TranSID: an SGML document manipulation language – Reference manual[1]

Jani Jaakkola
Pekka Kilpeläinen
Greger Lindén
Jyrki Niemi
Kimmo Paasiala

Department of Computer Science

University of Helsinki

Finland

June 3, 1999

Abstract

TranSID is a tree-based SGML transformation language, which can also be used for other SGML processing: for performing queries and for limited formatting. An evaluator of the TranSID language has been implemented, and tested to run in the Linux and Solaris environments. This report serves as a reference manual of the TranSID language. The report describes the syntax and informal semantics of the language and its built-in functions, as of version 0.038 of the evaluator.

[1] Automatically generated from SGML source by doc2tex.trs script.

Contents

1 Introduction

TranSID is a tree-based SGML transformation language, which can also be used for performing queries and for limited formatting. TranSID was designed and implemented in the research project *Structured and Intelligent Documents* (SID) at the Department of Computer Science of the University of Helsinki.

The TranSID language is a full-fledged SGML transformation language. The language is functional and largely declarative. The basic constructs of the language are queries, function definitions, and unnamed and named transformations. Queries and unnamed transformations are evaluated only once, when they are encountered, while functions and named transformations are evaluated only when called by name, which may be done multiple times. TranSID has also a comprehensive set of built-in functions, including functions for navigating the source SGML tree, and functions for manipulating lists and strings.

The TranSID system consists of an SGML parser, a TranSID parser, a TranSID evaluator and a linearizer. We use the SP parser as the SGML parser [Cla96]. A TranSID transformation or query process proceeds as follows. First, the TranSID parser parses the TranSID program specifying the transformations or queries to be applied to the source document. The parser constructs an internal representation of the TranSID program. Second, the SGML parser parses the source SGML document and builds an internal tree representation of the document, the *source tree*. Third, the evaluator builds a *target tree* by applying to the source tree the transformations or queries specified in the TranSID program. Finally, the linearizer linearizes the target tree, that is, it converts the tree into a linear form by traversing it in pre-order. The linearizer may perform some formatting of the output: it may output the target tree as SGML — with special characters replaced with entities where necessary — or for example as text with SGML tags stripped.

The TranSID process may be given more than one input document and it may also produce more than one output document. It is also possible to give to the process no input documents at all, in which case transformations are useless but queries may be used to produce output. A TranSID transformation or query program may refer to any part of the parse tree of the source SGML document. The semantics of the TranSID transformations are described in [JKL97].

TranSID may also be used as a client/server system using a protocol implemented on top of TCP/IP. In the client/server mode, the server is used to store a collection of documents and usually also user-defined TranSID functions for accessing and transforming the documents. A client may then pose queries to the server, which returns the linearized results to the client.

The rest of this report is organized as follows. Section 2 describes notational conventions used in this manual. Section 3 contains instructions for running the TranSID evaluator and also lists available command line options. Section 4 describes the lexical syntax of TranSID language. Section 5 describes data types of TranSID language. The result of linearization, i.e., the result output format of TranSID is described in Section 6. Section 7 describes operators of TranSID language. Section 8 describes expressions, the building blocks of the TranSID language. Section 9 describes different types of transformation programs. Section 10 describes context variables. Section 11 describes local variables. Section 12 describes orientation functions which are used to navigate in an SGML tree. Section 13 describes list manipulation functions. Section 14 describes list ordering functions. Section 15 describes list property functions. Section 16 describes list manipulation functions based on conditional expressions. Section 17 describes string manipulation functions. Section 18 describes string manipulation functions based on regular expressions. Section 19 describes functions for accessing the operating system.

2 Notations

The following list describes the notational conventions used in this manual:

`[optional]` for optionality.

.funcname or .funcname(param1, param2) for a built-in or a user-defined function.

... for any number of parameters to a function.

`A` $\rightarrow$ **`B`** for a production in Backus-Naur form.

`A` | **`B`** | **`C`** for multiple alternatives.

`B*` for zero or more occurrences of `B`.

`B?` for zero or one occurrences of `B`.

NTERM for a nonterminal.

`"string"` for a literal string.

`´c´` for a literal character.

( ) for grouping.

3 Running TranSID programs

The TranSID evaluator is invoked as follows:

```
transid [options] transformation-program [SGML files]
```

The options are

`-c` ***hostname*** Evaluate the given TranSID program in a TranSID server instead of evaluating it locally. In current version of TranSID, all given SGML documents are ignored when the `-c` option is used. Also all constants defined with the `-D` option are ignored.

`-q` Do not output the target document

`-l` This option is useful only when used in conjunction with the `-c` option. This option asks the TranSID client to output the server log to standard output during the evaluation of TranSID programs. Note that the server log and query output are not synchronized in client mode because of the way the TranSID protocol is implemented.

`-s` ***servermode*** Start server mode after parsing and evaluating the given documents and programs. In server mode TranSID is waiting for clients to connect and to request evaluation of queries or transformations. In server mode `foreground` TranSID stays on foreground and outputs the server log to standard output. In server mode `daemon` TranSID starts as a daemon after parsing and evaluating the programs and documents possibly given on the command line. In server mode it is possible to access operating system services using the functions described in Section 19. In server mode `secure` all access to the operating system is denied. In server mode `unsecure` it is possible to access the operating system, but the functions accessing the operating system must have been defined in the transformation program passed to the server at its startup.

`-L` ***debug level*** Select debugging level. Valid levels are 1–7 where level 7 produces vast amounts of debugging output and level 1 produces output only when TranSID panics.

`-L` ***debug section*** Debug a certain section of the TranSID evaluator. Section may be the name of a C source file or a section marked with C preprocessor macros.

`-D` ***name=(TranSID expression)*** This option defines the constant *name* to have the constant value specified by *expression*. In current version of TranSID, the expression may contain also non-constant values and still work; this is a bug and will be fixed some day.

4 Lexical syntax

This section describes the lexical syntax of TranSID language.

4.1 Whitespace

Space, horizontal tab and newline characters are whitespace characters in TranSID programs. Consecutive whitespace characters are regarded as one whitespace character unless they are part of a quoted string.

4.2 Comments

TranSID has C++-like comments. There are two forms of comments: sections starting with `/*` and ending in `*/`, and sections starting with `//` and ending in a newline. TranSID comments cannot be nested.

4.3 Quoted strings

A sequence of characters enclosed in double quotes forms a quoted string.

4.4 Integers

A sequence of one or more digits is interpreted as a 32-bit signed integer.

4.5 Identifiers

An identifier in TranSID language consists of a sequence of letters, digits and underscore characters. The first character must be a letter or an underscore character. All identifiers in the TranSID language are case-insensitive.

Note that in TranSID function definitions and calls the parenthesis preceding the parameter list must be placed right after the function name. Use `.first(2)` instead of `.first (2)`.

4.6 Reserved words

The following words are reserved and cannot be used as identifiers:

- `and`
- `becomes`
- `begin`
- `const`
- `current`
- `false`
- `function`
- `not`
- `null`
- `oneof`
- `or`
- `query`
- `set`
- `source`
- `strcmp`
- `these`
- `this`
- `thisnum`
- `transformation`
- `true`
- `var`
- `when`

5 Data types

TranSID has only one data type, *list*. A list consists of any number of *nodes*. A list of zero nodes is called an empty list or a null list.

A *node* is anything that can be located in an SGML tree, such as

- an element,
- an attribute,
- an entity,
- a PI (processing instruction), or
- content.

An element node consists of the name of the element, possible attributes and the possible content of the element. The name of the element is usually called a *generic identifier* (GI) in SGML terminology. We will use this term to refer to the name of an element later in this manual.

Content can be any of the following types:

- a string of virtually unlimited length,
- a 32-bit signed integer, or
- a Boolean value.

In TranSID, integers, strings and Booleans are only special cases of TranSID *lists* having only one *node* of type *content*.

5.1 Type conversions

TranSID implements lazy type conversion. When a TranSID operator or function requests a list of type *integer* as argument, an attempt is made to convert the argument list to an integer. If the requested type conversion cannot be performed (e.g. trying to convert the string "HELLO" to integer), a warning is given.

5.2 Converting list to string

Conversion from list to string uses the following rules:

- Consecutive strings are not concatenated together.
- Element nodes are converted to a string with their
  GI as the value of the string.
- Boolean nodes become either "TRUE" or "FALSE" according to their value.
- Integer nodes get trivially converted: `(1 2)` becomes `"12"`.

5.3 Converting list to integer

Conversion from list to integer uses the following rules:

- Empty lists and lists whose length is greater than one generate a warning and the type conversion fails.
- For element and attribute nodes, TranSID attempts to convert the name of the node to an integer.
- For *PI*, *entity* and *content* nodes, TranSID attempts to convert the content of the node to an integer.
- If the type conversion fails, a warning is generated (e.g., when trying to convert `("ONE")` to integer).

5.4 Converting list to Boolean

Conversion from list to Boolean uses the following rules:

- An empty list is considered `FALSE`.
- A Boolean node is `TRUE` or `FALSE` according to its content.
- All other lists are `TRUE`, including lists `("0")`, `("")` and `(0)`.

6 TranSID output

This section describes the result of TranSID linearization, i.e., the result output format. TranSID outputs the result of a transformation and the value of a query as a linearized string. Linearization is a straightforward process obeying the following rules:

- A line feed is appended to the end of the output. No other line breaks are added to the output.
- An empty element is output as a start tag of the form `<GI Attributes>`.
- A non-empty element is output in the form `<GI Attributes>Content</GI>`. Notice that the content may be empty even though the element wasn't declared empty.
- Each attribute node is preceded in the output by a space, and is represented as `ATTRNAME="Value"`.
- Boolean nodes are represented as strings `TRUE` and `FALSE`.

For example, consider the following query, which builds a small tree of three element nodes. The root of the tree has two attributes, and it has a string, a Boolean node and two element nodes as its children. The first of the subordinate elements is empty, while the other just has no content.

```
query <<"A" "False"=(2 <1) "k"=(1000+24 )>>
  { "This is true: " 1 < 2 <<"B">> <<"c">>{ } };
```

The result of the above query is:

```
<A False="FALSE" k="1024">This is true: TRUE<B><c></c></A>
```

7 Operators

All operators in the TranSID language can take any type of lists as their operands. Some operators (like equals) function in different ways depending on the type of their operands. However, all list and operator combinations are not meaningful. All operators are evaluated from left to right unless noted otherwise.

7.1 Precedence

All TranSID operators are in descending order of precedence:

- `()` (parenthesis) for grouping lists,
- `.` (a dot) followed by a function name,
- = for attribute assignment,
- `not` for negation,
- `*` for multiplication and `/` for integer division,
- + for addition and − for subtraction,
- == equals, != not equals, <,>,<= and >= for integer comparison,
- `strcmp` for string comparison,
- `and` and `or` for Boolean expressions, and
- *whitespace* for list concatenation.

7.2 List concatenation

Two lists are concatenated by writing them consecutively and inserting one or more whitespace characters between them. No string concatenation is performed however. Even though the lists `("one" "two")` and `("onetwo")` look identical when linearized, they are not equal: `("one" "two").count==2` but `("onetwo").count==1`.

7.3 Equals (==) and not equals (!=)

The equals and not equals operators function differently depending on the type of their operand lists.

- Two empty lists are considered equal.
- When only one list is empty, the operand lists are considered not equal.
- When the size of one or both of the lists exceeds one, the operands are considered not equal.
- When both operands are elements, attributes or trees, they are considered equal only if they are represented by the same physical object.
- Comparing two nodes with different types (e.g. testing for equality of an attribute and an element) generates a warning and the nodes are considered not equal.

- Comparison of two Boolean nodes works as expected: `true == true` and `true != false`.
- If one of the nodes is of type *integer*, an integer comparison is attempted. If the other node cannot be converted to an integer, the nodes are considered not equal.
- Finally, an exact string comparison is attempted. (Whitespace is considered as part of a string: `" a "!="a"`).

7.4 Boolean operators

Boolean operators `and`, `or` and `not` convert their operands to Boolean (See Section 5.1) and return the result as a list which consists of one Boolean node. The evaluation of binary Boolean operators is optimized by using so-called short-circuit evaluation: if the result of the expression is known without evaluating the right-hand side of the expression, the right-hand side is not evaluated. Thus it is not safe to call functions that cause side effects in Boolean expressions.

7.5 Arithmetic operators

Arithmetic operators (`*`, `/`, `+` and `-`) are evaluated exactly like their C-language counterparts and they follow the laws of integer arithmetics, i.e., 5 divided by 2 results in 2. If one or both of the operands could not be converted to integer, an empty list is returned.

7.6 Integer comparison operators

Integer comparison operators compare two integers and return a Boolean node. If one or both of the operands could not be converted to an integer, an empty list is returned. One should note that an empty list is considered as `FALSE`, so one has to be careful when using integer comparison operators on lists which possibly cannot be converted to integers.

7.7 String comparison operators

Two strings can be compared for equality (`==`), inequality (`!=`) and lexicographical order (`strcmp`). The `strcmp` operator works similarly to the corresponding function in the C language (Section 17.2).

7.8 Attribute assignment

An attribute assignment returns an *attribute* node with the left-hand operand as the attribute name and the right-hand operand as the attribute value. Both operands are converted to strings. One should note that since the attribute assignment has a very high precedence, one has to use expressions like `"ATT"=(8+3)` instead of `"ATT"=8+3`.

8 Expressions

A TranSID expression is a piece of program that returns a list. An expression has a type in the sense that the expression returns a list which contains nodes of a certain type, e.g., an arithmetic expression returns a list of integers nodes. TranSID expressions are usually built from constant lists, context variables, variable references or function calls, which act as operands for the various operators presented in Section 7. Some examples:

```
1 2 3               // a constant list of three integers.
(1 2 3).count       // the same list acting as input list to a function.
1 + 2               // arithmetic expression.
a and b             // Boolean expression.
current.a.b.c("1") // a context variable followed by three function
                    // calls. The function called c takes one
                    // parameter, which in this case is a list of one
                    // string node.
```

A typical TranSID expression is a dot-separated sequence of function calls, where each function takes as its input list the result of the expression immediately preceding it. For an example

```
current.children.first(3)
```

is a TranSID expression which can be used in a transformation rule (Section 9.5). First, `current` (Section 10.1) evaluates to the element being transformed. Then the function `.children` gets this node as its singleton input list and returns the list of its child nodes. Finally, the function call `.first(3)` gets this list of nodes as its input list and returns the first three nodes of it, i.e. the first three children of the current element.

8.1 Constant lists

Currently, TranSID supports constants of the following types:

- integers,
- strings,
- Booleans and
- empty lists (null).

Integer nodes are created by inserting an integer (Section 4.4) into a TranSID expression. A list consisting of the integer value 3 is constructed like this: `(3)`.

String nodes can be created by inserting a quoted string (Section 4.3) into a TranSID expression. A list consisting of the string 'Hello World!' is constructed like this: `("Hello World!")`. Various backslash escapes can be used inside strings. Unknown backslash escapes generate a warning during program parsing. Currently implemented backslash escapes are:

- `\n` for newline character,
- `\r` for carriage return,
- `\t` for horizontal tab,
- `\"` for double quote, and
- `\\` for backslash.

An empty list can be created with the reserved word `null`. If the TranSID expression is used as a conditional expression, then an empty list means `FALSE`.

Here is a simple example which produces some CDATA content:

```
((1 ". Hello world!" "\n") (2 ". Once more "  "hello world!\n"))
```

The above expression yields a list of six elements.

When the list is linearized, it produces the following output:

```
1. Hello world!
2. Once more hello world!
```

8.2 Predefined constant lists

The following predefined constant lists can be used in TranSID expressions:

- `false` for the Boolean value `FALSE`.
- `null` for an empty list.
- `true` for the Boolean value `TRUE`.

8.3 Arithmetic expressions

Arithmetic expressions in TranSID language are evaluated using the standard infix notation. Operands for arithmetic operators (Section 7.5) are converted to integers (Section 5.3) before evaluation.

8.4 Creating tree structures

TranSID expressions may be used to construct arbitrary SGML tree structures. Tree structures are created by creating element nodes. When an element node is created, its *GI* and content must be specified. The content of an element may also be empty. Also attributes may be specified for the element.

An element node is created with the following syntax:

```
<< GI Attributes >> { Content }
<< GI Attributes >> // creation of an EMPTY element
```

GI is the GI of the element to be created. It is a TranSID expression, which is converted to string.

Attributes is a whitespace separated list of TranSID expressions specifying the attributes of the element.

Content is a TranSID expression specifying the content of the new element.

Creating an EMPTY element is different from creating an element with no content. An EMPTY element is created by omitting the curly braces from the element definition.

Attributes are specified with TranSID expressions which evaluate to *attribute* nodes. These expressions include attribute assignments and fetching attributes from existing nodes. An attribute specification has the following syntax:

Attribute name = *Attribute value*

where *Attribute name* and *Attribute value* are both TranSID expressions interpreted as strings.

The *orientation function* `.attributes` (Section 12.3) evaluates to all the attributes of all the elements in its input list. The orientation function `.attribute(attribute name)` evaluates to all the attributes having the name `attribute name` of the elements in its input list.

8.5 Examples of TranSID expressions

The expression below creates an element node with the GI `A`, with the attribute `HREF` having the value `"http://www.cs.helsinki.fi/research/rati/transid/"` and with the string content "TranSID home page"

```
<<"A" "HREF"="http://www.cs.helsinki.fi/research/rati/transid/">> {
"TranSID home page"
}
```

The following expression creates an element with the GI `CURRENT` and having the same attributes and the same content as the element for which this rule is being evaluated. The *context variable* `current` (Section 10.1) is used to refer to the node being processed.

```
<<"CURRENT" current.attributes>> {
current.children
}
```

Change the content of the element for which this rule is being evaluated to "New content", but preserve the GI and attributes:

```
<< current.name current.attributes>> {
"New content"
}
```

Another way of doing the same as above:

```
current {
"New content"
}
```

9 TranSID programs

A TranSID program is a sequence of any of the following constructs: constant definitions, global variable definitions, queries, function definitions and transformation specifications. Transformation specifications may define either named or unnamed transformations. The TranSID evaluator executes queries and unnamed transformations immediately. Calls to functions and named transformations may be used in TranSID expressions.

Constant definitions, global variable definitions and queries must be terminated with a semicolon (`;`). Function definitions and transformation specifications are enclosed in `begin-end` groups and they may not be followed by a semicolon.

9.1 Constant definitions

A constant definition defines a constant that can be used later in the program. The definition consists of the keyword `const` followed by the name of the constant, an assignment operator (=) and an expression. The value of the constant cannot be changed later in the program. An example:

```
const TSID = "TranSID";
```

9.2 Global variable definitions

A global variable definition defines a variable whose value can be used and changed later in the program. A defined variable is visible to all parts of the program that follow the variable definition. The definition consists of the keyword `var` followed by the name of the variable, an assignment operator (=) and an expression. An example:

```
var A = 3;
```

9.3 Queries

Queries are a simple and efficient way of doing small transformations or specifying small queries to SGML trees. Only the parts of the source tree which are referred to in the query expression are processed, which makes query evaluation fast. A query has the following syntax:

```
query expression;
```

The expression is evaluated and the result list is linearized to the standard output.

9.4 Functions

A TranSID function is a named section of code which performs some operation on its *input list* and returns a new list as its result. A function can be thought as a named query which can be reused when needed. A function definition consists of the keyword `function`, the name of the function starting with a single dot (`.`), an optional parameter list enclosed in parentheses and a function body enclosed in a `begin-end` group. Parameters of a function are separated with commas. Inside the function body, the *input list* is referred to via the context-variable `these`. For example:

```
// return the sum of a and b
function .add(a,b) begin
   a + b
end
```

```
// return the number of content nodes in the input list
function .countcontent begin
   these.data.count
end
```

9.5 Transformation rules

A transformation is specified by a sequence of *transformation rules*. A *transformation rule* consists of a *source clause* and a *target clause*.

All TranSID transformation rules start with a *source clause*. The source clause locates a certain substructure in the source SGML tree. The source clause consists of a node type and a string containing the name of the node to which the rule is to be applied. An asterisk `(*)` in place of the node name in the source clause means that the rule is to be applied to all nodes of the specified type. The reserved word `source` in place of the node name means that the rule is to be applied to the root of the SGML source tree. The possible node types (Section 5) are:

attribute for attribute nodes,

data for content nodes,

element for element nodes,

entity for entity nodes,

node for any type of node, and

pi for PI nodes.

In addition to the source clause, a *context condition* may be defined. The context condition can be used to describe more specifically the context where the *target clause* is to be applied. The context condition is a TranSID expression which is interpreted as a Boolean value. A context condition begins with the reserved word `when` and is placed immediately following the source clause.

The *target clause* is a TranSID expression which evaluates to a target list. The target clause replaces the located substructure (SGML tree node) by the target list. The target clause is placed at the end of the rule; it begins with the reserved word `becomes` and ends with a semicolon.

The reserved word `current` can be used in the transformation rules to point to the node which is currently being processed. The keyword `current` can be used in both the context condition and the target clause to access the node currently being processed. The relatives of `current` are defined in a special way:

- The parent and the sibling nodes of the `current` node are nodes in the source tree.
- The children of `current` are nodes in the target tree, i.e., they are results of previously applied transformation rules.
- The original node in the source tree from which the current node was created can be accessed with `current.origin`.

The above rules reflect the bottom-up order which TranSID applies in transformation: the children of a node are transformed according to the transformation specification before their parent. The original descendants of the current node can be accessed via `current.origin`, which is a node in the source tree.

The order in which the transformation rules are declared is significant. If two different rules match the same node, then the rule declared earlier takes precedence. The declaration order of the rules is also taken into account when the result of a transformation rule matches the source clause of another rule in the transformation. If the rule matching the result is declared later than the rule producing the result, then the rule will be applied, otherwise not. Enclosing a sequence of transformation rules in a `oneof` - `begin` - `end` group disables this behavior, i.e. only the first matching rule is applied to the current node. An example:

```
transformation begin
  oneof begin
  !element "A" becomes <<"B">>{current.children};
  !element "B" becomes <<"C">>{current.children};
  end // oneof
end // transformation
```

The transformation above transforms an element "A" into element "B", but doesn't transform the resulting "B" element into element "C" because of the `oneof - begin` - `end` group.

9.6 Named transformations

A named transformation is a named sequence of transformation rules that are defined in the same way as the rules of an unnamed transformation. Like functions, named transformations can have parameters. A named transformation is analogous to a function in the sense that it is also a named code section performing some action on its input list. Context variable `source` is used to refer to the nodes in the input list one at a time. An example:

```
transformation .test(a,b) begin
!element "A" becomes
  << a >>{current.children};
!element "B" becomes
  << b >>{current.children};
end
```

The above transformation renames the elements "A" and "B" using the names given in parameters `a` and `b`.

9.7 Transformation specifications

The syntax of transformation specifications is as follows:

```
TransformationProgram ->   "TRANSFORMATION" "BEGIN" (RuleSet)* "END"
```

```
RuleSet ->   OneofRuleGroup
           | RuleGroup


OneofRuleGroup ->   "ONEOF" "BEGIN" (Rule)* "END"


RuleGroup ->   (Rule)*


Rule ->   SourceClause (ContextCondition)? "BECOMES" TargetClause ';'


SourceClause ->   '!' ElementType ElementName


ElementType ->   "ATTRIBUTE"
               | "DATA"
               | "ELEMENT"
               | "ENTITY"
               | "NODE"
               | "PI"


ElementName ->   QuotedString
               | '*'
               | "SOURCE"


ContextCondition ->   "WHEN" Expression


TargetClause ->   Expression
```

9.8 Examples of transformation specifications

Remove all elements with the GI `ssect`:

```
TRANSFORMATION BEGIN   // Start transformation specification
!ELEMENT "ssect"       // source clause
BECOMES null;          // target clause
END                    // End transformation specification
```

Remove all `ssect` subtrees which have a parent named `section`:

```
TRANSFORMATION BEGIN
!ELEMENT "ssect"
WHEN current.parent.name=="SECTION"
BECOMES null;
END
```

The default transformation which copies the source tree to the target tree:

```
TRANSFORMATION BEGIN
!NODE *                  // source clause which matches all possible nodes
BECOMES current;
END
```

Convert a definition list into LaTeX. This example is a small fragment of a TranSID program that transforms the TranSID documentation written in SGML into LaTeX.

```
TRANSFORMATION BEGIN
!ELEMENT "DLIST" BECOMES            // A "DLIST" element is transformed into
"\\begin{description}\n"            // \begin{description}
current.children                    //   contents of "DLIST" element
"\\end{description}\n";             // \end{description} LaTeX construction.

!ELEMENT "D"  BECOMES               // A "D" element is transformed into
"\\item[" current.children "]";     // \item[ contents of "D" ] LaTeX command.

!ELEMENT "LI"                       // Matches an "LI" element only if it has
WHEN current.parent.name=="DLIST"   // a "DLIST" element as its parent.
BECOMES current.children;           // An "LI" element is replaced by the
END                                 // character data it contains.
```

10 Context variables

Context variables are predefined lists which can be used in TranSID expressions. They are also a way to access the node that is currently being transformed and the context of conditional expressions.

10.1 current

The context variable `current` returns the node which is currently being transformed. The current node is a copy of the original node in the source tree. The relatives of the current node are defined as follows:

- The parent, left siblings and right siblings are in the source tree.
- The orientation function `.origin` points to the node in the source tree of which the current node is a copy.
- The children nodes of the current node are the results of the transformation rules already applied to the original children of the current node.

10.2 source

The context variable `source` returns the root node of the source tree. In a named transformation, `source` refers at a time to the root node of each subtree in the input list.

10.3 these

In conditional expressions in which a TranSID expression is applied to a list of nodes, `these` refers to that list. `These` is also used in functions (Section 9.4) to refer to the input list.

10.4 this

In conditional expressions in which a TranSID expression is applied to every node in a list, `this` refers to the node to which the expression is currently being applied.

10.5 thisnum

Whenever `this` is defined, also `thisnum` is defined. `thisnum` returns the number of the node `this` in the original list. For example:

```
query
("ONE" "TWO" "THREE" "FOUR").map(true,
thisnum "=" this " ");
```

The result of the above query is:

```
1=ONE 2=TWO 3=THREE 4=FOUR
```

11 Local variables

TranSID offers local variables for saving the results of TranSID expressions temporarily. This is useful when the result of a complex expression needs to be used many times. Local variables are local to a query or a rule. The value of each variable is a list. There can be an arbitrary number of variables, which may contain arbitrary lists. Note that variables contain references to SGML tree nodes instead of copies of nodes.

11.1 .set(varname)

Variables are assigned values with the function `.set`. This function assigns its input list to the given variable. The `.set` function returns its input list unchanged; this it makes possible to set a variable in the middle of an expression and to use the value of the variable immediately. If the same variable is assigned a new value, the previous value is discarded. It is also possible to assign values to variables by using regular expression string functions (see Section 18).

After a local variable is set, its name can be used in a TranSID expression. Using a variable which has not been assigned a value generates a warning and returns an empty list.

11.2 Examples of using variables

Produce a list which contains the word "TranSID" eight times:

```
query
        (("TranSID".set(v) v).set(v) v).set(v) v;
```

The result of the above query is:

```
TranSIDTranSIDTranSIDTranSIDTranSIDTranSIDTranSIDTranSID
```

12 Orientation functions

An SGML tree may be traversed using the TranSID orientation functions. Orientation functions get their input list from the left-hand side of the expression which they are part of, apply the orientation to the list and return a new list.

12.1 .ancestors

The orientation function `.ancestors` returns for each node in its input list a list of ancestors, which consists of the nodes on the path from the parent of the node to the root of the SGML tree. In the current version of TranSID, `.ancestors` gives a warning if the size of its input list exceeds one. Duplicate ancestors are not removed from the result, so the next query returns a list containing the root of input tree possibly several times, once for each of its children.

```
query source.children.ancestors;
```

12.2 .attribute(name)

The orientation function `.attribute(name)` returns all attribute nodes of all the elements in its input list such that the name of the attribute node is `name`. The value of an attribute node is obtained by function `.value` (Section 15.5). Since `.attribute` returns the *attribute* node instead of the attribute value, the following transformation rule makes sense:

```
!ELEMENT "TITLE" BECOMES
<<"A" current.attribute("HREF")>> {current.children};
```

12.3 .attributes

The orientation function `.attributes` returns all *attribute* nodes of all the *element* nodes in its input list. The value of an attribute node is obtained by function `.value` (Section 15.5). Since `.attributes` returns *attribute* nodes instead of *value* nodes, the following transformation makes sense:

```
!ELEMENT "REF" BECOMES
<<"REFERENCE" current.attributes>> {current.children};
```

12.4 .children

The orientation function `.children` returns all child nodes of all the elements in its input list. Note that only elements can have children. The value of an attribute is not considered a child of the attribute. Attributes are not considered children of element nodes. They can be obtained using orientation function `.attributes` (Section 12.3).

12.5 .descendants

The orientation function `.descendants` returns all descendants of all the nodes in its input list in pre-order. Note that the result may be bigger than the source document. An expression like `source.descendants.ancestors.descendants` probably hangs or aborts the TranSID evaluator.

12.6 .left

The orientation function `.left` returns the left sibling nodes of the nodes in its input list. Only a node which is a child of an element can have left nodes. The expression `current.left` refers to the nodes in the source tree. The left siblings are returned in order from left to right, i.e., the preceding sibling of `current` is `current.left.last`.

12.7 .noderef

The orientation function `.noderef` returns a list of nodes, whose *nodeid* identifiers (Section 15.3) are given in the input list.

12.8 .origin

The orientation function `.origin` returns the nodes from which the nodes of the input list originated. Currently these are always nodes in the source tree. If you want to access the original children of the current node instead of the children generated by the transformation, you should use `current.origin.children`.

12.9 .parent

The orientation function `.parent` returns the parent nodes of the nodes in its input list. The parent node of the current node is a node in the source tree.

12.10 .right

The orientation function `.right` returns the right sibling nodes of the nodes in its input list. Only a node which is a child of an element can have right nodes. The expression `current.right` refers to nodes in the source tree. The right siblings are returned in order from left to right, i.e., the next sibling of `current` is `current.right.first`.

13 List functions

List functions are used to select sublists from the input list.

13.1 .data

The list function `.data` returns the *content* nodes of its input list. *String*, *integer* and *Boolean* nodes are considered *content* nodes. The content nodes which are immediately subordinate to the *current* node can be obtained by expression `current.children.data`. The whole data content of the *current* node can be obtained by expression`current.descendants.data`.

13.2 .elements

The list function `.elements` returns all *element* nodes in its input list. The element nodes which are immediately subordinate to the *current* node can be obtained by `current.children.elements`.

13.3 .first and .first(number)

The list function `.first` returns the first node in its input list. The function `.first(number)` returns the first `number` nodes in its input list.

13.4 .last and .last(number)

The list function `.last` returns the last node in its input list. The list function `.last(number)` returns the last `number` nodes in its input list.

13.5 .null

The `.null` list function discards whatever its input list contains and returns an empty list.

13.6 .sublist(m,n)

The `.sublist` function takes two integer parameters. The semantics of the sublist boundaries follows the HyTime dimension specifications [ISO92]. The numbering of nodes in a list starts from one. Below, we denote a negative number by *-n*.

sublist(m,n) Select *n* nodes starting from node *m* from the beginning of the list.

sublist(-m,-n) Select *m* nodes starting from node *m+n-1* from the end of the list.

sublist(m,-n) Select middle nodes starting from node *m* from the beginning of the list and ending with the node *n* from the end of the list

sublist(-m,n) Select *n* nodes starting from node *m* from the end of the list.

For example, assume that we have that `list = (1, 2, 3, 4, 5, 6, 7, 8)`. Then we have

- list.sublist(2,2) = (2, 3)
- list.sublist(-2,-2) = (6, 7)
- list.sublist(2,-2) = (2, 3, 4, 5, 6, 7)
- list.sublist(-2,2) = (7, 8)

14 Ordering functions

Ordering functions are used to reorder the nodes in the input list.

14.1 .lexsort(key expression)

The function `.lexsort` returns its input list sorted in ascending lexicographical order. The sorting is based on the value of the key expression. The key expression is a TranSID expression that is applied to every node in the input list. The context variable `this` refers to the nodes in the input list one at a time. The key expression must be a string expression or at least convertible to string. The following example performs a case–insensitive sort on a list of three string nodes.

```
query ("bA" "aa" "Ab").lexsort(this.toupper);
```

The result of the above query is:

```
aaAbbA
```

14.2 .numsort(key expression)

The function `.numsort` returns its input list sorted in ascending numerical order. The sorting is based on the value of the key expression. The key expression must be an integer expression or at least convertible to integer.

14.3 .reverse

The `.reverse` ordering function returns its input list in reverse order.

15 List property functions

Each of these functions returns a property of its input list. Typically the return value is either a string or an integer.

15.1 .count

The `.count` function returns the number of nodes in its input list.

15.2 .name

For an *element* node the `.name` function returns the *GI* of the element. For an *attribute* node it returns the name of the attribute. For a *content* node it returns the content of the node.

15.3 .nodeid

The `.nodeid` function returns a unique identifier for each node in its input list. These identifiers can be used as references and they can dereferenced using the `.noderef` orientation function (Section 12.7). In the current version of TranSID, these identifiers are of type string.

15.4 .siblingnum

For nodes which have an element node as a parent, the `.siblingnum` function returns the number of the element among its siblings. The function returns `null` for nodes which do not have a parent node. If the input list contains several nodes which had a valid sibling number, the function returns a list containing all the numbers.

15.5 .value

The `.value` function returns the values of all the attribute nodes in its input list. Currently this is the only way to access the value of an attribute.

16 Conditional functions

Conditional functions are used to manipulate lists based on conditional expressions. They can be used to filter certain nodes from lists or to map list nodes to different kind of nodes. Two context variables, `these` and `this`, are visible inside conditional functions. The context variable `these` is usually used to refer to the whole input list of the function. In some conditional functions it is also used to refer to a sublist of the input list that satisfies some condition. The context variable `this` is used to refer to each node in the input list, one at a time.

16.1 .case(condition expression -> action expression, ...)

The parameters for the conditional function `.case` consist of `condition -> action` rules. The left-hand side of a rule is a condition expression and the right-hand side is a corresponding action expression. Inside the `.case` function the context variable `these` refers to the input list of the function. All the condition expressions are evaluated in the given order until a condition evaluates to `TRUE`. The result of this function is then determined by the corresponding action expression. If none of the condition expressions evaluates to `TRUE`, the result is the input list unmodified. An example:

```
.case(
  these.name=="PARA" -> these.handlepara,        // Handle paragraph elements.
```

```
  these.name=="SECTION" -> these.handlesection, // Handle section elements.
  true -> these.handlerest                      // Default rule.
)
```

16.2 .glue(start condition, end condition, action expression, [else action expression])

The function `.glue` maps sublists of the input list to another list. `.glue` is one of the most powerful of the conditional functions.

The conditional function `.glue` scans its input list and locates all sublists in the input list which start with a node satisfying the start condition and continue to a node satisfying the end condition. The node satisfying the end condition is not part of the located sublist. The end of the input list is considered as a satisfied end condition. When such a sublist is located, the action expression is evaluated with the context variable `these` set to the located sublist and `this` set to the node that satisfied the end condition (null if the satisfying condition was the end of list). The located sublist is then replaced with the result of the action expression. If the optional else action expression is present, then all consecutive nodes that are not part of any sublist satisfying the start and end conditions are grouped into a sublist. This sublist is then replaced with the result of the else action expression.

For example: find sublists which start with "[" and end with "]".

```
query
// Create a simple demonstration list
("1" " [ " "2" " ] " "3" "4" " [ " "5" " ][ " "6")
.glue(
this==" [ " or this == " ][ ",         // start condition
this==" ] " or this == " ][ ",         // end condition
" <GROUP> " these " </GROUP> "  // action expression
);
```

The result of the above query is:

```
1 <GROUP>  [ 2 </GROUP>  ] 34 <GROUP>  [ 5 </GROUP>  <GROUP>  ][ 6 </GROUP>
```

16.3 .group(condition expression, action expression [,else action expression])

The conditional function `.group` is used to group consecutive nodes of a list into sublists. The `.group` function operates as follows: First the condition expression is evaluated with the context variable `this` pointing to the first node of the input list and with the context variable `these` pointing to an empty list. If the condition expression evaluates to `TRUE`, the node is added to the list in the variable `these`. All the remaining nodes in the input list are tested with the condition expression and as long as the expression evaluates to `TRUE`, the nodes are added to the list in the variable `these`. Once the condition expression evaluates to `FALSE`, the action expression is evaluated with the context variable `these` pointing to the list collected earlier. The context variable `this` points to the first node that did not satisfy the condition expression (`null` if the end of the input list was encountered). Once the condition expression has evaluated to `FALSE`, all consecutive nodes in the input list that do not satisfy the condition expression are collected into a list. The else action expression is evaluated with the context variable `these` set to this list once the condition expression evaluates into `TRUE`.

Example: Group a list to sublists of at most three nodes:

```
var list = 1 1 1 2 2 2 3 3 3;
query list.group(these.count<3,<<"THREE">> {these},"NEVER");
```

The result of the above query is:

```
<THREE>111</THREE><THREE>222</THREE><THREE>333</THREE>
```

Example: Group a list of integers to sublists of even and odd numbers:

```
var list = 1 1 2 2 3 3;
query list.group((this/2)*2==this,<<"EVEN">> {these},<<"ODD">>  {these});
```

The result of the above query is:

```
<ODD>11</ODD><EVEN>22</EVEN><ODD>33</ODD>
```

16.4 .having(condition expression)

The conditional function `.having` applies the condition expression to every node in its input list and returns only the nodes for which the condition is `TRUE`. In the condition expression one may use `this` and `thisnum` to refer to the node to which the condition expression is currently being applied.

Example: Select the first three title nodes from the input list:

```
.having(this.name=="TITLE").having(thisnum<=3)
```

16.5 .if(condition expression, then action expression [,else action expression])

The conditional function `.if` applies the condition expression to the whole input list. If the condition expression evaluates to `true`, the result of the then action expression is the result of the function. Otherwise the result is the value of the optional else action expression. If the else part is missing, the result is the input list unchanged. An example:

```
.if(these.count > 1, <<"MANY">>{these}, <<"ONE">>{these})
```

16.6 .map(condition expression, true action expression [,false action expression])

The conditional function `.map` applies the condition expression to every node in its input list and replaces the nodes for which the condition is `TRUE` with the result of the true action expression. If present, the optional false action is applied to all the nodes not satisfying the condition expression.

Example: Change each child of a `BODY` element into an `A` element if the child has an attribute `IDREF`.

```
TRANSFORMATION BEGIN
!element "BODY" becomes
current {
current.children.map(
this.attribute("IDREF"),
<<"A" this.attributes>> {this.children}
)
};
END
```

17 String functions

String functions are used to manipulate strings. All these functions form their *input string* by performing an implicit `.strcat` operation on their input list.

17.1 .strcat

The `.strcat` string function concatenates all string nodes in its input list into a single string node.

```
query ("one" "two").strcat;
```

The result of the above query is:

```
onetwo
```

17.2 strcmp

The string comparison operator `strcmp` compares two strings and returns a signed integer value depending on the lexicographical order of the operand strings. If the first string is "less" than the second string, a negative value is returned. A positive value is returned if the first string is "greater" than the second string. A zero is returned if the strings are identical. For example: `query ("foo" strcmp "bar");` results in: `4` `query ("bar" strcmp "foo");` results in: `-4` and `query ("bar" strcmp "bar");` results in: `0`.

17.3 .tolower

The `.tolower` string function converts its input string to lower case.

17.4 .toupper

The `.toupper` string function converts its input string to upper case.

17.5 .find(string)

The string function `.find` takes one parameter, the pattern string which is searched for in the input string. The result is a list of integer nodes, each indicating the starting position of a matching substring. An empty list is returned if the pattern does not occur in the input string. More powerful string searching is possible with regular expression functions (Section 18).

17.6 .substr(m,n)

The `.substr` string function takes integer parameters. The semantics of the substring boundaries follows the HyTime dimension specifications [ISO92]. The numbering of characters in a string starts from one. Below, we denote a negative number by *-n*.

substr(m,n) Select *n* characters starting from character *m* from the beginning of the string.

substr(-m,-n) Select *m* characters starting from character *m+n-1* from the end of the string.

substr(m,-n) Select middle characters starting from character *m* from the beginning of the string and ending with the character *n* from the end of the string

substr(-m,n) Select *n* characters starting from character *m* from the end of the string.

For example:

```
query "abcdefgh".substr(2,2);
```

The result of the above query is:

```
bc
```

```
query "abcdefgh".substr(-2,-2);
```

The result of the above query is:

```
fg
```

```
query "abcdefgh".substr(2,-2);
```

The result of the above query is:

```
bcdefg
```

```
query "abcdefgh".substr(-2,2);
```

The result of the above query is:

```
gh
```

17.7 .strlen

The `.strlen` string function returns the length of its input string.

17.8 .trim(string)

The `.trim` string function returns the input string with the characters occurring in the parameter *string* removed from its beginning and end. The removing of the characters ends when a character not present in the parameter *string* is encountered.

```
query ("aa textb b").trim(" ba");
```

The result of the above query is:

```
text
```

17.9 .trimleft(string)

The `.trimleft` string function is similar to `.trim`, but characters are removed only from the beginning of the input string.

17.10 .trimright(string)

The `.trimright` string function is similar to `.trim`, but characters are removed only from the end of the input string.

17.11 .isXXX string functions

These functions are used to test if the input string contains certain types of characters.

17.11.1 .islower

The `.islower` function returns `TRUE` if its input string contains only lower-case characters, and `FALSE` otherwise.

17.11.2 .isupper

The `.isupper` function returns `TRUE` if its input string contains only upper-case characters, and `FALSE` otherwise.

17.11.3 .isdigit

The `.isdigit` function returns `TRUE` if its input string contains only digits, and `FALSE` otherwise.

17.11.4 .isalpha

The `.isalpha` function returns `TRUE` if its input string contains only alphabetic characters, and `FALSE` otherwise.

17.11.5 .isalnum

The `.isalnum` function returns `TRUE` if its input string contains only alpha-numeric characters, and `FALSE` otherwise.

17.11.6 .isspace

The `.isspace` function returns `TRUE` if its input string contains only whitespace characters, and `FALSE` otherwise.

17.12 Path name string functions

The following three functions are used to manipulate file system path names. The result of these operations depends on the type of the operating system used. Currently the only supported path name syntax is the one used in the UNIX operating system. A warning is issued and an empty list is returned if the input string cannot be interpreted as a path name.

17.12.1 .filedir

The `.filedir` function returns the directory path of its input string interpreted as a path name.

17.12.2 .filename

The `.filename` function returns the file name of its input string interpreted as a path name.

17.12.3 .filesuffix

The `.filesuffix` function returns the suffix part of the filename in its input string interpreted as a path name.

17.13 URL string functions

The following functions assume that the input string can be interpreted as a URL. A warning is issued if the string is not a valid URL and an empty list is returned.

17.13.1 .urlprotocol

The `.urlprotocol` function returns the protocol name of the URL.

17.13.2 .urlserver

The `.urlserver` function returns the server name of the URL.

17.13.3 .urldir

The `.urldir` function returns the directory path of the URL.

17.13.4 .urlanchor

The `.urlanchor` function returns the anchor of the URL.

18 Regular expression functions

Regular expression functions are used for searching patterns in the input string and also for complex string operations like substitutions.

18.1 Syntax of TranSID regular expressions

The syntax of TranSID regular expressions is very similar to the one used in the UNIX program egrep [Bel82]. One notable difference between TranSID and egrep regular expressions is the escape character, which is a backslash (`'\'`) in egrep and a percent sign (`'%'`) in TranSID regular expressions. The syntax of TranSID regular expressions is described in the following.

A single character All characters match themselves except characters `.*+?|()^$[]%` which have a special meaning in TranSID regular expressions. The character `'%'` is the escape character in TranSID regular expressions. To match a plain `'%'` character one has to write `"%%"`. The same escape convention applies to the other special characters. A percent sign followed by anything else but an alphabetic character or an underscore character (`'_'`) is taken as single character, i.e. `"%="` means a single `'='` -character.

Matching any single character A single dot ('.') matches any single character of the alphabet.

Character sets Characters enclosed in brackets (`'['` and `']'`) form a character set. A character set matches any single character in the character list between the brackets. If the first character of the list is a caret (`'^'`), the character set is complemented: it matches any character not present in the list. The list may contain character ranges that are given in the form `"a-z"`. If the list contains characters `']'`, `'^'` or `'-'`, they have to be placed in predefined places in the list. The character `']'` has to be the first character in the list, except when it is part of a complement in which case it has to be the second character. The `'^'` character is taken as a regular caret character if it is not the first character in the list. A regular `'-'` character can be included in the list by writing it at the beginning (after `'^'` and `']'`) or at the end of the list.

Concatenation Two regular expressions are concatenated by writing them consecutively. To match an `'a'` followed by a `'b'`, one writes `"ab"`.

Grouping Parentheses `'('` and `')'` can be used to group regular expressions.

Alternatives A regular expression matching either the regular expression `a` or the regular expression `b` is formed by writing a vertical bar character (`'|'`) between them. A regular expression `"bb|bc"` matches all strings that contain either `"bb"` or `"bc"` or both.

Zero or more occurrences A star ('*') following a regular expression means that the regular expression can be repeated in the matching substring zero or more times. Note that the star has a higher precedence than concatenation, so in order to repeat the regular expression "aa" one has to use grouping: "(aa)*".

One or more occurrences A plus ('+') operates like the star, but a string accepted by the regular expression must be repeated at least once in a matching substring.

Optionality A question mark ('?') marks optionality of a regular expression and it is used in the same way as the star is.

Beginning of string A caret ('^') at the beginning of a regular expression means that the matching substring must start from the beginning of the input string.

End of string A dollar ('$') at the end of a regular expression means that the matching substring must end at the end of the input string.

Variable assignments A variable assignment in a TranSID regular expression is a subexpression of the form "%name=(regex)", where "name" is a valid TranSID identifier (Section 4.5). If the input string contains a match for the whole regular expression, the match for the subexpression is assigned to the variable, and the value of this variable can be used later in the program.

```
Examples:
"a"             // matches a single 'a'
"%%"            // matches a regular '%'
"."             // matches any character including newline
"%."            // matches a regular '.' -character.
"[a-zA-Z0-9]"   // matches an alphanumeric character
"[^0-9]"        // matches any character except a digit
"[-a-z]"        // matches '-' and all lower-case letters
"[a-z-]"        // same as above
"[]a]"          // matches ']' and 'a'
"[^]-0-1]"      // matches all characters except ']', '-', '0' and '1'
"ab"            // matches 'a' followed by 'b'
"a|b"           // matches 'a' or 'b'
"a*"            // matches any number of the character 'a' ("" "a", "aa", ...)
"(ab)*"         // matches repeated "ab":s ("" "ab", "abab", "ababab", ...)
"ab*c"          // matches "ac" ,"abc", "abbc", "abbbc", ...
"ab+c"          // same as above but does not match "ac"
"ab?c"          // matches "ac" and "abc", nothing else
"^ab"           // matches "ab" at the beginning of string
"ab$"           // matches "ab" at the end of string
"a^$b"          // matches "a^$b"
"%foo=(a+)b+    // matches "aaabbb" and assigns "aaa" into variable foo
"%protocol=(http|ftp|telnet)://%server=([a-zA-Z0-9]+(%.[a-zA-Z0-9]+)*)/"
                // matches a valid URL and extracts the protocol and server
                // parts into variables protocol and server
"%a=(%b=(c))"   // Nested variable assignments. String "c" is assigned to both
                // variables a and b
```

Matching a regular expression against a string follows so-called leftmost-longest rule: If there are multiple overlapping substrings that could be used for the match, only the leftmost (having the smallest starting position) is chosen. If there are more than one possible matches starting from the same position, the longest one is chosen. For example, assume that we have a string s = "abcd" and we have a regular regular expressions `r = "ab|bcd"`. In this case the match for the regular expression `r` is `"ab"` because the "ab" has a smaller starting position than `"bcd"`. If the regular expression `r` is `"a|ab*"`, then the match for this regular expression is `"ab"`. This is because the string `s` contains a match for both of the alternatives starting from the same position, but the match for the expression `"ab*"` is longer than the match for the expression `"a"`.

18.2 .indices(regular expression)

The `.indices` function locates longest leftmost matches of the regular expression, given as its parameter, in the input string. For each match, two integers nodes are returned. The first node indicates the starting positions of the match, and the second node gives the position of the first character following the match. Note that a pair of integer nodes (n, n) where n is any position, represents a zero-length match. An empty list is returned if the regular expression did not match the string at all. If the regular expression contains variable assignments, only the first match of the whole regular expression is used for assignments. If the given regular expression is not syntactically correct, an empty list is returned and a warning is issued. Examples:

```
query "aabbcc".indices("a+|b+|d*").map(true, this " ");
```

The result of the above query is:

```
1 3 3 5 5 5 6 6
```

The next example demonstrates how the variable assignments are performed when there are multiple choices for the assignment.

```
query "aabbabb".indices("%foo=(a+)bb").null foo;
```

The result of the above query is:

```
aa
```

18.3 .substrings(regular expression)

The `.substrings` function returns the actual matches (substrings of the input string matching the regular expression) as a list of string nodes. If the regular expression contains variable assignments, only the first match of the whole regular expression is used for assignments. If the given regular expression is not syntactically correct, an empty list is returned and a warning is issued.

```
query "aabbcc".substrings("a+|b+|d*").map(true, <<"match">> {this});
```

The result of the above query is:

```
<match>aa</match><match>bb</match><match></match><match></match>
```

18.4 .matches(regular expression)

The `.matches` function returns a Boolean node: a `TRUE` node if the regular expression matches any substring of the input string, and a `FALSE` node otherwise. If the regular expression contains variable assignments, only the first match of the whole regular expression is used for assignments. If the given regular expression is not syntactically correct, an empty list is returned and a warning is issued.

```
query "abc".matches("ab*c");
```

The result of the above query is:

```
TRUE
```

```
query "ac".matches("ab+c");
```

The result of the above query is:

```
FALSE
```

18.5 .matches_exact(regular expression)

The `.matches_exact` function returns a Boolean node: a `TRUE` node if the regular expression matches the whole input string, a `FALSE` node otherwise. If the regular expression contains variable assignments, only the first match of the whole regular expression is used for assignments. If the given regular expression is not syntactically correct, an empty list is returned and a warning is issued.

```
query "abc".matches_exact("b");
```

The result of the above query is:

```
FALSE
```

18.6 .split(regular expression)

The `.split` function splits its input string into a list of strings, where matches of the regular expression and non–matching substrings are separated from each other. If the regular expression contains variable assignments, only the first match of the whole regular expression is used for assignments. If the given regular expression is not syntactically correct, an empty list is returned and a warning is issued.

```
query "axbxc".split("x+").map(true,<<"TMP">>{this});
```

The result of the above query is:

```
<TMP>a</TMP><TMP>x</TMP><TMP>b</TMP><TMP>x</TMP><TMP>c</TMP>
```

18.7 .clean(regular expression)

The `.clean` function returns a string where all the matches of the regular expression are removed from the input string. If the regular expression contains variable assignments, only the first match of the whole regular expression is used for assignments. If the given regular expression is not syntactically correct, an empty list is returned and a warning is issued.

```
query "axbxc".clean("x+").map(true, <<"TMP">>{this});
```

The result of the above query is:

```
<TMP>a</TMP><TMP>b</TMP><TMP>c</TMP>
```

18.8 .match_replace(regular expression -> replacement expression, ...)

The `.match_replace` function returns a string where all the matches of each given regular expression have been replaced using the corresponding match-replace rule. The left-hand side of a match-replace rule is a regular expression and the right-hand side is an expression to be evaluated when the actual replacement is done. The context variable `this` can be used in the replacement expression to refer to the actual substring matching the regular expression. It is also possible to use the results of variable assignments in the replacement expression. When multiple overlapping matches are found, the following rule is used to decide which match-replace rule is applied: A match with the smallest starting position is always considered first; if there are multiple matches starting from the same position, then the longest match is chosen. In the case of two or more identical matches, the rule declared first in the parameter list is chosen. If one or more of the given regular expressions are not syntactically correct, an empty list is returned and a warning is issued. Examples:

```
query "aaxbbxxccxxxdd".match_replace("x+"->"XXX");
```

The result of the above query is:

```
aaXXXbbXXXccXXXdd
```

```
query "This is a line of text".match_replace("[a-zA-Z]+$" -> this.toupper);
```

The result of the above query is:

```
This is a line of TEXT
```

```
query "aabcc".match_replace("ab" -> "1", "aab" -> "2", "ab*c" -> "3");
```

The result of the above query is:

```
2cc
```

```
query "http://www.cs.helsinki.fi/"
.match_replace("%prot=(http|ftp|telnet)://%svr=([a-zA-Z0-9]+(%.[a-zA-Z0-
9]+)*)/"
->
 "<URL \"PROTOCOL\"=\"" prot "\" \"SERVER\"=\"" svr "\"></URL>" ) ;
```

The result of the above query is:

```
<URL "PROTOCOL"="http" "SERVER"="www.cs.helsinki.fi"></URL>
```

19 Accessing the environment

TranSID provides access to the services of the underlying operating system by reading and writing SGML and text files and by executing subprocesses using pipes. However, by default these features cannot be used when running the TranSID evaluator in server mode, because of obvious security reasons. It is possible to use these features in server mode if the TranSID evaluator is started in the `unsecure` server mode. In the `unsecure` server mode functions defined during the startup of the server are considered "safe" and these functions have access to the features described above. All other functions are considered "unsafe".

19.1 .linearize(filename)

The function `.linearize` allows the programmer to output SGML trees while evaluating a transformation. It is useful for debugging and for splitting big SGML trees into multiple files. The input list is linearized to the given file. Since the file name is given as a TranSID expression, one can generate the file name "on the fly" using the current transformation context.

For example, the following rule writes every section of the source document to a different file and outputs a line like "processed section n" when it has processed a section.

```
TRANSFORMATION BEGIN
!element "SECTION" becomes
// Count the section number
(current.left.having(this.name=="SECTION").count+1).set(num).null
// Write section to file "section-n"
current.linearize("section-" num).null
// Inform the user
("processed section " num).linearize("/dev/tty").null
;
END
```

19.2 .loadtext(filename1, filename2, ...)

The `.loadtext` function reads the specified text files and returns a list with each line of text as a string node. Loadtext can be used to handle other file formats than SGML. Loadtext can also be used to read SGML files without parsing them.

19.3 .parse(file1, file2, ...)

The `.parse` function makes it possible to parse SGML documents also from a TranSID program, not only from the command line. The specified files are passed to the SGML parsing module and a document tree is constructed from them. The function returns the node for the root element of the parsed document tree. Possible parse errors are reported by the logging system using debugging level 6 (see Section 3). If there are any errors, `.parse` returns an empty list.

Normal memory management applies also to trees created with the `.parse` function. The complete document will be available only as long as there is a variable or an executing expression referencing the root of the parsed tree.

19.4 .system(command)

The `.system` function is used to execute arbitrary subprocesses from a TranSID program. `.system` linearizes its input list to the standard input of the subprocess, and reads the output of the subprocesses as text, like `.loadtext` does. The parameter `command` is passed to the command interpreter of the operating system, which enables a TranSID programmer to use all the power of the underlying operating system.

Using `.system` to get the current date:

```
query null.system("date");
```

The result of the above query is:

```
Thu Jun  3 14:31:22 EET DST 1999
```

References

[Bel82] Bell Telephone Laboratories, Inc. *UNIX Time-Sharing System: UNIX Programmer's Manual, Vol. 1.* Holt, Rinehart and Winston, 1982.

[Cla96] James Clark. An SGML system confining to International Standard ISO 8879 – Standard Generalized Markup Language, 1996. url: http://www.jclark.com/sp/.

[ISO92] *Information technology — Hypermedia — Time-based Structuring Language (HyTime), ISO/IEC 10744,* ISO and IEC, 1992.

[JKL97] J. Jaakkola, P. Kilpeläinen, and G. Lindén. TranSID: An SGML tree transformation language. In J. Paakki, editor, *Proceedings of the Fifth Symposium on Programming Languages and Software Tools*, pages 72–83, Jyväskylä, Finland, June 1997. Technical Report C-1997-37, University of Helsinki, Department of Computer Science, Finland.