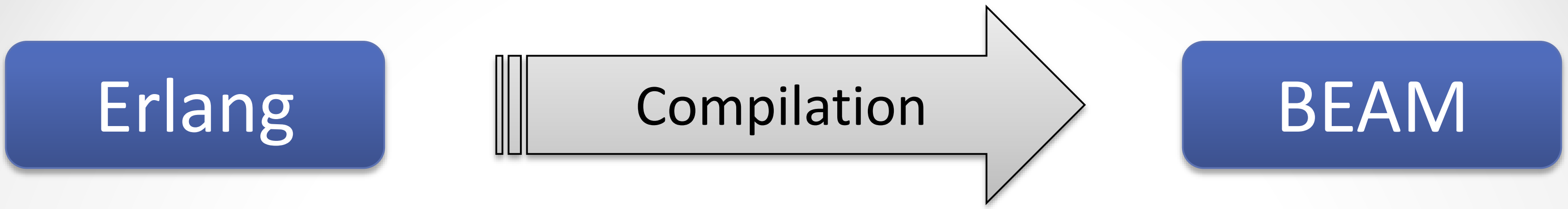
Extending Erlang by Utilizing RefactorErl

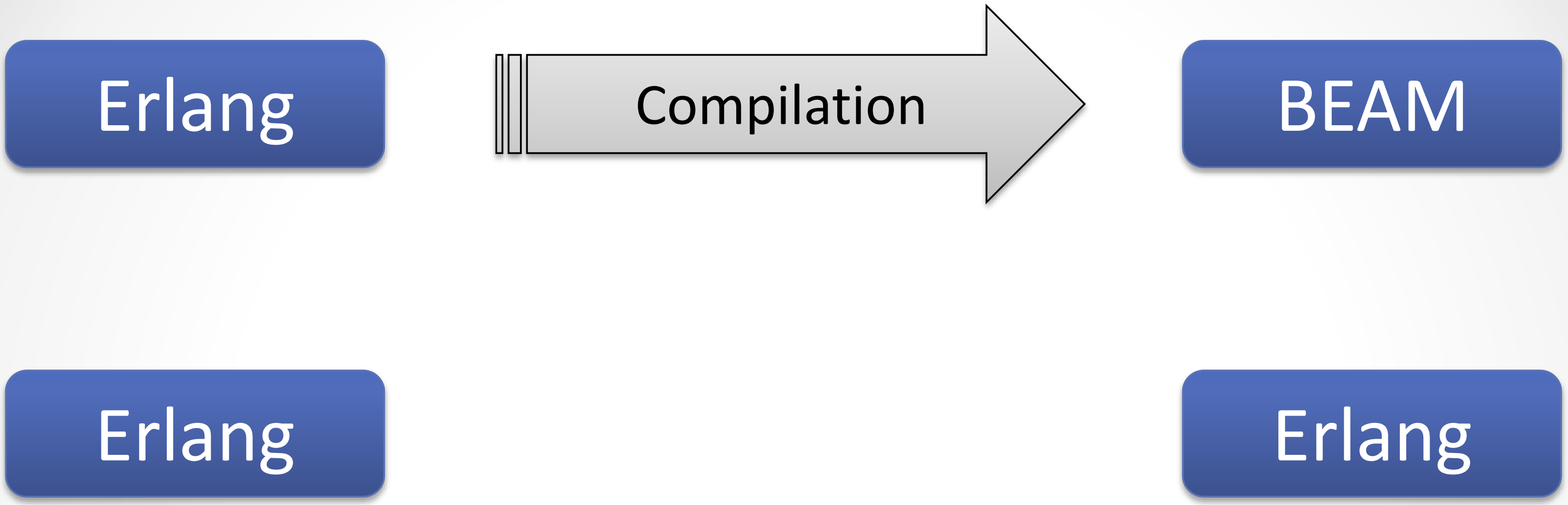
Dániel Horpácsi

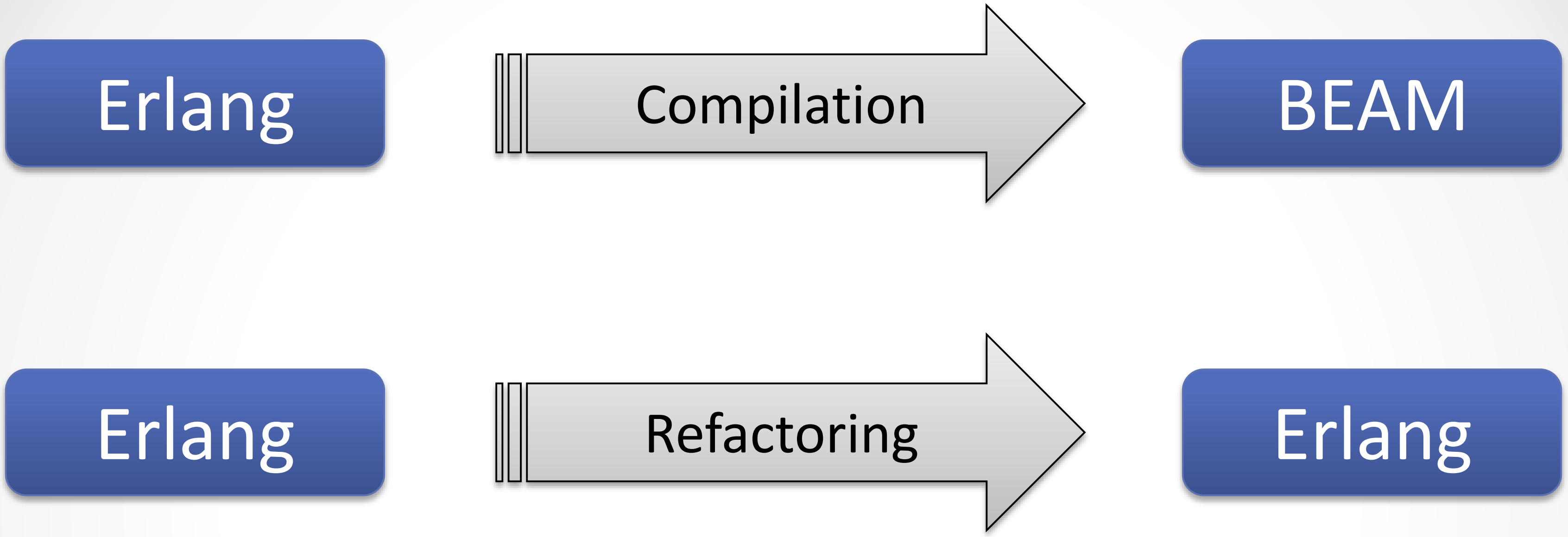
Erlang Factory Lite 2013 Budapest

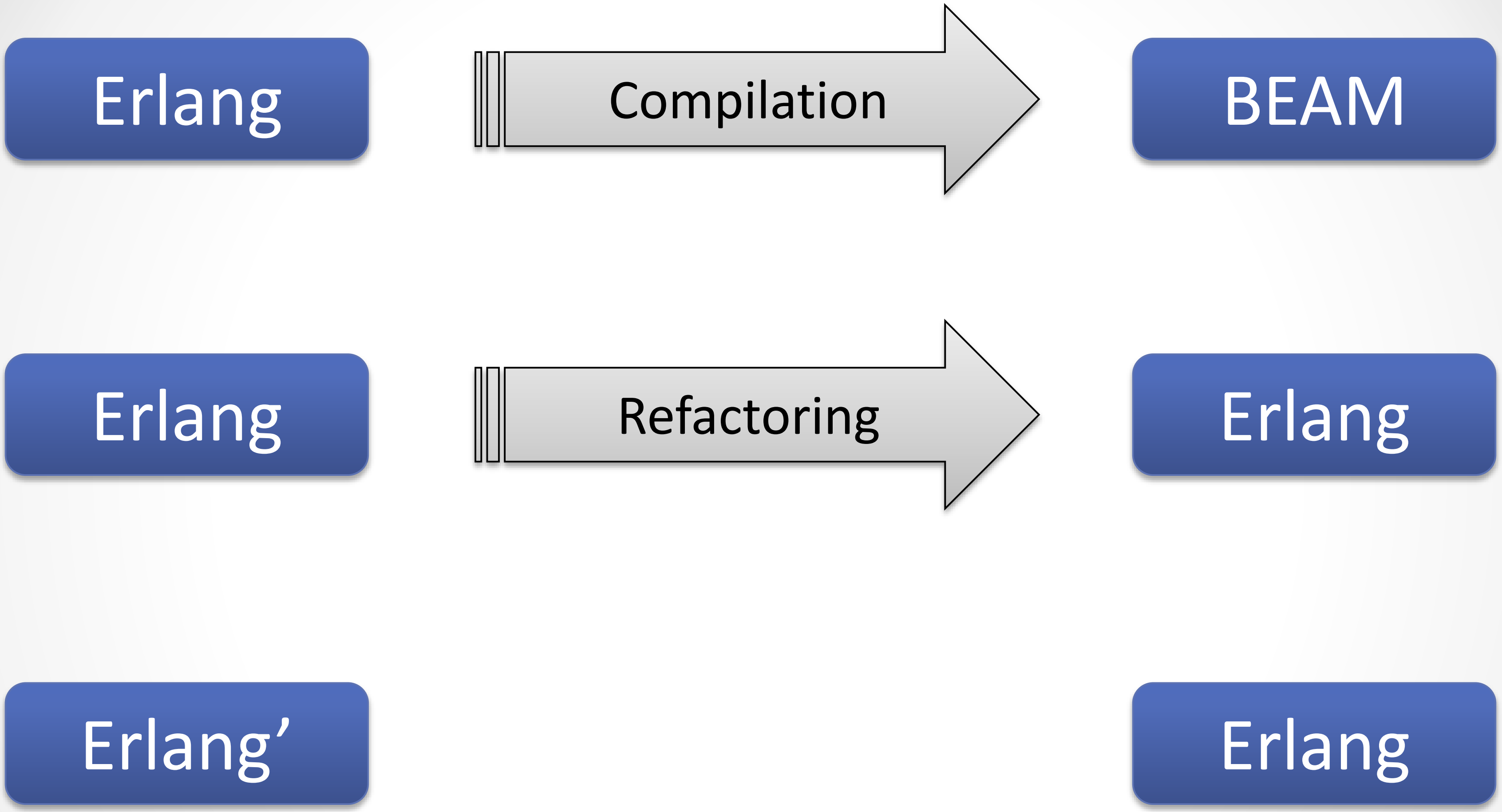
Erlang

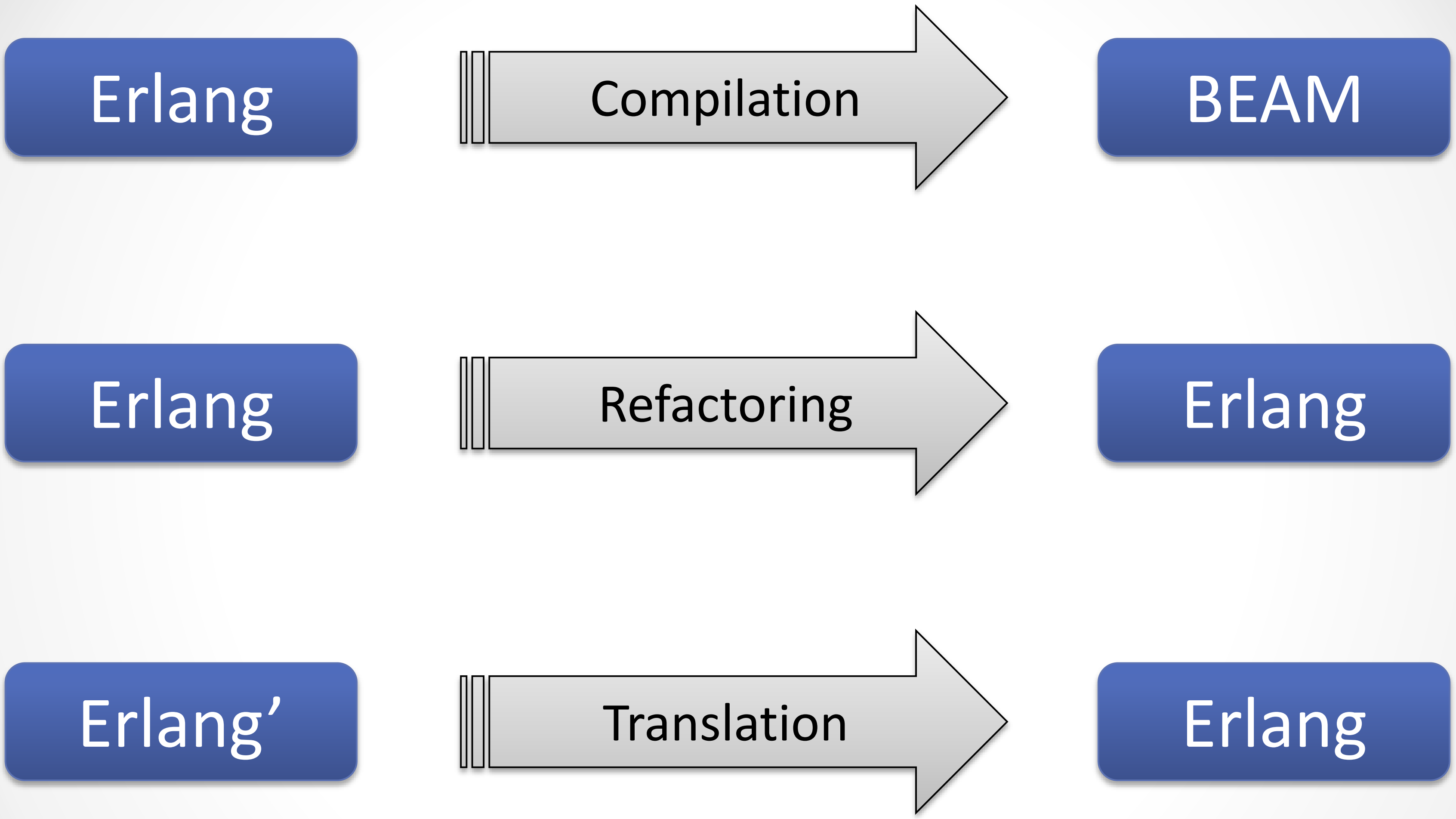
BEAM


Erlang
Compilation
BEAM


Erlang
Compilation
BEAM
Erlang
Erlang


Erlang
Compilation
BEAM
Erlang
Refactoring
Erlang


Erlang
Compilation
BEAM
Erlang
Refactoring
Erlang
Erlang’
Erlang


Erlang
Compilation
BEAM
Erlang
Refactoring
Erlang
Erlang’
Translation
Erlang

Program transformations

Translation

- Compilation
- Migration
- Code synthesis

Rephrasing

- Desugaring
- Refactoring
- Obfuscation

Program transformations

Translation

Higher-level language to lower-level language

- Compilation
- Migration
- nthesis

Rephrasing

- Desugaring
- Refactoring
- Obfuscation

Program transformations

Translation

Higher-level language to lower-level language

- Compilation
- Migration
- nthesis

Rephrasing

„Translation” to the same language

- Desugaring
- Refactoring
- ation

Program transformations

- „Program as Data”
- Generative programming
- Meta-programming

Translation

Higher-level language to lower-level language

nthesis

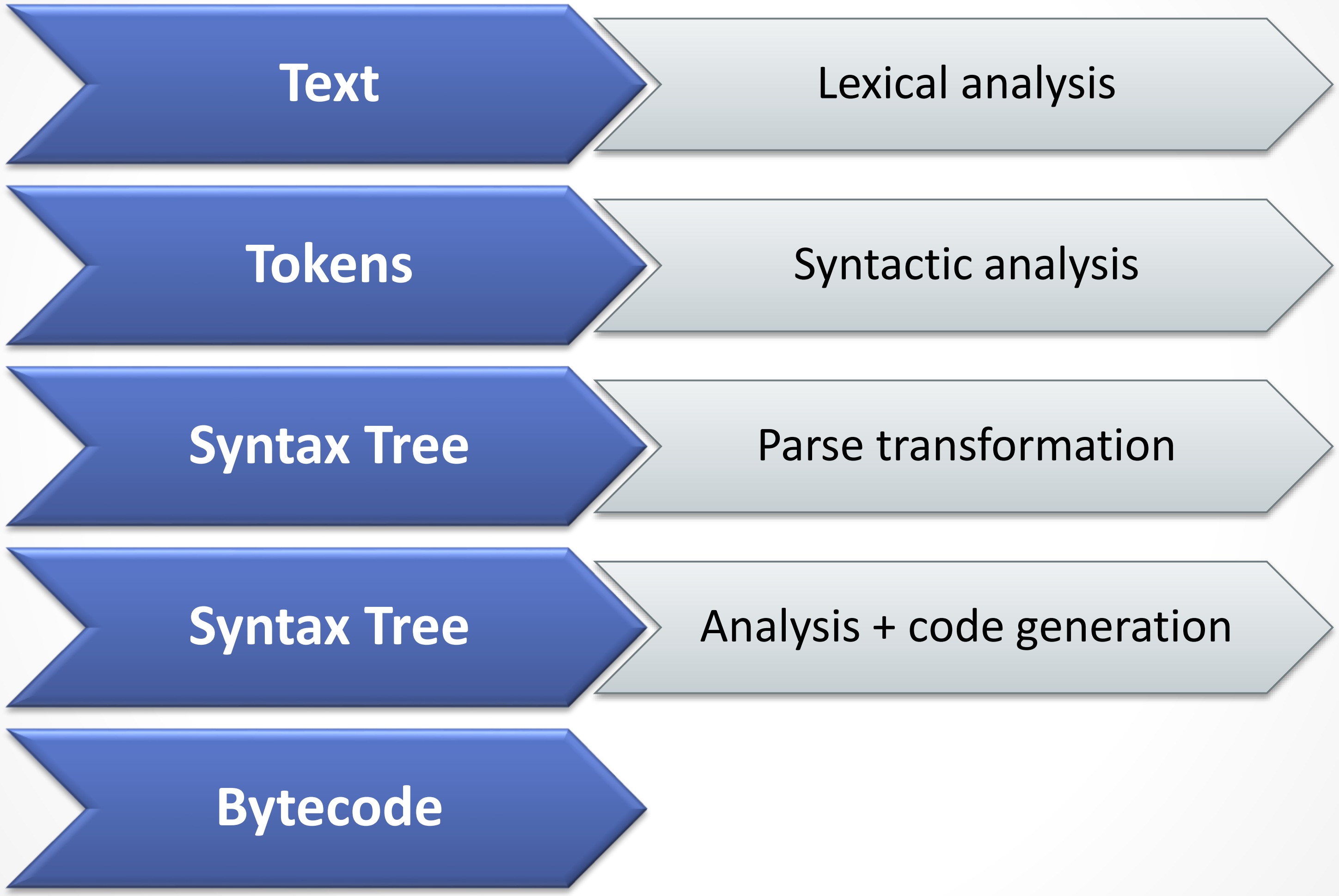
- Desugaring
- Refactoring

ation

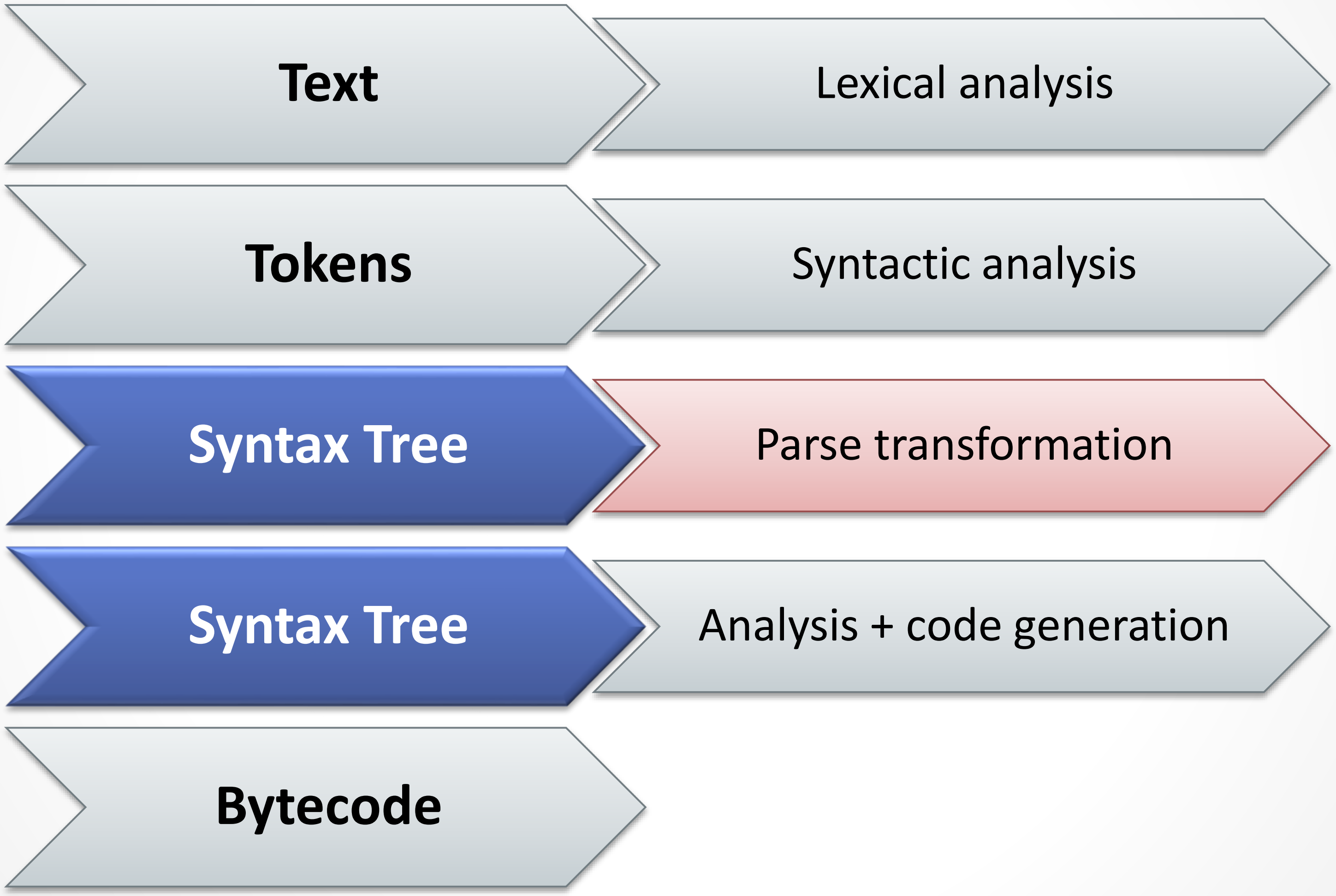
Rephrasing

„Translation” to the same language

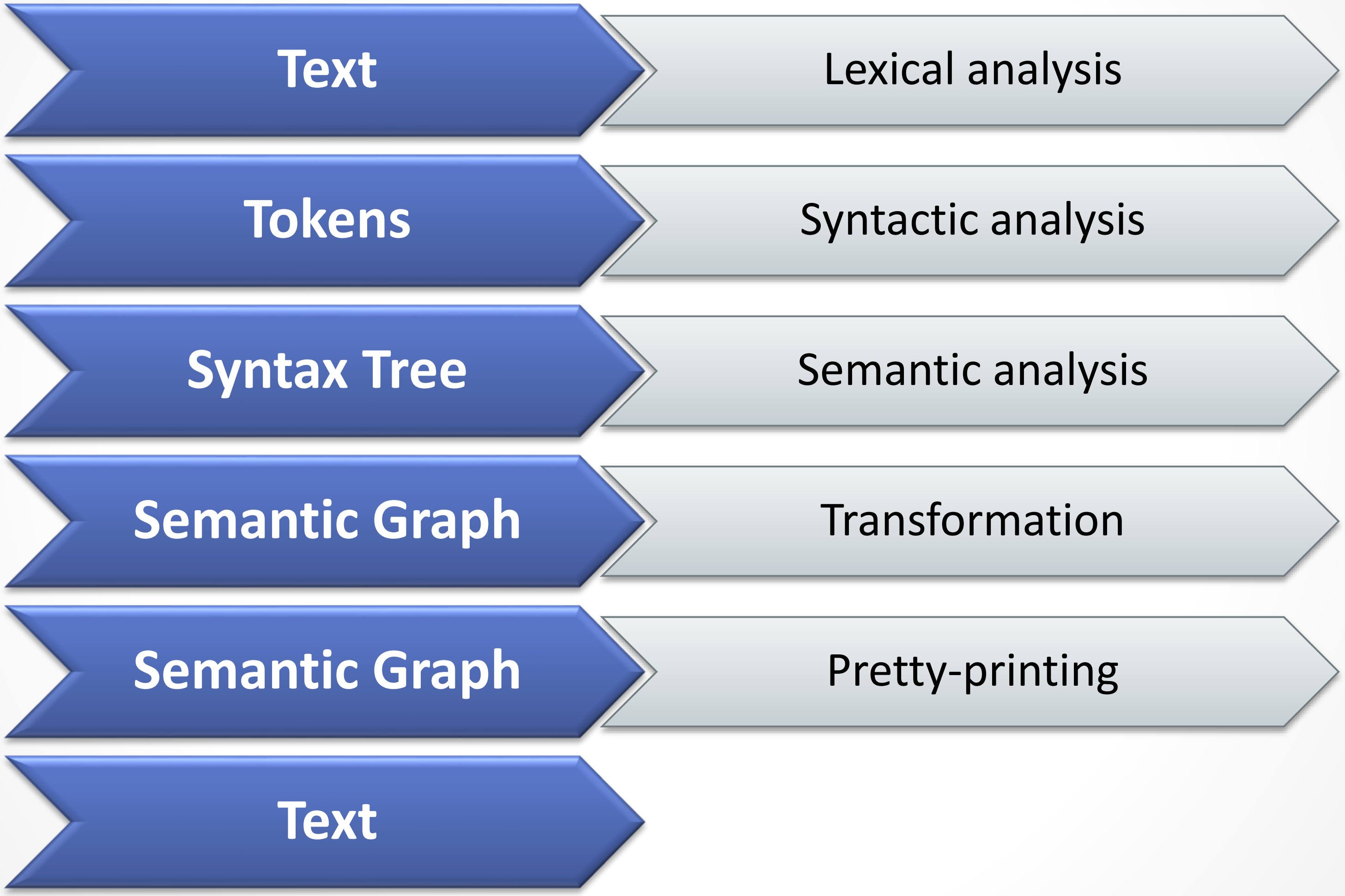
Erlang Compiler



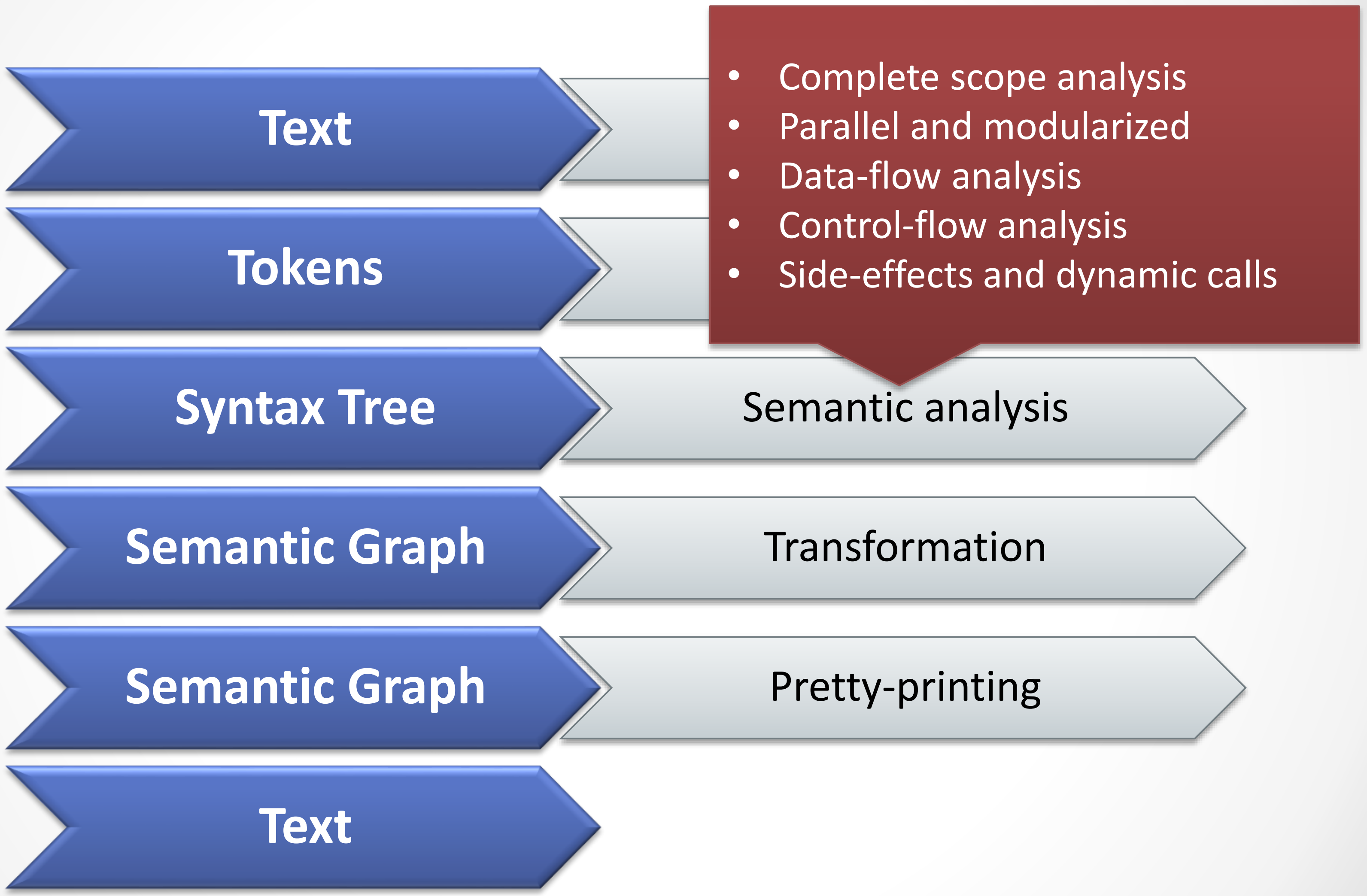
Erlang Parse Transformation



Erlang refactoring tool

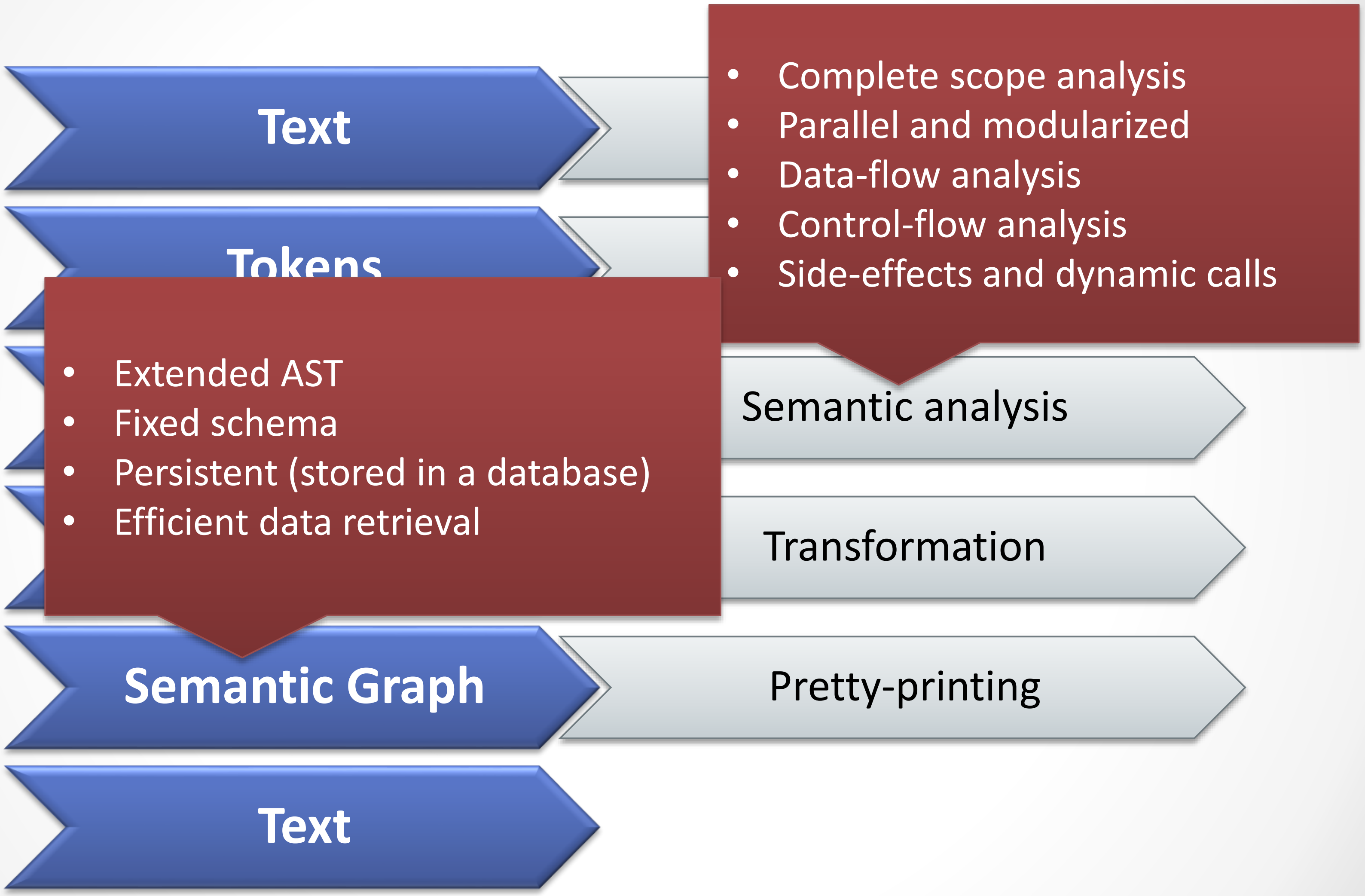


Erlang refactoring tool



- Complete scope analysis
- Parallel and modularized
- Data-flow analysis
- Control-flow analysis
- Side-effects and dynamic calls

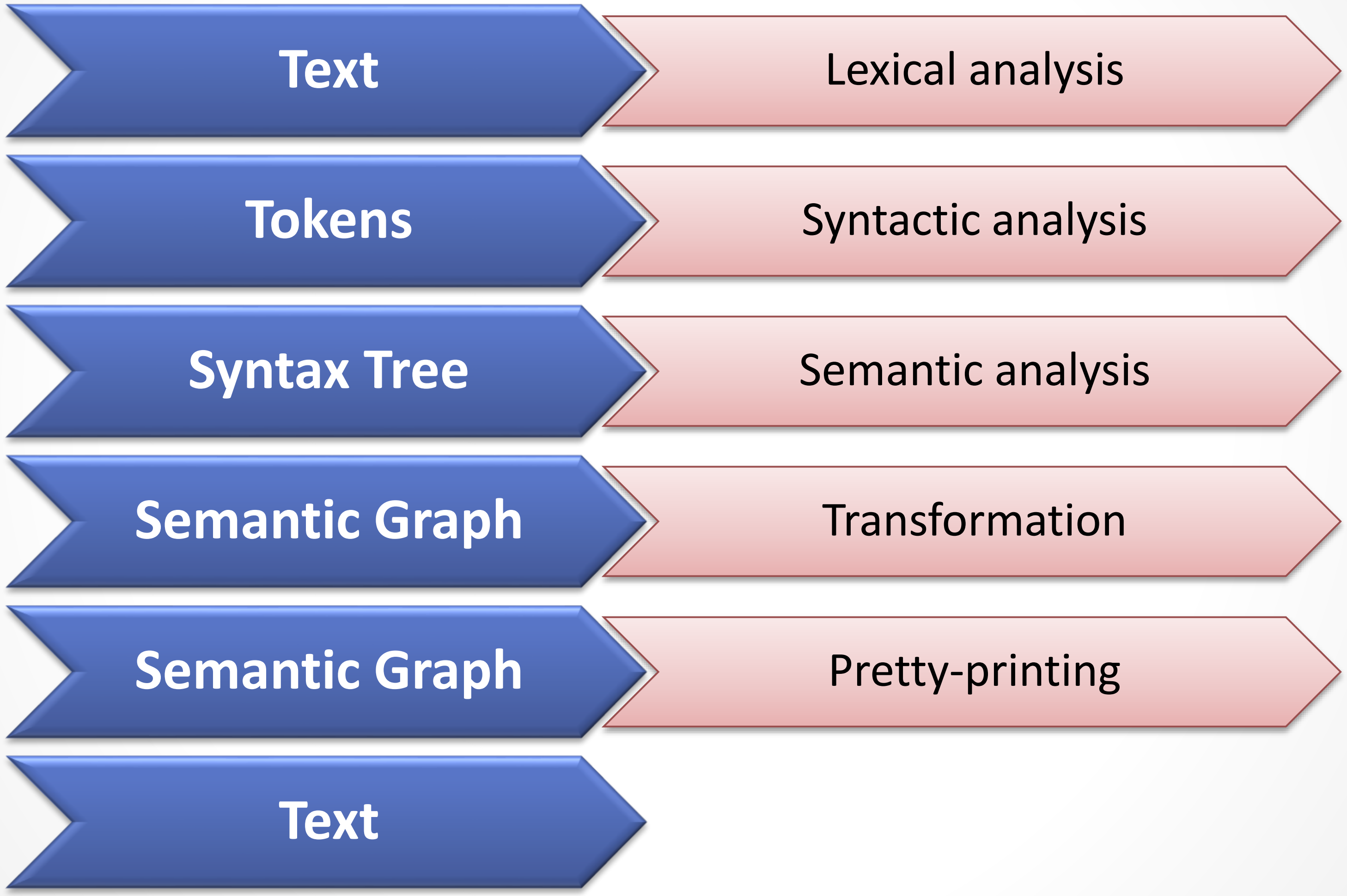
Erlang refactoring tool



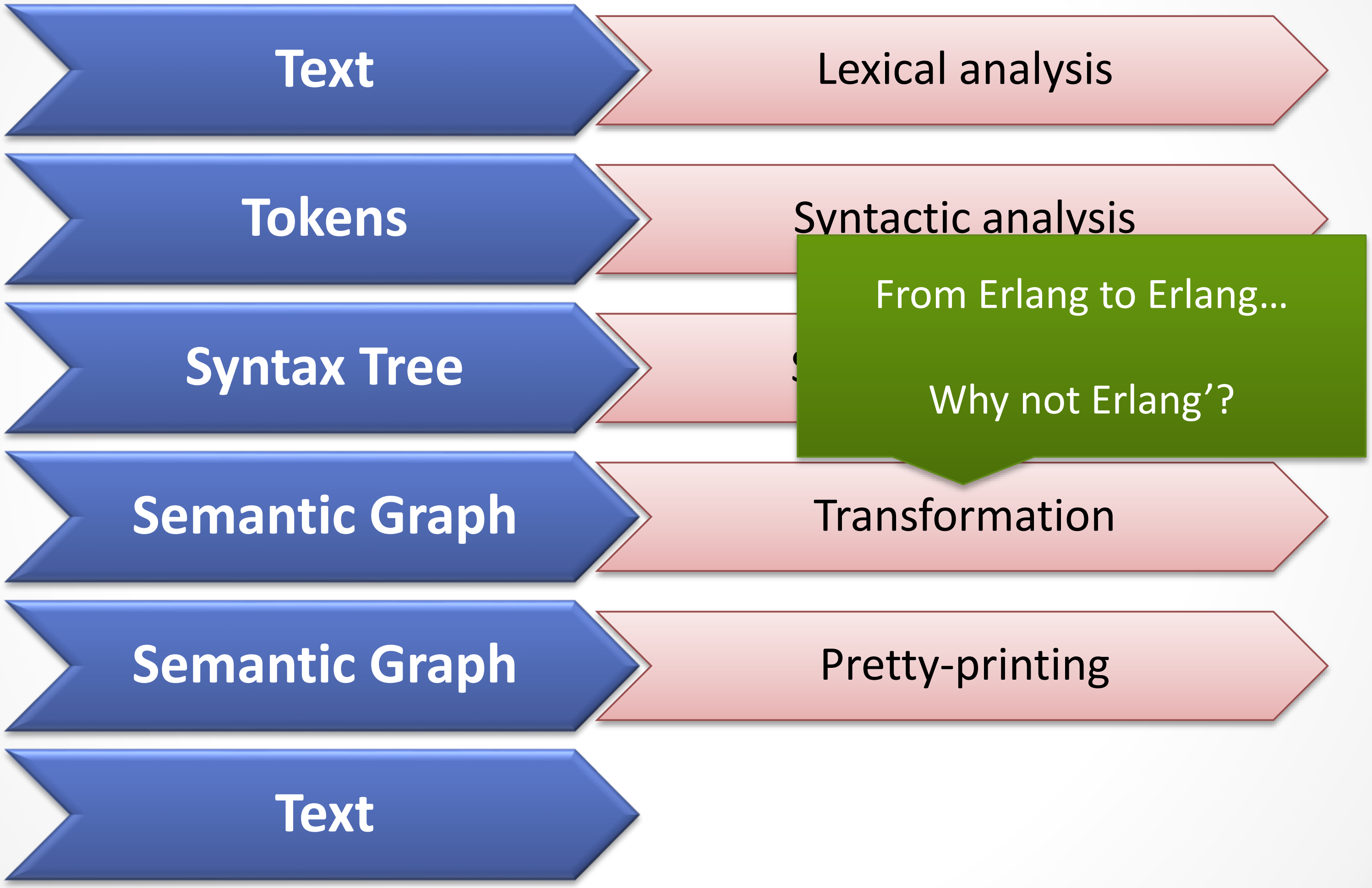
- Complete scope analysis
- Parallel and modularized
- Data-flow analysis
- Control-flow analysis
- Side-effects and dynamic calls

- Extended AST
- Fixed schema
- Persistent (stored in a database)
- Efficient data retrieval

Refactoring tool vs. compiler



Refactoring tool vs. compiler



What would Erlang' be?

EEP 0012	Extensions to comprehensions
EEP 0013	-enum declarations
EEP 0015	Portable funs
EEP 0019	Comprehension multigenerators
EEP 0021	Optional trailing commas for lists and tuples
EEP 0027	Multi-parameter type-checking BIFs
EEP 0037	Funs with names
EEP ?	User-defined operators

Implementing transformations

Analysis and Queries

- Complete semantic analysis
- Graph traversal library
- Semantic Query library

Transformation

- Aided construction of subtrees
- Automatically generated tokens
- On-the-fly calculated semantic layer

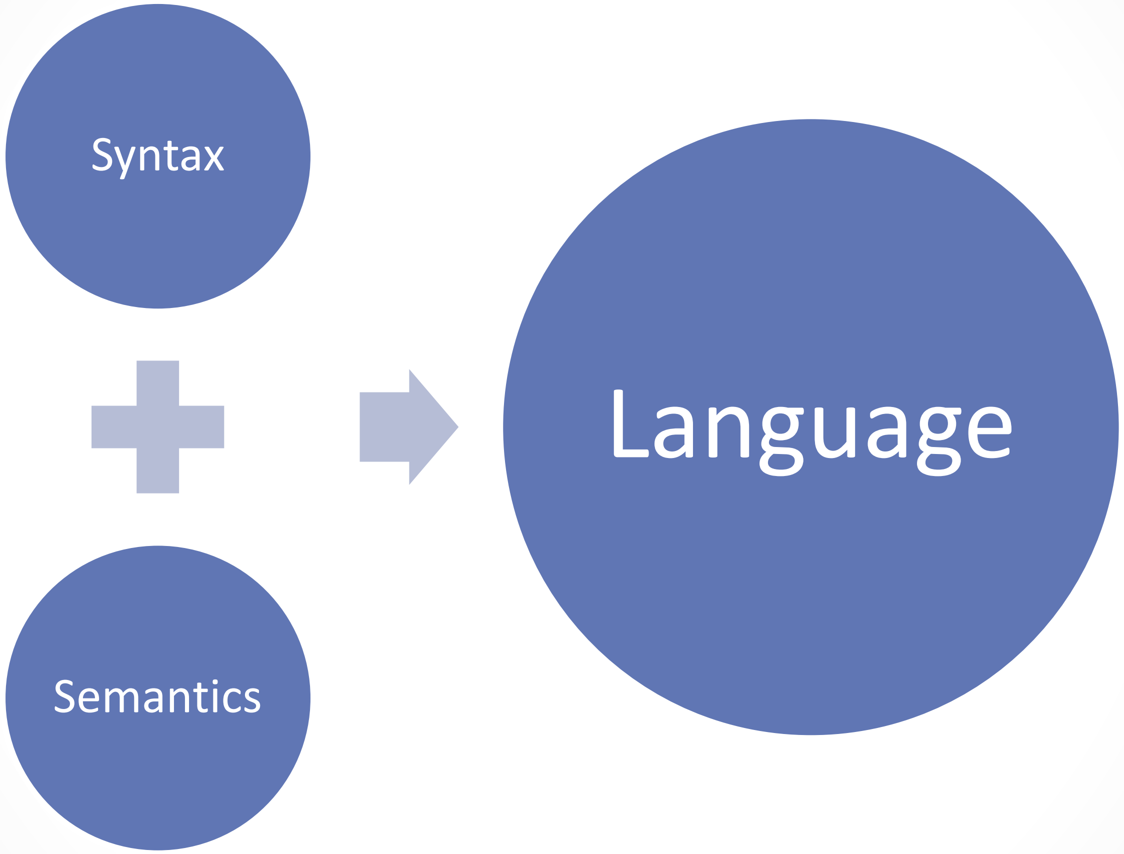
Implementing translations

Analysis

- Parser modified to accept Erlang’
- Same, or similar, abstract syntax
- Already present semantic analysis

Transformation

- Translation implemented like a refactoring
- Using the available query libraries
- Using the present transformation framework


Syntax
Semantics
Language

User-defined operators

User-defined operators

Syntax

Allow a list of special chars to be an operator of an infix expression.
Precedence and associativity is specified in Erlang module attributes.

- New token:
```
    operator        [<>+-*/=?!#:|@&.]+
```
- Infix expressions are allowed to be built by a custom operator

User-defined operators

Syntax

Allow a list of special chars to be an operator of an infix expression. Precedence and associativity is specified in Erlang module attributes.

- New token:
  ```
  operator        [<>+-*/=?!#:|@&.]+
  ```
- Infix expressions are allowed to be built by a custom operator

Semantics

Infix expressions in the code are parsed according to the precedence and associativity rules of the built-in and the user-defined operators.

User-defined operators

Syntax

Allow a list of special chars to be an operator of an infix expression.
Precedence and associativity is specified in Erlang module attributes.

Is this compatible with the AST format?

- New token:

  `oper`

- Infix expressions are allowed to be built by a custom operator

Semantics

Infix expressions in the code are parsed according to the precedence and associativity rules of the built-in and the user-defined operators.

```
-infixl(!!  / 2).
-infixl(>-< / 3).

f(N) -> [1,2,3] >-< [3,4,5] !! N.

!! (L,  I ) -> lists:nth(I, L).
>-<(L1, L2) -> lists:merge(L1, L2).
```

```
-infixl(!!  / 2).
-infixl(>-< / 3).

f(N) -> [1,2,3] >-< [3,4,5] !! N.

!! (L,  I ) -> lists:nth(I, L).
>-<(L1, L2) -> lists:merge(L1, L2).
```

```
f(N) -> '!!'('>-<'([1,2,3], [3,4,5]), N).

'!!' (L,  I ) -> lists:nth(I, L).
'>-<'(L1, L2) -> lists:merge(L1, L2).
```

```
-infixl(!!  / 2).
-infixl(>-< / 3).

f(N) -> [1,2,3] >-< [3,4,5] !! N.

!! (L,  I ) -> lists:nth(I, L).
>-<(L1, L2) -> lists:merge(L1, L2).
```

„Almost attributes”
(function/arity)

```
f(N) -> '!!'('>-<'([1,2,3], [3,4,5]), N).

'!!' (L,  I ) -> lists:nth(I, L).
'>-<'(L1, L2) -> lists:merge(L1, L2).
```

Portable funs

Portable funs

Syntax

Allow construction of special fun expressions (marked with a bang) that can be sent through message passing and can be stored in database.

- Fun expressions are allowed to be also of form
  `'fun' '!' FunExpClause {';' FunExpClause} 'end'`
- Applications of such funs should start with a bang

Portable funs

Syntax

Allow construction of special fun expressions (marked with a bang) that can be sent through message passing and can be stored in database.

- Fun expressions are allowed to be also of form
  `'fun' '!' FunExpClause {';' FunExpClause} 'end'`
- Applications of such funs should start with a bang

Semantics

These funs should be able to be sent and received via message passing. Dependencies of the funs should be handled by the compiler/runtime.

Portable funs: what to send?

```
f(X) ->
      Y = g(X, X - 2),
      F = fun!(A, B) -> h(A, B) + X * Y end,
      self() ! F,
      receive
            Fun -> io:format("~p", [!Fun(2,3)])
      end.
```

Portable funs: what to send?

What are dependencies?

```
f(X) ->
      Y = g(X, X - 2),
      F = fun!(A, B) -> h(A, B) + X * Y end,
      self() ! F,
      receive
            Fun -> io:format("~p", [!Fun(2,3)])
      end.
```

Portable funs: what to send?

What are dependencies?

How to represent the fun?
And the dependencies?

```
f(X) ->
      Y = g(X, X - 2),
      F = fun!(A, B) -> h(A, B) + X * Y end,
      self() ! F,
      receive
            Fun -> io:format("~p", [!Fun(2,3)])
      end.
```

Portable funs: what to send?

What are dependencies?

How to represent the fun?
And the dependencies?

```
f(X) ->
      Y = g(X, X - 2),
      F = fun!(A, B) -> h(A, B) + X * Y end,
      self() ! F,
      receive
            Fun -> io:format("~p", [!Fun(2,3)])
      end.
```

How do we invoke a ported fun?

Portable funs: what to send?

What are dependencies?

How to represent the fun?
And the dependencies?

```
f(X) ->
      Y = g(X, X - 2),
      F = fun!(A, B) -> h(A, B) + X * Y end,
      self() ! F,
      receive
            Fun -> io:format("~p", [!Fun(2,3)])
      end.
```

What if portable funs are nested?

How do we invoke a ported fun?

Funs as bytecodes

- No need for compilation on the receiving side
- Easily packed into a binary
- Kind of obfuscation

- VMs need to be compatible (opcodes)
- Not as lazy as could be
- Unused funs remain in the runtime system

```
f(X) ->
        Y = g(X,X-2),
        F = fun!(A,B) -> h(A,B)+X*Y end,
        self() ! F,
        receive
                Fun -> io:format("~p", [!Fun(2,3)])
        end.
```

```
f(X) ->
        Y = g(X,X-2),
        F = fun!(A,B) -> h(A,B)+X*Y end,
        self() ! F,
        receive
                Fun -> io:format("~p", [!Fun(2,3)])
        end.
```

```
f(X) ->
        Y = g(X,X-2),
        F = {'86431211'(), [X, Y], '86431211'},
        self() ! F,
        receive
                Fun -> io:format("~p", [apply_ported(Fun, [2, 3])])
        end.

'86431211'() ->
    binary:encode_unsigned(
        203175963351894316433811063389938421147976187795094109591…).
```

```
f(X) ->
        Y = g(X,X-2),
        F = fun!(A,B) -> h(A,B)+X*Y end,
        self() ! F,
        receive
                Fun -> io:format("~p", [!Fun(2,3)])
        end
```

```
apply_ported({B, A, N}, Args) ->
    case code:is_loaded(N) of
        false -> code:load_binary(N, N, B);
        _     -> ok
    end,
    erlang:apply((erlang:apply(N, portedfun, A)), Args).
```

```
f(X) ->
        Y =
        F =
        self() ! F,
        receive
                Fun -> io:format("~p", [apply_ported(Fun, [2, 3])])
        end.

'86431211'() ->
    binary:encode_unsigned(
        203175963351894316433811063389938421147976187795094109591…).
```

```
f(X) ->
        Y = g(X,X-2),
        F = fun!(A,B) -> h(A,B)+X*Y end,
        self() ! F,
        receive
                Fun -> io:format("~p", [!Fun(2,3)])
        end.
```

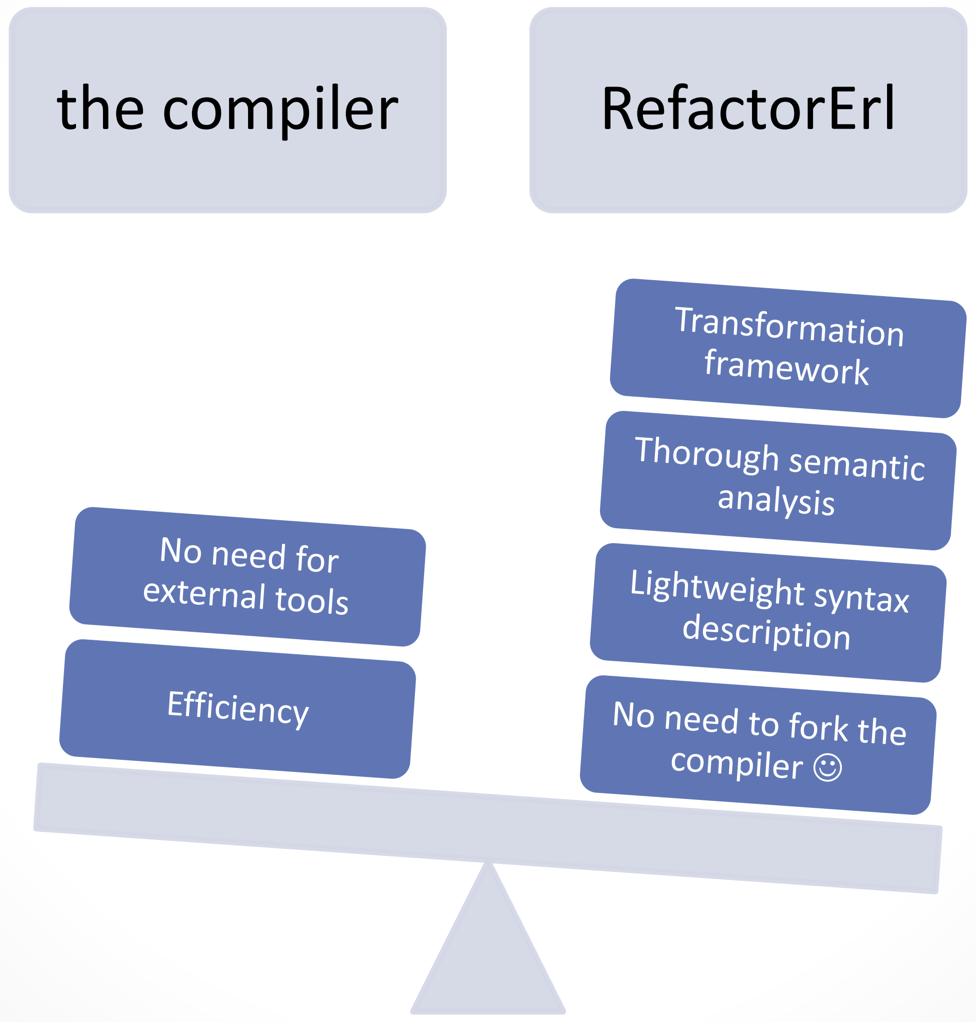
```
f(X) ->
        Y = g(X,X-2),
        F = {'86431211'(), [X, Y], '86431211'},
        self() ! F,
        receive
                Fun -> io:format("~p", [apply_ported(Fun, [2, 3])])
        end.

'86431211'() ->
    binary:encode_unsigned(
        203175963351894316433811063389938421147976187795094109591…).
```

```
f(X) ->
        Y = g(X,X-2),
        F = fun!(A,B) -> h(A,B)+X*Y end,
        self() ! F,
        receive
                Fun -> io:format("~p", [!Fun(2,3)])
        end.
```

```
f(X) ->
        Y = g(X,X-2),
        F = {'86431211'(
        self() ! F,
        receive
                Fun -> i                                      [2, 3])])
        end.

'86431211'() ->
    binary:encode_unsigned(
        203175963351894316433811063389938421147976187795094109591…).
```

```
-module('86431211').
-export([portedfun/2]).

portedfun(X,Y) ->
    fun(A,B) -> h(A,B)+X*Y end.

h(X,Y) -> X/Y.
```

Language extensions made with


the compiler
RefactorErl
Transformation framework
Thorough semantic analysis
Lightweight syntax description
No need to fork the compiler ☺
No need for external tools
Efficiency

Thank you!

refactorerl.com

daniel-h@elte.hu