Erlang Extensions Since 4.4

version 5.0

Typeset in LaTeX from SGML source using the DOCBUILDER 3.0 Document System.

Contents

Chapter 1

Erlang Extensions Since 4.4

This chapter describes extensions made to the Erlang language since version 4.4 (where nothing is said to the contrary, an extension was added in version 4.4). The chapter contains the following sections:

- Records
- Functional Objects (Funs)
- List Comprehensions
- Macros
- File inclusion
- Bit syntax
- Miscellaneous

1.1 Records

A record is a data structure intended for storing a fixed number of related data items. It is similar to a `struct` in C, or a `record` in Pascal.

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,
...
```

In the following sections we describe the different operations which can be performed on records:

Defining a Record

A record is defined with the following syntax:

```
-record(RecordName, {Field1 [= DefaultValue1],
                     Field2 [= DefaultValue2],
                     ...,
                     FieldN [= DefaultValueN]}).
```

The record name and field names must be atoms. The optional default values, which are terms, are used if no value is supplied for a field when a new instance of the record is created. If the default value is not supplied, then the atom `undefined` is assumed.

For example, in the following record definition, the address field is `undefined`.

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

This definition of a person will be used in many of the examples which follow.

Including a Record Definition

If the record is used in several modules, its definition should be placed in a `.hrl` header file. Each module which uses the record definition should have a `-include(FileName).` statement. For example:

```
-include("my_data_structures.hrl").
```

> Note:
> The definition of the record must come before it is used.

Creating a Record

A new record is created with the following syntax:

```
#RecordName{Field1=Expr1,
            ...,
            FieldM=ExprM}.
```

If any of the fields is omitted, then the default value supplied in the record definition is used. For example:

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

Selectors

The following syntax is used to select an individual field from a record:

```
Variable#RecordName.Field
```

> Note:
> The values contained in record names and fields must be constants, not variables.

> Note:
> For the purposes of illustration, we will demonstrate the use of records using an imaginary dialogue with the Erlang shell. Currently the Erlang evaluator does not support records so you may not be able to reproduce this dialogue.

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

> Note:
> Selectors for records are allowed in guards.

Updating a Record

The following syntax is used to create a new copy of the record with some of the fields changed. Only the fields to be changed need to be referred to, all other fields retain their old values.

```
OldVariable#RecordName{Field1 = NewValue1,
                                ...,
                       FieldM = NewValueM}
```

For example:

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

Type Testing

The following guard test is used to test the type of a record:

```
record(Variable, RecordName)
```

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

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

> Note:
> This test checks that `P` is a tuple of arity `N + 1`, where `N` is the number of fields in the record, and the first element in the tuple is the atom `person`.

Pattern Matching

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

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

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

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

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

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

Internal Representation of Records

It is often desirable to write generic functions which will work on any record, not just a record of a particular type. For this reason, records are represented internally as tuples and the ordering of the fields in the tuple is strictly defined.

For example, the record `-record(person, {name, phone, address}).` is represented internally by the tuple `{person, X, Y, Z}`.

The arity of the tuple is one more than the number of fields in the tuple. The first element of the tuple is the name of the record, and the elements of the tuple are the fields in the record. The variables `X`, `Y` and `Z` will store the data contained in the record fields.

The following two functions determine the indices in the tuple which refer to the named fields in the record:

- `record_info(fields, Rec) -> [Names]`. This function returns the names of the fields in the record `Rec`. For example, `record_info(fields, person)` evaluates to `[name, address, phone]`.
- `record_info(size, Rec) -> Size`. This function returns the size of the record `Rec` when represented as a tuple, which is one more than the number of fields. For example, `record_info(size, person)` returns 4.

In addition, `#Rec.Name` returns the index in the tuple representation of `Name` of the record `Rec`.

> Note:
> `Name` must be an atom.

For example, the following test function `test()` might return the result shown:

```
test() ->
    {record_info(fields, person),
     record_info(size, person),
     #person.name}.

> Mod:test().
{[name,address,phone],4,2}
```

The order in which records map onto tuples is implementation dependent.

> Note:
> `record_info` is a pseudo-function which cannot be exported from the module where it occurs.

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 Programming with Funs

This section introduces functional objects (Funs), which are a new data type introduced in Erlang 4.4. Functions which takes Funs as arguments, or which return Funs are called higher order functions.

- Funs can be passed as arguments to other functions, just like lists or tuples
- functions can be written which return Funs, just like any other data object.

Higher Order Functions

Funs encourages us to encapsulate common patterns of design into functional forms called higher order functions. These functions not only shortens programs, but also produce clearer programs because the intended meaning of the program is explicitly rather than implicitly stated.

The concepts of higher order functions and procedural abstraction are introduced with two brief examples.

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.

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_on_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_on_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.

Advantages of Higher Order Functions

Programming with higher order functions, such as `map` and `foreach`, has a number of advantages:

- It is much easier to understand the program and the intention of the programmer is clearly expressed in the code. The statement `foreach(fun(X) ->` clearly indicates that the intention of this program is to do something to each element in the list `L`. We also know that the function which is passed as the first argument of `foreach` takes one argument `X`, which will be successively bound to each of the elements in `L`.
- Functions which take Funs as arguments are much easier to re-use than other functions.

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 `function/1` in a guard. Example:

```
f(F, Args) when function(F) ->
   apply(F, Args);
f(N, _) when integer(N) ->
   N.
```

Note that Funs are currently represented internally using tuples, and therefore any code that needs to handle both tuples and Funs must use the `function/1` test before using the `tuple/1` test.

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, ...)
    ...
```

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.

```
1> Double = fun(X) -> 2 * X end.
#Fun<erl_eval>
2> 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.

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

```
6> lists:all(Big, [1,2,3,4,12,6]).
false
7> 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.

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

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:

```
9> 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:

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

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

```
14> lists:mapfoldl(fun(Word, Sum) ->
14>     {Upcase_word(Word), Sum + length(Word)}
14>                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.

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

When we combine maps and filters we can write very succinct and obviously correct 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.

```
16> 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`.

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

splitlist

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

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

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

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

first

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

`first` returns `{true, R}`, where `R` is the first element in a list satisfying a predicate or `false`:

```
19> lists:first(Big, [1,2,45,6,123]).
{true,45}
20> lists:first(Big, [1,2,4,5]).
false
```

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`.

```
21> Adder = fun(X) -> fun(Y) -> X + Y end end.
#Fun<erl_eval>
22> Add6 = Adder(6).
#Fun<erl_eval>
23> 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:

```
24> XX = lazy:ints_from(1).
#Fun<lazy>
25> XX().
[1|#Fun<lazy>]
26> hd(XX()).
1
27> Y = tl(XX()).
#Fun<lazy>
28> 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:

```
29> P1 = funparse:pconst(a).
#Fun<hof>
30> P1([a,b,c]).
{ok,{const,a},[b,c]}
31> 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:

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

1.3 List Comprehensions

List comprehensions are a feature of many modern functional programming languages. Subject to certain rules, they provide a succinct notation for generating elements in a list.

List comprehensions are analogous to set comprehensions in Zermelo-Frankel set theory and are called ZF expressions in Miranda and SASL. They are analogous to the `setof` and `findall` predicates in Prolog.

List comprehensions are written with the following syntax:

```
[Expression || Qualifier1, Qualifier2, ...]
```

`Expression` is an arbitrary expression, and each `Qualifier` is either a generator or a filter.

- A *generator* written as `Pattern <- ListExpr`. `ListExpr` must be an expression which evaluates to a list of terms.
- A *filter* is either a predicate or a boolean expression. A predicate is a function which returns `true` or `false`.

Examples of List Comprehensions

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}]
```

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.

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]]
```

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 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),
        B <- lists:seq(1,N-A+1),
        C <- lists:seq(1,N-A-B+2),
        A+B+C =< N,
        A*A+B*B == C*C ].
```

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)].
```

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 Macros

Macros in Erlang are written with the following syntax:

```
-define(Const, Replacement).
-define(Fun(Var1, Var2,.., Var), Replacement).
```

Macros are expanded when the syntax `?MacroName` is encountered.

Consider the macro definition:

```
-define(timeout, 200).
```

The expression `?timeout`, which can occur anywhere in the code which follows the macro definition, will be replaced by `200`.

Macros with arguments are written as follows:

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

This type of macro can be used as follows:

```
bar(X) ->
    ?macro1(a, b),
    ?macro1(X, 123)
```

This expands to:

```
bar(X) ->
    {a,a,b,b},
    {a,X,b,123}.
```

Macros and Tokens

Macro expansion works at a token level. We might define a macro as follows:

```
-define(macro2(X, Y), {a,X,b,Y).
```

The replacement value of the macro is not a valid Erlang term because the closing right curly bracket is missing. `macro2` expands into a sequence of tokens `{`, `a`, `X` which are then pasted into the place where the macro is used.

We might use this macro as follows:

```
bar() ->
    ?macro2(x,y)}.
```

This will expand into the valid sequence of tokens {a,x,y,b} before being parsed and compiled.

> Note:
> It is good programming practise to ensure that the replacement text of a macro is a valid Erlang syntactic form.

Pre-Defined Macros

The following macros are pre-defined:

`?MODULE`. This macro returns the name of the current module.

`?MODULE_STRING`. This macro returns the name of the current module, as a string.

`?FILE`. This macro returns the current file name.

`?LINE`. This macro returns the current line number.

`?MACHINE`. This macro returns the current machine name, `'BEAM'`,

Stringifying Macro Arguments

The construction `??Arg` for an argument to a macro expands to a string containing the tokens of the argument, similar to the `#arg` stringifying construction in C. This was added in Erlang 5.0 (OTP R7A).

Example:

```
-define(TESTCALL(Call), io:format("Call ~s: ~w~n", [??Call, Call])).

?TESTCALL(myfunction(1,2)),
?TESTCALL(you:function(2,1)).
```

results in

```
io:format("Call ~s: ~w~n",["myfunction ( 1 , 2 )",m:myfunction(1,2)]),
io:format("Call ~s: ~w~n",["you : function ( 2 , 1 )",you:function(2,1)]).
```

Flow Control in Macros

The following macro directives are supplied:

-undef(Macro). Causes the macro to behave as if it had never been defined.

-ifdef(Macro). Do the following lines if `Macro` is defined.

-ifndef(Macro). Do the following lines if `Macro` is not defined.

-else. "else" macro

-endif. "endif" macro.

The conditional macros must be properly nested. They are usually grouped as follows:

```
-ifdef(debug)
-define(....)
-else
-define(...)
-endif
```

The following example illustrates this grouping:

```
-define(debug, true).
-ifdef(debug).
-define(trace(Str, X), io:format("Mod:~w line:~w ~p ~p~n",
                                 [?MODULE,?LINE, Str, X])).
-else.
-define(trace(X, Y), true).
-endif.
```

Given these definitions, the expression `?trace("X=", X).` in line 10 of the module `foo` expands to:

```
io:format("Mod:~w line:~w ~p ~p~n",[foo,100,"X=",[X]]),
```

If we remove the `-define(debug, true).` line, then the same expression expands to `true`.

A Macro Expansion Utility

The following code can be used to expand a macro and display the result:

```
-module(mexpand).
-export([file/1]).
-import(lists, [foreach/2]).
file(File) ->
  case epp:parse_file(File ++ ".erl", [],[]) of
    {ok, L} ->
      {ok, Stream} = file:open(File ++ ".out", write),
      foreach(fun(X) ->
                io:format(Stream,"~s~n",[erl_pp:form(X)])
              end, L),
      file:close(Stream)
  end.
```

Alternatively, we can compile the file with the `'P'` option. `compile:file(File, ['P'])` produces a list file `File.P`, in which the result of any macro expansions can be seen.

1.5 Includes

There are two directives which can be used in an Erlang source file to cause the compiler to temporarily read input from another source. They are typically used to provide macro definitions and record definitions from header files. It is recommended to use the file name extension `".hrl"` for files which are meant to be included (the 'h' can be read as "header").

The -include directive

The compiler searches for the specified header file in each directory in the compiler's include path. The first file found is included. Example:

```
-include("my_records.hrl").
```

The current directory is implicitly a member of the compiler's include path.

The -include_lib directive

The `-include_lib` directive instructs the compiler/preprocessor to look for a header file in an application directory. The first part of the specified pathname) is taken as the name of an application, and the current version of that application will be used. Example:

```
-include_lib("mnesia/include/mnemosyne.hrl").
```

This instructs the compiler/preprocessor to look for the directory where the application called `mnesia` is installed and then looks in the subdirectory `include` for the file `mnemosyne.hrl`.

The preprocessor first looks in the ordinary preprocessor search path to allow explicit overloading of the header files.

1.6 The bit syntax

This section describes the “bit syntax” which was added to the Erlang language in release 5.0 (R7). Compared to the original bit syntax prototype by Claes Wikström and Tony Rogvall (presented on the Erlang User’s Conference 1999), this implementation differs primarily in the following respects,

1. the character pairs ’<<’ and ’>>’ are used to delimit a binary patterns and constructor (not ’<’ and ’>’ as in the prototype),
2. the tail syntax (’|Variable’) has been eliminated,
3. all size expressions must be bound,
4. a type `unit:U` has been added,
5. lists and tuples cannot be generated
6. there are no paddings whatsoever.

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 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.

Each element specifies a certain *segment* of the binary. A segment is 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 = 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.

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>>".

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" below.

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 sections 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` or `little`.

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.

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.

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` and `float` it is 1. For `binary` it is 8.

The default signedness is `unsigned`.

The default endianness is `big`.

Constructing binaries

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, but the total number of bits in all segments must be evenly divisible by 8, or in other words, the resulting binary must consist of a whole number of bytes. An `badarg` exception will be thrown if the resulting binary is not byte-aligned. Example:

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

The total number of bits is 7, which is not evenly divisible by 8; thus, there will be `badarg` exception (and a compiler warning as well). The following example

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

will successfully construct a binary of 8 bits, or one byte. (Provided that all of X, Y and Z 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>>
```

Matching binaries

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

There can be zero or more segments in a binary 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.

This means that the following head will never match:

```
foo(<<X:7/binary,Y:1/binary>>) ->
```

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

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

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

As always, the size of the tail must be evenly divisible by 8.

Traps and pitfalls

Assume that we need a function that creates a binary out of a list of triples of integers. A first (inefficient) version of such a function could look like this:

```
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
triples_to_bin([], Acc) ->
    Acc.
```

The reason for the inefficiency of this function is that for each triple, the binary constructed so far (`Acc`) is copied. (Note: The original bit syntax prototype avoided the copy operation by using segmented binaries, which are not implemented in R7.)

The efficient way to write this function in R7 is:

```
triples_to_bin(T) ->
    triples_to_bin(T, []).

triples_to_bin([{X,Y,Z} | T], Acc) ->
    triples_to_bin(T, [<<X:32, Y:32, Z:32>> | Acc]);
triples_to_bin([], Acc) ->
    list_to_binary(lists:reverse(Acc)).
```

Note that `list_to_binary/1` handles deep lists of binaries and small integers. (This fact was previously undocumented.)

1.7 Miscellaneous

In this chapter, a number of miscellaneous features of Erlang are described.

Token Syntax

In Erlang 4.8 (OTP R5A) the syntax of Erlang tokens have been extended to allow the use of the full ISO-8859-1 (Latin-1) character set. This is noticeable in the following ways:

- All the Latin-1 printable characters can be used and are shown without the escape backslash convention.
- Atoms and variables can use all Latin-1 letters.

The new characters from Latin-1 have the following classifications in Erlang:

Octal	*Decimal*		*Class*
200 - 237	128 - 159		Control characters
240 - 277	160 - 191	- ¿	Punctuation characters
300 - 326	192 - 214	À - Ö	Uppercase letters
327	215	×	Punctuation character
330 - 336	216 - 222	Ø - Dh	Uppercase letters
337 - 366	223 - 246	ß - ö	Lowercase letters
367	247	÷	Punctuation character
370 - 377	248 - 255	ø - ÿ	Lowercase letters

Table 1.1: Character classes

String concatenation

Two adjacent string literals are concatenated into one. This is done already at compile-time, and doesn't incur any runtime overhead. Example:

```
"string" "42"
```

is equivalent to

```
"string42"
```

This feature is convenient in at least two situations:

- when one of the strings is the result of a macro expansion;
- when a string is very long, and would otherwise either have to wrap, making the source code harder to read, or force the use of some runtime append operation.

The ++ list concatenation operator

Since list concatenation is a very common operation, it is convenient to have a terse way of expressing it. The ++ operator appends its second argument to its first. Example:

```
X = [1,2,3],
Y = [4,5],
X ++ Y.
```

results in `[1,2,3,4,5]`.

The ++ operator has precedence between the binary '+' operator and the comparison operators.

The – list subtraction operator

The – operator produces a list which is a copy of the first argument, subjected to the following procedure: for each element in the second argument, its first occurrence in the first argument is removed.

```
X = [1,2,3,2,1,2],
Y = [2,1,2],
X -- Y.
```

results in `[3,1,2]`.

The – operator has precedence between the binary '+' operator and the comparison operators.

Bitwise operator bnot

Apart from the binary bitwise operators `band`, `bor` and `bxor`, there is a unary operator `bnot` with the same precedence as the other unary operators + and -, i.e., higher than the binary operators. Example:

```
bnot 7.
```

returns -8.

Logical operators

The atoms `true` and `false` are usually used for representing Boolean values. With the binary operators `and`, `or` and `xor`, and the unary operator `not`, Boolean values can be combined. Example:

```
M1 = lists:member(A, List1),
M2 = lists:member(A, List2),
M1 and M2.
```

Note that the operators are strict, i.e., they always evaluate both their arguments.

`not` has the same priority as the other unary operators. The binary logical operators have precedence between the = operator and the comparison operators, the `and` operator having higher precedence than `or` and `xor`.

Match operator = in patterns

This extension was added in Erlang 4.8 (OTP R5A).

The = operator is also called the ‘match’ operator. The match operator can now be used in a pattern, so that `P1 = P2` is a valid pattern, where both `P1` and `P2` are patterns. This compound pattern when matched against a term causes the term to be matched against both `P1` and `P2`.

One use for this construction is to avoid reconstructing a term which was part of an argument to a function. Example:

```
f({'+',X,Y}=T) -> {X+Y,T}.
```

It also makes it possible to rewrite the construction

```
f(X) when X == #rec{x=1, y=a} -> ...
```

as

```
f(#rec{x=1, y=a} = X) -> ...
```

In the absence of optimization for the former case, the latter case is more efficient.

Literal string prefix in patterns

This extension was added in Erlang 4.8 (OTP R5A).

A new construction is allowed in patterns, namely a literal string as the first operand of the ++ operator. Example:

```
f("prefix" ++ L) -> ...
```

This is syntactic sugar for the equivalent, but harder to read

```
f([$p,$r,$e,$f,$i,$x | L]) -> ...
```

Disjunctions in guards

This extension was added in Erlang 4.9 (OTP R6A).

A new construction is allowed in guards, the disjunction operator ';'. The construction is syntactic sugar which removes the bother of writing the same body after several guards.

```
f(X) when xxx ; yyy ; zzz ->
        pop(X).
```

This is syntactic sugar for the equivalent

```
f(X) when xxx ->
        pop(X);
f(X) when yyy ->
        pop(X);
f(X) when zzz ->
        pop(X).
```

The abstract format has been changed accordingly to contain a list of (conjunctive) guards where there was previously only one guard.

Expressions in patterns

This extension was added in Erlang 5.0 (OTP R7A).

An arithmetic expression can be used within a pattern, if it uses only numeric or bitwise operators, and if its value can be evaluated to a constant at compile-time. This is especially useful when the expression is defined by a macro.

Example:

```
case X of
       {1+2, T} -> T
end.
```

List of Tables