Assigments week 5

Joe Armstrong

January 24, 2014

Contents

1 Introduction

These problems are for lectures F4 and F5, held in week 5 of 2014.

Additional material can be found in Chapters 7..10 of the course book *Programming Erlang, 2'nd edition*.

2 Problems

The problems this week deal with binaries, the bit syntax, and types.

To get *Väl Godkänd* you solve all the problems marked `advanced`.

You should also write unit tests for all for functions and add type declarations to all the exported functions. You should run the dialyzer on your code, do not use `-compile(export_all)` when you run the dialyzer.

3 LTV encoding

Binaries and the bit syntax were introduced into Erlang specifically to simplifying parsing data in communications protocols. One common scheme for representing data in is the so called "Tag-Length-Value" encoding scheme.

Suppose we want to transmit strings, integers, and floating point numbers between two programs on different machines. We can encode these as follows:

```
+---------------------------------------+
| 1 | Len (4 bytes) | String of Len Bytes |
+---------------------------------------+
```

```
+-----------------------------------------------+
| 2 | Len (4 bytes) | Integer encoded in Len bytes|
+-----------------------------------------------+

+-----------------------------------------------+
| 3 | Len (4 bytes) | Float encoded in Len Bytes |
+-----------------------------------------------+
```

The first byte is the tag, 1 means "string", 2 means "integer" and so on. This is followed by a 4 byte length count encoded as a signed big-endian integer. The remainder of the packet contains the encoding of the string, integer or float.

Given this encoding write the following routines:

3.1 `encode_seq(L) -> Bin`

`encode_seq(L)` takes an argument `L` which is a list of integers, floats and strings and returns a binary `Bin` containing the data in `L` as a sequence of TLV encoded forms.

3.2 `decode_seq(Bin) -> L`

Is the inverse of `encode_seq`.

3.3 advanced: `safe_encode_seq(L) -> Bin`

Having written `encode_seq`, you will realize there is a problem. In Erlang integers are represented as bignums. If the receiving program is C, then integers will have to be encoded as as signed or unsigned, and we might want to distinguish between 32 and 64 bit integers. Also C does not understand bignums.

Add new tags to the encoding scheme to represent signed, unsigned, 32- and 64- bit integers. Write `safe_encode_seq` to perform the encoding. This is the same as `encode_seq` but it should use the extended tag scheme and it should fail if called with invalid data (for example, if it is called with an integer that will not fit into 64 bits).

3.4 advanced: safe_decode_seq(L) -> Bin

This is the inverse of safe_encode_seq.

4 Packet manipulation

We define a packet as a binary where the first four bytes represent a 4 byte length header N (encoded as a 32 bit big-endian integer) followed by N bytes of data. Like this:

```
+--------------------------------+------------------+
| N (4 bytes big-endian integer) | N bytes of Data  |
+--------------------------------+------------------+
```

The N bytes of data, can be binary data or an Erlang term T encoded with term_to_binary(T).

Write the following functions:

4.1 binary_to_packet(B)

Write a function binary_to_packet(B) -> P that converts a binary B to a packet P.

4.2 packet_to_binary(Packet)

Write the inverse function packet_to_binary(Packet)-> B.

4.3 term_to_packet(Term)

Write the functions term_to_packet(Term) -> Packet.

4.4 packet_to_term(Packet)

Write the inverse function packet_to_term(P) -> Term where P is a packet.

5 Detecting corrupt data

When you have built a packet it might be a good idea to add a checksum to the data, so that you can test it later to see if the packet has been corrupted. The packet now looks like this:

```
+-------------+------------------+
| N | Checksum | N bytes of Data   |
+-------------+------------------+
```

To compute a checksum you can call one of the functions `erlang:crc32` or `erlang:md5`.

5.1 advanced: `term_to_packet_with_checksum(Term)`

Write a safe version of `term_to_packet` which includes a checksum.

5.2 advanced: `packet_to_term_with_checksum(Packet)`

Write the inverse to `term_to_packet_with_checksum(Term)`

Even more advance: is your code really safe – is a checkum enough – could a bad guy corrupt the data and cause a buffer overflow attack?

6 Markdown

Markdown is a language for converting a simple text markup language to HTML. The first exercise is to make a simple markdown processor.

Markdown is described at `http://daringfireball.net/projects/markdown/`

Assume that the input to a markdown processor uses the following conventions to delimit the different blocks in input:

1. Lines starting # denote HTML header (`<h1>`) blocks.
2. Lines starting ## denote HTML header (`<h2>`) blocks.
3. Lines starting with any other characters are paragraphs.

4. Paragraphs are terminated by completely blank lines, or header blocks.

6.1 advanced: `markdown:expand_binary(F)`

Start by writing a function `markdown:parse_binary(Bin)` which parses a binary and returns a list of markdown blocks in the binary. For example:

```
> markdown:parse_binary(<<"# hello
This is a paragraph
with two lines
#Another heading
and more data">>).
[{h1,<<"hello">>},
 {para,<<"This is a paragraph\nwith two lines\n"},
 {h1,<<"Another heading">>}]
```

6.2 advanced: `markdown:parse_to_html(L)`

Once we have a parse tree. We want to turn it into HTML, write a function to expand the parser tree that you obtained in the previous section, and convert it to HTML. For example:

```
> markdown:parse_to_html([
  {h1,<<"hello">>},
  {para,<<"This is a paragraph\nwith two lines\n"},
  {h1,<<"Another heading">>}]).
  <<"<h1>Hello</h1><p>This is a ...">>.
```

6.3 advanced: `markdown:expand_file(F)`

The function `markdown:expand_file(F)` should read the file `F` using `file:read_file(F)` the result is be a tuple `{ok, B}` where `B` is a binary containing the content of the file `F`. Parse the resultant binary using `markdown:parse_binary(B)` and turn the parse tree into HTML using

`markdown:parse_to_html`. Write the result in a file `File.html`. Test this in a browser for various inputs.

7 Notes

In writing `markdown:parse_binary` you'll probably notice that the markdown documentation is rather vague about whether things like leading and trailing newlines belong to the paragraphs and headings or not.

Assume the input to the parser is:

```
# Header1
The start of paragraph one ...
```

Does the newline separating the header and the first line of the paragraph belong to the header or to the first line of the paragraph or to neither?

Depending upon the answer to this question there are three possible Erlang parse trees, for example:

- `[{h1,<<"Header1">>},{para,<<"The start ...">>},...]`
- `[{h1,<<"Header1\n">>},{para,<<"The start ...">>},...]`
- `[{h1,<<"Header1">>},{para,<<"\nThe start ...">>},...]`

What about the white space following the # character, should this be represented in the parse tree or not?

Once we have a parse tree of the form `[{h1,...}, {para, ...}]` is is easy to render this as HTML or LaTeX, note that any white space issues (the kind I mentioned earlier) do not seem to affect the rendered HTML.

How white space is handled is crucial to accurately rendering markdown, if we extend the program to handle preformmated text, we'll get problems if this is problem is not correctly analyzed.

For fun, you might like to connect this code with last weeks exercises in rendering inlines, to make a more sophisticated markdown processor.