Intermediate representations of functional programming languages for software quality control

Melinda Tóth, Gordana Rakić

Eötvös Loránd University
Faculty of Informatics

University of Novi Sad
Chair of Computer Science

Aug. 27., 2013
The 13th Workshop on "Software Engineering Education and Reverse Engineering"

Static Analyser Tools

RefactorErl

- Static source code analyser and transformer
- Erlang

SSQSA

- Set of Software Quality Static Analyzers
- Language independent

Static Analyser Tools

RefactorErl

- Code Comprehension
- Refactoring
- Query Language
- Clone detection
- Metrics
- Dependence analysis

SSQSA

- Clone detection
- Metrics
- Software network analysis
- Analyze evolutionary changes

Static Analyser Tools

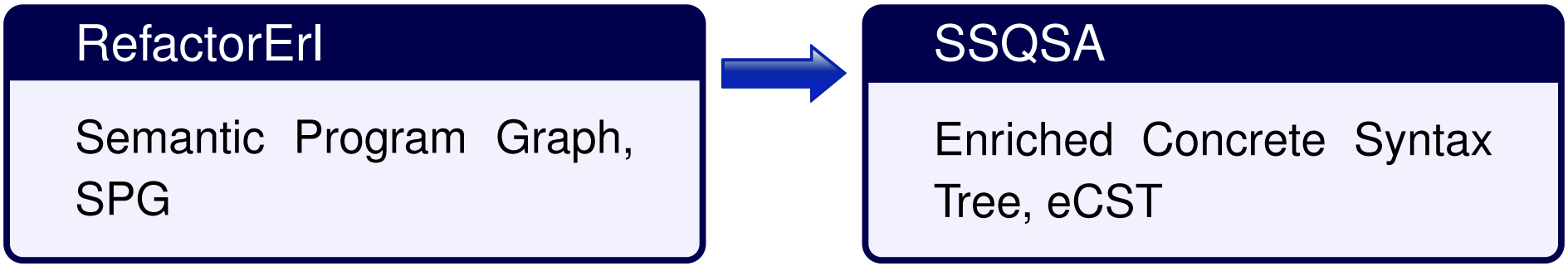
RefactorErl

Semantic Program Graph, SPG

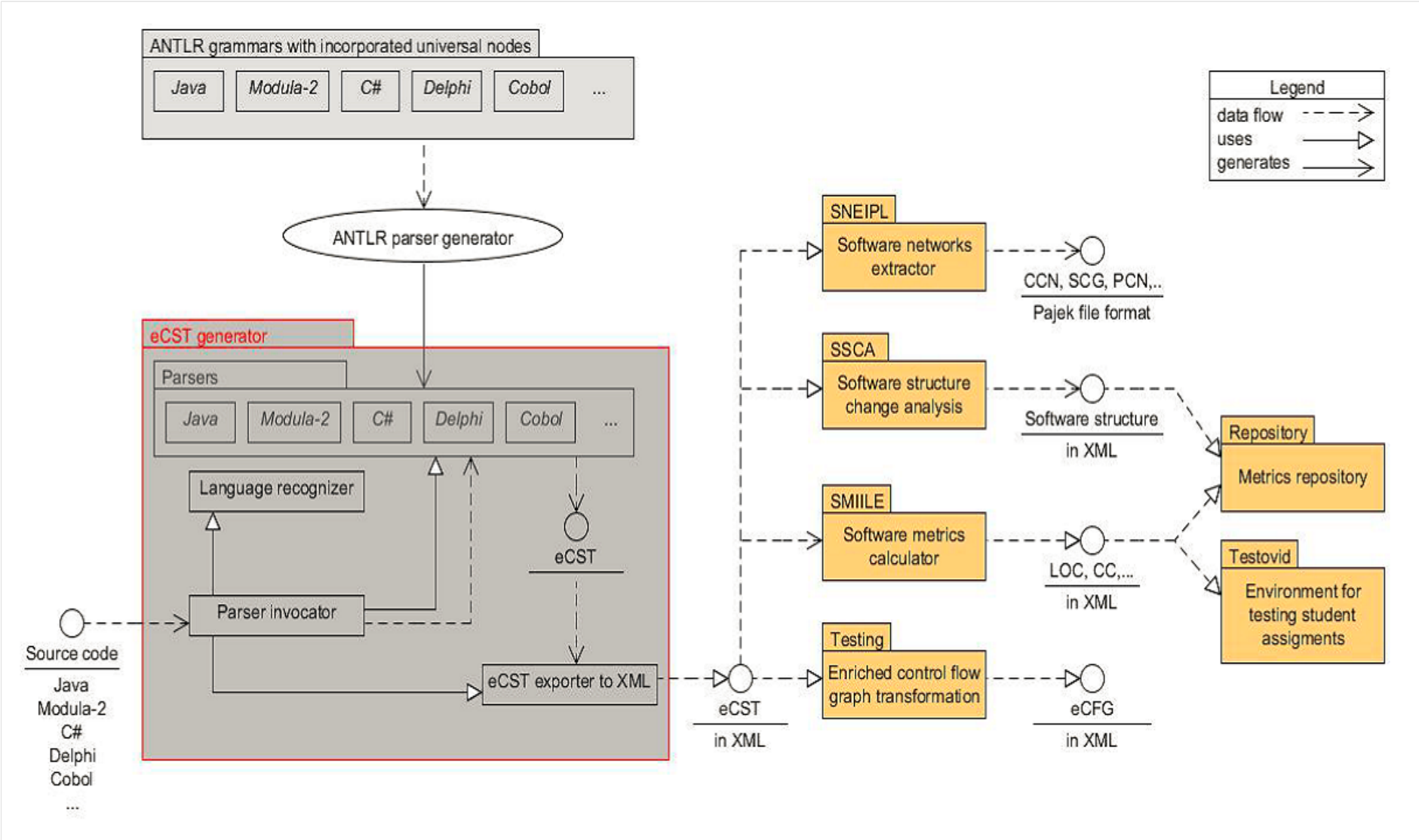
SSQSA

Enriched Concrete Syntax Tree, eCST

Static Analyser Tools



SSQSA


ANTLR grammars with incorporated universal nodes
Java
Modula-2
C#
Delphi
Cobol
...
ANTLR parser generator
eCST generator
Parsers
Java
Modula-2
C#
Delphi
Cobol
...
Language recognizer
eCST
Parser invocator
eCST exporter to XML
Source code
Java
Modula-2
C#
Delphi
Cobol
...
eCST
in XML
Legend
data flow
uses
generates
SNEIPL
Software networks extractor
CCN, SCG, PCN,..
Pajek file format
SSCA
Software structure change analysis
Software structure
in XML
SMIILE
Software metrics calculator
LOC, CC,...
in XML
Testing
Enriched control flow graph transformation
eCFG
in XML
Repository
Metrics repository
Testovid
Environment for testing student assigments

Erlang

- Functional
- Highly concurrent
- Distributed
- Fault-tolerant

- Supervisor
- Soft real-time
- Hot code swap
- Erlang VM

History

- Started in 2006
- Software Technology Lab (4*4 ECTS) since 2008
  - One of the 8 labs
  - 10-20 students (BSc, MSc, PhD)
- ELTE-Ericsson Software Technology Lab since 2011

RefactorErl

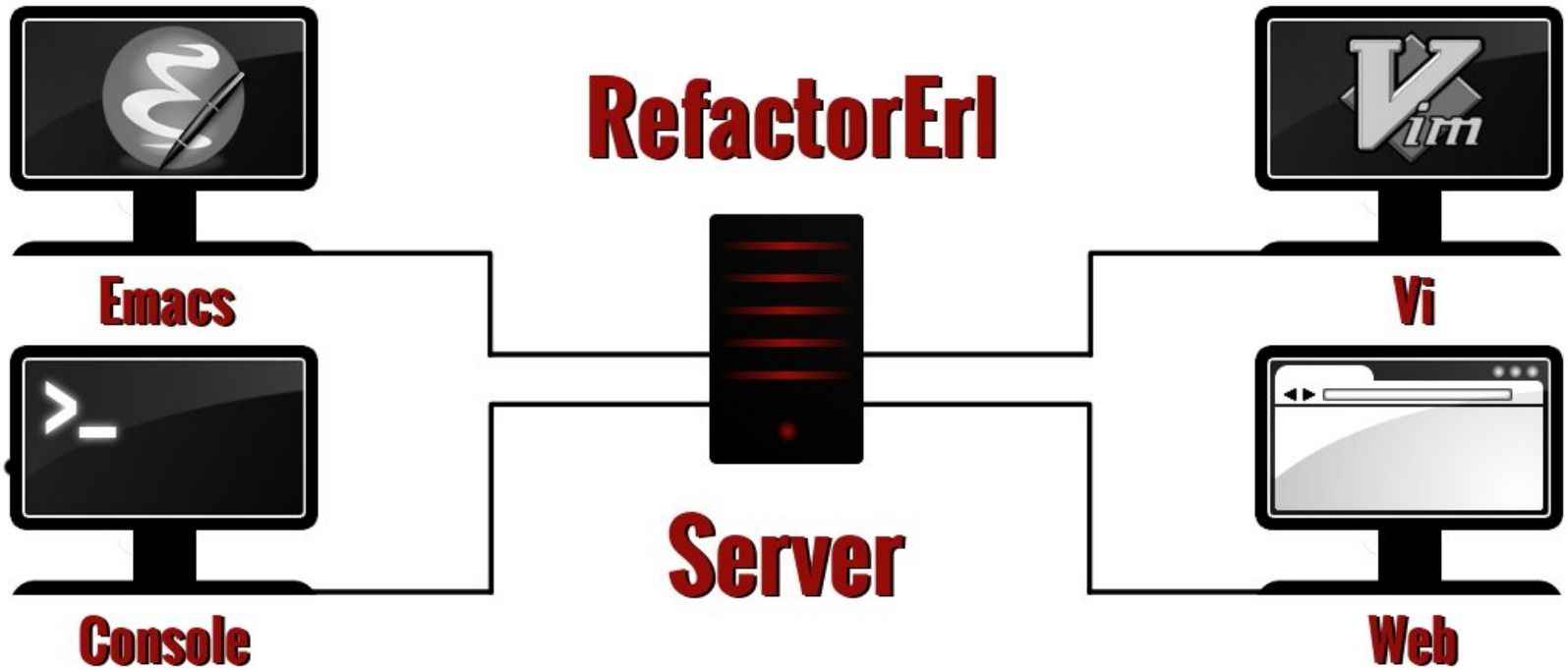
- Static source code analyser and transformer tool for Erlang
  - Refactoring – less error prone & fast
  - Program comprehension
- Support in everyday work & debugging & complex tasks, e.g.
  - Rename a function, search definition
  - Find the value of a variable
  - Program comprehension, component relation detection
  - Program restructuring

RefactorErl (cont.)

- Platform for source code transformations – 24 implemented refactorings
  - Rename/Move definition
  - Expression structure
  - Function interface
- Structural source code analysis – Clustering
- Support program comprehension
  - Call graph visualisation
  - Dependency analysis
  - Semantic Query Language / Metric Query Language

Knowledge sharing

User Interfaces



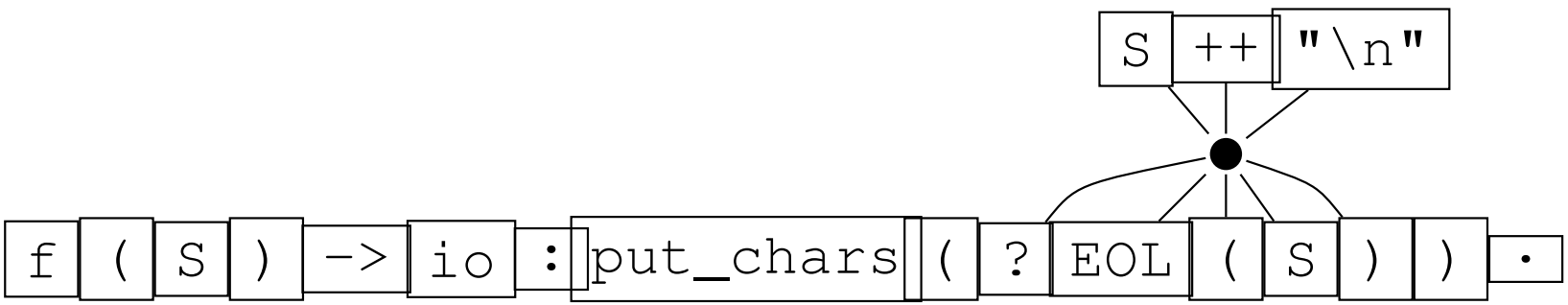
Semantic Program Graph

1. Lexical level
   - Tokens
   - Preprocessing
   - Comments, whitespace
2. Syntactic level
   - Abstract Syntax Tree
   - Files
3. Semantic level
   - Module, function, record, variable nodes
   - Links to definition and reference points

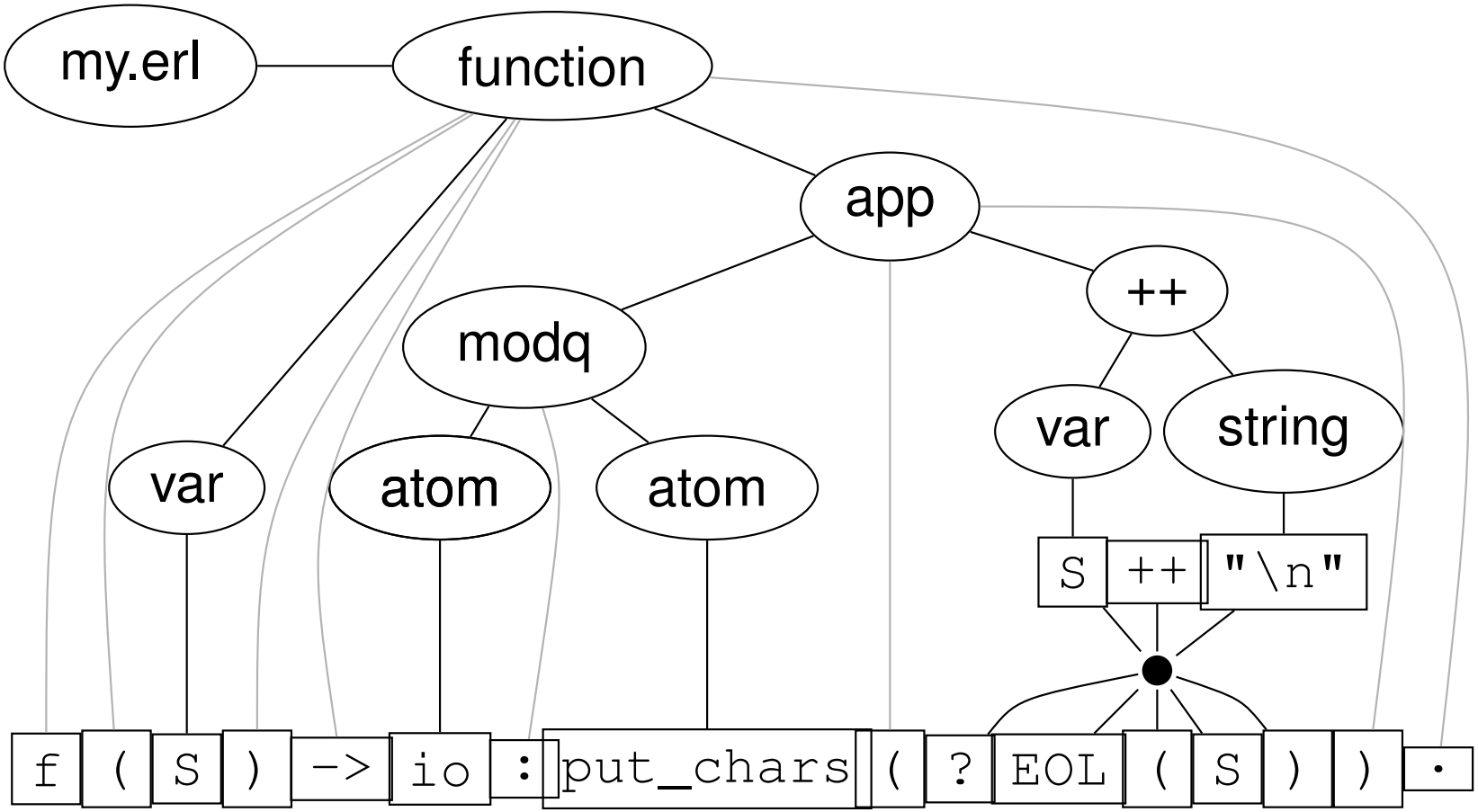
```
-module(my).
-define(EOL(X), X ++ "\n").
f(S) -> io:put_chars(?EOL(S)).
```

```
-module(my).
-define(EOL(X), X ++ "\n").
f(S) -> io:put_chars(?EOL(S)).
```

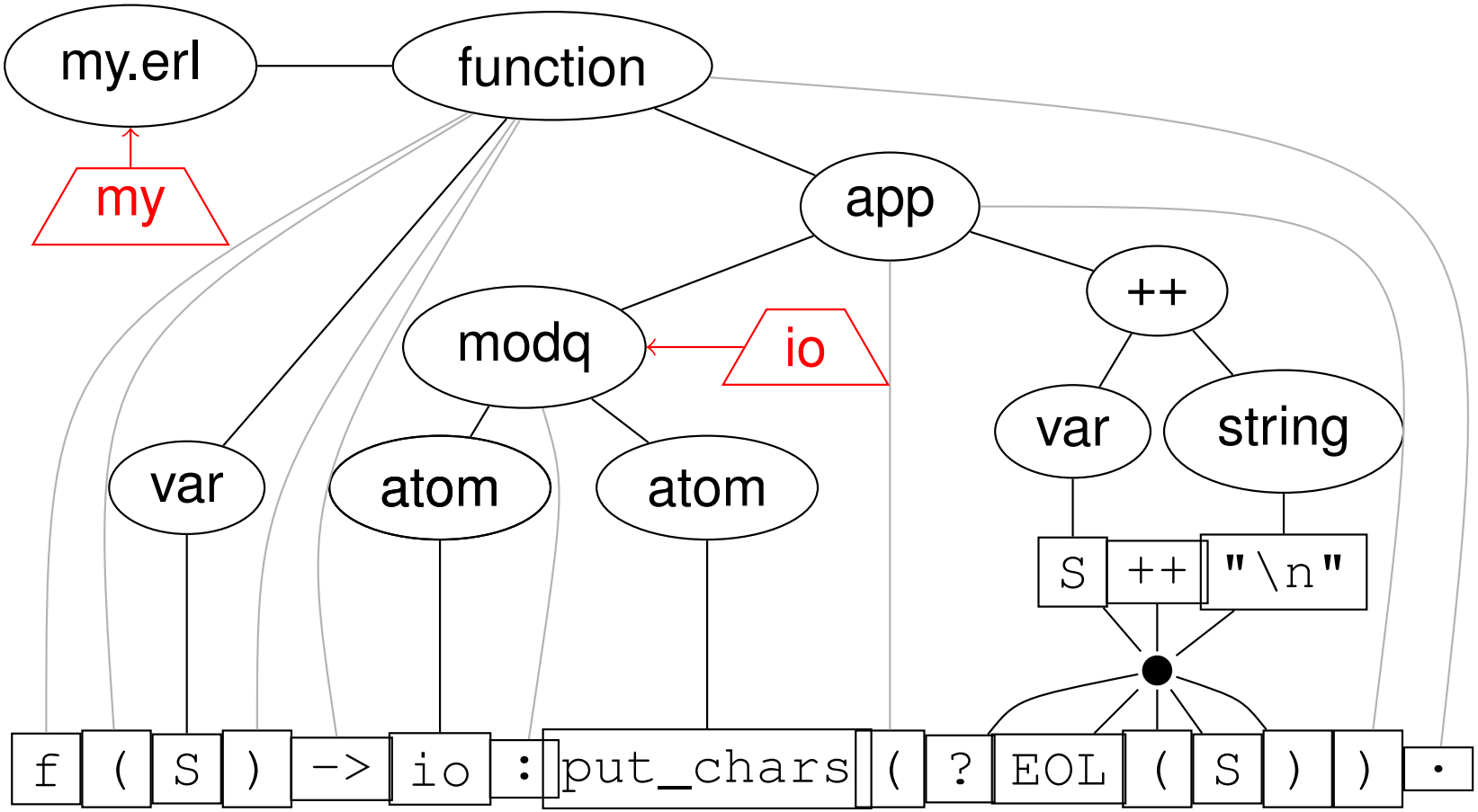
```
-module(my).
-define(EOL(X), X ++ "\n").
f(S) -> io:put_chars(?EOL(S)).
```



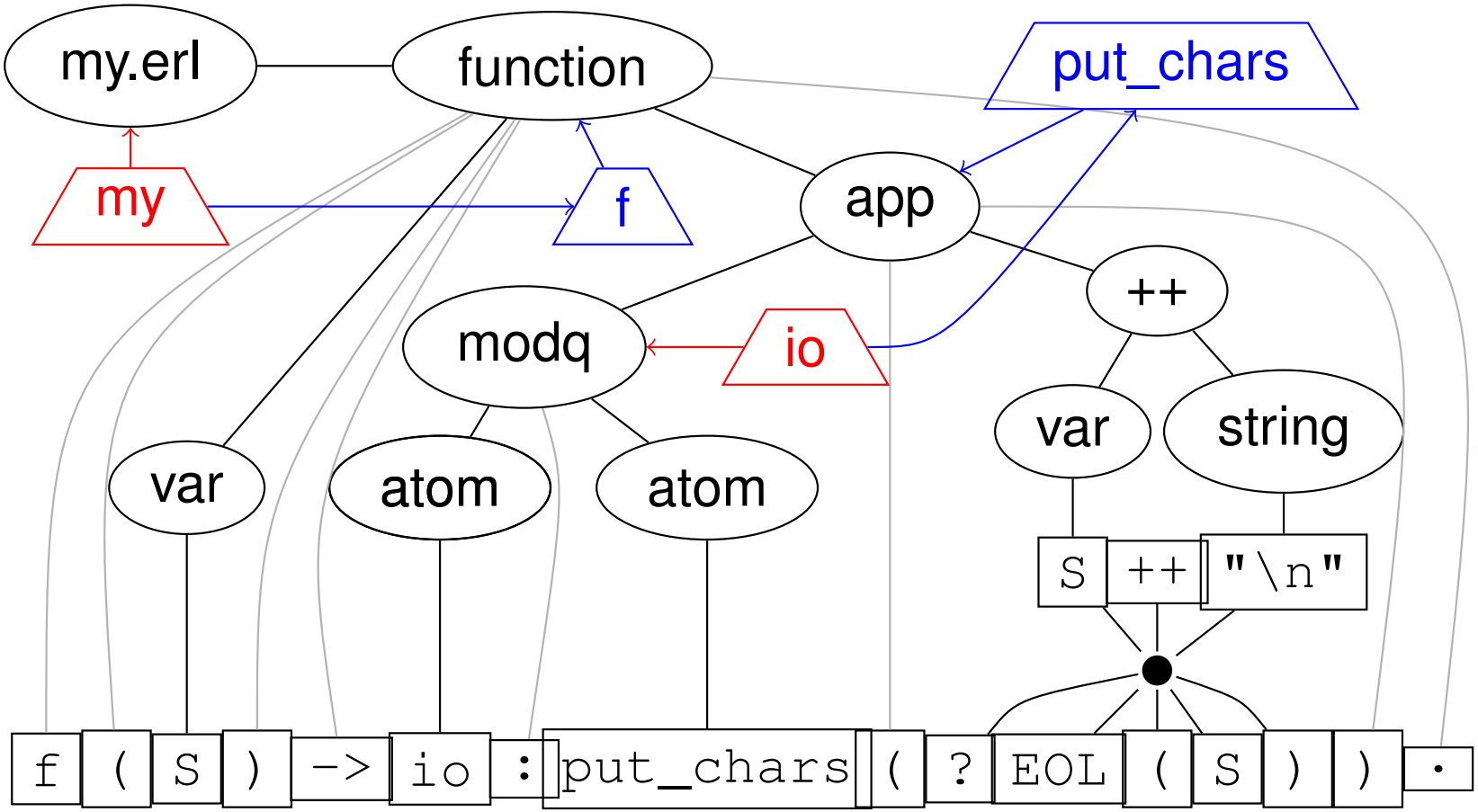
```
-module(my).
-define(EOL(X), X ++ "\n").
f(S) -> io:put_chars(?EOL(S)).
```



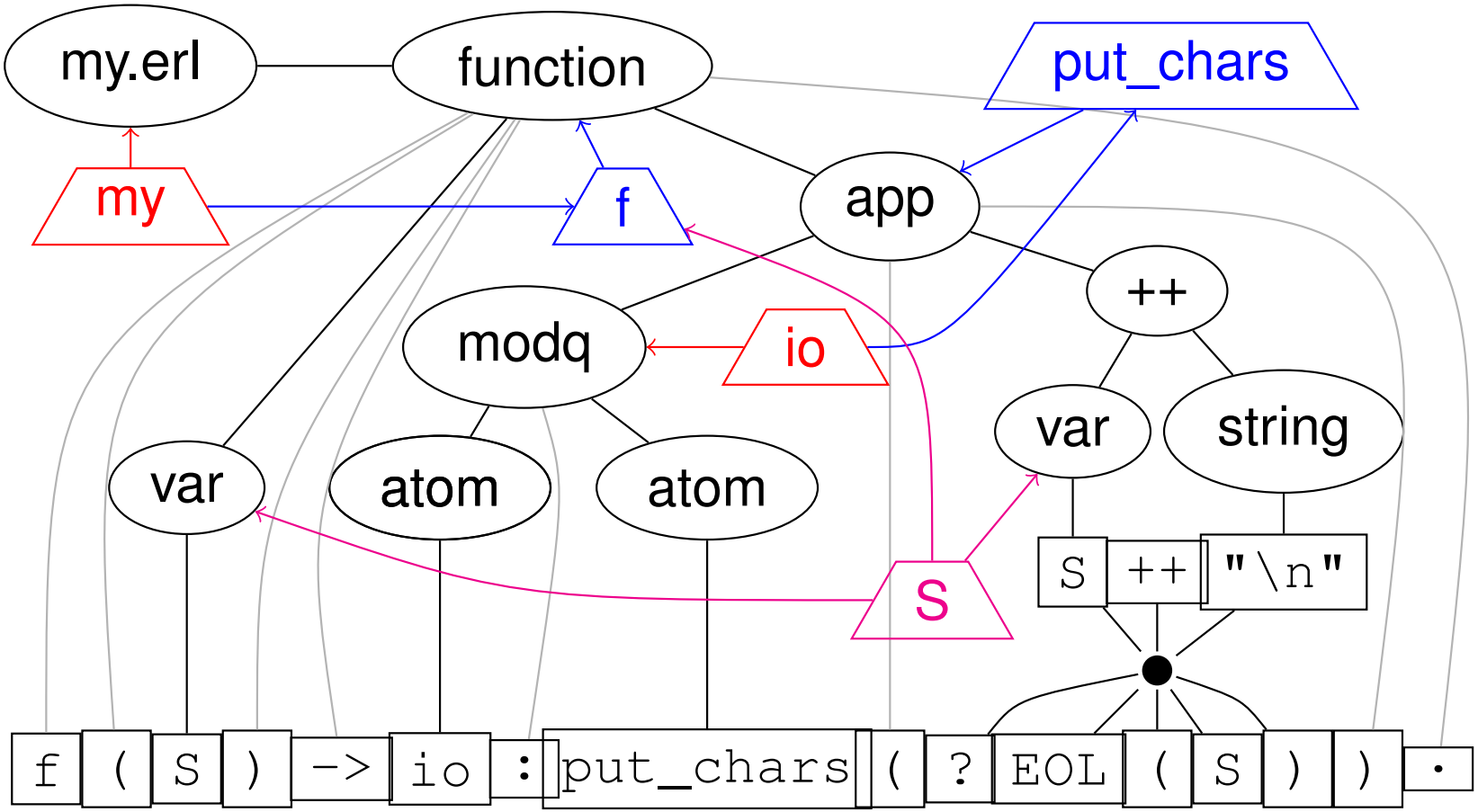
```
-module(my).
-define(EOL(X), X ++ "\n").
f(S) -> io:put_chars(?EOL(S)).
```



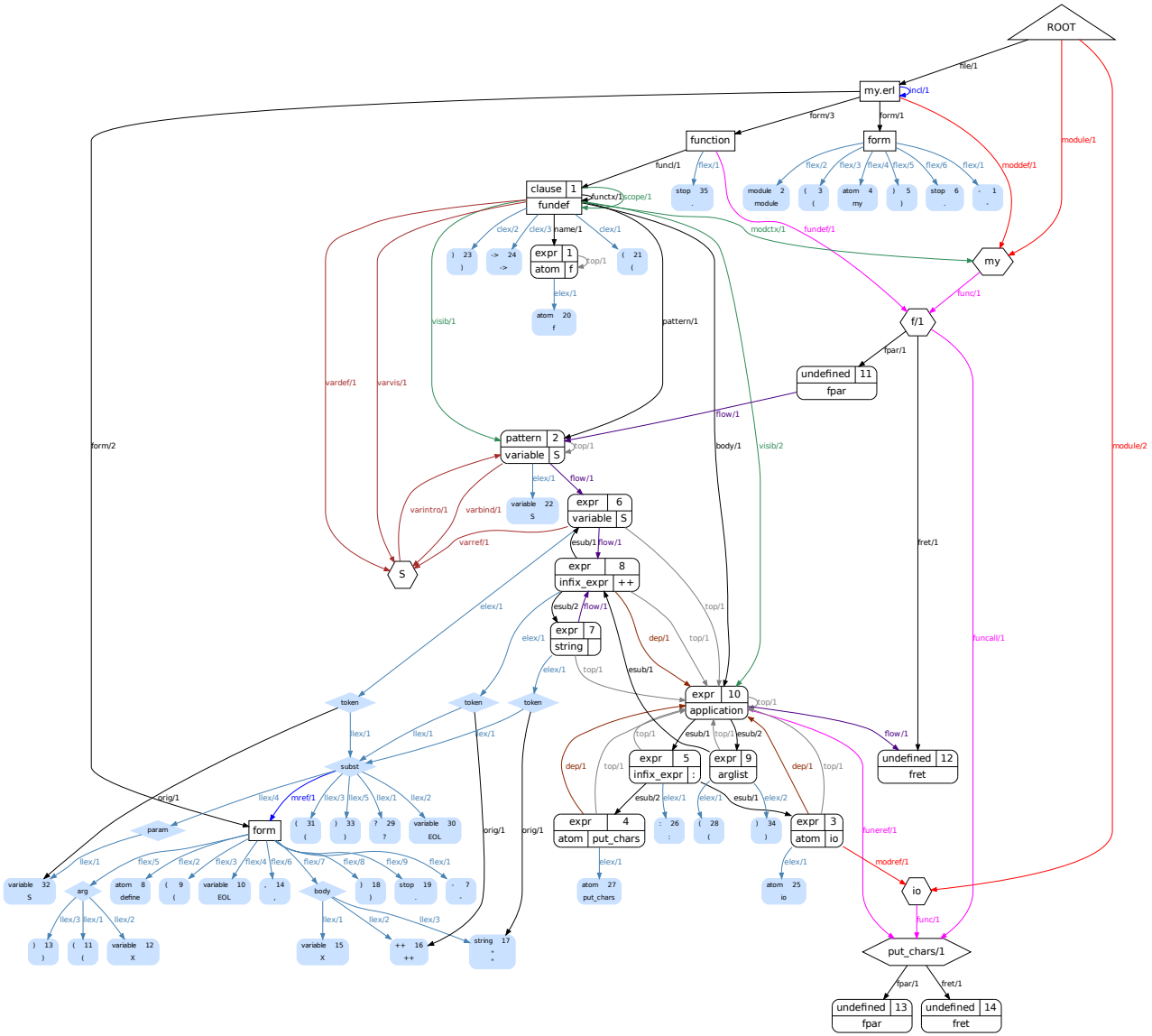
```
-module(my).
-define(EOL(X), X ++ "\n").
f(S) -> io:put_chars(?EOL(S)).
```



```
-module(my).
-define(EOL(X), X ++ "\n").
f(S) -> io:put_chars(?EOL(S)).
```



Example graph



Analysis Details

- Extended syntax description
  - Defines the representation
  - Source for parser, lexer, and token updater
- Semantic Analyser framework
  - Extensible, modular structure
  - Works on syntactic subtrees (incremental)
  - Asynchronous parallel execution
  - Side-effect analysis, data-flow analysis, dynamic function call analysis

Mapping

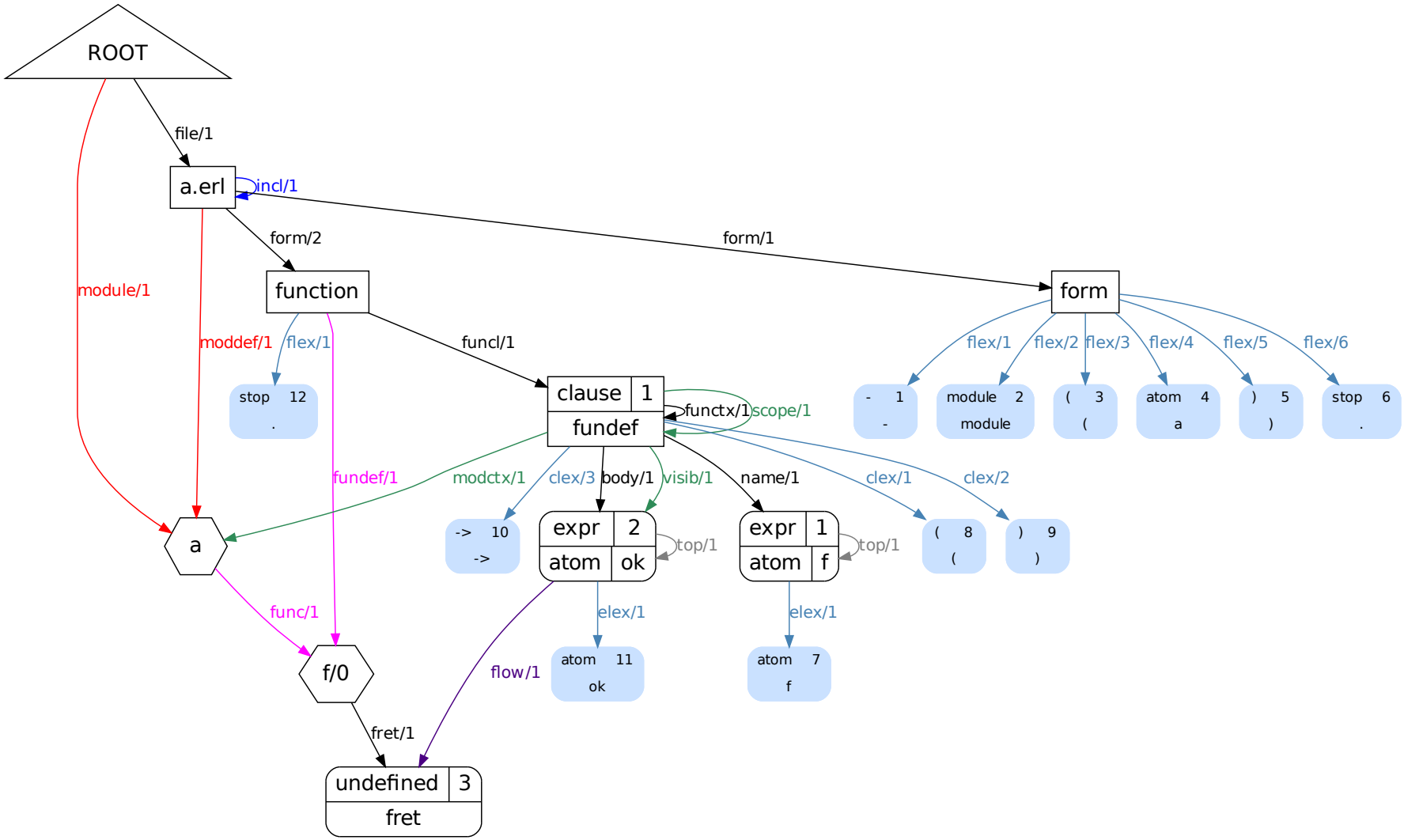
Attila Páter-Részeg, TDK Thesis,
Scientific Student Association Conference, June 2013

- convert: $SG \rightarrow ECST$
- $convert_N$: $SG_N \rightarrow ECST_M$
  - file
  - form
  - ...
  - COMPILATION_UNIT
  - FUNCTION_DECL
  - ...

Simple Erlang Module

```
-module(a).

f() ->
    ok.
```

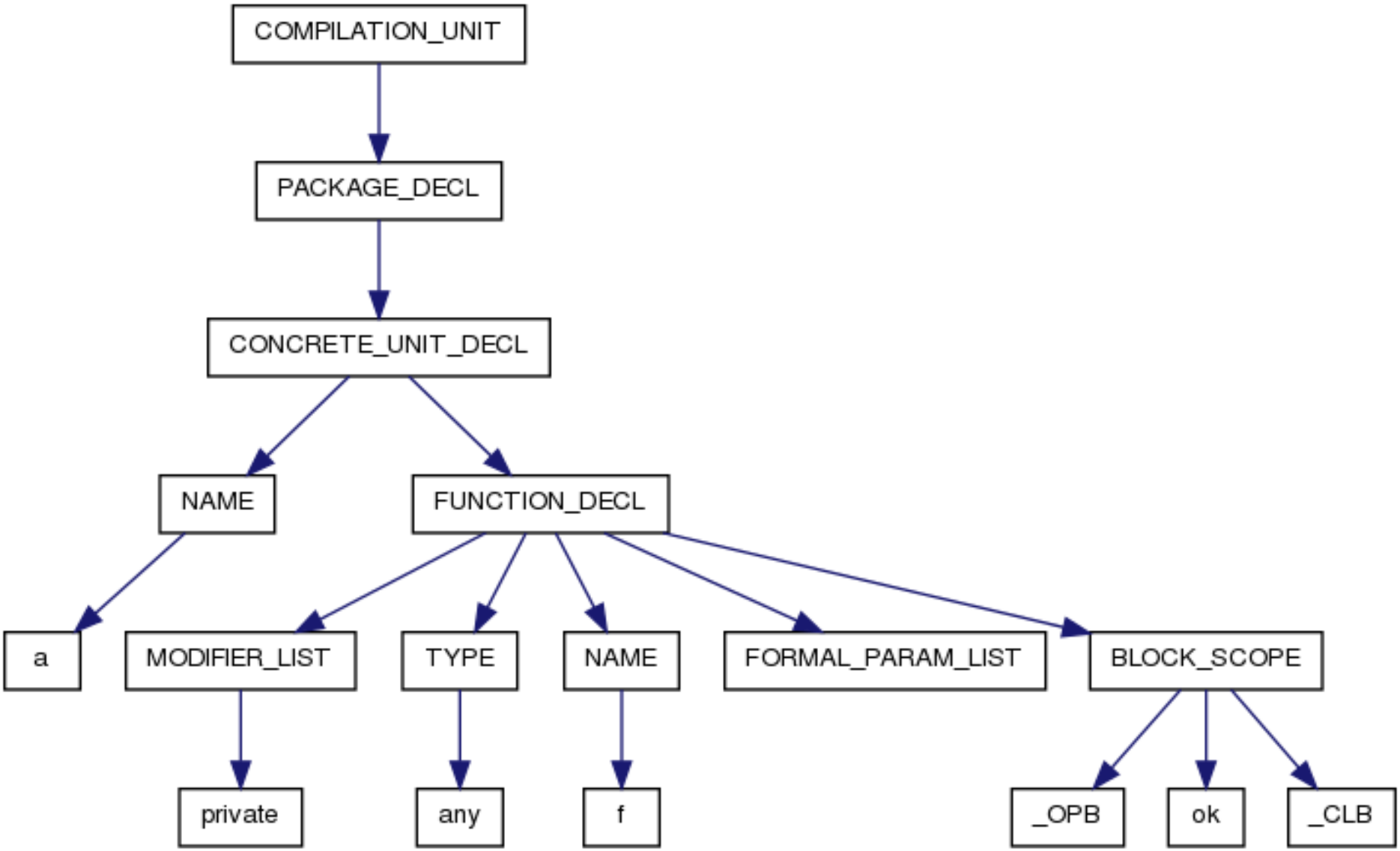
SPG representation for module a



XML representation for module `a`

```
▼<childElement>
  <token column="-1" index="-1" line="0" text="FUNCTION_DECL" type="115"/>
 ▼<childElement>
    <token column="-1" index="-1" line="0" text="MODIFIER_LIST" type="136"/>
   ▼<childElement>
      <token column="-1" index="-1" line="0" text="private" type="51"/>
    </childElement>
  </childElement>
 ▼<childElement>
    <token column="-1" index="-1" line="0" text="TYPE" type="119"/>
   ▼<childElement>
      <token column="1" index="0" line="1" text="any" type="0"/>
    </childElement>
  </childElement>
 ▼<childElement>
    <token column="-1" index="-1" line="0" text="NAME" type="118"/>
   ▼<childElement>
      <token column="1" index="0" line="5" text="f" type="0"/>
    </childElement>
  </childElement>
 ▼<childElement>
    <token column="-1" index="-1" line="0" text="FORMAL_PARAM_LIST" type="116"/>
  </childElement>
 ▼<childElement>
    <token column="-1" index="-1" line="0" text="BLOCK_SCOPE" type="109"/>
   ▼<childElement>
      <token column="0" index="0" line="0" text="{" type="0"/>
    </childElement>
   ▼<childElement>
      <token column="5" index="0" line="6" text="ok" type="0"/>
    </childElement>
   ▼<childElement>
      <token column="0" index="0" line="0" text="}" type="0"/>
    </childElement>
  </childElement>
</childElement>
```

eCST representation for module `a`



Example: $convert_{function}$(x) = y

Erlang source code

```
f(      ) -> ok.
```

Imperative source code

```
private any f( ){
ok;
}
```

Example: $convert_{function}$(x) = y

Erlang source code

```
-export([f/0]).
f(      ) -> ok.
```

Imperative source code

```
public any f( ){
ok;
}
```

Example: $convert_{function}$(x) = y

Erlang source code

```
-export([f/2]).
f(A, B) -> ok.
```

Imperative source code

```
public any f(any Var1, any Var2){
ok;
}
```

Example: $convert_{function}$(x) = y

Erlang source code

```
-export([f/2]).
f(A, A) -> ok.
```

Imperative source code

```
public any f(any Var1, any Var2){
if(Var1 == Var2) ok;
}
```

Example: $convert_{function}$(x) = y

Erlang source code

```
-export([f/2]).
f(A, A) -> ok;
f(_, _) -> nok.
```

Imperative source code

```
public any f(any Var1, any Var2){
if(Var1 == Var2) ok;
else nok;
}
```

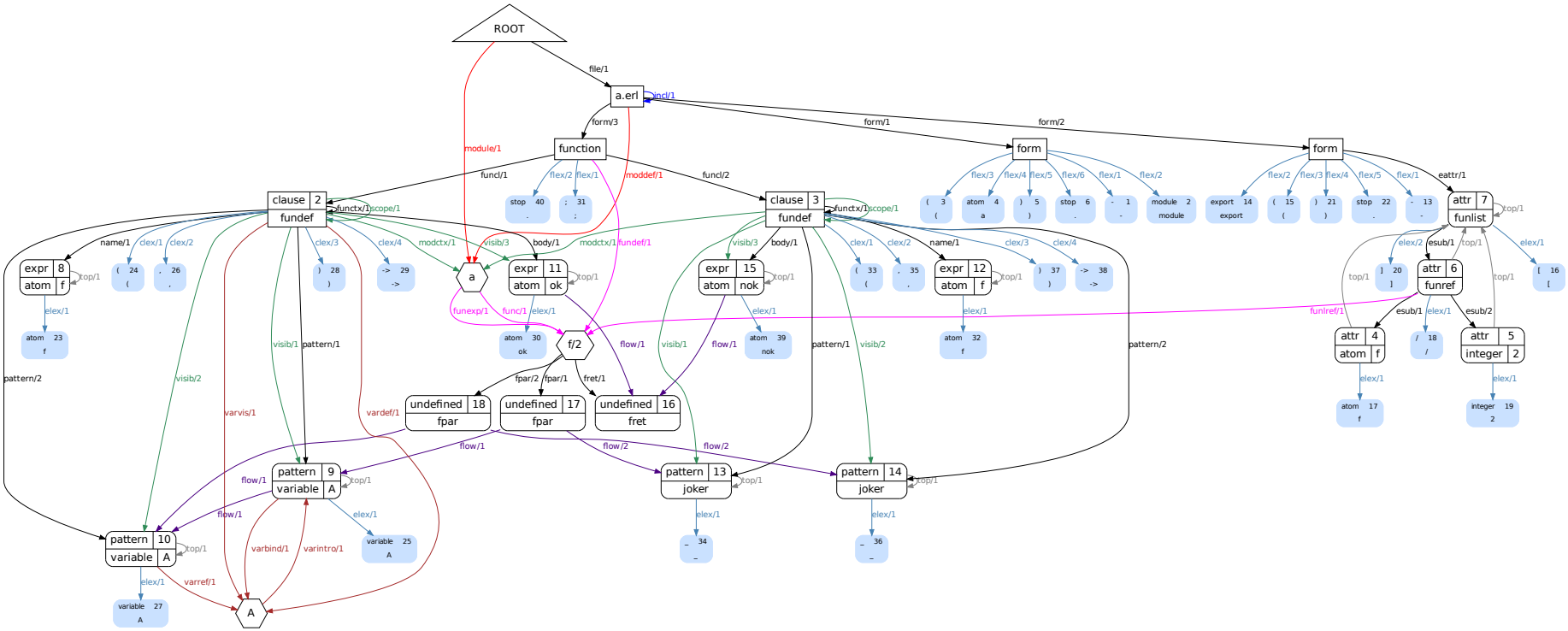
The steps of the mapping

- Exported $\rightarrow$ public/private
- Return value, type $\rightarrow$ any
- Formal parameter list $\rightarrow$ new variable
- Pattern matching, guards $\rightarrow$ branching conditions

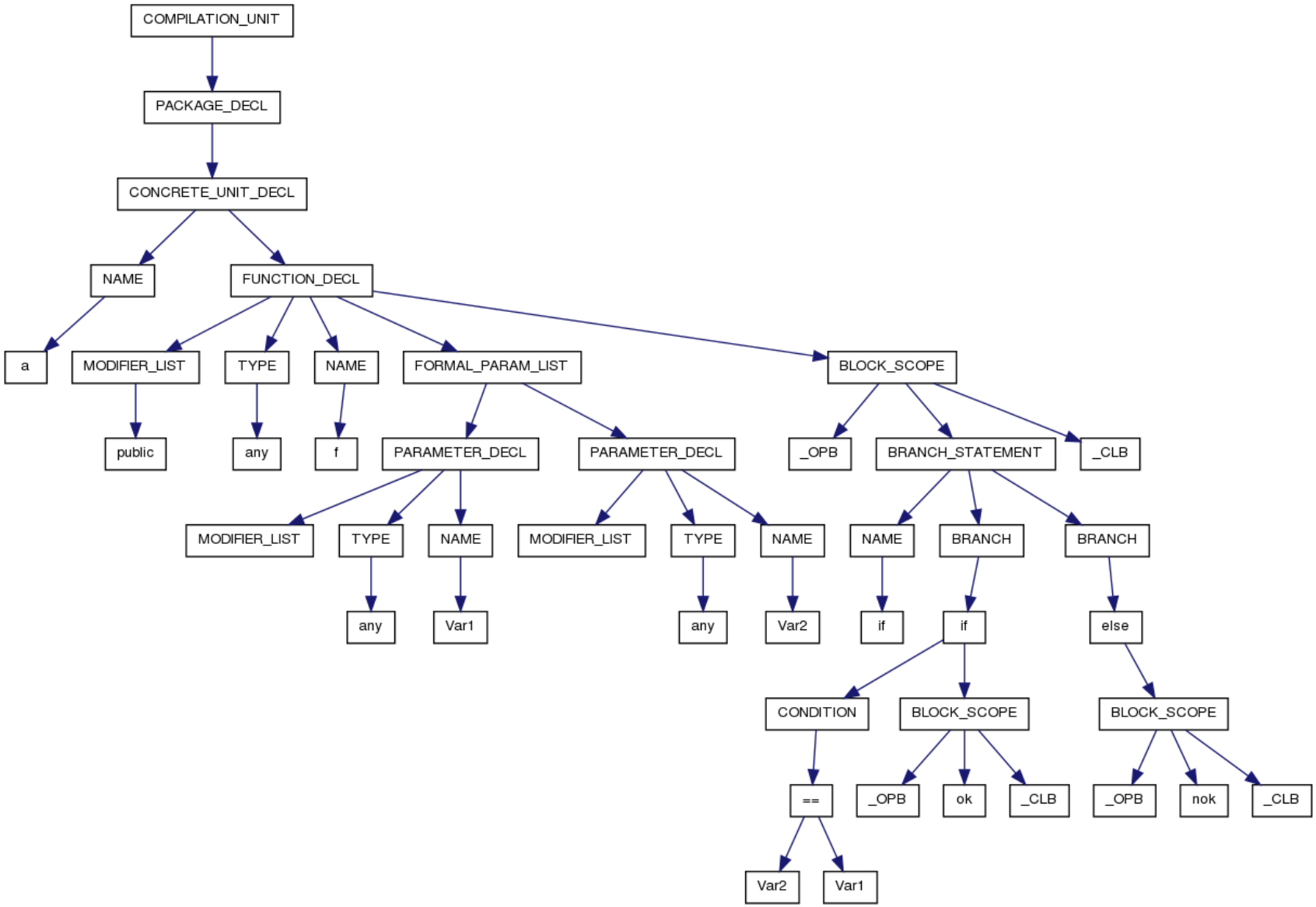
Simple Erlang Module

```
-module(a).

-export([f/2]).

f(A, A) ->
    ok;
f(_, _)->
    nok.
```

SPG representation for module a



eCST representation for module `a`



Results

- Defined Mapping
  - Syntax driven, recursive
- Prototype implementation
  - Available through RefactorErl

Future Work

- Refinements of the mapping
- Direct connection between the two tools
- Validating the results, example:
  - Compare the value of metrics
  - Evaluate the result