Erlang: Магия

Андрей Ушаков завершает «большое колдовство» с битовыми строками, раскодируя их.

Наш эксперт

Андрей Ушаков активно приближает тот день, когда функциональные языки станут мейнстримом.

В этом номере мы продолжим обсуждение большого примера использования битовых строк: рассмотрим, каким образом объекты Erlang, согласно правилам **ASN.1 BER** (то, чем мы занимались в прошлом номере), раскодировать обратно.

Первый шаг, как и во всех других примерах – определение модуля с функциональностью примера, подключение файлов с определениями и определение списка экспортируемых функций. Экспортируем мы всего две функции: **build/1** для построения функции диспетчера, которая выбирает должную функцию для декодирования данных по их типу, и **decode/2**, для самого декодирования.

```
-module(asn1_decoder).
-include("asn1_tag.hrl").
-export([build/1, decode/2]).
```

Функция-диспетчер использует список пар (кортежей из двух значений) из типа данных и соответствующей ему функции для декодирования объекта этого типа. При создании функции-диспетчера мы используем как внутренний (заданный нами) список пар из типа данных и соответствующей ему функции для декодирования, так и внешний (задаваемый пользователем функции через единственный параметр) список таких пар.

Внутренний список пар обрабатывает общие ситуации декодирования (когда объект является списком, кортежем, целым числом и т.д.); внешний список (задаваемый пользователем) служит для обработки специфичных ситуаций декодирования – например, если мы хотим декодировать записи отличным от обычных кортежей образом. Для этого пары из внешнего списка идут всегда перед парами из внутреннего списка, что означает более высокий их приоритет. Более того, если одна пара идет перед другой, то это означает, что приоритет у этой пары выше, т. к. эта пара будет использована в процедуре выбора подходящей функции декодирования первой.

Вполне очевидно, что внешний список пар, используемый для построения функции-диспетчера для декодирования объектов по их типу, должен соответствовать списку пар функций, используемый для построения соответствующей функции диспетчера для кодирования объектов (об этом мы говорили в предыдущем номере). Соответствие этих списков означает следующее: функции, кодирующие и декодирующие одни и те же типы данных должны располагаться одинаково в соответствующих списках. Если не соблюдать соответствие списков, то возможны ситуации, когда невозможно декодировать закодированные объекты, либо когда мы кодируем объект одного типа, а при декодировании получаем объект другого типа, отличного от исходного.

```
build(ExternalDecoders) when is_list(ExternalDecoders) ->
  InternalDecoders =
  [
    {#tag{class = universal, form = primitive, tag_value = 1}, fun decode_boolean/2},
    {#tag{class = universal, form = primitive, tag_value = 2}, fun decode_integer/2},
    {#tag{class = universal, form = primitive, tag_value = 9}, fun decode_real/2},
    {#tag{class = universal, form = primitive, tag_value = 4}, fun decode_octetstring/2},
    {#tag{class = universal, form = primitive, tag_value = 3}, fun decode_bitstring/2},
    {#tag{class = universal, form = constructed, tag_value = 16}, fun decode_sequence/2},
    {#tag{class = universal, form = constructed, tag_value = 32}, fun decode_tuple/2},
    {#tag{class = universal, form = primitive, tag_value = 33}, fun decode_atom/2}
  ],
  TotalDecodersList = ExternalDecoders ++ InternalDecoders,
  fun(Binary, DecodeDispatcher) ->
    {Tag, _BinaryRest} = decode_tag(Binary),
    case lists:keyfind(Tag, 1, TotalDecodersList) of
      {Tag, Decoder} -> {ok, Decoder(Binary, DecodeDispatcher)};
      false -> false
    end
end.
```

Функция **decode/2** использует функцию-диспетчер декодирования, которую мы построили при помощи функции **build/1**; эту функцию-диспетчер мы передаем в качестве первого параметра (в качестве второго параметра передается битовая строка, содержащая закодированный объект). Работа функции **decode/2** основана на использовании функции-диспетчера декодирования: мы вызываем функцию-диспетчер, передавая в качестве параметров декодируемый объект и саму функцию-диспетчер декодирования. Если передаваемые данные могут быть декодированы в объект, то будет возвращен кортеж, состоящий из атома **ok** и декодированного объекта; если же передаваемые данные не могут быть декодированы, то будет возвращен атом **false**. В последнем случае, мы генерируем исключение времени выполнения.

Следует также сказать, что в отличие от функции кодирования объектов **encode/2** (которую мы приводили в предыдущем номере), для работы функции **decode/2** не нужна функция для поиска первой подходящей пары, наподобие функции **first/3** (опять же приведенной в предыдущем номере). Это связано с тем, что в функции **decode/2** поиск пары происходит по ключу (которым является тип данных), и для этой операции достаточно функции **lists:keyfind/3**. В отличие от функции **decode/2**, в функции **encode/2** поиск первой подходящей пары производился в списке пар функций, и критерием окончания поиска было нахождение пары, первая функция которой возвращала атом **true** для исходного объекта. Такая функциональность не реализована ни среди функций модуля **lists** (модуля работы со списками), ни среди функций любых других стандартных модулей.

```
decode(Value, DecodeDispatcher) ->
  case DecodeDispatcher(Value, DecodeDispatcher) of
    {ok, Result} -> Result;
    false -> erlang:error(unsuitable_value)
  end.
```

Теперь перейдем непосредственно к декодированию данных. В первую очередь нам необходимо уметь декодировать тип данных (он же тэг данных). Связано это с тем, что по типу данных мы выбираем потом подходящую функцию для декодирования самого объекта. Как уже говорилось, тип данных состоит из трех компонент:

битовых строк 3

класса, формы и идентификатора типа данных. Класс и форма имеют фиксированный размер – 2 и 1 бит соответственно. Способ кодирования идентификатора типа данных зависит от его значения. Если значение идентификатора меньше 31, то он занимает оставшиеся 5 бит октета (байта) полностью. Если же его значение больше или равно 31, то в оставшиеся 5 бит октета записывается значение **2#11111**, после чего идет значение идентификатора, закодированное более сложным способом, о котором мы поговорим ниже. За декодирование типа данных отвечает метод **decode_tag/1**. Входящий параметр у него один – битовая строка с данными для декодирования. Входящий параметр в заголовке метода при помощи операции соответствия шаблону [pattern matching] разбивается на 2 бита для класса, 1 бит для формы, 5 битовый сегмент (либо для идентификатора типа, либо для величины **2#11111**) и оставшуюся часть битовой строки. Значение 5 битового сегмента определяет, можно ли сразу декодировать идентификатор типа данных или же необходимо извлечь из оставшейся битовой строки еще данные. Поэтому вполне логично, что на основе этого значения мы определяем два варианта метода **decode_tag/1**.

```
decode_tag(<<Class:2, Form:1, 2#11111:5, Rest/binary>>) ->
  {TagValue, TagRest} = decode_tag_value(Rest, []),
  {#tag{class = decode_tag_class(Class), form = decode_tag_
form(Form), tag_value = TagValue}, TagRest};
decode_tag(<<Class:2, Form:1, TagValue:5, Rest/binary>>) ->
  {#tag{class = decode_tag_class(Class), form = decode_tag_
form(Form), tag_value = TagValue}, Rest}.
```

Метод **decode_tag_class/1** служит для декодирования значения класса типа данных в соответствующий предопределенный атом. Для декодирования мы используем несколько вариантов функции **decode_tag_class/1**, которые покрывают весь диапазон возможных значений для класса типа данных.

```
decode_tag_class(2#00) -> universal;
decode_tag_class(2#01) -> application;
decode_tag_class(2#10) -> context_specific;
decode_tag_class(2#11) -> private.
```

Метод **decode_tag_form/1** служит для декодирования значения формы типа данных; принципы его работы полностью аналогичны предыдущему методу.

```
decode_tag_form(0) -> primitive;
decode_tag_form(1) -> constructed.
```

Извлечение идентификатора типа данных, который имеет произвольную длину – более сложная задача (простой случай, когда значение идентификатора типа данных содержится в сегменте размером 5 бит, мы рассматриваем отдельно). Как мы помним, при кодировании идентификатора мы преобразуем его значение в битовую строку, состоящую из целого числа сегментов размером 7 бит. После этого каждый 7-битный сегмент преобразуем в октет (8-битный сегмент), добавляя в качестве старшего бита 1 для всех сегментов, кроме последнего, и 0 для последнего сегмента. Поэтому при декодировании мы будем поступать следующим образом: брать очередной октет (8-битный сегмент), извлекать младшие 7 бит и добавлять к списку 7-битных сегментов. Делать эту операцию мы будем до тех пор, пока нам не встретится октет, старший бит у которого равен 0. После этого мы меняем порядок сегментов в списке на обратный (т.к. по соображениям производительности мы добавляем сегменты в конец списка), преобразуем список сегментов в битовую строку и извлекаем из полученной битовой строки целое число.

```
decode_tag_value(<<0:1, Segment:7, Rest/binary>>, SegmentList) ->
  TagValueBitstring = list_to_bitstring(lists:reverse([<<Segment:
7>>] ++ SegmentList)),
  BitSize = bit_size(TagValueBitstring),
  <<TagValue:BitSize/integer-big>>  = TagValueBitstring,
  {TagValue, Rest};
decode_tag_value(<<1:1, Segment:7, Rest/binary>>, SegmentList) ->
  decode_tag_value(Rest, [<<Segment:7>>] ++ SegmentList).
```

Следующий, не менее необходимый шаг – это декодирование длины (количества октетов), которую занимают данные. Давайте вспомним, как мы кодируем длину, занимаемую данными. Если значение длины меньше 128, то для хранения достаточно одного октета (следует отметить, что при этом у октета со значением длины старший бит будет равен 0). Если значение длины больше или равно 128, то первым идет октет, у которого старший бит установлен в 1, а остальные биты содержат количество октетов для хранения длины, после чего идет сама длина (занимающая целое число октетов). Функциональность по декодированию длины реализует функция **decode_length/1**; вполне логично, что она содержит два варианта, покрывающих два возможных случая хранения длины в закодированном виде.

```
decode_length(<<0:1, Length:7, Rest/binary>>) -> {Length, Rest};
decode_length(<<1:1, LengthOctetCount:7, Rest/binary>>) ->
  LengthBitCount = 8 * LengthOctetCount,
  <<Length:LengthBitCount, ParseRest/binary>> = Rest,
  {Length, ParseRest}.
```

Теперь перейдем непосредственно к декодированию объектов Erlang. Начнем с декодирования булевских значений (булевские значения в языке Erlang являются не особым типом, а двумя предопределенными атомами **true** и **false**). Тип данных для булевских значений равен 1 (класс – **universal**, форма – **primitive**, идентификатор – 1), для хранения собственно значения достаточно 1 октета. »

Стандарты ASN.1

- ITU-T Rec. X.680 | ISO/IEC 8824-1. Спецификация на базовую нотацию.
- ITU-T Rec. X.681 | ISO/IEC 8824-2. Спецификация на информационные объекты.
- ITU-T Rec. X.682 | ISO/IEC 8824-3. Спецификация на ограничения.
- ITU-T Rec. X.683 | ISO/IEC 8824-4. Спецификация на параметризацию ASN.1.
- ITU-T Rec. X.690 | ISO/IEC 8825-1. Спецификация на BER (Basic encoding rules), CER (Canonical encoding rules) и DER (Distinguished encoding rules).
- ITU-T Rec. X.691 | ISO/IEC 8825-2. Спецификация на PER (Packed encoding rules).
- ITU-T Rec. X.692 | ISO/IEC 8825-3. Спецификация на ECN (Encoding control notation).
- ITU-T Rec. X.693 | ISO/IEC 8825-4. Спецификация на XER (XML Encoding rules).
- ITU-T Rec. X.694 | ISO/IEC 8825-5. Спецификация на отображение на XSD.
- ITU-T Rec. X.695 | ISO/IEC 8825-6. Спецификация на регистрацию и применение инструкций кодирования PER (Packed encoding rules).

Стандартные типы данных ASN.1

Стандартные типы данных ASN.1 – это типы данных, описанные в стандартах ASN.1. У этих типов данных класс **universal**; работать с данными стандартных типов должна уметь любая ASN.1-совместимая реализация протокола. К стандартным типам данных относятся следующие:

- **Логические значения (BOOLEAN)** Принимают два значения – TRUE и FALSE.
- **Значение NULL** Собственно говоря, это не тип, а специальное значение, обрабатываемое и кодируемое специальным, отличным от любых других значений любых типов способом.
- **Целые числа (INTEGER)** Содержит целые числа произвольного размера.
- **Перечисления (ENUMERATED)** Подмножество целых чисел; с точки зрения правил кодирования ASN.1 BER, значения этого типа кодируются совершенно так же, как и целые числа.
- **Действительные числа (REAL)** Содержит действительные числа произвольного размера. С точки зрения правил кодирования ASN.1 BER действительные числа по основанию 10 и по основанию 2 кодируются по-разному. Правила кодирования ASN.1 DER определяют один формат кодирования для действительных чисел – как по основанию 10, так и по основанию 2.
- **Битовые строки (BIT STRING)** Содержат последовательность бит, количество которых не кратно 8.
- **Строки октетов (OCTET STRING)** Содержат последовательность октетов (или, что, то же самое, последовательность бит, количество которых кратно 8).
- **Идентификатор объектов (OBJECT IDENTIFIER и RELATIVE-OID)**.
- **Последовательность объектов (SEQUENCE и SEQUENCE OF)** Тип данных для хранения объектов других типов в определенной последовательности. Разница между SEQUENCE и SEQUENCE OF в том, что в последовательностях первого типа допускается хранение данных разных типов, тогда как в последовательностях второго типа допускается хранение данных одного типа. С точки зрения правил кодирования ASN.1 BER, SEQUENCE и SEQUENCE OF одинаковы.
- **Множество объектов (SET и SET OF)** Тип данных для хранения объектов других типов, при этом порядок хранения не определен. Разница между SET и SET OF в том, что во множествах первого типа допускается хранение данных разных типов, тогда как во множествах второго типа допускается хранение данных одного типа. С точки зрения правил кодирования ASN.1 BER SET и SET OF одинаковы.

Если закодированное значение равно 0, то соответствующее булевское значение равно **false**; если нет, то **true**.

```
decode_boolean(<<1:8, 1:8, Value:8, Rest/binary>>, _
DecodeDispatcher) ->
  if
    Value == 0 -> {false, Rest};
    Value /= 0 -> {true, Rest}
  end.
```

Еще один тип данных, которые мы хотим научиться декодировать – целые числа. Тип данных для булевских значений равен 2 (класс – **universal**, форма – **primitive**, идентификатор – 2); длина, необходимая для хранения закодированных данных (в отличие от предыдущего случая), может быть любой. Поэтому, прежде чем декодировать данные, мы должны получить длину (при помощи функции **decode_length/1**), после чего сможем декодировать целое число при помощи операции соответствия шаблону (для этого нам нужно знать количество бит, которые занимает целое число в битовой строке).

```
decode_integer(<<2:8, Rest/binary>>, _DecodeDispatcher) ->
  {OctetCount, OctetCountRest} = decode_length(Rest),
  Length = OctetCount * 8,
  <<Number:Length/integer-signed-big, ParseRest/binary>> =
OctetCountRest,
  {Number, ParseRest}.
```

Давайте пойдем дальше: рассмотрим, как мы будем декодировать действительные числа. Хранение действительных чисел в закодированном виде достаточно сложно, несмотря на то, что сами значения хранятся в виде строк. Тип данных для действительных чисел равен 9 (класс – **universal**, форма – **primitive**, идентификатор – 9). Если длина закодированных данных равно 0, то это означает, что мы закодировали действительное число 0.0. Если длина закодированных данных равна 1 (т.е. данные занимают один октет), а после длины идет октет со значением **2#01000000**, то это означает, что мы закодировали действительное число +∞ (в языке Erlang для этого мы используем самое большое действительное число $1.7976931348623157*10^{308}$). Если длина закодированных данных равна 1 (т.е. один октет), а после длины идет октет со значением **2#01000001**, то это означает, что мы закодировали действительное число -∞ (в языке Erlang для этого мы используем самое маленькое действительное число $-1.7976931348623157*10^{308}$). И, наконец, если длина закодированных данных больше 1, то это означает, что нам необходимо декодировать данные дальше.

То, как мы будем декодировать данные дальше, зависит от октета, который идет после закодированной длины (2 старших бита этого октета равны 0, а оставшиеся 6 бит называются **NR**). Если значение **NR** равно **2#000001**, то закодированные данные содержат целое число в строковом представлении. Если значение **NR** равно **2#000010**, то закодированные данные содержат действительное число с фиксированной запятой в строковом представлении. Если значение **NR** равно **2#000011**, то закодированные данные содержат действительное число с плавающей запятой в строковом представлении.

```
decode_real(<<9:8, 0:8, Rest/binary>>, _DecodeDispatcher) ->
{0.0, Rest};
decode_real(<<9:8, 1:8, 2#01000000:8, Rest/binary>>, _
DecodeDispatcher) -> {1.7976931348623157e308, Rest};
decode_real(<<9:8, 1:8, 2#01000001:8, Rest/binary>>, _
DecodeDispatcher) -> {-1.7976931348623157e308, Rest};
decode_real(<<9:8, Rest/binary>>, _DecodeDispatcher) ->
  {TotalOctetCount, OctetCountRest} = decode_length(Rest),
  OctetCount = TotalOctetCount - 1,
  <<2#00:2, NR:6, RealBinary:OctetCount/binary, ParseRest/
binary>> = OctetCountRest,
  RealStr = binary_to_list(RealBinary),
  case NR of
    2#000001 -> {list_to_integer(RealStr) * 1.0, ParseRest};
    2#000010 -> {list_to_float(RealStr), ParseRest};
    2#000011 -> {list_to_float(RealStr), ParseRest}
  end.
```

А мы пойдем дальше: рассмотрим, как декодировать строку октетов (байтовую строку). Тип данных для строки октетов равен 4 (класс – **universal**, форма – **primitive**, идентификатор – 4). После типа данных идет длина, а после нее октеты, составляющие строку октетов. Для их извлечения мы используем операцию соответствия шаблону (для этого нам нужно знать их количество, т.е. длину).

```
decode_octetstring(<<4:8, Rest/binary>>, _DecodeDispatcher) ->
  {OctetCount, OctetCountRest} = decode_length(Rest),
  <<Octet:OctetCount/binary, ParseRest/binary>> =
OctetCountRest,
  {Octet, ParseRest}.
```

Задача, близкая к предыдущей – декодирование битовой строки (строки, в которой количество бит не кратно 8). Тип данных для строки октетов равен 3 (класс – **universal**, форма – **primitive**, идентификатор – 3). После типа данных идет длина, за которой идут данные. Т.к. длина данных – это количество октетов, необходимых для их хранения, а количество бит в битовой строке не кратно 8, то нам необходимо как-то узнать размер остатка битовой строки,

» **Пропустили номер?** Узнайте на с. 104, как получить его прямо сейчас.

который хранится в последнем октете. Для этого первый октет закодированных данных содержит количество неиспользуемых бит в последнем октете, после чего располагается сама битовая строка. Очевидно, что в этом случае длина, требуемая для хранения данных, на 1 больше количества октетов, необходимых для хранения битовой строки.

```
decode_bitstring(<<3:8, Rest/binary>>, _DecodeDispatcher) ->
  {OctetCount, OctetCountRest} = decode_length(Rest),
  <<UnusedBitCount:8, UnusedBitCountRest/binary>> =
OctetCountRest,
  BitstringLength = 8 * (OctetCount - 1) - UnusedBitCount,
  <<Bitstring:BitstringLength/bitstring, _
UnusedBits:UnusedBitCount, ParseRest/binary>> =
UnusedBitCountRest,
  {Bitstring, ParseRest}.
```

Займемся декодированием составных типов данных. Начнем с декодирования последовательностей (или списков). Для них значение типа данных равняется 48 (класс – **universal**, форма – **constructed**, идентификатор – 16). После декодирования типа данных мы получаем длину закодированной последовательности, после чего строку-октет полученной длины мы декодируем так же, как исходную битовую строку. Именно для этого случая мы и передаем в функции декодирования отдельных типов данных функцию-диспетчер. Декодирование содержимого последовательности осуществляется в функции **decode_sequence_content/3**.

```
decode_sequence(<<0:2, 1:1, 16:5, Rest/binary>>, DecodeDis-
patcher) ->
  {OctetCount, OctetCountRest} = decode_length(Rest),
  <<SequenceBinary:OctetCount/binary, SequenceRest/binary>> =
OctetCountRest,
  Sequence = decode_sequence_content(SequenceBinary,
DecodeDispatcher, []),
  {Sequence, SequenceRest}.
```

Декодирование кортежей осуществляется по тем же самым принципам, что и декодирование последовательностей. Для кортежей значение типа данных равняется **16160 = 2#0011111100100000** (класс – **universal**, форма – **constructed**, идентификатор – 32). Отличие этой функции от предыдущей заключается в том, что после декодирования полученную последовательность (или список) мы преобразуем в кортеж.

```
decode_tuple(<<0:2, 1:1, 2#11111:5, 32:8, Rest/binary>>,
DecodeDispatcher) ->
  {OctetCount, OctetCountRest} = decode_length(Rest),
  <<SequenceBinary:OctetCount/binary, SequenceRest/binary>> =
OctetCountRest,
  Sequence = decode_sequence_content(SequenceBinary,
DecodeDispatcher, []),
  {list_to_tuple(Sequence), SequenceRest}.
```

Метод **decode_sequence_content/3** декодирует последовательность октетов, которую мы распознали как последовательность. Для этого данный метод использует передаваемую ему через один из параметров функцию-диспетчер и метод **decode/2**. Может возникнуть следующий вопрос: мы изначально вызываем функцию **decode/2** для декодирования данных, и в результате этого декодирования мы снова вызываем функцию **decode/2**; не приведет ли это к бесконечной рекурсии? Но если хорошо подумать, то можно ответить, что нет, т.к., мы всякий раз вызываем функцию **decode/2** для битовой строки, которая меньше исходной.

```
decode_sequence_content(<<>>, _DecodeDispatcher, ContentList) ->
  lists:reverse(ContentList);
decode_sequence_content(Binary, DecodeDispatcher, ContentList) ->
  {DecodedElement, DecodeRest} = decode(Binary,
DecodeDispatcher),
  decode_sequence_content(DecodeRest, DecodeDispatcher,
[DecodedElement] ++ ContentList).
```

И, наконец, последний метод для декодирования данных работает с закодированными атомами. Для атомов значение типа данных равняется **16161 = 2#0011111100100001** (класс – **universal**, форма – **constructed**, идентификатор – 33). Во всем остальном этот метод тривиален.

```
decode_atom(<<0:2, 0:1, 2#11111:5, 33:8, Rest/binary>>, _De-
codeDispatcher) ->
  {OctetCount, OctetCountRest} = decode_length(Rest),
  <<AtomBinary:OctetCount/binary, ParseRest/binary>> =
OctetCountRest,
  {binary_to_atom(AtomBinary, utf8), ParseRest}.
```

Наш пример по кодированию и декодированию объектов Erlang в соответствии с правилами **ASN.1 BER** закончен. Осталось только проверить, что все работает. Если подходить к такой проверке правильно, то необходимо убедиться, что каждый тип поддерживаемых объектов кодируется и декодируется нашими модулями должным образом (данные проверки удобно реализовать при помощи unit-тестов; мы поговорим о unit-тестировании приложений для Erlang в одной из будущих статей). Кроме того, мы помним, что **ASN.1** – это стандарт взаимодействия различных приложений, написанных на разных языках и под разные платформы. Поэтому необходимым шагом проверки будет проверка взаимодействия нашего приложения со сторонним приложением: данные, закодированные нашим приложением, должны быть раскодированы сторонним, и наоборот (за некоторыми исключениями, о которых мы поговорим в заключении). Однако из-за ограничения места под статью автор приведет лишь пример, что сложная структура данных Erlang после кодирования и декодирования не меняется; это можно считать неплохим smoke-тестом. Что мы делаем: запускаем среду выполнения Erlang, компилируем модули **asn1_encoder** и **asn1_decoder**, создаем функции-диспетчеры для кодирования и декодирования, после чего кодируем некоторый сложный объект и декодируем его. Если все работает правильно, то после декодирования мы должны получить точно такой же объект, как до кодирования.

```
c(asn1_encoder).
c(asn1_decoder).
Source = {abc, 12, 3.14, [], [{ab, true}, <<1:8>>, <<1:7>>], {}}.
Encoder = asn1_encoder:build([]).
Decoder = asn1_decoder:build([]).
Data = asn1_encoder:encode(Source, Encoder).
{Dest, Rest} = asn1_decoder:decode(Data, Decoder).
```

После выполнения всех вышеприведенных команд мы получаем, что переменная **Dest** (объект после кодирования и декодирования) содержит точно такой же объект, что и **Source**, а переменная **Rest** – пустую битовую строку (что логично, ибо у нас не должно остаться не декодированных данных). Можно считать, что наш пример прошел smoke-тест.

Мы закончили большой пример по использованию битовых строк. Про нашу реализацию кодирования и декодирования в соответствии с правилами **ASN.1 BER** можно сказать следующее: мы не реализовали поддержку всех стандартных типов **ASN.1** (мы не поддерживаем, например, идентификаторы, строки, значение NULL) и мы ввели специфичные только для языка Erlang типы (это кортежи и атомы). Что касается первого замечания, то мы исходили из целей построить систему, достаточную для примера, а не для реального использования (и, как кажется автору, это получилось). Что же касается второго замечания, то вполне нормальная практика, когда мы помимо стандартных типов вводим специфичные и известные только для нашего приложения.

На этом, пожалуй, можно поставить точку. Мы завершаем цикл статей, посвященный практикуму по функциональному программированию в языке Erlang. В следующем номере мы приступим к разговору про многозадачность и построение распределенных систем. LXF