Language.CoreErlang.Syntax

```
-----------------------------------------------------------------------------
-- |
-- Module      :  Language.CoreErlang.Syntax
-- Copyright   :  (c) Henrique Ferreiro García 2008
--                (c) David Castro Pérez 2008
--                (c) Eric Bailey 2016
-- License     :  BSD-style (see the file LICENSE)
--
-- Maintainer  :  Alex Kropivny <alex.kropivny@gmail.com>
-- Stability   :  experimental
-- Portability :  portable
--
-- A suite of datatypes describing the abstract syntax of CoreErlang 1.0.3.
-- <http://www.it.uu.se/research/group/hipe/cerl/>

-----------------------------------------------------------------------------
{-# LANGUAGE DeriveDataTypeable #-}

module Language.CoreErlang.Syntax (
  -- * Modules
  Module(..), ModHeader(..), Exports, Attributes, ModAttribute,
  -- * Declarations
  FunDef(..),
  -- * Expressions
  Exp(..), Exps(..), Clause(..), Guard(..),
  List(..), Timeout(..), BitString(..), FunName,
  -- * Patterns
  Pats(..), Pat(..), Alias(..),
  -- * Literals
  Literal(..), Const(..), Atom(..),
  -- * Variables
  VarName,
  -- * Annotations
  Ann(..),
  ) where
```

```
import           Data.Data (Data, Typeable)

-- | A CoreErlang source module.
data Module = Module Atom ModHeader ModBody
  deriving (Eq,Ord,Show,Data,Typeable)

-- | A CoreErlang module header, i.e. exports and attributes.
data ModHeader = ModHeader Exports Attributes
  deriving (Eq,Ord,Show,Data,Typeable)

type Exports = [FunName]

-- | This type is used to represent function names
type FunName = (Atom,Integer)
-- deriving (Eq,Ord,Show,Data,Typeable)

type Attributes   = [ModAttribute]
type ModAttribute = (Atom,Const)
type ModBody      = [FunDef]

-- | This type is used to represent lambdas
data FunDef = FunDef (Ann FunName) (Ann Exp)
  deriving (Eq,Ord,Show,Data,Typeable)

-- | Constant (c):
data Const = CLit Literal
           -- ^Atomic'Literal'
           | CTuple [Const]
           -- ^@{ c_1, ..., c_n }@ @(n >= 0)@
           | CList (List Const)
           -- ^@[ c_1, ..., c_n ]@ @(n >= 1)@
  deriving (Eq,Ord,Show,Data,Typeable)

-- | /literal/.
-- Values of this type hold the abstract value of the literal, not the
-- precise string representation used. For example, @10@, @0o12@ and @0xa@
-- have the same representation.
data Literal = LInt     Integer  -- ^ integer literal
             | LFloat   Double   -- ^ floating point literal
             | LAtom    Atom     -- ^ atom literal
             | LNil              -- ^ empty list
             | LChar    Char     -- ^ character literal
             | LString  String   -- ^ string literal
  deriving (Eq,Ord,Show,Data,Typeable)

-- | A list of expressions
```

```haskell
data List a = L [a]
            | LL [a] a
  deriving (Eq,Ord,Show,Data,Typeable)


-- | This type is used to represent variable names.
type VarName = String

-- | A pattern, to be matched against a value.
data Pat = PVar VarName             -- ^ variable
         | PLit Literal             -- ^ literal constant
         | PTuple [Pat]             -- ^ tuple pattern
         | PList (List Pat)         -- ^ list pattern
         | PBinary [BitString Pat] -- ^ list of bitstring patterns
         | PAlias Alias             -- ^ alias pattern
  deriving (Eq,Ord,Show,Data,Typeable)

-- | An alias, used in patterns
data Alias = Alias VarName Pat
  deriving (Eq,Ord,Show,Data,Typeable)

-- | CoreErlang expression.
data Exp = Lit Literal                                    -- ^ literal constant
         | Var VarName                                    -- ^ variable name
         | FunName FunName                                -- ^ function name
         | Tuple [Exps]                                   -- ^ tuple expression
         | List (List Exps)                               -- ^ list expression
         | Binary [BitString Exps]                        -- ^ binary expression
         | Let ([VarName],Exps) Exps                      -- ^ local declaration
         | Case Exps [Ann Clause]                         -- ^ case expression
         | Fun [Ann VarName] Exps                         -- ^ fun expression
         | Letrec [FunDef] Exps                           -- ^ letrec expression
         | App Exps [Exps]                                -- ^ application
         | ModCall (Exps,Exps) [Exps]                     -- ^ module call
         | PrimOp Atom [Exps]                             -- ^ primop call
         | Try Exps ([VarName],Exps) ([VarName],Exps) -- ^ try expression
         | Rec [Ann Clause] Timeout                       -- ^ receive expression
         | Seq Exps Exps                                  -- ^ sequencing
         | Catch Exps                                     -- ^ catch expression
  deriving (Eq,Ord,Show,Data,Typeable)

-- | CoreErlang expressions.
data Exps = Exp (Ann Exp)        -- ^ single expression
          | Exps (Ann [Ann Exp]) -- ^ list of expressions
  deriving (Eq,Ord,Show,Data,Typeable)
```

```haskell
-- | A bitstring.
data BitString a = BitString a [Exps]
  deriving (Eq,Ord,Show,Data,Typeable)

-- | A /clause/ in a @case@ expression
data Clause = Clause Pats Guard Exps
  deriving (Eq,Ord,Show,Data,Typeable)

data Pats = Pat Pat     -- ^ single pattern
          | Pats [Pat] -- ^ list of patterns
  deriving (Eq,Ord,Show,Data,Typeable)

-- | A guarded alternative @when@ /exp/ @->@ /exp/.
-- The first expression will be Boolean-valued.
data Guard = Guard Exps
  deriving (Eq,Ord,Show,Data,Typeable)


-- | This type is used to represent atoms
data Atom = Atom String
 deriving (Eq,Ord,Show,Data,Typeable)

-- | The timeout of a receive expression
data Timeout = Timeout Exps Exps
  deriving (Eq,Ord,Show,Data,Typeable)

-- | An annotation for modules, variables, ...
data Ann a = Constr a      -- ^ core erlang construct
           | Ann a [Const] -- ^ core erlang annotated construct
  deriving (Eq,Ord,Show,Data,Typeable)
```