Industrial Functional Programming [1]

Melinda Tóth, István Bozó

Dept. Programming Languages and Compilers
Eötvös Loránd University, Budapest, Hungary

[1] Supported by TÁMOP-4.1.2.A/1-11/1-2011-0052

Contents

1. Macros
2. Binaries
3. I/O operations

Macros

- Have to be defined before the first use
- Token level expansion
- Constant macro definition:
  `-define(Name, Replacement)`
- Usage: `?Name`
- Parametrised macro definition:
  `-define(Name(Var1, ..., VarN), Replacement)`
- Usage: `?Name(Var1, ..., VarN)`

Macros

- Stringifying macro arguments: `??Var`
- Predefined macros:
  `?MODULE, ?MODULE_STRING, ?FILE, ?LINE, ?MACHINE`
- Often put into headers:
  `-include("something.hrl")`

Conditional Macros

- `-undef(Flag).`
- `-ifdef(Flag).`
- `-ifndef(Flag).`
- `-else.`
- `-endif.`
- `c(Module, [{d, debug}]), c(Module, [{u, debug}]).`

Macros

```
-define(atom, ok).
-define(atom(), ok).
-define(atom(P), list_to_atom(P)).

f() ->
    {?atom, ?atom(), ?atom("String")}.
```

Crosscutting Macros

```
-define(name, f).
-define(par, ()).
-define(arrowbody, -> ok).

?name?par?arrowbody.
```

Conditional Macros

```
-ifdef(debug).
f() -> ok.
-else.
f() -> nok.
-endif.

c(mod, {d, debug}).
```

Bitstring and binary

- `<< Segment1, ...., SegmentN>>`
- Segment: `Data, Data:Sizerrr, Data:TypeSpecifier, Data:Size/TypeSpec`
- TypeSpec:

```
{integer, float, binary,
 bytes, bitstrings, bits} --
{utf8, utf16, utf32} --
{signed, unsigned} --
{big, little, native}
```

- Example: `<<X/integer-signed-little>> = <<Y>>`
- pattern matching: `<<Var1:4/bits, Remaining>>`

Binary Comprehensions

```
<< <<X:3>> || X <- [1, 2, 3, 4, 5, 6, 7]>>

<< <<X:8>> || <<X:3>> <= <<41, 203, 23:5>> >>

[    X     || <<X:3>> <= <<41, 203, 23:5>> ]

[  <<X>>   || <<X:3>> <= <<41, 203, 23:5>> ]
```

Input and Output

- io module
- Reading lines: `get_line("> ")`
- Reading chracters: `get_chars("> ", 2)`
- Reading terms: `read("ok, then »")`
- Writing terms: `write/1`
- Printing out values:
  `format(FormatString, [Values])`

Formatting

- `~c` – ASCII characters
- `~f` – float number with precision of six digits
- `~e` – float number with precision of six digits
- `~w` – Erlang term with the standard syntax
- `~p` – Erlang term ('pretty printed')
- `~B` – Number (default decimal base)
- `~W` and `~P` – similar to `~w` and `~p`, but takes an extra argument, the maximum depth of term printing

File handling

- `file` - open, close, read, write, list dirs
- `filename` - handling file names (platform independent)
- `filelib` - extension to the module `file`
- `io` - operations on the contents of the file, adding formated text to the opened file
- `{ok, Dev} = file:open(File, [Mode]), file:close(Dev)`
- Mode: `read`, `write`, `append`, `exclusive`, `binary`, etc.

File handling

Reading Erlang terms:

- `file:consult("file")`
- `io:read(Dev, Promt)`
- `io:read(Dev, Prompt, StartLine)`

Reading lines:

- `io:get_line(Dev, Prompt)`
- `io:fread(FileDescr, Prompt, FormatString)`
- `~d, ~u, ~-, ~f, ~#, ~s, ~a, ~c, ~l,` whitespaces, `~~`

Reading binaries:

- `file:read_file("file")`
- `io:pread(Dev, Start, LenB), file:close(Dev)`

File handling

Writing Erlang terms:

- `io:format(Dev, FormatString, DataList)`
- `~n`, `~s`, `~p`, `~w`, whitespaces `~10.2s`

Writing bytes to a file:

- `file:write(Dev, Bytes)`
- `file:write_file(Filename, Bytes)` – creates the file, if necessary
- `file:pwrite(Dev, [{Loc, Bytes}])`, `file:pwrite(Dev, Loc, Bytes)`
- Loc: `bof`, `eof`, `cur`

On the Next Lecture ...

- Concurrency/pararellism
- Processes
- Message sending and receiving