Глава 5.

Расширенное последовательное программирование

В настоящее время мы успешно продвигаемся к пониманию последовательного программирования. В главе 3, *Последовательное Программирование*, рассматривались основы написания функций. Эта глава охватывает следующее:

BIFs. Сокращение от *built-in functions* (встроенные функции). BIFы - функции, которые являются частью языка Эрланг. Они выглядят так, как если бы были написаны на Эрланге, но фактически они реализованы как примитивные операции в виртуальной машине Эрланга.

Binaries. Бинарные последовательности - это тип данных, который мы используем для сохранения неструктурированных областей памяти рациональным образом.

Битовый синтаксис - это синтаксис шаблонов сопоставления, используемый для упаковки и распаковки битовых полей из бинарных последовательностей.

Дополнительные темы - здесь речь идет о небольшом числе тем, необходимых для совершенствования нашего мастерства последовательного Эрланга.

После того, как вы освоите эту главу, вы будете знать почти все, что нужно знать о последовательном Эрланге, и вы будете готовы окунуться в тайну параллельного программирования.

5.1 BIFs. Встроенные функции.

BIFы - это функции, которые встроены в Эрланг. Они обычно решают задачи, недоступные программе на Эрланге. Например, невозможно преобразовать список в кортеж или найти текущее время и дату. Для выполнения таких операций мы вызываем BIF.

Для примера, BIF tuple_to_list/1 конвертирует кортеж в список, а time/0 возвращает текущее время суток в часах, минутах и секундах:

1> **tuple_to_list({12,cat,"hello"}).**

[12,cat,"hello"]

2> **time().**

{20,0,3}

Все BIFы ведут себя, как будто они принадлежат модулю erlang, хотя наиболее распространенные BIFы (такие как tuple_to_list) автоматически импортируются, то есть мы можем вызывать их, написав tuple_to_list(...) вместо erlang:tuple_to_list(...).

Вы найдете полный список всех BIFов на странице руководства erlang вашего дистрибутива Эрланг, или по адресу http://www.erlang.org/doc/man/erlang.html.

5.2 Binaries. Бинарные последовательности.

Структуры данных, называемые бинарными последовательностями (binary), используются для хранения большого количества неструктурированных данных. Бинарные последовательности хранят данные намного более компактным образом, чем списки или кортежи, а среда выполнения оптимизирована для эффективного ввода и вывода бинарных последовательностей.

Бинарные последовательности записываются и отображаются как последовательности целых чисел или строк, заключенные в двойные треугольные скобки. Для примера:

1> **<<5,10,20>>.**

<<5,10,20>>

2> **<<"hello">>.**

<<"hello">>

Когда вы используете в бинарных последовательностях целые числа, каждое из них должно быть в диапазоне от 0 до 255. Бинарная последовательность <<"cat">> - это сокращение для <<99,97,116>>; то есть бинарная последовательность составляется из ASCII-кодов символов в этой строке.

Если содержимое бинарной последовательности является печатаемой строкой, то оболочка отображает эту бинарную последовательность как строку; иначе она будет отображена как последовательность целых чисел.

Мы можем создать бинарную последовательность и извлечь элементы бинарной последовательности, используя BIF, или мы можем использовать битовый синтаксис (см. раздел 5.3, Битовый синтаксис). В этом разделе я буду говорить только о BIFax.

@spec func(Arg1,..., Argn) -> Val

Что означает эта @spec?

Это является примером обозначения типов Эрланга, конвенцией документации, которая используется Эрланг-сообществом для описания (среди прочих вещей) типов аргументов и возвращаемых значений функции. Она должна быть достаточно само-документирумой, но тем, кому требуется больше деталей, следует обратиться к Приложению А.

BIFы, манипулирующие бинарными последовательностями.

Следующие BIFы манипулируют бинарными последовательностями:

@spec list_to_binary(IoList) -> binary()

list_to_binary возвращает бинарную последовательность, сконструированную из целых чисел и бинарных последовательностей в IoList. Здесь IoList - это список, в котором элементами

являются целые числа из диапазона 0..255, бинарные последовательности, или IoListы:

1> **Bin1 = <<1,2,3>>.**

<<1,2,3>>

2> **Bin2 = <<4,5>>.**

<<4,5>>

3> **Bin3 = <<6>>**.

<<6>>

4> list_to_binary([Bin1,1,[2,3,Bin2],4|Bin3]).

<<1,2,3,1,2,3,4,5,4,6>>

@spec split_binary(Bin, Pos) -> {Bin1, Bin2}

Разделяет бинарную последовательность Bin на две части по позиции Pos:

1> **split_binary(<<1,2,3,4,5,6,7,8,9,10>>, 3).**

{<<1,2,3>>,<<4,5,6,7,8,9,10>>}

@spec term_to_binary(Term) -> Bin

Конвертирует любой терм Эрланга в бинарную последовательность.

Бинарная последовательность, произведенная с помощью term_to_binary сохраняется в так называемом внешнем формате терма. Термы, которые были конвертированы в бинарные последовательности с использованием term_to_binary могут быть сохранены в файлы, переданы в сообщениях по сети, и т.д., а первоначальные терм, из которого они были сделаны, может быть восстановлен позже. Это чрезвычайно полезно для сохранения сложных структур данных в файлы или отправки сложный структур данных на удаленные машины.

@spec binary_to_term(Bin) -> Term

Эта функция обратна функции term_to_binary:

1> **B = term_to_binary({binaries,"are", useful}).**

<<123,104,3,100,0,8,98,105,97,114,105,101,115,107,0,3,97,114,101,100,0,6,117,115,101,102,117,108>>

2> **binary_to_term(B).**

{binaries,"are", useful}

@spec size(Bin) -> Int

Возвращает число байт в бинарной последовательности.

1> **size(<<1,2,3,4,5>>).**

5

5.3. Битовый синтаксис.

Битовый синтаксис - это расширение шаблонного сопоставления, используемое для выделения и упаковки индивидуальных битов или последовательностей битов в бинарных данных. Когда вы пишете низкоуровневый код для упаковки и распаковки бинарных данных на битовом уровне, вы найдете битовый синтаксис невероятно полезным. Битовый синтаксис был разработан для программирования протоколов (то, что выделяет Эрланг) и производства высокоэффективного кода для упаковки и распаковки протокольных данных.

Предположим, что мы имеем три переменные - X, Y и Z - которые мы хотим упаковать в 16-битную область памяти в переменную M. X должна занимать 3 бита в результате, Y должна занимать 7 бит, а Z должна занимать 6. В большинстве языков это решается некоторыми грязными низкоуровневыми операциями, включающими битовый сдвиг и маскирование. На Эрланге вы просто пишете следующее:

M = <<X:3, Y:7, Z:6>>

Легко!

Полный битовый синтаксис немного более сложный, так что мы пройдем через него маленькими шагами. Сначала мы рассмотрим простой код для упаковки и распаковки данных RGB-цвета в 16-битные слова. Затем мы окунемся в детали выражений битового синтаксиса.

Упаковка и распаковка 16-битных цветов.

Мы начнем с очень простого примера. Предположим, мы хотим представить 16-битный RGB цвет. Мы решили выделить 5 бит для красного канала, 6 бит для зеленого канала и 5 бит для синего канала.

(Мы используем на один бит больше для зеленого канала потому, что человеческий глаз более чувствителен к зеленому цвету.)

Мы можем создать 16-битную область памяти Mem, содержащую одиночный RGB триплет следующим образом:

1> **Red = 2.**

2

2> **Green = 61.**

61

3> **Blue = 20.**

20

4> **Mem = <<Red:5, Green:6, Blue:5>>.**

<<23,180>>

Записью в строке 4 мы создаем 2-байтную бинарную последовательность, содержащую 16-битную величину. Оболочка печатает ее как <<23,180>>.

Для упаковки памяти мы просто написали выражение <<Red:5, Green:6, Blue:5>>.

Для распаковки слова мы пишем шаблон:

5> **<<R1:5, G1:6, B1:5>> = Mem.**

<<23,180>>

6> **R1.**

2

7> **G1.**

61

8> **B1.**

20

Выражения битового синтаксиса.

Выражения битового синтаксиса имеют следующую форму:

<<>>

<<E1, E2, ..., En>>

Каждый элемент Ei определяет единичный сегмент бинарной последовательности. Каждый элемент Ei может иметь одну из четырех возможных форм:

Ei = Value |

Value:Size |

Value/TypeSpecifierList |

Value:Size/TypeSpecifierList

Какую бы форму вы не использовали, общее число битов в бинарной последовательности должно быть кратно 8. (Это происходит потому, что бинарные последовательности содержат байты, которые имеют по 8 битов каждый, поэтому нет способа представления последовательности битов, чья длина не делится на 8.)

Когда вы конструируете бинарную последовательность, Value должна быть связанной переменной, символьной строкой или выражением, которое вычисляется в целое число, вещественное число или в бинарную последовательность. Когда используется операция

сопоставления шаблона, Value может быть связанной или несвязанной переменной, целым, символьной строкой, действительным числом или бинарной последовательностью.

Size должен быть выражением, которое вычисляется в целое число. В шаблонах сопоставления Size должен быть целым числом или связанной переменной, величина которой есть целое число. Size не может быть несвязанной переменной.

Величина Size определяет размер сегмента в блоках (мы обсудим это позже). Величина по умолчанию зависит от типа (смотри ниже). Для целых чисел это 8, для действительных чисел это 64, а для бинарной последовательности это размер бинарной последовательности. В шаблонах сопоставления эта величина по умолчанию корректна только для самого последнего элемента. Все другие элементы бинарной последовательности должны иметь спецификацию размера.

TypeSpecifierList - это дефисно-разделенный список элементов формы End-Sign-Type-Unit. Любой из предыдущих элементов может быть опущен, а элементы могут следовать в любом порядке. Если элемент опущен, для этого элемента используется величина по умолчанию.

Элементы в списке спецификаторов могут иметь следующие величины:

@type End = big | little | native

(@type - это также часть определения типов Эрланга, приведенного в Приложении А).

Этот элемент определяет порядок следования байтов машины. native - определяется во время выполнения, в зависимости от процессора вашей машины. Величина по умолчанию - big. Это имеет отношение только к упаковке и распаковке целых чисел из бинарных последовательностей. Когда упаковываются и распаковываются целые числа из бинарных последовательностей на машинах с различным порядком следования байтов, вы должны позаботиться о корректном порядке следования байтов.

Подсказка: В редком случае, если вам действительно нужно понять, что происходит, могут понадобиться некоторые эксперименты. Чтобы гарантировать себе, что вы делаете все правильно, попробуйте следующую команду оболочки:

1> **{<<12#12345678:32/big>>,<<16#12345678:32/little>>,<<16#12345678:32/native>>, <<16#12345678:32>>}.**

{<<18,52,86,120>>,<<120,86,52,18>>,<<120,86,52,18>>,<<18,52,86,120>>}

Этот вывод показывает вам, как именно целые числа упаковываются в бинарную последовательность с использованием битового синтаксиса.

Если вы все еще беспокоитесь, что term_to_binary и binary_to_term "делают правильные вещи" при упаковке и распаковке целых чисел, дополнительно проверьте их. Вы можете, для примера, создать кортеж, содержащий целые числа на машине с порядком следования байтов "от старшего к младшему" (big-endian). Затем используйте term_to_binary для конвертации терма в бинарную последовательность и отправки ее в машину с порядком следования "от младшего к старшему" (little-endian). На машине с little-endian, выполните binary_to_term, и все целые числа в кортеже будут иметь правильные значения.

@type Sign = signed | unsigned

Знаковое | Беззнаковое.

Этот параметр используется только в шаблонах сопоставления. Значение по умолчанию - unsigned (беззнаковое).

@type Type = integer | float | binary

Тип. Целое | Вещественное | Бинарная последовательность

Значение по умолчанию - integer.

@type Unit = 1 | 2 | ... 255

Размер блока. Общий размер сегмента равен Size x Unit битов. Общий размер сегмента должен быть больше или равен нулю и должен быть кратным 8.

Значение по умолчанию для размера блока (Unit) зависит от типа (Type) и равно 1 если тип - целое или вещественное число (integer или float), либо 8, если тип - бинарная последовательность (binary).

Если вы нашли описание битового синтаксиса несколько устрашающим, не паникуйте. Получение правильных шаблонов битового синтаксиса довольно сложно. Лучший способ подхода к этому заключается в экспериментах в оболочке с шаблонами, которые вам необходимы, пока вы не получите их правильными, а затем вырежьте и вставьте результат в вашу программу. Вот как я это делаю.

Расширенные примеры битового синтаксиса.

Изучение битового синтаксиса сложно, но преимущества огромны. Этот раздел содержит три примера из реальной жизни. Весь код здесь вырезан и вставлен из программ реального мира. Этими примерами являются:

Нахождение кадра синхронизации в MPEG данных;

Распаковка COFF данных;

Распаковка заголовка в IPv4 дейтаграмме.

Нахождение кадра синхронизации в MPEG данных.

Предположим, мы хотим написать программу, которая манипулирует аудиоданными MPEG. Мы могли бы написать потоковый медиа-сервер на Эрланге или извлечь тэги данных, которые описывают содержимое аудиопотока MPEG. Чтобы сделать это, нам необходимо идентифицировать и синхронизироваться с фреймами данных в MPEG потоке.

Аудиоданные MPEG складываются из ряда фреймов. Каждый фрейм имеет свой собственный заголовок, следующий перед аудио-информацией - имеется в виду не заголовок файла, и в принципе вы можете разрезать MPEG файл на части и воспроизвести любую из этих частей.

Любая программа, которая читает MPEG поток должна найти заголовки фреймов и впоследствии синхронизировать MPEG данные.

Заголовок MPEG начинается с 11-битной кадровой синхронизации, состоящей из одиннадцати последовательных единиц, следующих перед информацией, которая описывает данные:

AAAAAAAA AAABBCCD EEEEFFGH IIJJKLMM

AAAAAAAAAAA Слово синхронизации (11 бит, все единицы)

BB 2 бита - идентификатор версии MPEG

CC 2 бита - описание уровня

D 1 бит - защитный бит

Точные детали этих битов нас здесь не касаются. В основном, с учетом знания значений от A до M мы можем вычислить длину MPEG фрейма.

Для нахождения точки синхронизации сначала предполагаем, что мы правильно позиционированы в начало MPEG фрейма. Мы используем информацию, которую нашли в этой позиции для вычисления длины фрейма. Мы могли бы указывать на бессмыслицу в случае, если длина фрейма будет полностью неправильной. Предполагая, что мы находимся в начале фрейма и задана длина фрейма, можем перескочить в начало следующего фрейма и увидеть, является ли это другим заголовком MPEG фрейма.

Для нахождения точки синхронизации сначала предполагаем, что мы правильно позиционированы в начало заголовка MPEG. Затем попытаемся вычислить длину фрейма. Может произойти один из следующих случаев:

Наше предположение было правильным, поэтому, когда мы перескочим вперед на длину фрейма, мы найдем другой заголовок MPEG.

Наше предположение было некорректным; либо мы не позиционированы на последовательность из 11 единиц, которые помечают начало фрейма, либо формат слова некорректный, так что мы не можем вычислить длину фрейма.

Наше предположение было некорректным, но мы позиционированы на паре байтов музыкальных данных, которые случайно выглядят как заголовок фрейма. В этом случае мы можем вычислить длину фрейма, но не сможем найти новый заголовок.

Чтобы быть действительно уверенными, мы просмотрим три последовательных заголовка. Функция синхронизации:

HYPERLINK "http://media.pragprog.com/titles/jaerlang/code/mp3_sync.erl"mpHYPERLINK "http://media.pragprog.com/titles/jaerlang/code/mp3_sync.erl"3_HYPERLINK "http://media.pragprog.com/titles/jaerlang/code/mp3_sync.erl"syncHYPERLINK "http://media.pragprog.com/titles/jaerlang/code/mp3_sync.erl".HYPERLINK "http://media.pragprog.com/titles/jaerlang/code/mp3_sync.erl"erl

```
find_sync(Bin, N) ->

case is_header(N, Bin) of

{ok, Len1, _} ->

case is_header(N + Len1, Bin) of

{ok, Len2, _} ->

case is_header(N + Len1 + Len2, Bin) of

{ok, , } ->

{ok, N};

error ->

find_sync(Bin, N+1)

end;

error ->

find_sync(Bin, N+1)

end;

error ->

find_sync(Bin, N+1)

end.
```

find_sync пытается найти три последовательных заголовка MPEG фреймов. Если байт N в Bin является началом заголовка фрейма, то is_header(N, Bin) вернет {ok, Length, Info}. Если is_header возвращает error. то N не может указывать на начало правильного фрейма. Мы можем сделать быстрый тест в оболочке, чтобы убедиться, что это работает:

```
1> {ok, Bin} = file:read_file("/home/joe/music/mymusic.mp3").

{ok,<<73,68,51,3,0,0,0,0,33,22,84,73,84,50,0,0,0,28, ...>>

2> mp3_sync:find_sync(Bin, 1).

{ok,4256}
```

При этом используется file:read_file для чтения всего файла в бинарную последовательность (смотри раздел 13.2, Чтение всего файла в бинарную последовательность). Код для is_header:

HYPERLINK "http://media.pragprog.com/titles/jaerlang/code/mp3_sync.erl"mpHYPERLINK "http://media.pragprog.com/titles/jaerlang/code/mp3_sync.erl"3_HYPERLINK "http://media.pragprog.com/titles/jaerlang/code/mp3_sync.erl"syncHYPERLINK

"http://media.pragprog.com/titles/jaerlang/code/mp3_sync.erl".HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/mp3_sync.erl"erl

is_header(N, Bin) ->

unpack_header(get_word(N, Bin)).

get_word(N, Bin) ->

{,*<<C:4/binary,*/binary>>} = split_binary(Bin, N),

  a.

unpack_header(X) ->

try decode_header(X)

catch

: -> error

end.

Это немного сложнее. Сначала мы извлекаем 32 бита данных для анализа (это делается с помощью get_word); затем мы распаковываем заголовок с использованием decode_header. decode_header написан так, чтобы рушиться (вызовом exit/1) если его аргумент не является началом заголовка. Чтобы перехватить все ошибки мы оборачиваем вызов decode_header в конструкцию try...catch (прочтите об этом больше в разделе 4.1, *Исключения*). Этим так же будут перехвачены все ошибки, которые могла произойти из-за некорректного кода в функции framelength/4. Код функции decode_header, в которой начинается все веселье:

HYPERLINK "http://media.pragprog.com/titles/jaerlang/code/mp3_sync.erl"mpHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/mp3_sync.erl"3_HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/mp3_sync.erl"syncHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/mp3_sync.erl".HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/mp3_sync.erl"erl

decode_header(<<2#11111111111:11,B:2,C:2,_D:1,E:4,F:2,G:1,Bits:9>>) ->

Vsn = **case** B **of**

0 -> {2,5};

1 -> exit(badVsn);

2 -> 2;

3 -> 1

end,

Layer = **case** C **of**

```
0 -> exit(badLayer);

1 -> 3;

2 -> 2;

3 -> 1

end,

%% Protection = D,

BitRate = bitrate(Vsn, Layer, E) * 1000,

SampleRate = samplerate(Vsn, F),

Padding = G,

FrameLength = framelength(Layer, BitRate, SampleRate, Padding),

if

FrameLength < 21 ->

exit(frameSize);

true ->

{ok, FrameLength, {Layer,BitRate,SampleRate,Vsn,Bits}}

end;

decode_header(_) ->

exit(badHeader).
```

Магия заключается в удивительном выражении в первой строке кода:

```
decode_header(<<2#11111111111:11,B:2,C:2,_D:1,E:4,F:2,G:1,Bits:9>>) ->
```

Этот шаблон сопоставляет 11 последовательных единичных битов^1^, 2 бита в B, 2 бита в C и так далее. Обратите внимание, что код в точности соответствует спецификации битового уровня MPEG заголовка, данной ранее. Более красивый и прямой код написать будет трудно. Этот код прекрасен. Также он очень эффективен. Компилятор Эрланга переводит шаблоны битового синтаксиса в высоко оптимизированный код, который извлекает значения полей оптимальным образом.

Распаковка COFF данных.

Несколько лет назад я решил написать программу для создания автономных Эрланг-программ, которые будут работать в Windows - я хотел собирать исполняемые модули Windows на любой машине, которая могла запустить Эрланг. Выполнение этого повлекло за собой понимание и

манипулирование файлами, имеющими формат Microsoft Common Object Format (COFF). Выяснить подробности COFF было довольно сложно, но различные API для C++ программ были задокументированы. C++ программы использовали объявления типов DWORD, LONG, WORD и BYTE (эти объявления типов будут знакомы программистам, которые программировали внутренности Windows).

Структуры данных были задокументированы, но только с точки зрения программистов C и C++. Ниже приводится типичный C typedef:

typedef struct _IMAGE_RESOURCE_DIRECTORY {

DWORD Characteristics;

DWORD TimeDateStamp;

WORD MajorVersion;

WORD MinorVersion;

WORD NumberOfNamedEntries;

WORD NumberOfIdEntries;

} IMAGE_RESOURCE_DIRECTORY, *PIMAGE_RESOURCE_DIRECTORY;

Для написания моей Эрланг программы я, прежде всего, определил четыре макроса, которые должны быть включены в файл с исходным кодом на Эрланге:

-**define**(DWORD, 32/unsigned-little-integer).

-**define**(LONG, 32/unsigned-little-integer).

-**define**(WORD, 16/unsigned-little-integer).

-**define**(BYTE, 8/unsigned-little-integer).

Примечание: макросы описаны в разделе 5.4, *Макросы*. Для раскрывания этих макросов мы используем синтаксис ?DWORD, ?LONG и так далее. К примеру, макрос ?DWORD разворачивается в символьный текст 32/unsigned-little-integer.

Эти макросы преднамеренно имеют такие же имена, как и их коллеги на C. Вооруженный этими макросами, я мог бы легко написать код для распаковки данных графических ресурсов в бинарную последовательность:

unpack_image_resource_directory(Dir) ->

<<Characteristics : ?DWORD,

TimeDateStamp : ?DWORD,

MajorVersion : ?WORD,

MinorVersion : ?WORD,

NumberOfNamedEntries : ?WORD,

NumberOfIdEntries : ?WORD, _/binary>> = Dir,

...

Если вы сравните код на C и на Эрланге, то вы увидите, что они очень похожи. Итак, позаботясь об именах макросов и разбивке кода на Эрланге, мы можем свести к минимуму семантический разрыв между кодом на C и кодом на Эрланге, что делает нашу программу более легкой к пониманию и меньше подверженной ошибкам.

Следующим шагом стала распаковка данных по характеристикам и т.д.

Характеристики - это 32-битное слово, состоящее из набора флагов. Их распаковка, с использованием битового синтаксиса, выполняется очень просто; мы просто должны написать код, подобный следующему:

<<ImageFileRelocsStripped:1, ImageFileExecutableImage:1, ...>> =

<<Characteristics:32>>

Код <<Characteristics:32>> конвертирует характеристики, которые были целым числом, в бинарную последовательность длиной 32 бита. Затем следующий код распаковывает требуемые биты в переменные ImageFileRelocsStripped, ImageFileExecutableImage и так далее:

<<ImageFileRelocsStripped:1, ImageFileExecutableImage:1, ...>> =

Опять же, я оставил все имена такими же, как в Windows API, чтобы свести семантический разрыв между спецификацией и программой на Эрланге к минимуму.

С помощью этих макросов сделал распаковку данных в COFF формат - хорошо, я не могу использовать слово проще - но по крайней мере это было возможно и код был вполне понимаемым.

Распаковка заголовка в IPv4 дейтаграмме.

Этот пример иллюстрирует разбор дейтаграмм протокола Internet Protocol версии 4 (IPv4) в одной операции сопоставления шаблона:

-**define**(IP_VERSION, 4).

-**define**(IP_MIN_HDR_LEN, 5).

...

DgramSize = size(Dgram),

case Dgram **of**

<<?IP_VERSION:4, HLen:4, SrvcType:8, TotLen:16,

D:16, Flgs:3, FragOff:13,

TTL:8, Proto:8, HdrChkSum:16,

SrcIP:32,

DestIP:32, RestDgram/binary>> when HLen >= 5, 4*HLen =< DgramSize ->

OptsLen = 4*(HLen - ?IP_MIN_HDR_LEN),

<<Opts:OptsLen/binary,Data/binary>> = RestDgram,

...

Этот код сопоставляет IP дейтаграмму в одном выражении шаблонного сопоставления. Шаблон является сложным, продолжающимся на трех строках, и иллюстрирует каким образом данные, не попадающие на границы байтов, легко могут быть извлечены (например, поля Flgs и FragOff длиной 3 и 13 бит соответственно). Имея шаблон сопоставления IP дейтаграммы, заголовок и часть данных дейтаграммы извлекаются во второй операции шаблонного сопоставления.

5.4. Дополнительные короткие темы.

Сейчас мы охватили все основные темы последовательного Эрланга. То, что осталось, это несколько небольших странностей и дополнений, которые вы должны знать, но которые не подходят ни под одну из других тем. В них нет особенного логического порядка. Эти темы охватываются следующим образом:

apply: Как вычисляется значение функции по ее имени и аргументам, когда имена функции и модуля вычисляются динамически.

Атрибуты: Синтаксис и смысл атрибутов модуля на Эрланге.

Блоки выражений: Выражения, использующие begin и end.

Булевы выражения: Все булевы выражения.

Набор символов: Какой набор символов используется в Эрланге.

Комментарии: Синтаксис комментариев.

epp: Препроцессор Эрланга.

Escape-последовательности: Синтаксис управляющих последовательностей, используемых в строках и атомах.

Выражения и последовательности выражений: Что именно представляет из себя выражение?

Функциональные ссылки: Как ссылаться на функции.

Включаемые файлы: Как включать файлы во время компиляции.

Операции со списками: ++ и --.

Макросы: Макро-процессор Эрланга.

Оператор сопоставления в шаблонах: Как оператор сопоставления = может быть использован в шаблонах.

Числа: Синтаксис чисел.

Старшинство операторов: приоритет и ассоциативность всех операторов Эрланга.

Словарь процесса: Каждый процесс Эрланга имеет локальную область разрушаемой памяти, которая иногда может быть полезной.

Ссылки: Ссылки - это уникальные символы.

Упрощенные булевы выражения: Булевы выражения, которые не вычисляются полностью.

Сравнения термов: Все операторы сравнения термов и лексического упорядочения термов.

Подчеркнутые переменные: Переменные, которые компилятор рассматривает особым образом.

apply.

BIF apply(Mod, Func, [Arg1, Arg2, ..., ArgN]) применяет функцию Func из модуля Mod с аргументами Arg1, Arg2, ..., ArgN. Это эквивалентно вызову:

Mod:Func(Arg1, Arg2, ..., ArgN)

apply позволяет вам вызвать функцию в модуле, передавая ей аргументы. Что отличает ее от прямого вызова функции, это то, что имя модуля и/или функции может быть вычислено динамически.

Все BIFы Эрланга могут быть вызваны с использованием apply с предположением, что они принадлежат модулю erlang. Так, для создания динамического вызова BIF, мы можем написать следующее:

1> **apply(erlang, atom_to_list, [hello]).**

"hello"

Предупреждение: Если это возможно, следует избегать использования apply. Когда число аргументов функции известно заранее, намного лучше использовать вызов формы: M:F(Arg1, Arg2, ..., ArgN), чем apply. Когда вызовы функций строятся с использованием apply, многие инструменты анализа не могут понять, что происходит, а некоторые компиляторные оптимизации не могут быть выполнены. Итак, используйте apply редко и только когда это абсолютно необходимо.

Атрибуты.

Атрибуты модуля имеют синтаксис -AtomTag(...)^2\ ^и используются для определения некоторых свойств файла. Существует два типа атрибутов модуля: предопределенные и пользовательские.

Предопределенные атрибуты модуля.

Следующие атрибуты модуля имеют предопределенные значения и должны быть помещены перед любым определением функции.

-module(modname).

Декларация модуля. modname должен быть атомом. Этот атрибут должен быть первым атрибутом в файле. Общепринято, что код для modname должен быть сохранен в файле с именем modname.erl. Если вы не сделаете этого, автоматическая загрузка кода не будет работать корректно; подробнее смотри в разделе E.4, *Динамическая загрузка кода*.

-import(Mod, [Name1/Arity1, Name2/Arity2, ...]).

Определяет, что функция Name1 с числом аргументов Arity1 является импортированной из модуля Mod.

Если функция однажды импортирована из модуля, то вызов функции может быть выполнен без указания имени модуля.

Например:

-**module**(abc).

-**import**(lists, [map/2]).

f(L) ->

L1 = map(fun(X) -> 2*X end, L),

lists:sum(L1).

Вызов map не требует указания имени модуля, в то время как для вызова sum нам необходимо включить имя модуля в вызове функции.

-export([Name1/Arity1, Name2/Arity2, ...]).

Экспортирует функции Name1/Arity1, Name2/Arity2, и т.д. из текущего модуля. Заметьте, что только экспортированные функции могут быть вызваны снаружи модуля. Для примера:

HYPERLINK "http://media.pragprog.com/titles/jaerlang/code/abc.erl"abcHYPERLINK "http://media.pragprog.com/titles/jaerlang/code/abc.erl".HYPERLINK "http://media.pragprog.com/titles/jaerlang/code/abc.erl"erl

-**module**(abc).

-**export**([a/2, b/1]).

a(X, Y) -> c(X) + a(Y).

a(X) -> 2 * X.

b(X) -> X * X.

c(X) -> 3 * X.

Декларация экспорта указывает, что только a/2 и b/1 могут быть вызваны вне модуля abc. Так например, вызов abc:a(5) приведет к ошибке, т.к. a/1 не экспортирована из модуля.

1> **abc:a(1,2).**

7

2> **abc:b(12).**

144

3> **abc:a(5).**

** exited: {undef,[{abc,a,[5]},

{erl_eval,do_apply,5},

{shell,exprs,6},

{shell,eval_loop,3}]} ="session">

-compile(Options).

Добавляет Options к списку опций компилятора. Options - это одиночная опция компилятора или список опций компилятора (они описаны на странице руководства для модуля compile).

Примечание: Опция компилятора -compile(export_all). наиболее часто используется при отладке программ. Она экспортирует все функции из модуля без подробного использования аннотации -export.

-vsn(Version).

Определяет версию модуля. Version - любой символьный терм. Значение Version не имеет специального синтаксиса или смысла, но оно может быть использовано анализирующими программами или для целей документирования.

Пользовательские атрибуты.

Синтаксис пользовательских атрибутов модуля следующий:

-SomeTag(Value).

SomeTag должен быть атомом, а Value должна быть символьным термом. Значения атрибутов модуля компилируются в модуль и могут быть извлечены во время выполнения. Вот пример:

-**module**(attrs).

-vsn(1234).

```
-author({joe,armstrong}).

-purpose("example of attributes" ).

-export([fac/1]).

fac(1) -> 1;

fac(N) -> N * fac(N-1).

1> attrs:module_info().

[{exports,[{fac,1},{module_info,0},{module_info,1}]},

{imports,[]},

{attributes,[{vsn,[1234]},

{author,[{joe,armstrong}]},

{purpose,"example of attributes"}]},

{compile,[{options,[{cwd,"/home/joe/2006/book/JAERLANG/Book/code"},

{outdir,"/home/joe/2006/book/JAERLANG/Book/code"}]},

{version,"4.4.3"},

{time,{2007,2,21,19,23,48}},

{source,"/home/joe/2006/book/JAERLANG/Book/code/attrs.erl"}]}]

2> attrs:module_info(attributes).

[{vsn,[1234]},{author,[{joe,armstrong}]},{purpose,"example of attributes"}]

3> beam_lib:chunks("attrs.beam",[attributes]).

{ok,{attrs,[{attributes,[{author,[{joe,armstrong}]},

{purpose,"example of attributes"},

{vsn,[1234]}]}]}}
```

Пользовательские атрибуты, содержащиеся в файле исходного кода, повторяются как подтермы {attributes, ...}. Кортеж {compile, ...} содержит информацию, которая была добавлена компилятором. Значение {value,"4.4.3"} является версией компилятора и ее не следует путать с тегом vsn, определенным в атрибутах модуля. В предыдущем примере attrs:module_info() возвращает список свойств всех метаданных, ассоциированных с откомпилированным модулем, attrs:module_info(attributes)^3^ возвращает список всех атрибутов, ассоциированных с файлом.

Обратите внимание, что функции module_info/0 и module_info/1 автоматически создаются каждый раз при компиляции модуля.

Вывод в строках 2 и 3 немного тяжело читать. Чтобы сделать жизнь проще, вы можете написать небольшую функцию, которая извлекает конкретный атрибут и вызвать ее, как показано ниже:

4> **extract:attribute("attrs.beam", author).**

[{joe,armstrong}]

Код, выполняющий это прост:

HYPERLINK "http://media.pragprog.com/titles/jaerlang/code/extract.erl"extractHYPERLINK "http://media.pragprog.com/titles/jaerlang/code/extract.erl".HYPERLINK "http://media.pragprog.com/titles/jaerlang/code/extract.erl"erl

-**module**(extract).

-**export**([attribute/2]).

attribute(File, Key) ->

case beam_lib:chunks(File,[attributes]) **of**

{ok, {_Module, [{attributes,L}]}} ->

case lookup(Key, L) of

{ok, Val} ->

Val;

error ->

exit(badAttribute)

end;

_ ->

exit(badFile)

end.

lookup(Key, [{Key,Val}|_]) -> {ok, Val};

lookup(Key, [_|T]) -> lookup(Key, T);

lookup(_, []) -> error.

Чтобы запустить attrs:module_info, мы должны загрузить байт-код для модуля attrs. Модуль beam_lib содержит ряд функций для анализа модуля без загрузки кода. Например, в extract.erl используется beam_lib:chunks для извлечения данных атрибута без загрузки кода модуля.

Блок выражений.

---

begin

Expr1,

...,

ExprN

end

Вы можете использовать блок выражений для группировки последовательности выражений, аналогично телу условия. Значением блока begin ... end является значение последнего выражения в блоке.

Блок выражений используется когда синтаксис требует одиночное выражение, но вы хотите иметь последовательность выражений в этом месте кода.

Булевы значения.

В Эрланге не существует специального булева типа; вместо этого особую интерпретацию получают атомы true и false и они используются для представления булевых символов.

Булевы выражения.

Существует четыре булевых выражения:

not B1: Логическое НЕ;

B1 and B2: Логическое И;

B1 or B2: Логическое ИЛИ;

B1 xor B2: Логическое ИСКЛЮЧАЮЩЕЕ ИЛИ.

Во всех них B1 и B2 должны быть булевыми символами или выражениями, которые имеют булевы значения. Примеры:

1> **not true.**

false.

2> **true and false.**

false

3> **true or false.**

true

4> **(2 > 1) or (3 > 4).**

true

Усиление бинарных функций возвращением булевых значений

Иногда мы пишем функции, которые возвращают одно из двух возможных атомарных значений. Когда это происходит, правильной практикой будет убедиться, чтобы они возвращают булевы значения. Кроме того, хорошая идея именовать ваши функции так, чтобы можно было понять, что они возвращают логическое значение.

Например, предположим мы пишем программу, которая представляет состояние некоторого файла. Мы могли бы обнаружить себя пишущими функцию file_state(), которая возвращает open и closed. Когда мы пишем эту функцию, мы могли бы подумать о переименовании этой функции и позволить ей возвращать булево значение. Немного поразмыслив, мы перепишем нашу программу для использования функции is_file_open, которая возвращает true или false.

Почему мы должны сделать это?

Ответ прост. Есть большое количество функций в стандартных библиотеках, которые работают на функциях, возвращающих булевы значения. Поэтому, если мы убедимся, что все наши функции, которые могут возвращать одно из двух атомарных значений вместо этого возвращают булевы значения, то мы будем способны использовать их совместно со стандартными библиотечными функциями.

Набор символов.

Предполагается, что файлы с исходными текстами Эрланга кодируются в наборе символов ISO-8859-1 (Latin-1). Это означает, что все печатаемые символы Latin-1 могут быть использованы без использования любыми управляющими последовательностями.

Внутри Эрланг не имеет символьного типа данных. Строки реально не существуют, вместо этого они представляются списками целых чисел. Строки Юникод (Unicode) могут быть представлены списками целых чисел без-либо проблем, хотя существует ограниченная поддержка разбора и создания файлов в кодировке Юникод из целочисленных списков Эрланга.

Комментарии.

Комментарии в Эрланге начинаются с символа процента (%) и продолжаются до конца строки. Блочных комментариев нет.

Примечание: Вы часто будете видеть двойные символы процента (%%) в примерах кода. Двойные знаки процента распознаются в erlang-mode Емакса и разрешают автоматический отступ закомментированных строк.

```
% This is a comment

my_function(Arg1, Arg2) ->

case f(Arg1) of
```

{yes, X} -> % it worked

..

epp.

Перед тем, как модуль Эрланга будет откомпилирован, он автоматически обрабатывается препроцессором Эрланга epp. Препроцессор раскрывает все макросы, которые могут быть в исходном файле и вставляет все необходимые включаемые файлы.

Обычно вам не нужно наблюдать вывод препроцессора, но в исключительных обстоятельствах (например, при отладке неисправного макроса) вы можете захотеть сохранить вывод препроцессора. Вывод препроцессора может быть сохранен в файл с помощью команды compile:file(M, ['P']). Этим компилируется любой код в файле M.erl и выполняется листинг в файл M.P, где все макросы были раскрыты и все необходимые включаемые файлы были вставлены.

Управляющие последовательности.

В строках и атомах с кавычками вы можете использовать управляющие последовательности для ввода каких-либо непечатаемых символов. Все возможные управляющие последовательности показаны в таблице 5.1.

Приведем несколько примеров в оболочке, чтобы показать, как эти конвенции работают. (Примечание: \~w в строке формата печатает список без каких-либо попыток напечатать достаточный результат.)

%% Control characters

1> **io:format("\~w\~n", ["\b\d\e\f\n\r\s\t\v"]).**

[8,127,27,12,10,13,32,9,11]

ok

%% Octal characters in a string

3> **io:format("\~w\~n", ["\123\12\1"]).**

[83,10,1]

ok

%% Quotes and escapes in a string

4> **io:format("\~w\~n", ["\'\"\\"]).**

[39,34,92]

ok

%% Character codes

5> **io:format("\~w\~n", ["\a\z\A\Z"]).**

[97,122,65,90]

ok

Управляющая последовательность

Значение

Числовой код

\b

Возврат на одну позицию (Backspace)

8

\d

Удалить

127

\e

Выход (Escape)

27

\f

Прогон страницы (Form feed)

12

\n

Перевод строки

10

\r

Возврат каретки

13

\s

Пробел

32

\t

Табуляция (Tab)

9

\v

Вертикальная табуляция

11

\NNN \ NN \ N

Восьмиричные символы (N есть 0..7)

\\^a..\\^z или \\^A..\\^Z

От Ctrl+A до Ctrl+Z

1..26

\'

Одиночный апостроф

39

\"

Двойной апостроф

34

\\

Обратный слэш

92

\C

Код ASCII для C (C - символ)

Целое число

Таблица 5.1. Управляющие последовательности.

Выражения и последовательности выражений.

В Эрланге все, что может быть вычислено для производства значения называется *выражение*. Это означает, что такие вещи, как catch, if и try...catch являются выражениями. Такие вещи, как записи или атрибуты модулей не могут быть вычислены, поэтому они не являются выражениями.

Последовательности выражений - это ряды выражений, разделенных запятыми. Они находятся

повсюду сразу после стрелки ->. Значение последовательности выражений E1, E2, ..., En определяется значением последнего выражения в последовательности^4^. Это значение вычисляется с использованием любых привязок, созданных при вычислении значений E1, E2 и т.д.

Функциональные ссылки.

Часто мы хотим сослаться на функцию, которая определена в текущем модуле или в каком-либо внешнем модуле. Вы можете использовать следующее обозначение для этого:

fun LocalFunc/Arity

Используется для ссылки на локальную функцию с именем LocalFunc и числом аргументов Arity в текущем модуле.

fun Mod:RemoteFunc/Arity

Используется для ссылки на внешнюю функцию с именем RemoteFunc и числом аргументов Arity в модуле Mod.

Пример функциональной ссылки в текущем модуле:

-**module**(x1).

-**export**([square/1, ...]).

square(X) -> X * X.

...

double(L) -> lists:map(fun square/1, L).

Если мы хотим вызвать функцию в удаленном модуле, мы можем сослаться на функцию, как в следующем примере:

-**module**(x2).

...

double(L) -> lists:map(fun x1:square/1, L).

fun x1:square/1 означает функцию square/1 в модуле x1.

Включаемые файлы.

Файлы могут быть включены с использованием следующего синтаксиса:

-include(Filename).

В Эрланге есть договоренность, что включаемые файлы имеют расширение .hrl. FileName должно содержать абсолютный или относительный путь, по которому препроцессор может

обнаружить соответствующий файл. Файлы библиотечных хедеров могут быть включены с использованием следующего синтаксиса:

-include_lib(Name).

Например:

-include_lib(*"kernel/include/file.hrl"*).

В этом случае компилятор Эрланга будет искать соответствующие включаемые файлы. (kernel в предыдущем примере ссылается на приложение, в котором определен этот заголовочный файл.)

Включаемые файлы обычно содержат определения записей. Если нескольким модулям необходимо разделять общие определения записей, то эти общие определения записей помещаются во включаемые файлы, которые включаются всеми модулями, которым необходимы данные определения.

Операции со списками ++ и --.

++ и -- являются инфиксными операторами для сложения и вычитания списков.

A ++ B складывает (т.е. присоединяет) A и B.

A -- B вычитает список B из списка A. Вычитание означает, что каждый элемент B удаляется из A. Обратите внимание, что если некоторый символ X входит K раз в B, то только первые K вхождений X в A будут удалены.

Примеры:

1> **[1,2,3] ++ [4,5,6].**

[1,2,3,4,5,6]

2> **[a,b,c,1,d,e,1,x,y,1] -- [1].**

[a,b,c,d,e,1,x,y,1]

3> **[a,b,c,1,d,e,1,x,y,1] -- [1,1].**

[a,b,c,d,e,x,y,1]

4> **[a,b,c,1,d,e,1,x,y,1] -- [1,1,1].**

[a,b,c,d,e,x,y]

5> **[a,b,c,1,d,e,1,x,y,1] -- [1,1,1,1].**

[a,b,c,d,e,x,y]

++ в шаблонах.

++ может быть использован в шаблонах. Когда сопоставляются строки, мы можем писать такие шаблоны, как в следующем примере:

f("begin" ++ T) -> ...

f("end" ++ T) -> ...

...

Шаблон в первом случае расширяется в [$b,$e,$g, $i,$nlT].

Макросы.

Макросы в Эрланге записываются, как показано здесь:

-**define**(Constant, Replacement).

-**define**(Func(Var1, Var2,.., Var), Replacement).

Макросы раскрываются препроцессором Эрланга epp, когда встречается выражение формы ? MacroName. Переменные, встречающиеся в определении макроса, полностью совпадают по форме в соответствующем месте вызова макроса.

-**define**(macro1(X, Y), {a, X, Y}).

foo(A) ->

?macro1(A+10, b)

Этот пример раскладывается в:

foo(A) ->

{a,A+10,b}.

в дополнение, существует несколько предопределенных макросов, обеспечивающих информацию о текущем модуле. Это такие макросы, как:

?FILE раскладывается в текущее имя файла.

?MODULE раскладывается в текущее имя модуля.

?LINE раскладывается в текущий номер строки.

Управление потоком в Макросах.

Внутри определения макроса поддерживаются следующие директивы. Вы можете использовать их для направления потока управления внутри макроса:

-undef(Macro).

Разопределяет макрос; после этого вы не можете вызывать макрос.

-ifdef(Macro).

Вычисляет следующие строки только если Macro был определен.

-ifndef(Macro).

Вычисляет следующие строки только если Macro не определен.

-else.

Допустимо после утверждений ifdef или ifndef. Если условие было false, то утверждения, следующие за **else** будут вычислены.

-endif.

Отмечает конец утверждения ifdef или ifndef.

Условные макросы должны быть правильно вложены. Условно они группируются следующим образом:

```
-ifdef(debug).

-define(...).

-else.

-define(...).

-endif.
```

Мы можем использовать эти макросы для определения макроса TRACE. Например:

```
-module(m1).

-export([start/0]).

-ifdef(debug).

-define(TRACE(X), io:format("TRACE \~p:\~p \~p\~n" ,[?MODULE, ?LINE, X])).

-else.

-define(TRACE(X), void).

-endif.

start() -> loop(5).

loop(0) ->

void;

loop(N) ->
```

?TRACE(N),

loop(N-1).

Примечание: io:format(String, [Args]) печатает переменные в [Args] в оболочке Эрланга в соответствии с форматирующей информацией в String. Форматирующие коды представлены символом (\~). \~p является аббревиатурой для *pretty print* (достаточно для печати), а \~n создает новую строку.^5^

^^

Для компиляции с использованием включенного и выключенного макроса трассировки мы можем использовать дополнительные аргументы в c/2 следующим образом:

1> **c(m1, {d, debug}).**

{ok,m1}

2> **m1:start().**

TRACE m1:15 5

TRACE m1:15 4

TRACE m1:15 3

TRACE m1:15 2

TRACE m1:15 1

void

c(m1, Options) обеспечивает способ помещения опций в компилятор. {d, debug} устанавливает флаг отладки в true, что позволяет распознать его в секции ifdef(debug) определения макроса.

Когда макрос выключен, макрос трассировки просто раскладывается в атом void. Такой выбор названия не имеет никакого значения; это просто напоминание для меня, что значение макроса нигде не используется.

Оператор сопоставления в шаблонах.

Предположим, что мы имеем некоторый код, такой как:

func1([{tag1, A, B}|T]) ->

...

... f(..., {tag1, A, B}, ...)

...

В строке 1 мы шаблонно сопоставляем терм {tag1, A, B}, а в строке 3 мы вызываем f с

аргументом, которым является {tag1, A, B}. Когда мы делаем это, система пересобирает терм {tag1, A, B}. Гораздо более эффективным и менее подверженным ошибкам способом сделать это является присвоение шаблона временной переменной Z и помещение ее в f, например:

func1([{tag1, A, B}=Z|T]) ->

...

... f(... Z, ...)

...

Оператор сопоставления может быть использован в любой точке шаблона, так если мы имеем два терма, которые требуется пересобрать, как в следующем коде:

func1([{tag, {one, A}, B}|T]) ->

...

... f(..., {tag, {one,A}, B}, ...),

... g(..., {one, A}), ...)

...

То мы можем представить две новые переменные Z1 и Z2 и написать следующее:

func1([{tag, {one, A}=Z1, B}=Z2|T]) ->

..,.

... f(..., Z2, ...),

... g(..., Z1, ...),

...

Числа.

Числа в Эрланге являются либо целыми, либо действительными.

Целые числа.

Целочисленная арифметика является точной, а количество цифр, которые могут быть представлены в целом числе ограничивается только доступным объемом памяти.

Целые числа записываются одним из трех различных синтаксисов:

Обычный синтаксис: Здесь целые числа записаны так, как вы ожидаете. Например, 12, 12375 и -23427 являются целыми числами.

Целое по основанию K. Целые числа по основанию, отличному от десяти записываются с

помощью синтаксиса K#Digits; то есть мы можем написать двоичное число как 2#00101010 или шестнадцатиричное число как 16#af6bfa23. Для оснований больше, чем десять символы abc... (или ABC...) представляют числа 10, 11, 12 и так далее. Наибольшим основанием, которое мы можем представить этим способом есть основание 36.

$ синтаксис: Синтаксис $C представляет целочисленный код для ASCII-символа C. То есть $a является сокращением для 97, $1 - для 49 и так далее.

Непосредственно после $ мы можем также использовать любую управляющую последовательность, описанную на рис. 5.1. Таким образом, $\n это 10, $\\^c это 3 и так далее.

Несколько примеров целых чисел:

0 -65 2#010001110 -8#377 16#FE34 36#wow

(Их значениями являются: 0, -65, 142, -255, 65076, 65076 и 42368 соответственно.)

Действительные числа.

Числа с плавающей точкой имеют пять частей: необязательный знак, целая часть числа, десятичная точка, дробная часть и необязательная часть экспоненты.

Несколько примеров действительных чисел:

1.0 3.14159 -2.3e+6 23.56E-27

После разбора, числа с плавающей точкой внутренне представляются в 64-битном формате IEEE 754. Реальные числа в диапазоне от -10323 до 10308 могут быть представлены действительными числами Эрланга.

Старшинство операторов.

Таблица 5.2 показывает все операторы Эрланга в порядке убывания приоритета, вместе с их ассоциативностью. Старшинство операторов и ассоциативность используется для определения порядка вычисления в бесскобочных выражениях.

Выражения с высоким приоритетом (выше в таблице) вычисляются первыми, а затем вычисляются выражения с более низким приоритетом. Таким образом, например, для вычисления 3+4*5+6 мы сначала вычислим подвыражение 4*5, так как (*) выше в таблице, чем (+). Затем мы вычисляем 3+20+6. Так как (+) является лево-ассоциативным оператором, мы рассматриваем его как (3+20)+6, поэтому мы вначале вычисляем 3+20, получив 23; в заключение мы вычисляем 23+6.

В полной скобочной форме 3+4*5+6 представляется как ((3+(4*5))+6). Как и во всех остальных языках программирования, лучше использовать скобки для обозначения области действия, чем полагаться на правила приоритета.

Операторы

Ассоциативность

:

#

(унарный) +, (унарный) -, bnot, not

/, *, div, rem, band, and

Лево-ассоциативный

+, -, bor, bxor, bsl, bsr, or, xor

Лево-ассоциативный

++, --

Право-ассоциативный

==, /=, =<, <, >=, >, =:=, =/=

andalso

orelse

Таблица 5.2. Старшинство операторов.

Словарь процесса.

Каждый процесс в Эрланге имеет свое собственное персональное хранилище данных, называемое словарем процесса. Словарь процесса является ассоциативным массивом (в других языках это может называться картой, хэш-картой или хэш-таблицей), состоящим из набора ключей и значений. Каждый ключ имеет только одно значение.

Словарем можно манипулировать с использованием следующих BIFов:

@spec put(Key, Value) -> OldValue.

Добавить ассоциацию Ключ-Значение в словарь процесса. Значением put является OldValue, которое было предыдущим значением. ассоциированным с Key. Если предыдущего значения не было, возвращается атом undefined.

@spec get(Key) -> Value.

Просмотр значения Key. Если есть соответствующая ассоциация Key-Value в словаре, то возвращается Value; иначе возвращается атом undefined.

@spec get() -> [{Key,Value}].

Возвращение словаря полностью, как списка кортежей {Key, Value}.

@spec get_keys(Value) -> [Key].

Возвращает список ключей, который имеют значение Value в словаре.

@spec erase(Key) -> Value.

Возвращает значение, ассоциированное с Key, или атом undefined если нет ассоциированной величины. В завершение значение, ассоциированное с Key стирается.

@spec erase() -> [{Key,Value}].

Стирает словарь процесса полностью. Возвращаемым значением является список кортежей {Key, Value}, представляющий состояние словаря перед тем, как он был очищен.

Например:

1> **erase().**

[]

2> **put(x, 20).**

undefined

3> **get(x).**

20

4> **get(y).**

undefined

5> **put(y, 40).**

undefined

6> **get(y).**

40

7> **get().**

[{y,40},{x,20}]

8> **erase(x).**

20

9> **get().**

[{y,40}]

Как вы можете видеть, переменные в словаре процесса ведут себя во многом как обычные переменные в императивных языках программирования. Если вы используете словарь процесса, ваш код более не является свободным от побочных эффектов, а все преимущества использования неразрушаемых переменных, которые мы обсуждали в разделе 2.6, *Переменные,*

которые не меняются, не применяются. По этой причине вы должны использовать словарь процесса очень осторожно.

Примечание: Я редко использую словарь процесса. Использование словаря процесса может внести тонкие ошибки в вашу программу и сделать ее сложной для отладки. Одной из форм использования, которую я одобряю, является использование словаря процесса для хранения "однократно записанных" переменных. Если ключ принимает значение ровно один раз и не меняет значение, то сохранение его в словаре процесса иногда допустимо.

Ссылки.

Ссылки являются глобально уникальными термами Эрланга. Они создаются с помощью BIF erlang:make_ref(). Ссылки удобны для создания уникальных тэгов, которые могут быть включены в данные, а затем на более позднем этапе сравнены на равенство. Например, система баг-трекинга может добавлять ссылку на каждый новый отчет об ошибке для того, чтобы получить его уникальный идентификатор.

Упрощенные булевы выражения.

Упрощенные булевы выражения - это булевы выражения, аргументы которых вычисляются только если это необходимо.

Существует два упрощенных булевых выражения:

Expr1 orelse Expr2

Здесь сначала вычисляется Expr1. Если Expr1 вычисляется как истина, то Expr2 не вычисляется. Если Expr1 вычислена как ложь, Expr2 вычисляется.

Expr1 andalso Expr2

Здесь сначала вычисляется Expr1. Если Expr1 истинно, то Expr2 вычисляется. Если Expr1 ложно, Expr2 не вычисляется.

Примечание: В соответствующих булевых выражениях (A or B; A and B) оба аргумента вычисляются всегда, даже если правильное значение выражения может быть определено вычислением только первого выражения.

Сравнение термов.

Существует восемь возможных операций сравнения термов, показанных в таблице 5.3.

Для целей сравнения определен общий порядок по всем термам. Определено так, что следующее выражение истинно:

number < atom < reference < fun < port < pid < tuple < list < binary

Что это означает? Это означает, что, например, число (любое число) по определению меньше атома (любого атома), что кортеж больше, чем атом и так далее. (Заметьте, что в целях

сравнения, порты и PIDы включены в этот список. Мы поговорим об этом позже.)

Имеющийся общий порядок по всем термам означает, что мы можем отсортировать список любого типа и создать эффективные процедуры доступа на основе порядка сортировки ключей.

Все операторы сравнения термов, за исключением =:= и =/= ведут себя следующим образом, если их аргументы числа:

Если один аргумент целое, а другой - действительное число, то целое конвертируется в действительное перед выполнением сравнения.

Если оба аргумента целые, или оба аргумента действительные числа, то аргументы используются "как есть", т.е. без преобразования.

Вы должны также быть очень внимательными при использовании == (особенно если вы C или Java программист). В 99 из 100 случаев вы должны использовать =:=. == полезен только для сравнения целых и действительных чисел. =:= проверяет, являются ли два терма идентичными^6^. Если вы сомневаетесь, используйте =:=, и будьте подозрительными, если увидите ==. Заметьте, что похожие комментарии применимы к использованию /= и =/=, где /= означает "не равно", а =/= означает "не идентично".

Примечание: В большинстве библиотечного и опубликованного кода вы увидите ==, использованный когда должен быть оператор =:=. К счастью, такая ошибка не часто приводит в результате к неверной программе, поскольку если аргументы == не содержат действительных чисел, то поведение этих двух операторов то же самое.

Оператор

Значение

X > Y

X больше, чем Y

X < Y

X меньше, чем Y

X =< Y

X равен, либо больше, чем Y

X >= Y

X равен, либо меньше, чем Y

X == Y

X равен Y

X /= Y

X не равен Y

X =:= Y

X идентичен Y

X =/= Y

X не идентичен Y

Таблица 5.3. Сравнение термов.

Вы должны также знать, что сопоставление условия функции всегда подразумевает точное соответствие шаблона, так что если вы определите функцию F = fun(12) -> ... end, то попытка вычислить F(12) не удастся.

Подчеркнутые переменные.

Есть еще одна вещь, которую надо сказать о переменных. Специальный синтаксис _VarName используется для обычной, а не для анонимной переменной. Обычно компилятор генерирует предупреждение, если переменная используется только однажды в условии, так как обычно это является признаком ошибки. Если переменная используется только один раз, но начинается с подчеркивания, предупреждающее сообщение сгенерировано не будет.

Так как *Var является нормальной переменной, могут произойти очень трудноуловимые ошибки, вызванные забыванием этого и использовании ее как шаблона "не беспокоиться". В сложном шаблонном сопоставлении может быть трудно заметить, например, что* Int повторяется, хотя не должна, приводя к неудаче сопоставления шаблона.

Есть два основных порядка использования подчеркнутых переменных:

Именовать переменные, которые мы не намерены использовать. То есть написание open(File, *Mode) делает программу более читабельной, чем написание open(File,*).

В целях отладки. Например, предположим мы пишем следующее:

some_func(X) ->

{P, Q} = some_other_func(X),

io:format("Q = \~p\~n" , [Q]),

a.

Этот код компилируется без сообщения об ошибке.

Теперь закомментируем утверждение format:

some_func(X) ->

{P, Q} = some_other_func(X),

%% io:format("Q = \~p\~n", [Q]),

a.

Если мы это откомпилируем, компилятор выдаст предупреждение, что переменная Q не используется.

Если мы перепишем функцию следующим образом:

some_func(X) ->

{P, _Q} = some_other_func(X),

io:format("*Q = \~p\~n", [*Q]),

a.

то мы можем комментировать утверждение format и компилятор не будет жаловаться.

Теперь мы фактически прошли через последовательный Эрланг. Мы не упомянули о некоторых небольших разделах, но мы вернемся к ним, когда столкнемся с ними в прикладных разделах.

В следующей главе мы рассмотрим, как компилировать и запускать ваши программы различными способами.

\1. 2#11111111111 - это целое число основания 2.

\2. -**record**(...) и -**include**(...) имеют похожий синтаксис, но не описывают атрибуты модуля.

\3. Другими аргументами являются exports, imports и compile.

\4. Это эквивалентно progn в языке LISP.

\5. io:format понимает очень большое число опций форматирования. Для дополнительной информации смотри раздел 13.3, *Запись списка термов в файл*.

\6. Идентичность означает одинаковое значение (как EQUAL в Common Lisp). Поскольку значения являются неизменными, это не означает какого-либо понятия идентичности указателя.