Rascal Requirements and Design Document

Paul Klint
Tijs van der Storm
Jurgen Vinju

Table of Contents

Rationale 2
Introduction 2
Requirements 3
Rascal at a glance 4
  Modules 4
  Types and Values 5
  Annotations 5
  Variables 6
  Functions 6
  Patterns 7
  Expressions 8
  Statements 9
  Visitor definitions (UNDECIDED and INCOMPLETE) 10
  Failure and side-effects 11
Examples 12
  Booleans 12
  Abstract Graph datatype 14
  Tree traversal 15
  Substitution in Lambda 17
  Renaming in Let 18
  Renaming in Let using globals 19
  Concise Pico Typechecker 20
  Pico evaluator 21
  Pico evaluator with globals 23
  Pico control flow extraction 25
  Pico use def extraction 25
  Pico uninitialized variables 26
  Pico common subsexpression elimination 26
  Pico constant propagation 27
  Pico Reaching definitions 28
  Symbol table with scopes 29
  Innerproduct 30
  Bubble sort 30
  Generic Bubble sort [under discussion] 31
  Read-Eval-Print Loop (REPR) 31
Syntax Definition 32
Protyping/implementation of Rascal 32
Rascal advertisements 33
  Generic arguments 33
  Rascal for ASF+SDF programmers 34
  Rascal for imperative and object-oriented programmers 35

Rascal implementation ideas 36
Data structures 36
Interpreter 37
Compiler 37
Issues 38
Graveyard 39
Mapping features to datatypes 39
Outdated examples 40
Generating Graph files in Dot format 40
Integration with Tscripts (Outdated) 40
Introduction 40
Requirements 40
Different styles of Type Declarations 41
Global Flow of Control 41
Modularization 42

Note

This document is a braindump of ideas that is slowly converging to a coherent design. See the section called “*Issues*” [38] for the issues that have to be resolved.

Rationale

In the domain of software analysis and transformation, there exists a phletora of domain-specific languages for defining grammars, rewrite rules, software analysis and the like. So why embark on the design of yet another DSL in this area? We see the following arguments for this:

- Many existing DSLs are based on more or less exotic concepts that do excite researchers but are frightening for users without an appropriate research background.
- The notation used in many DSLs is uninviting to say the least.
- The scope of DSLs is usually narrow (this is where the word "domain-specific" kicks in). The tasks involved in carrying out a code analysis or renovation project require several DSLs. Integration between these DSLs is insufficient.
- As designers of various DSLs (ASF+SDF, Tscript, Rscript, ...) we have seen the positive as well as the negative side of DSLs. We certainly know howto implement DSLs in this area.
- We see an opportunity for a user-friendly, conceptually high-level, and rich DSL for all tasks related to software analysis and transformation.

With this background and an essential dose of optimism, we embark on this trip ... .

Introduction

The goals of the envisaged language (with working name Rascal) are:

- Providing a successor of ASF+SDF that has of all its benefits and fixes all of its shortcomings.
- Separating pure syntax definitions (SDF) from function definitions.
- Easy syntax-directed analysis of programming languages.
- Easy fact extraction.
- Easy connection of fact extraction with fact manipulation and reasoning.
- Easy feedback of analysis results in source code transformation.

- Efficient and scalable implementation.
- Unsurprising concepts, syntax and semantics for a wide audience. Where possible we will stay close to C and Java notation.

Many of the above goals are to a certain extent already met in the current design of ASF+SDF, and the current design of RScript. What is missing is the connection (and to be honest: an efficient implementation of relational operators). Alas, any bridge between the two languages is both complex to manage and an efficiency bottleneck. This work is an attempt to consolidate this engineering trade-off. This basically means that we include most features of the RScript language into ASF+SDF. Although we take these languages as conceptual starting point, Rascal is a completely new design that has an imperative semantics at it's core rather than a functional semantics. As a whole, Rascal is a simpler but more expressive language.

Requirements

- R1: Runtime speed: large-scale analysis of facts is expensive (frequently high-polynominal and exponential algorithms). A factor speedup can mean the difference between a feasible and an unfeasible case.
- R2: Old ASF+SDF programs are translatable to Rascal.
- R3: Edit Rascal and SDF and compile complex programs within a few minutes maximally (parsetable generation is a major bottleneck in current ASF+SDF. This needs to be fixed.)
- R4: Concrete syntax: for readability and easy parsing of a wide range of source languages.
- R5: (**dropped**) Functional (no side-effects), dropped due to incompatibility with R7, R6
- R6: File I/O (contradicts R5).
- R7: Easily accessible fact storage (similar to a heap, but remember R5 and the details of backtracking).
- R8: List matching (because of R2, influences R7, but also very handy for manipulating lists in concrete syntax).
- R9: Nesting of data-structures: relations can be nested to model nested features of programming languages (such as scoping), allowing to factor out common code.
- R10: Syntax trees can be elements of the builtin data-structures (but not vice versa).
- R11: Try to keep features orthogonal: try to keep the number of ways to write down a program minimal, this is not a law since other requirements take precedence
- R12 (**new**): Minimize possible syntactic ambiguities; resolve them by type checking.
- R13:(**new**): Integrates well with refactoring infra-structure (i.e. can provide appropriate interfaces with pre-condition checking, previews and commits as found in interactive refactoring contexts)
- R14 (**new**): no 'null' values, preventing common programming errors
- R15 (**new**): all values immutable, preventing common programming errors and allowing for certain kinds of optimizations
- R16 (**new**): should be able to match and construct strings using regular expressions (for making the simpler things simple, if you can do without a grammar, why not?)
- R17 (**new**): can get/set data from databases, such as the pdb from Eclipse IMP, but possibly also from ODBC/JDBC data sources.
- R18 (**new**): type safe, but flexible. We want a type system that prevents common programming errors, but still allows ample opportunity for reuse.

- R19 (**new**): syntax safe, programmers should not be allowed to construct programs that are syntactically incorrect w.r.t a certain context-free grammar.
- R20 (**new**): backtracking safe, programmers should not have to deal with the mind boggling feature interactions between side-effects and backtracking.
- R21 (**new**): traceable/debuggable, programmers should be able to easily trace through the execution of a Rascal program using the simplest of debugging tools, like printf statements, and the use of a simple debugging interface which allows to step through the source code and inspect values in a transparent fashion.
- R22 (**new**): minimize the use of type inference, such that the programmer must always declare her intentions by providing types for functions, data-types and variables. This makes debugging easier and providing clear error messages too. When variables are implicitly bound by pattern matching or related functionality, exceptions to this requirement might be made in favor of conciseness.
- R23 (**new**): allow the implementation of reusable modules and functions (i.e. parametric polymorphism and or functions as parameters).
- R24 (**new**): we need something like rewrite rules for implementing data-types that are always canonicalized/normalized. For some analysis algorithms this allows the programmer to implement domain specific optimizations over plain relational calculus or tree visiting that actually needed for scalability.

Rascal at a glance

Rascal consists of the following elements:

- Modules to group definitions, proving scopes and visibility constructs
- A type system and corresponding values, providing parameterized types, polymorphic functions and higher-order parameters.
- Variables to associate a name with a value in some scope.
- Parameterized functions.
- Abstract, regular expression and syntax patterns to deconstruct (match) values and to construct (make) them.
- Expressions provide the elementary computations on values.
- Statements for providing structured control flow and more advanced control flow in computations, such as visitors and fixed point computations

These elements are summarized in the following subsections.

Modules

Modules are the organizational unit of Rascal. They may:

- Import other modules (either Rascal modules or SDF modules) using `import`.
- Import Java modules (for the benefit of functions written in Java) using `java-import`.
- Define datatypes, subtypes, or rules and functions.
- Be parameterized with the names of formal types that are instantiated with an actual type when the module is imported.
- Contain a main function that is the starting point of execution:

```
public int main(list[str] argv) { ... }
```

Functions, subtypes and datatypes may be either private to a module or public to all modules that import the current module. Rules are always public and globally applied.

Modules introduce a namespace and qualified names may be used to uniquely identify elements of a module from the outside. Inside the module, this qualification is implicit.

Types and Values

The type system (and notation) are mostly similar to that of Rscript, but

- Symbols (as defined by an SDF module) are also types.
- There are built-in types (`bool`, `int`, `str`, `loc`) that have corresponding values.
- There is a single root to the type hierarchy, everything is a "`value`".
- Relations and tuples can have optional column names.
- There is subtyping (as opposed to aliasing of types in Rscript)
- All syntactic types are a subtype of the type `tree` that corresponds to UPTR. Up casts from a subtype to an enclosing type are automatic; Down casts require a run-time check.
- Datatype declarations may introduce new structured types.
- Types may include type variables like `&T` as in Rscript.

As a design strategy we try to offer the option to leave out as many type indications as possible.

The root of the type system is the type named "`value`". It has the following subtypes:

- `bool`
- `int`
- `str`
- `loc`
- `void`
- lists, sets, tuples, maps (single valued binary brelations) and relations of values.
- all structures defined with a datatype definition. One of the subtypes is called "`tree`".
  - Type `tree` is the type that corresponds with concrete syntax trees (UPTR in the case of the Meta-Environment). Type `tree` has as subtype all types that are derived from SDF definitions, i.e., all notions that are defined using a grammar.
  - The type `tree` is "special" in the following sense:
    - Parsers generate values of type `tree`.
    - Although the type `tree` can be defined in Rascal, its definition is built-in in order to preserve the consistency with the parser.

Annotations

Annotations may be associated with any value and are represented by a mapping of type `map[str,value]`, i.e., annotation names are strings and annotation values are arbitrary values of type `value`. The following library functions handle annotations:

```
%% Test whether a named annotation exists.
```

```
public bool has-annotation(&T Subject);

%% Get the value of a named annotation
%% Note: get-annotation throws an exception when the named
%% annotation does not exist.
public value get-annotation(&T Subject, str Name);

%% Get all annotations
public map[str,value] get-annotations(&T Subject);

%% Set the value of a named annotation
public &T set-annotation(&T Subject, str Name, value AValue);

%% Set all annotations
public &T set-annotations(&T Subject, map[str, value] Annos);
```

Variables

Variables are names that have an associated scope and in that scope they have a value. A variable declaration consists of a type followed by the variable name and---depending on the syntactic position---they are followed by an initialization. There are no null values, which implies that all variables must be initialized at declaration time. Also, this implies that all expressions must return a value. Especially for functions, this means that all execution paths of a function must have a return statement.

Variables may be introduced at the following syntactic positions:

- As formal parameters of a function. Their scope is the function and they get their initial value when the function is called.
- Local variables in a function body are declared and initialized. Their scope is the function body.
- Variables in patterns. For patterns in match positions, variables are initialized during the match and their scope is the rule in which they occur. For patterns in make positions, the values of variables are taken from the local scope.
- For variables ntroduced by pattern matching in conditional statements (if-then, and while), if the condition succeeds, the scope of the variables are the block of code that is executed conditionally.
- Variables in anti-patterns are never visible, but nevertheless their names are reserved in the scope that they would have had when the pattern was a normal positive matching pattern.
- Variables that are introduced by generators in comprehensions or for statement, have the comprehension, respectively, for statement as scope.
- Global variables are declared and ALWAYS initialized (**new**) at the top level of each module. Functions that use a global variable have to be explicitly declare it as well. The value of a global variable can be used and replaced by all functions that have locally declared it.

We will see below that there are certain contexts in which assignments to variables are undone in the case of failure.

Functions

A function declaration consists of a visibility declaration, the keyword `fun`, result type, a function name, typed arguments and a function body. Functions without a result type have type `void`.

A visibility declaration is one of the keywords `public` or `private` (default).

A function body is a list of statements, optionally separated by semi-colons. Each unique control flow path through a function must have a return statement, such that each function always returns a proper value.

Functions can raise exceptions but these are not declared as part of the function signature.

Functions can later be extended [STILL UNDER DISCUSSION]

- The keyword `extend` before a function declaration extends a previously defined function with the same signature.
- Local declarations in the extension function are added to the original function.
- If both bodies consist of a `switch` or `visit` construct, the cases are merged.
- Other cases may be considered: switch + expr, expr + switch, visit + expr, and expr + visit.
- No other extensions are allowed.

Functions preceeded by the keyword "`java`" have a body written in Java. They have the following properties:

- Arguments and result are pure Rascal values.
- Inside the body variables may be declared both with a Java Type or with a Rascal type. The latter will be converted by the Rascal compiler to appropriate implementation types.
- Java functions cannot acces the global state of the Rascal program.
- Side effects cause by Java functions in the Java state, are not undone in the case of backtracking.

Patterns

We distinguish three kinds of patterns:

- *Abstract* patterns: prefix dataterms that are generated by a signature.
- *Regular expression* patterns: conventional regular expressions
- *Syntax* patterns: textual fragments that are generated by a context-free grammar.

Patterns may contain variables and can occur in two syntactic positions:

- *Match* positions where the patterns is matched against another term and the variables in the pattern are bound when the match is successfull. Examples of match positions are:
    - After a `case` keyword in `switch` or `visit` statement.
    - In a generator, where generated values are matched against the pattern.
- *Make* positions where the pattern is used to construct a new term (after replacing any variables in the pattern by their values. Examples of make positions are:
    - If the pattern is preceeded by the keyword `make`.

Abstract Patterns

Datatype declarations introduce a signature of abstract terms.These terms (possibly including typed variables as introduced for concrete patterns) may be used as abstract patterns at the same position where concrete patterns are allowed. Subtype declarations define an inclusion relation between types.

Regular Expression Patterns

Regular expression patterns are ordinary regular expressions that are used to match a string value and to decompose it in parts and also to compose new strings. In a match position, a regular expression is followed by a newly declared variable (always of type `list[str]`) that contains the matches obtained by the regular expression.

Syntax Patterns

There is a notation of a *syntax pattern*: a (possibly quoted) concrete syntax fragment that may contain variables. We want to cover the whole spectrum from maximally quoted patterns that can unambiguously describe **any** syntax fragment to minimally quoted patterns as we are used to in ASF+SDF. Therefore we support the following mechanisms:

- Optionally typed variables, written as `<TYPE NAME>` or `<NAME>`.
- Quoted patterns enclosed between `[|` and `|]`. Inside a fully quoted string, the characters `<`, `>` and `|` can be escaped as `\<`, `\>`, `\|`. Fully quoted patterns may contain variables.
- Unquoted patterns are an (unquoted) syntax fragment that may contain variables.

Quoted and unquoted patterns form the *patterns* that are supported in Rascal.

Examples are:

- Quoted pattern with typed variables:

  ```
  [| while <EXP Exp> do <{STATEMENT ";"}* Stats> od |]
  ```

- Quoted pattern with untyped variables:

  ```
  [| while <Exp> do <Stats> od |]
  ```

- Unquoted pattern with typed variables:

  ```
  while <EXP Exp> do <{STATEMENT ";"}* Stats> od
  ```

- Unquoted pattern with untyped variables:

  ```
  while <Exp> do <Stats> od
  ```

Obviously, with less quoting and type information, the probability of ambiguities increases. Our assumption is that a type checker can resolve them.

Implementation hint. For every sort S in the syntax definition add the following rules:

```
S                      -> Pattern
"<" "S"? Variable ">" -> S
```

Expressions

Expressions correspond roughly to Rscript expressions with some extensions:

- There are lists, sets and relations together with comprehensions for these types.
- Generators in comprehensions may range over syntax trees.
- Generators may have a strategy option to indicate:
  - `top-down`
  - `top-down-break`
  - `bottom-up` (this is the default)
  - `bottom-up-break`

  The two other strategy option (`innermost` and `outermost`) are only meaningfull in the context of a `visit` statement.

- The complete repertoire of operators in Rscript is available but we have applied some rationalization:
  - + is now used for addition, union and concatenation (the operator `union` is gone).
  - - is now used for substraction, difference (the operator `\` is gone)
  - * is now used for multiplication and intersection (the operator `inter` is gone).
  - There are new assignment operators +=, -=, *=.
  - For a binary relation *R* that is a map (i.e. it associates a single value with each domain element) *R*(*N*), returns the single image value corresponding with *N*. This expression may also occur as left-hand side of an assignment:

    ```
    R(N) = V
    ```

    first removes from *R* the tuple with domain value *N* and then adds the tuple <*N*, *V*>. The net effect is that a new map value is assigned to *R*.
  - For an arbitrary binary relation, *R*{*N*} returns a set of corresponding image elements. This expression may also occur as left-hand side of an assignment:

    ```
    R{N} = V
    ```

    first removes all tuples with domain value *N* and then adds the tuple <*N*, *V*>. When the right-hand side is a set, new tuples are added for each element in the set:

    ```
    R{N} = {V1, ..., Vn}
    ```

    first removes all tuples with domain value *N* and then adds the tuples <*N*, *V1*>, ..., <*N*, *Vn*>.

    Note

    Consider to replace the above by standard array notation.
  - The Rscript image notation R[n] and R[-,n] are discontinued.

Statements

Rascal has the following statement types, which can be nested and composed in the usual structured manner:

- Variable declaration with initialization.
- An assignment statement assigns a value to a variable.
- If-then statement and if-then-else statement.
- A `return` statement returns a value from a function, or just returns (for functions with result type void). Note that return jumps out of an entire function, even if it is nested in a complicated control flow statement such as visit.
- A `yield` statement that delivers a replacement value during a traversal initiated by a visit statement, or a switch statement.
- A `fail` statement which jumps to the deepest nested choice point (i.e. a switch or a visit) and tries the next option available. Fail is the dual of return/yield.
- A throw statement can raise an exception.

- A `switch` statement is similar to a switch statement in C or Java and for a given subject term, it corresponds to the matching provided by the left-hand sides of a set of rewrite rules. However, it provides **only** matching at the top level of the subject term and does not traverse it. The type of each pattern must be identical to the type of the subject term (or be a subtype of it). It is an error if no case matches.
- A rewrite rule (not a top-level statement, but a child of switch and visit) consists of a Pattern followed by : and a statement that returns the replacement value:

  ```
  case Pattern : Statement
  ```

  A statement may consist of declarations and statements and is implicitly extended with a `fail` statement. The statement must therefore yield or return a value or the case as a whole will fail. To maintain some resemblance with rewrite rules, we also support the form

  ```
  case Pattern => Replacement
  ```

  which is an abbreviation for

  ```
  case Pattern : yield Replacement
  ```

- A `visit` statement corresponds to a traversal function from ASF+SDF. Given a subject term and a list of rewrite rules it traverses the term. Depending on the precise rules it may perform replacement (mimicking a transformer), update local variables (mimicking an accumulator) or a combination of these two. The visit statement may contain the same strategy options as a generator and also:
  - `innermost` = compute a fixed-point: repeat a bottom-up traversal as long as the traversal function changes values.
  - `outermost` = compute a fixed-point: repeat a traversal traversal as long as the traversal function changes values.
- A `for` statement to repeat a block of code.
- A `solve` statement to solve a set of linear equations.
- A `try` statement can be used to execute a statement and to catch any exeception raised by that statement.

Visitor definitions (UNDECIDED and INCOMPLETE)

Visitor definitions are a new idea that borrow the programmability of Systems S's single level traversals and add them to Rascal. The idea is to be able to define the strategy annotations of visit statements and generators using a simple expression language. A definition takes as formal argument the code block of the visit statement (s), which is what needs to be done at every node (the visitor).

```
%% first recurse to the arguments, then try v,
%% which if it fails returns the original structure.
visitor bottom-up(v) = all(bottom-up(v)) ; (v <+ id)

visitor innermost(t,v) = all(innermost(t,v)) ; repeat(v <+ id)
```

We demand that all visitors are infallible, which means that when the v block fails, they must return a default result of the correct type. In most cases, this would be the identity (id).

We can also try to give these definitions a more imperative look, as if they are patterns for generating code for the visitors, as in:

```
visitor bottom-up(v) {
  all {
     bottom-up(v);
```

```
  }

  try {             %% try is the '<+' of System S,
                    %% only if v fails the catch block is executed.
    v;
  } catch fail(t) {
    yield t;
  }
}

%% innermost goes down and only returns after
%% nothing changes anymore

visitor innermost(v) {
  all {
    innermost(v);     %% apply this to all children first.
  }
  while (true) { {    %% then until kingdom come, apply this block:
    try {
      v;              %% if v succeeds, it has a yield or a return
                      %% statement that updates the current node.
    } catch fail(t) { %% if v fails after all, we obtain a reference
                      %% to the current node visited which we can
                      %% return;
      yield t;
    }
  }
}

visitor bottom-up-dbg(v) {
  all {
     bottom-up(v);
  }

  try {              %% try is the '<+' of System S,
                     %% only if v fails the catch block is executed.
    v;
  } catch fail(t) {
    printf("DBG: bottom-up visitor failed on: " + t);
    yield t;
  }
}
```

After such definitions, most of which would be in the standard library of Rascal, we can use them to program actual visits:

```
visit bottom-up (t) {
  pattern => pattern
  pattern2 : { effect; }
}
```

Failure and side-effects

There are two contexts in which side-effects, i.e., assignment to variables, have to be undone in case of failure. These contexts are a rewrite rule in a switch or visit statement. If the pattern on the left-handside of the rule matches there are various possibilities:

- All control flow path through the right-hand side of the rule end in a return statement. In this case, the rule can not fail and all side-effects caused by the execution of the right-hand side are committed.

- One or more control path can fail. This can be caused by an explicit fail statement or an if-then statement with missing else-branch. In the case of failure all side-effects are undone.

  Note

  Will we undo all side-effects or only the side-effects on global variables?

- If a rule fails there are two possibilities:
  - the left-hand side contains a list pattern that has more matching options; the next option is tried.
  - the left-hand side contains a list patterns that has no more matching options or it contains no list pattern at all; the next rule is tried.

Examples

Here we list experimental examples of Rascal code to try out features.

Booleans

It seems that every language specification effort has to produce a specification of the Booleans at some moment, so let's try it now. We try the following versions:

- A version using visit, see the section called "*Booleans using visit*" [12].
- A version using an implicit reduction function

We use the following common syntax:

```
module Booleans-syntax
exports
  sorts Bool

  context-free syntax
    "true"        -> Bool
    "false"       -> Bool
    Bool "&" Bool -> Bool {left}
    Bool "|" Bool -> Bool {right}
```

Booleans using visit

A simple solution exists using the visit construct that we have encountered in the above examples.

```
module Bool-examples1

imports Booleans-syntax;

fun Bool reduce(Bool B) {
    visit bottom-up B {
      case true & <Bool B2>   => B2
      case false & <Bool B2>  => false

      case true | <Bool B2>   => true
      case false | <Bool B2>  => B2
    }
}
```

Observe that there are two styles:

- Using variables on the left-hand side: the visit is needed to fully normalize the result.

- A truth table: this is sufficient as is.

Abstract Booleans

In the above example we used concrete syntax for Booleans expressions. It also possible to define Booleans as abstract terms.

```
module Bool-abstract

data Bool btrue;
data Bool bfalse;
data Bool band(Bool, Bool);
data Bool bor(Bool, Bool);

fun Bool reduce(Bool B) {
    visit bottom-up B {
      case band(btrue, <Bool B2>)   => B2  %% Style 1: Use Variables
      case band(bfalse, <Bool B2>)  => bfalse

      case bor(btrue, btrue)        => btrue %% Style 2: Use a truth table
      case bor(btrue, bfalse)       => btrue
      case bor(bfalse, btrue)       => btrue
      case bor(bfalse, bfalse)      => bfalse
    }
}
```

First, type declarations are used to define the abstract syntax of the type Bool. Next, a similar reduce function is defined as before, but now we use abstract patterns.

Booleans with implicit reduce (NOT YET DECIDED)

In ASF, values are always reduced to a normal form before they are created. For some applications this normalization or canonicalization feature is very handy. We introduce the following syntax, which can also help in the transformation of old ASF+SDF programs to Rascal:

```
rules (Bool) {
  and(true, <Bool B2>) => B2
  and(false, Bool) => false
}
%% or
rules (Bool) {
  and(true, <Bool B2>) : { yield B2 }
  and(false, Bool) : { yield false }
}
```

These rules are applied on every Bool that is constructed. Like in ASF+SDF it is the responsibility of the programmer to make sure the rules are confluent and terminating. The body of a rules definition has the same syntax and semantics as the switch construct, allowing backtracking, side-effects and checking of conditions.

There are some issues here:

- It should be dissallowed to have private rules on public constructors; normalization is a global effect on public data-structures. On the other hand, constructors that are local to a module may have some private rules applied to them; but public rules on private constructors are dissallowed too.
- We previously had other ideas about this feature. We introduced a reduce keyword that would call a certain named function at the construction of every Bool. The problem with that feature is that the operational semantics (in the eyes of the Rascal programmer) really looks like first some term is to be created, and then a function will be applied to normalize it. In reality though, we want

to normalize the terms in an innermost fashion, and at construction time, so that the terms on the left-hand side of the rules are never actually created or allocated.

The following seems more declarative and more close to the semantics of rewrite rules being applied everywhere and in any order:

```
rule and(true,B2) => B2
rule and(true,B2) : { print("rule!"); yield B2 }
```

Abstract Graph datatype

In the Meta-Environment we use an abstract data type to exchange data representing graphs. It can be defined as follows.

```
module Graph

data Graph graph(NodeList nodes, EdgeList edges, AttributeList attributes);

type list[Node] Nodelist;

data Node node(NodeId id, AttributeList attributes);
data Node subgraph(NodeId id, NodeList nodes, EdgeList edges, AttributeList att

data NodeId id(term id);

type list[Attribute] AttributeList;

data Attribute bounding-box(Point first, Point second);
data Attribute color(Color color);
data Attribute curve-points(Polygon points);
data Attribute direction(Direction direction);
data Attribute fill-color(Color color);
data Attribute info(str key, term value);
data Attribute label(str label);
data Attribute tooltip(str tooltip);
data Attribute location(int x, int y);
data Attribute shape(Shape shape);
data Attribute size(int width, int height);
data Attribute style(Style style);
data Attribute level(str level);
data Attribute file(File file);
data Attribute file(term file);

data Color rgb(int red, int green, int blue);

data Style bold | dashed | dotted | filled | invisible | solid;

data Shape box | circle | diamond | egg | elipse | hexagon | house |
           octagon | parallelogram | plaintext | trapezium | triangle;

data Direction forward | back | both | none;

type list[Edge] Edgelist;

data Edge edge(NodeId from, NodeId to, AttributeList attributes);

type list[Point] Polygon;

data Point point(int x, int y);
```

Tree traversal

Here is the binary tree example that we use in explaining traversal functions in ASF+SDF.

```
module BTree-syntax
imports basic/Integers

exports
  sorts BTREE
  context-free syntax
    Integer         -> BTREE
    f(BTREE,BTREE)  -> BTREE
    g(BTREE,BTREE)  -> BTREE
    h(BTREE,BTREE)  -> BTREE
    i(BTREE,BTREE)  -> BTREE
```

```
module BTree-Examples
imports BTree-syntax;

%% Ex1: Count leaves in a BTREE
%% Idea: int N : T generates alle Integer leaves in the tree
%% # is the built-in length-of operator

fun int cnt(BTREE T) {
    return #{N | int N : T}
}

%% Ex1: an equivalent,  more purist, version of the same function:
fun int cnt(BTREE T) {
    return #{N | <Integer N> : T}
}

%% Ex1: alternative solution using trafo functions:

fun int cnt(BTREE T) {
    int C = 0;
    visit T {
      case <Integer N> : C = C+1
    };
    return C;
}

%% Ex2: Sum all leaves in a BTREE
%% NB sum is a built-in that adds all elements in a set or list.
%% Here we see immediately the need to identify
%% - the built-in sort "int"
%% - the syntactic sort "Integer"

fun int sumtree(BTREE T) {
    return sum({N | int N : T});
}

%% Ex2: using accumulator

fun int cnt(BTREE T) {
    int C = 0;
    visit T {
      case <Integer N> : C = C+N
    };
    return C;
}

%% Ex3: Increment all leaves in a BTREE
%% Idea: using the construct "visit T { ... }" visit all leaves in
%% thetree T that match an integer and replace each N in T by N+1.
%% The expression as a whole returns the modified term.
%% Note that two conversions are needed here:
%% - from int to NAT to match subterms
%% - from int to NAT to convert the result of addition into
%%   a NAT tree

fun BTREE inc(BTREE T) {
    visit T {
      case <Integer N>: yield (N + 1)
    };
}

%% Ex4: full replacement of g by i
%% The whole repertoire of traversal functions is available:
%% - visit bottom-up T { ... }
%% - visit bottom-up-break  T { ... }
%% - etc.
%% A nice touch is that these properties are not tied to the
%% declaration of a travesal function (as in ASF+SDF) but to
%% its use.

fun BTREE frepl(BTREE T) {
    visit bottom-up T {
      case g(<BTREE T1>, <BTREE T2>) =>
           make i(<BTREE T1>, <BTREE T2>)
    };
}

%% Ex5: Deep replacement of g by i

fun BTREE frepl(BTREE T) {
    visit bottom-up-break T {
      case g(<BTREE T1>, <BTREE T2>) =>
           make i(<BTREE T1>, <BTREE T2>)
    };
}

%% Ex6: shallow replacement of g by i (i.e. only outermost
%% g's are replaced);

fun BTREE srepl(BTREE T) {
    visit top-down-break T {
      case g(<BTREE T1>, <BTREE T2>) =>
           make i(<BTREE T1>, <BTREE T2>)
    };
}

%% Ex7: We can also add the first/td directives to all generators
%% (where "all td" would be the default):

fun set[BTREE] find-outer-gs(BTREE T) {
    return
    { S | STATEMENT S : top-down-break T,
          g(<BTREE T1>, <BTREE T2>) := S };
}

%% Ex8: accumulating transformer that increments leaves with
%% amount D and counts them
fun tuple[int, BTREE] count-and-inc(BTREE T, int C, int D) {
    int C = 0;

    visit T {
      case <Integer N>: { C = C + 1; yield N+D }
     };
     return <C, T>;
}
```

Substitution in Lambda

Below a definition of substitution in lambda expressions. It would be nice to get this as simple as possible since it is a model for many binding mechanisms. It is also a challenge to write a generic substitution function that only depends on the syntax of variables and argument binding.

```
module examples/Lambda/Lambda-syntax

exports
sorts Var %% variables
      Exp %% expressions
lexical syntax
     [a-z]+                -> Var
context-free syntax
     "prime" "(" Var ")"  -> Var  %% generate unique name
     Var                   -> Exp  %% single variable
     "fn" Var "=>" Exp     -> Exp  %% function abstraction
     Exp Exp               -> Exp  %% function application
```

Examples:

```
module Lambda-Examples
imports Lambda-syntax;

fun set[Var] allVars(Exp E) {
    return {V | Var V : E}
}

fun set[Var] boundVars(Exp E) {
    return {V | fn <Var V> => <Exp E1> : E};
}

fun set[Var] freeVars(Exp E) {
    return allVars(E) - boundVars(E);
}

%% Generate a fresh variable if V does not occur in
%% given set of variables.

fun Var fresh(Var V, set[Var] S) {
    if (V in S){ return prime(V); } else {return V;};
}
```

```
%% Substitution: replace all occurrences of V in E2 by E1

fun Exp subst(Var V1, Exp E1, Exp E2) {

    switch E2 {
      case <Var V2>: if(V1 != V2){ yield V2; }

      case <Var V2>: if(V1 == V2){ yield E1; }

      case <Exp Ea> <Exp Eb>: {
        Exp EaS = subst(V, E, Ea);
        Exp EbS = subst(V, E, Eb);
        return make <Exp EaS> <Exp EbS>;
      }

      case fn <Var V2> => <Var Ea>:
        if (V1 == V2) { yield make fn <Var V2> => <Exp Ea> }

      case fn <Var V2> => <Exp Ea>:
        if(V1 != V2 and not(V1 in freeVars(E2) and
           V2 in freeVars(E1))){
           Exp E1S = subst(V1, E1, Ea);
           yield make fn <Var V2> => <Exp E1S>;
        }

      case fn <Var V2> => <Exp Ea>:
        if(V1 != V2 and V1 in freeVars(Ea) and
           V2 in freeVars(E1)){
           Var V3 = fresh(V2, freeVars(Ea) + freeVars(E1));
           Exp EaS = subst(V1, E1, subst(V2, V3, E2));
           yield make fn <Var V3> => <Exp EaS>;
        }
    };
}
```

Renaming in Let

```
module Let-syntax
exports
sorts Var %% variables
      Exp %% expressions
lexical syntax
     [a-z]+                          -> Var
context-free syntax
     Var                             -> Exp
     "let" Var "=" Exp "in" Exp "end" -> Exp
```

Examples:

```
module Let-Example
imports Let;

%% Rename all bound variables in an Exp
%% Version 1: purely functional
%% Exp: given expression to be renamed
%% rel[Var,Var]: renaming table
%% Int: counter to generate global variables

```

```
fun Exp rename(Exp E, rel[Var,Var] Rn, Int Cnt) {
    switch E {
    case let <Var V> = <Exp E1> in <Exp E2> end: {
         Var Y = "x" + Cnt;  %% this + operator concatenates
                             %% (after converting the int to str)
         int Cnt1 = Cnt + 1;
         Exp E1R = rename(E1, Rn, Cnt);
         Exp E2R = rename(E2, {<V, Y>} + Rn, Cnt1);
         return make let <Var Y >= <Exp E1R>
                     in
                        <Exp E2R>
                     end;
          }

    case <Var V>: return Rn[V]

    default: return E
    };
}
```

Renaming in Let using globals

Here is the same renaming function now using two global variables.

```
module Let-Example
imports Let;

%% Rename all bound variables in an Exp
%% Version 2: using global variables
%% Cnt: global counter to generate fresh variables
%% rel[Var,Var]: global renaming table

global int Cnt = 0;
global rel[Var,Var] Rn = {};

fun Var newVar() {
    global int Cnt;
    Cnt = Cnt + 1;
    return "x" + Cnt
}

fun Exp rename(Exp E) {
    global int Cnt;
    global rel[Var,Var] Rn;
    switch E {
    case let <Var V> = <Exp E1> in <Exp E2> end: {
         Var Y = newVar();
         Rn = {<V, Y>} + Rn;
         Exp E1R = rename(E1);
         Exp E2R = rename(E2);
         return make let <Var Y >= <Exp E1R>
                     in
                        <Exp E2R>
                     end;
          }

    case <Var V>: return Rn(V)
    default: return E
    };
}
```

Concise Pico Typechecker

The following example shows the tight integration ASF with comprehensions.

```
module Typecheck

imports Pico-syntax;
imports Errors;

subtype Env rel[PICO-ID,TYPE];

fun list[Error] tcp(PROGRAM P) {
    switch P {
      case begin <DECLS Decls> <{STATEMENT ";"}* Series> end: {
           Env Env = {<Id, Type> |
                      [| <PICO-ID Id> : <TYPE Type> |] : Decls};
           return [ tcst(S, Env) | Stat S : Series ]
      }
    };
}

fun list[Error] tcst(Stat Stat, Env Env) {
    switch Stat {
      case [| <PICO-ID Id> = <EXP Exp>|]: {
        TYPE Type = Env(Id);
        return type-of(Exp, Type, Env);
      }

      case if <EXP Exp> then <{STATEMENT ";"}* Stats1>
                   else <{STATEMENT ";"}* Stats1> fi:
        return type-of(Exp, natural, Env) +
               tcs(Stats1, Env) + tcs(Stats2, Env)

      case while <EXP Exp> do <{STATEMENT ";"}* Stats1> od:
        yield type-of(Exp, natural, Env) + tcs(Stats, Env)
    };
}

fun list[Error] type-of(Exp E, TYPE Type, Env Env) {
    switch E {
      case <NatCon N>: if(Type == natural){ return []; }

      case <StrCon S>: if(Type == string) { return []; }

      case <PICO-ID Id>: {
         TYPE Type2 = Env(Id);
         if(Type2 == Type) { return []; }
      }

      case <EXP E1> + <EXP E2>:
        if(Type == natural){
           return type-of(E1, natural, Env) +
                  type-of(E1, natural, Env);
        }
      case <EXP E1> + <EXP E2>:
        if(Type == natural){
           return type-of(E1, natural, Env) +
                  type-of(E1, natural, Env);
        }

      case <EXP E1> || <EXP E2>:
        if(Type == string){
          return type-of(E1, string, Env) +
                 type-of(E1, string, Env)
        }

      default: return [error("Incorrect type")]
    };
}
```

Pico evaluator

```
module Pico-eval
imports pico/syntax/Pico;

subtype PICO-VALUE int;
subtype PICO-VALUE str;

subtype VEnv rel[PICO-ID, PICO-VALUE];

fun VEnv evalProgram(PROGRAM P){
    switch P {
      case begin <DECLS Decls> <{STATEMENT ";"}* Series> end: {
          VEnv Env = evalDecls(Decls);
          return evalStatements(Series, Env);
      }
   }
};

fun VEnv assign(VEnv Env, PICO-ID Id, PICO-VALUE V){
  return Env +> {<Id, V>}
  %% we need a nice tuple replacement operator here
}

fun PICO-VALUE valueOf(VEnv Env, PICO-ID Id){
  return Env(Id);
}

fun VEnv evalDecls(DECLS Decls){
    VEnv Env = {};
    visit Decls {
      case <PICO-ID Id> : string:  Env = assign(Env, Id, "")
      case <PICO-ID Id> : natural: Env = assign(Env, Id, 0)
    }
}

fun VEnv evalStatements({STATEMENT ";"}* Series, VEnv Env){
    switch Series {
      case <STATEMENT Stat>; <{STATEMENT ";"}* Series2>: {
        Env Env2 = evalStatement(Stat, Env);
        return evalStatements(Series2, Env2);
      }
      case [| |]: return Env
    }
}

fun VEnv evalStatement(STATEMENT Stat, VEnv Env){
    switch Stat {
      case [| <PICO-ID Id> = <EXP Exp> |]: {
        PICO-VALUE Val = evalExp(Exp, Env);
        return assign(Env, Id, Val)
      }

      case if <EXP Exp> then <{STATEMENT ";"}* Stats1>
                   else <{STATEMENT ";"}* Stats1> fi:{
        PICO-VALUE Val = evalExp(Exp, Env);
        if(Val != 0)
          return evalStatments(Stats1, Env)
        else
          return evalStatements(Stats2, Env)

      }

      case while <EXP Exp> do <{STATEMENT ";"}* Stats1> od:{
        PICO-VALUE Val = evalExp(Exp, Env);
        if(Val != 0)
          return Env
        else {
          VEnv Env2 = evalStatements(Stats1, Env);
          return evalStatement(Stat, Env2)
        }
      }
    };
};

fun PICO-VALUE evalExp(Exp exp, VEnv Env) {
    switch exp {
      case <NatCon N>: return N

      case <StrCon S>: return S

      case <PICO-ID Id>: return valueOf(Env, Id)

      case <EXP exp1> + <EXP exp2>: {
           Natural nat1 = evalExp(exp1, Env);
           Natural nat2 = evalExp(exp2, Env);
           return nat1 + nat2;
      }
      case <EXP exp1> - <EXP exp2>: {
           Natural nat1 = evalExp(exp1, Env);
           Natural nat2 = evalExp(exp2, Env);
           return nat1 + nat2;
      }
      case <EXP exp1> || <EXP exp2>: {
           StrCon str1 = evalExp(exp1, Env);
           StrCon str2 = evalExp(exp2, Env);
           return concat(str1, str2);
      }
   }
```

```
}
```

Pico evaluator with globals

Here is the same evaluator but now using a global variable to represent the value environment.

```
module Pico-eval
imports pico/syntax/Pico;

subtype PICO-VALUE int;
subtype PICO-VALUE str;

subtype VEnv rel[PICO-ID, PICO-VALUE];

fun void evalProgram(PROGRAM P){
    switch P {
      case begin <DECLS Decls> <{STATEMENT ";"}* Series> end: {
          evalDecls(Decls);
          evalStatements(Series);
      }
   }
};

fun void assign(PICO-ID Id, PICO-VALUE V){
  global Venv Env;
 Env = Env +> {<Id, V>}
  return;
}

fun PICO-VALUE valueOf(PICO-ID Id){
  global Venv Env;
  return Env(Id);
}

fun VEnv evalDecls(DECLS Decls){
    global VEnv Env = {};
    visit Decls {
      case <PICO-ID Id> : string:  assign(Id, "")
      case <PICO-ID Id> : natural: assign(Id, 0)
    }
}

fun void evalStatements({STATEMENT ";"}* Series){
    global VEnv Env;
    switch Series {
      case <STATEMENT Stat>; <{STATEMENT ";"}* Series2>: {
        evalStatement(Stat);
        evalStatements(Series2);
        return
      }
      case [| |]: return
    }
}

fun void evalStatement(STATEMENT Stat){
    global VEnv Env;
    switch Stat {
      case [| <PICO-ID Id> = <EXP Exp> |]: {
        PICO-VALUE Val = evalExp(Exp);
        assign(Id, Val);
        return
      }

      case if <EXP Exp> then <{STATEMENT ";"}* Stats1>
                   else <{STATEMENT ";"}* Stats1> fi:{
        PICO-VALUE Val = evalExp(Exp);
        if(Val != 0) {
          evalStatements(Stats1);
          return
        } else {
          evalStatements(Stats2);
          return
        }
      }

      case while <EXP Exp> do <{STATEMENT ";"}* Stats1> od:{
        PICO-VALUE Val = evalExp(Exp);
        if(Val != 0)
          return
        else {
          evalStatements(Stats1);
          evalStatement(Stat);
          return
        }
      }
    };
};

fun PICO-VALUE evalExp(Exp exp) {
    global VEnv Env;
    switch exp {
      case <NatCon N>: return N

      case <StrCon S>: return S

      case <PICO-ID Id>: return valueOf(Id)

      case <EXP exp1> + <EXP exp2>: {
           Natural nat1 = evalExp(exp1);
           Natural nat2 = evalExp(exp2);
           return nat1 + nat2;
      }
      case <EXP exp1> - <EXP exp2>: {
           Natural nat1 = evalExp(exp1);
           Natural nat2 = evalExp(exp2);
           return nat1 + nat2;
      }
      case <EXP exp1> || <EXP exp2>: {
           StrCon str1 = evalExp(exp1);
           StrCon str2 = evalExp(exp2);
           return concat(str1, str2);
      }
   }
}
```

Pico control flow extraction

```
module Pico-controlflow
imports pico/syntax/Pico;

subtype CP EXP;         %% A Code Point, union of two types
subtype CP STATEMENT;

subtype CFSEGMENT tuple(set[CP] entry,
                        rel[CP,CP] graph,
                        set[CP] exit);

fun CFSEGMENT cflow({STATEMENT ";"}* Stats){
    switch Stats {
      case <STATEMENT Stat> ; <{STATEMENT ";"}* Stats2>: {
           <set[CP] En1, rel[CP,CP] R1, set[CP] Ex1> = cflow(Stat);
           <set[CP] En2, rel[CP,CP] R2, set[CP] Ex2> =
                                                     cflow(Stats2);
           return <En1, R1 + R2 + (Ex1 x En2), Ex2>
      }

      case [| |]: return <{}, {}, {}>
    }
};

fun CFSEGMENT cflow(STATEMENT Stat){
    switch Stat {
      case [| while <EXP Exp> do <{STATEMENT ";"}* Stats> od |] : {
           <set[CP] En,  rel[CP,CP] R,  set[CP] Ex> = cflow(Stats);
           return <{Exp}, ({Exp} x En) + R + (Ex x {Exp}),{Exp}>;
      }

      case [| if <EXP Exp> then <{STATEMENT ";"}* Stats1>
                   else <{STATEMENT ";"}* Stats2> fi |]: {
           <set[CP] En1, rel[CP,CP] R1, set[CP] Ex1> =
                                                     cflow(Stats1);
           <set[CP] En2, rel[CP,CP] R2, set[CP] Ex2> =
                                                     cflow(Stats2);
           return < {Exp},
                    ({Exp} x En1) + ({Exp} x En2) + R1 + R2,
                    Ex1 + Ex2
                  >;
      }

      case [| <STATEMENT Stat> |]: return <{Stat}, {}, {Stat}>
    };
}
```

Pico use def extraction

```
module Pico-use-def

imports pico/syntax/Pico;


fun rel[PICO-ID, EXP] uses(PROGRAM P) {
  return {<Id, E> | EXP E : P, PICO-ID Id := E}
}

fun rel[PICO-ID, STATEMENT] defs(PROGRAM P) {
  return {<Id, S> | STATEMENT S : P,
                    [| <PICO-ID Id> := <EXP Exp> |] := S}
}
```

The above uses a "matching condition" to decompose S. The problem solved is that we want to have a name for the whole assignment *and* for the lhs identifier. Also note that, compared to older definitions of these functions, the iudentifier is placed as first element in each tuple.

Pico uninitialized variables

```
module Pico-uninit
imports pico/syntax/Pico
        Pico-controlflow
        Pico-use-def;

fun set[PICO-ID] uninit(PROGRAM P) {
    rel[EXP,PICO-ID] Uses = uses(P);
    rel[PICO-ID, STATEMENT] Defs = defs(P);
    CFSEGMENT CFLOW = cflow(P);
    set[CP] Root = CFLOW.entry;
    rel[CP,CP] Pred = CFLOW.graph;

    return {Id | tuple(EXP E, PICO-ID Id) : Uses,
                 E in reachX(Root, Defs{Id}, Pred)
    };
}
```

Pico common subsexpression elimination

```
module Pico-common-subexpression

imports pico/syntax/Pico
        Pico-controlflow
        Pico-use-def;

fun PROGRAM cse(PROGRAM P) {
    rel[PICO-ID, STATEMENT] Defs = defs(P);
    rel[CP,CP] Pred = cflow(P).graph;
    rel[EXP, PICO-ID] replacements =
       {<E2, Id> | STATEMENT S : P,
                   [| <PICO-ID Id> := <EXP E> |] := S,
                   Id notin E,
                   EXP E2 : reachX({S}, Defs{Id}, Pred)
       };

    visit P {
      case <EXP E>: if({ PICO-ID Id } := replacements{E}) yield Id
    };
}
```

Paraphrased: Replace in P all expressions E2 by Id, such that

- P contains a statement S of the form Id := E,
- Id does not occur in E,

- E2 can be reached from S,
- There is no redefinition of Id between S and E2.

Note that a slight abbreviation is possible if we introduce labelled patterns (here S): [UNDER DISCUSSION]

```
 rel[EXP, PICO-ID] replacements :=
       {<E2, Id> | <PICO-ID Id> := <EXP E> S : P,
                   Id notin E,
                   EXP E2 : reachX({S}, Defs(Id), Pred)
       };
```

Also note that we could factor out the assignment pattern to make cse more generic if we introduce patterns as first class citizens:

```
fun PROGRAM cse(PROGRAM P,
                pat STATEMENT Assign(PICO-ID Id, EXP E)) {
 ...
    rel[EXP, PICO-ID] replacements :=
       {<E2, Id> | Assign S : P,
                   Id notin E,
                   EXP E2 : reachX({S}, Defs(Id), Pred)
       };

 ...
}
```

Example invocations (Pico style)

```
cse(P, <PICO-ID Id> := <EXP E>)
```

or (Cobol style):

```
cse(P, move <EXP E> to <PICO-ID Id>)
```

Note that the order of variables in the pattern and its declaration may differ.

It is to be determined how the instantiation of a pattern looks, e.g.

```
Assign([|x|], [| y = 1 |])
```

Pico constant propagation

```
module Pico-constant-propagation

imports pico/syntax/Pico
        Pico-controlflow
        Pico-use-def;

fun Boolean is-constant(EXP E) {
   switch E {
     case <NatCon N> => true

     case <StrCon S> => true

     case <EXP E> => false
   }
}
```

```
fun PROGRAM cp(PROGRAM P) {
    rel[PICO-ID, STATEMENT] Defs = defs(P);
    rel[CP,CP] Pred = cflow(P).graph;

    rel[PICO-ID, EXP] replacements =
      {<Id2, E> | STATEMENT S : P,
                  [| <PICO-ID Id> := <EXP E> |] := S,
                  is-constant(E),
                  PICO-ID Id2 : reachX({S},Defs{Id},Pred),
                  Id2 == Id
      };

    visit P {
     case <PICO-ID Id>: if({ EXP E } := replacements{Id}) yield E
    };
}
```

Paraphrased: Replace in P all expressions Id2 by the constant E, such that

- P contains a statement S of the form Id := E,
- E is constant,
- Id2 can be reached from S,
- Id2 is equal to Id,
- There is no redefinition of Id between S and Id2.

Pico Reaching definitions

Recall the equations construct as used, for example, in the reaching definitions example in the Rscript guide. It computes the values of a set of variables until none of them changes any longer. The "solve" statement achives the same effect.

```
module Pico-reaching-defs

subtype Def  tuple(Stat theStat, Var theVar);
subtype Use  tuple(Stat theStat, Var theVar);

fun set[Stat] predecessor(rel[Stat,Stat] P, Stat S) {
    return P[-,S] %% outdated image operator
}

fun set[Stat] successor(rel[Stat,Stat] P, Stat S) {
    return P[S,-]
}

fun rel[Stat, Def] reaching-definitions(rel[Stat,Var] DEFS,
                                        rel[Stat,Stat] PRED) {

    set[Stat] STATEMENT = carrier(PRED);

    rel[Stat,Def] DEF  = {<S,<S,V>> | <Stat S, Var V> : DEFS};

    rel[Stat,Def] KILL =
        {<S1, <S2, V>> | <Stat S1, Var V> : DEFS,
                         tuple(Stat S2, V) : DEFS,
                         S1 != S2
        };

     rel[Stat,Def] IN = {};
     rel[Stat,Def] OUT = DEF;

     solve {
           IN  =  {<S, D> | int S : STATEMENT,
                            Stat P : predecessor(PRED,S),
                            Def D : OUT{P}};
           OUT = {<S, D> |  int S : STATEMENT,
                            Def D : DEF{S} +  (IN{S} - KILL{S})}
    };
    return IN;
}
```

Symbol table with scopes

Here is a (probably naive) implementation of a symbol table that maintains a list of numbered scopes as well as a (Name, Value) mapping in each scope. Note that we introduce parameterized modules to do this right.

```
module SymTable[Name, Value]

%% A scope-oriented symbol table.
%% Each scope consists of a map from names to values.
%% THis more intended to explore whether this can be expressed
%% *at all* than that the datatype is well designed.

subtype ScopeMap rel[Name, Value];
subtype ScopeId  int
datatype STable  stable(ScopeId scope, rel[int, ScopeMap] scopes);

%% Create a new, empty,  table
fun STable new(){
  return make stable(0, {<0, {}>});
}

%% Create a new, non-empty, table
fun STable new(ScopeId scope, rel[int, ScopeMap] scopes){
  return make stable(scope, scopes);
}

%% Update, in a given scope, the value of a variable
fun STable update(STable ST, ScopeId scope, Name N, Value V){
  X = ST.scopes(scope) + {<N, V>};
  return new(scope, ST.scopes + (ST.scopes(scope) + {<N, V>}));
}

%% Get, in a given scope, the value of a variable
fun STable value(STable ST, ScopeId scope, Name N){
  ScopeMap smap = ST.scopes(scope);
  return smap(N);
}

%% update, in the current scope, the value of a variable
fun STable update(STable ST, Name N, Value V){
  ScopeId scope = ST.scope;
  ScopeMap smap = ST.scopes(scope) + {<N,V>};
  return new(scope, ST.scopes +
                    (ST.scopes(scope) + {<scope, smap>}));
}

%% Get, in the current scope, the value of a variable
fun STable value(STable ST, Name N){
  ScopeMap smap = ST.scopes(ST.scope);
  return smap(N);
}

%% add a new scope and make it the current scope
fun STable new-scope(STable ST){
  ScopeId scope = ST.scope + 1;
  return new(scope, ST.scopes + {<scope, {}>});
}

%% switch to another scope
fun STable switch-scope(STable ST, ScopeId scope){
  return new(scope, ST.scopes);
}
```

Innerproduct

[Example taken from TXL documentation]

Define nnerproduct on vectors of integers, e.g. (1 2 3).(3 2 1) => 10.

```
module examples/Vectors/Vector-syntax

exports
  imports basic/Integers
sorts Vector

context-free syntax
   "(" Integer* ")"    -> Vector
   Vector "." Vector   -> Integer
```

```
module Innerproduct

imports Vector-syntax

fun int innerProduct(Vector V1, V2){
    if ( ( <Integer N1> <Integer* Rest1> ) := V1 &&
         ( <Integer N2> <Integer* Rest2> ) := V2
       )
       return (N1*N2) + innerProduct( (<Rest1>), (<Rest2>) )
    else
       return 0;
}
```

Bubble sort

[Example taken from TXL documentation]

```
module Bubble

fun Integer* sort(Integer* Numbers){
    visit Numbers {
      case <Integer* Rest1>
              <Integer N1> <Integer N2>
           <Integer* Rest2>:
           if(N1 > N2){
              return sort(make <Integer* Rest1>
                                  <Integer N2> <Integer N1>
                               <Integer* Rest>);
           }
    };
    return Numbers
}
```

This example raises a number of issues about the execution of visit.

Another way to write this is:

```
module Bubble2

fun Integer* sort(Integer* Numbers){
    visit repeat Numbers {
      case <Integer N1> <Integer N2>:
        if( N1 > N2)
            yield make <Integer N2> <Integer N1>
    };
    return Numbers
}
```

The visit will replace all adjacent pairs that are in the wrong order in the current list. This is repeated (fixed point operator) until no more changes are possible.

Generic Bubble sort [under discussion]

Here is a generic bubble sort wich uses type parameters (&ELEM) and a function parameter.

```
module Bubble-Gen

fun &Elem* sort(&Elem* Elements,
                fun bool GreaterThan(&Elem, &Elem)){

  visit repeat Elements {
      case <Elem E1> <Elem E2>:
        if(GreaterThan(E1, E2))
           yield make <Elem E2> <Elem E1>

     }
}
```

Do we want this generality? What are the implications for the implementation? The current syntax does not yet allow type variables in patterns.

Read-Eval-Print Loop (REPR)

For the scripting of application it is important to have a command language and read-eval-print loop. Here is an attempt. The command prompt is ">".

```
> import lang.java.syntax.Main as Java
> str source := read("program.java");
> CU program := Java.CU.parse(source);
> accu int count(CU P, int cnt) {
>   switch P {
>     Java.Statements.IF => cnt++;
>   }
> }

> count(program)
17
```

There are several innovations here:

- The import associates a name with the imported module.

  Note

  This means that "grammar" and "rule" become notions that can be manipulated.

- There is a read functions that reads a text file into a string.

  Note

  We need an io library that reads/writes strings and data values.

- We associate a parse function with every non-terminal in a grammar.
- The notation `Java.Statements.IF` consists of three parts:
  - Language name
  - Sort name
  - Rule name (currently implemented with the "cons" attribute).

  It can be used as pattern. Other potential uses are as generator:

  ```
  {S | Java.Statements.IF S : P}
  ```

  It generates all if statements in P.

Syntax Definition

See separate SDF definition.

Protyping/implementation of Rascal

Every prototype will have to address the following issues:

- Parsing/typechecking/evaluating Rascal.
- How to implement the relational operations.
- How to implement matching.
- How to implement replacement.
- How to implement traversals.

The following options should be considered:

- Implementation of a typechecker in ASF+SDF:
    - Gives good insight in the type system and is comparable in complexity to the Rscript typechecker.
    - Work: 2 weeks
- Implementation of an evaluator in ASF+SDF.
    - Requires reimplementation of matching & rewriting in ASF+SDF.
    - Bound to be very slow.
    - Effort: 4 weeks
- Implementation of a typechecker in Rascal.
    - Interesting exercise to asses Rascal.
    - Not so easy to do without working Rascal implementation.
    - Not so easy when Rascal is still in flux.
    - Effort: 1 week
- Implementation of an evaluator in Rascal.
    - Ditto.
- Extending the current ASF+SDF interpreter.
    - This is a viable options. It requires extensions of UPTR.
    - Effort: 4 weeks
- Translation of Rascal to ASF+SDF in ASF+SDF.
    - Unclear whether this has longer term merit.
    - Allows easy experimentation and reuse of current ASF+SDF implementation.
    - Effort: 4 weeks
- Implementation of an interpreter in Java.
    - A future proof and efficient solution.
    - Requires reimplementation of matching & rewriting in Java.
    - Effort: 8 weeks.

Rascal advertisements

In this section we enumerate numerous facts about Rascal that advertise it to different audiences

Generic arguments

What is good about Rascal in a few words?

- Rascal is a DSL for source code analysis and transformation. It provides a pletora of high level statements and expressions, taking away the boilerplate of implementing and debugging tools that manipulate programs.

- Rascal combines the best features of imperative programming with the best features of functional programming and term rewriting.
    - Simple structured statements for control flow and variable assignments for data flow are powerful and simple features of the imperative programming paradigm. They allow control flow and data flow to be understandable and traceable.
    - From functional programming we borrow that all values are immutable and non-null. Issues with aliasing and referential integrity, such as frequently occur in imperative and OO programming therefore do not exist in Rascal.
    - The Rascal type system is as powerful as most functional languages (higher-order polymorphic functions), however to make the language debuggable and understandable it, in principle, does not provide type inference.
    - From term rewriting we inherit powerful pattern matching facilities, integration with context-free parsing and concrete syntax.
- Rascal supports both a scripting experience, and a compiled program experience.
- Rascal is type safe, but flexible. It's type system prevents common programming errors, but still allows ample opportunity for reusable code. The reasons are the we allow co-variance in the sub-typing relationship, high-order polymorphic functions and parameterized data-types.
- Rascal allows different styles of programming. From extremely high level specification, down to straight imperative programming.
- Rascal was inspired by and borrows from several other DSL's for program analysis and transformation in academia and industry, namely ASF+SDF, Rscript, TXL, TOM, DMS, Stratego, Elan, Maude, Grok, Haskell, ML and Setl.
- Rascal integrates seemlessly with Eclipse IMP and The Meta-Environment.

Rascal for ASF+SDF programmers

Rascal is the successor of ASF+SDF. What's the difference? What's the same?

- Rascal has roughly all the high level features of ASF+SDF and some more. Old ASF+SDF specifications can be transformed to Rascal programs using a conversion tool.
- Rascal still uses SDF for syntax definition and parser generation.
- Rascal has a module system that is independent of SDF. Rascal modules introduce a namespace scope for variables and functions, which can be either private or public. Rewrite rules are global as in ASF+SDF. Modules can have type parameters as in SDF, which are instantiated by import statements.
- In Rascal, patterns and variables in concrete syntax may optionally be quoted and escaped, and support explicit declaration of the top non-terminal to solve ambiguity.
- Rascal rules read in the order of execution instead of first the left-hand side, then the conditions, and then the right-hand side of ASF+SDF equations
- Rascal has primitive and efficient implementations for sets, relations and maps
- Rascal can be used without SDF, supporting for example regular expressions and abstract data types (pure ATerms)
- Rascal has primitive support for functions, which have a fixed syntax, always return a value and have imperative control flow statements. Adding a function will not trigger the need for regenerating

parse tables. Function types can be polymorphic in their parameters and also allow functions as arguments to implement reusable algorithms.

- The imperative nature of Rascal allows you to factor out common code and nest conditionals, unlike in ASF+SDF where alternative control flow paths had to be encoded by enumerating equations with non-overlapping conditions.
- Rascal is an imperative language, which natively supports I/O and other side-effects without work-arounds. When backtracking occurs, for example over list matching, Rascal makes sure that most side-effects are undone, and that I/O is delayed until no more backtracking can occur. Even rewrite rules support side-effects in Rascal.
- Rascal has native support for traversals, instead of the add-on it used to be in ASF+SDF. The visit statement is comparable to a traversal function, and is as type-safe as the previous, and more programmeable.
- Instead of accumulator values of traversal functions in ASF+SDF, Rascal simply supports lexically scoped variables that can be updated using assignments.
- Rascal adds specific expressions for relational calculus, all borrowed directly from RScript.
- When programming using Rascal functions, instead of rules, the control flow of a program becomes easily traceable and debuggable. It is simply like stepping through well structured code.
- Rascal is based on a Java interpreter, or a Java run-time when compiled. So the code is more portable.

Rascal for imperative and object-oriented programmers

Rascal is an imperative DSL with high level statements and expressions specifically targeted at the domain of analysis and transformation of source code:

- Rascal is safe: there are no null values, and all values are immutable. Source code and abstract syntax trees, and the facts extracted from them are immutable. The Rascal interpreter and compiler make sure this is implemented efficiently. Without mutability it is easy to combine stages of your programs that analyse or annotate with stages that transform. Sharing a value does not introduce a coupling like in OO, simply because changes are only visible to the code that changes the values.
- Rascal is extra safe: it has a type system that prevents casting exceptions and other run-time failures. Still the type system specifically allows many kinds of combinations. For example, unlike in Java a set of integers is a subtype of a set of numbers (co-variance), which allows you to reuse algorithm for sets of numbers on sets of integers. It also provides true polymorphic and functions (no erasure), and functions can safely be parameters to other functions.
- Rascal provides high level statements and expressions for:
  - Visitors in all kinds of orders, expressed very concisely, and type safe.
  - Pattern matching and construction (with concrete syntax!)
  - Equation/constraint solving
  - Relational calculus
  - Rewrite rules for normalization/canonicalization of any kind of data-structure
  - Support for parsing using context-free grammars (via importing modules from the SDF language).
  - (de)Serialization of values

  - Communication with databases
- Rascal provides typed data constructors for common mathematical structures, such as:
  - terms (a.k.a. abstract data types, tree nodes)
  - parse trees (derivations of context-free grammars, for concrete syntax and direct manipulation of source code)
  - relations
  - sets
  - maps
  - tuples
- In Rascal you can implement high-fidelity source-to-source transformations. Without too much overhead, programs can do extensive rewriting of the source code without the loss of particular layout standards or source code comments.
- Rascal is syntax-safe. When you use Rascal to generate or transform source code, it statically detects whether the resulting source code is syntactically correct.
- Rascal is executed by an interpreter written in Java, or it can be compiled to Java classes.

Rascal implementation ideas

Rascal needs to support both a scripting experience as an optimized compiled language experience. Also, it needs to integrate fully with Meta-Environment and Eclipse IMP. Therefore, we have both a simple and unoptimized interpreter in mind, as well as a compiler that aggressively, but correctly, optimizes Rascal programs. The run-time of compiled programs and the interpreter will share the implementation of data-structures.

Data structures

Both the compiled code and the interpreter will run on the same data-structures which are defined by the IMP PDB project.

- We could start with the simple implementation that is now in IMP already which is based on the Java library and use the clone method to implement immutability. This will prove to be slow, but its an easy start.
- Integration with the ATerms; extend the ATerm library with all the features of the PDB, such that it becomes an implementation of the PDB's interfaces.
  - PDB's terms are typed, while ATerms are not.
  - ATerms demand canonicalization/sharing, which may prove to be hard to implement for maps, sets and relations.
  - PDB does not yet have any story for serialization.
  - ATerms will need to "implement" the PDB's interfaces which will add a dependency and seriously break other peoples code if we are not careful
  - ATerms need to be typed in order to implement correct visiting behavior when AFun's are overloaded.
- The C story is harder

  - Extension of C ATerms is hard due to the nature of C, the ATerm garbage collector, the ATerm header implementation and the amount of users of the ATerm library

- It may be a good idea to generate Rascal data-types from SDF definitions as an intermediate step, however, Rascal should still implement special code for UPTR trees for performance reasons (unlike Apigen which does not now anything about UPTR).
- The current PDB implementation does type checking at run-time. After implementing an type-checker for Rascal, we can easily add an implementation which does not do type checking at run-time in order to improve performance.
- The immutability feature of Rascal data is implemented in the data-structures and not by the compiler or the interpeter

Interpreter

We just enumerate the thoughts that pop up once in a while:

- Write the interpreter in Java, and use it later to bootstrap the compiler which will be written in Rascal
- Provide a REPL prompt such that experimenting can be done on-the-fly, both on the commandline, and in an Eclipse view.
- "fail" can be implemented using an Java exception, the catch will be at the choice points (switch).
- "return" can also be implemented using a Java exception; remember return can jump out of the context of a visitor that could be nested deeply in the structure of a term or a tree.
- List matching, and especially the kind of backtracking it requires will be implemented using exceptions instead of using continuations.
- Pattern matching needs to be implemented separately for both builtin data-types, abstract data types and concrete parse trees. Possibly using three "adapters" we can factor out the algorithm.
- We use apigen to bootstrap the interpreter. The interpreter will traverse the apigen object trees to implement it's functionality using seperate classes.
- When the compiler is finished and bootstrapped on the interpreter, it may be worthwhile to reimplement/bootstrap the interpreter on the compiler again.

Compiler

The compiler will mainly follow the design of Mark's ASF+SDF compiler, which has proven to generate the fastest code in the world for these kinds of applications. Furthermore, these ideas have popped up:

- Bootstrap the compiler using the interpreter
- Generate as readable function names as possible, mainly taking hints from the Rascal programs and of course from the SDF definitions
- Generate Java code, one class per module
- There is an issue with the globality of rewrite rules, they probably need to be collected and merged into a single factory per application. Rules apparently break modular compilation, especially if you want to optimize matching automatons
- For visitors we could first generate a tree node type reachability graph, and use it to generate a full traversal for a certain visitor. The generated visitor would not recurse into subtrees that will not be visited.

- After generating the visitors, non-recursive visits (i.e. the backbone of the grammar) can be inlined as much as possible to prevent using the stack for visiting trees.
- Inlining in general should be done very aggressively. This will allow other kinds of optimizations, like preventing superfluous condition checking. The ASF+SDF compiler does not do this yet, and it could mean a serious performance improvement. The cost is compilation time obviously, since the Java compiler is going to have to compile a lot more code.
- The simple control flow constructs of Rascal almost map one-to-one to Java
- "fail" is always in the current context/frame, so we need no exception implementation for fail.
- Like in Mark's compiler, list matching is to be implemented using nested while loops.
- "return" can be mapped to normal return statement in Java, except in the context of a visitor, where it should be an exception that is caught by the containing function of the visit, which immediately returns the result in the catch block that surrounds the call to the generated function that implements the visitor.
- Important optimizations:
  - Matching automaton:
    - Sharing prefixes (note that we can not reorder cases of a switch, or the rules?!?)
    - Common subexpression elimination
  - Constant detection and propagation
  - Aggressive inlining (where to stop?)
  - Specialization and instantiation of visitors using grammars and data-type definitions

Issues

- See the section called “*Syntax Patterns*”[8] for a description of patterns. There are still some questions about patterns:
  - Do we want the subexpressions in patterns? [Proposal: no since it complicates the syntax]
  - Do we want string variables in patterns? [Undecided]
  - Do we want to add regular expression matching primitives to patterns? Ex.
    - `[| if @any@ $Stats fi |]`
- How do we identify built-in sorts (bool, int, etc) with their syntactic counterparts?
- In a list comprehension: do list values splice into the list result?
- Ditto for set comprehensions.
- Some clean up of Rscript notation (operators "o" and "x" should go).
- We need an io-library.
- We need memo functions.
- How is lexical matching incorporated?
- Unexplored idea: add (possibly lazy) generators for all types; this allows to generate, for instance, all statements in a program.

- Shopping list of ideas in Tom:
  - Named patterns to avoid building a term, i.e. w@[| while $Exp do $stat od |].
  - Anti-patterns, i.e. the complement of a patterns: ! [| while $Exp do $stat od |] matches anything but a while. (we had that in ASF+SDF too, it's needed)
  - Anonymous variables a la Prolog: [| while $_ do $stat od |].
  - String matching in patterns.
  - Tom uses the notation %[...]% for quoted strings with embedded @...@ constructs that are evaluated. It also has a backquote construct.
- Shopping list of ideas from TXL:
  - "redefine" allows modification of an imported grammar.
  - An "any" sort that matches anything.

Graveyard

Don't read the following sections; they are leftovers from earlier versions of this document but may still contain material that can be reused.

Mapping features to datatypes

Note

This section has played a role during initial design; it is now outdated.
Emphasized cells indicate a new datatype/feature combination that needs to be thought out.

Table 1. Features vs datatypes

Which features work on which datatypes?	**CF syntax trees**	**CF syntax lists**	**Lexical syntax trees**	**Lexical syntax lists**	**Lists**	**Sets**	**Relations**	**Tuples**
Pattern matching	Y (CS)	Y, CS, LM	Y, PS	Y, PS, LM	Y, HT	Y, HT	Y, HT	Y
Pattern construction	Y, CS	Y, PS	Y, PS	Y	Y, HT	Y, HT	Y, HT	Y
Generator/ Comprehension	N	N	N	N	LC	SC	SC	N
Complete Functions	Y	Y	Y	Y	Y	Y	Y	Y
Equations	Y, BC	Y, BC	Y, BC	Y, BC	Y	Y	Y	Y
Polymorphism	N	N	N	N	Y	Y	Y	Y
Serialization	UPTR	UPTR	UPTR	UPTR	Y	Y	Y	Y
Traversal Functions	Y	Y	Y	Y	Y	Y	Y	Y
Subtyping	N	N	N, except character class inclusion	N	Y	Y	Y	Y

- BC = Backward Compatible with ASF

- CS = Concrete Syntax
- HT= head/tail matching
- LC = List comprehension
- LM = List Matching
- N = No
- PS = Prefic Syntax
- SC = Set Comprehension
- Y = Yes

Outdated examples

Generating Graph files in Dot format

This example illustrates the use of a comprehension on the right-hand side of an equation.

```
module Dot-generation

imports Dot-syntax

fun Dot gen-dot(rel[ID, ID] Rel) {
    [| digraph example {
                        $([ node(Tup) | <ID,ID> Tup : Rel ])
                       }
    |]
}

fun DotElem* gen-node(<ID Id1, ID Id2>) {
    [| node $Id1; node $Id2; $Id1 -> $Id2 |}}
}
```

Integration with Tscripts (Outdated)

Note

It is not yet clear whether we will also include Tscript in the integration effort. For the time being, this section is considered outdated.

Introduction

It is possible to speculate on an even further integration of formalisms and combining the above amalgan of ASF+SDF and Rscript with Tscripts.

Requirements

- R12: The resulting language uses a single type system. This means that relational types (possible including syntactic objects) can be used in Tscripts.
- R13: The current "expressions" in Tscript (terms that occur at the rhs of an assignment) are replaced by calls to ASF+SDF or Rscript functions.
- R14: There is minimal duplication in functionality between ASF+SDF/Rscript/Tscript.

Different styles of Type Declarations

We have at the moment, unfortunately, a proliferation of declaration styles for types.

Functions are declared in ASF+SDF as:

```
typecheck(PROGRAM) -> Boolean
```

Observe that only the type of the parameter is given but that it does not have a name.

In Rscript we have:

```
int sum(set[int] SI) = ...
```

while in Tscript processes are declared as

```
process mkWave(N : int) is ...
```

For variables a similar story applies. Variables are declared in ASF+SDF as:

```
"X" [0-9]+    -> INT
```

In Rscript we have:

```
int X

int X : S   (in comprehensions)
```

and in Tscript we have:

```
X : int
```

In order to unify these styles, we might do the following:

- The type of an entity is always written before the entity itself.
- Formal parameters have a name.

In essence, this amounts to using the declaration style as used in Rscript. So we get:

```
Boolean typecheck(PROGRAM P) is ...

process mkWave(int N) is ...
```

Or do we want things like:

```
function typecheck(P : PROGRAM) -> Boolean is ...

var X -> int

process mkWave(int N) is ...
```

Advantages are:

- The category of the entity is immediately clear (function, var, process, tool, ...).
- It is readable to further qualify the category, i.e., traversal function, hidden var, restartable tool)

Global Flow of Control

We have to settle the possible flow of control between the three entities ASF+SDF, Rscript and Tscript. Since Tscript imposes the notion of an atomic action it would be problematic to have completely unrestricted flow of control. Therefore it is logical to use Tscript for the top-level control and to limit the use of ASF+SDF and Rscript to computations within atomic actions. There is no reason to restrict the flow of control between ASF+SDF and Rscript.

What are the consequences of the above choice? Let's analyze two cases:

- Parse a file from within an ASF+SDF specification. This (and similar built-ins) that use the operating system are removed from ASF+SDF. Their effect has to be achieved at the Tscript-level which is the natural place for such primitives.
- Describe I/O for a defined language. Consider Pico extended with a read statement. Here the situation is more complicated. We cannot argue that the flow of control in the Pico program (as determined by an interpreter written in ASF+SDF) should be moved to the Tscript level since Tscript simply does not have the primitives to express this. On the other hand, we have to interrupt the flow of control of the Pico interpreter when we need to execute a read statement. The obvious way to achieve this is
    - At the Tscript level, a loop repeatedly calls the Pico interpreter until it is done.
    - After each call the Pico interpreter returns with either:
        - An indication that the eecution is complete (and possibly a final state and/or final value).
        - An indication that an external action has to be executed, for instance the read statement. This indication should also contain the intermediate state of the interpreter. When the external action has been executed, the Pico interpreter can be restarted with as arguments the value of the external action and the intermediate state.

Experimentation will have to show whether such a framework is acceptable.

Modularization

Rscript and Tscript have no, respectively, very limited mechanisms for modularization. ASF+SDF, however, provides a module mechanism with imports, hiding, parameters and renaming. This mechanism was originally included in ASF, was taken over by SDF and is now reused in ASF+SDF. Currently, there are not yet sufficiently large Rscripts to feel the need for modules. In Tscript, there is a strong need for restricted name spaces and for imposing limitations on name visibility in order to limit the possible interactions of a process with its surroundings and to make it possible to create nested process definitions. What are the design options we have to explore?

First, we can design a new module system that is more suited for our current requirements. The advantage is that we can create an optimal solution, the disadvantage is that there are high costs involved regarding implementation effort and migrations of existing ASF+SDF specifications to the new module scheme.

Second, we can design an add-on to the ASF+SDF module system that addresses our current needs.

Third, we can try to reuse the current ASF+SDF module system.

As a general note, parameterized modules, polymorphics types and renamings are competing features. We should understand what we want. It is likely that we do not need all of them.

Before delving into one of the above alternative approaches, let's list our requirements first.

- We need grammar modules that allow the following operations: import, renaming, deletion (currently not supported but important to have a fixed base grammar on many variations on it). Parameterization and export/hidden: unclear.
- We need function modules (ASF+SDF and Rscript) that provide: import, maybe parametrization, and export/hidden.

- We need process modules (Tscript) that provide: import, export, hiding.