Domain Specific Languages in Erlang

Dennis Byrne
dbyrne@thoughtworks.com

An Overview ...

1. Erlang is ...
2. A Domain Specific Language is ...
    1. Internal DSLs
    2. External DSLs
3. A simple DSL implementation
    1. Dynamic code loading
4. A complex DSL implementation
    1. erl_scan module
    2. erl_parse module
    3. erl_eval module

ThoughtWorks®

Erlang is ...

a functional programming language with native constructs for concurrency and reliability.

- Message passing / Actor Model
- Dynamic code loading
- Single Assignment
- Strict Evaluation
- Tail Recursion
- Created more than 20 years ago
- Open sourced in 1998

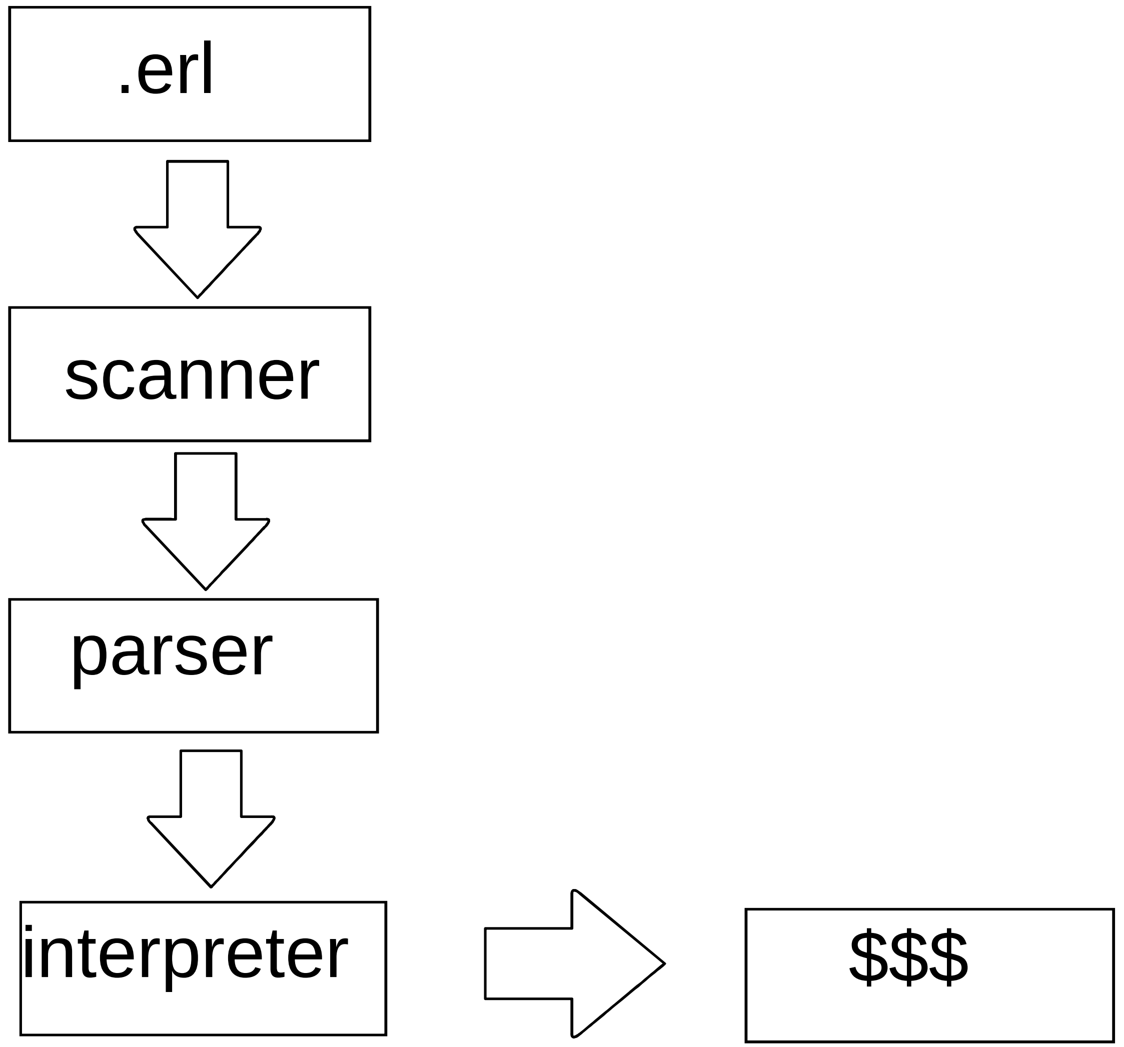
A Domain Specific Language is ...

a narrowly focussed language dedicated to solving a specific problem very well.

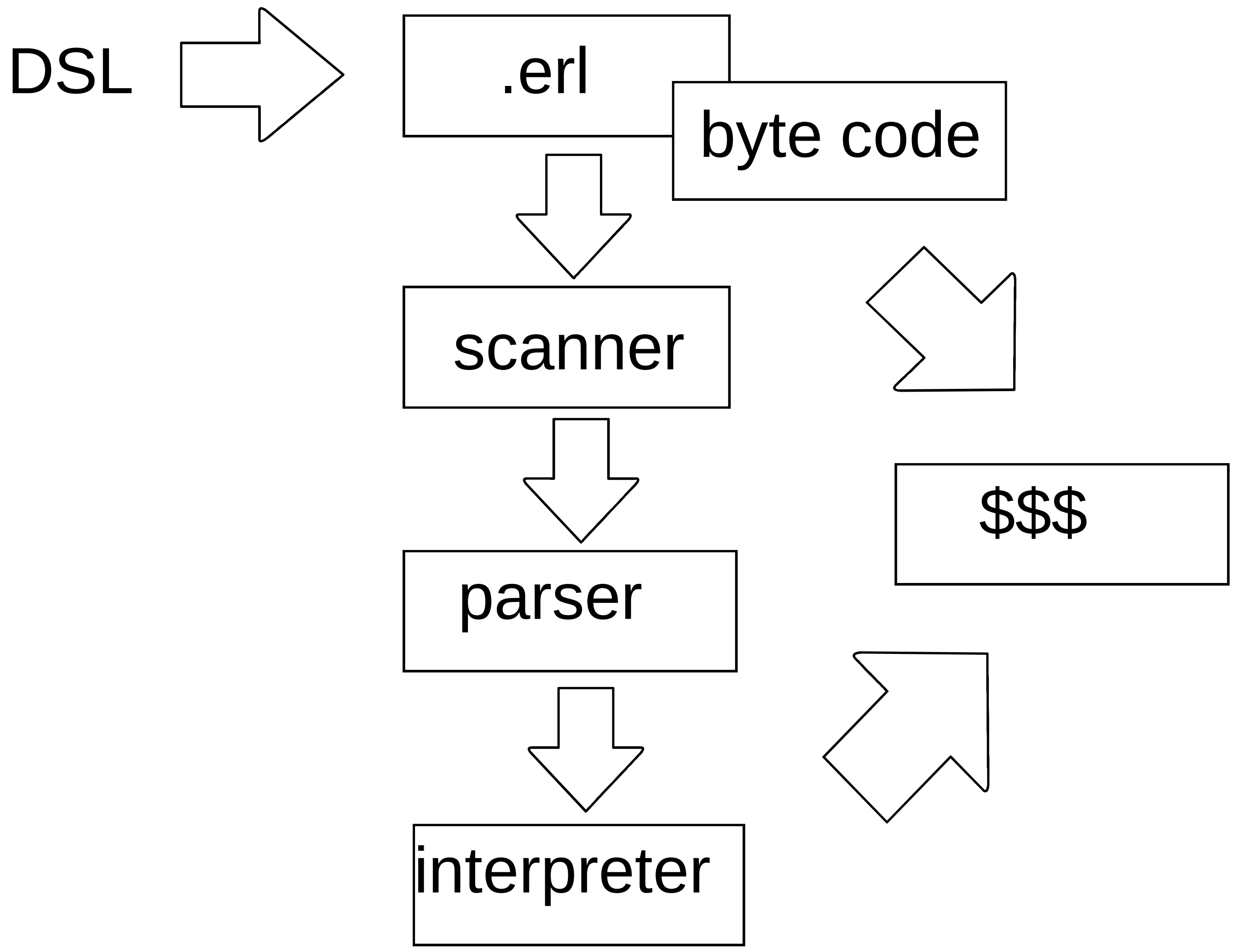
- Internal DSL
  - Written in the host language
  - Also known as an embedded DSL
  - Common in Ruby and Lisp communities
- External DSL
  - Implemented outside of the host language
  - Read and/or interpreted by the host platform
  - Examples: J2EE deployment descriptors, SQL, Regex
  - We will be creating two external DSLs

ThoughtWorks®

There are many ways to do this ...



A Simple DSL Implementation



Credit DSL

```
grant loan if
    credit score exceeds 770

grant loan if
    loan amount is below 20000 and
    down payment exceeds 2000

deny loan if
    loan amount exceeds 500000

deny loan if
    credit score is below 600 or
    liabilities exceeds assets
```

Credit DSL

grant loan if
credit score exceeds 770

grant loan if
loan amount is below 20000 and
down payment exceeds 2000

deny loan if
loan amount exceeds 500000

deny loan if
credit score is below 600 or
liabilities exceeds assets

Return Values
Guards
Variables
Operators

From CreditDSL to credit_dsl.erl

```
grant loan if
    credit score exceeds 770

qualify_loan( args ) when CreditScore > 770 -> grant;
```

```
grant loan if
    loan amount is below 20000 and
    down payment exceeds 2000

qualify_loan( args ) when
    Loan < 20000 andalso DownPayment > 2000 -> grant;
```

ThoughtWorks®

From CreditDSL to credit_dsl.erl

```
deny loan if
    loan amount exceeds 500000

qualify_loan( args ) when Loan > 500000 -> deny;
```

```
deny loan if
    credit score is below 600 or
    liabilities exceeds assets

qualify_loan( args ) when
    CreditScore < 600 orelse Liabilities > Assets -> deny;
```

Guard Sequence, the credit_dsl Module

```
qualify_loan( args ) when
    CreditScore > 770 -> grant;
qualify_loan( args ) when
    Loan < 20000 andalso DownPayment > 2000 -> grant;
qualify_loan( args ) when
    Loan > 500000 -> deny;
qualify_loan( args ) when
    CreditScore < 600 orelse Liabilities > Assets -> deny;
 qualify_loan( args ) -> true.
```

A Contrived Module: loan_processor.erl

```
start() -> spawn( fun() -> loop() end ).

loop() ->
   receive {Loan, CreditScore, Down, Liabilities, Assets} ->
      Outcome = credit_dsl:qualify_loan(Loan,
                                        CreditScore,
                                        Down,
                                        Liabilities,
                                        Assets),
      io:format("loan processed : ~p~n", [Outcome])
   end,
   loop().
```

Dynamic (Re)Loading CreditDSL

```
1> c(credit_dsl).          % compiles and loads credit_dsl module
2> c(loan_processor).
3> ProcessId = loan_processor:start().
4> ProcessId ! {600000, 700, 10000, 10000, 10000}.
   loan processed : deny
5> ProcessId ! {600000, 780, 10000, 10000, 10000}.
   loan processed : grant
6> c(credit_dsl).          % reloads after changing our DSL rules
7> ProcessId ! {600000, 780, 10000, 10000, 10000}.
   loan processed : deny
```

Dynamic Code Loading Gotchas

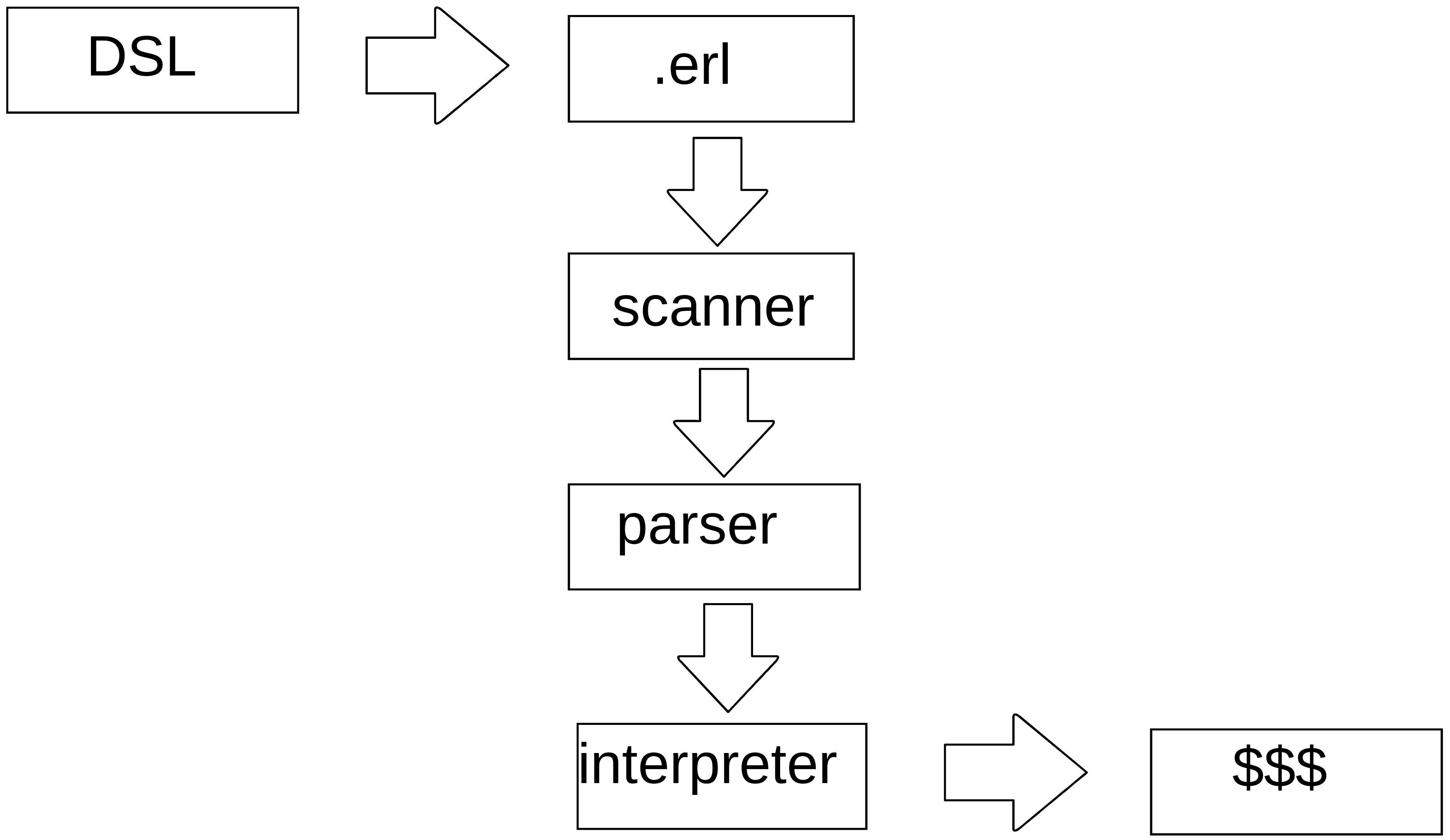
- Erlang can run two and only two versions of a module
- Code running in a process has a version lifecycle
  - Current
  - Old
  - Dead
- Keep it simple
  - Try to put reloaded code in a separate module
  - Try to keep reloaded code out of processes
  - Keep business logic separate from infrastructural code
  - Keep concurrent code separate from sequential code

Modifying loan_processor.erl

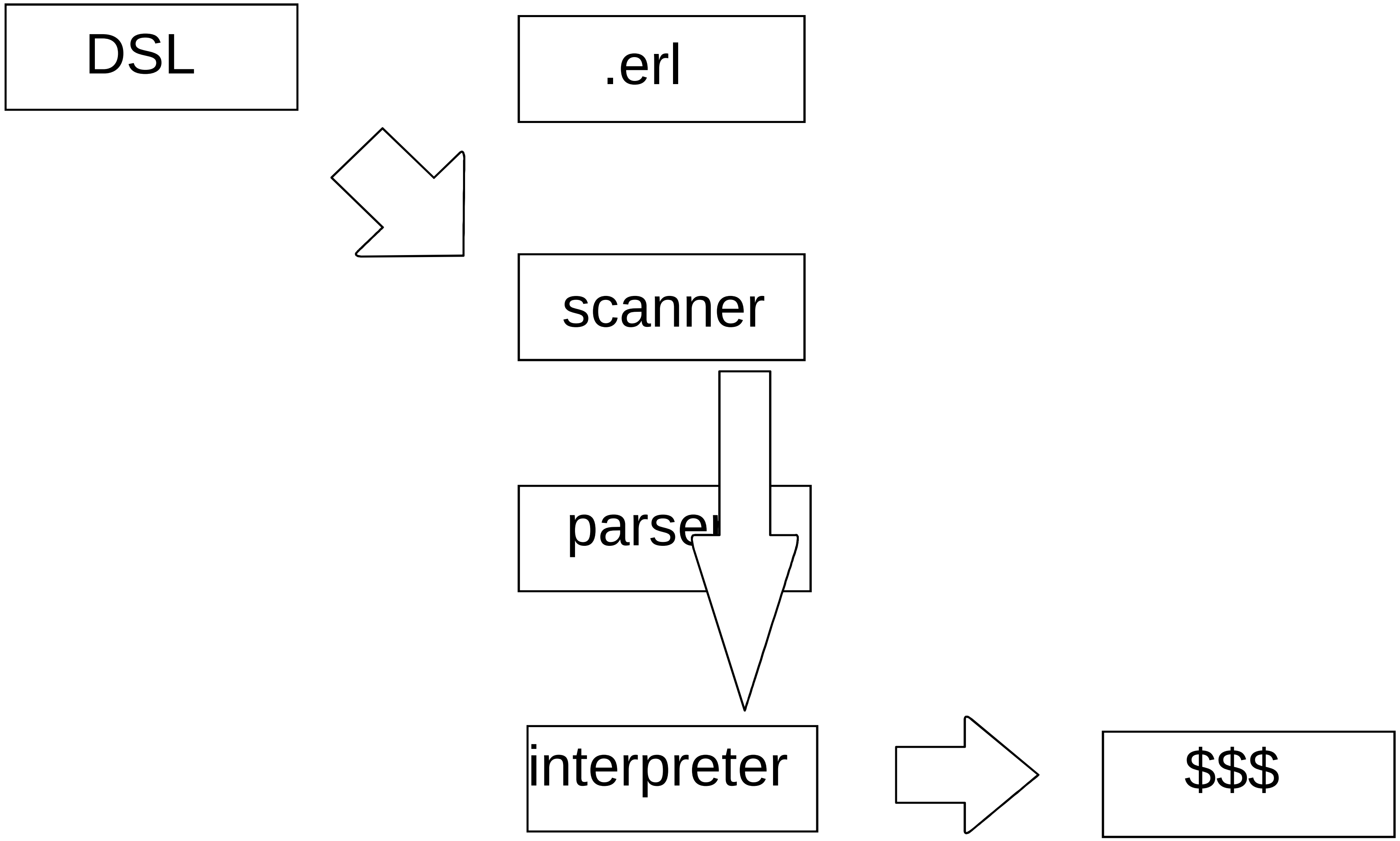
```
start() -> spawn( fun() -> loop() end ).

loop() ->
  receive {Loan, CreditScore, Down, Liabilities, Assets} ->
    Outcome = credit_dsl:qualify_loan(Loan,
                                      CreditScore + 100,
                                      Down,
                                      Liabilities,
                                      Assets),
    io:format("loan processed : ~p~n", [Outcome])
  end,
  loop().
```

A Simple DSL Implementation



A Complex DSL Implementation



Module: erl_scan

Contains functions for tokenizing characters into Erlang tokens.

- erl_scan:string/1 -> { }
  - Used to tokenize a list of characters
- erl_scan:token/2 -> { }
  - Reentrant scanner, scans up to a dot (.)
- erl_scan:reserved_word/1 -> true|false
  - Used to determine if a token is a reserved word

erl_scan:string/1 -> { }

```
> { ok, Tokens, EndLine } = erl_scan:string( "1 + 1." ).
> erlang:display( EndLine ).
  1
>erlang:display( Tokens ).
  [ {integer,1,1},  {'+',1},  {integer,1,1},  {dot,1} ]


> % syntax rules are not applied at this point
> { ok, Tokens, EndLine } = erl_scan:string( "1 + foo." ).
> erlang:display( Tokens ).
  [ {integer,1,1},  {'+',1},  {atom,1,foo},  {dot,1} ]
```

Module: erl_parse

Converts tokens into the abstract form.

- erl_parse:parse_exprs/1 -> { }
  - Parses tokens as a list of expressions
- erl_parse:abstract/1 -> { }
  - Converts an Erlang term into it's abstract form
  - Inverse of erl_parse:normalise/1
- erl_parse:normalise/1 -> Term
  - Converts abstract form into an Erlang term
  - Inverse of erl_parse:abstract/1

erl_parse:parse_exprs/1 -> { }

erl_parse:parse_exprs/1

```
> { ok, Tokens, EndLine } = erl_scan:string( "1 + 1." ).
> { ok, Abstract } = erl_parse:parse_exprs( Tokens ).
> erlang:display( Abstract ).
  [ {op, 1, '+', {integer,1,1}, {integer,1,1} } ]
```

The infix operator notation of the concrete syntax tree has become converted to an abstract form using postfix operator notation.

Module: erl_parse

erl_parse:parse_exprs/1 applied syntax rules

```
> { ok, Tokens, _ } = erl_scan:string( "1 + 1" ).
> { error, Msg } = erl_parse:parse_exprs( Tokens ).
> erlang:display( Msg ).
  { 999999, erl_parse, ["syntax error before: ",[] ] }
```

erl_parse:abstract/1 & normalise/1

```
> { integer, 0, 1 } = erl_parse:abstract( 1 ).
> 1 = erl_parse:normalise(  {integer, 0, 1} ).
```

yecc (LALR-1 Parser Generator)

```
> % produce the module credit_dsl_parser.erl
> % takes a BNF grammar definition
> yecc:yecc(“credit_dsl.yrl”,“credit_dsl_parser.erl”).
> % dynamically load the parser
> c(credit_dsl_parser).
> % parsing tokens produced by erl_scan
> credit_dsl_parser:parse(Tokens).
```

ThoughtWorks®

Module: erl_eval

- erl_eval:expr/2 -> { }
  - Evaluates an expression with a set of bindings Bindings
- erl_eval:exprs/2 -> { }
  - Evaluates expressions with a set of bindings Bindings
- erl_eval:expr_list/2 -> { }
  - Evaluates a list of expressions
- erl_eval:new_bindings/0 -> [ ]
  - Used to create a new binding structure
- erl_eval:add_binding/3 -> [ ]
  - Used to add a binding to a binding structure
- erl_eval:del_binding/2 -> [ ]
  - Used to remove a binding from a binding structure

Happy Path ... erl_eval:exprs/2 -> { }

```
> { _, Tokens, _ } = erl_scan:string("1 + 1.").
> { ok, Abstract } = erl_parse:parse_exprs( Tokens ).
> erlang:display( Abstract ).
  [ {op,1,'+',{integer,1,1},{integer,1,1}} ].
> Bindings = erl_eval:new_bindings( ).
> { _, Value, _ } = erl_eval:exprs( Abstract, Bindings ).
> erlang:display( Value ).
  2
```

erl_eval:exprs/2 -> {} ... with bindings

```
> { ok, Tokens, _ } = erl_scan:string( "1 + X." ).
> { ok, Abstract } = erl_parse:parse_exprs( Tokens ).
> erlang:display( Abstract ).
  { ok, [{op,1,'+',{integer,1,1}, {var,1,'X'}}] }
> Bindings = erl_eval:new_bindings( ).
> erlang:display( Bindings ).
  [ ]
> NewBindings = erl_eval:add_binding( 'X', 2, Bindings ).
  [{'X',2}]
> { _, Result, _ } = erl_eval:exprs( Abstract, NewBindings ).
> erlang:display( Result ).
  3
```

ThoughtWorks®

A Complex DSL for Stocks

```
buy 9000 shares of GOOG when price is less than 500
sell 400 shares of MSFT when price is greater than 30
buy 7000 shares of AAPL when price is less than 160
```

InfoQueue
Enterprise Software Development Community

StocksDSL Example

```
> c(stocks_dsl). % compile and load the module, once

> % convert a collection of rules into a collection of  closures
> Functions = stocks_dsl:load_biz_rules("path/to/biz_rules.txt").

> MarketData = [{'GOOG', 498}, {'MSFT', 30}, {'AAPL', 158}].
> dsl:apply_biz_rules(Functions, MarketData).
  Order placed: buying 9000 shares of 'GOOG'
  Order placed: buying 7000 shares of 'AAPL'
```

StocksDSL Implementation

```
load_biz_rules(File) ->
    {ok, Bin} = file:read_file(File),
    Rules = string:tokens(erlang:binary_to_list(Bin), "\n"),
    [rule_to_function(Rule) || Rule <- Rules].

rule_to_function(Rule) ->
    {ok, Scanned, _} = erl_scan:string(Rule),
    [{_,_,Action},{_,_,Quantity},_,_|Tail] = Scanned,
    [{_,_,Ticker},_,_,_,{_,_,Operator},_,{_,_,Limit}] = Tail,
    rule_to_function(Action, Quantity, Ticker, Operator, Limit).
```

StocksDSL Implementation

```
% creates and returns a single closure

rule_to_function(Action, Quantity, Ticker, Operator, Limit) ->
  Abstract = build_abstract_form(Action, Quantity, Ticker,
                                                  Operator, Limit),
  fun(Symbol, Price) ->
    Bin = erl_eval:new_bindings(),
    TickerBin = erl_eval:add_binding('Ticker', Symbol, Bin),
    AllBindings = erl_eval:add_binding('Price', Price, TickerBin),
    erl_eval:exprs(Abstract, AllBindings)
  end.
```

Domain Specific Languages in Erlang

Dennis Byrne
dbyrne@thoughtworks.com

- http://www.infoq.com/articles/erlang-dsl
- http://erlang.org/
- http://martinfowler.com/dslwip/Intro.html