Programming Examples

version 5.6

Chapter 1

Programming Examples

This chapter contains examples on using records, funs, list comprehensions and the bit syntax.

1.1 Records

1.1.1 Records vs Tuples

The main advantage of using records instead of tuples is that fields in a record are accessed by name, whereas fields in a tuple are accessed by position. To illustrate these differences, suppose that we want to represent a person with the tuple {`Name, Address, Phone`}.

We must remember that the `Name` field is the first element of the tuple, the `Address` field is the second element, and so on, in order to write functions which manipulate this data. For example, to extract data from a variable `P` which contains such a tuple we might write the following code and then use pattern matching to extract the relevant fields.

```
Name = element(1, P),
Address = element(2, P),
...
```

Code like this is difficult to read and understand and errors occur if we get the numbering of the elements in the tuple wrong. If we change the data representation by re-ordering the fields, or by adding or removing a field, then all references to the person tuple, wherever they occur, must be checked and possibly modified.

Records allow us to refer to the fields by name and not position. We use a record instead of a tuple to store the data. If we write a record definition of the type shown below, we can then refer to the fields of the record by name.

```
-record(person, {name, phone, address}).
```

For example, if `P` is now a variable whose value is a `person` record, we can code as follows in order to access the name and address fields of the records.

```
Name = P#person.name,
Address = P#person.address,
...
```

Internally, records are represented using tagged tuples:

```
{person, Name, Phone, Address}
```

1.1.2 Defining a Record

This definition of a person will be used in many of the examples which follow. It contains three fields, name, phone and address. The default values for name and phone is "" and [], respectively. The default value for address is the atom undefined, since no default value is supplied for this field:

```
-record(person, {name = "", phone = [], address}).
```

We have to define the record in the shell in order to be able use the record syntax in the examples:

```
> rd(person, {name = "", phone = [], address}).
person
```

This is due to the fact that record definitions are available at compile time only, not at runtime. See shell(3) for details on records in the shell.

1.1.3 Creating a Record

A new person record is created as follows:

```
> #person{phone=[0,8,2,3,4,3,1,2], name="Robert"}.
#person{name = "Robert",phone = [0,8,2,3,4,3,1,2],address = undefined}
```

Since the address field was omitted, its default value is used.

There is a new feature introduced in Erlang 5.1/OTP R8B, with which you can set a value to all fields in a record, overriding the defaults in the record specification. The special field _, means "all fields not explicitly specified".

```
> #person{name = "Jakob", _ = '_'}.
#person{name = "Jakob",phone = '_',address = '_'}
```

It is primarily intended to be used in ets:match/2 and mnesia:match_object/3, to set record fields to the atom '_'. (This is a wildcard in ets:match/2.)

1.1.4 Accessing a Record Field

```
> P = #person{name = "Joe", phone = [0,8,2,3,4,3,1,2]}.
#person{name = "Joe",phone = [0,8,2,3,4,3,1,2],address = undefined}
> P#person.name.
"Joe"
```

1.1.5 Updating a Record

```
> P1 = #person{name="Joe", phone=[1,2,3], address="A street"}.
#person{name = "Joe",phone = [1,2,3],address = "A street"}
> P2 = P1#person{name="Robert"}.
#person{name = "Robert",phone = [1,2,3],address = "A street"}
```

1.1.6 Type Testing

The following example shows that the guard succeeds if `P` is record of type `person`.

```
foo(P) when is_record(P, person) -> a_person;
foo(_) -> not_a_person.
```

1.1.7 Pattern Matching

Matching can be used in combination with records as shown in the following example:

```
> P3 = #person{name="Joe", phone=[0,0,7], address="A street"}.
#person{name = "Joe",phone = [0,0,7],address = "A street"}
> #person{name = Name} = P3, Name.
"Joe"
```

The following function takes a list of `person` records and searches for the phone number of a person with a particular name:

```
find_phone([#person{name=Name, phone=Phone} | _], Name) ->
    {found,  Phone};
find_phone([_| T], Name) ->
    find_phone(T, Name);
find_phone([], Name) ->
    not_found.
```

The fields referred to in the pattern can be given in any order.

1.1.8 Nested Records

The value of a field in a record might be an instance of a record. Retrieval of nested data can be done stepwise, or in a single step, as shown in the following example:

```
-record(name, {first = "Robert", last = "Ericsson"}).
-record(person, {name = #name{}, phone}).

demo() ->
  P = #person{name= #name{first="Robert",last="Virding"}, phone=123},
  First = (P#person.name)#name.first.
```

In this example, `demo()` evaluates to `"Robert"`.

1.1.9 Example

```
%% File: person.hrl

%%--------------------------------------------------------------
%% Data Type: person
%% where:
%%    name:  A string (default is undefined).
%%    age:   An integer (default is undefined).
%%    phone: A list of integers (default is []).
%%    dict:  A dictionary containing various information
%%           about the person.
%%           A {Key, Value} list (default is the empty list).
%%--------------------------------------------------------------
-record(person, {name, age, phone = [], dict = []}).

-module(person).
-include("person.hrl").
-compile(export_all). % For test purposes only.

%% This creates an instance of a person.
%%   Note: The phone number is not supplied so the
%%         default value [] will be used.

make_hacker_without_phone(Name, Age) ->
   #person{name = Name, age = Age,
           dict = [{computer_knowledge, excellent},
                   {drinks, coke}]}.

%% This demonstrates matching in arguments

print(#person{name = Name, age = Age,
              phone = Phone, dict = Dict}) ->
  io:format("Name: ~s, Age: ~w, Phone: ~w ~n"
            "Dictionary: ~w.~n", [Name, Age, Phone, Dict]).

%% Demonstrates type testing, selector, updating.

birthday(P) when record(P, person) ->
   P#person{age = P#person.age + 1}.

register_two_hackers() ->
   Hacker1 = make_hacker_without_phone("Joe", 29),
   OldHacker = birthday(Hacker1),
   % The central_register_server should have
   % an interface function for this.
   central_register_server ! {register_person, Hacker1},
   central_register_server ! {register_person,
             OldHacker#person{name = "Robert",
                              phone = [0,8,3,2,4,5,3,1]}}.
```

1.2 Funs

1.2.1 Example 1 - map

If we want to double every element in a list, we could write a function named `double`:

```
double([H|T]) -> [2*H|double(T)];
double([])    -> [].
```

This function obviously doubles the argument entered as input as follows:

```
> double([1,2,3,4]).
[2,4,6,8]
```

We now add the function `add_one`, which adds one to every element in a list:

```
add_one([H|T]) -> [H+1|add_one(T)];
add_one([])    -> [].
```

These functions, `double` and `add_one`, have a very similar structure. We can exploit this fact and write a function `map` which expresses this similarity:

```
map(F, [H|T]) -> [F(H)|map(F, T)];
map(F, [])    -> [].
```

We can now express the functions `double` and `add_one` in terms of `map` as follows:

```
double(L)  -> map(fun(X) -> 2*X end, L).
add_one(L) -> map(fun(X) -> 1 + X end, L).
```

`map(F, List)` is a function which takes a function `F` and a list `L` as arguments and returns the new list which is obtained by applying `F` to each of the elements in `L`.

The process of abstracting out the common features of a number of different programs is called procedural abstraction. Procedural abstraction can be used in order to write several different functions which have a similar structure, but differ only in some minor detail. This is done as follows:

1. write one function which represents the common features of these functions
2. parameterize the difference in terms of functions which are passed as arguments to the common function.

1.2.2 Example 2 - foreach

This example illustrates procedural abstraction. Initially, we show the following two examples written as conventional functions:

1. all elements of a list are printed onto a stream
2. a message is broadcast to a list of processes.

```
print_list(Stream, [H|T]) ->
    io:format(Stream, "~p~n", [H]),
    print_list(Stream, T);
print_list(Stream, []) ->
    true.

broadcast(Msg, [Pid|Pids]) ->
    Pid ! Msg,
    broadcast(Msg, Pids);
broadcast(_, []) ->
    true.
```

Both these functions have a very similar structure. They both iterate over a list doing something to each element in the list. The “something” has to be carried round as an extra argument to the function which does this.

The function `foreach` expresses this similarity:

```
foreach(F, [H|T]) ->
    F(H),
    foreach(F, T);
foreach(F, []) ->
    ok.
```

Using `foreach`, `print_list` becomes:

```
foreach(fun(H) -> io:format(S, "~p~n",[H]) end, L)
```

`broadcast` becomes:

```
foreach(fun(Pid) -> Pid ! M end, L)
```

`foreach` is evaluated for its side-effect and not its value. `foreach(Fun ,L)` calls `Fun(X)` for each element `X` in `L` and the processing occurs in the order in which the elements were defined in `L`. `map` does not define the order in which its elements are processed.

1.2.3 The Syntax of Funs

Funs are written with the syntax:

```
F = fun (Arg1, Arg2, ... ArgN) ->
        ...
    end
```

This creates an anonymous function of `N` arguments and binds it to the variable `F`.

If we have already written a function in the same module and wish to pass this function as an argument, we can use the following syntax:

```
F = fun FunctionName/Arity
```

With this form of function reference, the function which is referred to does not need to be exported from the module.

We can also refer to a function defined in a different module with the following syntax:

```
F = {Module, FunctionName}
```

In this case, the function must be exported from the module in question.

The follow program illustrates the different ways of creating funs:

```
-module(fun_test).
-export([t1/0, t2/0, t3/0, t4/0, double/1]).
-import(lists, [map/2]).

t1() -> map(fun(X) -> 2 * X end, [1,2,3,4,5]).

t2() -> map(fun double/1, [1,2,3,4,5]).

t3() -> map({?MODULE, double}, [1,2,3,4,5]).

double(X) -> X * 2.
```

We can evaluate the fun `F` with the syntax:

```
F(Arg1, Arg2, ..., Argn)
```

To check whether a term is a fun, use the test `is_function/1` in a guard. Example:

```
f(F, Args) when is_function(F) ->
   apply(F, Args);
f(N, _) when is_integer(N) ->
   N.
```

Funs are a distinct type. The BIFs erlang:fun_info/1,2 can be used to retrieve information about a fun, and the BIF erlang:fun_to_list/1 returns a textual representation of a fun. The check_process_code/2 BIF returns true if the process contains funs that depend on the old version of a module.

> **Note:**
> In OTP R5 and earlier releases, funs were represented using tuples.

1.2.4 Variable Bindings Within a Fun

The scope rules for variables which occur in funs are as follows:

- All variables which occur in the head of a fun are assumed to be “fresh” variables.
- Variables which are defined before the fun, and which occur in function calls or guard tests within the fun, have the values they had outside the fun.
- No variables may be exported from a fun.

The following examples illustrate these rules:

```
print_list(File, List) ->
    {ok, Stream} = file:open(File, write),
    foreach(fun(X) -> io:format(Stream,"~p~n",[X]) end, List),
    file:close(Stream).
```

In the above example, the variable `X` which is defined in the head of the fun is a new variable. The value of the variable `Stream` which is used within within the fun gets its value from the `file:open` line.

Since any variable which occurs in the head of a fun is considered a new variable it would be equally valid to write:

```
print_list(File, List) ->
    {ok, Stream} = file:open(File, write),
    foreach(fun(File) ->
                io:format(Stream,"~p~n",[File])
            end, List),
    file:close(Stream).
```

In this example, `File` is used as the new variable instead of `X`. This is rather silly since code in the body of the fun cannot refer to the variable `File` which is defined outside the fun. Compiling this example will yield the diagnostic:

```
./FileName.erl:Line: Warning: variable 'File'
      shadowed in 'lambda head'
```

This reminds us that the variable `File` which is defined inside the fun collides with the variable `File` which is defined outside the fun.

The rules for importing variables into a fun has the consequence that certain pattern matching operations have to be moved into guard expressions and cannot be written in the head of the fun. For example, we might write the following code if we intend the first clause of `F` to be evaluated when the value of its argument is `Y`:

```
f(...) ->
    Y = ...
    map(fun(X) when X == Y ->
              ;
           (_) ->
              ...
        end, ...)
    ...
```

instead of

```
f(...) ->
    Y = ...
    map(fun(Y) ->
              ;
           (_) ->
             ...
        end, ...)
    ...
```

1.2.5 Funs and the Module Lists

The following examples show a dialogue with the Erlang shell. All the higher order functions discussed are exported from the module `lists`.

map

```
map(F, [H|T]) -> [F(H)|map(F, T)];
map(F, [])    -> [].
```

`map` takes a function of one argument and a list of terms. It returns the list obtained by applying the function to every argument in the list.

```
> Double = fun(X) -> 2 * X end.
#Fun<erl_eval.6.72228031>
> lists:map(Double, [1,2,3,4,5]).
[2,4,6,8,10]
```

When a new fun is defined in the shell, the value of the Fun is printed as `Fun#<erl_eval>`.

any

```
any(Pred, [H|T]) ->
    case Pred(H) of
        true  ->  true;
        false ->  any(Pred, T)
    end;
any(Pred, []) ->
    false.
```

`any` takes a predicate `P` of one argument and a list of terms. A predicate is a function which returns `true` or `false`. `any` is true if there is a term `X` in the list such that `P(X)` is `true`.

We define a predicate `Big(X)` which is `true` if its argument is greater that 10.

```
> Big =  fun(X) -> if X > 10 -> true; true -> false end end.
#Fun<erl_eval.6.72228031>
> lists:any(Big, [1,2,3,4]).
false
> lists:any(Big, [1,2,3,12,5]).
true
```

all

```
all(Pred, [H|T]) ->
    case Pred(H) of
        true  ->  all(Pred, T);
        false ->  false
    end;
all(Pred, []) ->
    true.
```

`all` has the same arguments as `any`. It is true if the predicate applied to all elements in the list is true.

```
> lists:all(Big, [1,2,3,4,12,6]).
false
> lists:all(Big, [12,13,14,15]).
true
```

foreach

```
foreach(F, [H|T]) ->
    F(H),
    foreach(F, T);
foreach(F, []) ->
    ok.
```

`foreach` takes a function of one argument and a list of terms. The function is applied to each argument in the list. `foreach` returns `ok`. It is used for its side-effect only.

```
> lists:foreach(fun(X) -> io:format("~w~n",[X]) end, [1,2,3,4]).
1
2
3
4
ok
```

foldl

```
foldl(F, Accu, [Hd|Tail]) ->
    foldl(F, F(Hd, Accu), Tail);
foldl(F, Accu, []) -> Accu.
```

`foldl` takes a function of two arguments, an accumulator and a list. The function is called with two arguments. The first argument is the successive elements in the list, the second argument is the accumulator. The function must return a new accumulator which is used the next time the function is called.

If we have a list of lists `L = ["I","like","Erlang"]`, then we can sum the lengths of all the strings in `L` as follows:

```
> L = ["I","like","Erlang"].
["I","like","Erlang"]
10> lists:foldl(fun(X, Sum) -> length(X) + Sum end, 0, L).
11
```

foldl works like a while loop in an imperative language:

```
L =  ["I","like","Erlang"],
Sum = 0,
while( L != []){
    Sum += length(head(L)),
    L = tail(L)
end
```

mapfoldl

```
mapfoldl(F, Accu0, [Hd|Tail]) ->
    {R,Accu1} = F(Hd, Accu0),
    {Rs,Accu2} = mapfoldl(F, Accu1, Tail),
    {[R|Rs], Accu2};
mapfoldl(F, Accu, []) -> {[], Accu}.
```

mapfoldl simultaneously maps and folds over a list. The following example shows how to change all letters in L to upper case and count them.

First upcase:

```
> Upcase =  fun(X) when $a =< X,  X =< $z -> X + $A - $a;
(X) -> X
end.
#Fun<erl_eval.6.72228031>
> Upcase_word =
fun(X) ->
lists:map(Upcase, X)
end.
#Fun<erl_eval.6.72228031>
> Upcase_word("Erlang").
"ERLANG"
> lists:map(Upcase_word, L).
["I","LIKE","ERLANG"]
```

Now we can do the fold and the map at the same time:

```
> lists:mapfoldl(fun(Word, Sum) ->
{Upcase_word(Word), Sum + length(Word)}
end, 0, L).
{["I","LIKE","ERLANG"],11}
```

filter

```
filter(F, [H|T]) ->
    case F(H) of
        true  -> [H|filter(F, T)];
        false -> filter(F, T)
    end;
filter(F, []) -> [].
```

`filter` takes a predicate of one argument and a list and returns all element in the list which satisfy the predicate.

```
> lists:filter(Big, [500,12,2,45,6,7]).
[500,12,45]
```

When we combine maps and filters we can write very succinct code. For example, suppose we want to define a set difference function. We want to define `diff(L1, L2)` to be the difference between the lists `L1` and `L2`. This is the list of all elements in `L1` which are not contained in `L2`. This code can be written as follows:

```
diff(L1, L2) ->
    filter(fun(X) -> not member(X, L2) end, L1).
```

The AND intersection of the list `L1` and `L2` is also easily defined:

```
intersection(L1,L2) -> filter(fun(X) -> member(X,L1) end, L2).
```

takewhile

```
takewhile(Pred, [H|T]) ->
    case Pred(H) of
        true  -> [H|takewhile(Pred, T)];
        false -> []
    end;
takewhile(Pred, []) ->
    [].
```

`takewhile(P, L)` takes elements `X` from a list `L` as long as the predicate `P(X)` is true.

```
> lists:takewhile(Big, [200,500,45,5,3,45,6]).
[200,500,45]
```

dropwhile

```
dropwhile(Pred, [H|T]) ->
    case Pred(H) of
        true  -> dropwhile(Pred, T);
        false -> [H|T]
    end;
dropwhile(Pred, []) ->
    [].
```

`dropwhile` is the complement of `takewhile`.

```
> lists:dropwhile(Big, [200,500,45,5,3,45,6]).
[5,3,45,6]
```

splitwith

```
splitwith(Pred, L) ->
    splitwith(Pred, L, []).

splitwith(Pred, [H|T], L) ->
    case Pred(H) of
        true  -> splitwith(Pred, T, [H|L]);
        false -> {reverse(L), [H|T]}
    end;
splitwith(Pred, [], L) ->
    {reverse(L), []}.
```

`splitwith(P, L)` splits the list `L` into the two sub-lists {`L1, L2`}, where `L = takewhile(P, L)` and `L2 = dropwhile(P, L)`.

```
> lists:splitwith(Big, [200,500,45,5,3,45,6]).
{[200,500,45],[5,3,45,6]}
```

1.2.6 Funs Which Return Funs

So far, this section has only described functions which take funs as arguments. It is also possible to write more powerful functions which themselves return funs. The following examples illustrate these type of functions.

Simple Higher Order Functions

`Adder(X)` is a function which, given `X`, returns a new function `G` such that `G(K)` returns `K + X`.

```
> Adder = fun(X) -> fun(Y) -> X + Y end end.
#Fun<erl_eval.6.72228031>
> Add6 = Adder(6).
#Fun<erl_eval.6.72228031>
> Add6(10).
16
```

Infinite Lists

The idea is to write something like:

```
-module(lazy).
-export([ints_from/1]).
ints_from(N) ->
    fun() ->
            [N|ints_from(N+1)]
    end.
```

Then we can proceed as follows:

```
> XX = lazy:ints_from(1).
#Fun<lazy.0.29874839>
> XX().
[1|#Fun<lazy.0.29874839>]
> hd(XX()).
1
> Y = tl(XX()).
#Fun<lazy.0.29874839>
> hd(Y()).
2
```

etc. - this is an example of “lazy embedding”.

Parsing

The following examples show parsers of the following type:

```
Parser(Toks) -> {ok, Tree, Toks1} | fail
```

`Toks` is the list of tokens to be parsed. A successful parse returns `{ok, Tree, Toks1}`, where `Tree` is a parse tree and `Toks1` is a tail of `Tree` which contains symbols encountered after the structure which was correctly parsed. Otherwise `fail` is returned.

The example which follows illustrates a simple, functional parser which parses the grammar:

```
(a | b) & (c | d)
```

The following code defines a function `pconst(X)` in the module `funparse`, which returns a fun which parses a list of tokens.

```
pconst(X) ->
    fun (T) ->
       case T of
           [X|T1] -> {ok, {const, X}, T1};
           _      -> fail
       end
    end.
```

This function can be used as follows:

```
> P1 = funparse:pconst(a).
#Fun<funparse.0.22674075>
> P1([a,b,c]).
{ok,{const,a},[b,c]}
> P1([x,y,z]).
fail
```

Next, we define the two higher order functions `pand` and `por` which combine primitive parsers to produce more complex parsers. Firstly `pand`:

```
pand(P1, P2) ->
    fun (T) ->
        case P1(T) of
            {ok, R1, T1} ->
                case P2(T1) of
                    {ok, R2, T2} ->
                        {ok, {'and', R1, R2}};
                    fail ->
                        fail
                end;
            fail ->
                fail
        end
    end.
```

Given a parser `P1` for grammar `G1`, and a parser `P2` for grammar `G2`, `pand(P1, P2)` returns a parser for the grammar which consists of sequences of tokens which satisfy `G1` followed by sequences of tokens which satisfy `G2`.

`por(P1, P2)` returns a parser for the language described by the grammar `G1` or `G2`.

```
por(P1, P2) ->
    fun (T) ->
        case P1(T) of
            {ok, R, T1} ->
                {ok, {'or',1,R}, T1};
            fail ->
                case P2(T) of
                    {ok, R1, T1} ->
                        {ok, {'or',2,R1}, T1};
                    fail ->
                        fail
                end
        end
    end.
```

The original problem was to parse the grammar `(a | b) & (c | d)`. The following code addresses this problem:

```
grammar() ->
    pand(
         por(pconst(a), pconst(b)),
         por(pconst(c), pconst(d))).
```

The following code adds a parser interface to the grammar:

```
parse(List) ->
    (grammar())(List).
```

We can test this parser as follows:

```
> funparse:parse([a,c]).
{ok,{'and',{'or',1,{const,a}},{'or',1,{const,c}}}}
> funparse:parse([a,d]).
{ok,{'and',{'or',1,{const,a}},{'or',2,{const,d}}}}
> funparse:parse([b,c]).
{ok,{'and',{'or',2,{const,b}},{'or',1,{const,c}}}}
> funparse:parse([b,d]).
{ok,{'and',{'or',2,{const,b}},{'or',2,{const,d}}}}
> funparse:parse([a,b]).
fail
```

1.3 List Comprehensions

1.3.1 Simple Examples

We start with a simple example:

```
> [X || X <- [1,2,a,3,4,b,5,6], X > 3].
[a,4,b,5,6]
```

This should be read as follows:

> The list of X such that X is taken from the list `[1,2,a,...]` and X is greater than 3.

The notation `X <- [1,2,a,...]` is a generator and the expression `X > 3` is a filter.
An additional filter can be added in order to restrict the result to integers:

```
> [X || X <- [1,2,a,3,4,b,5,6], integer(X), X > 3].
[4,5,6]
```

Generators can be combined. For example, the Cartesian product of two lists can be written as follows:

```
> [{X, Y} || X <- [1,2,3], Y <- [a,b]].
[{1,a},{1,b},{2,a},{2,b},{3,a},{3,b}]
```

1.3.2 Quick Sort

The well known quick sort routine can be written as follows:

```
sort([Pivot|T]) ->
    sort([ X || X <- T, X < Pivot]) ++
    [Pivot] ++
    sort([ X || X <- T, X >= Pivot]);
sort([]) -> [].
```

The expression `[X || X <- T, X < Pivot]` is the list of all elements in `T`, which are less than `Pivot`.

`[X || X <- T, X >= Pivot]` is the list of all elements in `T`, which are greater or equal to `Pivot`.

To sort a list, we isolate the first element in the list and split the list into two sub-lists. The first sub-list contains all elements which are smaller than the first element in the list, the second contains all elements which are greater than or equal to the first element in the list. We then sort the sub-lists and combine the results.

1.3.3 Permutations

The following example generates all permutations of the elements in a list:

```
perms([]) -> [[]];
perms(L)  -> [[H|T] || H <- L, T <- perms(L--[H])].
```

We take take `H` from `L` in all possible ways. The result is the set of all lists `[H|T]`, where `T` is the set of all possible permutations of `L` with `H` removed.

```
> perms([b,u,g]).
[[b,u,g],[b,g,u],[u,b,g],[u,g,b],[g,b,u],[g,u,b]]
```

1.3.4 Pythagorean Triplets

Pythagorean triplets are sets of integers `{A,B,C}` such that `A**2 + B**2 = C**2`.

The function `pyth(N)` generates a list of all integers `{A,B,C}` such that `A**2 + B**2 = C**2` and where the sum of the sides is equal to or less than `N`.

```
pyth(N) ->
    [ {A,B,C} ||
        A <- lists:seq(1,N),
        B <- lists:seq(1,N),
        C <- lists:seq(1,N),
        A+B+C =< N,
        A*A+B*B == C*C
    ].
```

```
> pyth(3).
[].
> pyth(11).
[].
> pyth(12).
[{3,4,5},{4,3,5}]
> pyth(50).
[{3,4,5},
 {4,3,5},
 {5,12,13},
 {6,8,10},
 {8,6,10},
 {8,15,17},
 {9,12,15},
 {12,5,13},
 {12,9,15},
 {12,16,20},
 {15,8,17},
 {16,12,20}]
```

The following code reduces the search space and is more efficient:

```
pyth1(N) ->
   [{A,B,C} ||
       A <- lists:seq(1,N-2),
       B <- lists:seq(A+1,N-1),
       C <- lists:seq(B+1,N),
       A+B+C =< N,
       A*A+B*B == C*C ].
```

1.3.5 Simplifications with List Comprehensions

As an example, list comprehensions can be used to simplify some of the functions in `lists.erl`:

```
append(L)   ->  [X || L1 <- L, X <- L1].
map(Fun, L) -> [Fun(X) || X <- L].
filter(Pred, L) -> [X || X <- L, Pred(X)].
```

1.3.6 Variable Bindings in List Comprehensions

The scope rules for variables which occur in list comprehensions are as follows:

- all variables which occur in a generator pattern are assumed to be “fresh” variables
- any variables which are defined before the list comprehension and which are used in filters have the values they had before the list comprehension
- no variables may be exported from a list comprehension.

As an example of these rules, suppose we want to write the function `select`, which selects certain elements from a list of tuples. We might write `select(X, L) -> [Y || {X, Y} <- L].` with the intention of extracting all tuples from `L` where the first item is `X`.

Compiling this yields the following diagnostic:

```
./FileName.erl:Line: Warning: variable 'X' shadowed in generate
```

This diagnostic warns us that the variable `X` in the pattern is not the same variable as the variable `X` which occurs in the function head.

Evaluating `select` yields the following result:

```
> select(b,[{a,1},{b,2},{c,3},{b,7}]).
[1,2,3,7]
```

This result is not what we wanted. To achieve the desired effect we must write `select` as follows:

```
select(X, L) ->  [Y || {X1, Y} <- L, X == X1].
```

The generator now contains unbound variables and the test has been moved into the filter. This now works as expected:

```
> select(b,[{a,1},{b,2},{c,3},{b,7}]).
[2,7]
```

One consequence of the rules for importing variables into a list comprehensions is that certain pattern matching operations have to be moved into the filters and cannot be written directly in the generators. To illustrate this, do not write as follows:

```
f(...) ->
    Y = ...
    [ Expression || PatternInvolving Y  <- Expr, ...]
    ...
```

Instead, write as follows:

```
f(...) ->
    Y = ...
    [ Expression || PatternInvolving Y1  <- Expr, Y == Y1, ...]
    ...
```

1.4 Bit Syntax

1.4.1 Introduction

In Erlang a Bin is used for constructing binaries and matching binary patterns. A Bin is written with the following syntax:

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

A Bin is a low-level sequence of bits or bytes. The purpose of a Bin is to be able to, from a high level, construct a binary,

```
Bin = <<E1, E2, ... En>>
```

in which case all elements must be bound, or to match a binary,

```
<<E1, E2, ... En>> = Bin
```

where `Bin` is bound, and where the elements are bound or unbound, as in any match.

In R12B, a Bin need not consist of a whole number of bytes.

A *bitstring* is a sequence of zero or more bits, where the number of bits doesn't need to be divisible by 8. If the number of bits is divisible by 8, the bitstring is also a binary.

Each element specifies a certain *segment* of the bitstring. A segment is a set of contiguous bits of the binary (not necessarily on a byte boundary). The first element specifies the initial segment, the second element specifies the following segment etc.

The following examples illustrate how binaries are constructed or matched, and how elements and tails are specified.

Examples

Example 1: A binary can be constructed from a set of constants or a string literal:

```
Bin11 = <<1, 17, 42>>,
Bin12 = <<"abc">>
```

yields binaries of size 3; `binary_to_list(Bin11)` evaluates to `[1, 17, 42]`, and `binary_to_list(Bin12)` evaluates to `[97, 98, 99]`.

Example 2: Similarly, a binary can be constructed from a set of bound variables:

```
A = 1, B = 17, C = 42,
Bin2 = <<A, B, C:16>>
```

yields a binary of size 4, and `binary_to_list(Bin2)` evaluates to `[1, 17, 00, 42]` too. Here we used a *size expression* for the variable `C` in order to specify a 16-bits segment of `Bin2`.

Example 3: A Bin can also be used for matching: if `D`, `E`, and `F` are unbound variables, and `Bin2` is bound as in the former example,

```
<<D:16, E, F/binary>> = Bin2
```

yields `D = 273`, `E = 00`, and F binds to a binary of size 1: `binary_to_list(F) = [42]`.

Example 4: The following is a more elaborate example of matching, where `Dgram` is bound to the consecutive bytes of an IP datagram of IP protocol version 4, and where we want to extract the header and the data of the datagram:

```
-define(IP_VERSION, 4).
-define(IP_MIN_HDR_LEN, 5).

DgramSize = byte_size(Dgram),
case Dgram of
    <<?IP_VERSION:4, HLen:4, SrvcType:8, TotLen:16,
      ID: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,
    ...
end.
```

Here the segment corresponding to the `Opts` variable has a *type modifier* specifying that `Opts` should bind to a binary. All other variables have the default type equal to unsigned integer.

An IP datagram header is of variable length, and its length - measured in the number of 32-bit words - is given in the segment corresponding to `HLen`, the minimum value of which is 5. It is the segment corresponding to `Opts` that is variable: if `HLen` is equal to 5, `Opts` will be an empty binary.

The tail variables `RestDgram` and `Data` bind to binaries, as all tail variables do. Both may bind to empty binaries.

If the first 4-bits segment of `Dgram` is not equal to 4, or if `HLen` is less than 5, or if the size of `Dgram` is less than `4*HLen`, the match of `Dgram` fails.

1.4.2 A Lexical Note

Note that "`B=<<1>>`" will be interpreted as "`B =< <1>>`", which is a syntax error. The correct way to write the expression is "`B = <<1>>`".

1.4.3 Segments

Each segment has the following general syntax:

`Value:Size/TypeSpecifierList`

Both the `Size` and the `TypeSpecifier` or both may be omitted; thus the following variations are allowed:

`Value`

`Value:Size`

`Value/TypeSpecifierList`

Default values will be used for missing specifications. The default values are described in the section Defaults [page 22].

Used in binary construction, the `Value` part is any expression. Used in binary matching, the `Value` part must be a literal or variable. You can read more about the `Value` part in the section about constructing binaries and matching binaries.

The `Size` part of the segment multiplied by the unit in the `TypeSpecifierList` (described below) gives the number of bits for the segment. In construction, `Size` is any expression that evaluates to an integer. In matching, `Size` must be a constant expression or a variable.

The `TypeSpecifierList` is a list of type specifiers separated by hyphens.

Type The type can be `integer`, `float`, or `binary`.

Signedness The signedness specification can be either `signed` or `unsigned`. Note that signedness only matters for matching.

Endianness The endianness specification can be either `big`, `little`, or `native`. Native-endian means that the endian will be resolved at load time to be either big-endian or little-endian, depending on what is "native" for the CPU that the Erlang machine is run on.

Unit The unit size is given as `unit:IntegerLiteral`. The allowed range is 1-256. It will be multiplied by the `Size` specifier to give the effective size of the segment. In R12B, the unit size specifies the alignment for binary segments without size (examples will follow).

Example:

```
X:4/little-signed-integer-unit:8
```

This element has a total size of 4*8 = 32 bits, and it contains a signed integer in little-endian order.

1.4.4 Defaults

The default type for a segment is integer. The default type does not depend on the value, even if the value is a literal. For instance, the default type in '<<3.14>>' is integer, not float.

The default `Size` depends on the type. For integer it is 8. For float it is 64. For binary it is all of the binary. In matching, this default value is only valid for the very last element. All other binary elements in matching must have a size specification.

The default unit depends on the the type. For `integer`, `float`, and `bitstring` it is 1. For binary it is 8.

The default signedness is `unsigned`.

The default endianness is `big`.

1.4.5 Constructing Binaries and Bitstrings

This section describes the rules for constructing binaries using the bit syntax. Unlike when constructing lists or tuples, the construction of a binary can fail with a `badarg` exception.

There can be zero or more segments in a binary to be constructed. The expression '<<>>' constructs a zero length binary.

Each segment in a binary can consist of zero or more bits. There are no alignment rules for individual segments of type `integer` and `float`. For binaries and bitstrings without size, the unit specifies the alignment. Since the default alignment for the `binary` type is 8, the size of a binary segment must be a multiple of 8 bits (i.e. only whole bytes). Example:

```
<<Bin/binary,Bitstring/bitstring>>
```

The variable `Bin` must contain a whole number of bytes, because the `binary` type defaults to `unit:8`. A `badarg` exception will be generated if `Bin` would consist of (for instance) 17 bits.

On the other hand, the variable `Bitstring` may consist of any number of bits, for instance 0, 1, 8, 11, 17, 42, and so on, because the default `unit` for bitstrings is 1.

> **Warning:**
> For clarity, it is recommended not to change the unit size for binaries, but to use `binary` when you need byte alignment, and `bitstring` when you need bit alignment.

The following example

```
<<X:1,Y:6>>
```

will successfully construct a bitstring of 7 bits. (Provided that all of X and Y are integers.)

As noted earlier, segments have the following general syntax:

`Value:Size/TypeSpecifierList`

When constructing binaries, `Value` and `Size` can be any Erlang expression. However, for syntactical reasons, both `Value` and `Size` must be enclosed in parenthesis if the expression consists of anything more than a single literal or variable. The following gives a compiler syntax error:

```
<<X+1:8>>
```

This expression must be rewritten to

```
<<(X+1):8>>
```

in order to be accepted by the compiler.

Including Literal Strings

As syntactic sugar, an literal string may be written instead of a element.

```
<<"hello">>
```

which is syntactic sugar for

```
<<$h,$e,$l,$l,$o>>
```

1.4.6 Matching Binaries

This section describes the rules for matching binaries using the bit syntax.

There can be zero or more segments in a binary pattern. A binary pattern can occur in every place patterns are allowed, also inside other patterns. Binary patterns cannot be nested.

The pattern '<<>>' matches a zero length binary.

Each segment in a binary can consist of zero or more bits.

A segment of type `binary` must have a size evenly divisible by 8 (or divisible by the unit size, if the unit size has been changed).

A segment of type `bitstring` has no restrictions on the size.

As noted earlier, segments have the following general syntax:

`Value:Size/TypeSpecifierList`

When matching `Value` value must be either a variable or an integer or floating point literal. Expressions are not allowed.

`Size` must be an integer literal, or a previously bound variable. Note that the following is not allowed:

```
foo(N, <<X:N,T/binary>>) ->
   {X,T}.
```

The two occurrences of `N` are not related. The compiler will complain that the `N` in the size field is unbound.

The correct way to write this example is like this:

```
foo(N, Bin) ->
   <<X:N,T/binary>> = Bin,
   {X,T}.
```

Getting the Rest of the Binary or Bitstring

To match out the rest of a binary, specify a binary field without size:

```
foo(<<A:8,Rest/binary>>) ->
```

The size of the tail must be evenly divisible by 8.

To match out the rest of a bitstring, specify a field without size:

```
foo(<<A:8,Rest/bitstring>>) ->
```

There is no restriction on the number of bits in the tail.

1.4.7 Appending to a Binary

In R12B, the following function for creating a binary out of a list of triples of integers is now efficient:

```
triples_to_bin(T) ->
    triples_to_bin(T, <<>>).

triples_to_bin([{X,Y,Z} | T], Acc) ->
    triples_to_bin(T, <<Acc/binary,X:32,Y:32,Z:32>>);   % inefficient before R12B
triples_to_bin([], Acc) ->
    Acc.
```

In previous releases, this function was highly inefficient, because the binary constructed so far (`Acc`) was copied in each recursion step. That is no longer the case. See the Efficiency Guide for more information.