привязки к архитектуре (программа максимально абстрактна, поэтому её можно откомпилировать для любой конкретной ЭВМ);
- возможность так называемых “ленивых” (отложенных) вычислений, при которых непосредственно в ходе выполнения программы отсекаются избыточные вызовы подпрограмм;
- простота работы с динамическими рекурсивными структурами данных (списками, деревьями);
- реализация так называемого параметрического (“истинного”) полиморфизма, при котором одна и та же функция может работать с аргументами разных типов;
- работа с потенциально бесконечными типами данных;
- удобство анализа программы формальными математическими методами (например, при её верификации).

Есть и свои минусы:

- невозможно организовать интерактивные интерфейсы;
- при написании кода программист не может использовать дополнительные возможности конкретной архитектуры.

На практике функциональные языки чаще всего используются для научных и критически важных вычислений, то есть там, где необходимо реализовывать и верифицировать весьма сложные алгоритмы обработки данных без интерактивных интерфейсов.

Первым языком функционального программирования является язык LISP, созданный в 1960-е годы Джоном Маккарти. К последнему поколению функциональных языков принадлежат такие языки, как Standard ML (и SML/NJ — его самый популярный диалект), Haskell, Miranda, LISP/Scheme.

1.2. Работа в системе Lucent SML/NJ

Интерпретатор SML/NJ фирмы Lucent Technologies распространяется на условиях freeware и доступен для скачивания по адресу: http://cm.bell-labs.com/cm/cs/what/smlnj/index.html.

Запуск системы — команда sml в командной строке. Выход из системы — Ctrl-Z. Прерывание вычисления — Ctrl-C.

Базовым элементом языка SML является не оператор, а выражение. Выполнение программы состоит в вычислении (упрощении) введённых пользователем выражений. Система возвращает вычисленный результат с указанием типа:

```
- "Hello World";
val it = "Hello World" : string
```

Здесь и далее в рамку обведены примеры диалога пользователя с системой. Вводимые пользователем команды — строки, начинающиеся с символа "-" (приглашения). Остальные строки — ответы интерпретатора. Последнее вычисленное выражение получает имя по умолчанию it. Можно явно указать имя введённого выражения (команда val) и использовать это имя в дальнейших вычислениях:

```
- 3+4;
val it = 7 : int
- val x = 5*6;
val x = 30 : int
- x-it;
val it = 23 : int
```

Язык SML — case sensitive. В идентификаторах не принято использовать заглавные буквы.

Вводимые пользователем выражения отделяются друг от друга (закрываются) символом “;”. Во второй и последующих незакрытых строках приглашение автоматически меняется на “=”:

```
- 4 + 4 +
= 4;
val it = 12 : int
```

Операторы if-then-else и case:

```
- val x = if Math.pi > 3.0 then 5 else 0;
val x = 5 : int
- case x
=   of 0 => "bad"
=   |  5 => "good"
=   |  _ => "?";
val it = "good" : string
```

Символ подчёркивания является универсальной подстановкой (аналог раздела default в операторе switch языка C).

Примеры сообщений об ошибках:

```
- zzz;
stdIn:1.1-1.4 Error: unbound variable or constructor: zzz
- 5+true;
stdIn:1.1-2.2 Error: operator and operand don't agree
```

В первом случае используется некорректный (не связанный ни с одним выражением) идентификатор zzz, во втором — тип оператора + (int) не соответствует типу второго операнда (bool).

Комментарии в тексте программы оформляются следующим образом: (* это комментарий *). Их разрешено вставлять только в те места программы, где допускается наличие пробела (например, нельзя разрывать комментарием идентификатор).

Определение функции в SML имеет следующий синтаксис:

```
fun <имя> <аргумент> = <выражение>;
```

Примеры:

```
- fun sqr x = x*x;
val sqr = fn : int -> int
- sqr 55;
val it = 3025 : int
```

Функцию с тем же именем можно переопределить:

```
- fun sqr x = x = "квадрат";
val sqr = fn : string -> bool
- sqr "zzz";
val it = false : bool
- sqr "квадрат";
val it = true : bool
```

Все имена выражений (и функций), которые были введены в ходе данного сеанса работы с системой, сохраняются до конца сеанса (но если имя было переопределено, то используется последнее объявленное выражение с данным именем). Также можно подгружать содержимое внешних файлов при помощи команды `use`.

Содержимое файла `my.sml`:

```
fun divide_by_2 x = x / 2.0; (* функция деления на 2 *)
```

Работа с функцией из файла `my.sml`:

```
- use "my.sml";
[opening my.sml]
val divide_by_2 = fn : real -> real
val it = () : unit
- divide_by_2 1.0;
val it = 0.5 : real;
```

Клавиши управления курсором $\Uparrow$ и $\Downarrow$ позволяют перебрать предыдущие значения строки ввода. Также можно работать в текстовом редакторе (например, Notepad), а затем переносить набранный текст через буфер обмена в окно `SML/NJ`.

2. Базовый язык

2.1. Основные элементы языка

Привязки. В функциональных языках программирования любому синтаксически правильному выражению можно сопоставить символьное обозначение. Это называется *привязкой имени*:

```
- val x = 3;
val x = 3 : int
```

В данном случае выражению `3` было присвоено имя `x`. Теперь это выражение является "квалифицированным" (именованным) и к нему можно обращаться по имени `x`:

```
- x+5;
val it = 8 : int
```

По синтаксису имена выражений несколько напоминают переменные императивных языков программирования, однако имеется фундаментальное отличие: выражению не всегда соответствует какое-то конкретное значение. Имена в функциональных языках могут присваиваться произвольным выражениям, в том числе функциям, частично вычисленным выражениям и т.п. В следующем примере новое имя присваивается функции синуса:

```
- val sinus = Math.sin;
val sinus = fn : real -> real
```

Если есть возможность, введённое выражение упрощается сразу (как в примере с `x+5`). Однако этот процесс не следует рассматривать как "вычисление" — выражение не вычисляется, а именно упрощается в соответствии со свойствами входящих в него подвыражений.

Ещё один вид привязки — привязка имени типа при помощи команды `type`. Мы можем назначить произвольный псевдоним любому базовому типу данных: `type my_float = real`. Теперь ве-

щественные числа можно обозначать как `my_float`.

Область действия (область видимости) привязки можно ограничить. В выражении — командой `let`, в другой привязке — командой `local`:

```
let
  val a = 3+4
  val b = 7*a-9
in
  (a*b)-a
end;
```

```
local
  val x = 5.0
  val y = 10.2-x
in
  val z = x*y
end;
```

Выражения `a` и `b` доступны только в выражении `(a*b)-a`; выражения `x` и `y` доступны только в привязке `z`.

Базовые типы данных и операторы. Любое выражение имеет тип. В большинстве случаев компилятор сам способен определить тип по синтаксису выражения. Однако есть способ указать тип введенного выражения явно — через двоеточие.

```
- 5.0 : real;
val it = 5.0 : real
```

Это может потребоваться для ограничения области определения аргумента функции или в качестве дополнительного контроля.

В таблице приведены базовые типы данных языка `SML`:

Тип	Запись	Операции	Отношения
`unit`	`()`		
`bool`	`true`, `false`	`andalso`, `orelse`, `not`	`=`, `<>`
`int`	`5`, `0`, `~5`	`+`, `-`, `*`, `div`, `mod`, `~`	`=`, `<`, `>`, `<=`, `>=`, `<>`
`real`	`3.14`, `31416e~4`, `inf`, `nan`	`+`, `-`, `*`, `/`, `~`	`<`, `>`, `<=`, `>=`, `<>`
`char`	`#"z"`, `#"\n"`		`=`, `<`, `>`, `<=`, `>=`, `<>`
`string`	`"zzz"`, `"1\n2\n3"`	`^`	`=`, `<`, `>`, `<=`, `>=`, `<>`

Простейший тип — `unit`. Выражения этого типа могут принимать только одно значение — `()`.

Логические выражения имеют тип `bool`. Для связок `andalso` и `orelse` (эквивалентных обычным `and` и `or`) компилятор использует ленивое (отложенное) вычисление. Первый операнд вычисляется всегда, второй — только при необходимости. Например, если в выражении `x andalso y` вычисление подвыражения `x` дало результат `false`, то выражение `y` вычисляться уже не будет.

В типе `int` нет деления, есть только деление нацело и остаток. Делить на ноль в целых числах нельзя. В сам язык `SML` не заложено никаких ограничений на размер целого числа, однако система `SML/NJ` отводит под целые числа 31 бит, поэтому они могут принимать значение от `~1073741824` до `1073741823`. Для работы с произвольными целыми можно использовать структуру `IntInf`.

Унарный минус обозначается тильдой: `~5.0`, а не `-5.0`.

В действительных числах имеются особые значения — бесконечность `inf` (infinity) и неопределенность `nan` (not a number). Делить на ноль конечное действительное число можно, результатом будет `inf`, при делении `inf/inf` получается `nan`.

Особенностью языка является запрет на сравнение действительных чисел при помощи отношения равенства (`=`). Любое вычисленное действительное число считается приближением. Точные вычисления допустимы только с рациональными дробями.

Отсутствует неявное преобразование типов. Например, запись `5/2` приведет к сообщению об ошибке, так как в `int` нет деления. Правильная запись — `5.0/2.0` или `5 div 2`.

Для строкового типа имеются функции длины строки (`size`) и подстроки (`substring`), а также оператор конкатенации `^`:

```
- size("Hello" ^ " " ^ "World");
val it = 11 : int
- substring("Hello World", 6, 5);
val it = "World" : string
```

В следующей таблице представлены наиболее важные встроенные функции для работы с данными различных типов:

Модуль	Функция	Сигнатура
Int	abs	int -> int
	min	(int * int) -> int
	max	(int * int) -> int
	toString	int -> string
	fromString	string -> int option
Real	abs	real -> real
	min	(real * real) -> real
	max	(real * real) -> real
	isFinite	real -> bool
	isNan	real -> bool
	toString	real -> string
	fromString	string -> real option
	floor	real -> Int.int
	ceil	real -> Int.int
Math	sqrt	real -> real
	sin	real -> real
	cos	real -> real
	tan	real -> real
	asin	real -> real
	acos	real -> real
	atan	real -> real
	exp	real -> real
	pow	(real * real) -> real
	ln	real -> real
	pi	real
	e	real
Char	ord	char -> int
	chr	int -> char
	succ	char -> char
	pred	char -> char

При вызове любой из этих функций необходимо указывать имя модуля, которому она принадлежит (через точку):

```
- Math.sin 0.5;
val it = 0.479425538604 : real
```

Составные типы данных: наборы и записи. Набор (tuple) объявляется в скобках, при этом через запятую перечисляются все его элементы (поля):

```
- val a = (1, 2, "apple");
val a = (1,2,"apple") : int * int * string
```

В строке `int * int * string`, описывающей тип введенного выражения, символ `*` обозначает декартово произведение. Ограничений на типы полей в наборе нет.

Для извлечения значения i-го элемента набора используется функция `#i`:

```
- #3 a;
val it = "apple" : string
```

Набор из одного элемента эквивалентен самому этому элементу. Набор из нуля элементов эквивалентен значению `()` типа данных `unit`.

Запись (record) — это набор, в котором поля не упорядочены (как в обычном наборе), а именованы. Записи объявляются в фигурных скобках, при этом через запятую перечисляются поля с указанием их имён:

```
- val man = {Fam="Ivanov", Age=21};
val man = {Age=21,Fam="Ivanov"} : {Age:int, Fam:string}
```

Извлечение значения поля:

```
- #Age man;
val it = 21 : int
```

2.2. Функции

Объявление функции. Первая форма синтаксиса объявления функции (при помощи оператора `fun`):

```
- fun incr x = x + 1;
val incr = fn : int -> int
```

Вторая форма синтаксиса объявления функции (при помощи привязки `val` и оператора `fn`):

```
- val incr = fn x => x + 1;
val incr = fn : int -> int
```

Вторая форма позволяет определять безымянные функции:

```
- (fn x => x + 1) 6;
val it = 7 : int
```

При объявлении функции можно явно указывать тип аргумента и тип возвращаемого значения:

```
- fun incr (x : int) : int = x + 1;
val incr = fn : int -> int
```

Все функции в `SML` являются функциями от одного аргумента. Однако существует и некий аналог функции от нескольких аргументов:

```
- fun subtr (x:int) (y:int) = x - y;
val subtr = fn : int -> int -> int
- subtr 6 4;
val it = 2 : int
```

Тип выражения `fn:int->int->int` (эквивалентная запись типа `fn:int->(int->int)`) соответствует функции, которая получает в качестве аргумента целое число и возвращает функцию с типом `fn:int->int`.

Можно указывать не все аргументы, тогда результатом будет не конкретное значение, а функция с меньшим числом аргумен-

тов. Рассмотрим в качестве примера функцию от одного аргумента, полученную при помощи частичного вычисления функции от двух аргументов:

```
- val subtr_from_10 = subtr 10;
val subtr_from_10 = fn : int -> int
- subtr_from_10 8;
val it = 2 : int
```

В выражениях скобки всегда ставятся левоассоциативно, поэтому `subtr 6 4` означает `(subtr 6) 4`, а не `subtr (6 4)`.

Ещё один способ передачи функции сразу нескольких входных значений (выражений) — аргумент типа набор или запись:

```
- fun foo (x:int, y:int, z:int) : int = x + y + z;
val foo = fn : int * int * int -> int
- foo (3, 4, 5);
val it = 12 : int
```

Функции, которые возвращают несколько значений одновременно (точнее, одно значение типа набор или запись):

```
- fun sum_prod (x : int, y : int) : int * int = (x+y, x*y);
val sum_prod = fn : int * int -> int * int
- sum_prod (3, 4);
val it = (7, 12) : int * int
```

В `SML` есть возможность определения инфиксных операторов. Вначале при помощи ключевого слова `infix` задается тип функции, затем объявляется сама функция:

```
- infix conc;
infix conc
- fun (x:string) conc (y:string) = x ^ y;
val conc = fn : string * string -> string
- "one" conc "two";
val it = "onetwo" : string
```

При помощи ключевого слова `op` к инфиксным операторам (в

том числе стандартным) можно обращаться как к функциям:

```
- (op +) (5, 6);
val it = 11 : int
```

Сопоставление с образцом (шаблоном). Один из наиболее часто используемых приемов при описании функций — так называемое сопоставление с образцом (pattern matching). Синтаксис сопоставления похож на синтаксис оператора `case`:

```
- fun int2str 1 = "One"
= |   int2str 2 = "Two"
= |   int2str 3 = "Three"
= |   int2str _ = "some number";
val int2str = fn : int -> string
- int2str 2;
val it = "Two" : string
- int2str 789;
val it = "some number" : string
```

Тело функции делится на несколько частей (вариантов), каждая из которых запускается в том случае, когда значение аргумента подходит соответствующему варианту. Перебор вариантов аргумента производится слева направо, поэтому универсальную подстановку (символ `_`) имеет смысл ставить только последней.

Функция вычисления числа Фибоначчи:

```
fun fib 0 = 0
|   fib 1 = 1
|   fib n = fib(n-1) + fib(n-2);
```

Здесь шаблон n соответствует произвольному целому аргументу (не равному 0 и 1). Символьное обозначение `n` *сопоставлено* аргументу, и внутри функции (точнее, внутри раздела `fib n`) мы можем обращаться к нему по имени.

В качестве вариантов (шаблонов) могут выступать не только значения, переменные или универсальная подстановка `_`, но и достаточно сложные выражения, позволяющие структурировать

аргумент. В частности, наборы:

```
- val (x, y, z, _) = (1, {r1=5, r2=6}, "a", "b");
val x = 1 : int
val y = {r1=5, r2=6} : {r1:int, r2:int}
val z = "a" : string
```

Приведенный пример показывает, что шаблоны можно использовать и в привязках.

Сопоставление с образцом позволяет создавать очень компактные и элегантные выражения. Например, объявление функции вычисления факториала практически не отличается от математического определения самого факториала:

```
fun fact 0 = 1
|   fact n = n * fact(n-1);
```

Шаблоны могут покрывать не все варианты значений аргумента. Допустимы неполные определения:

```
- fun fz 0 = 0;
stdIn:164.1-164.13 Warning: match nonexhaustive
          0 => ...
val fz = fn : int -> int
- fz 0;
val it = 0 : int
- fz 1;
uncaught exception nonexhaustive match failure
  raised at: stdIn:164.12
```

В шаблонах сопоставления с образцом нельзя использовать значения и переменные типов, не допускающих сравнение при помощи отношения равенства (в частности, типа `real`).

Функция как аргумент. В функциональных языках легко реализовать так называемые высокоуровневые функции (функции-шаблоны), получающие в качестве аргументов другие функции и выражения. Пример функции-шаблона:

```
fun apply_predicate(x, y, f) = if f(x,y) then x else y;
```

Третий аргумент шаблона — функция, возвращающая истину или ложь. Если её значение от первых двух аргументов — истина, то возвращается первый аргумент. Если ложь — второй.

Например, строки можно сравнивать двумя способами — по длине и по содержанию. При помощи шаблона удобно определить соответствующие функции нахождения наибольшей строки:

```
fun comp_len(x:string, y:string) = size(x) >= size(y);
fun comp_lex(x:string, y:string) = x >= y;
fun max_by_len(x, y) = apply_predicate(x, y, comp_len);
fun max_by_lex(x, y) = apply_predicate(x, y, comp_lex);
```

Определить функцию-аргумент можно непосредственно в месте вызова (безымянная функция):

```
fun max_by_len (x, y) =
  apply_predicate (x, y, fn (x, y) => size(x) >= size(y));
```

Для определения функции как композиции двух других функций используется специальный оператор `o`:

```
fun drop_negative x = if x < 0.0 then 0.0 else x;
val nonnegative_sin = drop_negative o Math.sin;
```

Вызов функции `nonnegative_sin(x)` эквивалентен вызову выражения `drop_negative(Math.sin(x))`.

Рекурсия. Пример простой рекурсии — функция вычисления суммы целых чисел от `0` до `n`.

```
fun sumUpTo 0 = 0
|   sumUpTo n = n + sumUpTo (n-1);
```

Кроме обычной рекурсии, в SML можно организовать взаимную рекурсию, при которой несколько функций вызывают друг друга:

```
fun even (n:int):bool =
  if (n=0) then true else odd (n-1)
and odd (n:int):bool =
  if (n=0) then false else even (n-1);
```

Отдельного упоминания заслуживает способ повышения эффективности рекурсии — так называемая ”хвостовая” рекурсия.

Расмотрим последовательность вычисления (упрощения) выражения `sumUpTo (3)`:

```
sumUpTo (3)
3 + sumUpTo (2))
3 + (2 + sumUpTo (1))
3 + (2 + (1 + sumUpTo (0)))
3 + (2 + (1 + 0))
3 + (2 + 1)
3 + 3
6
```

Вычисление выполняется за $(2n + 2)$ шагов, где n — аргумент функции. Кроме того, для хранения промежуточных значений требуется n ячеек памяти. Это весьма неэффективно. Хорошей альтернативой является использование промежуточного "хранилища" значения параметра — так называемого *аккумулятора*:

```
fun sumUpTo' (n:int):int =
  let
    fun sumUpToIter (n:int, acc:int):int =
      case n
        of 0 => acc
        |  _ => sumUpToIter (n-1, acc+n)
  in
    sumUpToIter (n,0)
  end;
```

Процесс вычисления теперь выглядит следующим образом:

```
sumUpTo' (3)
sumUpToIter (3,0)
sumUpToIter (2,3)
sumUpToIter (1,5)
sumUpToIter (0,6)
6
```

Вычисление завершается за $(n + 2)$ шагов, при этом используются всего лишь две целочисленные ячейки памяти. Такую рекурсию уже вполне можно считать итерацией.

Далее приведен вариант функции вычисления числа Фибоначчи с хвостовой рекурсией:

```
fun fib' (n) =
  let
    fun fib_iter (0,a,b) = b
    |   fib_iter (n,a,b) = fib_iter (n-1,a+b,a)
  in
    fib_iter (n,1,0)
  end;
```

2.3. Списки

Конструкторы списков. Список — это линейно упорядоченная последовательность однотипных элементов. Пустой список обозначается как `nil` или `[]`. Определение списка через его элементы: `[element_1, element_2, ..., element_n]`. Пример:

```
- [1,2,3];
val it = [1,2,3] : int list
```

Для присоединения в начало списка нового элемента используется оператор `::` (список начинается слева):

```
- 1::[2,3];
val it = [1,2,3] : int list
```

Оператор конкатенации (@) соединяет два списка:

```
- [1,2,3] @ [4,5];
val it = [1,2,3,4,5] : int list
```

Примеры списков:

```
[(2,3), (2,2), (9,1)]    : (int * int) list
[[], [1], [1,2]]         : int list list
[Math.sin, Math.cos]     : (real -> real) list
```

Оператор `::` однозначно структурирует непустой список, поэтому его можно использовать в шаблоне при сопоставлении:

```
- val x::t = [1,2,3,4,5];
val x = 1 : int
val t = [2,3,4,5] : int list
```

Приемы программирования списков. Практически любая функция, работающая со списками, имеет два шаблона (образца) аргумента — пустой список `nil` и выражение `h::t` (непустой список, содержащий как минимум один элемент `h`). Далее приведены определения функций нахождения длины списка, присоединения элемента к списку справа и переворота списка:

```
fun length nil = 0
|   length (h::t) = 1 + length t;
fun append (nil, l) = [l]
|   append (h::t, l) = h :: append (t, l);
fun rev nil = nil
|   rev (h::t) = rev t @ [h];
```

В языке `SML` имеются две специальные функции — `hd` и `tl`, возвращающие первый элемент и остаток списка соответственно:

```
- hd [1,2,3,4];
val it = 1 : int
- tl [1,2,3,4];
val it = [2,3,4] : int list
```

Для удобства работы со строками определены функции, преобразующие строку в список символов и наоборот — `explode` и `implode`:

```
- explode "abc";
val it = [#"a",#"b",#"c"] : char list
- implode [#"1",#"2"];
val it = "12" : string
```

(map) Функция, применяющая функцию-аргумент к элементам списка-аргумента:

```
fun map f nil = nil
|   map f (h::t) = (f h)::(map f t);
```

Рассмотрим следующие низкоуровневые функции:

```
fun doublist(nil) = nil
|   doublist(h::t) = 2*h :: doublist(t);
fun inclist(nil) = nil
|   inclist(h::t) = (h+1) :: inclist(t);
```

Первая удваивает элементы, вторая увеличивает их на единицу:

```
doublist [1,2,3,4] = [2,4,6,8]
inclist  [1,2,3,4] = [2,3,4,5]
```

Те же функции можно переопределить при помощи функции `map`:

```
val doublist = map (fn x=>2*x);
val inclist = map (fn x=> x+1);
```

(reduce) Функция, которая получает в качестве аргументов другую функцию, базовое значение и список значений, а затем последовательно применяет функцию-аргумент к списку, начиная с базового значения. Схема вычисления:

```
reduce f b [i1, i2, i3] = f(i1, f(i2, f(i3, b)))
```

Определение функции `reduce`:

```
fun reduce f b nil = b
|   reduce f b (h::t) = f(h, reduce f b t);
```

Функции суммирования списка и преобразования списка списков в один список:

```
fun sum(nil) = 0
|   sum(h::t) = h + sum(t);
fun flatten(nil) = nil
|   flatten(h::t) = h @ flatten(t);
```

Пример:

```
sum [10, 20, 30] = 60
flatten [[1,2],[3,4],[5,6,7]] = [1,2,3,4,5,6,7]
```

Те же функции `sum` и `flatten`, реализованные при помощи `reduce`:

```
val sum = reduce (fn(a,b)=>a+b) 0;
val flatten = reduce (fn(a,b)=>a@b) nil;
```

(zip) Функция, применяющая функцию-аргумент к элементам двух списков:

```
fun zip f nil nil = nil
|   zip f (h::t) (i::u) = f(h,i)::zip f t u;
```

Пример:

```
zip (op +) [1,2,3] [2,4,6] = [3,6,9]
```

(filter) Функция, которая возвращает список элементов исходного списка, удовлетворяющих условию (предикату).

```
fun filter f nil = nil
|   filter f (h::t) =
      (if f(h) then [h] else nil) @ (filter f t);
```

Пример:

```
fun even n = (n mod 2 = 0);
filter even [1,2,3,4,5,6] = [2,4,6]
```

Аккумулирование параметров. Простейшая функция "переворота" списка:

```
fun rev nil = nil
|   rev (h::t) = rev t @ [h];
```

Вычисление функции займёт $2n$ шагов, где n — длина списка. Определение с "аккумулятором" (трудоемкость — n шагов):

```
local
  fun helper (nil, a) = a
  |   helper (h::t, a) = helper (t, h::a)
in
  fun rev' l = helper (l, nil)
end;
```

3. Дополнительные возможности

3.1. Работа с типами данных

Переменная типа. Полиморфизм. Рассмотрим определение высокоуровневой функции-шаблона `f`, которая получает в качестве первого аргумента функцию `g`, в качестве второго аргумента — выражение `x`, и возвращает результат применения `g` к `x`:

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

Типы выражений нигде не указаны явно. Компилятор определил сигнатуру (тип) функции как `fn : ('a -> 'b) * 'a -> 'b`. Используемые здесь обозначения `'a` и `'b` — это так называемые переменные типа. Они показывают, что соответствующие выражения могут быть любого типа. Единственное ограничение — типы выражения `x` и аргумента выражения `g` должны совпадать (т.к. они обозначены одной и той же переменной `'a`).

Признаком переменной типа является символ апострофа перед именем типа. В качестве имени может выступать любой идентификатор.

Функция `f` из предыдущего примера может работать с данными любых типов и поэтому называется *полиморфной*. Такой полиморфизм называют также "истинным полиморфизмом", так как, в отличие от полиморфизма функций в императивных языках, здесь компилятор действительно для всех типов данных использует одну и ту же функцию, а не одноименные функции.

Полиморфизм является одним из основных приемов функционального программирования. Например, все функции работы со списками полиморфны, так как должны работать с элементами любого типа:

```
- fun length nil = 0
= |   length (_::t) = 1 + length t;
val length = fn : 'a list -> int
- length [1,2,3];
val it = 3 : int
- length [true,false];
val it = 2 : int
```

Двумя апострофами обозначается переменная типа, которая допускает замену только такими типами данных, для которых разрешена операция сравнения при помощи равенства (так называемые equality types). Например, поиск возможен только в списке, элементы которого можно сравнивать с образцом:

```
- fun find (l:''a list, y:''a):bool =
=   case l of  nil  => false
=          |  x::xs => (x=y) orelse find(xs, y);
val find = fn : ''a list * ''a -> bool
- find ([1, 2, 3, 4, 5], 4);
val it = true : bool
- find ([1.0, 2.0, 3.0], 2.0);
stdIn:30.1-30.23 Error: [equality type required]
```

Пользовательские типы. Объявление псевдонима для уже существующего типа (`type`):

```
- type pair_of_ints = int * int;
type pair_of_ints = int * int
- val a = (3,3);
val a = (3,3) : int * int
- val a : pair_of_ints = (3,3);
val a = (3,3) : pair_of_ints
```

Обратите внимание: контроль типов не определяет автоматически все подходящие выражения как выражения данного типа.

Объявление нового типа (`datatype`):

```
- datatype color = Red | Blue | Yellow;
datatype color = Blue | Red | Yellow
```

Здесь `Red`, `Blue`, `Yellow` — конструкторы. Их принято называть с большой буквы. Конструкторы могут обладать параметрами:

```
- datatype int_or_real = I of int | R of real;
datatype int_or_real = I of int | R of real
- val p = (I(55), R(55.0));
val p = (I 55, R 55.0) : int_or_real * int_or_real
```

Типы, подобные `int_or_real`, могут потребоваться для дополнительного ограничения типа выражения (здесь запрет на использование всех типов, кроме `int` и `real`).

Рекурсивные типы данных. На языке `SML` достаточно просто организуются данные, имеющие рекурсивную структуру. Рассмотрим в качестве примера определение бинарного дерева, каждый узел которого может содержать в качестве информационной составляющей значение какого-то типа `'a` (число, строку и т.п.):

1. Пустое дерево `Empty` есть бинарное дерево типа `'a`.

2. Если `tree1` и `tree2` — бинарные деревья, `val` — значение типа `'a`, то `Node(tree1,val,tree2)` — бинарное дерево типа `'a`.

3. Других деревьев нет.

Соответствующее определение на языке `SML`:

```
datatype 'a tree = Empty | Node of 'a tree * 'a * 'a tree;
```

Пример задания конкретного дерева типа `'a tree`:

```
val t = Node(Node(Empty, 5, Empty), 4, Empty);
```

Для работы с рекурсивным типом очевидным образом определяются рекурсивные функции:

```
fun height(Empty) = 0
|   height(Node(lft, _, rht)) =
        1 + Int.max(height(lft), height(rht));
```

```
fun size(Empty) = 0
|   size(Node(lft, _, rht)) =
        1 + size(lft) + size(rht);
fun inc(Empty) = Empty
|   inc(Node(lft, v, rht)) =
        Node(inc(lft), v + 1, inc(rht));
```

Определение произвольного дерева (не только бинарного):

```
datatype 'a tree = Empty | Node of 'a * 'a tree list;
```

Здесь у каждого узла не два потомка, а список потомков.

3.2. Императивное программирование в SML

Язык SML не является чистым функциональным языком, так как в нём всё же имеются механизмы, свойственные императивным языкам — работа с памятью и организация последовательности вычислений. Использование этих механизмов не соответствует стилю функционального программировании, но при решении некоторых прикладных задач без них не обойтись.

Ячейки памяти. Память представляет собой набор абстрактных ячеек. Тип у ячейки может быть любым (целое число, список, дерево и т.п.), он определяется в момент выделения памяти под ячейку. Функция выделения ячейки `ref` имеет тип `'a ->'a ref`. Аргумент функции — значение, которое записывается в ячейку первоначально. При каждом вызове создается новая ячейка.

```
- val r = ref 0;
val r = ref 0 : int ref
```

Для получения содержимого ячейки (хранимого там значения) используется оператор `!` типа `'a ref -> 'a`:

```
- !r;
val it = 0 : int
```

Для изменения содержимого ячейки используется оператор `:=` типа `'a ref * 'a -> unit`:

```
- r := 1;
val it = () : unit
```

Оператор присваивания не возвращает никакого значения, его вызов эквивалентен выражению:

```
val _ = r := 1;
```

Пример императивной последовательности операторов:

```
val r = ref 0;
val s = ref 0;
val _ = r := 3;
val x = !s + !r;
val t = r;
val _ = t := 5;
val y = !s + !r;
val z = !t + !r;
```

Результат: `x = 3, y = 5, z = 10`.

Функции типа `'a -> unit` называются *процедурами*, так как они ничего не возвращают и вызываются только ради производимого на память эффекта. Заметим, что процедуры никоим образом не могут затронуть своим эффектом "неимперативную" часть программы ("правильный" `SML`).

Ссылки допускают переприсваивание (создание алиасов):

```
val r = ref 0;
val s = ref 1;
val s = r;
```

Такое переприсваивание — один из важнейших источников ошибок в программе, поэтому его следует избегать.

Рассмотрим следующее выражение:

```
let
  val _ = exp1
in
  exp2
end;
```

Здесь выражение `exp1` вычисляется до вычисления выражения `exp2`. В сущности, получается *последовательное* вычисление двух выражений. Для того же самого можно использовать краткую запись: `exp1; exp2`.

Императивный факториал:

```
fun imperative_fact (n:int) =
  let
    val result = ref 1
    val i = ref 0
    fun loop () =
      if !i = n then ()
      else (i:=!i+1; result:=!result*!i; loop ())
  in
    loop (); !result
  end;
```

Ввод/вывод. Императивное программирование используется в `SML/NJ` при работе с файлами, так как с точки зрения операционной системы доступ к файлу на чтение или на запись представляет собой не одно, а несколько действий.

В `SML/NJ` имеется несколько специальных структур, содержащих функции ввода/вывода для различных типов интерфейсов, в частности, `TextIO`, `StreamIO`, `BinIO`. Далее приведены простейшие функции чтения и записи текстового файла:

```
fun fileToString fileName =
  let
    val stream = TextIO.openIn(fileName)
    val str = TextIO.inputAll(stream)
    val _ = TextIO.closeIn(stream)
  in
    str
  end;
fun stringToFile(str, fileName) =
  let
    val stream = TextIO.openOut(fileName)
```

```
    val _ = TextIO.output(stream, str)
    val _ = TextIO.closeOut(stream)
  in
    str
  end;
```

В библиотеках языка имеются и другие структуры, в которых собраны функции работы с операционной системной, например, `OS`, `Date` и `Time`.

3.3. Исключения

Существует набор стандартных исключений, например, `Overflow` и `Dividebyzero`, но можно объявлять и свои собственные. Синтаксис объявления исключения и его вызова:

```
exception <имя исключения>;
raise <имя исключения>;
```

Блок обработки исключений ставится непосредственно в конце того выражения, которое может их вызвать. Синтаксис:

```
<выражение>
handle <имя исключения 1> => <выражение-обработчик 1>
...
|      <имя исключения n> => <выражение-обработчик n>;
```

Пример обработки исключений при вычислении факториала:

```
exception MyException;
local
  fun fact 0 = 1
  |   fact n = n * fact (n-1)
in
  fun checked_factorial n = (
    if n >= 0 then Int.toString(fact n)
    else raise MyException)
    handle MyException => "Out of range"
    |      Overflow  => "Integer overflow"
end;
```

Исключения с параметром могут быть использованы для передачи данных. Расширенный синтаксис:

```
exception <имя исключения> of <тип>;
raise <имя исключения> <выражение соответствующего типа>;
<выражение> handle <имя исключения> <аргумент> =>
  <выражение-обработчик, использующее аргумент>;
```

Пример исключения с параметром:

```
exception SyntaxError of string;
...
raise SyntaxError "Integer expected"
...
raise SyntaxError "Identifier expected"
...
handle SyntaxError msg => print "Syntax error: " ^ msg;
```

Еще один интересный пример использования исключений в SML — поддержка перебора с возвратом. Рассмотрим в качестве примера функцию определения сдачи:

```
exception Change;
fun change _ 0 = nil
|   change nil _ = raise Change
|   change (coin::coins) amt =
  if coin > amt then
    change coins amt
  else
    (coin :: change (coin::coins) (amt-coin))
      handle Change => change coins amt;
```

Даны список номиналов монет в кошельке (первый аргумент) и сумма сдачи, которую нужно набрать этими монетами. Число монет любого номинала не ограничено. Функция change реализует “жадный” алгоритм и всегда стремится набрать сумму самыми первыми номиналами монет из списка. В случае попадания в тупик программа посредством обработки исключения возвращается к ближайшему “жадному” решению и выбирает альтернативный путь. Ответ не всегда существует (например, нельзя набрать 11

рублей при наличии только монет по 5 рублей и по 2 рубля), но в случае наличия решения функция всегда его найдёт. В случае отсутствия решения исключение `Change` будет выброшено наружу.

3.4. Модульное программирование

В языке `SML` имеется механизм разбиения исходного кода на отдельные блоки — модули. Функционально и даже синтаксически он немного напоминает механизм классов в объектно-ориентированных языках. Используются три вида элементов — структуры, сигнатуры и функторы.

Структуры. Структура — это набор компонентов (типов, функций, значений и т.д.), сгруппированных в отдельный модуль. Примеры структур — математический модуль `Math` и модуль `OS`, содержащий интерфейсы системных функций.

Определим структуру, в которой инкапсулированы все средства для работы с типом данных "реверсивный стек" (стек с возможностью "переворота"):

```
structure RevStack = struct
  type 'a stack = 'a list
  exception Empty
  val empty = []
  fun isEmpty (s:'a stack):bool =
    (case s of [] => true | _ => false)
  fun top (s:'a stack):'a =
    (case s of [] => raise Empty | x::xs => x)
  fun pop (s:'a stack):'a stack =
    (case s of [] => raise Empty | x::xs => xs)
  fun push (s:'a stack,x:'a):'a stack = x::s
  fun rev (s:'a stack):'a stack =
    (case s of [] => [] | x::xs => rev (xs) @ [x])
end;
```

Здесь объявлены: тип данных `'a stack`, исключение `Empty` (обо-

значающее некорректное обращение к пустому стеку), выражение `empty` (константа, обозначающая пустой стек), а также функции проверки на пустоту, чтения вершины, стирания вершины, записи в вершину и переворота стека.

При обращении к компонентам структуры нужно указывать её имя (через точку):

```
- val st = [1,2,3] : int RevStack.stack;
val st = [1,2,3] : int RevStack.stack
- RevStack.rev st;
val it = [3,2,1] : int RevStack.stack
```

В рассмотренном примере использовался базовый синтаксис объявления структуры:

```
structure <имя структуры> = struct <компоненты> end;
```

Есть ещё один вариант синтаксиса:

```
structure <имя структуры> = <объявление структуры>;
```

В разделе `<объявление структуры>` можно указать имя другой структуры (назначение псевдонима):

```
structure AlsoRevStack = RevStack;
```

Более сложные варианты создания структур (при помощи сигнатур и функторов) будут описаны в следующих параграфах.

Возможно объявление вложенных структур:

```
structure X = struct
  structure Y = struct
    val z = 0
  end
end;
```

Обращение к компоненту — `X.Y.z`.

Сигнатуры. Сигнатуру можно рассматривать как тип структуры. Точнее, сигнатура — это набор типов компонентов структуры. Сигнатуры принято называть заглавными буквами:

```
signature REV_STACK = sig
  type 'a stack
  exception Empty
  val empty : 'a stack
  val isEmpty : 'a stack -> bool
  val push : 'a stack * 'a -> 'a stack
  val pop : 'a stack -> 'a stack
  val top : 'a stack -> 'a
  val rev : 'a stack -> 'a stack
end;
```

Сигнатура — это спецификация интерфейса некоей абстрактной структуры. Одной сигнатуре может соответствовать множество структур (как одному интерфейсу может соответствовать множество реализаций).

Важно понять, что сигнатура может существовать сама по себе и быть не связанной ни с одной структурой. Например, мы можем объявить сигнатуру, описывающую интерфейс модуля работы с обычным стеком:

```
signature STACK = sig
  type 'a stack
  exception Empty
  val empty : 'a stack
  val isEmpty : 'a stack -> bool
  val push : 'a stack * 'a -> 'a stack
  val pop : 'a stack -> 'a stack
  val top : 'a stack -> 'a
end;
```

В отличие от сигнатуры `REV_STACK`, здесь нет функции переворота. Типы всех остальных компонентов совпадают.

Сигнатура `STACK` объявлена, однако её наполнение пока что отсутствует. Мы можем заново описывать все функции, создавая с нуля новую структуру, но можем использовать структуры с подходящим интерфейсом, например, `RevStack`:

```
structure Stack : STACK = RevStack
```

Теперь при обращении к структуре `Stack` доступны все компоненты `RevStack`, кроме функции `rev`.

Механизм выглядит похожим на наследование в объектно-ориентированном программировании, однако это скорее не наследование, а ограничение доступа (видимости). Новых компонентов (типов, функций и т.д.) не создается, все объявления из `RevStack` остаются в неприкосновенности, просто их не видно через `Stack`.

Рассмотрим ещё один пример ограничения доступа при помощи сигнатур:

```
structure A = struct
  val a = ref (0)
  val b = true
end;
signature ONLY_A = sig
    val a : int ref
end;
structure B : ONLY_A = A;
```

Присваивание `B.a:=1` изменит и содержимое ячейки `A.a` (это одна и та же ячейка):

```
- B.a := 1;
val it = () : unit
- !A.a;
val it = 1 : int
```

Функторы. Функтор — это параметризованная структура, зависящая от другой структуры (о которой нам известна только её сигнатура, которая и выступает в качестве параметра).

Синтаксис объявления функтора:

```
functor <имя> (X:<имя сигнатуры>) =
  <параметризованная структура (с параметром X)>
```

Объявление функтора, реализующего работу с очередью:

```
functor Queue(S:REV_STACK) = struct
  type 'a queue = ('a S.stack * 'a S.stack)
  val empty = (S.empty,S.empty)
  fun isEmpty ((inS,outS):'a queue):bool =
    S.isEmpty (inS) andalso S.isEmpty (outS)
  fun enqueue ((inS,outS):'a queue,x:'a):'a queue =
    if (S.isEmpty (outS)) then (inS,S.push (outS,x))
    else (S.push (inS,x),outS)
  fun head ((inS,outS):'a queue):'a =
    if (S.isEmpty (outS)) then raise Empty
    else S.top (outS)
  fun dequeue ((inS,outS):'a queue):'a queue =
    if (S.isEmpty (outS)) then raise Empty
    else let
      val _ = S.top (outS)
      val xs = S.pop (outS)
    in
      if (S.isEmpty (xs)) then (S.empty,S.rev (inS))
      else (inS,xs)
    end
end;
```

Здесь в качестве базовой структуры (параметра) используется абстрактная структура `S` с сигнатурой `REV_STACK`. Посредством вызова функтора `Queue` от какого-то аргумента ("конкретной" структуры) мы создаём новую "конкретную" структуру для работы с очередью:

```
structure QueueStru = Queue(RevStack);
structure QueueStru2 = Queue(RevStack2);
```

В структуре `QueueStru` работа с данными осуществляется при помощи методов из `RevStack`, в структуре `QueueStru2` — каким-либо другим способом.

Литература

[1] Бежанова, М.М. Современные понятия и методы программирования / М.М. Бежанова М.М., И.В. Поттосин. — М.: Научный мир, 2000.

[2] Зыков, С.В. Современные языки программирования / С.В. Зыков. — Ч.I. Функциональный подход к программированию. — М.: МИФИ, 2003.

[3] Городняя, Л.В. Основы функционального программирования: курс лекций / Л.В. Городняя. — М.: ИнтУИТ, 2004.

[4] Непейвода, Н.Н. Стили и методы программирования / Н.Н. Непейвода. — М.: ИнтУИТ, 2005.

[5] Филд, А. Функциональное программирование / А. Филд, П. Харрисон. — М.: Мир, 1993.

[6] Хендерсон, П. Функциональное программирование: применение и реализация / П. Хендерсон. — М.: Мир, 1983.

[7] Harper, R. Programming in Standard ML / R. Harper. — 2005. `http://www.cs.cmu.edu/~rwh/introsml/`

[8] Pucella, R. Notes on Programming Standard ML of New Jersey / R. Pucella. — 2001. `http://www.cs.cornell.edu/riccardo/prog-smlnj/notes.pdf`

[9] Cumming, A. A Gentle Introduction to ML / A. Cumming. — 1998. `http://www.dcs.napier.ac.uk/course-notes/sml/manual.html`

[10] Gordon, M. Introduction to Functional Programming / M. Gordon. — 1996. `http://www.cl.cam.ac.uk/teaching/FuncProg/FuncProg.html`

Учебное издание

Башкин Владимир Анатольевич

Функциональное программирование на языке SML

Методические указания

Редактор, корректор И.В. Бунакова
Компьютерный набор, вёрстка В.А. Башкина

Подписано в печать 25.06.07. Формат 60×84/16.
Бумага тип. Усл. печ. л. 2,32. Уч.-изд. л. 1,5.
Тираж 100 экз. Заказ

Оригинал-макет подготовлен в редакционно-издательском отделе ЯрГУ

150000 Ярославль, ул. Советская, 14