

The Stratego Reference Manual

Eelco Visser

STRATEGO

For Stratego version 0.4.15

www.stratego-language.org

Department of Information and Computing Sciences
Universiteit Utrecht

June 25, 2000

Copyright © 1998, 1999, 2000 Eelco Visser

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided that they are marked clearly as modified versions, that the author's names and title are unchanged (though subtitles and additional authors' names may be added), and that other clearly marked sections held under separate copyright are reproduced under the conditions given within them, and that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions, except that this permission notice may be stated in a translation approved by the Free Software Foundation.

Address:
Department of Computer Science
Universiteit Utrecht
P.O.Box 80089
3508 TB Utrecht
email: visser@acm.org
<http://www.cs.uu.nl/~visser/>

The Stratego Reference Manual

Eelco Visser

June 25, 2000

Summary

Stratego is a language for the specification of transformation rules and strategies for applying them. Specifications consist of a collection of modules that define the signature of the object language(s) of the transformation, transformation rules and strategies. The Stratego compiler translates specifications to C code. Together with a provided run-time system these generated programs can be used to apply the specified transformations.

This documents provides a reference manual for the language.

CONTENTS

Introduction	3
STRATEGO: SPECIFYING PROGRAM TRANSFORMATION SYSTEMS CONCISELY	3
TRANSFORMING PROGRAMS WITH STRATEGO	5
DESCRIBING STRATEGO WITH SDF	7
Organization	9
ORGANIZING SPECIFICATIONS INTO MODULES	9
LAYING OUT AND COMMENTING SPECIFICATIONS	11
Terms	13
REPRESENTING PROGRAMS BY MEANS OF TERMS	13
REPRESENTING CONSTRUCTORS AND CONSTANTS	15
DECLARING CONSTRUCTORS WITH SIGNATURES	17
Strategies	19
NAMING TRANSFORMATIONS WITH STRATEGY DEFINITIONS	19
THE PRIMITIVES OF TRANSFORMATION ARE MATCHING AND BUILDING PATTERNS	21
COMBINING STRATEGIES	23
TRAVERSING TERMS	25
GENERICALLY TRAVERSING TERMS	27
CARRYING INFORMATION ALONG A TRAVERSAL	29
Rules	30
DEFINING BASIC TRANSFORMATIONS WITH RULES	30
APPLYING STRATEGIES IN TERMS AND MATCHING TERMS IN STRATEGIES	32
STRATEGIC PATTERN MATCHING RULES	34
DESCRIBING DEEPLY EMBEDDED PATTERNS WITH CONTEXTS	36
ABSTRACTING FROM PATTERNS WITH OVERLAYS	38
AS PATTERNS	40
Connecting	41
CONNECTING TO THE WORLD THROUGH PRIMITIVES	41
Related Work	43
REFERENCES TO RELATED WORK	43
BIBLIOGRAPHY	44

STRATEGO: SPECIFYING PROGRAM TRANSFORMATION SYSTEMS CONCISELY

Program transformations can be described concisely by means of a combination of rewrite rules that describe basic transformations and rewrite strategies that describe their application to a program.

Program transformations are often naturally described by means of (term) rewrite rules. However, to achieve transformations of an entire program such rules need to be applied with care. Often a more fine grained control over their application is needed than is offered by standard rewriting strategies such as innermost or outermost normalization.

One solution to this problem is to encode the transformation strategy in the rules. However, this leads to large specifications in which traversal over the syntax structure of a program needs to be spelled out explicitly. This is not necessary in ‘pure’ rewriting, since the built-in strategy finds out where to apply a rule. In addition this style leads to a loss of reusability of transformation rules.

Another solution is to make strategies programmable artifacts of a transformation system. Stratego is a language for the specification of transformation rules and transformation strategies.

The main features of Stratego are summarized in Figure 1.

AUDIENCE

This reference manual describes the Stratego language from first principles. That means that it is organized along the lines of the design of the language rather than along the lines of learning the language. Therefore, it is aimed at Stratego programmers and implementers with some experience with the language who want to understand the details of its definition.

The Stratego Tutorial [10] gives a description of the language and its programming environment by means of examples and is aimed at novice users. Several publications exist that explore aspects of the language and its applications; see the bibliography.

Section: Introduction

- Program represented by means of abstract syntax trees (terms)
- Basic transformations expressed as rewrite rules
- Programmable strategies for control over application of rules
- Matching and building patterns are first-class citizens
- Organization of specifications into modules
- Primitives for connection to the system

Figure 1: Features of Stratego

TRANSFORMING PROGRAMS WITH STRATEGO

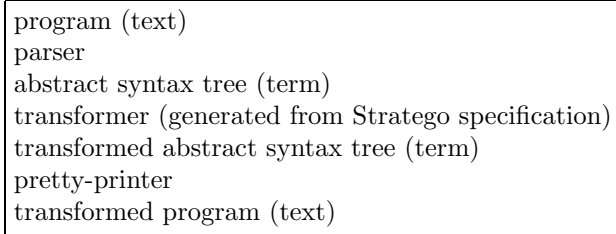
Program transformation systems defined in Stratego are combined with parsers and pretty-printers to create source-to-source transformation systems.

A transformation system defined by a Stratego specification constitutes a program that transforms a term to another term or fails.

Stratego programs define transformations on abstract syntax trees. In order to transform programs that are represented as texts, it is necessary to define a parser that transforms a program text to an abstract syntax tree and a pretty-printer that transforms an abstract syntax tree to text (Figure 2).

This manual will not go into the details of implementing such tools. There is a wide range of possibilities for defining parsers and pretty-printers. One possibility is the use of SDF for parsing and the Box language of the GPP package for pretty-printing. These packages are combined in the XT bundle of transformation tools [1].

Section: Introduction



```
graph TD; A[program (text)] --> B[parser]; B --> C[abstract syntax tree (term)]; C --> D[transformer (generated from Stratego specification)]; D --> E[transformed abstract syntax tree (term)]; E --> F[pretty-printer]; F --> G[transformed program (text)];
```

program (text)
parser
abstract syntax tree (term)
transformer (generated from Stratego specification)
transformed abstract syntax tree (term)
pretty-printer
transformed program (text)

Figure 2: (** Architecture of program transformation systems **)

Section: Introduction

DESCRIBING STRATEGO WITH SDF

This reference manual describes Stratego by means of a formal syntax definition in the syntax definition formalism SDF2. The semantics of the language is described informally.

Section: Introduction

□

Figure 3:

ORGANIZING SPECIFICATIONS INTO MODULES

A Stratego specification defines signatures, rules, strategies and overlays. A specification can be divided into a collection of modules.

A specification consists of a set of basis specifications that define signatures, rules, strategies and overlays (Figure 4). Each of these items is introduced by one of the keywords **signature**, **rules**, **strategies** and **overlays**, followed by a list of basic signatures, rule definitions, strategy definitions or overlays, respectively.

A specification can be divided into a collection of modules (Figure 5). A module has a name and combines a number of module items, which are basic specifications or import declarations. A module with name **Modname** is assumed to reside in a file with name **Modname.r**. Note that **imports**, **signature**, **rules** and **strategies** are reserved words that cannot be used as module names.

An import of the form **imports m1 ... mn** denotes the structural inclusion of the sections of the modules **mi** into the importing module. Here structural inclusion means inclusion at the level of abstract syntax as opposed to textual inclusion as is in the C **#include** statement.

The complete syntax of Stratego is defined in a number of modules, which are combined in module **Stratego-Syntax** (Figure 6).

```
module Stratego-Specifications
imports Stratego-Signatures Stratego-Overlays
       Stratego-Rules Stratego-Strategies
exports
  %%sorts BSpec Spec
  context-free syntax
    "specification" BSpec*  -> Spec  {cons("Specification")}

    "signature"      BSig*   -> BSpec {cons("Signature")}
    "rules"          RDef*   -> BSpec {cons("Rules")}
    "strategies"     SDef*   -> BSpec {cons("Strategies")}
    "overlays"       Overlay* -> BSpec {cons("Overlays")}
```

Figure 4: Syntax of Stratego specifications

```

module Stratego-Modules
imports Stratego-Specifications
exports
  sorts Module %% ModItem
  context-free syntax
    "module" ModName ModItem* -> Module {cons("Module")}
    BSpec                      -> ModItem
    "imports" ModName*         -> ModItem {cons("Imports")}

  lexical syntax
    [A-Za-z] [A-Za-z0-9\_\\-]* -> ModName
    "imports"                  -> ModName {reject}
    "signature"                -> ModName {reject}
    "rules"                    -> ModName {reject}
    "strategies"               -> ModName {reject}

  lexical restrictions
    ModName -/- [A-Za-z0-9\_\\'\-]

```

Figure 5: Syntax of Stratego modules

```

module Stratego-Syntax
imports InstantiateLayout.sdf
      Stratego-Rules.sdf           Stratego-Signatures.sdf
      Stratego-Application.sdf     Stratego-Specifications.sdf
      Stratego-Congruences.sdf     Stratego-Strategic-Rules.sdf
      Stratego-Contexts.sdf        Stratego-Strategies.sdf
      Stratego-Layout.sdf          Stratego-Strategy-Definitions.sdf
      Stratego-Lexicals.sdf        Stratego-Sugar.sdf
      Stratego-Match-Build.sdf     Stratego-Syntax.sdf
      Stratego-Modules.sdf         Stratego-Terms.sdf
      Stratego-Overlays.sdf        Stratego-Traversal.sdf
      Stratego-Primitives.sdf

```

Figure 6: The syntax of Stratego is defined in several modules.

LAYING OUT AND COMMENTING SPECIFICATIONS

The tokens making up a specification can be separated by means of white space and comments.

Spaces, tabs and newlines can be used at will in between tokens of Stratego specifications, but not within tokens (identifiers, keywords, literals, etc.).

In addition various kinds of comments are supported. A small comment on a single line can be inserted after `//`. All characters after `//` until the end of the line are comment.

A larger comment can be written between `(*` and `*)`. No nested occurrences of `*)` are allowed in such comments.

To write large pieces of documenting text, the literate programming convention can be used. A literate Stratego module starts with the keyword `\literate`. Fragments of specification text are enclosed in the phrases `\begin{code}` and `\end{code}`. In between pieces of code one can write arbitrary text. The general structure of a literate Stratego specification is:

```
\literate
  LaTeX text
\begin{code}
  specification text
\end{code}
  LaTeX text
\begin{code}
  specification text
\end{code}
```

Only the code parts of a module are interpreted by the compiler. A literate Stratego specification can be `\input` directly in \LaTeX . The code parts are set in verbatim (provided the right style-file is used), the non-code parts are interpreted as regular \LaTeX material.

Section: Organization

```
module InstantiateLayout
imports Stratego-Layout
exports
  lexical syntax
    Whitespace -> LAYOUT
    Comment    -> LAYOUT
  context-free restrictions
    LAYOUT? -/- [\ \t\n] | [\(\).[\*]
```

Figure 7: Syntax of layout

```
module Stratego-Layout
exports
  %%sorts Identifier Natural Modname Variable String
  lexical syntax
    [\ \t\n]                                -> Whitespace

    "//" ~[\n]*                             -> Comment

    ~[\*]                                   -> CommentChar
    Asterix                                 -> CommentChar
    "(*" CommentChar* ")"                  -> Comment

    ~[]                                     -> LChar
    "\\literate" LChar* "\\begin{code}" -> Comment
    "\\end{code}" LChar* "\\begin{code}" -> Comment

  lexical restrictions
    Asterix -/- [\)]
    LChar+ -/- [\\]. [b]. [e]. [g]. [i]. [n]. [\{]. [c]. [o]. [d]. [e]. [\}]
```

Figure 8: Syntax of layout

Section: Terms

REPRESENTING PROGRAMS BY MEANS OF TERMS

Programs are represented by means of first-order terms.

The abstract syntax of programs to be transformed is represented by means of terms (Figure 9).

a constructor is an identifier

constructor application `Succ(Zero())`

constant (parentheses) `Succ(Zero)`

strings `App(Abs("x", "x"), Var("y"))`

natural number `App(Abs(Var(1)), Var(2))`

LISTS

lists abbreviations for Cons/Nil lists

`[a,b,c|[]] ≡ [a,b,c]`

TUPLES

tuples are abbreviations for ...

ATERMS

Stratego terms are implemented by means of ATerms

Section: Terms

```
module Stratego-Terms
imports Stratego-Lexicals
exports
  %%sorts Term
  context-free syntax
  Identifier          -> Term {cons("Var")}
  Identifier "(" {Term ","}* ")" -> Term {cons("Op")}
  String              -> Term {cons("Str")}
  Natural             -> Term {cons("Int")}
  "[" {Term ","}* ("|" Term)? "]" -> Term {cons("List")}
  "(" {Term ","}* ")" -> Term {cons("Tuple")}
```

Figure 9: Syntax of terms

Section: Terms

REPRESENTING CONSTRUCTORS AND CONSTANTS

The symbols are divided into identifiers, natural numbers and strings.
--

Figure 10 defines the syntax of the lexical tokens of Stratego.

Identifiers start with a letter and are followed by zero or more letters, digits, underscores, primes or dashes. An identifier may not be followed by any of these characters.

A natural number is a string of one or more decimal digits.

A string is list of characters enclosed by double quotes.

Figure 10 also defines the keywords of Stratego, i.e., the words that cannot be used as identifiers.

```

module Stratego-Lexicals
exports
  lexical syntax
    [A-Za-z] [A-Za-z0-9\_\'\\-]* -> Identifier
    [0-9]+ -> Natural
    "\"" ~["\\n"]* "\"" -> String

  lexical restrictions
    Identifier -/- [A-Za-z0-9\_\'\\-]
    Natural -/- [0-9]

  lexical syntax
    "module" -> Identifier {reject}
    "signature" -> Identifier {reject}
    "sorts" -> Identifier {reject}
    "operations" -> Identifier {reject}
    "rules" -> Identifier {reject}
    "strategies" -> Identifier {reject}
    "id" -> Identifier {reject}
    "fail" -> Identifier {reject}
    "all" -> Identifier {reject}
    "some" -> Identifier {reject}
    "one" -> Identifier {reject}
    "thread" -> Identifier {reject}
    "not" -> Identifier {reject}
    "test" -> Identifier {reject}
    "where" -> Identifier {reject}
    "rec" -> Identifier {reject}
    "let" -> Identifier {reject}
    "imports" -> Identifier {reject}

```

Figure 10: The syntax of lexicals

Section: Terms

DECLARING CONSTRUCTORS WITH SIGNATURES

Constructors are declared by means of signatures

A signature is of the form

```
signature
  sorts Sort ... Sort
  operations OpDecl ... OpDecl
```

A constructor declaration of the form

```
C : Sort
```

declares a nullary constructor (a constant) `C` of type `Sort`. A declaration of the form

```
C : Sort1 * ... * Sortn -> Sort
```

declares an n -ary constructor `C` that takes a tuple of terms (t_1, \dots, t_n) where t_i is of sort `Sorti` into a term `C(t1, ..., tn)` of sort `Sort`.

Predefined sorts are `String` denoting strings of characters between double quotes, and `Int` denoting natural numbers in decimal notation.

For example, the following signature describes a language of lambda expressions.

```
signature
  sorts Exp
  operations
    Var      : String -> Exp
    Apply    : Exp * Exp -> Exp
    Lambda   : String * Exp -> Exp
```

Terms with variables are only used in a specification. Transformations apply only to ground terms.

nullary constructors that are not declared are variables

(At this point typechecking is not done by the implementation. It is nonetheless important to write signatures because the definition of the congruence operators is derived from them.)

derivation of congruence operators

```

module Stratego-Signatures
exports
  %%sorts BSig SortDecl OpDecl Type
  context-free syntax
  "signature" BSig*      -> BSpec    {cons("Signature")}
  "sorts"      SortDecl* -> BSig     {cons("Sorts")}
  "operations" OpDecl*   -> BSig     {cons("Operations")}

  Identifier "(" {Identifier ","}* ")"?
                                -> SortDecl {cons("Sort")}

  Identifier ":" Type          -> OpDecl   {cons("OpDecl")}
  Term                        -> Type      {cons("ConstType")}
  {Term "*" }* "->" Term      -> Type      {cons("FunType")}

```

Figure 11: Syntax of Signatures

NAMING TRANSFORMATIONS WITH STRATEGY DEFINITIONS

A specification defines a strategy for transforming terms. Strategies can be named by means of strategy definitions.

The purpose of a Stratego specification is the definition of a *strategy*, i.e., a program that transforms a term into another term (or fails). Strategies are built from primitive transformations using strategy operators. There are a number of standard operators. Other operators can be defined in terms of the primitive operators by means of strategy definitions.

A strategy section of a specification has the form

```
strategies
  StrategyDefinition*
```

and consists of a list of strategy definitions that give names to strategy expressions.

A strategy definition has the form

```
Id(Id1,...,Idn) = Strategy
```

Here *Id* is the strategy operator defined by the definition, the *Idi* are strategy arguments to the operator and *Strategy* is the body of the definition. A call *Id(s1,...,sn)* to this operator is equivalent to the body of the definition with the *si* substituted for the *Idi*. Definitions cannot be recursive. Recursion is defined using the recursion operator.

For example,

```
strategies
  repeat(s) = rec x(s; x <+ id)
```

defines the operator *repeat*.

A defined operator can be used by calling it

```
(** OVERLOADING **)
```

Operators can be overloaded

```
(** MULTIPLE DEFINITIONS **)
```

in case of multiple definitions for operators *with the same arity* the bodies are shared.

Section: Strategies

```
module Stratego-Strategy-Definitions
exports
  context-free syntax

  "strategies" SDef* -> BSpec {cons("Strategies")}

  %% calling a strategy

  Identifier                               -> SVar {cons("SVar")}
  SVar "(" {Strat ","}* ")"? -> Strat {cons("Call")}

  %% definition of a nullary strategy operators

  Identifier "=" Strat -> SDef

  %% definition of a parameterized strategy operators

  Identifier "(" {Identifier ","}* ")"
                =" Strat -> SDef {cons("SDef")}

  %% local strategy definition
  %% Note: not yet supported in stratego-0.4.*

  "let" SDef "in" Strat -> Strat {cons("Let")}
```

Figure 12: Syntax of strategy definitions

Section: Strategies

THE PRIMITIVES OF TRANSFORMATION ARE MATCHING AND BUILDING PATTERNS

Rewrite rules are defined in terms of more primitive actions, i.e., matching terms against patterns and building instantiations of patterns.

a strategy transforms a term

there is always a term that is the subject of transformation

a term pattern is a term with variables

MATCH

?**t** matches the subject term against the pattern **t**

if **t** is a ground term, this entails that the subject term is equal to **t**

if **t** contains variables, this means that the subject term corresponds to **t** as far as the pattern goes. at the positions where **t** has variables any term can occur. the variable are bound to the terms at the corresponding positions in the subject term.

Example: ?App(Abs(**x**, **e1**), **e2**)

BUILD

!**t** builds an instantiation of the term pattern **t**, i.e., the subject term is replaced by (an instantiation of) **t**.

if **t** contains no variables then the subject term is replaced by **t**

if **t** contains variables then the subject term is replaced by **t**, where the variables in **t** are replaced by there bindings

Example: !Let(**x**, **e2**, **e1**)

TRANSFORMATION RULES

A transformation rule first matches the subject term against a pattern and then replaces it with the instantiatio of another pattern. This can be formulated as the sequential composition of a match and a build. For example,

strategies

Beta = ?App(Abs(**x**, **e1**), **e2**); !Let(**x**, **e2**, **e1**)

SCOPE

In the definition above the variables **x**, **e1** and **e2** are implicitly scoped by definition

variable scope: {**x1**, ..., **xn**: **s**}

For example, definition with explicit scope

Section: Strategies

```
strategies
  Beta = {x, e1, e2: ?App(Abs(x, e1), e2); !Let(x, e2, e1)}
```

```
module Stratego-Match-Build
exports
  context-free syntax
    "?" Term -> Strat {cons("Match")}
    "!" Term -> Strat {cons("Build")}
    "{" {Identifier ","}+ ":" Strat "}" -> Strat {cons("Scope")}
    "where" "(" Strat ")" -> Strat {cons("Where")}
    "_" -> Term {cons("Wld")}
```

Figure 13: Syntax of match and build primitives

COMBINING STRATEGIES

Basic strategies can be combined into more complex strategies by means of a number of primitive strategy operators.

SUCCESS AND FAILURE

The operator `id` always succeeds without any transformation. The operator `fail` always fails.

SEQUENTIAL COMPOSITION

If `s1` and `s2` are strategies, then `s1 ; s2` is the strategy that first applies `s1` and if that succeeds applies `s2` to the result.

NON-DETERMINISTIC CHOICE

If `s1` and `s2` are strategies, then `s1 + s2` is the strategy that chooses either `s1` or `s2`, but such that the strategy chosen succeeds.

DETERMINISTIC CHOICE

If `s1` and `s2` are strategies, then `s1 <+ s2` is the strategy that first tries `s1` and if that succeeds is done. If `s1` fails, then `s2` is applied to the original term. The operator is also called *left choice* because it prefers its left argument. An example of the use of left choice is the definition of `try`:

$$\text{try}(s) = s <+ \text{id}$$

The strategy `try(s)` succeeds with the result of `s` if `s` succeeds. Otherwise, if `s` fails, `s` succeeds without any effect.

RECURSION

If `s` is a strategy, then `rec x. s` is the strategy that at each point in `s` where the strategy variable `x` is called, calls itself. An example of the use of recursion is the definition of `repeat`:

$$\text{repeat}(s) = \text{rec } x \ ((s; x) <+ \text{id})$$

The strategy `repeat(s)` applies the strategy `s` as long as it succeeds.

TEST

If `s` is a strategy, then `test(s)` is the strategy that succeeds if `s` succeeds, but throws away the effect of its transformation.

NEGATION

If `s` is a strategy, then `not(s)` is the strategy that fails if `s` succeeds and succeeds if `s` fails.

OPERATOR PRECEDENCE

Section: Strategies

Operators in decreasing binding strength

- ;
- +, <+ (mutually right associative)

In other words `a ; b + c <+ d ; e` is parsed as `(a ; b) + (c <+ (d ; e))`

```
module Stratego-Strategies
imports Stratego-Terms
exports
  %%sorts Strat SVar
  context-free syntax
    "id"                -> Strat {cons("Id")}
    "fail"              -> Strat {cons("Fail")}
    "test" "(" Strat ")" -> Strat {cons("Test")}
    "not"  "(" Strat ")" -> Strat {cons("Not")}
    Strat ";" Strat      -> Strat {cons("Seq"), left}
    Strat "+" Strat      -> Strat {cons("Choice"), left}
    Strat "<+" Strat      -> Strat {cons("LChoice"), left}
    "rec" SVar "(" Strat ")" -> Strat {cons("Rec")}
    "(" Strat ")"        -> Strat {bracket}
  context-free priorities
    Strat ";" Strat -> Strat
  > {left:
    Strat "+" Strat -> Strat
    Strat "<+" Strat -> Strat }
```

Figure 14: Syntax of sequential programming operators

TRAVERSING TERMS

To apply transformations below the root of a term it is necessary to traverse it. Stratego provides a number of traversal operators.

The operators defined above combine strategies that apply transformations to the root of a term into strategies that repeatedly apply transformations to the root of a term. In order to achieve transformations throughout a term we need operators to traverse the term.

CONGRUENCE

If C is an n -ary operator defined in the signature, and s_1, \dots, s_n are strategies, then $C(s_1, \dots, s_n)$ is a strategy that only applies to C terms and then applies s_i to the i -th. For example, consider the following signature defining the list constructors `Nil` and `Cons`:

```
signature
operations
  Nil  : List(a)
  Cons : a * List(a) -> List(a)
```

The strategy `Nil` applies to the term `Nil` without any change and fails for all other terms. The strategy `Cons(s1, s2)` applies to a term `Cons(t1, t2)` if s_i applies to t_i with result t_i' and has as result `Cons(t1', t2')`. Using these operators we can define the operator `map` as:

```
map(s) = rec x(Nil + Cons(s, x))
```

The strategy `map(s)` applies a strategy s to each element of a list. Observe how recursion is used to apply the strategy to the tail of the list.

CONGRUENCES ON STRINGS AND NUMBERS

each string and numeric constant is congruence on itself

CONGRUENCES ON LISTS

list congruences can also be defined using the `[]` syntax for lists.

for example,

```
map(s) = rec x([] + [s|x])
```

CONGRUENCES ON TUPLES

```

module Stratego-Congruences
exports
  context-free syntax
    String                                -> Strat {cons("StrCong")}
    Natural                              -> Strat {cons("IntCong")}
    "(" Strat "," {Strat ","}+ ")" -> Strat {cons("TupleCong")}
    "[" {Strat ","}* "]"              -> Strat {cons("ListCong")}

    SVar "^" Id "(" "(" {Strat ","}* ")" )? -> Strat {cons("CallModified")}

    %% Note: for each constructor there is a corresponding
    %% congruence operator with the following syntax:
    %%
    %% Identifier "(" {Strat ","}* ")" -> Strat {cons("Cong")}
    %%
    %% since this syntax overlaps with the syntax for strategy
    %% operator applications, congruences are defined implicitly
    %% by means of strategy definitions:
    %%
    %% strategies
    %%   C(s1,...,sn) = Cong(C)(s1,...,sn)
    %%
    %% where Cong(S) is a construct in the abstract syntax only.

```

Figure 15: Syntax of congruence traversal operators

GENERICALLY TRAVERSING TERMS

Congruences define traversals over known constructors. Stratego also provides several operators to define traversals generically.

PATH

If **s** is a strategy, then **n(s)** (with **n** a natural number) is the strategy that applies **s** to the *n*-th argument of the term. It fails if the term has less than *n* arguments, or if **s** fails on the *n*-th argument.

ALL

If **s** is a strategy, then **all(s)** is the strategy that applies **s** to each of the children of the root of the term. It succeeds if each of these applications succeeds and fails otherwise. In particular, **all(s)** succeeds on a term that has no arguments (a constant). For example, the following strategies apply a strategy **s** to each node in a term, in preorder (top-down), postorder (bottom-up) and a combination of pre- and postorder (downup):

```
bottomup(s) = rec x(all(x); s)
topdown(s)  = rec x(s; all(x))
downup(s)   = rec x(s; all(x); s)
```

A typical usage of these operators is the strategy

```
bottomup(try(s))
```

that tries to apply **s** at each node of a term.

ONE

The strategy **one(s)** applies **s** non-deterministically to one direct subterm. It fails if there is no subterm for which it succeeds. In particular, it fails for constants, since they have no child for which **s** can succeed. As we did with **all** we can construct bottom-up and top-down traversals with **one**:

```
oncebu(s) = rec x(one(x) <+ s)
oncetd(s) = rec x(s <+ one(x))
```

These strategies succeed if they find a single node in the term where the application of **s** is successful. The first one searches the term in bottom-up order: it first tries to find a successful application in one of the children **one(x)** and otherwise tries **s** at the root.

SOME

The strategy **some(s)** applies **s** to as many children as possible, but at least to one.

Examples of the use of **some** are:

Section: Strategies

```
somebu(s)      = rec x(some(x); try(s) <+ s)
sometd(s)      = rec x(s; all(try(x)) <+ some(x))
somedownup(s) = rec x(s; all(x); try(s) <+ some(x); try(s))
```

THREAD

threading environments through a traversal

```
module Stratego-Traversal
exports
  context-free syntax
    Natural  "(" Strat ")" -> Strat {cons("Path")}
    "one"    "(" Strat ")" -> Strat {cons("One")}
    "some"   "(" Strat ")" -> Strat {cons("Some")}
    "all"     "(" Strat ")" -> Strat {cons("All")}
    "thread" "(" Strat ")" -> Strat {cons("Thread")}
```

Figure 16: Syntax of generic traversal operators

CARRYING INFORMATION ALONG A TRAVERSAL

Environment distribution and threading congruences

Distributing congruences: for each constructor c there is a corresponding distributing congruence operator c^D defined according to the following scheme:

$$\frac{\text{Pair}(t_1, e) \text{ -- } s_1 \text{ --> } t_1' \quad \dots \quad \text{Pair}(t_n, e) \text{ -- } s_n \text{ --> } t_n'}{\text{Pair}(F(t_1, \dots, t_n), e) \text{ -- } F^D(s_1, \dots, s_n) \text{ --> } F(t_1', \dots, t_n')}$$

Threading congruences: for each constructor c there is a corresponding threading congruence operator c^T defined according to the following scheme:

$$\frac{\text{Pair}(t_1, e_1) \text{ -- } s_1 \text{ --> } \text{Pair}(t_1', e_2) \quad \dots \quad \text{Pair}(t_n, e_n) \text{ -- } s_n \text{ --> } \text{Pair}(t_n', e')}{\text{Pair}(F(t_1, \dots, t_n), e_1) \text{ -- } F^T(s_1, \dots, s_n) \text{ --> } \text{Pair}(F(t_1', \dots, t_n'), e')}$$

IMPLEMENTATION

These congruences can be generated at compile time by need, i.e., whenever there is a call to $D(F)$ or $T(F)$, the instantiation of the schemes above should be generated. This can be done during needed definition computation.

```

c^D(s1,...,sn) : Pair(c(x1,...,xn),env) -> c(y1,...,yn)
where <s1> Pair(x1,env) => y1;
    ...;
    <sn> Pair(xn, envn) => yn

c^T(s1,...,sn) : Pair(c(x1,...,xn),e1) -> Pair(c(y1,...,yn), e-last)
where <s1> Pair(x1,e1) => Pair(y1,e2);
    ...
    <sn> Pair(xn, en) => Pair(yn, e-last)

```

Section: Rules

DEFINING BASIC TRANSFORMATIONS WITH RULES

Standard rewrite rules are a notation for a special kind of strategies.

A rule section has the form

```
rules
  Rule*
```

A rule has the form

```
Label : Term -> Term
```

An example is the following rule that defines beta-reduction on lambda expressions.

```
rules
  Beta : Apply(Lambda(x, e1), e2) -> Subs(e2, x, e1)
```

denotes

```
strategies
  Beta = ?App(Abs(x, e1), e2); !Let(x, e2, e1)
```

A rewrite rule defines a one step transformation on terms. If the term matches the left-hand side of the rule it is replaced by the appropriately instantiated right-hand side of the rule. This transformation happens at the root of a term. Later on we will introduce ways to apply rules to the subterms of a term.

CONDITIONAL RULES

The simple rules that we introduced above apply if the left-hand side of the rule matches the root of the term. Sometimes we want to restrict the applicability of a rule to a subset of terms that match the pattern of the left-hand side. A conditional rule of the form

```
Label : Term -> Term where Strategy
```

applies if the left-hand side matches and in addition the strategy in the where clause succeeds when applied to the term.

For example, let `in` be the strategy that when applied to a pair `(t1, t2)` of terms, determines whether `t1` occurs as a subterm of `t2`, then the following conditional rule defines eta reduction of a lambda term:

```
Eta : Lambda(x, Apply(e, Var(x))) -> e
      where <not(in)> (Var(x), e)
```

Section: Rules

The notation

`<Strategy> Term`

is used to build an intermediate term to which the testing strategy should be applied.

If more than one condition has to hold, they can be composed using the strategy operators introduced in previous sections.

ANONYMOUS RULES

sometimes useful to define rule without a name

`\ t1 -> t2 where s \`

PARAMETERIZED AND OVERLOADED RULES

follows mapping to strategy definitions

```
module Stratego-Rules
imports Stratego-Strategies
exports
  %%sorts Rule RDef
  context-free syntax
    "rules" RDef*    -> BSpec {cons("Rules")}

    %% parameterized rule definition

    Identifier "(" {Identifier ","}* ")"?
      ":" Rule -> RDef {cons("RDef")}

    %% standard rewrite rule

    Term "->" Term ("where" Strat)? -> Rule {cons("Rule")}

    %% anonymous rule

    "\\\" Rule "\\\" -> Strat {cons("LRule")}
```

Figure 17: Syntax of rules

Section: Rules

APPLYING STRATEGIES IN TERMS AND MATCHING TERMS IN STRATEGIES

Strategies can be applied to (sub)terms of build patterns.

The notation

`<Strategy> Term`

is used to build an intermediate term to which the testing strategy should be applied.

If more than one condition has to hold, they can be composed using the strategy operators introduced in previous sections.

MATCHING CONDITIONS

Conditions can also be used to compute an intermediate result to be used in the right-hand side of the rule. A strategy of the form

`Strategy => Term`

applies a strategy to the current term and matches the result against the term. This can have the effect of binding the variables in the term, which can then be used in the right-hand side of the rule.

For example, the following rule defines the eta expansion of a lambda expression.

```
EtaExp: e -> Lambda(x, Apply(e, Var(x)))
      where new => x
```

The condition `new => x` generates a new string and matches the result against the variable `x`.

STRATEGY APPLICATIONS

Application of a strategy to a subterm in the right-hand side of a rule.

`<Strategy> Term`

For example, let `subs` be the strategy that when applied to a triple `(a, b, c)` replaces occurrences of `b` in `c` by `a`, then beta reduction can also be defined using the following rule:

```
Beta : Apply(Lambda(x, e1), e2) -> <subs> (e2, Var(x), e1)
```

Using matching conditions this rule is a shorthand for the rule

```
Beta : Apply(Lambda(x, e1), e2) -> e3
      where <subs> (e2, Var(x), e1) => e3
```

Section: Rules

```
module Stratego-Application
imports Stratego-Strategies
exports
  context-free syntax
    "<" Strat ">" Term    -> Term    {cons("App")}

    Strat "=>" Term      -> Strat    {cons("AM")}
    "<" StratAux ">" Term -> Strat    {cons("BA")}
    Strat                -> StratAux

  context-free priorities
    "<" StratAux ">" Term -> Strat >
    Strat "=>" Term -> Strat >
    Strat ";" Strat -> Strat
```

Figure 18: Syntax of strategy application

Section: Rules

STRATEGIC PATTERN MATCHING RULES

□

Section: Rules

```
module Stratego-Strategic-Rules
exports
  context-free syntax
    %% rule with strategic patterns

    Strat "-->" Strat ("where" Strat)?
                                -> SRule {cons("StratRule")}

    Identifier "(" (" {Identifier ","}* ")" )?
      "::" SRule -> RDef {cons("RDef")}
```

Figure 19:

DESCRIBING DEEPLY EMBEDDED PATTERNS WITH CONTEXTS

□

For some applications matching with a fixed pattern is not sufficient because some part of the pattern can be at an arbitrary depth from the root. In such circumstances contextual matching can be used.

For example, assume that our lambda terms satisfy the variable assumption, i.e. no two bound variables are the same if their binding positions are different, then we can define alpha renaming by means of the rule

```
Alpha : Lambda(x, e[Var(x)](sometd)) -> Lambda(y, e[Var(y)])
        where new => y
```

This rule replaces the abstracted variable **x** and all occurrences of **Var(x)** in the body of the abstraction by the fresh variable **Var(y)**. The strategy **sometd** specified with the context determines what traversal strategy to use to find occurrences of **Var(x)**. (Later on we will discuss how a bound variables can be renamed in order to adhere to the variable assumption.)

Section: Rules

```
module Stratego-Contexts
exports
  context-free syntax
  Identifier "[" Term "]" ("(" SVar ")")? -> Term {cons("Con")}
```

Figure 20:

Section: Rules

ABSTRACTING FROM PATTERNS WITH OVERLAYS

□

Section: Rules

```
module Stratego-Overlays
exports
  %%sorts Overlay
  context-free syntax
    Identifier ("(" {Identifier ","}* ")")?
      "=" Term -> Overlay {cons("Overlay")}
```

Figure 21: Syntax of overlays

Section: Rules

AS PATTERNS

As patterns of the form $x @ t$ can be used to deconstruct a term using pattern matching and at the same time bind the entire term to the variable x . For example, given the rule

```
rules
  R : F(x, y @ G(A, z)) -> H(x, y, z)
```

we have that

$$\langle R \rangle \ F(B, G(A, C)) \Rightarrow H(B, G(A, C), C)$$

Section: Connecting

CONNECTING TO THE WORLD THROUGH PRIMITIVES

□

Section: Connecting

```
module Stratego-Primitives.sdf
exports
  context-free syntax
  "prim" "(" String ")" -> Strat
```

Figure 22: Syntax of primitives

Section: Related Work

REFERENCES TO RELATED WORK

The application of Stratego is described in several publications.

BIBLIOGRAPHY

- [1] <http://www.cs.uu.nl/~visser/xt/>.
- [2] www.stratego-language.org.
- [3] Patricia Johann and Eelco Visser. Warm fusion in Stratego: A case study in the generation of program transformation systems. Technical report, Department of Computer Science, Universiteit Utrecht, Utrecht, The Netherlands, 1999.
- [4] Patricia Johann and Eelco Visser. Warm fusion in Stratego: A case study in the generation of program transformation systems. *Annals of Mathematics and Artificial Intelligence*, 2000. (Accepted for publication).
- [5] Bas Luttik and Eelco Visser. Specification of rewriting strategies. In M. P. A. Sellink, editor, *2nd International Workshop on the Theory and Practice of Algebraic Specifications (ASF+SDF'97)*, Electronic Workshops in Computing, Berlin, November 1997. Springer-Verlag.
- [6] Eelco Visser. Strategic pattern matching. In P. Narendran and M. Rusinowitch, editors, *Rewriting Techniques and Applications (RTA'99)*, volume 1631 of *Lecture Notes in Computer Science*, pages 30–44, Trento, Italy, July 1999. Springer-Verlag.
- [7] Eelco Visser. *The Stratego Compiler*. Department of Computer Science, Universiteit Utrecht, Utrecht, The Netherlands, 1999.
- [8] Eelco Visser. *The Stratego Library*. Department of Computer Science, Universiteit Utrecht, Utrecht, The Netherlands, 1999.
- [9] Eelco Visser. *The Stratego Reference Manual*. Department of Computer Science, Universiteit Utrecht, Utrecht, The Netherlands, 1999.
- [10] Eelco Visser. *The Stratego Tutorial*. Department of Computer Science, Universiteit Utrecht, Utrecht, The Netherlands, 1999.
- [11] Eelco Visser. Language independent traversals for program transformation. In *Workshop on Generic Programming*, 2000. (Accepted for publication).
- [12] Eelco Visser and Zine-el-Abidine Benaissa. A core language for rewriting. *Electronic Notes in Theoretical Computer Science*, 15, September 1998. In C. Kirchner and H. Kirchner, editors, Proceedings of the Second International Workshop on Rewriting Logic and its Applications (WRLA'98), Pont-à-Mousson, France.
- [13] Eelco Visser, Zine-el-Abidine Benaissa, and Andrew Tolmach. Building program optimizers with rewriting strategies. *ACM SIGPLAN Notices*, 34(1):13–26, January 1999. Proceedings of the International Conference on Functional Programming (ICFP'98).