Rascal Requirements and Design Document

Paul Klint
Tijs van der Storm
Jurgen Vinju

Table of Contents

Introduction 2
Rascal for various audiences 3
- Generic arguments 3
- Rascal for ASF+SDF programmers 3
- Rascal for imperative and object-oriented programmers 4

Requirements 5
Rascal at a glance 7
- Modules 7
- Names 8
- Types and Subtypes 8
- Type Equivalence 11
- Attributes 11
- Variables 12
- Functions 13
- Patterns 15
- Expressions 18
- Statements 19
- Rules 20
- Failure and side-effects 21

Examples 21
- Hello world 22
- Table of squares 22
- Table of word counts per file 22
- Booleans 23
- Abstract Graph datatype 25
- Tree traversal 26
- Substitution in Lambda 29
- Renaming in Let 30
- Renaming in Let using globals 31
- Pico Typechecker 32
- Pico evaluator 33
- Pico evaluator with globals 34
- Pico control flow extraction 36
- Pico use def extraction 37
- Pico uninitialized variables 37
- Pico common subsexpression elimination 38
- Pico constant propagation 38
- Pico Reaching definitions 39
- Structured lexicals: numbers 40
- Structured Lexicals: strings 40
- Symbol table with scopes 41

Innerproduct 42
Bubble sort 43
Generic Bubble sort 43
Read-Eval-Print Loop (REPR) [Needs further discussion] 44
The Rascal standard library 45
Main functions 45
Additional functions on lists sets, maps and relations 46
Other functions 47
Source code 47
Syntax Definition 63
Protyping/implementation of Rascal 63
Rascal implementation ideas 64
Data structures 64
Interpreter 65
Compiler 65
Issues 66
Graveyard 67
Mapping features to datatypes 67
Visitor definitions (UNDECIDED and INCOMPLETE) 68
Outdated examples 69
Generating Graph files in Dot format 69
Integration with Tscripts (Outdated) 69
Introduction 70
Requirements 70
Different styles of Type Declarations 70
Global Flow of Control 71
Modularization 71

Note

This document is a braindump of ideas that is slowly converging to a coherent design. See the section called "*Issues*" [66] for the issues that have to be resolved.

Introduction

We have identified the need for better language support for specifying and implementing software analysis and transformation tools. In this document we embark on the design of a domain-specific language that is intended to provide a comprehensive and easy to use set of concepts for that domain. 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.

Rascal for various audiences

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 that 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.

  Caution

  How?

- 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 a body consisting of imperative control flow statements. Adding a function will not trigger the need for regenerating parse tables as is the case in the current ASF+SDF implementation. 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 have 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 during 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
  - graphs
  - 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.

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: File I/O (contradicts R5).
- R6: Easily accessible fact storage (similar to a heap, but remember R5 and the details of backtracking).
- R7: List matching (because of R2, influences R7, but also very handy for manipulating lists in concrete syntax).
- R8: Nesting of data-structures: relations can be nested to model nested features of programming languages (such as scoping), allowing to factor out common code.
- R9: Syntax trees can be elements of the builtin data-structures (but not vice versa).
- R10: 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
- R11: Minimize possible syntactic ambiguities; resolve them by type checking.
- R12: 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)
- R13: no 'null' values, preventing common programming errors
- R14: all values immutable, preventing common programming errors and allowing for certain kinds of optimizations
- R15: 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?)
- R16: can get/set data from databases, such as the pdb from Eclipse IMP, but possibly also from ODBC/JDBC data sources.
- R17: type safe, but flexible. We want a type system that prevents common programming errors, but still allows ample opportunity for reuse.
- R18: syntax safe, programmers should not be allowed to construct programs that are syntactically incorrect w.r.t a certain context-free grammar.
- R19: backtracking safe, programmers should not have to deal with the mind boggling feature interactions between side-effects and backtracking.
- R20: 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.
- R21: 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.
- R22: allow the implementation of reusable modules and functions (i.e. parametric polymorphism and or functions as parameters).
- R23: 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 another Rascal module (suffix: `.ras`) using `import`. Imports are not transitive. We do allow circular imports.
- Import SDF modules (suffix: `.sdf`) using `import`.
- Import Java modules (for the benefit of functions written in Java, suffix `.java`) using `import`.
- Extend another Rascal module using `extend`. This includes a verbatim copy (similar to `#include`) of the-module-to-be-extended in the current module. We allow redefinitions of declared names. We do not allow circular extension.
- 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. We allow several flavours of main function:

  ```
  public void main() { ... }
  ```

  does not give access to program arguments while

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

  gives access to all program arguments. We may add extra types that help in parsing command line arguments.

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 are globally applied.

Modules introduce a namespace and qualified names (using the `::` operator) may be used to uniquely identify elements of a module from the outside. Inside the module, this qualification is implicit. The qualified name consist of two parts: a directory name (a list of names separated by `/`) and a module name.

Rascal modules are located in a file with the name of the module, with suffix `.ras`. They should be located in a directory corresponding to the directory part of the module name. From other modules, `M::F` denotes function `F` from module `M`.

Names

Rascal aims at seamless integration with Java and its names adhere to the following conventions:

- A Rascal Name is identical to a Java Indentifier. All reserved words in both Java and Rascal cannot be used as a Name. *For better syntax errors it is probably better to warn for Java keywords later on.*
- Sorts and Symbols are inherited from SDF but we extend sort (that may only start with an uppercase letter in SDF) to be a more liberal Name. *SDF Sorts may include a dash symbol (-) which is not allowed in Java Identifiers. Forbidding the dash requires changing all existing SDF definitions. Allowing a dash in Names requires an added check in the compiler that Identifiers with dashes are generated as Java names. Suggestion: add dash and check in compiler. Allow - and _ and not $.*

Types and Subtypes

Types

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

- We have a type hierarchy that defines a partial order on types.
- There is a single top for this type hierarchy, it is called "`value`" and a single bottom that is called "`void`".
- There are built-in types (`bool`, `int`, `double`, `str`, `loc`).
- Symbols (as defined by an SDF module) are also types.

  ### Note

  There is no automatic mapping between the built-in types and types generated by an SDF module. The programmer is responsible for conversion. Typical library functions that are helpfull are:

  - `unparseToString` converts any value of an SDF type to a string.
  - `toInt` converts strings to integers.
  - `toDouble` converts strings to doubles.
- 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. 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.
- Types for sets, maps, relations and tuples can be formed from types; Maps, relations and tuples can have optional column names.
- Functions can be polymorphic in their parameters as well as in their return type.
- Function parameters can be function types, meaning that the name of a properly typed function or a locally defined anonymous function can be passed as a parameter.

- The last argument of a function may be of the form `list[T] Name...` and indicates a variable list of remaining arguments of type *T*.
- Data declarations may introduce new structured types and have the form

  ```
  data N Pat1 | Pat2 | ...
  ```

  where N is the name of the datatype and Pat1, Pat2, are prefix patterns describing the variants of the datatype.

  > **Note**
  >
  > We assume that all constructors for a datatype lead to a corresponding function to construct a term of that datatype. This eliminates the need to quote abstract terms in statements.

  > **Caution**
  >
  > Constructors can also generate types.

  > **Note**
  >
  > Constructor names of datatypes may be quoted in order to avoid clashes with reserved words or illegal names.

- We have *views* on data types that define templates or overlays over an existing type. The form is:

  ```
  view Bool <: tree and appl(...) | or appl(...)
  ```

  This defines the type `Bool` as a view on trees with two alternatives named `and` and `or`. Each view should be of type `tree`. Views for the same type are exclusive.
- Types may include type variables like `&T` as in Rscript with the following refinement:
  - In the type `rel[&*T]`, the type variable `&T` is bound to the tuple type of the relation, i.e. an actual type `rel[int,str]` will bind `&*T` to the type `tuple[int,str]`.
  - Composition is only allowed in the result type.
  - In a similar fashion, in `map[&*T]`, the type variable `&*T` is bound to the tuple type of the map.
- A type declaration introduces a name for a new type that is a subtype of a given type, e.g.,

  ```
  type rel[node,node] Graph
  ```

  introduces the type `Graph` as a subtype of `rel[node,node]`. There are no built-in subtypes.
- The name that is introduced by a type declaration may even be parameterized with one or more type variables, e.g.,

  ```
  type rel[&Node, &Node] Graph[&Node]
  ```

  Of course, the type variables that are used in the type in the left part should occur as parameters in the right part of the definition and vice versa.
- Types may be declared only once and are mutually exclusive.

Subtypes

The type hierarchy leads to a subtype relation <: that can be defined (in ASF+SDF) as follows:

```
%% Subtyping rules (<:) on Types
```

```
[let00] void <: T                          = true

[let01] T <: value                         = true

[let02] T <: T                             = true

[let03] T <: & N                           = true

[let04] T <: &N <: T'                      = T <: T'

[let05] N[&T] <: N[&U]                     = true

[let06] list[T] <: list[T']                = T <: T'

[let07] set[T] <: set[T']                  = T <: T'

[let08] tuple[ONTs] <: tuple[ONTs']        = ONTs <: ONTs'

[let09] map[T1, T2] <: map[T1', T2']       = T1 <: T1' &  T2 <: T2'

[let10] map[T1, T2] <: rel[T1', T2']       = T1 <: T1' &  T2 <: T2'

[let11] rel[ONTs] <: rel[ONTs']            = ONTs <: ONTs'

[let12] rel[ONTs] <: set[tuple[ONTs']]     = ONTs <: ONTs'

[let13] set[tuple[ONTs]] <: rel[ONTs']     = ONTs <: ONTs'

[let14] T <: T' == true,        %% covariant
        ONTs' <: ONTs == true   %% contravariant
        ========================================
        fun T N (ONTs) <: fun T' N (ONTs') = true
[default-let]
        T <: T'                            = false
```

Here T, T1, T2, T' etc represent types, and ONTs, ONTs' etc represented lists of OptionallyNamedTypes, e.g., a type that optionally followed by a named. We omit the definition of <= on OptionallyNamedTypes. The above rules also describe certain equivalences between types, e.g., a map can be compared with a relation, a relation can be compared with a set of tuples, and the like.

We throw the names in types away in the subtype relation.

The subtyping relation for function types is tricky and treats result type and argument types differently. We have, for instance, that

```
set[int] f(int i) <: fun set[&T] f(int i)  %% since int < &T

int f(number x) <: fun number f (int x)    %% assuming int <: number
```

but *not* that

```
number f(int x) <: fun number f (number x) %% assuming int <: number
```

In addition to the above rules for subtyping, each type definition introduces an additional subtype relation, e.g., the previous definition for Graph,

```
type rel[node,node] Graph
```

introduces the following subtype relation:

```
Graph <: rel[node,node]
```

Type Equivalence

Types are compatible if they occur on the same path from void to value in the type lattice. This is a middle road between structural and nominal type equivalence, we call it *intensional structural type equivalence*. An example can illustrate this. Assume we have the following definitions for general graphs, control flow graphs and data flow graphs:

```
type rel[node,node] Graph
type Graph CFGraph
type Graph DFGraph
```

`CFGraph` and `Graph` are compatible, and the same holds for `DFGraph` and `Graph`. This implies that the types `CFGraph` and `Graph` (resp. `DFGraph` and `Graph`) can be freely passed as parameter, assigned, returned and value. This is, however, not true for CFGraph and DFGraph since they are incompatible. Passing a value from the one to the other type can only be done via the common type `Graph`. This uncomventional type scheme allows flexible conversion between subtypes (without casting or runtime checks) while preserving the opaqueness of disjoint subtypes of the same ancestor type.

Attributes

Attributes are adornments of data and programs and come in two flavours:

- *value annotations* that allow associating one or more named values to another value.
- *declaration tags* that allow associating one or more named values to a declaration in a Rascal program.

The former are intended to attach application data to values, like adding position information or control flow information to source code or adding visualization information to a relation. The latter are intended to add metadata to a Rascal program and allow to influence the execution of the Rascal program, for instance, by adding memoization hints or database mappings for relations.

Value 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 folllowing three operators are provided for handling annotations:

- `Val ? Anno`: tests whether `Val` has an annotation with name `Anno`.
- `Val @ Anno`: retrieves the value of annotation `Anno` of value `Val`.
- `Var @ Anno = Val`: set the value of annotation `Anno` of the value of variable `Var` to `Val`.

Caution

Do we want a symbol type?

Any value of any named type can be annotated and the type of these annotations can be declared more precisely. For instance, the graph datatype introduced earlier, could be annotated with an indication which graph layout algorithm to apply to a particular graph, e.g.,

```
anno Graph LayoutStrategy "dot" | "tree" | "force" | "hierarchy" | "fisheye"
```

In a similar way, certain syntactic constructs of programs could be annotated with location information:

```
anno STATEMENT posinfo loc
```

or location information could be added for all syntax trees:

```
anno tree posinfo loc
```

Declaration tags

All declarations in a Rascal program may be *preceeded* by one or more declaration tags (`tag`). A tag is defined by declaring its name, the declaration type to which it can be attached, and the name and type of the annotation. The declaration type `all`, makes the declaration tag applicable for all possible declaration types. All declaration tags have the generic format `@Name{ ... }`, with arbitrary text between the brackets that is further constrained by the declared type. Here is an example of a license tag:

```
tag module license str
```

This will allow to write things like:

```
@license{This module is distributed under the GPL}
module Booleans
...
```

Other examples of declaration tags are:

```
tag all todo str             %% a todo note for all declaration types
tag function deprecated void %% marks a deprecated function
tag function memo int        %% bounded memoization of
                             %% function calls
tag all doc str              %% documentation string
```

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, *declared* variables are initialized during the match and their scope is the rule in which they occur. For *used* variables, their value is substituted during the match.
- For variables introduced 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 at the top level of each module and **these initialization may not contain circular calls.**
- 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

Overview

A function declaration consists of a visibility declaration, 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, each terminated by a semi-colon. 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 and these may -- for documentation purposes -- optionally be declared as part of the function signature (i.e., we have unchecked exceptions in contrast with checked exceptions in Java).

Functions with the tag "`java`" have a body written in Java. They have the following properties:

- Arguments and result are pure Rascal values.
- Java functions cannot acces the global state of the Rascal program. Their only interface is via input parameters and a result value. They cannot access Rascal global variables.
- Side effects caused by Java functions in the Java state, are not undone in the case of backtracking.

Note

These Java functions are primarily intended for implementing library functions and for interfacing with Eclipse.

Functions with the tag `@memo` are memo functions that cache previous arguments/result combinations.

Overloading

Functions may be *overloaded*, i.e., functions with the same name but with different argument and result types can be declared. Overloading is subject to the restriction that if several versions of a function are declared with the same name, then one or more of the following conditions should hold:

- the number of arguments differs;
- there is at least one argument position with incomparable types w.r.t. the type hierarchy, i.e., not (A <: B) and not (B<:A).

Here are some examples.

Here are, after all the warnings, the original examples:

```
%% OK, since int and double are incomparable

int +(int a, int b) { ... }
double +(double a, double b) { ... }

%% KO, since int is comparable to value
%% (int is -- like all types -- a subtype of value)

int +(value a, value b) { ... }
int +(int a, int b) { ... }

%% OK since for every two definitions selected from this
%% set there is at least one parameter position different;
%% in this example the overloading on result types will never
%% be used to select the function
```

```
int    +(int a, int b)
double +(int a, double b)
double +(double a, int b)
double +(double a, double b)

%% KO, since arguments cannot be resolved.

int randomValue()
str randomValue()
```

How should overloading be resolved in the presence of type parameters? Our approach is simple but effective: a type parameter `&T` is taken to be of type `value`, unless it is explicitly constrained to be smaller than (or smaller than or equal to) another type U by writing `&T <: U`. For the sake of overloading resolution, a constrained type parameter will be considered to have the type of the constraint, i.e., in `fun F(&T < U)` we assume the `&T` to be of type `U`.

Here are more examples:

```
%% KO, because &T might be any type (i.e. value),
%% which is comparable to int

str f(int x)
str f(&T x)

%% KO, since both T's may bind to comparable types and
%% value and int are comparable

int f(&T x, value y)
int f(&T y, int x)

%% OK, since the second arguments have incomparable types

int f(&T x, int y)
int f(&T y, double x)
```

Formal description of overloading

Given the subtype relation `<=:` defined earlier, we can easily formalize overloading resolution by defining two predicates: `incomparable` describes when two lists of OptionallyNamedTypes can be compared and `may-overload` describes when two function types satisfy the overloading restrictions:

```
%% Incomparable

[com1] incomparable(ONTs, ONTs') =
       not((ONTs <: ONTs') | (ONTs' <: ONTs))

%% May-overload

[mo-1] may-overload(fun T N (ONTs), fun T' N' (ONTs')) =
       not(N == N') |
       incomparable(ONTs, ONTs')
```

Anonymous Functions

We also allow *anonymous functions*, i.e., functions that are declared locally and can be passed as argument to another function, be returned as value of a function, or even be stored as value in a set or relation.

Caution

Causes aliasing; How doe this interfere with backtracking? We may forbid write access to globals outside the current context.

When such functions are called they are called in the lexical scope in which they were defined. For instance, the Rascal standard Library defines a `mapper` function that applies a function to a list or set. An anonymous function can be used to define such function arguments. Here is a function `addOne`, that adds 1 to each element of its argument list:

```
list[int] addOne(list[int] L)
    return mapper(L, fun int (int N) { return N +1 })
```

```
fun fun(int) -> int makeAdder(int n)
    return fun int (int x) { return x + 1 }
```

Patterns

We distinguish four 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.
- *Lexical* patterns: a special case of syntax patterns dealing with lexical notions like identifiers, numeric constants, and the like.

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:
  - In a case construct, immediately after the `case`, `acase` or `rcase` keyword.
  - The left-hand side of a `rules` or `arules` 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:
  - In a case construct, immediately after the `=>` operator.
  - The right-hand side of a `rules` or `arules` statement.
  - As an ordinary expression in the form of a quoted syntax pattern, or a call to a constructor for a dataterm.

A pattern may be turned into an *anti-pattern* by prefixing it with the symbol `!`. An anti-pattern matches in all cases where the original pattern does not match. A match of an anti-pattern cannot bind any variables but these variables are nonetheless reserved in the corresponding scope.

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. Type declarations define an inclusion relation between types.

Regular Expression Patterns (or operator)?

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. Regular expression patterns bind variables of type str when the match succeeds, otherwise they do not bind anything. Their syntax and semantics parallels abstract and concrete syntax patterns as much as possible.

Caution

Is a regular expression a pattern or an expression. If it is a pattern, it may occur on the left-hand side of the matching operator as well as in visit and match statements. In this case, the pattern works by returning the bindings of the first match it can find. To find all occurrences of a pattern in a string, the programmer should program a cursor or something similar around it.

An alternative is that regular expressions are not patterns, but functions or operators. A regular expression is a function that takes a string and returns a binding. In this case we can also use regular expressions as generators which take a string and return all possible bindings.

Caution

We use the regular expression language of Java regex.

Caution

Note that Java regular expressions might have optional groups, which may introduce null bindings. Since null pointers are not allowed in Rascal, we have to limit the kinds of expressions people can write here.

Caution

Note: we do not allow nesting of named groups.

Caution

We allow named groups, syntax (name:regex), which introduce a variable of type str named 'name'.

Caution

Named groups have to outermost, such that they can only bind in one way.

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:

- A *pattern variable declaration*, written as `<TYPE NAME>` declares a new variable with a scope determined by the syntactic context of the pattern.
- A *pattern variable use*, written as `<NAME>`, uses the value of an already declared variabele during the use of the pattern.
- Quoted patterns enclosed between `[|` and `|]`. Inside a fully quoted string, the characters `<`, `>` and `|` can be escaped as `\<`, `\>`, `\|`. Fully quoted patterns may contain pattern variable declarations and pattern variable uses.
- A quoted pattern may be optionally preceeded by an SDF symbol to define its desired syntactic type.

- Unquoted patterns are (unquoted) syntax fragments that may contain pattern variable declarations and pattern variable uses.

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

Examples are:

- Quoted syntax pattern with two pattern variable declarations:

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

- Quoted syntax pattern with two pattern variable uses:

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

- Identical to the previous example, but with a declaration of the desired syntactic type:

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

- Unquoted syntax pattern with two pattern variable declarations:

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

- Unquoted syntax pattern with two pattern variable uses:

```
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.

Note

Implementation hint (used to check the examples in this document). For every sort S in the syntax definition add the following rules:

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

Lexical Patterns

A special case of syntax patterns, are lexical patterns that describe lexical notions such as identifiers, numeric constants and the like. Lexical patterns can appear as part of a concrete syntax pattern. Our solution is a middle road between the original solution used in ASF+SDF (simple but not type safe) and the current solution in ASF+SDF (complex but type safe). The key idea is that a constructor function is implicitly created for every lexical definition, i.e., when the SDF definition defines the lexical sort `LS`, then the following lexical constructor function `ls` (the sort name in lower case) is implicitly defined:

```
    ls( LEXARGS ) -> LS
```

`LEXARGS` consist of zero or more:

- String constants.
- Typed Variables enclosed in angle brackets (as used in other patterns).

The `LEXARGS` argument combined should form a strings of sort `LS`.

Note

The current solution in ASF+SDF requires that all intermediate lexical constructors are explicitly written in the pattern. Here we relax this requirement.

For instance, given a lexical syntax for numbers (part of an SDF definition):

```
sorts DIGIT NAT-CON
lexical syntax
   [0-9]    -> DIGIT
   DIGIT+   -> NAT-CON
```

A rule that would remove leading zeros looks like this:

Caution

Check how ALL SDF types behave in our type system. Results:

- Charclasses are included in Symbol (and hence in Type).
- Single and double quoted literals are not yet included.
- Other cases for Symbols not yet checked.

```
  natcon("0" <DIGIT+ Ds>) => natcon(<Ds>)
```

Note

Unlike the ASF+SDF solution, it is not necessary to make a distinction between ordinary variables, lexical variables, and layout variables. Here, all variables are treated equal and their syntactic position determines how they are used.

We also allow arbitrary character classes as type in a pattern variable, e.g., one could also write:

```
  natcon("0" <[0-9]+ Ds>) => natcon(<Ds>)
```

Expressions

Expressions correspond roughly to Rscript expressions with some extensions:

- There are lists, sets, maps and relations together with comprehensions for these types.
- Generators in comprehensions may range over syntax trees.
- Generators in comprehension may range over integer ranges of the form `[B..E]` or `[B,I,..E]`, where `B` indicates the begin value, `E` the end value, and `I` the optional increment between values.
- 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 options (`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. Note that with each operator, we also associate a named function that can be used as function argument.
  - `+` is used for addition, union (|) and concatenation (++)
  - `-` is used for substraction, difference (\)
  - `*` is used for multiplication.

Caution

product => *

- `/` is used for integer and double division.
- `&` is used for intersection.
- There are new assignment operators +=, -=, *=, `&=`.
- The relational operators `==` (equal) and `<` (less) are available for values of all types. They are used to define `<=` (less or equal), `>` (greater) and `>=` (greater or equal) as well as the functions `min` and `max`.
- The predicates `and`, `or`, `not` and `implies` are written as such (and not using operator symbols)

Caution

&&, ||, ->, !

- 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*.

- An index operator `R[N]` can be applied to values of types list, tuple, string and returns the *N*-th element. It thows the exception `out_of_range(str msg)` when the index value if out of range.
- The Rscript image notation `R[-,n]` is 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.

Caution

yield => replace? subst?

- A `fail` statement which jumps to the deepest nested choice point (i.e. a switch or a visit) and tries the next option available. Both yield and fail should occur in the lexical context of a choice point.

- 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
  ```

  Depending on the kind of pattern we use a different keyword:

  - `case` for (concrete) syntax patterns.
  - `acase` for abstract patterns.
  - `rcase` for regular patterns.

  > **Note**
  >
  > These three different keywords are not very elegant but they help to remove disambiguities.

- 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 exception raised by that statement.

Rules

> **Caution**
>
> We have not defined the rules, we do use them in examples.

Caution

The syntax of rules should reflect their ungrouped and non-deterministic natur

- Rules are applied non-deterministically, and in any order of matching
- The right hand side of rule declarations can contain fail statements, which cause backtracking over the alternative rules for a certain constructor.

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 (of local and global variables) are undone.
- 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.
- The possible choice points are:
  - case in a switch, visit statement.
  - match of left-hand side of rule, arule.
  - Selection of a binding of list variables during list matching, except if there is one possibility left.
  - Selection of an element by a generator.
- The possible failure points are:
  - fail statement
  - a predicate used as generator in a comprehension.
- The possible success points are:
  - return (jumps out of function scope and pops all choice points).
  - yield/replace (jumps to enclosing switch or visit or rule choice)
  - true predicate in comprehension (jumos to next assignment of generator).
- On failure, the currently active choice point is used to undo side-effects and to continue execution.
- Side effects caused by built-in functions (like file i/o, sockhet communication, etc.) are **not** undone.

Examples

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

Hello world

The ritual first example:

```
module Hello

public void main() {
   println("Hello, this is my first Rascal program");
}
```

Table of squares

Another ritual example, printing a table of squares:

```
module Squares

public void main(list[str] argv){
  int N = toInt(argv[0]);
  map[int,int] squares = {};
  for(int I : [1..N]){
     squares[I] = I * I
  }
  println("Table of squares from 1 to ", N);
  println(squares);
}
```

Caution

We need a mechanism to insert values in strings, e.g.

```
 println("Table of squares from 1 to <N>");
```

Caution

#{ ...} toelaten?

Caution

What is the syntax of a map value: { x=>y } or { x : y} ??

Caution

Map comprehensions should also be possible as in { int x : int y | x : {1,2,3}, y : {1,2,3} }

Caution

Maps should also be generators, but notice the overloading of :. { <int x, int y> | x:y : { 1:2 } } or { <int x, int y> | x=>y : { 1=>2 } }

Table of word counts per file

```
module WordCount

public void main(list[str] argv){
  map[str, int] counts = {};
  for(str fileName : argv){
    try {
      counts[fileName] = wordCount(readFile(fileName));
```

```
    }
    finally: println("Skipping file <fileName>");
  };

  int total = sum(range(counts));
  println("In total <total> words in all files");
  println("Word counts per file: <counts>");
}

int wordCount(str S){
  switch S {
    rcase list[str] words = /\<(.*)\>/:  // caution: this example does not work
        return size(words)
  }
}

int alternativeWordCount(str S){ // this example is ugly, since the programmer
  int count = 0;

  while (S != "") {
    switch (S) {
      case /.*(woord:\<.*\>)(rest:.*)/ : { count++; S = rest; } // groups have
      default: return count;
    }
  }

  while (/^.*?(woord:\w+)(rest:.*)/m := S) {
    count += 1;
    S = rest;
  }


  return count;
}
```

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 several variations.

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}
```

Concrete Booleans defined with visit

Using the visit construct that we have described above, we can write the definitions for the functions `&` and `|` as follows:

```
module Bool-examples1

import languages/Booleans/syntax;

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

      case true | true        => true
      case true | false       => true
      case false | true       => true
      case false | false      => false
    }
}
```

Observe that there are two styles:

- In the definition for `&`, we use variables on the left-hand side: the visit is needed to fully normalize the result.
- In the definition of `|`, we use a truth table.

Abstract Booleans defined with visit

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 L, Bool R);
data Bool bor(Bool L, Bool R);

Bool reduce(Bool B) {
    visit bottom-up B {
      acase band(btrue, <Bool B2>)   => <B2>     %% Use Variables
      acase band(bfalse, <Bool B2>)  => bfalse

      acase bor(btrue, btrue)        => Bool-abstract::btrue  %% Use a truth ta
      acase bor(btrue, bfalse)       => btrue
      acase bor(bfalse, btrue)       => btrue
      acase 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.

Abstract Booleans defined with arules

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:

```
module Bool-rules

data Bool btrue;
```

```
data Bool bfalse;
data Bool band(Bool L, Bool R);
data Bool bor(Bool L, Bool R);

arule a1 band(btrue, <Bool B2>)  => <B2>
arule a2 band(bfalse, <Bool B2>) => bfalse

arule o1 bor(btrue, btrue)       => btrue
arule o2 bor(btrue, bfalse)      => btrue
arule o3 bor(bfalse, btrue)      => btrue
arule o4 bor(bfalse, bfalse)     => bfalse
```

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.

Concrete Booleans defined with rules

In a similar fashion, the concrete syntax version of Booleans can be defined using rules:

```
module Bool-conc-rules

import languages/Booleans/syntax;

rule a1 true & <Bool B2>   => <B2>
rule a2 false & <Bool B2>  => false

rule o1 true | true        => true
rule o2 true | false       => true
rule o3 false | true       => true
rule o4 false | false      => false
```

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 attributes);

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, value 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(value 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
import BTree-syntax;

%% Ex1: Count leaves in a BTREE
%% Idea: int N : T generates alle Integer leaves in the tree
%% Observe that there is no need to touch the contents of each Integer
%% since we only count them.

int cnt(BTREE T) {
    return size({N | Integer N : T});
}

%% Ex1: alternative solution using a visit statement

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 convert between
%% - the syntactic sort "Integer"
%% - the built-in sort "int"
%% We use the toInt function that attempts convert any tree
%% to an int.

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

%% Ex2: using visit statement

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

%% Ex3: Increment all leaves in a BTREE
%% Idea: using the visit statement visit all leaves in
%% the tree T that match an integer and replace each N in T by N+1.
%% The expression as a whole returns the modified tree
%% Note that two conversions are needed here:
%% - from Integer to int (using toInt)
%% - from int back to Integer (using parseString)
```

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

%% Ex4: full replacement of g by i
%% The whole repertoire of ASF+SDF 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.

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

%% Ex5: Deep replacement of g by i

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

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

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

%% Ex7: We can also add the top-down-break directive to the generator
%% to get only outermost nodes.

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
tuple[int, BTREE] count_and_inc(BTREE T, int D) {
    int C = 0;

    visit T {
      case <Integer N>: { C = C + 1;
                          int N1 = toInt(N) + D;
```

```
                            yield parse(unparseToString(N1));
                          }
     }
     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

import Lambda-syntax;

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

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

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

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

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

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

    switch E2 {
      case <Var V2>: if(V1 == V2){ yield E1 ; } else { yield V2; }

      case <Exp Ea> <Exp Eb>: {
        Exp EaS = subst(V, E, Ea);
```

```
        Exp EbS = subst(V, E, Eb);
        return [| <EaS> <EbS> |];
      }

      case fn <Var V2> => <Var Ea>:
        if (V1 == V2) { yield [| fn <V2> =\> <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 [| 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 [| fn <Var V3> =\> <Exp EaS> |];
        }
    }
}
```

Caution

\ in quoted patterns.

Caution

Ignore layout in SyntaxPatterns.

Caution

Call inside < ... >

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

import 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 unique variables
```

```
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" + toString(Cnt);    %% PARSE!!!!!!!!!!!!!!!!!
         Exp E1R = rename(E1, Rn, Cnt);
         Exp E2R = rename(E2, {<V, Y>} + Rn, Cnt1);
         return [| let <Y> = <E1R>
                   in
                      <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

import 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 = {};

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

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 [| let <Y>= <E1R>
                   in
                      <E2R>
                   end
                |];
          }
```

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

    default: return E;
    }
}
```

Pico Typechecker

The following example shows the tight integration ASF with comprehensions.

```
module Typecheck

import Pico-syntax;
import Errors;

type map[PICO_ID,TYPE] Env;

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 ];
      }
    }
}

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:
        return type_of(Exp, natural, Env) + tcs(Stats, Env);
    }
}

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
import pico/syntax/Pico;

data PICO_VALUE int ival | str sval;

type map[PICO_ID, PICO_VALUE] VEnv;

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

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

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;
    }
}

VEnv evalStatement(STATEMENT Stat, VEnv Env){
    switch Stat {
      case [| <PICO_ID Id> = <EXP Exp> |]: {
```

```
        Env[Id] = evalExp(Exp, Env);
        return Env;
      }

      case STATEMENT[| if <EXP Exp> then <{STATEMENT ";"}* Stats1>
                       else <{STATEMENT ";"}* Stats1> fi |]:{
        PICO_VALUE Val = evalExp(Exp, Env);
        if(Val.ival != 0){
          return evalStatments(Stats1, Env);
        } else {
          return evalStatements(Stats2, Env);
        }
      }

      case STATEMENT[| while <EXP Exp> do <{STATEMENT ";"}* Stats1> od|]: {
        PICO_VALUE Val = evalExp(Exp, Env);
        if(Val.ival != 0){
          return Env;
        } else {
          VEnv Env2 = evalStatements(Stats1, Env);
          return evalStatement(Stat, Env2);
        }
      }
    }
}

PICO_VALUE evalExp(Exp exp, VEnv Env) {
    switch exp {
      case NatCon[| <NatCon N> |]:
           return toInt(unparseToString(N));

      case StrCon[| <StrCon S> |]:
           return unparseToString(S);

      case PICO_ID[| <PICO_ID Id> |]:
           return Env[Id];

      case <EXP exp1> + <EXP exp2>:
           return evalExp(exp1, Env).ival +
                  evalExp(exp2, Env).ival;

      case <EXP exp1> - <EXP exp2>:
           return evalExp(exp1, Env).ival -
                  evalExp(exp2, Env).ival;

      case <EXP exp1> || <EXP exp2>:
           return evalExp(exp1, Env).sval +
                  evalExp(exp2, Env).sval;
   }
}
```

Pico evaluator with globals

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

```
module Pico-eval

import pico/syntax/Pico;
```

```
data PICO_VALUE int ival | str sval;

type map[PICO_ID, PICO_VALUE] VEnv;

global VEnv Env = {};

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

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

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

void evalStatement(STATEMENT Stat){
    global Venv Env;
    switch Stat {
      case [| <PICO_ID Id> = <EXP Exp> |]: {
        Env[Id] = evalExp(Exp);
        return;
      }

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

      case while <EXP Exp> do <{STATEMENT ";"}* Stats1> od:{
        PICO_VALUE Val = evalExp(Exp);
        if(Val.ival != 0){
```

```
          return;
        } else {
          evalStatements(Stats1);
          evalStatement(Stat);
          return;
        }
      }
    }
};

PICO_VALUE evalExp(Exp exp) {
    global Venv Env;
    switch exp {
      case <NatCon N>: toInt(unparseToString(N));

      case <StrCon S>: return unparseToString(S);

      case <PICO-ID Id>: return Env[Id];

      case <EXP exp1> + <EXP exp2>:
           return evalExp(exp1, Env).ival +
                  evalExp(exp2, Env).ival;

      case <EXP exp1> - <EXP exp2>:
           return evalExp(exp1, Env).ival -
                  evalExp(exp2, Env).ival;

      case <EXP exp1> || <EXP exp2>:
          return evalExp(exp1, Env).sval +
                  evalExp(exp2, Env).sval;
   }
}
```

Pico control flow extraction

```
module Pico-controlflow

import pico/syntax/Pico;

data CP EXP exp | STATEMENT stat;

type tuple[set[CP] entry,
           rel[CP,CP] graph,
           set[CP] exit] CFSEGMENT;

CFSEGMENT cflow({STATEMENT ";"}* Stats){
    switch Stats {
      case <STATEMENT Stat> ; <{STATEMENT ";"}* Stats2>: {
           CFSEGMENT CF1 = cflow(Stat);
           CFSEGMENT CF2 = cflow(Stats2);
           return <CF1.entry,
                   CF1.graph + CF2.graph + product(CF1.exit, CF2.entry),
                   CF2.exit>;
      }

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

```
CFSEGMENT cflow(STATEMENT Stat){
    switch Stat {
      case while <EXP Exp> do <{STATEMENT ";"}* Stats> od : {
           CFSEGMENT CF = cflow(Stats);
           return <{Exp}, product({Exp}, CF.entry) + CF.graph +
                          product(CF.exit, {Exp}),
                   {Exp}>;
      }

      case if <EXP Exp> then <{STATEMENT ";"}* Stats1>
                   else <{STATEMENT ";"}* Stats2> fi: {
           CFSEGMENT CF1 = cflow(Stats1);
           CFSEGMENT CF2 = cflow(Stats2);
           return < {Exp},
                    product({Exp}, CF1.entry) +
                    product({Exp}, CF2.entry) + CF1.graph + CF2.graph,
                    CF1.exit + CF2.exit
                  >;
      }

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

Pico use def extraction

```
module Pico-use-def

import pico/syntax/Pico;

rel[PICO_ID, EXP] uses(PROGRAM P) {
  return {<Id, E> | EXP E : P, PICO_ID Id := E};
}

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

import pico/syntax/Pico;

import Pico-controlflow;
import Pico-use-def;

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

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

PROGRAM cse(PROGRAM P) {
    rel[PICO_ID, STATEMENT] Defs = defs(P);
    rel[CP,CP] Pred = cflow(P).graph;
    map[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.

Pico constant propagation

```
module Pico-constant-propagation

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

Boolean is_constant(EXP E) {
   switch E {
     case <NatCon N> => true

     case <StrCon S> => true

     case <EXP E>    => false
```

```
   }
}

PROGRAM cp(PROGRAM P) {
    rel[PICO_ID, STATEMENT] Defs = defs(P);
    rel[CP,CP] Pred = cflow(P).graph;

    map[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

type tuple[Stat theStat, Var theVar] Def;
type tuple[Stat theStat, Var theVar] Use;

set[Stat] predecessor(rel[Stat,Stat] P, Stat S) {
    return invert(P)[S];
}

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

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>> | tuple[Stat S, Var V]: DEFS};
```

```
    rel[Stat,Def] KILL =
        {<S1, <S2, V>> | tuple[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;
}
```

Structured lexicals: numbers

Given the SDF definition:

```
sorts Digit Number Real
lexical syntax
   [0-9]             -> Digit
   Digit+            -> Number
   Number "." Number -> Real
```

we can write normalization rules for `Number` that remove leading zeros:

```
rules(Number){
   number("0" <Digit+ Ds>) => number(<Ds>)
}
```

Note that a character class can be used instead of the sort `Digit`:

```
rules(Number){
   number("0" <[0-9]+ Ds>) => number(<Ds>)
}
```

A truncation function on `Real` can replace the mantissa by 0:

```
Real truncate(Real R){
    switch R {
       case real(<Number Num> "." <Digit+ Ds>) => real(<Num> "." "0")
    }
}
```

Structured Lexicals: strings

Given the SDF definition:

```
sorts String NQChar
lexical syntax
   ~[\"]             -> NQChar
   [\"] NQChar* [\"] -> String
```

A function that removes the *first* newline from a string can be written as:

```
String removeFirstNL(String S){
    switch S {
      case string("\"" <NQChar* Chars1> "\n" <NQChar* Chars2> "\"") =>
           string("\"" <Chars1> <Chars2> "\"")
    }
}
```

A function that removes all newline from a string:

```
String removeAllNL(String S){
    visit S {
      case string("\"" <NQChar* Chars1> "\n" <NQChar* Chars2> "\"") =>
           string("\"" <Chars1> <Chars2> "\"")
    }
}
```

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 is more intended to explore whether this can be expressed
%% *at all* than that the datatype is well designed.

type rel[&Name, &Value] ScopeMap;
type int ScopeId;
data STable[&Name, &Value] stable(ScopeId scope,
                                  rel[int, ScopeMap] scopes);

%%Create a new, empty,  table
fun STable[&Name, &Value] new_table(){
  return stable(0, {<0, {}>});
}

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

%% Update, in a given scope, the value of a variable
fun STable[&Name, &Value] update(STable[&Name, &Value] ST,
                                 ScopeId scope,
                                 &Name N,
                                 &Value V){
  ST.scopes(scope) = V;
  return new_table(scope, ST.scopes)
}

%% Get, in a given scope, the value of a variable
fun STable[&Name, &Value] value_of(STable[&Name, &Value] ST,
                                   ScopeId scope,
                                   &Name N){
```

```
  return ST.scopes(scope)(N)
}

%% update, in the current scope, the value of a variable
fun STable[&Name, &Value] update(STable[&Name, &Value] ST,
                                 &Name N,
                                 &Value V){
%%  ST.scopes(scope)(N)= V;
  return new_table(scope, ST.scopes)
}

%% Get, in the current scope, the value of a variable
fun STable value_of(STable[&Name, &Value] ST,
                 &Name N){
  return ST.scopes(ST.scope)(N)
}

%% add a new scope and make it the current scope
fun STable[&Name, &Value] new_scope(STable[&Name, &Value] ST){
  ScopeId scope = ST.scope + 1;
  return new_table(scope, ST.scopes);
}

%% switch to another scope
fun STable[&Name, &Value] switch_scope(STable[&Name, &Value] ST,
                                       ScopeId scope){
  return new_table(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

import Vector-syntax;

int innerProduct(Vector V1, V2){
    if ( <Integer N1> <Integer* Rest1> := V1 and
         <Integer N2> <Integer* Rest2> := V2
       )
       return (N1*N2) + innerProduct( [| (<Rest1>) |], [| (<Rest2>) |] )
    else
       return 0
}
```

Bubble sort

[Example taken from TXL documentation]

```
module Bubble

import Vector-syntax;

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

Another way to write this is:

```
module Bubble

Integer* sort(Integer* Numbers){
    visit Numbers {
      case <Integer N1> <Integer N2>:
        if( N1 > N2 )
            yield [| <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 until no more changes are possible.

Caution

In the two above examples it is not elegant that the pattern following the case needs no quoting, and that the pattern following the `return`/`yield` has to be quoted.

Generic Bubble sort

Caution

The current syntax does not yet allow type variables in patterns. What are the implications for the implementation? Do we want this generality? Proposal: no since the ramifications are unclear.

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 <&Elem E2> <&Elem E1>

     }
}
```

Of course, we can write a generic sort on arbitrary lists.

Read-Eval-Print Loop (REPR) [Needs further discussion]

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);

> int count(CU P) {
    int cnt := 0;
>   visit P {
>     Java.Statements.IF => cnt++;
>   }
    return cnt
> }

> count(program)
17
```

There are several innovations here:

- The import associates a name with the imported module.

  ### Caution

  Work out the details.

  ### 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.

  ### Caution

  We need an io library that reads/writes strings and data values. The current version is not yet ok.

- 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.

The Rascal standard library

In this section we summarize the (over 300!) functions in the Rascal Standard Library.

Main functions

The main functions of the library are listed in the following table.

Table 1. Main functions and operators in Rascal Standard Library

	bool	int	double	str	loc	list	set	map	rel	tuple_
_==_	x	x	x	x	x	x	x	x	x	x
_<_	x	x	x	x	x	x	x	x	x	x
_+_	x	x	x	x		x	x			x
_-_	x	x	x				x			
_*_	x	x	x							
_/_		x	x							
_&_							x	x	x	
size				x		x	x	x		x
get				x		x				x
arb	x	x	x			x	x	x	x	
toInt	x	-	x	x						
toDouble	x	x	-	x						
toString	x	x	x	x	x	x	x	x	x	x
toList				x		-	x	x	x	
toSet				x		x	-	x	x	
toMap						x	x	-	x	
toRel						x	x	x	-	
reverse				x		x				
reducer						x	x	x	x	
mapper						x	x	x	x	
min	x	x	x	x		x	x			
max	x	x	x	x		x	x			
sum						x	x			
multiply						x	x			
average						x	x			
get_location					x					
set_location					x					

Notes:

- Operator `==` implements equality on all types. It is extended for data types using structural equality.
- Operator `<` implements less-than on all types.
- Operators +, -, *, / implement these operators for most types. For structured types these operatords exist in three flavours: structured value op struct, structured value op element, and element op structured value.
- `size` gives the number of elements of many types.
- `get` (corresponds to the indexing notation R[N]) gives the N-th element of a structure.
- `arb` generates an arbitrary element from a structured value.
- `toString` converts all values to a string representation.
- `toList`, `toSet`, `toMap`, `toRel` provide conversions between structured types. They may be implemented as just a type conversion (and not a data conversion).
- `reverse` reverses the elements of ordered structured types.
- `reducer` and `mapper` take a function as argument and traverse a structured value.

  Note

  Due to a name class with the type constructor `map`, we use the name `mapper` for a function that is ussually called `map`. The usual function `reduce` is therefore called `reducer`.

- `min` and `max` compute the smallest (largest) of two basic values or all elements of structured values.
- `average` computes the average value of structured values that contain integers or doubles.
- get_location and set_location are access functions for values of type `loc`.

Additional functions on lists sets, maps and relations

Additional functions on lists, sets maps and relations are listed in the following table.

Table 2. Additional functions on lists, sets, maps and relations

	list	set	map	rel	graph
first	x				
rest	x				
makeString	x				
sort	x				
id		x			
power		x			
product			x	x	
compose			x	x	
invert			x	x	
complement			x	x	
domain			x	x	
range			x	x	
carrier			x	x	
domainR			x	x	
rangeR			x	x	
carrierR			x	x	
domainX			x	x	
rangeX			x	x	
carrierX			x	x	
top				x	x
bottom				x	x
reachR				x	x
reachX				x	x

Notes:

- These functions are borrowed from the Rscript library.

Caution

closure is missing

Other functions

The remaining functions take care of annotations, input/output and communication with the global database with program facts, the Program Database (PDB).

Source code

This is a very first draft of the library. When the structure is stable we will split it in modules.

Caution

The following code is maintained externally and then copied :-(to this document, so be carefull with changes.

```
module RascalStandardLibrary
```

```
tag function primitive str;
tag function memo void;
tag function java CompilationUnit;
tag all doc str;

%% Here is a flat list of library functions that will
%% structured in to coherent submodules.

%% --- Comparison operators ----------------------------------
%% Each type defines == and <. The following
%% functions extend them.

public bool <=(&T A, &T B){
  return A < B or A == B;
}

public bool >(&T A, &T B){
  return B < A and not A == B;
}

public bool >=(&T A, &T B){
  return B < A;
}

public &T min(&T A, &T B){
  if(A < B){
    return A;
  } else {
    return B;
  }
}

public &T max(&T A, &T B){
  if(A < B){
    return B;
  } else {
    return A;
  }
}

%% --- Boolean (bool) ----------------------------------------

public bool ==(bool B1, bool B2)
  @doc{Equality on Booleans}
  @primitive{Bool.equal}

public bool <(bool B1, bool B2)
  @primitive{Bool.less}

public bool +(bool B1, bool B2)
  @primitive{Bool.add}

public bool -(bool B1, bool B2)
  @primitive{Bool.sub}

public bool *(bool B1, bool B2)
  @primitive{Bool.mul}
```

```
public int arb()
  @primitive{Bool.arb}

public str toString(bool B)
  @primitive{Bool.toString}

public int toInt(bool B)
  @primitive{Bool.toInt}

public double toDouble(bool B)
  @primitive{Bool.toDouble}

%% --- Integer (int)-----------------------------------------

public bool ==(int I1, int I2)
  @primitive{Int.equal}

public bool <(int I1, int I2)
  @primitive{Int.less}

public int +(int I1, int I2)
  @primitive{Int.add}

public int -(int I1, int I2)
  @primitive{Int.sub}


@primitive{Int.mul}
  public int *(int I1, int I2)

public int /(int I1, int I2)
  throws divide_by_zero(str msg)
  @primitive{Int.div}

public int arb(int bgn, int end)
throws illegal_argument(str msg) %% if(end - bgn <= 0)
  @primitive{Int.arb}

public double toDouble(int I)
  @primitive{Int.toDouble}

public str toString(int I)
  @primitive{toStringInt}

%% --- Double (double) --------------------------------------

public bool ==(double D1, double D2)
  @primitive{Double.equal}

public bool ==(double D, int I){
  return D == toDouble(I);
}

public bool ==(int I, double D){
  return toDouble(I) == D;
}

public bool <(double D1, double D2)
```

```
  @primitive{Double.less}

public bool <(double D, int I){
  return D < toDouble(I);
}

public bool <(int I, double D){
  return toDouble(I) < D;
}

public double +(double D1, double D2)
  @primitive{Double.add}

public bool +(double D, int I){
  return D + toDouble(I);
}

public bool +(int I, double D){
  return toDouble(I) + D;
}

public double -(double D1, double D2)
  @primitive{Double.sub}

public bool -(double D, int I){
  return D - toDouble(I);
}

public bool -(int I, double D){
  return toDouble(I) - D;
}

public double *(double D1, double D2)
  @primitive{Double.mul}

public bool *(double D, int I){
  return D * toDouble(I);
}

public bool *(int I, double D){
  return toDouble(I) * D;
}

public double /(double D1, double D2)
throws divide_by_zero(str msg)
  @primitive{Double.div}

public bool /(double D, int I){
  return D / toDouble(I);
}

public bool /(int I, double D){
  return toDouble(I) / D;
}

public double arb(double bgn, double end)
throws illegal_argument(str msg) %% if(end - bgn <= 0)
  @primitive{Double.arb}
```

```
public int toInt(double D)
  @primitive{Double.toInt}

public str toString(double D)
  @primitive{Double.toString}

%% --- Strings (str) ----------------------------------------

public bool ==(str S1, str S2)
  @primitive{String.equal}

public bool <(str S1, str S2)
  @primitive{String.less}

public str +(str S1, str S2)
  @primitive{String.add}

public int size(str S)
  @primitive{String.size}

public int get(str S, int N)
  throws out_of_range(str msg)
  @primitive{String.get}

public int toInt(str S)
  throws cannot_convert_to_int(str msg)
  @primitive{String.toInt}

public int toDouble(str S)
  throws cannot_convert_to_double(str msg)
  @primitive{String.toDouble}

public list[int] toList(str S)
  throws cannot_convert_int_to_char(str msg)
  @primitive{String.toList}

public set[int] toSet(str S)
  @primitive{String.toSet}

public str reverse(str S)
  @primitive{String.reverse}

%% Add functions for other methods of Java's String class

%% --- Locations ---------------------------------------------

%% The location datatype
data location location(str filename,
                       int beginline,
                       int endline,
                       int begincol,
                       int endcol,
                       int offset,
                       int length);

anno tree posinfo loc;
```

```
public bool ==(loc L1, loc L2)
  @primitive{Location.equal}

public bool <(loc L1, loc L2)
  @primitive{Location.less}

public str toString(loc L)
  @primitive{Location.toString}

public loc get_location(&T Subject)
  throws location_missing(str msg)
  @primitive{Location.get_location}

public &T set_location(&T Subject, loc L)
  @primitive{Location.set_location}

%% --- Lists ---------------------------------------------------

public bool ==(list[&T] L1, list[&T] L2)
  @primitive{List.equal}

public bool <(list[&T] L1, list[&T] L2)
  @primitive{List.less}

public list[&T] +(list[&T] L1, list[&T] L2)
  @primitive{List.add}

public list[&T] +(list[&T] L, &T E){
  return L + [E];
}

public list[&T] +(&T E, list[&T] L){
  return [E] + L;
}

public int size(list[&T] L)
  @primitive{List.size}

public &T get(list[&T] L, int N)
  throws out_of_range(str msg)
  @doc{Get list element: get}
  @primitive{List.get}

public &T arb(list[&T] L)
  @primitive{List.arb}

public str toString(list[&T] L)
  @primitive{List.toString}

public str makeString(list[int] L)
  throws cannot_convert_int_to_char(str msg)
  @primitive{List.makeString}

public set[&T] toSet(list[&T] L)
  @primitive{List.toSet}


public map[&T, &U] toMap(list[tuple[&T, &U]] L)
```

```
  throws domain_not_unique(str msg)
  @primitive{List.toMap}

public rel[&T] toRel(list[&T] L)
  @primitive{List.toRel}

public list[&T] reverse(list[&T] L)
  @primitive{List.reverse}

public &T reducer(list[&T] L, &T F (&T,&T), &T unit){
  &T result = unit;
  for(&T E : L){
    result = F(result, E);
  }
  return result;
}

public list[&T] mapper(list[&T] L, &T F (&T,&T)){
  return [F(E) | &T E : L];
}

public &T min(list[&T] L)
@doc{Minimum element of a list: min}
{
  &T result = arb(L);
  for(&T E : L){
   if(less(E, result)){
      result = min(result, E);
   }
  }
  return result;
}

public &T max(list[&T] L)
@doc{Maximum element of a list: max}
{
  &T result = arb(L);
  for(&T E : L){
   if(less(result, E)){
      result = max(result, E);
   }
  }
  return result;
}

public &T sum(list[&T] L, &T zero)
@doc{Add elements of a List: sum}
{
  return reducer(L, +, zero);
}

public &T multiply(set[&T] R, &T one)
@doc{Multiply elements of a list: multiply}
{
  return reducer(L, *, one);
}

public &T average(list[&T] L, &T zero)
```

```
@doc{Average of elements of a list: average}
{
  return sum(L, zero)/size(L);
}

public &T first(list[&T] L)
  throws empty_list(str msg)
  @doc{First element of list: first}
  @primitive{List.first}

@primitive{List.rest}
@doc{Remaining elements of list: rest}
  public &T rest(list[&T] L)
  throws empty_list(str msg)

public list[&T] sort(list[&T] L, bool less(&T, &T))
  @doc{Sort elements of list: sort}
  @primitive{list.sort}

%% --- Tuples ------------------------------------------------

public bool ==(tuple[&T] R, tuple[&T] S)
  @primitive{Tuple.equal}

public bool <(tuple[&T] R, tuple[&T] S)
  @primitive{Tuple.less}

public tuple[&T + &U] +(tuple[&T] R, tuple[&U] S)
  @primitive{Tuple.add}

public tuple[&T + &U] +(tuple[&T] R, &U E){
  return R + <E>;
}

public tuple[&T + &U] +(&T E, tuple[&U] R){
  return <E> + R;
}

public int size(tuple[&T] R)
  @primitive{Tuple.size}

public value get(tuple[&T] R, int N)
  throws out_of_range(str msg)
  @primitive{Tuple.get}

public str toString(tuple[&T] R)
  @primitive{Tuple.toString}

%% --- Sets  -------------------------------------------------

public bool ==(set[&T] R, set[&T] S)
  @primitive{Set.equal}

public bool <(set[&T] R, set[&T] S)
  @primitive{Set.less}

public set[&T] +(set[&T] R, set[&T] S)
  @doc{Add two sets (set union): operator +}
```

```
  @primitive{Set.add}

public set[&T] +(set[&T] S, &T E){
  return S + {E};
}

public set[&T] +(&T E, set[&T] S){
  return {E} + S;
}

public set[&T] -(set[&T] R, set[&T] S)
  @doc{Substract two sets (set difference): operator -}
  @primitive{Set.sub}

public set[&T] -(set[&T] S, &T E){
  return S - {E};
}

public set[&T] -(&T E, set[&T] S){
  return {E} - S;
}

public set[&T] &(set[&T] R, set[&T] S)
  @doc{Intersection of two sets: operator &}
  @primitive{Set.intersection}

public set[&T] &(set[&T] S, &T E){
  return S & {E};
}

public set[&T] &(&T E, set[&T] S){
  return {E} & S;
}

public int size(set[&T] S)
  @primitive{Set.size}

public &T arb(set[&T] S)
  @primitive{Set.arb}

public str toString(set[&T] S)
  @primitive{Set.toString}

public list[&T] toList(set[&T] S)
  @primitive{List.toList}

public map[&T, &U] toMap(set[tuple[&T, &U]] S)
  throws non_unique_domain(str msg)
  @primitive{Set.toMap}

public rel[&T] toRel(set[&T] S)
   @primitive{Set.toRel}

public &T reducer(set[&T] S, &T F (&T,&T), &T unit){
  &T result = unit;
  for(&T E : S){
    result = F(result, E);
  }
```

```
  return result;
}

public set[&T] mapper(set[&T] S, &T F (&T,&T)){
  return {F(E) | &T E : S};
}

public &T min(set[&T] S)
  @doc{Minimum of a set: min}
{
  &T result = arb(S);
  for(&T E : S){
    result = min(result, E);
  }
  return result;
}

public &T max(set[&T] R)
  @doc{Maximum of a set: max}
{
  &T result = arb(R);
  for(&T E : R){
    result = max(result, E);
  }
  return result;
}

public &T sum(set[&T] S, &T zero)
  @doc{Sum elements of a Set: sum}
{
  return reducer(S, +, zero);
}

public &T multiply(set[&T] S, &T one)
  @doc{Multiply elements of a Set: multiply}
{
  return reducer(S, *, one);
}

public &T average(set[&T] S, &T zero)
  @doc{Average of elements of a set: average}
{
  return sum(S, zero)/size(R);
}

%% TODO

%% Powerset: power0
%%public set[set[&T]] power0(set[&T] R)
%% throw unimplemented("power0")

%% Powerset: power1
%%public set[set[&T]] power1(set[&T] R)
%%  throw unimplemented("power0")

%% --- Maps ----------------------------------------------------

public bool ==(map[&T, &U] M1, map[&T, &U] M2)
```

```
  @primitive{Map.equal}

public bool <(map[&T, &U] M1, map[&T, &U] M2)
  @primitive{Map.less}

public map[&T] +(map[&T] R, map[&T] S)
  @primitive{Map.add}

public map[&T] +(map[&T] S, &T E){
  return S + {E};
}

public map[&T] +(&T E, map[&T] S){
  return {E} + S;
}

public map[&T] -(map[&T] R, map[&T] S)
  @primitive{Map.sub}
  @doc{Substract two maps (set difference): operator -}

public map[&T] -(map[&T] S, &T E){
  return S - {E};
}

public map[&T] -(&T E, map[&T] S){
  return {E} - S;
}

public map[&T] &(map[&T] R, map[&T] S)
  @primitive{Map.mul}
  @doc{Intersection of two maps: operator &}

public map[&T] &(map[&T] S, &T E){
  return S & {E};
}

public map[&T] &(&T E, map[&T] S){
  return {E} & S;
}

public int size(map[&T] M)
  @primitive{Map.size}

public &T arb(map[&T] M)
  @primitive{Map.arb}

public str toString(map[&T] R)
  @primitive{Map.toString}

public list[tuple[&T, &U]] toList(map[&T, &U] M)
  @primitive{Map.toList}

public rel[tuple[&T, &U]] toSet(map[&T, &U] M)
  @primitive{Map.toSet}

public rel[tuple[&T, &U]] toRel(map[&T, &U] M)
  @primitive{Map.toRel}
```

```
%% Is this meaningfull?
public &T reducer(map[&T] S, &T F (&T,&T), &T unit){
  &T result = unit;
  for(&T E : S){
    result = F(result, E);
  }
  return result;
}

public map[&T] mapper(map[&T] M, &T F (&T,&T)){
  return {F(E) | &T E : M};
}

%% --- Relations ---------------------------------------------

public bool ==(rel[&T] R, rel[&T] S)
  @primitive{Rel.equal}

public bool <(rel[&T] R, rel[&T] S)
  @primitive{Rel.less}

public rel[&T] +(rel[&T] R1, rel[&T] R2)
  @doc{Add two relations (union): operator +}
  @primitive{Rel.add}

public rel[&T] +(&T E, rel[&T] R){
  return toRel({E}) + R;
}

public rel[&T] +(rel[&T] R, &T E){
  return R + toRel({E});
}

public rel[&T] -(rel[&T] R1, rel[&T] R2)
  @doc{Substract two relations (difference): operator -}
  @primitive{Rel.sub}

public rel[&T] &(rel[&T] R1, rel[&T] R1)
  @doc{Intersection of two relations: operator &}
  @primitive{Rel.inter}

public int size(rel[&T] R)
  @primitive{Rel.size}

public &T arb(rel[&T] R)
  @primitive{Rel.arb}

public str toString(rel[&T] R)
  @primitive{Rel.toString}

%% Note: in rel[&T], the type variable &T refers
%% to the tuple type of the relation.

public list[&T] toList(rel[&T] R)
  @primitive{Rel.toList}

public set[&T] toSet(rel[&T] R)
  @primitive{Rel.toSet}
```

```
public rel[&T] toRel(set[&T] S)
  @primitive{Set.toRel}

public rel[&T] mapper(rel[&T] R, &T F (&T,&T)){
  return {F(E) | &T E : R};
}

%% Carthesian product of two sets: product (was: x)
public rel[&T1, &T2] product(rel[&T1] R, rel[&T2] S){
  return {<X, Y> | &T1 X : R, &T2 Y : S};
}

%% Compose two relations: compose (was: o)
public rel[&T1, &T3] compose(rel[&T1, &T2] R,
                                rel[&T2, &T3] S){
   return {<X, Z> | tuple[&T1 X, &T2 Y1]: R,
                    tuple[&T2 Y2, &T3 Z]: S, Y1 == Y2};
}

%% Identity relation: id
public rel[&T, &T] id(set[&T] S){
  return { <X, X> | &T X : S};
}

%% Inverse relation: invert (was: inv)
public rel[&T2, &T1] invert (rel[&T1, &T2] R){
  return { <Y, X> | tuple[&T1 X, &T2 Y] : R };
}

%% Complement of relation: complement (was: compl)
public rel[&T1, &T2] complement(rel[&T1, &T2] R){
  return product(domain(R), range(R)) - R;
}

%% Domain of relation: domain
public set[&T1] domain (rel[&T1,&T2] R){
  return { X | tuple[&T1 X, &T2 Y] : R };
}

%% Range of relation: range
public set[&T1] range (rel[&T1,&T2] R){
  return { Y | tuple[&T1 X, &T2 Y] : R };
}

%% Carrier of relation: carrier
public set[&T]  carrier (rel[&T,&T] R) {
  return domain(R) + range(R);
}

%% Domain Restriction of a Relation: domainR
public rel[&T1,&T2] domainR (rel[&T1,&T2] R, set[&T1] S){
  return { <X, Y> | tuple[&T1 X, &T2 Y] : R, X in S };
}

%% Range Restriction of a Relation: rangeR
public rel[&T1,&T2] rangeR (rel[&T1,&T2] R, set[&T2] S){
  return { <X, Y> | tuple[&T1 X, &T2 Y] : R, Y in S };
```

```
}

%% Carrier Restriction of a Relation: carrierR
public rel[&T,&T] carrierR (rel[&T,&T] R, set[&T] S){
  return { <X, Y> | tuple[&T X, &T Y] : R, X in S, Y in S };
}

%% Domain Exclusion of a Relation: domainX
public rel[&T1,&T2] domainX (rel[&T1,&T2] R, set[&T1] S){
  return { <X, Y> | tuple[&T1 X, &T2 Y] : R, X notin S };
}

%% Range Exclusion of a Relation: rangeX
public rel[&T1,&T2] rangeX (rel[&T1,&T2] R, set[&T2] S){
  return { <X, Y> | tuple[&T1 X, &T2 Y] : R, Y notin S };
}

%% Carrier Exclusion of a Relation: carrierX
public rel[&T,&T] carrierX (rel[&T,&T] R, set[&T] S){
  return { <X, Y> | tuple[&T1 X, &T2 Y] : R,
                    X notin S, Y notin S };
}

%% Relations viewed as graphs

type rel[&T,&T] graph[&T];

%% Top of a Graph: top
public set[&T] top(graph[&T] G){
  return domain(G) - range(G);
}

%% Bottom of a Graph: bottom
public set[&T] bottom(graph[&T] G){
  return range(G) - domain(G);
}

%% Reachability with Restriction: reachR
public set[&T] reachR(set[&T] Start, set[&T] Restr,
                          graph[&T] G){
  return range(compose(domainR(G, Start),
                       carrierR(G, Restr)+));
}

%% Reachability with Exclusion: reachX
public set[&T] reachX(set[&T] Start, set[&T] Excl,
                          graph[&T] G){
  return range(compose(domainR(G, Start),
                       carrierX(G, Excl)+));
}

public list[&T] shortestPathPair(&T From, &T To, graph[&T] G)
  @doc{Shortest path between pair of nodes}
  @primitive{Graph.shortestPathPair}

public set[list[&T]] shortestPathFrom(&T From, graph[&T] G)
  @doc{Shortest path between one node and all others}
  @primitive{Graph.shortestPathFrom}
```

```
public set[list[&T]] shortestPathAll(graph[&T] G)
  @doc{Shortest path between all nodes}
  @primitive{Graph.shortestPathAll}

%% TO DO

public rel[&T, &T] closure(rel[&T, &T])
  @primitive{Rel.closure}

%% --- Annotations -------------------------------------------

%% TODO: Sync this with new annotation operators!

public bool has_annotation(&T Subject, str Name)
  @doc{Test whether a named annotation exists}
  @primitive{Annotation.has_annotation}

public value get_annotation(&T Subject, str Name)
  throws missing_annotation(str msg)
  @doc{Get the value of a named annotation}
  @primitive{Annotation.get_annotation}

public map[str,value] get_annotations(&T Subject)
  @doc{Get all annotations}
  @primitive{Annotation.get_annotations}

public &T set_annotation(&T Subject,
                             str Name, value AValue)
  @doc{Set the value of a named annotation}
  @primitive{Annotation.set_annotation}


public &T set_annotations(&T Subject,
                              map[str, value] Annos)
  @doc{Set all annotations}
  @primitive{Annotation.set_annotations}

%% --- Parsing and Unparsing -------------------------------------

public tree parseFile(str filename)
  throws file_does_not_exist(str msg)
  @primitive{Parse.parseFile}

public tree parseString(str source)
  @primitive{Parse.parseString}

public str unparseToString(tree Subject)
  @primitive{Parse.unparseToString}

public str unparseToFile(tree Subject, str filename)
  throws cannot_create(str msg)
  throws write_error(str msg)
  @primitive{Parse.unparseToFile}

%% --- trees -------------------------------------------------

public int toInt(tree Subject)
```

```
  throws cannot_convert(str msg)
  @primitive{Tree.toInt}

public int toDouble(tree Subject)
  throws cannot_convert(str msg)
  @primitive{Tree.toDouble}

public int toString(tree Subject)
  throws cannot_convert(str msg)
  @primitive{Tree.toString}

%% --- io -----------------------------------------------------

public str readFile(str filename)
  throws does_not_exist(str msg)
  throws read_error(str msg)
  @primitive{IO.read}


public &T readTerm(str filename)
  throws does_not_exist(str msg)
  throws read_error(str msg)
  throws term_error(str msg, loc l)
  @primitive{IO.readTerm}


public void write(str filename, &T Subject)
  throws cannot_create(str msg)
  throws write_error(str msg)
  @primitive{IO.write}

public void print(list[value] V...)
  @primitive{IO.print}

public void println(list[value] V...)
  @primitive{IO.println}

%% --- Interface with the Program Database -------------------
%% The following function provide a bare minimum and will
%% have to compared with the current PDB interface.
%% We assume one active PDB that can be opened and closed.
%% Values can be written to and read from the PDB.

%% Idea: it would be nice to model the PDB as a value
%% of type map[str, value] and to access it that way!

public void openPDB(str name)
  throws cannot_open(str msg)
  @primitive{PDB.open}

public void closePDB()
  throws cannot_close(str msg)
  @primitive{PDB.close}

public void writePDB(str name, &T val)
  throws cannot_write(str msg)
  @primitive{PDB.write}
```

```
public &T readPDB(str name)
  throws cannot_read(str msg)
  @primitive{PDB.readPDB}

%% If name is of type set[&T], then the set incr
%% is added to it. Similar for a rel[&T]

public void addSetPDB(str name, set[&T] incr)
  throws does_not_exist(str msg)
  @primitive{PDB.addSet}

public void addRelPDB(str name, rel[&T] incr)
  throws does_not_exist(str msg)
  @primitive{PDB.addRel}
```

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 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 know anything about UPTR).

- The current PDB implementation does type checking at run-time. After implementing a 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 a 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 separate 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
- Rascal functions with a Java body need the following:
  - Generate for each argument an Java argument with appropirtae Java type.
  - For each return statement, check the type of the resulting value against the return type in the function header.
  - Catch any exceptions raised by the Java code, convert them to string and rethrow as Rascal exception.

Issues

- Which comment convention will we use? *Let's use Java style comments.*
- Extensibility of functions. *Proposal: let's not provide facilities for this.*
- 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.
- 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 |].
  - 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 3. 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

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; }
}
```

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.