lfe_gen(3)

Robert Virding

2008-2016

NAME

lfe_gen - Lisp Flavoured Erlang (LFE) dynamic code generator

SYNOPSIS

This module provides an experimental interface for dynamically generating modules.

DATA TYPES

sexpr()

An LFE s-expression, a list structure.

EXPORTS

new_module(Name) -> ModDef.

add_exports([[Name,Arity]], ModDef) -> ModDef.

add_imports([from,Module|Functions], ModDef) -> ModDef.

add_imports([rename,Module|Renames], ModDef) -> ModDef.

add_attribute(Attribute, ModDef) -> ModDef.

add_form(Form, ModDef) -> ModDef.

build_mod(ModDef) -> Forms.

compile_mod(Mod) -> CompRet

where

```
CompRet = BinRet | ErrRet
BinRet = {ok,ModuleName,Binary,Warnings}
ErrRet = {error,Errors,Warnings}
```

These functions are used to incrementally create a module which can at the end be compiled by `compile_mod/1`. The various components have the same formats as they do when defining a module in a file. A simple module which defines one function `a/0` could be defined with:

```
M0 = lfe_gen:new_module(foo),
M1 = lfe_gen:add_exports([[a,0]], M0),
M2 = lfe_gen:add_form([defun,a,[],[quote,yes]], M1),
lfe_gen:compile_mod(M2)
```

EXAMPLE

In this example we build a module of parameters where each parameter has a number of features which each have a value. We will create one function for each parameter and the feature is the functions argument. The value for each feature is returned.

We are creating code equivalent to:

```
-module(Name).
-export([<param1>/1,...]).

<param1>(Feature) ->
    case Feature of
        <feature1> -> <value1>;
        ...
        _ -> erlang:error({unknown_feature,<param1>,Feature)
    end.
...
```

but generating it and compiling it directly in memory without generating a text file. We assume that we have collected the data and have it in the form:

```
Params = [{Parameter,[{Feature,Value}]}]
```

The equivalent LFE code which we will be generating is:

```
(defmodule Name
  (export (<param1> 1) (<param2> 1) ... ))

(defun <param1> (f)
  (case f
    ('<feature1> '<value1>)
    ...
    (f (: erlang error (tuple 'unknown_feature '<param1> f)))))
...
```

The following code builds and compiles a module from the parameter data:

```
make_module(Name, Params) ->
    Mod0 = lfe_gen:new_module(Name),
    Exps = lists:map(fun ({Param,_}) -> [Param,1] end, Params),
    Mod1 = lfe_gen:add_exports(Exps, Mod0),
    Mod2 = make_funcs(Params, Mod1),
    lfe_gen:compile_mod(Mod2).

make_funcs([{Param,Fs}|Ps], Mod) ->
    %% Define catch-all which generates more explicit exit value.
    CatchAll = [f,[':',erlang,error,
                   [tuple,unknown_feature,[quote,Param],f]]],
    %% Build case clauses
    Fold = fun ({Feature,Value}, Cls) ->
                   [[[quote,Feature],[quote,Value]]|Cls]
           end
    Cls = lists:foldr(Fold, [CatchAll], Params),
    %% Build function.
    Func = [defun,Param,[f],['case',f,Cls]],
    make_funcs(Ps, lfe_gen:add_form(Func, Mod));
make_funcs([], Mod) -> Mod.                     %All done
```

This module could be generated and then be loaded into the system by doing:

```
{ok,ModuleName,Binary,Warnings} = make_module(Name, Params),
code:load_binary(ModuleName, "nofile", Binary)
```

The second argument to `code:load_binary/3`, here `"nofile"`, is irrelevant in this case.

ERROR INFORMATION

The ErrorInfo mentioned above is the standard ErrorInfo structure which is returned from all IO modules. It has the following format:

{ErrorLine,Module,ErrorDescriptor}

A string describing the error is obtained with the following call:

```
apply(Module, format_error, ErrorDescriptor)
```

SEE ALSO

lfe_comp(3), **lfe_macro(3)**