Programming Efficiently with Binaries and Bit Strings

Per Gustafsson

Department of Information Technology
Uppsala University, Sweden   Ericsson AB, Sweden
pergu@it.uu.se

Abstract

A new datatype, the *bit string*, and a new construct for manipulating binaries, binary comprehensions, are included in the R12B release of Erlang/OTP. In addition to this the implementation of binary construction and matching have been altered to make straightforward programs that operates on binaries or bit strings more efficient.

This paper will describe the new additions to the language and show how they can be used efficiently given the new optimizations of binary pattern matching and binary construction. It also includes some performance numbers to give an idea of the gains that can be made with the new optimizations.

1. Introduction

Binaries have received a makeover in the R12B release of Erlang with the introduction of *bit strings* and *extended comprehensions* as well as optimization of both binary construction and pattern matching.

Binaries have been a part of Erlang for a long time, and there has been a nice syntax for manipulating binaries since the R7B release of Erlang/OTP [3]. There has been some complaints about the using binaries for formats that are bit-oriented rather than byte-oriented since this tends to lead to complicated and error-prone code [2]. Bit strings are introduced to solve exactly this problem.

List comprehensions are used a lot in Erlang. They tend to make programs more compact and readable avoiding boilerplate code. In 2004 Jay Nelson suggested that there could also be binary comprehensions, a compact syntax for operating on binaries. The suggestion was formalized at the 2005 Erlang Workshop [1] and with some syntax changes this proposal was added as an optional feature in Erlang R11B. It will finally be a supported feature in R12B. The feature not only allows binary comprehension but also the use of binary generators in list comprehensions as well as list generators in binary comprehensions. Togeteher we call these features extended comprehensions which give users versatile abstractions for converting data between structured term formats and binary formats.

In addition to this binary comprehensions give the users a surefire way to use the new optimizations of binary construction and pattern matching. The optimization of construction of binaries might be the most important of the two as it makes it possible to build binaries in a piece-wise manner in linear time. This has been a problem in previous versions of Erlang forcing programmers to create lists of binaries which are then concatenated at the end to get efficient algorithms. This pattern tends to make algorithms more complicated than necessary.

The optimization of binary pattern matching is also important as it decreases the need to do unrolling of code that iterates over binary or keeping a counter to iterate over a binary. This optimization tends to make short natural implementations of functions which iterates over a binary efficient. Which is good as the hand-written optimizations above can introduce subtle bugs.

In this paper we will describe the new additions to the language in Section 2 and 3. Then we will give a short introduction to the implementation of operations on bit strings and binaries in Section 4 in order to be able to explain the new optimizations in Section 5 and give the reader some idea of how he should program to make use of them. Finally we have some performance measurements in Section 6 and conlusions in Section 7.

2. Bitstrings and binaries

A new datatype the *bit string* is introduced into Erlang. A bit string is a sequence of bits of any length this separates it from a *binary* which is a sequence of bits where the number of bits is evenly divisible by eight. These definitions implies that any binary is also a bit string.

2.1 Manipulating bit strings using the bit syntax

A bit syntax expression:

```
<<Seg1,...,SegN>>
```

Evaluates to a bit string. If the sum of the sizes of all segments in the expression is divisible by eight the result is also a binary. Previously such expression could only evaluate to binaries and a runtime error was raised if this was not the case.

With this extension the expression `Bin = <<1:9>>` which previously caused a runtime error now creates a 9-bit binary. To be able to use this bit string to build a new bigger bit string we can write:

```
<<Bin/bitstring, 0:1>>
```

Note the use of bitstring as the type. This expands to binary-unit:1 where as the binary type would have expanded to binary-unit:8. Since bitstring is a long word to write in a binary pattern there is an alias *bits* which is used in the rest of this paper, similarily for binary there is a new shorthand *bytes*.

To match out a bit-level binary we also use the bit string type as in :

```
case Bin of
  <<1:1,Rest/bits>> -> Rest;
  <<0:1,_/bits>> -> 0
end
```

This allows us to avoid situations were we previously had to calculate padding.

Example 2.1 A format from the IS 683-PRL protocol which consists of a 5-bit field describing how many 11-bit fields it was followed by. Decoding this format required a complicated calculation of padding to implement in a straightforward manner. The result is shown in Program 1.

Program 1 Decoding a format in the IS 683-PRL protocol

```
decode(<<NumChans:5, _Pad:3, _Rest/binary>> = Bin) ->
  decode(Bin, NumChans, NumChans, []).

decode(_, _, 0, Acc) ->
  Acc;
decode(Bin, NumChans, N, Acc) ->
  SkipBef = (N - 1) * 11,
  SkipAfter = (NumChans - N) * 11,
  Pad = (8 - ((NumChans * 11 + 5) rem 8)) rem 8,
  <<_:5, _:SkipBef, Chan:11, _:SkipAfter, _:Pad>> = Bin,
  decode(Bin, NumChans, N - 1, [Chan | Acc]).
```

With the introduction of bit strings it can be implemented without any padding calculations at all as:

```
decode(<<NumChans:5, Rest/bits>>) ->
  decode(NumChans, Rest, []).

decode(0, _, Acc) ->
  lists:reverse(Acc);
decode(N, <<Chan:11,Rest/bits>>, Acc) ->
  decode(N-1, Rest, [Chan|Acc]).
```

2.2 BIFs for manipulating bit strings

The current builtin functions for manipulating binaries will still only be defined for binaries. We will instead introduce four new BIFs which operate on bit strings .hey are described in Table 1.

3. Bit String Comprehensions

Bit string comprehensions are analogous to List Comprehensions. They are used to generate bit strings efficiently and succintly. Bit string comprehensions are written with the following syntax:

```
<< BitString || Qualifier1,...,QualifierN >>
```

`BitString` is a bit string expression, and each `Qualifier` is either a *generator*, a *bit string generator* or a *filter*.

generator: `Pattern <- ListExpr`
Where `ListExpr` must be an expression which evaluates to a list of terms.

bit string generator: `BitstringPattern <= BitStringExpr`
Where `BitStringExpr` must be an expression which evaluates to a bitstring.

filter : `Expr`
Where `Expr` must be an expression which evaluates to `true` or `false`

The variables in the generator patterns shadow variables in the function clause surrounding the bit string comprehensions. A bit string comprehension returns a bit string, which is created by concatenating the results of evaluating `BitString` for each combination of bit string generator or ordinary generator elements for which all filters are true.

Example 3.1 A simple comprehension which changes all lower case ascii characters in the bit string `Bits` into upper case characters.

```
<< <<(to_upper(X))>> || <<X>> <= Bits >>
```

This has the same semantics as the following expression:

```
bits_to_upper(Bits)

bits_to_upper(<<X,Rest/bits>>) ->
  <<(to_upper(X)), (bits_to_upper(Rest))/bits>>;
bits_to_upper(_) -> <<>>.
```

The translation to Erlang code is pretty straightforward, but the runtime for the Erlang program above is quadratic in the size of `Bits`, whereas the comprehension will be evaluated in linear time.

Since both ordinary list generators and bit string generators are allowed in bit string comprehensions they can be used to convert a list of data structures to a bit string representation.

Example 3.2 Consider the case where you have a list of three tuples where the first value in the tuple is one of 6 different atoms, the second value is a 16-bit integer and the third value is a float. Than you can turn that into a compact format using the following code:

```
<< <<(atom_to_int(Atom)):3,Int:16,Float/float>> ||
     {Atom,Int,Float} <- List >>.
```

Where `atom_to_int` maps the six different atoms to integers between 0 and 5.

3.1 Bit String Generators in List Comprehensions

In addition to introducing bit string comprehensions we also allow bit string generators in list comprehensions. This is useful for turning bit strings into structured data. One example when it is useful is for the problem described in Example 2.1. Using a bit string generator in a list comprehension this can be written as:

```
decode(<<N:5,Chans:N/bits-unit:11,_/bits>>) ->
  [Chan || <<Chan:11>> <- Chans].
```

4. Implementation

In order to describe the new optimizations of binary pattern matching and construction I must first describe how bit strings are represented and bit string operations are implemented.

4.1 The bit string datatype

The layout of bitstrings is a little bit complicated. The actual data in a bit string resides off heap. There is a data structure on the heap that is called a REFC binary that points to the off heap data. Bitstrings are so called sub-binaries which also reside on the heap. They point to REFC binary and they also contain offset and size fields. They never point directly to the off-heap data. The situation is described in Figure 4.1.

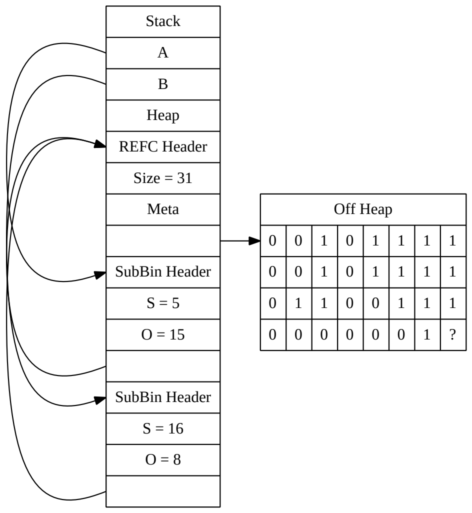
4.2 Bit String Construction

A bit string construction expression that has the form:
$\texttt{<<ve}_1\texttt{:se}_1\texttt{/t}_1\texttt{,...,ve}_n\texttt{:se}_n\texttt{/t}_n\texttt{>>}$ is translated as follows. We start by evaluating all the value and size expressions so that we end up with an expression of the form $\texttt{<<v}_1\texttt{:s}_1\texttt{/t}_1\texttt{,...,v}_n\texttt{:s}_n\texttt{/t}_n\texttt{>>}$ where all the $\texttt{v}_i$:s are values and all the $\texttt{s}_i$:s are non-negative integers. If any $\texttt{s}_i$ is a negative value, a run-time exception is raised.

Then the the resulting size of the binary we are building is calculated as $\sum_{i=1}^{n} s_i$. An appropriate amount of off heap space is allocated for the data and the REFC binary is created on the heap. Then each segment is written into the data part. When this is done the sub binary which becomes the result of the expression is created.

Signature	Definition
`bit_size/1::bitstring() -> integer()`	Returns the size of a bit string in bits. This BIF is allowed in guards.
`list_to_bitstring/1::bitstring_list() -> bitstring()`	Concatenates the bit strings and chars in the bitstring list to create a bit string. A bitstring list is an io list which can contain bit strings as well as binaries the chars in the bitstring list are treated as if they were bit strings consisting of 8 bits.
`bitstring_to_list/1::bitstring() -> [char()|bitstring()]`	Turns a bit string into a list of characters and if the number of bits in the bit string is not evenly divisible by eight the last element in the list is a bit string consisting of the last 1-7 bits of the original bit string.
`is_bitstring/1::any() -> bool()`	Returns true if the argument is a bit string, otherwise it returns false. This BIF is allowed in guards.

Table 1. New Builtin Functions for manipulating bit strings



4.3 Binary Pattern Matching

Consider the following expression:

```
<<8:16, X:32, Bin/bits>> = Bits
```

This gets compiled into the sequence shown in Figure 1. The instruction `create_matchstate` takes a bit string and creates a matchstate to be used during the matching. The matchstate contains the size of the bitstring we are matching against, the offset we are at and a pointer to the data. The `get_integer` instruction takes a matchstate and a size and reads that number of bits, turns it into an integer and updates the offset in the matchstate. The `get_bitstring` function creates a sub-binary from the matchstate.

A more complicated matching with several patterns is compiled into a tree of instuctions for exampleif we have:

```
case Bits of
  <<A:8, 1:8, X:8, Bin/bits>>  -> cont1;
  <<A:8, 2:8, X:16, Bin/bits>> -> cont2;
  <<>>                         -> cont3
end
```

We end up with the tree of instructions shown in Figure 2.

We have some new instructions:

`save_matchstate(N,MS)` This instruction saves the present offset in the matchstate in save slot N.

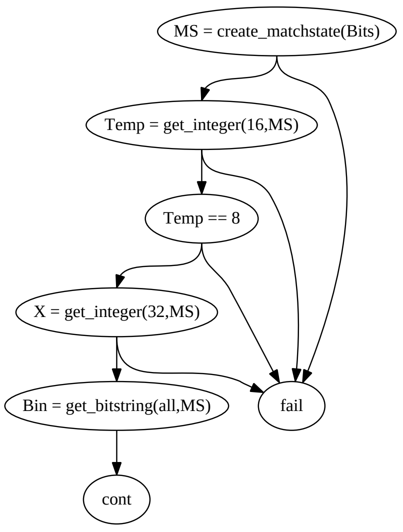

MS = create_matchstate(Bits)
Temp = get_integer(16,MS)
Temp == 8
X = get_integer(32,MS)
Bin = get_bitstring(all,MS)
fail
cont

Figure 1. Matching graph for `<<8:16, X:32, Bin/bits>> = Bits`

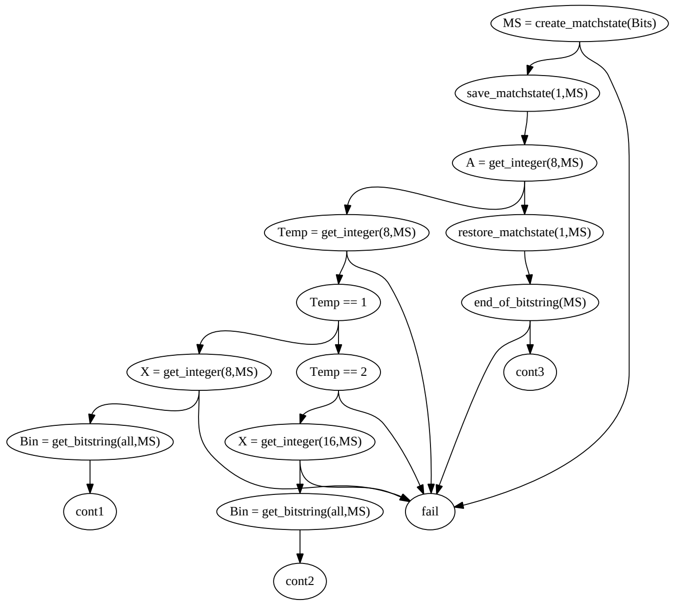


Figure 2. Matching graph for case statement

`restore_matchstate(N,MS)` This instruction loads the offset value from slot N and makes it the present offset value.

`end_of_bitstring` This instruction checks that the offset in the matchstate is equal to the size. That is that we have reached the end of the bit string

5. Optimizations

In R12B both binary construction and binary pattern matching has been optimized. In this section we will describe these optimizations and discuss how to write code that best utilizes trhe optimizations.

5.1 Binary Construction Optimization

The basis of this optimization is that it the emulator can create bit strings with extra uninitialized space, so if a bit string is built by continously appending to a binary the data does not need to be copied if there is enough uninitialized data at the end of the bit string.

`Bits` contains a bit string of 1000 bits followed by 600 bits of uninitialized data.

In the expression

```
NewBits = <<Bits/bits, 12:32>>
```

`NewBits` gets bound to a bit string of 1032 bits followed by 568 bits of uninitialized data, `Bits` on the other hand can no longer be appended to.

On the other hand if we have this expression:

```
NewBits = <<Bits/bits, 12:640>>
```

Since there is not enough uninitalized data `NewBits` becomes a new bit string consisting of 1640 bits followed by 1640 bits of uninitialized data. `Bits` remains the same a bit string of 1000 bits with 600 bits uninitialized data.

What does this mean in practice when your programming?

- It means you can build bit strings piecewise in linear time
- It means that when your building a bit string from a list or from an other bit string and you want to have the same order of your pieces you should use tail calls and an acumulator
- It means that you can reverse a bit string efficiently without turning it into a list

Let us see some examples of efficient programs for building bit strings:

Example 5.1 This function reverses a bit string consisting of 32 bit integers:

```
reverse_32bit(<<X:32,Rest/bits>>) ->
  <<(reverse_32bit(Rest))/bits,X:32>>;
reverse_32bit(<<>>) ->
  <<>>.
```

Not that when we are constructing the answer the first element of the new bit string is the growing bit string.

Note that we use direct recursion in order to get the reverse order in the result in the following example we want to perserve the order of the input.

Example 5.2 This simple function stores a double in 32-bits if it is prefaced by a zero if it is prefaced by a one it uses 64-bits.

```
save_floats(Bits) ->
  save_floats(Bits, <<>>).
```

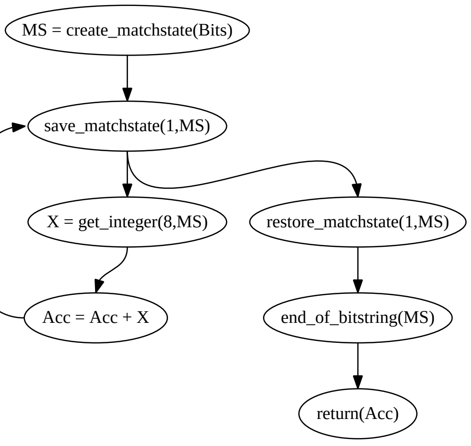


Figure 4. Optimized code for `sum1/2`

```
save_floats(<<0:1,F:64/float,Rest/bits>>, Acc) ->
  save_floats(Rest, <<Acc/bits,0:1,F:32/float>>);
save_floats(<<1:1,F:64/float,Rest/bits>>, Acc) ->
  save_floats(Rest, <<Acc/bits,1:1,F:64/float>>);
save_floats(<<>>,Acc) ->
  Acc.
```

5.2 Binary Pattern Matching Optimization

To describe the new optimization of binary pattern matching consider these two functions which calculates the sum of the bytes in a bit string:

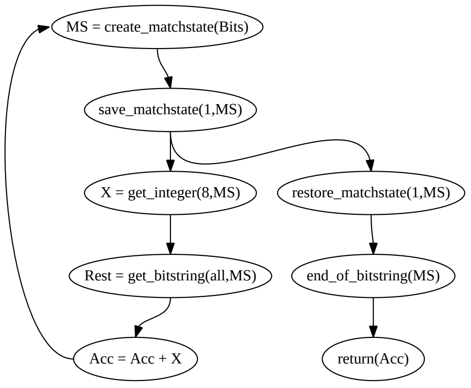
```
sum1(Bits) ->
  sum1(Bits, 0).

sum1(<<X,Rest/bits>>, Acc) ->
  sum1(Rest, Acc+X);
sum1(<<>>, Acc) -> Acc.

sum2(Bits) ->
  sum2(Bits,0,0).

sum2(Bits,N,Acc) ->
  case Bits of
    <<_:N,X,Rest/bits>> ->
       sum2(Bits,N+8,Acc+X);
    <<_/bits>> ->
       Acc
  end.
```

The generated code for `sum1/2` is shown in Figure 3(a). In each iteration of the loop a sub-binary is created from the match sate only to promptly be turned in to a new match state in the next iteration.

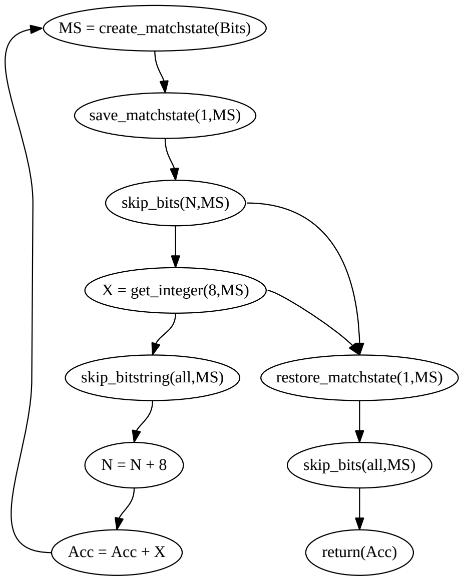
For `sum2/3` we avoid creating this sub-binary, but we still have to create the match state in each iteration.

The new optimization of binary pattern matching follows from the observation that it is unnecessary to convert a match state into sub-binary only to immediatly convert it back to a match state. Instead we can keep the match state in the loop. Using this optimization the code for `sum1/2` is shown in Figure 4.

How should we write code to make it possible to apply this optimization? The most important thing is to make sure that the binary we are matching against is not used for anything else in the function. In addition to this we need to make sure that the sub-binary we are creating is only used in a self recursive call.



(a) Generated code for `sum1/2`



(b) Generated code for `sum2/3`

Figure 3. Code generated for two different functions calculating the byte sum of a bit string

```
f(<<Pattern1,...,Rest/bits>>,...) ->
   ... % Rest is not used here
   f(Rest,...);
f(<<Pattern2,...,Rest/bits>>,...) ->
   ... % Rest is not used here
   f(Rest,...);

...

f(<<>>, ...) ->
  ReturnValue
```

Figure 5. Function skeleton that will be optimized

A good model for functions that want to make sure they use this optimization is shown in figure 5.

6. Performance

In this section we will give some performance figures and compare some different approaches to write programs operating on bit strings as well as comparing handling of bit strings in R11B-5 and R12B. All of the benchmarks in this section have been run on a unicore 2.4 GHz Pentium 4 with 1 GB of memory, running Ubuntu 7.10.

6.1 IP-packet checksum

This program exists in four different flavors. Two which creates sub-binaries like the program in Figure 3(a) the difference between these programs is that one of them unrolls the loop eight times whereas the other program does no unrolling. The programs are called *Sub* and *SubUnrolled*. The other two programs use the same type of iteration as the program in Figure 3(b), one of these progams is also unrolled. They are called *Iter* and *IterUnrolled*. They each calculate the checksum for a 658 kB file one hundred times. The runtimes can be found in Table 2. The four different functions can be found in Program [**?**] in the appendix.

Program	BEAM R12B-0	HiPE R12B-0
Bit String Comprehension	10.43	2.69
Bit String Recursion	14.49	3.41
List Comprehension	10.22	6.21

Table 3. Runtimes in seconds for making 65.8 MB of data all upper case

The results suggest that performance of binary pattern matching in general is better in R12B, but paritcularily when using sub-binaries. The effect of doing unrolling decrease from a factor four in R11B to less than approximatly a factor 1.5, which suggests that good performance can be had without adding ugly unrolling.

6.2 Upper Case

In the second experiment binaries are both constructed and pattern matched on, but it is a pretty simple program. It simply turns a binary string into an all upper case binary string. There are three different versions of the function all of them are shown in Program 2 in the appendix.

It was not really relevant to run this benchmark on R11B-5 since the bit string recursion function had a quadratic cost and bit string comprehensions were very inefficient. They were thus only run on R12B. The input was the same as in the IP-checksum case, a 658 kB file that was turned into an all upper case file one hundred times. The results are shown in Table 3.

The results seem to suggest that with BEAM bit string comprehensions are competitive with operating on a list while it becomes superior when native compilation is used. It is also superior to explicit recursion. This is the case since it is easier to analyze a bit string comprehension and thus construction and matching of bit strings can be optimized further.

The implementation of bit string comprehensions can be improved further. In many cases the size of the resulting bit string can be computed beforehand. This is not done yet, but we expect to implement this in future releses of Erlang/OTP.

Program	BEAM R11B-5	HiPE R11B-5	BEAM R12B-0	HiPE R12B-0
Sub	10.18	3.69	2.66	0.62
SubUnrolled	2.17	0.90	1.13	0.38
Iter	8.31	2.90	5.09	2.15
IterUnrolled	2.16	0.78	1.54	0.58

Table 2. Runtimes in seconds for calculating checksums of 65.8 MB of data

7. Conclusions

This is not a comprehensive description of how to use binaries and bit strings efficiently in your programs. It is simply a short description of how binaries have been extended into bit strings and how various operations on bit strings are implemented. We also try to describe how we optimize these operations. Hopefully this description will help you write shorter and easier and more efficient programs in the future. What we want you to take away from this paper is summarized in the following bullet points.

- Bit strings makes it much easier to deal with bit-oriented data in Erlang
- When you are building new bit strings make sure you append new data to the end of an old bit string
- When you iterate over a bit string use a direct style matching similar to what you would do for lists
- If you are doing a map operation over bit strings use bit string comprehensions to get efficient and concise code
- Write simple straight-forward code first to see if the optimizations makes it fast enough. Then you can try various approaches to make it faster.

References

[1] P. Gustafsson and K. Sagonas. Bit-level binaries and generalized comprehensions in Erlang. In *Proceedings of the Fourth ACM SIGPLAN Erlang Workshop*, pages 1–8. ACM Press, Sept. 2005.

[2] M. Läng. Erlang in the corelatus mtp2 signalling gateway, Oct. 2001. Available at http://www.erlang.se/euc/01/.

[3] P. Nyblom. The bit syntax - the released version. In *Proceedings of the Sixth International Erlang/OTP User Conference*, Oct. 2000. Available at http://www.erlang.se/euc/00/.

8. Appendix

Program 2 Three ways to make a binary all upper case

```
bit_string_comp(Bin) ->
  << <<(to_upper(X))>> || <<X>> <= Bin >>.

bit_string_recursion(Bin) ->
  bit_string_recursion(Bin, <<>>).

bit_string_rec(<<X,Rest/binary>>, Acc) ->
  bit_string_rec(Rest,<<Acc/binary,(to_upper(X))>>);
bit_string_rec(<<>>, Acc) -> Acc.

list_comprehension(Bin) ->
  list_to_binary([to_upper(X) ||
                  X <- binary_to_list(Bin)]).

to_upper(X) when X >= $a, X =< $z ->
  X + ($A-$a);
to_upper(X) ->
  X.
```

Program 3 Four ways to calculate an IP checksum

```
-define(INT16MAX, 65535).

sub(<<N1:16, Rem/binary>>,Csum) ->
   sub(Rem, do_trunc(Csum+N1));
sub(<<N1:8>>,Csum) ->
  sub(<<>>,do_trunc(Csum+(N1 bsl 8)));
sub(<<>>,Csum) when Csum > ?INT16MAX ->
  Val=(Csum band ?INT16MAX) + (Csum bsr 16),
   sub(<<>>,Val);
sub(<<>>,Csum) -> (bnot Csum) band ?INT16MAX.

sub_unrolled(<<N1:16,N2:16,N3:16,N4:16,N5:16,N6:16,
               N7:16,N8:16,Rem/binary>>, Csum) ->
  sub_unrolled(Rem,do_trunc(Csum+N1+N2+N3+N4+N5+N6+N7+N8));
sub_unrolled(<<N1:16, Rem/binary>>,Csum) ->
  sub_unrolled(Rem, do_trunc(Csum+N1));
sub_unrolled(<<N1:8>>,Csum) ->
  sub_unrolled(<<>>,Csum+(N1 bsl 8));
sub_unrolled(<<>>,Csum) when Csum > ?INT16MAX ->
  Val=(Csum band ?INT16MAX) + (Csum bsr 16),
  sub_unrolled(<<>>,Val);
sub_unrolled(<<>>,Csum) ->
  (bnot Csum) band ?INT16MAX.

iter(N,Bin,Csum) ->
  case Bin of
    <<_:N/binary, N1:16,_/binary>> ->
      iter(N+2,Bin,do_trunc(Csum+N1));
    <<_:N/binary, Num:8>> ->
      iter(N+1,Bin,do_trunc(Csum+(Num bsl 8)));
    _ when Csum > ?INT16MAX ->
       Val = (Csum band ?INT16MAX + (Csum bsr 16)),
      iter(N,Bin,Val);
    _ ->
      (bnot Csum) band ?INT16MAX
 end.

iter_unrolled(N,Bin,Csum) ->
  case Bin of
    <<_:N/binary, N1:16,N2:16,N3:16,
      N4:16,N5:16,N6:16,N7:16,N8:16,
      _/binary>> ->
      iter_unrolled(N+16,Bin,
        do_trunc(Csum+N1+N2+N3+N4+N5+N6+N7+N8));
    <<_:N/binary, N1:16,_/binary>> ->
      iter_unrolled(N+2,Bin,do_trunc(Csum+N1));
    <<_:N/binary, Num:8>> ->
      iter_unrolled(N+1,Bin,Csum+(Num bsl 8));
    _ when Csum > ?INT16MAX ->
       Val = (Csum band ?INT16MAX + (Csum bsr 16)),
      iter_unrolled(N,Bin,Val);
    _ ->
      (bnot Csum) band ?INT16MAX
  end.

do_trunc(Csum) when Csum > 16#6ffffff, Csum < 16#7ffffff  ->
  Csum band ?INT16MAX + (Csum bsr 16);
do_trunc(Csum) -> Csum.
```