lfe_io(3)

Robert Virding

2008-2024

NAME

lfe_io - Lisp Flavoured Erlang (LFE) io functions

SYNOPSIS

This module provides a standard set of io functions for LFE. In the following description, many functions have an optional parameter IoDevice. If included, it must be the pid of a process which handles the IO protocols such as the IoDevice returned by file:open/2.

Two functions in this module are used to generateœ aesthetically attractive representations of abstract forms, which are suitable for printing. These functions return (possibly deep) lists of characters and generate an error if the form is wrong.

DATA TYPES

```
chars() = [char() | chars()]

prompt() = atom() | unicode:chardata()

filesexpr() = {Sexpr,Line}
```

This is the format returned by `lfe_io:parse_file/1` and is used by the compiler to give better error information.

EXPORTS

```
get_line() -> Data | {error, ErrorInfo} | eof

get_line(Prompt) -> Data | {error, ErrorInfo} | eof

get_line(IoDevice, Prompt) -> Data | {error, ErrorInfo} | eof
```

Reads a line from the standard input (`IoDevice`), prompting it with prompt (`Prompt`). Note that this call guarantees that the input is saved in the input hisotry.

```
read() -> {ok, Sexpr} | {error, ErrorInfo} | eof

read(Prompt) -> {ok, Sexpr} | {error, ErrorInfo} | eof

read(IoDevice, Prompt) -> {ok, Sexpr} | {error, ErrorInfo} | eof
```

Read an s-expr from the standard input (`IoDevice`) with a prompt (`Prompt`). Note that this is not line-oriented in that it stops as soon as it has consumed enough characters.

```
read_line() -> {ok, Sexpr} | {error, ErrorInfo} | eof

read_line(Prompt) -> {ok, Sexpr} | {error, ErrorInfo} | eof

read_line(IoDevice, Prompt) -> {ok, Sexpr} | {error, ErrorInfo} | eof
```

Read the first s-expr from the standard input (`IoDevice`) with a prompt (`Prompt`). Note that this is line-oriented in that it reads whole lines discarding left-over characters in the last line.

`read_string(String) -> {ok, [Sexpr]} | {error, ErrorInfo}`

Read all the s-exprs from `String`.

`print(Sexpr) -> ok`

`print(IoDevice, Sexpr) -> ok`

Print the s-expr `Sexpr` to the standard output (`IoDevice`).

`print1(Sexpr) -> DeepCharList`

Return the list of characters which represent the s-expr `Sexpr`.

`prettyprint(Sexpr) -> ok`

`prettyprint(IoDevice, Sexpr) -> ok`

Pretty print the s-expr `Sexpr` to the standard output (`IoDevice`).

`prettyprint1(Sexpr) -> DeepCharList`

`prettyprint1(Sexpr, Depth) -> DeepCharList`

`prettyprint1(Sexpr, Depth, Indentation) -> DeepCharList`

`prettyprint1(Sexpr, Depth, Indentation, LineLength) -> DeepCharList`

Return the list of characters which represents the prettyprinted s-expr `Sexpr`. Default values for `Depth` is 30, `Indentation` is 0 and `LineLength` is 80.

`format(Format, Args) -> ok`

`format(IoDevice, Format, Args) -> ok`

fwrite(Format, Args) -> ok

`fwrite(IoDevice, Format, Args) -> ok`

Print formatted output. The following commands are valid in the format string:

- **`~w, ~W`** – print LFE terms
- **`~p, ~P`** – prettyprint LFE terms
- **`~s`** – print a string
- **`~e, ~f, ~g`** – print floats
- **`~b, ~B`** – based integers
- **`~x, ~X`** – based integers with a prefix
- **`~+, ~#`** – based integers in vanilla erlang format
- **`~~`** – prints ~
- **`~c, ~n, ~i`**

Currently they behave as for vanilla erlang except that `~w`, `~W`, `~p`, `~P` print the terms as LFE sexprs.

`format1(Format, Args) -> DeepCharList`

`fwrite1(Format, Args) -> DeepCharList`

Return the formatted output with same arguments as `format`/`fwrite`.

`read_file(FileName|Fd) -> {ok,[Sexpr]} | {error,ErrorInfo}`

`read_file(FileName|Fd, Line) -> {ok, [Sexpr]} | {error, ErrorInfo}`

Read the file `Filename` or the already opened file's file descriptor `Fd` returning a list of s-exprs.

`parse_file(FileName|Fd) -> {ok, [FileSexpr]} | {error, ErrorInfo}`

`parse_file(FileName|Fd, Line) -> {ok, [FileSexpr]} | {error, ErrorInfo}`

where

FileSexpr = filesexpr()

Read the file `Filename` or the already opened file's file descriptor `Fd` returning a list of pairs containing s-expr and line number of the start of the s-expr.

scan_sexpr(Cont, Chars) -> {done, Ret, RestChars} | {more, Cont1}

scan_sexpr(Cont, Chars, Line) -> {done, Ret, RestChars} | {more, Cont1}

This is a re-entrant call which scans tokens from the input and returns a parsed sepxr. If there are enough characters to parse a sexpr or it detects and error then it returns `{done,...}` otherwise it returns `{more,Cont}` where `Cont` is used in the next call to `scan_sexpr` with more characters to try and parse a sexpr. This is continued until a sexpr has been parsed. `Cont` is initially `[]`.

It is not designed to be called directly by an application but used through the i/o system where it can typically be called in an application by:

```
io:request(In, {get_until,unicode,Prompt,Module,scan_sexpr,[Line]})
```

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)
```