# Transforming an ASF+SDF Specification into a TOOLBUS application

Joris Hillebrand

*Programming Research Group, University of Amsterdam*

*Kruislaan 403, 1098 SJ Amsterdam, The Netherlands*

*Email:* `joris@fwi.uva.nl`

**Abstract**

An algebraic specification of the natural numbers is being translated into C code, such that the modular structure of the specification is preserved. We use the TOOLBUS, a component interconnection architecture, to couple the resulting C programs. The result is a calculation tool, consisting of independent components, which can work with infinitely large (positive) integers without running into some maximum integer value. It supports different forms of representation of numbers in a flexible way.

*Key Words & Phrases:* Formal specifications, Software engineering, Distributed systems, ASF+SDF, TOOLBUS, code generation.

# Contents

# 1 Introduction

Not so long ago, formal specifications were considered the end point of a design process. If the resulting specifications were executable, this design process could be followed by prototyping or a simulation phase: the rewriting of data terms or the simulation of processes.

Nowadays, specifications serve more often as an intermediate stage, from which the source code for other programs can be generated. Examples are the prettyprinter generator in [BV96], and the well-formedness checker for the language $\mu$CRL in [HK95]. Both these applications have first been formally specified using the ASF+SDF meta-environment [Kli93], and then been compiled into C code using the ASF2C compiler [KW93]. Important advantages of generating, for instance, C code out of specifications are:

- The resulting program can be executed in a stand-alone way, or perhaps as a component of some distributed system. The environment used for designing and executing the original specification, is no longer needed. The application thus generated is therefore more portable.
- A C program is usually faster than interpreted (term) rewriting.

A disadvantage of the current method for generating C code from ASF+SDF specifications, is that the modular structure of the specifications is lost. An entire specification is translated into one C program, even worse: into one C function. It would be useful to preserve structure of the original specification. Parts of the C program generated could then be replaced by existing software, for instance.

This paper is an experiment in (manually) transforming an ASF+SDF specification into a stand-alone application, while (roughly) preserving the modular structure of the original specification. For this purpose, we aim at constructing a ToolBus tool in the language C. The ToolBus [BK96b, BK96a] is a perfect application architecture, since it has been designed for the connection of different software components. The ToolBus itself is also an example of an ASF+SDF specification which is manually translated into C code.

The specification that we are going to transform is the specification of the natural numbers from [BS96]. It defines a list structure for the construction of the natural numbers. One can imagine that at some point it may be necessary to change the internal representation of this list structure, or to replace it by an existing list component. Such an act of software engineering can be done much easier when the part to be replaced is a separate module. This is not only true for the original specification, but also for the resulting application. We will, therefore, first transform the entire specification into a single ToolBus tool. Next, we will split it into a program that deals with the list structure, and one that performs arithmetic operations on the naturals denoted by these lists.

The specification from [BS96] has a number of properties that will be explained in the next section. It is our goal to preserve these properties when translating the specification into C, to serve as a ToolBus tool. The fact that the specification uses a list structure of the representation of the natural numbers is an advantage for the resulting C code. When performing arithmetic operations on very large integers in C, we easily run into MAXINT, the maximum integer value defined by the C implementation. By defining the (positive) integers as lists of digits, the only limitation we have is the amount of memory available. As long as the lists representing the natural numbers can be stored, there is no problem.

The organization of this paper is as follows. In the next section, we give a short explanation of the specification style used in [BS96]. Also, we give an introduction to the working of the ToolBus. In Section 3, we translate the ASF+SDF specification of the natural numbers into C, to serve as a ToolBus tool. In Section 4, conversions between the list representation of the natural numbers and text strings are defined. The resulting components are connected to the ToolBus in Section 5. Then in Section 6 we split the C program obtained in Section 3 into a program for list manipulation and a program for arithmetic operations. Finally, in Section 7, we discuss our experiences and point out some directions for further development.

**Related work.** Various projects in the field of code generation and language transformation exist. We will consider a few.

In the Munich project CIP (Computer-aided Intuition-guided Programming) the Wide Spectrum Language CIP-L is introduced [BBB+85]. This language comprises a number of expressive levels ranging from algebraic specifications down to a machine-oriented level using jumps and pointers. Transformation rules between the different levels have been defined. Modularization constructs are available, but the transformation rules immediately eliminate this modularization.

The SPRINT method (Specification, Prototyping & Reusability INTegration) is a formal method for the development of control software of audio/video systems used at Philips [Jon93]. The backbone of the method is the formal specification and design language COLD [FJO+86]. Different components can be compiled into C separately. However, a description of the signature of imported components has to be provided.

In [BO95], a method for generating database application code from entity-relation diagrams is presented. Although each object in these diagrams is translated into a separate piece of code (for instance, an *SQL* program), these objects can not be translated separately. The resulting pieces of application code, generated from the objects in a diagram, need each other for consistent operation.

**Acknowledgements.** Jan Bergstra came up with the idea for this project. He and Alex Sellink explained details of their original ASF+SDF specification. This project would have been impossible to carry out, if Pieter Olivier had not assisted with immense patience in operating the ToolBus. Paul Klint and Mark van den Brand gave useful hints and commented on earlier versions of this paper. Finally, Leon Moonen suggested the title for this paper.

# 2 Primitives

## 2.1 Sequential data algebras

The specification of the natural numbers from [BS96] is a *sequential data algebra* specification. Sequential Data Algebras, SDAs for short, are first-order many-sorted algebras of which the data types comply a number of requirements. In this section, we give a short listing of the requirements that have an immediate visible effect on specifications, illustrated by some samples from the SDA specification of the natural numbers. For a complete description of the requirements for SDAs see [BS96].

**Divergent and Meaningless objects.** The most imposing requirement for SDAs is the presence of a Divergent and a Meaningless element in every sort. One can imagine M to be the result of the top function applied to an empty stack, whereas D can be viewed as a default value for non-terminating computations. It is demanded that D and M are normal forms in every sort. Data elements not equal to D or M will be called *proper* or *well-defined*. A consequence of this approach is that we work with four-valued logic, since D and M are also normal forms in the Boolean sort.

**Sequential computability.** The values D and M can not be altered by function application. Every unary function must return D when applied to D, and must return M when applied to M. Functions with higher arity must be *strict* in one of their arguments, meaning that they return D or M when the argument concerned contains the value D or M respectively. The convention in [BS96] is to use a numerical annotation to indicate in which argument a prefix function is strict. The function eq1 is strict in its first argument. Binary infix functions have an open circle attached to the operator on the side of the argument in which it is strict, so $+\!\circ$ is strict in its second argument.

**Equality function available.** For every sort there has to be an equality function $\mathsf{eq1}(x, y)$, which is strict in its first argument. If $x$ is proper and $y$ is not, $\mathsf{eq1}$ returns $y$. If both $x$ and $y$ are proper, $\mathsf{eq1}$ returns $\mathsf{T}$ or $\mathsf{F}$, depending on whether $x = y$ or not.

Moreover, we will define a conditional operator and a True predicate for every sort. The conditional operator $(x \triangleleft b \triangleright y)$ is strict in its second argument ($b$), meaning that $(x \triangleleft \mathsf{D} \triangleright y)$ returns $\mathsf{D}$ and $(x \triangleleft \mathsf{M} \triangleright y)$ returns $\mathsf{M}$. If $b$ evaluates to $\mathsf{T}$, the conditional operator returns $x$. It returns $y$ if $b$ evaluates to $\mathsf{F}$. The predicate $\mathsf{T}(x)$ returns $\mathsf{T}$ if $x$ is proper.

**Representation independence.** Functions and constants that are introduced in an imported module can not occur in left-hand sides of equations of the importing module. In this way, functions in the importing module do not need to know the internal structure of objects from the imported module. A consequence of this approach is the use of access functions instead of matching on term structure. The constraint does not hold for the universal values $\mathsf{D}$ and $\mathsf{M}$, and the boolean values $\mathsf{T}$ and $\mathsf{F}$.

To illustrate some of the conventions listed above, we present the modules Bool1DA and Bool2DA from [BS96], which define the (four-valued) booleans. The complete specification from [BS96] can be found in Appendix A.

**Bool1DA**

**imports** Layout
**exports**
  **sorts** T4
  **context-free syntax**
    "D" → T4
    "T" → T4
    "F" → T4
    "M" → T4
    "eq1" "(" T4 "," T4 ")" → T4
    "(" T4 "◁" T4 "▷" T4 ")" → T4
    "(" T4 ")" → T4 {**bracket**}
  **variables**
    *b*[*0-9'*]* → T4
**equations**

| | |
|---|---|
| $(b_1 \triangleleft \mathsf{D} \triangleright b_2) = \mathsf{D}$ | [condT1] |
| $(b_1 \triangleleft \mathsf{T} \triangleright b_2) = b_1$ | [condT2] |
| $(b_1 \triangleleft \mathsf{F} \triangleright b_2) = b_2$ | [condT3] |
| $(b_1 \triangleleft \mathsf{M} \triangleright b_2) = \mathsf{M}$ | [condT4] |

$\mathsf{eq1}(b_1, b_2)$ [eqT1]
$= ((\mathsf{T} \triangleleft b_2 \triangleright \mathsf{F}) \triangleleft b_1 \triangleright (\mathsf{F} \triangleleft b_2 \triangleright \mathsf{T}))$

**Bool2DA**

**imports** Bool1DA$^{A.1}$
**exports**
  **sorts** T4-OP
  **context-free syntax**
    "¬" T4 → T4

    T4 "${}^{\circ}\!\vee$" T4 → T4 {**assoc**}
    T4 "$\vee^{\circ}$" T4 → T4 {**assoc**}
    T4 "${}_{\circ}\!\wedge$" T4 → T4 {**assoc**}
    T4 "$\wedge_{\circ}$" T4 → T4 {**assoc**}

    "T" "(" T4 ")" → T4
  **priorities**
    "¬" T4 → T4 >
    {**non-assoc**: T4 "${}^{\circ}\!\vee$" T4 → T4, T4 "${}_{\circ}\!\wedge$" T4 → T4,
    T4 "$\vee^{\circ}$" T4 → T4, T4 "$\wedge_{\circ}$" T4 → T4}
**equations**

| | |
|---|---|
| $\neg\, b = (\mathsf{F} \triangleleft b \triangleright \mathsf{T})$ | [negT1] |
| $b_1 \,{}_{\circ}\!\wedge\, b_2 = (b_2 \triangleleft b_1 \triangleright \mathsf{F})$ | [lfcT1] |
| $b_1 \,{}^{\circ}\!\vee\, b_2 = (\mathsf{T} \triangleleft b_1 \triangleright b_2)$ | [lfdT1] |
| $b_1 \wedge_{\circ} b_2 = b_2 \,{}_{\circ}\!\wedge\, b_1$ | [rfcT1] |
| $b_1 \vee^{\circ} b_2 = b_2 \,{}^{\circ}\!\vee\, b_1$ | [rfdT1] |

$\mathsf{T}(b) = \mathsf{eq1}(b, b)$ [trueT1]

## 2.2 The ToolBus

The ToolBus [BK96b, BK96a] is a component interconnection architecture. It is used for defining the interaction of different components of software, running independently. These days, systems become more and more distributed, meaning that the design, implementation, and execution of different parts of these systems takes place in different environments. Systems can, therefore, become heterogeneous: running on different platforms, or being implemented in different languages. The possibility of reusing existing components contributes to this phenomenon as well.

By letting the ToolBus take care of the interaction between the different components of a system, i.e., the transfer of data, these components can run independently of each other as different tools. These tools, possibly running on different hardware platforms at different places in a network, have to comply to a certain message protocol based on synchronous communication with the ToolBus. This message protocol uses a common data representation format, called 'terms'. Adapters are needed to translate between the native data types of the tool and ToolBus terms.

### 2.2.1 The ToolBus architecture

Following the ToolBus approach, tools are independent programs. Inter-tool communication is therefore not allowed. Inside the ToolBus, several processes take care of the messages sent by the tools. These processes can communicate with each other synchronously by sending messages, or asynchronously by broadcasting notes. In this paper, we will only use synchronous communication. A visual representation of the ToolBus architecture can be found in Figure 1.

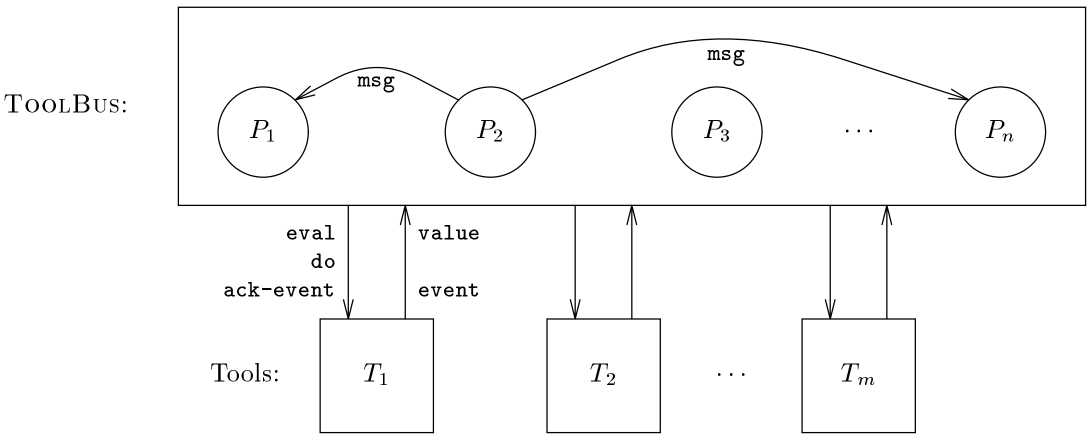


Figure 1: The ToolBus.

It is not necessary to have a one-to-one correspondence between tools and processes. A process can control multiple tools, and single tools can be controlled by more than one process. In this paper, each tool is monitored by one process. Among others, the following messages can be exchanged between ToolBus and tools:

- a tool can send an `event` to the ToolBus. Such an event can for instance be generated by a user pushing some button in a user interface.
- The ToolBus can send an event acknowledgement (`ack-event`) to a tool. This way, the tool can be informed that the appropriate action has been taken as a reaction to an event sent earlier by that tool.
- The ToolBus can send an evaluation request (`eval`) to a tool.

- A tool can send a `value` to the TOOLBUS as the result of a previous evaluation request.
- The TOOLBUS can send a `do` command to a tool. This invokes some action by the tool concerned, but no value is sent back.

Naturally, there are no constraints concerning the implementation of the tools. The TOOLBUS processes are defined by process expressions based on process algebra ([BK84], see [BW90] for an overview). In this paper, we use process expressions of the following form:

- atomic actions, including the sending or receiving of messages, events, event acknowledgements, evaluation requests, and values as mentioned above;
- `delta`, modeling deadlock or inaction;
- $P_1$ `.` $P_2$: first execute $P_1$, and then $P_2$;
- $P_1$ `+` $P_2$: either execute $P_1$, or $P_2$, but not both;
- $P_1$ `*` $P_2$: execute $P_1$ zero or more times, followed by executing $P_2$ only once;
- a call to another defined process: the body of the process concerned is executed.

In this enumeration, $P_1$ and $P_2$ represent arbitrary process expressions. The operator `*` has higher precedence than `.`, which has in turn higher precedence than `+`. We will often use the expression $P$ `* delta`, to model that some process expression $P$ is executed infinitely. The overall behaviour of a TOOLBUS configuration is represented by the parallel composition of a number of TOOLBUS processes, given by the TOOLBUS configuration command `toolbus(`$P_1, \ldots, P_n$`)`.

A common sequence of actions is the following:

```
process P1 is
  let Tid : tool1, E : term, V : term
  in
     execute(tool1, Tid?) .
     ( rec-event(Tid, expr(E?)) .
       snd-msg(compute, E) .
       rec-msg(compute, E, V?) .
       snd-do(Tid, display(V)) .
       snd-ack-event(Tid, expr(E))
     ) * delta
  endlet

tool tool1 is {command = "/usr/bin/..."}
```

```
process P2 is
  let Tid : tool2, X : term, Y : term
  in
     execute(tool2, Tid?) .
     ( rec-msg(compute, X?) .
       snd-eval(Tid, calc(X)) .
       rec-value(Tid, Y?) .
       snd-msg(compute, X, Y)
     ) * delta
  endlet

tool tool2 is {command = "/usr/bin/..."}
```

First, the processes `P1` and `P2` start up the tools `tool1` and `tool2`, respectively. The *tool definitions* `tool <name> is` {`command = "<command>"`} in both scripts link operating system level processes to the abstract tool names used in the script. Also, they implicitly define a new type for each tool. A tool identifier of that type is returned (`Tid`), which the subsequent actions that address the tool must use as first argument. In both processes, the remaining actions are executed infinitely. Process `P2` waits for a message `compute` to receive. Process `P1` can receive an event from `tool1` with an expression to be evaluated. This expression is stored in the variable `E` (a question mark behind a variable name makes that a value is assigned to that variable). Now `P2` sends a message `compute` for the evaluation of `E`. This message is received by `P2`, by which `X` is set to `E`. Next, `P2` sends an evaluation command to `tool2`. This tool does the desired calculation and returns the result, which is stored in `Y`. Process `P2` then answers the `compute` message by sending the same message with the result as extra argument. This message is received by `P1`, by which `V` is set to the value of `Y`. Next, `P1` sends a command to `tool1`, apparently to display the result. Finally, the event originally received from `tool1` is acknowledged.

For a complete treatment of the TOOLBUS, we refer to [BK96b, BK96a].

### 2.2.2 Writing tools for the ToolBus

Tools must be able to communicate with the ToolBus, using the messages listed above. The ToolBus provides a library of C functions for writing tools in C. In this section, the functions that will be mentioned frequently in this paper are explained shortly. A complete overview of ToolBus programming can be found in [Kli96], of which this section contains some samples.

**Terms.** The data type `term` is used to represent tree structures in which information is exchanged between tools and the ToolBus. These tree structures can be built of, among others, integers, strings, terms again, and lists of terms. In C, terms are defined as large unions of these types. The ToolBus library supplies high-level functions for constructing and decomposing terms so that it is not necessary to know the internal structure of terms. These functions will be explained below.

**Term manipulation.** To construct terms, the ToolBus provides the function `TBmake`. The first argument of this function is a character string containing a *term pattern.* This is a literal string that would be obtained by a preorder traversel of the term's tree structure. For instance, the call `TBmake("or(true, false)")` constructs a term of which the root is labeled with the symbol `or`, and of which the children are labeled with `true` and `false`, respectively.

When a term must be constructed from subterms that have to be computed separately, the term pattern can be parameterized with these subterms by using *directives.* These directives indicate that a subterm should be obtained from some given variable. The directive `%s` denotes that this variable should contain a character string, `%d` corresponds to an integer, and `%t` corresponds to a term. Other directives have been defined for binary strings, function symbols, and lists of terms. Now consider the following declarations:

```
int ten = 10;
char *name = "any";
term *bool_trm = TBmake("or(true, false)"), *t;
```

The statement

```
t = TBmake("example(%t,%d,9,%d,%s)", &bool_trm, 8, &ten, &name);
```

will construct a term `t` with the value

```
example(or(true, false), 8, 9, 10, "any")
```

Reversely, the ToolBus function `TBmatch` can be used to anatomize terms. The first argument of `TBmatch` is the term to dissect. The second argument is a term pattern describing the expected anatomy of this term. For each directive in the term pattern, `TBmatch` has an extra argument containing the address of a variable of the directive's type. If the term to be analysed matches the expected pattern, `TBmatch` returns true, and all subterms indicated by the directives in the term pattern are assigned to the corresponding variables. For instance, given the term `t` above and the following declarations

```
int n1, n2, n3;
char *str;
term *trm;
```

The call

```
TBmatch(t, "example(%t,8,%d,%d,%s)", &trm, &n1, &n2, &n3, &str)
```

returns true and causes the following assignments:

```
trm = TBmake("or(true, false)");
n1 = 8;
n2 = 9;
n3 = 10;
str = "any";
```

**Term communication.** Communication between ToolBus and tools can be initialized either by the ToolBus sending a term to a tool, or vice versa. The sending and receiving of terms by tools is done with the ToolBus functions `TBsend` and `TBreceive`. The function `TBreceive` directs incoming terms to so-called *event handlers*. These are functions which examine incoming terms and call other functions within the tool that are mentioned in the incoming term. The result is sent back to the ToolBus. This interface with event handlers for the actual communication with the ToolBus can be generated automatically from a ToolBus script. The tool programmer does not need to take care of this, so we will not go into it. What is important to know is the following correspondence between actions in ToolBus scripts and functions within tools.

- An action `snd-eval(Tid, f(E))` in a ToolBus script leads to calling a function named `f` with argument `E` in tool `Tid`, which returns a term.
- An action `snd-do(Tid, f(E))` in a ToolBus script leads to calling a function named `f` with argument `E` in tool `Tid`, which returns nothing.
- An action `snd-ack-event(Tid, E)` in a ToolBus script leads to calling a function named `rec_ack_event` with argument `E` in tool `Tid`, which returns nothing.

To illustrate this, consider the processes `P1` and `P2` in Section 2.2.1. The `snd-do` action in process `P1` leads to calling a function `display` in `tool1` with argument `V`. The `snd-eval` action in process `P2` leads to calling a function `calc` in `tool2` with argument `X`.

Many tools have the form of an infinite loop that processes incoming events (terms). The function `TBeventloop` takes care of this by repeatedly calling `TBreceive`.

**Memory management.** The functions in the ToolBus library provide automatic memory management of terms. Each time an event handler is invoked (by the function `TBreceive`), terms that are no longer referenced are removed via garbage collection. During the execution of an event handler, it is guaranteed that no garbage collection takes place. Temporary terms that are constructed during the execution of an event handler will therefore not be discarded in their lifetime, but terms that should exist longer must be protected. This is done by the ToolBus function `TBprotect`. Protection can be undone by the function `TBunprotect`.

# 3 The SDA specification in C

In this section, we present a transformation of the specification presented in [BS96] into C, to serve as a ToolBus tool. We do the translation by hand, to be able to use the ToolBus library. This way, the resulting C code will be optimally geared to be used as a ToolBus tool. An important goal is that the resulting C code has strong resemblance with the original specification. In particular, we aim at preserving the qualities of SDAs in the resulting C code. We will assume a basic knowledge of the C programming language. See any C reference manual for details, for instance [KR78].

The SDA specification of the natural numbers can be divided into four parts:

1. four-valued booleans,
2. digits,
3. lists of digits, and
4. arithmetic operations.

For each of these parts, we will point out how we obtained the corresponding C code for the ToolBus. See Appendix C for the complete specification in C.

## 3.1 Booleans

In Section 2.1, we listed the ASF+SDF specification of the (four-valued) booleans. Defining a corresponding C-type is easy. In [BS96] a fixed convention for identifying truth-values with non-negative integers is proposed: 0 represents F, 1 represents T, 2 represents D, and 3 represents M. Such an identification corresponds directly to the notion of enumeration types in C. The type definition

```
typedef enum T4 {F, T, D, M} T4;
```

defines the type `T4` to consist of the constants `F`, `T`, `D`, and `M`, in which the first constant is associated with the integer 0, the second constant with the integer 1, and so on.

Having given a type definition for the booleans, we proceed with the functions defined on the booleans. See modules Bool1DA and Bool2DA in Section 2.1 for the ASF+SDF equations. An exceptional case is the conditional operator. A straightforward equivalent in C would be the following function:

```
T4 cond(T4 b1, T4 op, T4 b2)
{
  switch(op){
     case F: return b2;
     case T: return b1;
     case D: return D;
     case M: return M;
  }
}
```

The problem with a function definition like this is that in case it is called, first all arguments are evaluated. This will cause problems in the case of the conditional operator. Other functions with recursion will very often have their recursive call in the first or the third argument of the conditional operator, with the stop condition in the second argument. It is therefore essential that the second argument is evaluated first, to avoid infinite loops. It is for this that we model the conditional operator as a conditional expression.

```
#define cond(b1, op, b2) (op == F ? b2 : (op == T ? b1 : (op == D ? D : M)))
```

With this definition, `b1` and `b2` are only evaluated when `op` is equal to `T` or `F` respectively. We use a macro to be able to keep using the call `cond(b1, op, b2)`.

The other functions of Bool1DA and Bool2DA can be defined in C using ordinary function definitions. When the corresponding ASF+SDF function has only one equation, it could be considered a macro. We will not use C macros, because the equation may contain recursive calls. Since macros are substituted compile-time, this would result in infinite preprocessing. As an illustration, we present the C version of the equality function.

```
T4 eq1(T4 b1, T4 b2)
{
  return cond(cond(T, b2, F), b1, cond(F, b2, T));
}
```

What we see is a direct translation of the corresponding ASF+SDF equation. See Appendix C.1 for the C version of the other functions defined on the booleans.

## 3.2 Digits

Defining a C-type for the digits is not as straightforward as it is for the booleans. The digits in ASF+SDF have been defined as follows:

**sorts** D4
**lexical syntax**
  [0-9] $\rightarrow$ D4
**context-free syntax**
  "D" $\rightarrow$ D4
  "M" $\rightarrow$ D4

Because we want to be able to perform arithmetic operations on very large numbers without running into MAXINT, we avoid the use of integers as much as possible. Therefore, we have chosen to represent the digits 0–9 as strings `"0"`, `"1"`, ..., `"9"`. In the next section, we define the natural numbers as lists of these strings. The special constants D and M must also be elements of the sort D4. We can of course define another enumeration type containing the strings `"0"`, `"1"`, ..., `"9"` and D and M. In the next section, however, we would encounter serious problems if we tried to define a C-type that contains both a list structure and the constants D and M. This is exactly what the built-in ToolBus type `term` was defined for (see Section 2.2.2), so we will use this type to define the digits and, later on, the naturals.

When we use the `term` type to represent the digits, we have to declare variables representing 0–9, D, and M. For 0–9, this is done in the following piece of C code:

```
term *d[10];

void initDigitDA()
{
  int i;
  char buf[2];

  for(i=0; i<10; i++) {
     sprintf(buf, "%d", i);
     d[i] = TBmake("%s", buf);
     TBprotect(&d[i]);
  }
}

#define base d[9]
```

The variable `d` is an array of ten (pointers to) terms. Next, for the values 0 to 9, a text buffer (`buf`) containing that value is created. Then the ToolBus function `TBmake` creates a term out of the buffer and assigns it to the corresponding array element. The first argument of `TBmake` makes clear that the term has to be a string; the second argument gives the contents of the string: `buf`. We now have variables `d[0]` to `d[9]`, containing the strings `"0"` to `"9"`, representing the values 0–9. Because we must be able to use the digits during the entire program, we protect the variables against garbage collecting with the ToolBus function `TBprotect`. The last line is used to mark that we work with decimal numbers, and corresponds to the ASF+SDF equation

base = 9 [base1]

We still have to define terms for D and M. We can not use the `D` and `M` objects we defined for the booleans, because those were elements of an enumeration type, while we need objects of type `term`. We therefore define variables `Dtrm` and `Mtrm`, containing the strings `"D"` and `"M"` respectively.

```
term *Dtrm;
term *Mtrm;

void init()
{
```

```
    Dtrm = TBmake("\"D\"");
    TBprotect(&Dtrm);
    Mtrm = TBmake("\"M\"");
    TBprotect(&Mtrm);
  }
```

The variables Dtrm and Mtrm must be protected as well. Given these preliminaries, we can define the conditional operator on terms:

```
#define cond_trm(d1, op, d2) (op == F ? d2 : (op == T ? d1 : (op == D ? Dtrm : Mtrm)))
```

This definition is similar to the one for the booleans, except for the D and M objects to be returned: Dtrm and Mtrm instead of D and M.

Now that we have defined terms representing the digits, D, and M, we are able to define functions on the digits. We already saw for the booleans how functions consisting of only one equation can be translated into C directly. In case of the digits, the True predicate is such a function:

$\mathsf{T}(d) = \mathsf{Z}(d) \mathbin{\overset{\circ}{\vee}} \mathsf{T}$ [trueD1]

```
T4 True_D4(term *dt)
{
  return or1(Z(dt), T);
}
```

When an ASF+SDF function is defined with more equations, these are either conditional equations, or equations with pattern matching in at least one of their arguments. The corresponding C function must then make the same distinctions. See for example the Predecessor function defined on the digits:

| | | |
|---|---|---|
| $\mathsf{P}(0)$ | $= \mathsf{M}$ | [pred0] |
| $\mathsf{P}(1)$ | $= 0$ | [pred1] |
| $\mathsf{P}(2)$ | $= 1$ | [pred2] |
| $\mathsf{P}(3)$ | $= 2$ | [pred3] |
| $\mathsf{P}(4)$ | $= 3$ | [pred4] |
| $\mathsf{P}(5)$ | $= 4$ | [pred5] |
| $\mathsf{P}(6)$ | $= 5$ | [pred6] |
| $\mathsf{P}(7)$ | $= 6$ | [pred7] |
| $\mathsf{P}(8)$ | $= 7$ | [pred8] |
| $\mathsf{P}(9)$ | $= 8$ | [pred9] |
| $\mathsf{P}(\mathsf{M})$ | $= \mathsf{M}$ | [predM] |
| $\mathsf{P}(\mathsf{D})$ | $= \mathsf{D}$ | [predD] |

```
term *P(term *dt)
{ char *s;

  if(TBmatch(dt, "%s", &s)) {
     switch(s[0]){
        case '0': return Mtrm;
        case '1': return d[0];
        case '2': return d[1];
        case '3': return d[2];
        case '4': return d[3];
        case '5': return d[4];
        case '6': return d[5];
        case '7': return d[6];
        case '8': return d[7];
        case '9': return d[8];
        case 'D': return Dtrm;
        case 'M': return Mtrm;
        default: assert(0);
     }
  } else {
      TBmsg("panic in P: %t\n", dt);
      return Mtrm;
  }
}
```

In equation [pred0] we see how M can be used in equations. The predecessor of 0 has no meaning. Note that D and M stay unaltered after application of P.

The corresponding C function receives a term, which should contain a string (one of the strings "0" to "9"). This is checked by the ToolBus function TBmatch. The term pattern "%s" points out that we expect the term dt to consist of a string. If dt indeed contains a string, it is assigned to the variable `s`. Subsequently, we examine the first (and only) character of the string, which can be found on position 0 (`s[0]`), and return the appropriate value.

If the term `dt` does not contain a string, an error message is generated, and `Mtrm` is returned.

## 3.3 Lists of digits

Because the built-in ToolBus type `term` can also store lists of terms, we use this type again to define the lists of digits. Let us first see how the lists of digits have been defined in Asf+Sdf.

**sorts** LD4
**context-free syntax**

| | | | |
|---|---|---|---|
| "D" | $\rightarrow$ LD4 | | |
| "M" | $\rightarrow$ LD4 | | |
| "[ ]" | $\rightarrow$ LD4 | | |
| "[" D4 "]" | $\rightarrow$ LD4 | | |
| LD4 "$^\circ\!\frown$" LD4 | $\rightarrow$ LD4 | {**non-assoc**} | |

**equations**

$$[\mathsf{D}] = \mathsf{D} \qquad [\text{lscL1}]$$
$$[\mathsf{M}] = \mathsf{M} \qquad [\text{lscL2}]$$
$$[\;] \,^\circ\!\frown\, Y = Y \qquad [\text{lscL3}]$$
$$Y \,^\circ\!\frown\, [\;] = Y \qquad [\text{lscL4}]$$
$$\mathsf{M} \,^\circ\!\frown\, Y = \mathsf{M} \qquad [\text{lscL5}]$$
$$\mathsf{D} \,^\circ\!\frown\, Y = \mathsf{D} \qquad [\text{lscL6}]$$
$$[d] \,^\circ\!\frown\, \mathsf{D} = (\mathsf{D} \lhd \mathsf{T}(d) \rhd \mathsf{D}) \qquad [\text{lscL7}]$$
$$[d] \,^\circ\!\frown\, \mathsf{M} = (\mathsf{M} \lhd \mathsf{T}(d) \rhd \mathsf{M}) \qquad [\text{lscL8}]$$
$$(Y_1 \,^\circ\!\frown\, Y_2) \,^\circ\!\frown\, Y_3 = Y_1 \,^\circ\!\frown\, (Y_2 \,^\circ\!\frown\, Y_3) \qquad [\text{lscL9}]$$

Besides the special objects D and M, we have an empty list ([ ]), a unary list ([ D4 ]), and the concatenation of two lists ($^\circ\!\frown$). The ToolBus provides a number of built-in functions for the construction and manipulation lists of terms, which we will use. We should, however, take the specific behaviour of the functions towards the D and M elements into consideration. To keep this behaviour under control, we define our own equivalents of the empty list, the unary list, and list concatenation. These equivalents in turn make use of the built-in functions, but after having checked that the elements are well-defined. To start with, we present the equivalents of the empty and the unary list.

```
term *EMPTY(void)
{
  return NULL;
}
```

```
term *Ulist(term *dt)
{ char *s;

  if(TBmatch(dt, "%s", &s)) {
     switch(s[0]){
        case 'D': return Dtrm;
        case 'M': return Mtrm;
        default: return list_concat_term(EMPTY(), dt);
     }
  } else {
      TBmsg("panic in Ulist: %t\n", dt);
      return Mtrm;
  }
}
```

The function EMPTY just returns a pointer to NULL, which corresponds to the internal representation of an empty term list. When constructing a unary list, we have to respect equations [lscL1] and [lscL2]. This is done in the function `Ulist`. The term argument of this function should represent a digit, and therefore contain a string. If this is the case, we check the first character of the string. If this is the character D or M, we are dealing with one of the cases modeled in equations [lscL1] and [lscL2], and we return Dtrm or Mtrm respectively. In the other cases we call the ToolBus function `list_concat_term` to construct a unary list out of the empty list and the term representing the digit.

The equations [lscL3] to [lscL9] define the behaviour of the concatenation operator with respect to the empty list and the special constants D and M. We are not interested in the internal representation of term lists, so we can ignore equation [lscL9]. The equations [lscL5] to [lscL8] model the concatenation operator being strict in its first argument. First the left argument is checked for properness. If this is the case, the right argument is inspected. Only if both arguments are well-defined, the application of the concatenation operator does not rewrite into D or M. This behaviour is simulated in the C function Lappend:

```
term *Lappend(term *y1, term *y2)
{
  return cond_trm(cond_trm(list_append(y1, y2),
                           True_LD4(y2),
                           Mtrm),
                  True_LD4(y1),
                  Mtrm);
}
```

The function True_LD4 is the True predicate for digit lists. The ToolBus function `list_append` is only called if True_LD4 is true for both arguments of Lappend. The function `list_append` has been implemented such that equations [lscL3] and [lscL4] are obeyed. If one of the arguments is the empty list, the other argument is returned, or NULL if both arguments contain the empty list.

Having defined C equivalents for the constructor functions of the digit lists, we can implement the other functions as usual. As an example we give the function that returns the leftmost element of a digit list.

$$\Lsh\,\mathsf{D} = \mathsf{D} \qquad [\mathsf{lhdL1}]$$
$$\Lsh\,\mathsf{M} = \mathsf{M} \qquad [\mathsf{lhdL2}]$$
$$\Lsh\,[\,] = \mathsf{M} \qquad [\mathsf{lhdL3}]$$
$$\Lsh\,[d] = d \qquad [\mathsf{lhdL4}]$$

$$\Lsh\,([d] \,{}^{\circ}\!\frown\, Y) \qquad [\mathsf{lhdL5}]$$
$$= (d \lhd \mathsf{T}(d) \mathbin{{}_{\circ}\!\wedge} \mathsf{T}(Y) \rhd \mathsf{M})$$

```
term *LeftHead(term *y)
{ char *s;
  term_list *t;

  if(TBmatch(y, "%s", &s)) {
     switch(s[0]){
        case 'D': return Dtrm;
        case 'M': return Mtrm;
        default: assert(0);
     }
  } else if(TBmatch(y, "[%l]", &t)) {
      if(!t) return Mtrm;
      else return
             cond_trm(list_first(t),
                      and1(True_D4(list_first(t)),
                           True_LD4(list_next(t))),
                      Mtrm);
  } else {
      TBmsg("panic in LeftHead: %t\n", y);
      return Mtrm;
  }
}
```

The function `LeftHead` receives a term `y` representing a list of digits. In case of a well-defined list, `y` contains a list of terms. The special constants D and M, however, are also elements of the sort LD4, and can be passed to `LeftHead`. In that case, `y` contains the string `"D"` or `"M"`. The function `LeftHead` first checks whether this is the case. Every other string is erroneous, because lists of digits are represented as lists of terms. If `y` does not contain a string, `LeftHead` checks as to whether it contains a list of terms. This is represented by the term pattern `"[%l]"` in the second argument of the function `TBmatch`. If `y` does contain a list of terms, this list is assigned to the term `t`[1]. If the list is empty (`!t`), the `LeftHead` function has no meaning. If it is not empty, the first element is returned using the ToolBus function `list_first`, provided that all list elements are well-defined. The Asf+Sdf equations [lhdL4] and [lhdL5] can be combined here, because `list_next` returns `NULL` in case of a unary list and `NULL` is proper.

## 3.4 Arithmetic operations

Up to this point, we have defined the digits as string terms and the natural numbers as lists of these terms. A number of operations on lists have been transformed from Asf+Sdf to C. What still needs to be done is to give C versions of the functions modeling addition, subtraction, multiplication, division, and less-than. These functions have all been defined with only one equation, so we can immediately translate the equations into C. In [BS96], two versions of the arithmetic operations have been given. One version follows the standard recursive definitions. In this version, addition is defined as follows:

$$n_1 \mathbin{+\!\circ} n_2 = (\mathsf{rlz}(n_1) \lhd \mathsf{Z}(n_2) \rhd \mathsf{S}(n_1 \mathbin{+\!\circ} \mathsf{P}(n_2))) \qquad [\mathsf{AddLsp}]$$

Here, S and P are the successor and predecessor function for natural numbers. The predicate Z checks whether a natural number is equal to zero. The function rlz removes leading zeros from natural numbers. This equation looks like this in C:

```
term *add2(term *n1, term *n2)
{
  return cond_trm(rlz(n1), lZ(n2), lS(add2(n1, lP(n2))));
}
```

The successor function, the predecessor function, and the zero predicate for natural numbers (lists of digits) are called `lS`, `lP`, and `lZ` here. This is necessary because C has no function overloading, and there are already functions called `S`, `P`, and `Z` for digits.

The other version defines the arithmetic operations in terms of list manipulation. These definitions are less standard, but follow more standard algorithms. Moreover, these definitions do not have exponential complexity, as is the case with the recursive definitions. In this version, we have the following definition of addition:

$$\frac{d_1 = \mathsf{carry1}(n_1 \Lsh, n_2 \Lsh),\ d_2 = \mathsf{modsum1}(n_1 \Lsh, n_2 \Lsh)}{\begin{aligned} n_1 \mathbin{+\!\circ} n_2 = {} & (\mathsf{rlz}(n_1) \lhd \mathsf{Z}(n_2) \\ & \rhd (\mathsf{rlz}(n_2) \lhd \mathsf{Z}(n_1) \rhd (n_1 \lrcorner \mathbin{+\!\circ} n_2 \lrcorner \mathbin{+\!\circ} [d_1]) \frown [d_2])) \end{aligned}} \qquad [\mathsf{Add1}]$$

The functions $\Lsh$ and $\lrcorner$ return the rightmost element of a list and the list minus the rightmost element respectively. These functions are called `RightHead` and `RLTail` (right-left tail) in C. The function carry1 returns 1, if the sum of two digits is greater than 9, and 0 otherwise. The function modsum1 returns the sum of two digits modulo 10. The C equivalent of this equation looks as follows:

```
term *add2(term *n1, term *n2)
{
  return cond_trm(rlz(n1),
```

[1] The type `term_list` is just another name for `term` and is used to stress that we are dealing with a term list.

```
                    lZ(n2),
                    cond_trm(rlz(n2),
                             lZ(n1),
                             Lappend(add2(add2(RLTail(n1), RLTail(n2)),
                                           Ulist(carry1(RightHead(n1),
                                                        RightHead(n2)))),
                                     Ulist(modsum1(RightHead(n1),
                                                   RightHead(n2))))));
}
```

This completes the definition of the basic arithmetic operations on the natural numbers. In the next section, we will present various ways to represent the natural numbers.

# 4 Representation

The specification in [BS96] defines the naturals as lists of digits and defines a number of arithmetic operations on them. The list structure for the representation of natural numbers has appeared to be useful when performing calculations on very large numbers. A problem is the reading, and perhaps even more the writing, of natural numbers. It is not that easy to write down $[1] \circ\!\!\frown [2] \circ\!\!\frown [3] \circ\!\!\frown [4]$ when we mean 1234. This is why we specify transformation functions between text strings and the list representation of natural numbers. We support the following notations:

1. a string consisting of only digits (12345);
2. a digit string with a dot before every three digits (12.345);
3. the same, but with a space instead of a dot (12 345);
4. as in 2, but no dot in case of four digits (12.345, but 1234).

We will first give an SDA specification of these transformation functions in Asf+Sdf. The complete specification can be found in Appendix B. Then we show how this is implemented in C. The C version of the representation functions is presented in Appendix C.7.

## 4.1 Representation functions in Asf+Sdf

The specification of a transformation function between the lists of digits and text strings starts with defining a sort for the text strings. These text strings consist of the digits 0–9, possibly interrupted by dots or spaces. Also the universal values D and M can be returned by the calculations, as well as the boolean values T and F in case of the less-than operator. This yields us the following sort definition for the output strings:

**sorts** OutString4
**lexical syntax**
  [0-9_.]+ → OutString4
  [FTMD] → OutString4
**context-free syntax**
  "(" OutString4 "◁" T4 "▷" OutString4 ")" → OutString4
  "eq1" "(" OutString4 "," OutString4 ")" → T4
  "T" "(" OutString4 ")" → T4
  "(" OutString4 ")" → OutString4 {**bracket**}

In lexical constructor functions, the use of spaces is problematic, because the space is a layout symbol. If we would define the space to be part of the sort OutString4 as well, all spaces used in the specification would be interpreted as instances of this sort. We will therefore use an underscore to represent the space. In the C version, we simply use spaces. We aim at following the conventions for sequential data algebras, so we present a conditional operator, an equality function, and a True predicate. Manipulating a data type that is constructed out of lexical syntax only, requires some technical knowledge of the ASF+SDF formalism. Let us therefore look at the equations for the True predicate for the sort OutString4.

| | | | |
|---|---|---|---|
| $\mathsf{T}(0) = \mathsf{T}$ | [trueOS-1] | $\mathsf{T}(8) = \mathsf{T}$ | [trueOS-9] |
| $\mathsf{T}(1) = \mathsf{T}$ | [trueOS-2] | $\mathsf{T}(9) = \mathsf{T}$ | [trueOS-10] |
| $\mathsf{T}(2) = \mathsf{T}$ | [trueOS-3] | $\mathsf{T}(\_) = \mathsf{T}$ | [trueOS-11] |
| $\mathsf{T}(3) = \mathsf{T}$ | [trueOS-4] | $\mathsf{T}(.) = \mathsf{T}$ | [trueOS-12] |
| $\mathsf{T}(4) = \mathsf{T}$ | [trueOS-5] | $\mathsf{T}(\mathtt{T}) = \mathsf{T}$ | [trueOS-13] |
| $\mathsf{T}(5) = \mathsf{T}$ | [trueOS-6] | $\mathsf{T}(\mathtt{F}) = \mathsf{T}$ | [trueOS-14] |
| $\mathsf{T}(6) = \mathsf{T}$ | [trueOS-7] | $\mathsf{T}(\mathtt{D}) = \mathsf{D}$ | [trueOS-15] |
| $\mathsf{T}(7) = \mathsf{T}$ | [trueOS-8] | $\mathsf{T}(\mathtt{M}) = \mathsf{M}$ | [trueOS-16] |

$$\mathsf{T}(\mathsf{outstring4}(\mathit{Chs}_1\ \ \mathit{Chs}_2)) = (\mathsf{T}(\mathsf{outstring4}(\mathit{Chs}_2)) \lhd \mathsf{T}(\mathsf{outstring4}(\mathit{Chs}_1)) \rhd \mathsf{M}) \qquad \text{[trueOS-17]}$$

We see that for a one-character string the True predicate is defined as we would expect. For every character in the sort OutString4 this predicate returns T, except for the characters D and M. Thus the requirements for sequential data algebras are met. Equation [trueOS-17] deals with strings of length greater than one. One should read this equation as follows: a string consisting of two substrings is well-defined if both substrings are well-defined. The variables $\mathit{Chs}_1$ and $\mathit{Chs}_2$ are of sort CHAR+, meaning that they represent one or more arbitrary characters. The sort CHAR is a built-in sort of ASF+SDF. For every lexical sort XYZ, a context-free function

$$\mathsf{xyz}\ (\ \mathsf{CHAR+}\ ) \rightarrow \mathsf{XYZ}$$

is implicitly available in ASF+SDF specifications, which can be used when writing equations over this sort. So the term outstring4($\mathit{Chs}_1$ $\mathit{Chs}_2$) is a legal function of sort OutString4. The substrings outstring4($\mathit{Chs}_1$) and outstring4($\mathit{Chs}_2$) of sort OutString4 are each passed to the function T again, where they in turn are split, until substrings of length one are left, which can be handled by one of the equations [trueOS-1] to [trueOS-16]. The sort CHAR nor the function outstring4 is imported from another module; they are built-in, so we do not violate the representation independence rule for SDAs.

### 4.1.1 Generating strings

We have four functions for the transformation of digit lists to output strings.

$$\begin{aligned}
&\mathsf{repres\text{-}n}(\mathsf{LD4}) &&\rightarrow \mathsf{OutString4}\\
&\mathsf{repres\text{-}s}(\mathsf{LD4}) &&\rightarrow \mathsf{OutString4}\\
&\mathsf{repres\text{-}d}(\mathsf{LD4}) &&\rightarrow \mathsf{OutString4}\\
&\mathsf{repres\text{-}nfd}(\mathsf{LD4}) &&\rightarrow \mathsf{OutString4}
\end{aligned}$$

The function repres-n generates strings with no dots or spaces, repres-d generates strings with dots before every three digits, repres-s does the same with spaces. The function repres-nfd generates strings with dots before every three digits, but only for strings with at least five digits (not with four digits). The same four functions have been defined on the booleans. They all have the same behaviour: generating the characters F, T, D, or M.

$$\mathsf{repres\text{-}n}(\mathsf{F}) = \mathtt{F} \qquad \text{[repres-n-1]}$$
$$\mathsf{repres\text{-}n}(\mathsf{T}) = \mathtt{T} \qquad \text{[repres-n-2]}$$
$$\mathsf{repres\text{-}n}(\mathsf{D}) = \mathtt{D} \qquad \text{[repres-n-3]}$$

$$\mathsf{repres\text{-}n}(\mathsf{M}) = \mathtt{M} \qquad \text{[repres-n-4]}$$

$$\begin{aligned}
\mathsf{repres\text{-}n}(Y) \;=\; & (0 \lhd \mathsf{eq1}(Y, [\,]) \\
& \rhd (0 \lhd \mathsf{eq1}(Y, [0]) \\
& \quad \rhd (1 \lhd \mathsf{eq1}(Y, [1]) \\
& \quad\quad \rhd (2 \lhd \mathsf{eq1}(Y, [2]) \\
& \quad\quad\quad \rhd (3 \lhd \mathsf{eq1}(Y, [3]) \\
& \quad\quad\quad\quad \rhd (4 \lhd \mathsf{eq1}(Y, [4]) \\
& \quad\quad\quad\quad\quad \rhd (5 \lhd \mathsf{eq1}(Y, [5]) \\
& \quad\quad\quad\quad\quad\quad \rhd (6 \lhd \mathsf{eq1}(Y, [6]) \\
& \quad\quad\quad\quad\quad\quad\quad \rhd (7 \lhd \mathsf{eq1}(Y, [7]) \\
& \quad\quad\quad\quad\quad\quad\quad\quad \rhd (8 \lhd \mathsf{eq1}(Y, [8]) \\
& \quad\quad\quad\quad\quad\quad\quad\quad\quad \rhd (9 \lhd \mathsf{eq1}(Y, [9]) \\
& \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \rhd \mathsf{repres\text{-}n}([\urcorner\, Y]) \circ\!\!+\!\!+\; \mathsf{repres\text{-}n}(\llcorner\, Y)))))))))))))
\end{aligned} \qquad \text{[repres-n-5]}$$

For digit lists, repres-n compares the current list to the empty list and every possible unary list. In case of a match, the appropriate digit is returned. In case of an empty list, zero is returned (some arithmetic operations return the empty list instead of zero). The empty list can never be part of a non-empty list (as a constructor), so there is no danger of producing too many zeros. When the list is of length greater than one, it is split into its first element and the rest. The two resulting strings are then concatenated by the function $\circ\!\!+\!\!+$. If one of the list elements is D or M, then eq1 immediately returns this value. Subsequently, the entire result string turns into D or M via $\circ\!\!+\!\!+$.

The functions repres-d and repres-s are defined in terms of repres-n.

$$\begin{aligned}
\mathsf{repres\text{-}d}(Y) \;=\; & (\mathsf{repres\text{-}n}(Y) \\
& \lhd \mathsf{eq1}(\mathsf{length}(Y), [\,]) \mathrel{\mathring{\vee}} \mathsf{eq1}(\mathsf{length}(Y), [1]) \\
& \quad \mathrel{\mathring{\vee}} \mathsf{eq1}(\mathsf{length}(Y), [2]) \mathrel{\mathring{\vee}} \mathsf{eq1}(\mathsf{length}(Y), [3]) \\
& \rhd \mathsf{repres\text{-}d}(Y \lrcorner \lrcorner \lrcorner) \circ\!\!+\!\!+ \;\mathtt{.}\; \circ\!\!+\!\!+ \mathsf{repres\text{-}n}([Y \lrcorner \lrcorner \ulcorner] \mathbin{\mathring{\frown}} ([Y \lrcorner \ulcorner] \mathbin{\mathring{\frown}} [Y \ulcorner])))
\end{aligned} \qquad \text{[repres-d-2]}$$

Lists with less than four elements can be handled by repres-n. Longer lists are split into the three rightmost elements and the rest. The list with the three rightmost elements ($[Y\lrcorner\lrcorner\ulcorner] \mathbin{\mathring{\frown}} [Y\lrcorner\ulcorner] \mathbin{\mathring{\frown}} [Y\ulcorner]$) can also be handled by repres-n. A dot is put in front of the result and all this is appended to the conversion of the first part of the list ($Y\lrcorner\lrcorner\lrcorner$).

The function repres-nfd behaves like repres-d, except for lists of length four. In that case it behaves like repres-n.

$$\mathsf{repres\text{-}nfd}(Y) = (\mathsf{repres\text{-}n}(Y) \lhd \mathsf{eq1}(\mathsf{length}(Y), [4]) \rhd \mathsf{repres\text{-}d}(Y)) \qquad \text{[repres-nfd-2]}$$

### 4.1.2 Converting strings

For the conversion of strings to digit lists, we have functions read-d for the conversion of strings containing dots, read-s for strings containing spaces, and read-n for strings containing neither dots nor spaces. The main function read decides which one is needed.

$$\begin{aligned}
\mathsf{read}(Os) \;=\; & ((\mathsf{M} \lhd \mathsf{contains\text{-}d}(Os) \rhd \mathsf{read\text{-}s}(Os)) \\
& \lhd \mathsf{contains\text{-}s}(Os) \\
& \rhd (\mathsf{read\text{-}d}(Os) \lhd \mathsf{contains\text{-}d}(Os) \rhd \mathsf{read\text{-}n}(Os)))
\end{aligned} \qquad \text{[read-1]}$$

The functions contains-d and contains-s check whether a string contains dots or spaces. If a string contains both dots and spaces, read returns M. The actual conversion of strings to lists of digits is done by read-n. The functions read-d and read-s eventually make use of this function.

| | | | |
|---|---|---|---|
| read-n(0) = [0] | [read-n-1] | read-n(8) = [8] | [read-n-9] |
| read-n(1) = [1] | [read-n-2] | read-n(9) = [9] | [read-n-10] |
| read-n(2) = [2] | [read-n-3] | read-n(_) = M | [read-n-11] |
| read-n(3) = [3] | [read-n-4] | read-n(.) = M | [read-n-12] |
| read-n(4) = [4] | [read-n-5] | read-n(T) = M | [read-n-13] |
| read-n(5) = [5] | [read-n-6] | read-n(F) = M | [read-n-14] |
| read-n(6) = [6] | [read-n-7] | read-n(D) = D | [read-n-15] |
| read-n(7) = [7] | [read-n-8] | read-n(M) = M | [read-n-16] |

read-n(outstring4($Chs_1$ $Chs_2$)) = read-n(outstring4($Chs_1$)) $\frown$ read-n(outstring4($Chs_2$)) [read-n-17]

For each of the digits 0–9, the function read-n returns a unary list containing that digit. The character D gives D, and all other characters result in M, including the dot and the space, because read-n does not expect dots or spaces. Strings of length greater than one are split and the resulting lists are concatenated with $\frown$. The functions read-d and read-s are a bit more complicated. We have to distinct strings of different length and check the fourth character from the right. Here we present the equations for read-d, which handles strings containing dots.

| | | |
|---|---|---|
| read-d(outstring4($Ch$)) | = read-n(outstring4($Ch$)) | [read-d-1] |
| read-d(outstring4($Ch_1$ $Ch_2$)) | = read-n(outstring4($Ch_1$ $Ch_2$)) | [read-d-2] |
| read-d(outstring4($Ch_1$ $Ch_2$ $Ch_3$)) | = read-n(outstring4($Ch_1$ $Ch_2$ $Ch_3$)) | [read-d-3] |
| read-d(outstring4($Ch_1$ $Ch_2$ $Ch_3$ $Ch_4$)) | = M | [read-d-4] |
| read-d(outstring4($Chs$ "0" $Ch_2$ $Ch_3$ $Ch_4$)) | = M | [read-d-5] |
| read-d(outstring4($Chs$ "1" $Ch_2$ $Ch_3$ $Ch_4$)) | = M | [read-d-6] |
| read-d(outstring4($Chs$ "2" $Ch_2$ $Ch_3$ $Ch_4$)) | = M | [read-d-7] |
| read-d(outstring4($Chs$ "3" $Ch_2$ $Ch_3$ $Ch_4$)) | = M | [read-d-8] |
| read-d(outstring4($Chs$ "4" $Ch_2$ $Ch_3$ $Ch_4$)) | = M | [read-d-9] |
| read-d(outstring4($Chs$ "5" $Ch_2$ $Ch_3$ $Ch_4$)) | = M | [read-d-10] |
| read-d(outstring4($Chs$ "6" $Ch_2$ $Ch_3$ $Ch_4$)) | = M | [read-d-11] |
| read-d(outstring4($Chs$ "7" $Ch_2$ $Ch_3$ $Ch_4$)) | = M | [read-d-12] |
| read-d(outstring4($Chs$ "8" $Ch_2$ $Ch_3$ $Ch_4$)) | = M | [read-d-13] |
| read-d(outstring4($Chs$ "9" $Ch_2$ $Ch_3$ $Ch_4$)) | = M | [read-d-14] |
| read-d(outstring4($Chs$ "_" $Ch_2$ $Ch_3$ $Ch_4$)) | = M | [read-d-15] |
| read-d(outstring4($Chs$ "F" $Ch_2$ $Ch_3$ $Ch_4$)) | = M | [read-d-16] |
| read-d(outstring4($Chs$ "T" $Ch_2$ $Ch_3$ $Ch_4$)) | = M | [read-d-17] |
| read-d(outstring4($Chs$ "D" $Ch_2$ $Ch_3$ $Ch_4$)) | = D | [read-d-18] |
| read-d(outstring4($Chs$ "M" $Ch_2$ $Ch_3$ $Ch_4$)) | = M | [read-d-19] |

read-d(outstring4($Chs$ "." $Ch_1$ $Ch_2$ $Ch_3$)) [read-d-20]
= read-d(outstring4($Chs$)) $\frown$ read-n(outstring4($Ch_1$ $Ch_2$ $Ch_3$))

Here the variables $Ch$ and $Ch_1$ to $Ch_4$ represent single characters. The variable $Chs$ represents a sequence of one or more characters. Strings with less than four characters can be handled by read-n ([read-d-1] to [read-d-3]). A string with dots can not be of length four, because the fourth character from the right should be the dot, but then there must be at least one more digit in front of the dot. In strings of length five or more, the fourth character from the right should be the dot. If so, the three rightmost characters are handled by read-n and appended to the list generated out of the part of the string in front of the dot. Strings of length five or more that do not have a dot at the fourth position from the right all generate M.

Note that read-d returns M for strings containing spaces, because read-d only expects strings built out of digits and dots. Strings containing spaces will never be given to read-d, but to read-s, which has a similar behaviour.

## 4.2 The representation functions in C

In C, we will use the standard type `char *` for the text strings that are returned by the representation functions. In ASF+SDF the input strings could only contain digits, dots, spaces (underscores), and the characters D and M. Because the C-type `char *` contains strings of all characters, the functions that convert text strings to lists of digits will have to be more general than their ASF+SDF equivalents.

The conditional operator on strings now returns the strings "D" and "M" if the second argument is equal to D or M respectively.

```
#define cond_str(s1, op, s2) (op == F ? s2 : (op == T ? s1 : (op == D ? "D" : "M")))
```

Also in case of the True predicate for strings we have to take into account that the strings can contain every possible character.

```
T4 True_str(char *s)
{
  while(*s == '0' || *s == '1' || *s == '2' || *s == '3' || *s == '4' ||
        *s == '5' || *s == '6' || *s == '7' || *s == '8' || *s == '9' ||
        *s == ' ' || *s == '.' || *s == 'T' || *s == 'F') s++;
  if (*s == '\0') return T;
  else if (*s == 'D') return D;
  else return M;
}
```

The text string that `True_str` receives is checked character by character. The checking stops at the first character that is not a digit, dot, space, T, or F. If this character is `'\0'`, the end marker for strings, the entire string consisted of the characters mentioned above, meaning that the string was well-defined. In case of the character `'D'`, the boolean D is returned. In all other cases, including the character `'@'` for instance, `True_str` returns M.

The equivalents of the function repres-n for booleans and digit lists are immediate translations from ASF+SDF:

```
char *represb_n(T4 b)
{
  switch(b) {
    case F: return "F";
    case T: return "T";
    case D: return "D";
    default: return "M";
  }
}
```

```
char *repres1_n(term *y)
{
  return cond_str("0", eq1_LD4(y, EMPTY()),
         cond_str("0", eq1_LD4(y, Ulist(d[0])),
         cond_str("1", eq1_LD4(y, Ulist(d[1])),
         cond_str("2", eq1_LD4(y, Ulist(d[2])),
         cond_str("3", eq1_LD4(y, Ulist(d[3])),
         cond_str("4", eq1_LD4(y, Ulist(d[4])),
         cond_str("5", eq1_LD4(y, Ulist(d[5])),
         cond_str("6", eq1_LD4(y, Ulist(d[6])),
         cond_str("7", eq1_LD4(y, Ulist(d[7])),
         cond_str("8", eq1_LD4(y, Ulist(d[8])),
         cond_str("9", eq1_LD4(y, Ulist(d[9])),
         concat1(repres1_n(Ulist(LeftHead(y))),
                 repres1_n(LRTail(y))))))))))))));
}
```

The inverse function `read_str` (the name `read` is already in use by some internal TOOLBUS function) is translated immediately from ASF+SDF as well:

```
term *read_str(char *s)
{
  return cond_trm(cond_trm(Mtrm, contains_d(s), read_s(s)),
                  contains_s(s),
                  cond_trm(read_d(s), contains_d(s), read_n(s)));
}
```

The functions read_n, read_d, and read_s are the C equivalents of read-n, read-d, and read-s respectively. Here we only show read-n.

```
term *read_n(char *s)
{ term *y;

  if(*s == '\0') return Mtrm;
  else { y = readchar(*s);
         while (*++s != '\0')
           y = Lappend(y, readchar(*s));
         return y;
  }
}
```

```
term *readchar(char c)
{
  switch(c) {
    case '0': return Ulist(d[0]);
    case '1': return Ulist(d[1]);
    case '2': return Ulist(d[2]);
    case '3': return Ulist(d[3]);
    case '4': return Ulist(d[4]);
    case '5': return Ulist(d[5]);
    case '6': return Ulist(d[6]);
    case '7': return Ulist(d[7]);
    case '8': return Ulist(d[8]);
    case '9': return Ulist(d[9]);
    case 'D': return Dtrm;
    default: return Mtrm;
  }
}
```

As long as the input string has not ended, the characters are converted to unary strings by the auxiliary function readchar. These unary lists are then concatenated as usual by Lappend. The function readchar makes sure that strings containing non-proper characters are turned into the terms Dtrm or Mtrm.

# 5 Coupling the components

At this point, we have C versions of the naturals, arithmetic operations, and representation functions, all following the conventions of sequential data algebras. We are now going to connect these components to the ToolBus. We have two versions of the arithmetic operations: one recursive (slow) approach, and one based on list manipulation (fast). This results in the ToolBus architecture shown in Figure 2, with two calculation tools, a representation tool, and a user interface.

## 5.1 User interface

We have written a simple user interface in Tcl/Tk [Ous94]. It consists of the windows displayed in Figure 3.

The first window allows the user to choose which calculation method should be used: the one based on recursion or the one based on list manipulation. After a choice has been made, a new window appears in which two naturals can be entered, followed by choosing an arithmetic operation. Finally, the preferred representation used to display the result can be selected. This is done using the window in Figure 4.

After picking one of the four representations, the result of the requested computation will be displayed.

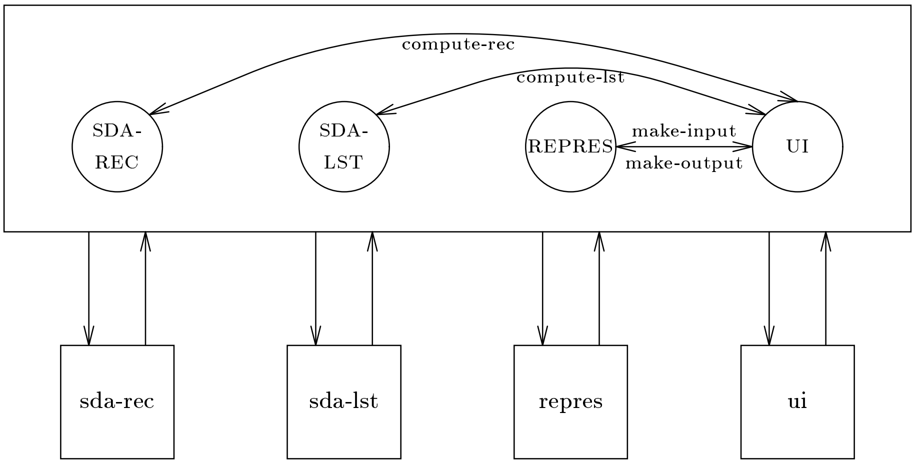


Figure 2: ToolBus architecture with user-interface, representation tool, and calculation tools.


Figure 3: User interface for picking a calculation method and entering an expression.

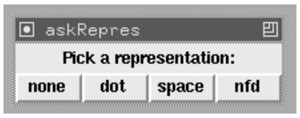


Figure 4: Window for choosing a representation.

## 5.2 ToolBus scripts

All choices regarding the method of computation and representation are made by the user. Consequently, the process monitoring the user interface directs the other processes.

```
process UI is
  let Tid : ui
  in
     execute(ui, Tid?) .
     ( CALC-REC-BUTTON(Tid)
       +
       CALC-LST-BUTTON(Tid)
       +
       QUIT-BUTTON(Tid)
     ) * delta
  endlet
```

The `execute` statement in this script starts the Tcl/Tk application and makes the first window in Figure 3 pop up. For each of the three buttons in the window, a subprocess has been defined. The processes `CALC-REC-BUTTON` and `CALC-LST-BUTTON` are almost identical, they only differ in the tool to which they pass their commands. The first process controls the tool for recursive calculation, the second process does the same for the tool that is based on list manipulation.

```
process CALC-REC-BUTTON(Tid : ui) is
  let  D : str , E : str, U : term , V : term, W : term, S : str
  in
     rec-event(Tid, button(calc-rec)) .
     snd-eval(Tid, get-expr-dialog) .
     ( rec-value(Tid, add(D?, E?)) .
       snd-msg(make-input, D) .  rec-msg(make-input, D, U?) .
       snd-msg(make-input, E) .  rec-msg(make-input, E, V?) .
       snd-msg(compute-rec, add(U, V)) .  rec-msg(compute-rec, add(U, V), W?)
       +
  %% The same for subtraction,
  %% multiplication, and division.
       +
       rec-value(Tid, less(D?, E?)) .
       snd-msg(make-input, D) .  rec-msg(make-input, D, U?) .
       snd-msg(make-input, E) .  rec-msg(make-input, E, V?) .
       snd-msg(compute-rec, less(U, V)) .  rec-msg(compute-rec, less(U, V), W?)
     ) .
     snd-eval(Tid, get-repres-dialog) .
     ( rec-value(Tid, none) .
       snd-msg(make-output, none(W)) .  rec-msg(make-output, none(W), S?) .
       snd-do(Tid, display(S))
       +
  %% The same for the dot and
  %% space representation.
       +
       rec-value(Tid, nfd) .
       snd-msg(make-output, nfd(W)) .  rec-msg(make-output, nfd(W), S?) .
       snd-do(Tid, display(S))
     ) . snd-ack-event(Tid, button(calc-rec))
  endlet
```

The process `CALC-REC-BUTTON` is invoked by receiving the term `button(calc-rec)`. Subsequently, the command `get-expr-dialog` is sent to the user interface tool, which makes the second window from

Figure 3 appear. Depending on which operation button is pushed in that window, one of the terms `add(D?, E?)`, `sub(D?, E?)`, `mult(D?, E?)`, `div(D?, E?)`, or `less(D?, E?)` is received. The strings are first sent to the representation tool to be transformed into digit lists. This happens by sending the message `make-input` with the string to be transformed. The same message is received back, together with the resulting terms `U` and `V`. Next, these lists are sent to the calculation tool, as arguments of a calculation order. In the case of the recursive tool, this is done by sending the message `compute-rec`. The term `W` containing the result of the computation is received back.

The result must now be converted back to a string. For this, the command `get-repres-dialog` is sent to the user interface tool. The effect is the window from Figure 4, in which the user can choose how the result should be displayed. When a button is pushed, one of the values `none`, `dot`, `space`, or `nfd` is received. This results in the message `make-output` being sent to the representation tool, together with the term to be transformed as argument of the proper representation command. The resulting string `S`, which is received back, is sent to the the user interface tool to be displayed.

The process `CALC-REC-BUTTON` above communicates via the `snd-msg` and `rec-msg` actions with the processes `SDA-REC` and `REPRES`. The first process monitors the recursive calculation tool, the other one has control over the representation tool.

```
process SDA-REC is
  let Tid : sda-rec, E : term, V : term
  in
     execute(sda-rec, Tid?) .
     ( rec-msg(compute-rec, E?) .
       snd-eval(Tid, expr(E)) .
       rec-value(Tid, V?) .
       snd-msg(compute-rec, E, V)
     ) * delta
  endlet
```

```
process REPRES is
  let Tid : repres, E : str, V : term
  in
     execute(repres, Tid?) .
     ( rec-msg(make-input, E?) .
       snd-eval(Tid, input(E)) .
       rec-value(Tid, V?) .
       snd-msg(make-input, E, V)
       +
       rec-msg(make-output, V?) .
       snd-eval(Tid, output(V)) .
       rec-value(Tid, E?) .
       snd-msg(make-output, V, E)
     ) * delta
  endlet
```

Both these processes have the same functionality. A request from the process `CALC-REC-BUTTON` is received. In case of `SDA-REC` this is `compute-rec`, the process `REPRES` receives `make-input` or `make-output`. Next, the appropriate command is sent to the tool monitored. The value returned by the tool is then sent back to `CALC-REC-BUTTON`.

The complete ToolBus script can be found in Appendix D.1.

## 5.3 C interfaces

In Section 3, we gave a C equivalent of the SDA specification of the natural numbers. All functions used in the specification have been transformed into C functions. We have also presented ToolBus scripts for processes that can receive signals from a user interface and give orders to other tools. These orders should eventually result in calls to the C functions defined. What we still need is a main C program that takes care of the communication with the ToolBus. Incoming orders have to be transformed into the appropriate function calls and results have to be sent back to the ToolBus.

In Section 2.2.2, we learned that the communication with the ToolBus is taken care of by an interface that is generated automatically. We only have to take care of the functions that are called by the interface. We can read from the actions in a ToolBus process which functions are needed. In the process `SDA-REC` above, this is the function `expr`, called with argument `E` (because `expr(E)` is argument of the action `snd-eval`). We have, therefore, to provide a function `expr` that takes the term received from the ToolBus, anatomizes it, and calls the necessary function that performs the calculation. The function `expr` for the `sda-rec` tool looks as follows:

```
term *expr(term *trm)
{ term *t1, *t2;

  if (TBmatch(trm, "add(%t, %t)", &t1, &t2))
    return TBmake("snd-value(%t)", add2(t1, t2));
  else if (TBmatch(trm, "sub(%t, %t)", &t1, &t2))
    return TBmake("snd-value(%t)", sub2(t1, t2));
  else if (TBmatch(trm, "mult(%t, %t)", &t1, &t2))
    return TBmake("snd-value(%t)", mult2(t1, t2));
  else if (TBmatch(trm, "div(%t, %t)", &t1, &t2))
    return TBmake("snd-value(%t)", div2(t1, t2));
  else if (TBmatch(trm, "less(%t, %t)", &t1, &t2))
    return TBmake("snd-value(%t)", TBmake("%d", less2(t1, t2)));
  else return TBmake("snd-value(sda-error(%t))", trm);
}
```

In the ToolBus process `CALC-REC-BUTTON`, shown in Section 5.2, one of the expressions `add(D?, E?)`, `sub(D?, E?)`, `mult(D?, E?)`, `div(D?, E?)`, or `less(D?, E?)` was sent to the process `SDA-REC`. This expression is the argument `E` in the term `expr(E)` that the process `SDA-REC` sends to its tool. The term that the function `expr` above receives, therefore contains one of these five expressions. We use the ToolBus function `TBmatch` to see which expression has been received. If the term received contains none of the five expressions, an error message is sent back to the ToolBus.

Suppose that the term received contains the expression `add(n1, n2)`, with `n1` and `n2` terms each containing a list of digits, representing a natural. Then `TBmatch(trm, "add(%t, %t)", &t1, &t2)` returns true, and `n1` and `n2` are assigned to respectively `t1` and `t2`. Next, a term containing `snd-value(%t)` is created. At the place of `%t` a term will be substituted, which is the result of the function call `add2(t1, t2)`. This way, a term received from the ToolBus leads to a function call inside a tool. The tool interface takes care again of the sending of the term returned by `expr` to the ToolBus.

Note that the function `less2` does not return a term, but a boolean. Because the booleans have been defined as an enumeration type, of which the elements are actually integers, the resulting integer first has te be transformed to a term. This is done by the function `TBmake` again. The directive `%d` defines the contents of the term to be a digit. The resulting term is the argument of `snd-value`.

For completeness, we also show the main function of the `sda-rec` tool:

```
void main(int argc, char *argv[])
{
  TBinit("sda_rec", argc, argv, sda_rec_handler, sda_rec_check_in_sign);

  init();
  initDigitDA();

  TBeventloop();
}
```

First some initializations are being made by calling the ToolBus function `TBinit`. Next, the terms `Dtrm` and `Mtrm` are created by `init`. The digits `d[0]` to `d[9]` are created by `initDigitDA`. We explained these functions in Section 3.2. Finally, by calling the ToolBus function `TBeventloop`, the tool enters a loop in which it waits for terms from the ToolBus to be processed.

As the tool `sda-rec` requires a top function `expr`, we can read from the ToolBus process `REPRES` that the representation tool needs functions `input` and `output`. Both these functions have indeed been defined for the tool `repres`. Here, we only show the function `input`.

```
term *input(char *s)
{
  return TBmake("snd-value(%t)", read_str(s));
}
```

The string received by `input` is passed to the function `read_str` that we defined in Section 4.2. The resulting term is sent back to the ToolBus as argument of `snd-value`.

The full version of the C interface functions can be found in Appendices C.8 to C.13.

# 6 Splitting the specification into multiple tools

As a first exercise in distributing an Asf+Sdf specification over multiple ToolBus tools, we are now going to split the two tools `sda-rec` and `sda-lst` into three tools. The main tools will be called `calc-rec` and `calc-lst`, and contain the arithmetic functions. All functions operating on lists and digits will be located in a tool `digit-list`. This results in the ToolBus architecture presented in Figure 5.

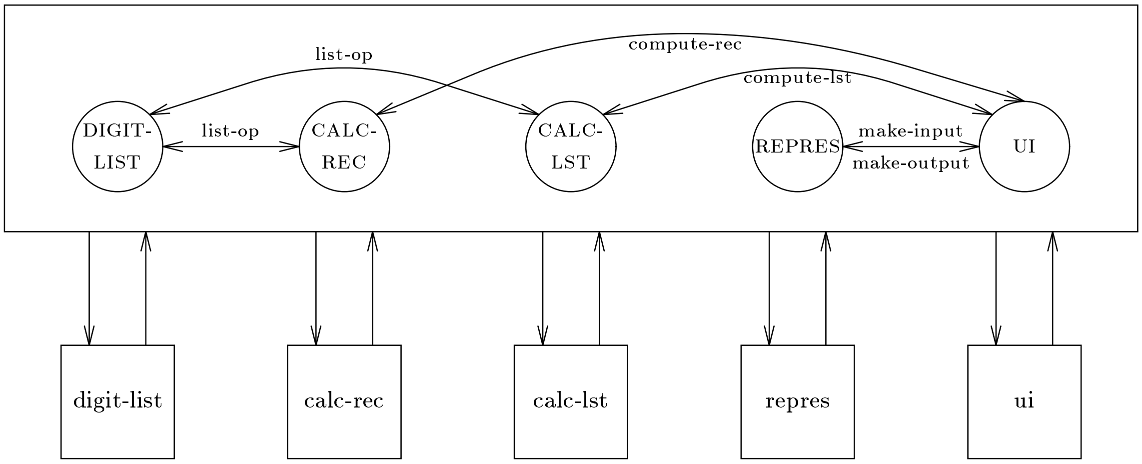


Figure 5: ToolBus architecture with the sda tools split into calculation tools and a digit-list tool.

The tools `calc-rec` and `calc-lst` receive calculation orders from the ToolBus. Whenever an operation on lists or digits is required, the tool sends an event to the ToolBus. This event is a term containing the operation needed. The ToolBus will then order the tool `digit-list` to evaluate this term. The result is sent back to the tool that requested the operation. We work towards an approach in which tools are black boxes with a number of access functions, e.g., `RightHead`. This can be compared to an object-oriented system with objects and methods. The processes inside the ToolBus know the exact names of the access functions. For instance, the tool `calc-rec` could request an operation as "left-most-element". The ToolBus interprets this request and calls the function `LeftHead` in the tool `digit-list`.

To obtain the desired behaviour, the following steps have to be taken. We will only describe the modifications needed for the tool based on recursion. An identical procedure must be followed for the tool based on list manipulation.

- The ToolBus process `SDA-REC` must be changed into a process `CALC-REC` such that it is able to receive requests for operations on lists or digits.
- A ToolBus process has to be written for a new tool `digit-list`.
- The functions that perform operations on lists or digits have to be located in the new tool.
- In the tool `calc-rec`, the functions that perform operations on lists or digits have to be replaced by functions sending a request for the desired computation to the ToolBus. These functions should return the result of this computation that is received from the ToolBus.

**Adapting `SDA-REC`.** We start with the version of `SDA-REC` that we presented in Section 5.2. Between the sending of the original evaluation command (`snd-eval`) and the receiving of the result of this computation (`rec-value`), zero or more requests (events) for operations on lists or digits can be received (`rec-event`). The adapted version of `SDA-REC` is called `CALC-REC`.

```
process CALC-REC is
  let Tid : calc-rec, E : term, K : term, L : term, V : term
  in
     execute(calc-rec, Tid?) .
     ( rec-msg(compute-rec, E?) .
       snd-eval(Tid, expr(E)) .
       ( rec-event(Tid, op(K?)) .
         snd-msg(list-op, K) .
         rec-msg(list-op, K, L?) .
         snd-ack-event(Tid, op(K, L))
       ) * rec-value(Tid, V?) .
           snd-msg(compute-rec, E, V)
     ) * delta
  endlet
```

In this process, the event `op(K?)` is received, with `K` the term to be evaluated. This term is sent to the process `DIGIT-LIST`, which monitors the tool `digit-list`. The result of the requested operation is returned in the term `L`, which is sent back to the tool `calc-rec` in the acknowledgement of the event (`snd-ack-event`).

**The process `DIGIT-LIST`.** The process `DIGIT-LIST` for the tool `digit-list` has the same structure as the process `SDA-REC` originally had in Section 5.2.

```
process DIGIT-LIST is
  let Tid : digit-list, E : term, V : term
  in
     execute(digit-list, Tid?) .
     ( rec-msg(list-op, E?) .
       snd-eval(Tid, expr(E)) .
       rec-value(Tid, V?) .
       snd-msg(list-op, E, V)
     ) * delta
  endlet
```

It receives a message with a term to be evaluated. A command is sent to the tool, and the result is sent back. The complete ToolBus script for the new configuration can be found in Appendix D.2.

**The new tool `digit-list`.** The new tool `digit-list` will contain all the functions operating on lists or digits that used to be located in the tool `sda-rec`. A C interface can be generated from the process `DIGIT-LIST` above. One can read from this process that a function `expr` is needed to analyse the terms received from the ToolBus by the tool `digit-list`. These terms can contain a call to any function defined on lists or digits. The function `expr` has the same behaviour as the function of the same name in the tool `sda-rec`, which we discussed in Section 5.3. Here we display only one statement; the entire function can be found in Appendix E.3.

```
if (TBmatch(trm, "P(%t)", &t))
  return TBmake("snd-value(%t)", P(t));
```

If the term received contains the expression `P(dt)` with `dt` a term representing a digit, the function `P` that we defined in Section 3.2 is called.

**The tool `calc-rec`.** The result of a calculation by the tool `digit-list` is an intermediate result. It is needed by the tool `calc-rec` for further computation. It is important to understand now that everything that is sent from the ToolBus to a tool is handled separately by the tool in question. The intermediate value returned can, therefore, not be transferred directly to the function that invoked the event. This has to go via a global `result` variable.

```
term *result;

void initResult()
{
   result = NULL;
   TBprotect(&result);
}
```

The sending of the intermediate result from the ToolBus to the tool `calc-rec` is done with the action `snd-ack-event` in the process `CALC-REC` above. This invokes a function `rec_ack_event` in the tool `calc-rec`.

```
void rec_ack_event(term *trm)
{ term *t1, *t2;

  if (TBmatch(trm, "op(%t,%t)", &t1, &t2))
    result = t2;
  else TBprintf(stderr, "calc-error: %t\n", trm);
}
```

This function expects a term of the form `op(t1, t2)`, with `t1` and `t2` two terms. This corresponds with the argument of `snd-ack-event` in the process `CALC-REC`. The second term should be the result of the evaluation of the first term by `digit-list`. This second term is then assigned to the variable `result`.

We can now demonstrate how the sending of events to the ToolBus is invoked. The tool `calc-rec` initially receives a calculation command. At some point in the calculation, a function that works on lists or digits is called, say `P`, the predecessor function for digits. The original definition of `P` has moved to the `digit-list` tool. It is replaced by the following definition:

```
term *P(term *dt)
{
  TBsend(TBmake("snd-event(op(P(%t)))", dt));
  TBreceive();
  return result;
}
```

We see that a term `snd-event(op(P(dt)))` is constructed and sent to the ToolBus. There, the tool `digit-list` is ordered to calculate the term `P(dt)`. The ToolBus function `TBreceive` must be used to take in terms from the ToolBus. Normally, this is done automatically by the main function `TBeventloop`, but since we are in the middle of the execution of a function (`P`), we have to do this ourselves. By calling TBreceive, the term containing the result is handled by rec_ack_event. We just showed that this comes down to storing the result in the variable result, which is returned subsequently.

Perhaps a nasty side effect of the use of TBreceive is the garbage collection this function invokes. This means that we have to protect all variables that are in use at the moment that TBreceive is called, including function parameters. This has, for example, resulted in the following adaptation of the recursive variant of the function add2, which we presented earlier in Section 3.4.

```
term *add2(term *n1, term *n2)
{ term *t;

  TBprotect(&n1);
  TBprotect(&n2);
  t = cond_trm(rlz(n1), lZ(n2), lS(add2(n1, lP(n2))));
  TBunprotect(&n1);
  TBunprotect(&n2);
  return t;
}
```

During the calculation, parameters n1 and n2 are protected. The result is temporarily assigned to the variable t.

In Appendix E, all adaptations of the C program for the new ToolBus configuration can be found.

# 7 Conclusions and future work

We have demonstrated how an Asf+Sdf specification can be transformed to a C program that can serve as a ToolBus tool. This process is simplified considerably by making use of the built-in type term in combination with the term manipulation functions. When an Asf+Sdf function is defined by only one equation, an equivalent in C can be written down immediately, since no decisions based on the input are made. Functions with more than one equation use pattern matching on the input to decide which equation is taken. This is a bit more complicated, but it can always be mirrored by using the TBmatch function from the ToolBus library.

The part of the C program that takes care of the communication with the ToolBus can be generated automatically. We only have to provide the functions that are called from the ToolBus. These functions analyse their input terms and subsequently pass the argument(s) to one of the functions defined earlier.

We have experienced that it is not trivial to divide an Asf+Sdf specification over different tools. To define tools that have to use other tools for parts of their computation has appeared to be on the edge of what is possible in the ToolBus. We had to interfere in the communication protocol between ToolBus and tools, by calling the function TBreceive, and we had to use global variables to obtain the desired behaviour. A side effect of this approach is that we had to protect all function parameters, because TBreceive activates garbage collection. One can understand that this is not the way the ToolBus was meant to be used. The question is whether it is desirable to use it this way. If so, some work has to be done on the handling of the terms containing intermediate results. One possibility is to create a built-in function which can be used to send a term to the ToolBus for evaluation, and which returns the result of this evaluation. This would make the use of TBreceive and global variables unnecessary. Such a function should also take care of the protection of variables within the current scope.

**Future work.** Now that we have split the original Asf+Sdf specification from two tools into three tools, the next step is of course to divide it over more tools. This will not alter the method of sending terms for evaluation to the ToolBus, and receiving the result, as defined in the previous section.

What will change is how terms are directed within the ToolBus. In case of two tools, a term received from one tool can immediately be directed to the other tool. When working with more tools, it will have to be decided to which tool the term is passed. The tool that sent the term to be evaluated can not have this information. Only within the ToolBus it should be known which tool can perform which operations. In that case, it would be best to define a separate ToolBus process for directing evaluation requests to the appropriate tools. The would result in the following version of the process `CALC-REC`:

```
process CALC-REC is
  let Tid : calc-rec, E : term, K : term, L : term, V : term
  in
     execute(calc-rec, Tid?) .
     ( rec-msg(compute-rec, E?) .
       snd-eval(Tid, expr(E)) .
       ( rec-event(Tid, op(K?)) .
         snd-msg(inter-op, K) .
         rec-msg(inter-op, K, L?) .
         snd-ack-event(Tid, op(K, L))
       ) * rec-value(Tid, V?) .
           snd-msg(compute-rec, E, V)
     ) * delta
  endlet
```

The only difference with the version presented in Section 6 is that not the message `list-op` is sent to the process `DIGIT-LIST`, but the message `inter-op` to some process `ASSIGN`. The other processes monitoring a tool will be defined likewise. So every tool process sends its intermediate-evaluation requests to the process `ASSIGN`. This process would then contain the following sequence of actions for every function defined in one of the tools:

```
rec-msg(inter-op, Lappend(D?,E?)) .
snd-msg(list-op, Lappend(D,E)) .
rec-msg(list-op, Lappend(D,E), V?) .
snd-msg(inter-op, Lappend(D,E), V)
```

We see that the process `ASSIGN` receives the `inter-op` message, in this case containing an application of the function `Lappend`. Now this process has the knowledge that the function `Lappend` has been defined in the `digit-list` tool, so it sends the message `list-op` to the process `DIGIT-LIST`. The result is sent back to the process that sent the original `inter-op` message.

A problem with the implementation of the method described above is that in ToolBus processes, function names should not start with capitals or contain underscores. Functions as `Lappend` would therefore have to be renamed. Of course, this would only be a small effort.

# References

[BBB+85] F.L. Bauer, R. Berghammer, M. Broy, W. Dosch, F. Geiselbrechtinger, R. Gnatz, E. Hangel, W. Hesse, B. Krieg-Brückner, A. Laut, T. Matzner, B. Möller, F. Nickl, H. Partsch, P. Pepper, K. Samelson, M. Wirsing, and H. Wössner. *The Munich Project CIP. Volume I: The Wide Spectrum Language CIP-L*, volume 183 of *Lecture Notes in Computer Science*. Springer-Verlag, 1985.

[BK84] J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information and Computation*, 60(1/3):109–137, 1984.

[BK96a] J. A. Bergstra and P. Klint. The discrete time toolbus. In M. Wirsing and M. Nivat, editors, *Algebraic Methodology and Software Technology (AMAST '96)*, volume 1101 of *LNCS*, pages 288–305. Springer-Verlag, 1996.

[BK96b] J.A. Bergstra and P. Klint. The toolbus coordination architecture. In P. Ciancarini and C. Hankin, editors, *Coordination Languages and Models*, volume 1061 of *LNCS*, pages 75–88. Springer-Verlag, 1996.

[BO95] F. Bonsu and R. Oudejans. Graphic generation language: automatic code generation from design. Master thesis, Programming Research Group, University of Amsterdam, 1995. available via annonymous ftp from `ftp.wins.uva.nl`, directory `pub/programming-research/MasterTheses`.

[BS96] J.A. Bergstra and M.P.A. Sellink. Sequential data algebra primitives. Report P9602b, Programming Research Group, University of Amsterdam, 1996.

[BV96] M.G.J. van den Brand and E. Visser. Generation of formatters for context-free languages. *ACM Transactions on Software Engineering and Methodology*, 5(1):1–41, 1996.

[BW90] J.C.M. Baeten and W.P. Weijland. *Process Algebra*. Cambridge Tracts in Theoretical Computer Science 18. Cambridge University Press, 1990.

[FJO+86] L.G.M. Feys, H.B.M. Jonkers, J.H. Obbink, C.P.J. Koymans, G.R. Renardel de Lavalette, and P.H. Rodenburg. A survey of the design language COLD. Technical Report METEOR/t7/PRLE/3, METEOR, 1986.

[HK95] J.A. Hillebrand and H.P. Korver. A well-formedness checker for $\mu$CRL. Report P9501, Programming Research Group, University of Amsterdam, 1995.

[Jon93] H.B.M. Jonkers. An overview of the SPRINT method. In J.C.P. Woodcock and P.G. Larsen, editors, *FME'93: Industrial-Strength Formal Methods*, volume 670 of *Lecture Notes in Computer Science*, pages 403–427. Springer-Verlag, 1993.

[Kli93] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2(2):176–201, 1993.

[Kli96] P. Klint. A guide to ToolBus programming. Technical report, 1996. Provided with the ToolBus distribution, available via annonymous ftp from `ftp.cwi.nl`, directory `pub/gipe/ToolBus`.

[KR78] B.W. Kernighan and D.M. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, N.J., 1978.

[KW93] J.F.T. Kamperman and H.R. Walters. ARM, abstract rewriting machine. Technical Report CS-9330, CWI, Amsterdam, Netherlands, 1993.

[Ous94] J.K. Ousterhout. *Tcl and the Tk Toolkit*. Professional Computing Series. Addison-Wesley, Reading, Massachusetts, 1994.

# A The SDA specification of the natural numbers

This appendix contains the complete specification from [BS96].

## A.1 Bool1DA

**imports** Layout
**exports**
  **sorts** T4
  **context-free syntax**
    "D" $\rightarrow$ T4
    "T" $\rightarrow$ T4
    "F" $\rightarrow$ T4
    "M" $\rightarrow$ T4
    "eq1" "(" T4 "," T4 ")" $\rightarrow$ T4
    "(" T4 "$\lhd$" T4 "$\rhd$" T4 ")" $\rightarrow$ T4
    "(" T4 ")" $\rightarrow$ T4 {**bracket**}
  **variables**
    *b[0-9′]∗* $\rightarrow$ T4
**equations**

| | |
|---|---|
| $(b_1 \lhd \mathsf{D} \rhd b_2) = \mathsf{D}$ | [condT1] |
| $(b_1 \lhd \mathsf{T} \rhd b_2) = b_1$ | [condT2] |
| $(b_1 \lhd \mathsf{F} \rhd b_2) = b_2$ | [condT3] |
| $(b_1 \lhd \mathsf{M} \rhd b_2) = \mathsf{M}$ | [condT4] |

| | |
|---|---|
| $\mathsf{eq1}(b_1, b_2) = ((\mathsf{T} \lhd b_2 \rhd \mathsf{F}) \lhd b_1 \rhd (\mathsf{F} \lhd b_2 \rhd \mathsf{T}))$ | [eqT1] |

## A.2 Bool2DA

**imports** Bool1DA$^{A.1}$
**exports**
  **sorts** T4-OP
  **context-free syntax**
    "$\neg$" T4 $\rightarrow$ T4

    T4 "${}^{\circ}\!\vee$" T4 $\rightarrow$ T4 {**assoc**}
    T4 "$\vee^{\circ}$" T4 $\rightarrow$ T4 {**assoc**}
    T4 "${}_{\circ}\!\wedge$" T4 $\rightarrow$ T4 {**assoc**}
    T4 "$\wedge_{\circ}$" T4 $\rightarrow$ T4 {**assoc**}

    "T" "(" T4 ")" $\rightarrow$ T4
  **priorities**
    "$\neg$"T4 $\rightarrow$ T4 $>$ {**non-assoc**: T4 "${}^{\circ}\!\vee$"T4 $\rightarrow$ T4, T4 "${}_{\circ}\!\wedge$"T4 $\rightarrow$ T4, T4 "$\vee^{\circ}$"T4 $\rightarrow$ T4,
    T4 "$\wedge_{\circ}$"T4 $\rightarrow$ T4}
**equations**

| | |
|---|---|
| $\neg\, b = (\mathsf{F} \lhd b \rhd \mathsf{T})$ | [negT1] |

$$b_1 \mathbin{{}_{\circ}\!\wedge} b_2 = (b_2 \lhd b_1 \rhd \mathsf{F}) \qquad \text{[lfcT1]}$$
$$b_1 \mathbin{{}^{\circ}\!\vee} b_2 = (\mathsf{T} \lhd b_1 \rhd b_2) \qquad \text{[lfdT1]}$$
$$b_1 \mathbin{\wedge\!{}_{\circ}} b_2 = b_2 \mathbin{{}_{\circ}\!\wedge} b_1 \qquad \text{[rfcT1]}$$
$$b_1 \mathbin{\vee\!{}^{\circ}} b_2 = b_2 \mathbin{{}^{\circ}\!\vee} b_1 \qquad \text{[rfdT1]}$$

$$\mathsf{T}(b) = \mathsf{eq1}(b, b) \qquad \text{[trueT1]}$$

## A.3 PrepDigit

**imports** Bool2DA$^{A.2}$
**exports**
  **sorts** D4
  **context-free syntax**
    "D" $\rightarrow$ D4
    "M" $\rightarrow$ D4
    "(" D4 "$\lhd$" T4 "$\rhd$" D4 ")" $\rightarrow$ D4
    "eq1" "(" D4 "," D4 ")" $\rightarrow$ T4
    "T" "(" D4 ")" $\rightarrow$ T4
    "(" D4 ")" $\rightarrow$ D4 {**bracket**}
  **variables**
    *d[0-9']∗* $\rightarrow$ D4
**equations**

$$(d_1 \lhd \mathsf{D} \rhd d_2) = \mathsf{D} \qquad \text{[condD1]}$$
$$(d_1 \lhd \mathsf{T} \rhd d_2) = d_1 \qquad \text{[condD2]}$$
$$(d_1 \lhd \mathsf{F} \rhd d_2) = d_2 \qquad \text{[condD3]}$$
$$(d_1 \lhd \mathsf{M} \rhd d_2) = \mathsf{M} \qquad \text{[condD4]}$$

## A.4 Digit1DA

**imports** PrepDigit$^{A.3}$
**exports**
  **lexical syntax**
    [0-9] $\rightarrow$ D4
  **context-free syntax**
    "base" $\rightarrow$ D4
    "S" "(" D4 ")" $\rightarrow$ D4
    "P" "(" D4 ")" $\rightarrow$ D4
    "Z" "(" D4 ")" $\rightarrow$ T4
  **variables**
    *d[0-9']∗* $\rightarrow$ D4
**equations**

$$\mathsf{base} = 9 \qquad \text{[base1]}$$

$$\mathsf{S}(0) = 1 \quad [\mathsf{suc0}]$$
$$\mathsf{S}(1) = 2 \quad [\mathsf{suc1}]$$
$$\mathsf{S}(2) = 3 \quad [\mathsf{suc2}]$$
$$\mathsf{S}(3) = 4 \quad [\mathsf{suc3}]$$
$$\mathsf{S}(4) = 5 \quad [\mathsf{suc4}]$$
$$\mathsf{S}(5) = 6 \quad [\mathsf{suc5}]$$
$$\mathsf{S}(6) = 7 \quad [\mathsf{suc6}]$$
$$\mathsf{S}(7) = 8 \quad [\mathsf{suc7}]$$
$$\mathsf{S}(8) = 9 \quad [\mathsf{suc8}]$$
$$\mathsf{S}(9) = \mathsf{M} \quad [\mathsf{suc9}]$$
$$\mathsf{S}(\mathsf{M}) = \mathsf{M} \quad [\mathsf{sucM}]$$
$$\mathsf{S}(\mathsf{D}) = \mathsf{D} \quad [\mathsf{sucD}]$$

$$\mathsf{P}(0) = \mathsf{M} \quad [\mathsf{pred0}]$$
$$\mathsf{P}(1) = 0 \quad [\mathsf{pred1}]$$
$$\mathsf{P}(2) = 1 \quad [\mathsf{pred2}]$$
$$\mathsf{P}(3) = 2 \quad [\mathsf{pred3}]$$
$$\mathsf{P}(4) = 3 \quad [\mathsf{pred4}]$$
$$\mathsf{P}(5) = 4 \quad [\mathsf{pred5}]$$
$$\mathsf{P}(6) = 5 \quad [\mathsf{pred6}]$$
$$\mathsf{P}(7) = 6 \quad [\mathsf{pred7}]$$
$$\mathsf{P}(8) = 7 \quad [\mathsf{pred8}]$$
$$\mathsf{P}(9) = 8 \quad [\mathsf{pred9}]$$
$$\mathsf{P}(\mathsf{M}) = \mathsf{M} \quad [\mathsf{predM}]$$
$$\mathsf{P}(\mathsf{D}) = \mathsf{D} \quad [\mathsf{predD}]$$

$$\mathsf{Z}(0) = \mathsf{T} \quad [\mathsf{zero0}]$$
$$\mathsf{Z}(1) = \mathsf{F} \quad [\mathsf{zero1}]$$
$$\mathsf{Z}(2) = \mathsf{F} \quad [\mathsf{zero2}]$$
$$\mathsf{Z}(3) = \mathsf{F} \quad [\mathsf{zero3}]$$
$$\mathsf{Z}(4) = \mathsf{F} \quad [\mathsf{zero4}]$$
$$\mathsf{Z}(5) = \mathsf{F} \quad [\mathsf{zero5}]$$
$$\mathsf{Z}(6) = \mathsf{F} \quad [\mathsf{zero6}]$$
$$\mathsf{Z}(7) = \mathsf{F} \quad [\mathsf{zero7}]$$
$$\mathsf{Z}(8) = \mathsf{F} \quad [\mathsf{zero8}]$$
$$\mathsf{Z}(9) = \mathsf{F} \quad [\mathsf{zero9}]$$
$$\mathsf{Z}(\mathsf{M}) = \mathsf{M} \quad [\mathsf{zeroM}]$$
$$\mathsf{Z}(\mathsf{D}) = \mathsf{D} \quad [\mathsf{zeroD}]$$

$$\mathsf{T}(d) = \mathsf{Z}(d) \overset{\circ}{\vee} \mathsf{T} \quad [\mathsf{trueD1}]$$

$$\mathsf{eq1}(d_1, d_2) = (\mathsf{Z}(d_2) \lhd \mathsf{Z}(d_1) \rhd \neg\, \mathsf{Z}(d_2) \underset{\circ}{\wedge} \mathsf{eq1}(\mathsf{P}(d_1), \mathsf{P}(d_2))) \quad [\mathsf{eqD1}]$$

## A.5 Digit2DA

**imports** Digit1DA$^{A.4}$

**exports**

**context-free syntax**

"carry1" "(" D4 "," D4 ")" $\rightarrow$ D4
"borrow2" "(" D4 "," D4 ")" $\rightarrow$ D4
"compl" "(" D4 ")" $\rightarrow$ D4
"modsum1" "(" D4 "," D4 ")" $\rightarrow$ D4
"modsub2" "(" D4 "," D4 ")" $\rightarrow$ D4
D4 "$<_\circ$" D4 $\rightarrow$ T4
"t2d" "(" T4 ")" $\rightarrow$ D4

**equations**

$$\mathsf{t2d}(b) = (1 \lhd b \rhd 0) \qquad \text{[T2D]}$$

$$\mathsf{carry1}(d_1, d_2) = (0 \lhd \mathsf{Z}(d_1) \rhd (1 \lhd \mathsf{eq1}(d_2, \mathsf{base}) \rhd \mathsf{carry1}(\mathsf{P}(d_1), \mathsf{S}(d_2)))) \qquad \text{[Carry]}$$

$$\mathsf{borrow2}(d_1, d_2) = \mathsf{t2d}(d_1 <_\circ d_2) \qquad \text{[Borrow]}$$

$$\mathsf{compl}(d_1) = (\mathsf{base} \lhd \mathsf{eq1}(d_1, 1) \rhd \mathsf{P}(\mathsf{compl}(\mathsf{P}(d_1)))) \qquad \text{[Compl]}$$

$$\begin{aligned}\mathsf{modsum1}(d_1, d_2) \ = \ & (d_2 \lhd \mathsf{Z}(d_1) \\ & \rhd (\mathsf{P}(d_1) \lhd \mathsf{eq1}(d_2, \mathsf{base}) \rhd \mathsf{modsum1}(\mathsf{P}(d_1), \mathsf{S}(d_2))))\end{aligned} \qquad \text{[Modsum]}$$

$$\begin{aligned}\mathsf{modsub2}(d_1, d_2) \ = \ & (d_1 \lhd \mathsf{Z}(d_2) \\ & \rhd (\mathsf{compl}(d_2) \lhd \mathsf{Z}(d_1) \rhd \mathsf{modsub2}(\mathsf{P}(d_1), \mathsf{P}(d_2))))\end{aligned} \qquad \text{[Modsub]}$$

$$d_1 <_\circ d_2 = (\mathsf{F} \lhd \mathsf{Z}(d_2) \rhd (\mathsf{T} \lhd \mathsf{Z}(d_1) \rhd \mathsf{P}(d_1) <_\circ \mathsf{P}(d_2))) \qquad \text{[LessD]}$$

## A.6 PrepLDigit

**imports** Bool2DA$^{A.2}$

**exports**

**sorts** LD4

**context-free syntax**

"D" $\rightarrow$ LD4
"M" $\rightarrow$ LD4
"(" LD4 "$\lhd$" T4 "$\rhd$" LD4 ")" $\rightarrow$ LD4
"eq1" "(" LD4 "," LD4 ")" $\rightarrow$ T4
"T" "(" LD4 ")" $\rightarrow$ T4
"(" LD4 ")" $\rightarrow$ LD4 {**bracket**}

**variables**

*Y* [*0-9'*]$*$ $\rightarrow$ LD4

**equations**

$$(Y_1 \lhd \mathsf{D} \rhd Y_2) = \mathsf{D} \qquad \text{[condLD1]}$$
$$(Y_1 \lhd \mathsf{T} \rhd Y_2) = Y_1 \qquad \text{[condLD2]}$$
$$(Y_1 \lhd \mathsf{F} \rhd Y_2) = Y_2 \qquad \text{[condLD3]}$$
$$(Y_1 \lhd \mathsf{M} \rhd Y_2) = \mathsf{M} \qquad \text{[condLD4]}$$

## A.7 L1DigitDA

**imports** PrepDigit$^{A.3}$ PrepLDigit$^{A.6}$
**exports**
**context-free syntax**

$$
\begin{array}{lll}
\text{“[ ]”} & \to \mathsf{LD4} & \\
\text{“[” D4 “]”} & \to \mathsf{LD4} & \\
\mathsf{LD4} \text{ “}{\circ\!\frown}\text{” } \mathsf{LD4} & \to \mathsf{LD4} & \{\textbf{non-assoc}\} \\
\text{“}\Lsh\text{” } \mathsf{LD4} & \to \mathsf{D4} & \\
\text{“}\Rsh\text{” } \mathsf{LD4} & \to \mathsf{LD4} & \\
\text{“E” “(” LD4 “)”} & \to \mathsf{T4} &
\end{array}
$$

**priorities**

$\{$**non-assoc**: “$\Lsh$”, “$\Rsh$”$\}$ > “${\circ\!\frown}$”

**equations**

$$
\begin{array}{lll}
[\mathsf{D}] & = \mathsf{D} & [\mathsf{lscL1}] \\
[\mathsf{M}] & = \mathsf{M} & [\mathsf{lscL2}] \\
[\,]\ {\circ\!\frown}\ Y & = Y & [\mathsf{lscL3}] \\
Y\ {\circ\!\frown}\ [\,] & = Y & [\mathsf{lscL4}] \\
\mathsf{M}\ {\circ\!\frown}\ Y & = \mathsf{M} & [\mathsf{lscL5}] \\
\mathsf{D}\ {\circ\!\frown}\ Y & = \mathsf{D} & [\mathsf{lscL6}] \\
[d]\ {\circ\!\frown}\ \mathsf{D} & = (\mathsf{D} \lhd \mathsf{T}(d) \rhd \mathsf{D}) & [\mathsf{lscL7}] \\
[d]\ {\circ\!\frown}\ \mathsf{M} & = (\mathsf{M} \lhd \mathsf{T}(d) \rhd \mathsf{M}) & [\mathsf{lscL8}] \\
(Y_1\ {\circ\!\frown}\ Y_2)\ {\circ\!\frown}\ Y_3 & = Y_1\ {\circ\!\frown}\ (Y_2\ {\circ\!\frown}\ Y_3) & [\mathsf{lscL9}]
\end{array}
$$

$$
\begin{array}{lll}
\mathsf{T}(\mathsf{D}) & = \mathsf{D} & [\mathsf{trueL1}] \\
\mathsf{T}(\mathsf{M}) & = \mathsf{M} & [\mathsf{trueL2}] \\
\mathsf{T}([\,]) & = \mathsf{T} & [\mathsf{trueL3}] \\
\mathsf{T}([d]) & = \mathsf{T}(d) & [\mathsf{trueL4}] \\
\mathsf{T}(Y_1\ {\circ\!\frown}\ Y_2) & = \mathsf{T}(Y_1) \mathbin{{}_{\circ}\!\wedge} \mathsf{T}(Y_2) & [\mathsf{trueL5}]
\end{array}
$$

$$
\begin{array}{lll}
\mathsf{E}(\mathsf{D}) & = \mathsf{D} & [\mathsf{empL1}] \\
\mathsf{E}(\mathsf{M}) & = \mathsf{M} & [\mathsf{empL2}] \\
\mathsf{E}([\,]) & = \mathsf{T} & [\mathsf{empL3}] \\
\mathsf{E}([d]) & = \neg\ \mathsf{T}(d) & [\mathsf{empL4}] \\
\mathsf{E}([d]\ {\circ\!\frown}\ Y) & = \neg\ (\mathsf{T}(d) \mathbin{{}_{\circ}\!\wedge} \mathsf{T}(Y)) & [\mathsf{empL5}]
\end{array}
$$

$$
\begin{array}{lll}
\Lsh\ \mathsf{D} & = \mathsf{D} & [\mathsf{lhdL1}] \\
\Lsh\ \mathsf{M} & = \mathsf{M} & [\mathsf{lhdL3}] \\
\Lsh\ [\,] & = \mathsf{M} & [\mathsf{lhdL3}] \\
\Lsh\ [d] & = d & [\mathsf{lhdL4}] \\
\Lsh\ ([d]\ {\circ\!\frown}\ Y) & = (d \lhd \mathsf{T}(d) \mathbin{{}_{\circ}\!\wedge} \mathsf{T}(Y) \rhd \mathsf{M}) & [\mathsf{lhdL5}]
\end{array}
$$

$$
\begin{array}{lll}
\Rsh\ \mathsf{D} & = \mathsf{D} & [\mathsf{lrtL1}] \\
\Rsh\ \mathsf{M} & = \mathsf{M} & [\mathsf{lrtL2}]
\end{array}
$$

$$\downarrow\!\!\!\!\rightarrow [\,] = \mathsf{M} \qquad [\mathsf{lrtL3}]$$
$$\downarrow\!\!\!\!\rightarrow [d] = ([\,] \lhd \mathsf{T}(d) \rhd \mathsf{M}) \qquad [\mathsf{lrtL4}]$$
$$\downarrow\!\!\!\!\rightarrow ([d] \circ\!\!\frown Y) = (Y \lhd \mathsf{T}(d) \rhd \mathsf{M}) \qquad [\mathsf{lrtL5}]$$

$$\mathsf{eq1}(Y_1, Y_2) = (\mathsf{E}(Y_1) \wedge \mathsf{E}(Y_2)) \vee (\neg\, \mathsf{E}(Y_1) \wedge \neg\, \mathsf{E}(Y_2) \wedge \mathsf{eq1}(\urcorner Y_1, \urcorner Y_2) \wedge \mathsf{eq1}(\downarrow\!\!\!\!\rightarrow Y_1, \downarrow\!\!\!\!\rightarrow Y_2)) \qquad [\mathsf{eqLN1}]$$

## A.8 L2DigitDA

**imports** L1DigitDA[A.7]
**exports**
**context-free syntax**
"inv" "(" LD4 ")" → LD4
LD4 "↱" → D4
LD4 "↲" → LD4
LD4 "⌢∘" LD4 → LD4
**equations**

$$\mathsf{inv}(Y) = ([\,] \lhd \mathsf{E}(Y) \rhd \mathsf{inv}(\downarrow\!\!\!\!\rightarrow Y) \circ\!\!\frown [\urcorner Y]) \qquad [\mathsf{invL1}]$$

$$Y \uparrow\!\!\!\!\rightarrow = \urcorner\, \mathsf{inv}(Y) \qquad [\mathsf{rhdL1}]$$
$$Y \leftarrow\!\!\!\!\downarrow = \mathsf{inv}(\downarrow\!\!\!\!\rightarrow \mathsf{inv}(Y)) \qquad [\mathsf{rltL1}]$$
$$Y_1 \frown\!\!\circ\, Y_2 = \mathsf{inv}(\mathsf{inv}(Y_2) \circ\!\!\frown \mathsf{inv}(Y_1)) \qquad [\mathsf{rfcL1}]$$

## A.9 Nat1DA

**imports** Digit2DA[A.5] L2DigitDA[A.8]
**exports**
**context-free syntax**
"rlz" "(" LD4 ")" → LD4
"S" "(" LD4 ")" → LD4
"P" "(" LD4 ")" → LD4
"Z" "(" LD4 ")" → T4
"length" "(" LD4 ")" → LD4
**variables**
*n[0-9′]*∗ → LD4
**equations**

$$\mathsf{rlz}(n) = (n \lhd \mathsf{E}(n) \vee \neg\, \mathsf{Z}(\urcorner n) \rhd \mathsf{rlz}(\downarrow\!\!\!\!\rightarrow n)) \qquad [\mathsf{RLZ1}]$$

$$\mathsf{S}(n) = \mathsf{rlz}(([1] \lhd \mathsf{E}(n) \rhd (\mathsf{S}(n \leftarrow\!\!\!\!\downarrow) \circ\!\!\frown [0] \lhd \mathsf{eq1}(n \uparrow\!\!\!\!\rightarrow, \mathsf{base}) \rhd n \leftarrow\!\!\!\!\downarrow \circ\!\!\frown [\mathsf{S}(n \uparrow\!\!\!\!\rightarrow)]))) \qquad [\mathsf{SucL}]$$

$$\mathsf{P}(n) = \mathsf{rlz}((\mathsf{P}(n \leftarrow\!\!\!\!\downarrow) \circ\!\!\frown [\mathsf{base}] \lhd \mathsf{eq1}(n \uparrow\!\!\!\!\rightarrow, 0) \rhd n \leftarrow\!\!\!\!\downarrow \circ\!\!\frown [\mathsf{P}(n \uparrow\!\!\!\!\rightarrow)])) \qquad [\mathsf{PredL}]$$

$$\mathsf{Z}(n) = \mathsf{E}(\mathsf{rlz}(n)) \qquad \text{[ZL]}$$

$$\mathsf{length}(n) = ([\ ] \lhd \mathsf{E}(n) \rhd \mathsf{S}(\mathsf{length}(\llcorner\, n))) \qquad \text{[Len1]}$$

## A.10 Nat2DA

**imports** Nat1DA$^{A.9}$
**exports**
  **context-free syntax**
    LD4 "$+\!\circ$" LD4 → LD4 {**left**}
    LD4 "$-\!\circ$" LD4 → LD4 {**left**}
    LD4 "$\times_\circ$" LD4 → LD4 {**left**}
    LD4 "$\div\!\circ$" LD4 → LD4 {**left**}
    LD4 "$<_\circ$" LD4 → T4
  **priorities**
    "$\circ\frown$" > {**non-assoc**: "$\times_\circ$", "$\div\!\circ$"} > {**non-assoc**: "$+\!\circ$", "$-\!\circ$"}

**equations**

$$n_1 \ +\!\circ\ n_2 = (\mathsf{rlz}(n_1) \lhd \mathsf{Z}(n_2) \rhd \mathsf{S}(n_1 \ +\!\circ\ \mathsf{P}(n_2))) \qquad \text{[AddLsp]}$$

$$n_1 \ -\!\circ\ n_2 = (\mathsf{rlz}(n_1) \lhd \mathsf{Z}(n_2) \rhd \mathsf{P}(n_1 \ -\!\circ\ \mathsf{P}(n_2))) \qquad \text{[SubLsp]}$$

$$n_1 \times_\circ n_2 = ([\ ] \lhd \mathsf{Z}(n_2) \rhd n_1 \ +\!\circ\ n_1 \times_\circ \mathsf{P}(n_2)) \qquad \text{[MultLsp]}$$

$$n_1 \ \div\!\circ\ n_2 = (\mathsf{M} \lhd \mathsf{Z}(n_2) \rhd ([\ ] \lhd n_1 <_\circ n_2 \rhd \mathsf{S}((n_1 \ -\!\circ\ n_2) \ \div\!\circ\ n_2))) \qquad \text{[DivLsp]}$$

$$n_1 <_\circ n_2 = (\mathsf{F} \lhd \mathsf{Z}(n_2) \rhd (\mathsf{T} \lhd \mathsf{Z}(n_1) \rhd \mathsf{P}(n_1) <_\circ \mathsf{P}(n_2))) \qquad \text{[LessLsp]}$$

## A.11 Nat3DA

**imports** Nat1DA$^{A.9}$
**exports**
  **context-free syntax**
    LD4 "$+\!\circ$" LD4 → LD4 {**left**}
    LD4 "$-\!\circ$" LD4 → LD4 {**left**}
    LD4 "$\times_\circ$" LD4 → LD4 {**left**}
    "mult2" "(" LD4 "," D4 ")" → LD4
    LD4 "$\div\!\circ$" LD4 → LD4 {**left**}
    "div2" "(" LD4 "," LD4 ")" → D4
    LD4 "mod" LD4 → LD4
    "fill2" "(" LD4 "," LD4 ")" → LD4
    "dr2" "(" LD4 "," LD4 "," LD4 ")" → LD4

“less2” “(” LD4 “,” LD4 “)” $\rightarrow$ T4
LD4 “$<_\circ$” LD4 $\rightarrow$ T4

**priorities**

{**non-assoc**: “$\Lsh$”, “$\Rsh$”, “↳”, “↲”} > “$\circ\!\frown$” > {**non-assoc**: “$\times_\circ$”, “$\div\!\circ$”} > {**non-assoc**: “$+\!\circ$”, “$\multimap$”}

**equations**

$$\frac{d_1 = \mathsf{carry1}(n_1 \Rsh,\ n_2 \Rsh),\quad d_2 = \mathsf{modsum1}(n_1 \Rsh,\ n_2 \Rsh)}{\begin{array}{l} n_1 +\!\circ\ n_2 = (\mathsf{rlz}(n_1) \lhd \mathsf{Z}(n_2) \\ \qquad \rhd (\mathsf{rlz}(n_2) \lhd \mathsf{Z}(n_1) \rhd (n_1 \text{↲} +\!\circ\ n_2 \text{↲} +\!\circ [d_1]) \circ\!\frown [d_2])) \end{array}} \quad [\mathsf{Add1}]$$

$$\frac{d_1 = \mathsf{borrow2}(n_1 \Rsh,\ n_2 \Rsh),\quad d_2 = \mathsf{modsub2}(n_1 \Rsh,\ n_2 \Rsh)}{n_1 \multimap n_2 = (\mathsf{rlz}(n_1) \lhd \mathsf{Z}(n_2) \rhd (\mathsf{M} \lhd \mathsf{Z}(n_1) \rhd (n_1 \text{↲} \multimap n_2 \text{↲} \multimap [d_1]) \circ\!\frown [d_2]))} \quad [\mathsf{Sub1}]$$

$$\mathsf{mult2}(n, d) = ([\ ] \lhd \mathsf{Z}(d) \rhd n +\!\circ\ \mathsf{mult2}(n, \mathsf{P}(d))) \quad [\mathsf{DMult1}]$$

$$\begin{array}{l} n_1 \times_\circ n_2 = ([\ ] \lhd \mathsf{Z}(n_1) \vee^{\circ} \mathsf{Z}(n_2) \\ \qquad \rhd (n_1 \times_\circ n_2 \text{↲}) \circ\!\frown [0] +\!\circ\ \mathsf{mult2}(n_1, n_2 \Rsh)) \end{array} \quad [\mathsf{Mult1}]$$

$$\mathsf{fill2}(n_1, n_2) = (\mathsf{M} \lhd \mathsf{Z}(n_2) \rhd (\mathsf{rlz}(n_2) \lhd n_1 <_\circ n_2 \rhd \mathsf{fill2}(n_1, n_2 \circ\!\frown [0]))) \quad [\mathsf{Fill1}]$$

$$\mathsf{div2}(n_1, n_2) = (\mathsf{M} \lhd \mathsf{Z}(n_2) \rhd (0 \lhd n_1 <_\circ n_2 \rhd \mathsf{S}(\mathsf{div2}(n_1 \multimap n_2, n_2)))) \quad [\mathsf{DDiv1}]$$

$$\frac{d = \mathsf{div2}(n_1, n_3)}{\mathsf{dr2}(n_1, n_2, n_3) = ([\ ] \lhd n_3 <_\circ n_2 \rhd [d] \circ\!\frown \mathsf{dr2}(n_1 \multimap \mathsf{mult2}(n_3, d), n_2, n_3 \text{↲}))} \quad [\mathsf{DR1}]$$

$$n_1 \div\!\circ\ n_2 = \mathsf{dr2}(n_1, n_2, \mathsf{fill2}(n_1, n_2) \text{↲}) \quad [\mathsf{Div1}]$$

$$\mathsf{less2}(n_1, n_2) = (\mathsf{F} \lhd \mathsf{E}(n_2) \rhd (\mathsf{less2}(\text{↳} n_1, \text{↳} n_2) \lhd \mathsf{eq1}(\Lsh n_2, \Lsh n_1) \rhd \Lsh n_1 <_\circ \Lsh n_2)) \quad [\mathsf{Less1}]$$

$$\frac{n_1' = \mathsf{rlz}(n_1),\quad n_1'' = \mathsf{length}(n_1'),\quad n_2' = \mathsf{rlz}(n_2),\quad n_2'' = \mathsf{length}(n_2')}{\begin{array}{l} n_1 <_\circ n_2 = (\mathsf{F} \lhd \mathsf{Z}(n_2) \\ \qquad \rhd (\mathsf{T} \lhd \mathsf{Z}(n_1) \rhd (\mathsf{less2}(n_1', n_2') \lhd \mathsf{eq1}(n_1'', n_2'') \rhd n_1'' <_\circ n_2''))) \end{array}} \quad [\mathsf{Less2}]$$

# B SDA representation module

## B.1 Representations

**imports** Nat1DA$^{A.9}$
**exports**
  **sorts** OutString4
  **lexical syntax**
    [0-9_.]+ → OutString4
    [FTMD] → OutString4
  **context-free syntax**
    "(" OutString4 "◁" T4 "▷" OutString4 ")" → OutString4
    "eq1" "(" OutString4 "," OutString4 ")" → T4
    "T" "(" OutString4 ")" → T4
    "(" OutString4 ")" → OutString4 {**bracket**}

    OutString4 "∘++" OutString4 → OutString4 {**left**}

    repres-n(LD4) → OutString4
    repres-n(T4) → OutString4

    repres-s(LD4) → OutString4
    repres-s(T4) → OutString4

    repres-d(LD4) → OutString4
    repres-d(T4) → OutString4

    repres-nfd(LD4) → OutString4
    repres-nfd(T4) → OutString4

    read(OutString4) → LD4
  **variables**
    *Os* [0-9′]∗ → OutString4
**hiddens**
  **context-free syntax**
    os2nat(OutString4) → LD4

    read-n(OutString4) → LD4
    read-d(OutString4) → LD4
    read-s(OutString4) → LD4

    contains-s(OutString4) → T4
    contains-d(OutString4) → T4
  **variables**
    *Ch* [0-9′]∗ → CHAR
    *Chs* [0-9′]∗ → CHAR+
**equations**

$(Os_1 \lhd \mathsf{D} \rhd Os_2) = \mathtt{D}$ [condOS-1]
$(Os_1 \lhd \mathsf{T} \rhd Os_2) = Os_1$ [condOS-1]
$(Os_1 \lhd \mathsf{F} \rhd Os_2) = Os_2$ [condOS-1]
$(Os_1 \lhd \mathsf{M} \rhd Os_2) = \mathtt{M}$ [condOS-1]

$$\mathsf{T}(\mathtt{0}) = \mathsf{T} \tag{trueOS-1}$$
$$\mathsf{T}(\mathtt{1}) = \mathsf{T} \tag{trueOS-2}$$
$$\mathsf{T}(\mathtt{2}) = \mathsf{T} \tag{trueOS-3}$$
$$\mathsf{T}(\mathtt{3}) = \mathsf{T} \tag{trueOS-4}$$
$$\mathsf{T}(\mathtt{4}) = \mathsf{T} \tag{trueOS-5}$$
$$\mathsf{T}(\mathtt{5}) = \mathsf{T} \tag{trueOS-6}$$
$$\mathsf{T}(\mathtt{6}) = \mathsf{T} \tag{trueOS-7}$$
$$\mathsf{T}(\mathtt{7}) = \mathsf{T} \tag{trueOS-8}$$
$$\mathsf{T}(\mathtt{8}) = \mathsf{T} \tag{trueOS-9}$$
$$\mathsf{T}(\mathtt{9}) = \mathsf{T} \tag{trueOS-10}$$
$$\mathsf{T}(\mathtt{\_}) = \mathsf{T} \tag{trueOS-11}$$
$$\mathsf{T}(\mathtt{.}) = \mathsf{T} \tag{trueOS-12}$$
$$\mathsf{T}(\mathtt{T}) = \mathsf{T} \tag{trueOS-13}$$
$$\mathsf{T}(\mathtt{F}) = \mathsf{T} \tag{trueOS-14}$$
$$\mathsf{T}(\mathtt{D}) = \mathsf{D} \tag{trueOS-15}$$
$$\mathsf{T}(\mathtt{M}) = \mathsf{M} \tag{trueOS-16}$$
$$\mathsf{T}(\mathsf{outstring4}(\mathit{Chs}_1\ \mathit{Chs}_2)) = (\mathsf{T}(\mathsf{outstring4}(\mathit{Chs}_2)) \triangleleft \mathsf{T}(\mathsf{outstring4}(\mathit{Chs}_1)) \triangleright \mathsf{M}) \tag{trueOS-17}$$

$$\begin{aligned}&\mathsf{eq1}(\mathsf{outstring4}(\mathit{Ch}_1), \mathsf{outstring4}(\mathit{Ch}_2))\\ &=\ \mathsf{eq1}(\mathsf{os2nat}(\mathsf{outstring4}(\mathit{Ch}_1)), \mathsf{os2nat}(\mathsf{outstring4}(\mathit{Ch}_2)))\end{aligned} \tag{eqOS-1}$$

$$\begin{aligned}&\mathsf{eq1}(\mathsf{outstring4}(\mathit{Ch}_1), \mathsf{outstring4}(\mathit{Ch}_2\ \mathit{Chs}))\\ &=\ ((\mathsf{F} \triangleleft \mathsf{T}(\mathsf{outstring4}(\mathit{Ch}_2\ \mathit{Chs})) \triangleright \mathsf{M}) \triangleleft \mathsf{T}(\mathsf{outstring4}(\mathit{Ch}_1)) \triangleright \mathsf{M})\end{aligned} \tag{eqOS-2}$$

$$\begin{aligned}&\mathsf{eq1}(\mathsf{outstring4}(\mathit{Ch}_1\ \mathit{Chs}), \mathsf{outstring4}(\mathit{Ch}_2))\\ &=\ ((\mathsf{F} \triangleleft \mathsf{T}(\mathsf{outstring4}(\mathit{Ch}_2)) \triangleright \mathsf{M}) \triangleleft \mathsf{T}(\mathsf{outstring4}(\mathit{Ch}_1\ \mathit{Chs})) \triangleright \mathsf{M})\end{aligned} \tag{eqOS-3}$$

$$\begin{aligned}&\mathsf{eq1}(\mathsf{outstring4}(\mathit{Ch}_1\ \mathit{Chs}_1), \mathsf{outstring4}(\mathit{Ch}_2\ \mathit{Chs}_2))\\ &=\ \mathsf{eq1}(\mathsf{outstring4}(\mathit{Ch}_1), \mathsf{outstring4}(\mathit{Ch}_2)) \mathbin{\underset{\circ}{\wedge}} \mathsf{eq1}(\mathsf{outstring4}(\mathit{Chs}_1), \mathsf{outstring4}(\mathit{Chs}_2))\end{aligned} \tag{eqOS-4}$$

$$\mathsf{os2nat}(\mathtt{0}) = [0] \tag{os2nat-1}$$
$$\mathsf{os2nat}(\mathtt{1}) = [1] \tag{os2nat-2}$$
$$\mathsf{os2nat}(\mathtt{2}) = [2] \tag{os2nat-3}$$
$$\mathsf{os2nat}(\mathtt{3}) = [3] \tag{os2nat-4}$$
$$\mathsf{os2nat}(\mathtt{4}) = [4] \tag{os2nat-5}$$
$$\mathsf{os2nat}(\mathtt{5}) = [5] \tag{os2nat-6}$$
$$\mathsf{os2nat}(\mathtt{6}) = [6] \tag{os2nat-7}$$
$$\mathsf{os2nat}(\mathtt{7}) = [7] \tag{os2nat-8}$$
$$\mathsf{os2nat}(\mathtt{8}) = [8] \tag{os2nat-9}$$
$$\mathsf{os2nat}(\mathtt{9}) = [9] \tag{os2nat-10}$$
$$\mathsf{os2nat}(\mathtt{\_}) = [1] \mathbin{{}^{\circ}\frown} [0] \tag{os2nat-11}$$
$$\mathsf{os2nat}(\mathtt{.}) = [1] \mathbin{{}^{\circ}\frown} [1] \tag{os2nat-12}$$
$$\mathsf{os2nat}(\mathtt{T}) = [1] \mathbin{{}^{\circ}\frown} [2] \tag{os2nat-13}$$
$$\mathsf{os2nat}(\mathtt{F}) = [1] \mathbin{{}^{\circ}\frown} [3] \tag{os2nat-14}$$
$$\mathsf{os2nat}(\mathtt{D}) = \mathsf{D} \tag{os2nat-15}$$
$$\mathsf{os2nat}(\mathtt{M}) = \mathsf{M} \tag{os2nat-16}$$

$$\mathsf{outstring4}(\mathit{Chs}_1) \mathbin{\circ\!\!+\!\!+} \mathsf{outstring4}(\mathit{Chs}_2) \tag{concat-1}$$

= ((outstring4($Chs_1$ $Chs_2$) ◁ T(outstring4($Chs_2$)) ▷ `M`) ◁ T(outstring4($Chs_1$)) ▷ `M`)

repres-n(F) = `F` [repres-n-1]
repres-n(T) = `T` [repres-n-2]
repres-n(D) = `D` [repres-n-3]
repres-n(M) = `M` [repres-n-4]

repres-n(*Y*) = (0 ◁ eq1(*Y*, [ ]) [repres-n-5]
▷ (0 ◁ eq1(*Y*, [0])
▷ (1 ◁ eq1(*Y*, [1])
▷ (2 ◁ eq1(*Y*, [2])
▷ (3 ◁ eq1(*Y*, [3])
▷ (4 ◁ eq1(*Y*, [4])
▷ (5 ◁ eq1(*Y*, [5])
▷ (6 ◁ eq1(*Y*, [6])
▷ (7 ◁ eq1(*Y*, [7])
▷ (8 ◁ eq1(*Y*, [8])
▷ (9 ◁ eq1(*Y*, [9])
▷ repres-n([↰ *Y*]) ∘++ repres-n(↳ *Y*)))))))))))))

repres-d(*b*) = repres-n(*b*) [repres-d-1]

repres-d(*Y*) = (repres-n(*Y*) [repres-d-2]
◁ eq1(length(*Y*), [ ]) ∘∨ eq1(length(*Y*), [1])
∘∨ eq1(length(*Y*), [2]) ∘∨ eq1(length(*Y*), [3])
▷ repres-d(*Y* ↲ ↲ ↲) ∘++ . ∘++ repres-n([*Y* ↲ ↲ ↱] ∘⌢ ([*Y* ↲ ↱] ∘⌢ [*Y* ↱])))

repres-s(*b*) = repres-n(*b*) [repres-s-1]

repres-s(*Y*) = (repres-n(*Y*) [repres-s-2]
◁ eq1(length(*Y*), [ ]) ∘∨ eq1(length(*Y*), [1])
∘∨ eq1(length(*Y*), [2]) ∘∨ eq1(length(*Y*), [3])
▷ repres-s(*Y* ↲ ↲ ↲) ∘++ _ ∘++ repres-n([*Y* ↲ ↲ ↱] ∘⌢ ([*Y* ↲ ↱] ∘⌢ [*Y* ↱])))

repres-nfd(*b*) = repres-n(*b*) [repres-nfd-1]
repres-nfd(*Y*) = (repres-n(*Y*) ◁ eq1(length(*Y*), [4]) ▷ repres-d(*Y*)) [repres-nfd-2]

read(*Os*) = ((M ◁ contains-d(*Os*) ▷ read-s(*Os*)) [read-1]
◁ contains-s(*Os*)
▷ (read-d(*Os*) ◁ contains-d(*Os*) ▷ read-n(*Os*)))

read-n(0) = [0] [read-n-1]
read-n(1) = [1] [read-n-2]
read-n(2) = [2] [read-n-3]
read-n(3) = [3] [read-n-4]

$\text{read-n}(\texttt{4}) = [4]$ [read-n-5]

$\text{read-n}(\texttt{5}) = [5]$ [read-n-6]

$\text{read-n}(\texttt{6}) = [6]$ [read-n-7]

$\text{read-n}(\texttt{7}) = [7]$ [read-n-8]

$\text{read-n}(\texttt{8}) = [8]$ [read-n-9]

$\text{read-n}(\texttt{9}) = [9]$ [read-n-10]

$\text{read-n}(\texttt{\_}) = \text{M}$ [read-n-11]

$\text{read-n}(\texttt{.}) = \text{M}$ [read-n-12]

$\text{read-n}(\texttt{T}) = \text{M}$ [read-n-13]

$\text{read-n}(\texttt{F}) = \text{M}$ [read-n-14]

$\text{read-n}(\texttt{D}) = \text{D}$ [read-n-15]

$\text{read-n}(\texttt{M}) = \text{M}$ [read-n-16]

$\text{read-n}(\text{outstring4}(Chs_1\ Chs_2)) = \text{read-n}(\text{outstring4}(Chs_1)) \;{}^{\circ}\!\frown\; \text{read-n}(\text{outstring4}(Chs_2))$ [read-n-17]

$\text{read-d}(\text{outstring4}(Ch)) = \text{read-n}(\text{outstring4}(Ch))$ [read-d-1]

$\text{read-d}(\text{outstring4}(Ch_1\ Ch_2)) = \text{read-n}(\text{outstring4}(Ch_1\ Ch_2))$ [read-d-2]

$\text{read-d}(\text{outstring4}(Ch_1\ Ch_2\ Ch_3)) = \text{read-n}(\text{outstring4}(Ch_1\ Ch_2\ Ch_3))$ [read-d-3]

$\text{read-d}(\text{outstring4}(Ch_1\ Ch_2\ Ch_3\ Ch_4)) = \text{M}$ [read-d-4]

$\text{read-d}(\text{outstring4}(Chs\ \texttt{"0"}\ Ch_2\ Ch_3\ Ch_4)) = \text{M}$ [read-d-5]

$\text{read-d}(\text{outstring4}(Chs\ \texttt{"1"}\ Ch_2\ Ch_3\ Ch_4)) = \text{M}$ [read-d-6]

$\text{read-d}(\text{outstring4}(Chs\ \texttt{"2"}\ Ch_2\ Ch_3\ Ch_4)) = \text{M}$ [read-d-7]

$\text{read-d}(\text{outstring4}(Chs\ \texttt{"3"}\ Ch_2\ Ch_3\ Ch_4)) = \text{M}$ [read-d-8]

$\text{read-d}(\text{outstring4}(Chs\ \texttt{"4"}\ Ch_2\ Ch_3\ Ch_4)) = \text{M}$ [read-d-9]

$\text{read-d}(\text{outstring4}(Chs\ \texttt{"5"}\ Ch_2\ Ch_3\ Ch_4)) = \text{M}$ [read-d-10]

$\text{read-d}(\text{outstring4}(Chs\ \texttt{"6"}\ Ch_2\ Ch_3\ Ch_4)) = \text{M}$ [read-d-11]

$\text{read-d}(\text{outstring4}(Chs\ \texttt{"7"}\ Ch_2\ Ch_3\ Ch_4)) = \text{M}$ [read-d-12]

$\text{read-d}(\text{outstring4}(Chs\ \texttt{"8"}\ Ch_2\ Ch_3\ Ch_4)) = \text{M}$ [read-d-13]

$\text{read-d}(\text{outstring4}(Chs\ \texttt{"9"}\ Ch_2\ Ch_3\ Ch_4)) = \text{M}$ [read-d-14]

$\text{read-d}(\text{outstring4}(Chs\ \texttt{"\_"}\ Ch_2\ Ch_3\ Ch_4)) = \text{M}$ [read-d-15]

$\text{read-d}(\text{outstring4}(Chs\ \texttt{"F"}\ Ch_2\ Ch_3\ Ch_4)) = \text{M}$ [read-d-16]

$\text{read-d}(\text{outstring4}(Chs\ \texttt{"T"}\ Ch_2\ Ch_3\ Ch_4)) = \text{M}$ [read-d-17]

$\text{read-d}(\text{outstring4}(Chs\ \texttt{"D"}\ Ch_2\ Ch_3\ Ch_4)) = \text{D}$ [read-d-18]

$\text{read-d}(\text{outstring4}(Chs\ \texttt{"M"}\ Ch_2\ Ch_3\ Ch_4)) = \text{M}$ [read-d-19]

$\text{read-d}(\text{outstring4}(Chs\ \texttt{"."}\ Ch_1\ Ch_2\ Ch_3))$ [read-d-20]

$= \ \text{read-d}(\text{outstring4}(Chs)) \;{}^{\circ}\!\frown\; \text{read-n}(\text{outstring4}(Ch_1\ Ch_2\ Ch_3))$

$\text{read-s}(\text{outstring4}(Ch)) = \text{read-n}(\text{outstring4}(Ch))$ [read-s-1]

$\text{read-s}(\text{outstring4}(Ch_1\ Ch_2)) = \text{read-n}(\text{outstring4}(Ch_1\ Ch_2))$ [read-s-2]

$\text{read-s}(\text{outstring4}(Ch_1\ Ch_2\ Ch_3)) = \text{read-n}(\text{outstring4}(Ch_1\ Ch_2\ Ch_3))$ [read-s-3]

$\text{read-s}(\text{outstring4}(Ch_1\ Ch_2\ Ch_3\ Ch_4)) = \text{M}$ [read-s-4]

$\text{read-s}(\text{outstring4}(Chs\ \texttt{"0"}\ Ch_2\ Ch_3\ Ch_4)) = \text{M}$ [read-s-5]

$\text{read-s}(\text{outstring4}(Chs\ \texttt{"1"}\ Ch_2\ Ch_3\ Ch_4)) = \text{M}$ [read-s-6]

$\text{read-s}(\text{outstring4}(Chs\ \texttt{"2"}\ Ch_2\ Ch_3\ Ch_4)) = \text{M}$ [read-s-7]

$\text{read-s}(\text{outstring4}(Chs\ \texttt{"3"}\ Ch_2\ Ch_3\ Ch_4)) = \text{M}$ [read-s-8]

$\text{read-s}(\text{outstring4}(Chs\ \texttt{"4"}\ Ch_2\ Ch_3\ Ch_4)) = \text{M}$ [read-s-9]

$\text{read-s}(\text{outstring4}(Chs\ \texttt{"5"}\ Ch_2\ Ch_3\ Ch_4)) = \text{M}$ [read-s-10]

$\text{read-s}(\text{outstring4}(Chs\ \texttt{"6"}\ Ch_2\ Ch_3\ Ch_4)) = \text{M}$ [read-s-11]

$\text{read-s}(\text{outstring4}(Chs\ \texttt{"7"}\ Ch_2\ Ch_3\ Ch_4)) = \text{M}$ [read-s-12]

read-s(outstring4($Chs$ "8" $Ch_2$ $Ch_3$ $Ch_4$)) = M [read-s-13]
read-s(outstring4($Chs$ "9" $Ch_2$ $Ch_3$ $Ch_4$)) = M [read-s-14]
read-s(outstring4($Chs$ "." $Ch_2$ $Ch_3$ $Ch_4$)) = M [read-s-15]
read-s(outstring4($Chs$ "F" $Ch_2$ $Ch_3$ $Ch_4$)) = M [read-s-16]
read-s(outstring4($Chs$ "T" $Ch_2$ $Ch_3$ $Ch_4$)) = M [read-s-17]
read-s(outstring4($Chs$ "D" $Ch_2$ $Ch_3$ $Ch_4$)) = D [read-s-18]
read-s(outstring4($Chs$ "M" $Ch_2$ $Ch_3$ $Ch_4$)) = M [read-s-19]

read-s(outstring4($Chs$ "_" $Ch_1$ $Ch_2$ $Ch_3$)) [read-s-20]
= read-s(outstring4($Chs$)) $\overset{\circ}{\frown}$ read-n(outstring4($Ch_1$ $Ch_2$ $Ch_3$))

contains-s(0) = F [contains-s-1]
contains-s(1) = F [contains-s-2]
contains-s(2) = F [contains-s-3]
contains-s(3) = F [contains-s-4]
contains-s(4) = F [contains-s-5]
contains-s(5) = F [contains-s-6]
contains-s(6) = F [contains-s-7]
contains-s(7) = F [contains-s-8]
contains-s(8) = F [contains-s-9]
contains-s(9) = F [contains-s-10]
contains-s(_) = T [contains-s-11]
contains-s(.) = F [contains-s-12]
contains-s(T) = F [contains-s-13]
contains-s(F) = F [contains-s-14]
contains-s(D) = D [contains-s-15]
contains-s(M) = M [contains-s-16]

contains-s(outstring4($Chs_1$ $Chs_2$)) [contains-s-17]
= contains-s(outstring4($Chs_1$)) $\overset{\circ}{\vee}$ contains-s(outstring4($Chs_2$))

contains-d(0) = F [contains-d-1]
contains-d(1) = F [contains-d-2]
contains-d(2) = F [contains-d-3]
contains-d(3) = F [contains-d-4]
contains-d(4) = F [contains-d-5]
contains-d(5) = F [contains-d-6]
contains-d(6) = F [contains-d-7]
contains-d(7) = F [contains-d-8]
contains-d(8) = F [contains-d-9]
contains-d(9) = F [contains-d-10]
contains-d(_) = F [contains-d-11]
contains-d(.) = T [contains-d-12]
contains-d(T) = F [contains-d-13]
contains-d(F) = F [contains-d-14]
contains-d(D) = D [contains-d-15]
contains-d(M) = M [contains-d-16]

contains-d(outstring4($Chs_1$ $Chs_2$)) [contains-d-17]
= contains-d(outstring4($Chs_1$)) $\overset{\circ}{\vee}$ contains-d(outstring4($Chs_2$))

# C The SDA specification in C

## C.1 BoolDA.c

```
typedef enum T4 {F, T, D, M} T4;

#define cond(b1, op, b2) (op == F ? b2 : (op == T ? b1 : (op == D ? D : M)))

T4 eq1(T4 b1, T4 b2)
{
return cond(cond(T, b2, F), b1, cond(F, b2, T));
}

T4 not(T4 b)
{
return cond(F, b, T);
}

T4 or1(T4 b1, T4 b2)
{
return cond(T, b1, b2);
}

T4 or2(T4 b1, T4 b2)
{
return or1(b2, b1);
}

T4 and1(T4 b1, T4 b2)
{
return cond(b2, b1, F);
}

T4 and2(T4 b1, T4 b2)
{
return and1(b2, b1);
}

T4 True_T4(T4 b)
{
return eq1(b, b);
}
```

## C.2 init.c

```
#include "TB.h"
#include "BoolDA.c"

term *Dtrm;
term *Mtrm;

void init()
{
  Dtrm = TBmake("\"D\"");
  TBprotect(&Dtrm);
```

```
  Mtrm = TBmake("\"M\"");
  TBprotect(&Mtrm);
}

#define cond_trm(d1, op, d2) \
        (op == F ? d2 : (op == T ? d1 : (op == D ? Dtrm : Mtrm)))
```

## C.3 initDigitDA.c

```
#include "TB.h"

term *d[10];

void initDigitDA()
{
   int i;
   char buf[2];

   for(i=0; i<10; i++) {
      sprintf(buf, "%d", i);
      d[i] = TBmake("%s", buf);
      TBprotect(&d[i]);
   }
}

#define base d[9]
```

## C.4 DigitDA.c

```
#include "init.c"
#include "initDigitDA.c"

term *S(term *dt)
{  char *s;

   if(TBmatch(dt, "%s", &s)) {
      switch(s[0]){
         case '0': return d[1];
         case '1': return d[2];
         case '2': return d[3];
         case '3': return d[4];
         case '4': return d[5];
         case '5': return d[6];
         case '6': return d[7];
         case '7': return d[8];
         case '8': return d[9];
         case '9': return Mtrm;
         case 'D': return Dtrm;
         case 'M': return Mtrm;
         default: assert(0);
      }
   } else {
      TBmsg("panic in S: %t\n", dt);
      return Mtrm;
   }
```

```
}

term *P(term *dt)
{  char *s;

   if(TBmatch(dt, "%s", &s)) {
      switch(s[0]){
         case '0': return Mtrm;
         case '1': return d[0];
         case '2': return d[1];
         case '3': return d[2];
         case '4': return d[3];
         case '5': return d[4];
         case '6': return d[5];
         case '7': return d[6];
         case '8': return d[7];
         case '9': return d[8];
         case 'D': return Dtrm;
         case 'M': return Mtrm;
         default: assert(0);
      }
   } else {
      TBmsg("panic in P: %t\n", dt);
      return Mtrm;
   }
}

T4 Z(term *dt)
{  char *s;

   if(TBmatch(dt, "%s", &s)) {
      switch(s[0]){
         case '0': return T;
         case '1': return F;
         case '2': return F;
         case '3': return F;
         case '4': return F;
         case '5': return F;
         case '6': return F;
         case '7': return F;
         case '8': return F;
         case '9': return F;
         case 'D': return D;
         case 'M': return M;
         default: assert(0);
      }
   } else {
      TBmsg("panic in Z: %t\n", dt);
      return M;
   }
}

T4 eq1_D4(term *dt1, term *dt2)
{
return cond(Z(dt2), Z(dt1), and1(not(Z(dt2)), eq1_D4(P(dt1), P(dt2))));
```

```
}

T4 True_D4(term *dt)
{
return or1(Z(dt), T);
}

term *t2d(T4 b)
{
return cond_trm(d[1], b, d[0]);
}

T4 lt2(term *dt1, term *dt2)
{
return cond(F, Z(dt2), cond(T, Z(dt1), lt2(P(dt1), P(dt2))));
}

term *carry1(term *dt1, term *dt2)
{
return cond_trm(d[0],
                Z(dt1),
                cond_trm(d[1], eq1_D4(dt2, base), carry1(P(dt1), S(dt2))));
}

term *borrow2(term *dt1, term *dt2)
{
return t2d(lt2(dt1, dt2));
}

term *compl(term *dt)
{
return cond_trm(base, eq1_D4(dt, d[1]), P(compl(P(dt))));
}

term *modsum1(term *dt1, term *dt2)
{
return cond_trm(dt2,
                Z(dt1),
                cond_trm(P(dt1), eq1_D4(dt2, base), modsum1(P(dt1), S(dt2))));
}

term *modsub2(term *dt1, term *dt2)
{
return cond_trm(dt1,
                Z(dt2),
                cond_trm(compl(dt2), Z(dt1), modsub2(P(dt1), P(dt2))));
}
```

## C.5 LDigitDA.c

```
#include "DigitDA.c"

term *EMPTY(void)
{
   return NULL;
```

```
}

term *Ulist(term *dt)
{  char *s;

   if(TBmatch(dt, "%s", &s)) {
      switch(s[0]){
         case 'D': return Dtrm;
         case 'M': return Mtrm;
         default: return list_concat_term(EMPTY(), dt);
      }
   } else {
      TBmsg("panic in Ulist: %t\n", dt);
      return Mtrm;
   }
}

T4 True_LD4(term *y)
{  char *s;
   term_list *t;

   if(TBmatch(y, "%s", &s)) {
      switch(s[0]){
         case 'D': return D;
         case 'M': return M;
         default: assert(0);
      }
   } else if(TBmatch(y, "[%l]", &t)) {
      if(!t) return T;
      else return and1(True_D4(list_first(t)), True_LD4(list_next(t)));
   } else {
      TBmsg("panic in True_LD4: %t\n", y);
      return M;
   }
}


term *Lappend(term *y1, term *y2)
{
return cond_trm(cond_trm(list_append(y1, y2), True_LD4(y2), Mtrm),
                True_LD4(y1),
                Mtrm);
}


T4 E(term *y)
{  char *s;
   term_list *t;

   if(TBmatch(y, "%s", &s)) {
      switch(s[0]){
         case 'D': return D;
         case 'M': return M;
         default: assert(0);
      }
```

```
   } else if(TBmatch(y, "[%l]", &t)) {
      if(!t) return T;
      else return not(and1(True_D4(list_first(t)), True_LD4(list_next(t))));
   } else {
      TBmsg("panic in E: %t\n", y);
      return M;
   }
}


term *LeftHead(term *y)
{  char *s;
   term_list *t;

   if(TBmatch(y, "%s", &s)) {
      switch(s[0]){
         case 'D': return Dtrm;
         case 'M': return Mtrm;
         default: assert(0);
      }
   } else if(TBmatch(y, "[%l]", &t)) {
      if(!t) return Mtrm;
      else return cond_trm(list_first(t),
                           and1(True_D4(list_first(t)),
                                True_LD4(list_next(t))),
                           Mtrm);
   } else {
      TBmsg("panic in LeftHead: %t\n", y);
      return Mtrm;
   }
}


term *LRTail(term *y)
{  char *s;
   term_list *t;

   if(TBmatch(y, "%s", &s)) {
      switch(s[0]){
         case 'D': return Dtrm;
         case 'M': return Mtrm;
         default: assert(0);
      }
   } else if(TBmatch(y, "[%l]", &t)) {
      if(!t) return Mtrm;
      else return cond_trm(list_next(t), True_D4(list_first(t)), Mtrm);
   } else {
      TBmsg("panic in LeftHead: %t\n", y);
      return Mtrm;
   }
}


T4 eq1_LD4(term *y1, term *y2)
{
```

```
return cond(T,
            and1(E(y1), E(y2)),
            and1(and1(and1(not(E(y1)), not(E(y2))),
                      eq1_D4(LeftHead(y1), LeftHead(y2))),
                 eq1_LD4(LRTail(y1), LRTail(y2))));
}

term *inv(term *y)
{
return cond_trm(EMPTY(), E(y), Lappend(inv(LRTail(y)), Ulist(LeftHead(y))));
}

term *RightHead(term *y)
{
return LeftHead(inv(y));
}

term *RLTail(term *y)
{
return inv(LRTail(inv(y)));
}

term *Rappend(term *y1, term *y2)
{
return inv(Lappend(inv(y2), inv(y1)));
}
```

## C.6 Nat1DA.c

```
#include "LDigitDA.c"

term *rlz(term *n)
{
return cond_trm(n, or1(E(n), not(Z(LeftHead(n)))), rlz(LRTail(n)));
}

term *lS(term *n)
{
return rlz(cond_trm(Ulist(d[1]),
                    E(n),
                    cond_trm(Lappend(lS(RLTail(n)), Ulist(d[0])),
                             eq1_D4(RightHead(n), base),
                             Lappend(RLTail(n), Ulist(S(RightHead(n)))))));
}

term *lP(term *n)
{
return rlz(cond_trm(Lappend(lP(RLTail(n)), Ulist(base)),
                    eq1_D4(RightHead(n), d[0]),
                    Lappend(RLTail(n), Ulist(P(RightHead(n))))));
}

T4 lZ(term *n)
{
return E(rlz(n));
```

```
}

term *length(term *n)
{
return cond_trm(EMPTY(), E(n), 1S(length(LRTail(n))));
}
```

## C.7 Representations.c

```
#include "Nat1DA.c"

#define cond_str(s1, op, s2) \
        (op == F ? s2 : (op == T ? s1 : (op == D ? "D" : "M")))

T4 True_str(char *s)
{
  while(*s == '0' || *s == '1' || *s == '2' || *s == '3' || *s == '4' ||
        *s == '5' || *s == '6' || *s == '7' || *s == '8' || *s == '9' ||
        *s == ' ' || *s == '.' || *s == 'T' || *s == 'F') s++;
  if (*s == '\0') return T;
  else if (*s == 'D') return D;
  else return M;
}

T4 eq1_str(char *s1, char *s2)
{
  return cond(cond(strcmp(s1, s2) == 0, True_str(s2), M), True_str(s1), M);
}

char *helpcat(char *s1, char *s2)
{ char *str;

  str = (char *) malloc(strlen(s1) + strlen(s2) + 1);
  if (str != NULL) {
    strcpy(str, s1);
    strcat(str, s2);
  }
  return str;
}

char *concat1(char *s1, char *s2)
{
  return cond_str(cond_str(helpcat(s1, s2), True_str(s2), "M"), True_str(s1), "M");
}

char *represb_n(T4 b)
{
  switch(b) {
    case F: return "F";
    case T: return "T";
    case D: return "D";
    default: return "M";
  }
}
```

```
char *represl_n(term *y)
{
  return cond_str("0", eq1_LD4(y, EMPTY()),
         cond_str("0", eq1_LD4(y, Ulist(d[0])),
         cond_str("1", eq1_LD4(y, Ulist(d[1])),
         cond_str("2", eq1_LD4(y, Ulist(d[2])),
         cond_str("3", eq1_LD4(y, Ulist(d[3])),
         cond_str("4", eq1_LD4(y, Ulist(d[4])),
         cond_str("5", eq1_LD4(y, Ulist(d[5])),
         cond_str("6", eq1_LD4(y, Ulist(d[6])),
         cond_str("7", eq1_LD4(y, Ulist(d[7])),
         cond_str("8", eq1_LD4(y, Ulist(d[8])),
         cond_str("9", eq1_LD4(y, Ulist(d[9])),
         concat1(represl_n(Ulist(LeftHead(y))), represl_n(LRTail(y))))))))))))));
}

char *represb_s(T4 b)
{
  return represb_n(b);
}

char *represl_s(term *y)
{
  return cond_str(represl_n(y),
                  or1(or1(or1(eq1_LD4(length(y), EMPTY()),
                              eq1_LD4(length(y), Ulist(d[1]))),
                          eq1_LD4(length(y), Ulist(d[2]))) ,
                      eq1_LD4(length(y), Ulist(d[3]))),
                  concat1(concat1(represl_s(RLTail(RLTail(RLTail(y)))), " "),
                          represl_n(Lappend(Ulist(RightHead(RLTail(RLTail(y)))),
                                            Lappend(Ulist(RightHead(RLTail(y))),
                                                    Ulist(RightHead(y)))))));
}

char *represb_d(T4 b)
{
  return represb_n(b);
}

char *represl_d(term *y)
{
  return cond_str(represl_n(y),
                  or1(or1(or1(eq1_LD4(length(y), EMPTY()),
                              eq1_LD4(length(y), Ulist(d[1]))),
                          eq1_LD4(length(y), Ulist(d[2]))) ,
                      eq1_LD4(length(y), Ulist(d[3]))),
                  concat1(concat1(represl_d(RLTail(RLTail(RLTail(y)))), "."),
                          represl_n(Lappend(Ulist(RightHead(RLTail(RLTail(y)))),
                                            Lappend(Ulist(RightHead(RLTail(y))),
                                                    Ulist(RightHead(y)))))));
}

char *represb_nfd(T4 b)
{
  return represb_n(b);
```

```
}

char *represl_nfd(term *y)
{
  return cond_str(represl_n(y), eq1_LD4(length(y), Ulist(d[4])), represl_d(y));
}

T4 contains_s(char *s)
{
  while(*s == '0' || *s == '1' || *s == '2' || *s == '3' || *s == '4' ||
        *s == '5' || *s == '6' || *s == '7' || *s == '8' || *s == '9' ||
        *s == '.' || *s == 'T' || *s == 'F') s++;
  if (*s == '\0') return F;
  else if (*s == ' ') return True_str(s++);
  else if (*s == 'D') return D;
  else return M;
}

T4 contains_d(char *s)
{
  while(*s == '0' || *s == '1' || *s == '2' || *s == '3' || *s == '4' ||
        *s == '5' || *s == '6' || *s == '7' || *s == '8' || *s == '9' ||
        *s == ' ' || *s == 'T' || *s == 'F') s++;
  if (*s == '\0') return F;
  else if (*s == '.') return True_str(s++);
  else if (*s == 'D') return D;
  else return M;
}

term *readchar(char c)
{
  switch(c) {
    case '0': return Ulist(d[0]);
    case '1': return Ulist(d[1]);
    case '2': return Ulist(d[2]);
    case '3': return Ulist(d[3]);
    case '4': return Ulist(d[4]);
    case '5': return Ulist(d[5]);
    case '6': return Ulist(d[6]);
    case '7': return Ulist(d[7]);
    case '8': return Ulist(d[8]);
    case '9': return Ulist(d[9]);
    case 'D': return Dtrm;
    default: return Mtrm;
  }
}

term *read_n(char *s)
{ term *y;

  if(*s == '\0') return Mtrm;
  else { y = readchar(*s);
         while (*++s != '\0') y = Lappend(y, readchar(*s));
         return y;
  }
```

```
}

term *read_s(char *s)
{ int len, i;
  char *s1, *s2;

  len = strlen(s);
  if(len < 4) return read_n(s);
  else if (len == 4) return Mtrm;
  else switch(s[len-4]) {
         case ' ': s1 = (char *) malloc(len - 3);
                   if (s1 != NULL) {
                     for (i = 0; i < len - 4; i++)
                       s1[i] = s[i];
                     s1[i] = '\0';
                   }
                   s2 = s + len - 3;
                   return Lappend(read_s(s1), read_n(s2));
         case 'D': return Dtrm;
         default: return Mtrm;
       }
}

term *read_d(char *s)
{ int len, i;
  char *s1, *s2;

  len = strlen(s);
  if(len < 4) return read_n(s);
  else if (len == 4) return Mtrm;
  else switch(s[len-4]) {
         case '.': s1 = (char *) malloc(len - 3);
                   if (s1 != NULL) {
                     for (i = 0; i < len - 4; i++)
                       s1[i] = s[i];
                     s1[i] = '\0';
                   }
                   s2 = s + len - 3;
                   return Lappend(read_d(s1), read_n(s2));
         case 'D': return Dtrm;
         default: return Mtrm;
       }
}

term *read_str(char *s)
{
  return cond_trm(cond_trm(Mtrm, contains_d(s), read_s(s)),
                  contains_s(s),
                  cond_trm(read_d(s), contains_d(s), read_n(s)));
}
```

## C.8 sda-rec.tif.c

```
/* GENERATED AUTOMATICALLY, DO NOT MODIFY */
/* Tool Interface File for tool: sda-rec */
```

```
#include "TB.h"

/* Prototypes for functions used in event handler */
term * expr(term *);
void   rec_terminate(term *);

/* Event handler for tool: sda-rec */

term *sda_rec_handler(term *e){
term *term_arg[1];

if(TBmatch(e, "rec-eval(expr(%t))", &term_arg[0])){
return expr(term_arg[0]);
} else if(TBmatch(e, "rec-terminate(%t)", &term_arg[0])){
rec_terminate(term_arg[0]);
return NULL;
}  else  {
TBprintf(stderr, "Ignored: %t\n", e);
return NULL;
}
}

/* Check input signature for tool: sda-rec
 * returns: NULL (requested input signature is subset of tool's input signature);
 *          non-null term: first element not in tool's input signature.
 * NOTE:    In the signature all terms have as first argument <sda-rec>, but this
 *          does NOT appear in the actual terms that will be sent to the tool.
 */

term *sda_rec_check_in_sign(term_list *reqs){
char *in_sign[2];
int i;

in_sign[0] = "rec-eval(<sda-rec>,expr(<term>))";
in_sign[1] = "rec-terminate(<sda-rec>,<term>)";

for( ; reqs; reqs = next(reqs)){
for(i = 0; i < 2; i++){
if(TBmatch(first(reqs), in_sign[i])) goto found;
}
return first(reqs);
found:;
}
return NULL;
}
```

## C.9 sda-rec.c

```
#include "sda-rec.tif.c"
#include "Nat2DA.c"

term *expr(term *trm)
{ term *t1, *t2;
```

```
  if (TBmatch(trm, "add(%t, %t)", &t1, &t2))
    return TBmake("snd-value(%t)", add2(t1, t2));
  else if (TBmatch(trm, "sub(%t, %t)", &t1, &t2))
    return TBmake("snd-value(%t)", sub2(t1, t2));
  else if (TBmatch(trm, "mult(%t, %t)", &t1, &t2))
    return TBmake("snd-value(%t)", mult2(t1, t2));
  else if (TBmatch(trm, "div(%t, %t)", &t1, &t2))
    return TBmake("snd-value(%t)", div2(t1, t2));
  else if (TBmatch(trm, "less(%t, %t)", &t1, &t2))
    return TBmake("snd-value(%t)", TBmake("%d", less2(t1, t2)));
  else return TBmake("snd-value(sda-error(%t))", trm);
}

void rec_terminate(term *t)
{
  exit(0);
}

void main(int argc, char *argv[])
{
  TBinit("sda_rec", argc, argv, sda_rec_handler, sda_rec_check_in_sign);

  init();
  initDigitDA();

  TBeventloop();
}
```

## C.10 sda-lst.tif.c

```
/* GENERATED AUTOMATICALLY, DO NOT MODIFY */
/* Tool Interface File for tool: sda-lst */

#include "TB.h"

/* Prototypes for functions used in event handler */
term * expr(term *);
void   rec_terminate(term *);

/* Event handler for tool: sda-lst */

term *sda_lst_handler(term *e){
term *term_arg[1];

if(TBmatch(e, "rec-eval(expr(%t))", &term_arg[0])){
return expr(term_arg[0]);
} else if(TBmatch(e, "rec-terminate(%t)", &term_arg[0])){
rec_terminate(term_arg[0]);
return NULL;
}  else  {
TBprintf(stderr, "Ignored: %t\n", e);
return NULL;
}
}
```

```
/* Check input signature for tool: sda-lst
 * returns: NULL (requested input signature is subset of tool's input signature);
 *          non-null term: first element not in tool's input signature.
 * NOTE:    In the signature all terms have as first argument <sda-lst>, but this
 *          does NOT appear in the actual terms that will be sent to the tool.
 */

term *sda_lst_check_in_sign(term_list *reqs){
char *in_sign[2];
int i;

in_sign[0] = "rec-eval(<sda-lst>,expr(<term>))";
in_sign[1] = "rec-terminate(<sda-lst>,<term>)";

for( ; reqs; reqs = next(reqs)){
for(i = 0; i < 2; i++){
if(TBmatch(first(reqs), in_sign[i])) goto found;
}
return first(reqs);
found:;
}
return NULL;
}
```

## C.11 sda-lst.c

```
#include "sda-lst.tif.c"
#include "Nat3DA.c"

term *expr(term *trm)
{ term *t1, *t2;

  if (TBmatch(trm, "add(%t, %t)", &t1, &t2))
    return TBmake("snd-value(%t)", add2(t1, t2));
  else if (TBmatch(trm, "sub(%t, %t)", &t1, &t2))
    return TBmake("snd-value(%t)", sub2(t1, t2));
  else if (TBmatch(trm, "mult(%t, %t)", &t1, &t2))
    return TBmake("snd-value(%t)", mult2(t1, t2));
  else if (TBmatch(trm, "div(%t, %t)", &t1, &t2))
    return TBmake("snd-value(%t)", div2(t1, t2));
  else if (TBmatch(trm, "less(%t, %t)", &t1, &t2))
    return TBmake("snd-value(%t)", TBmake("%d", less2(t1, t2)));
  else return TBmake("snd-value(sda-error(%t))", trm);
}

void rec_terminate(term *t)
{
  exit(0);
}

void main(int argc, char *argv[])
{
  TBinit("sda_lst", argc, argv, sda_lst_handler, sda_lst_check_in_sign);

  init();
```

```
  initDigitDA();

  TBeventloop();
}
```

## C.12 repres.tif.c

```
/* GENERATED AUTOMATICALLY, DO NOT MODIFY */
/* Tool Interface File for tool: repres */

#include "TB.h"

/* Prototypes for functions used in event handler */
term * input(char *);
term * output(term *);
void   rec_terminate(term *);

/* Event handler for tool: repres */

term *repres_handler(term *e){
term *term_arg[1];
char *str_arg[1];

if(TBmatch(e, "rec-eval(input(%s))", &str_arg[0])){
return input(str_arg[0]);
} else if(TBmatch(e, "rec-eval(output(%t))", &term_arg[0])){
return output(term_arg[0]);
} else if(TBmatch(e, "rec-terminate(%t)", &term_arg[0])){
rec_terminate(term_arg[0]);
return NULL;
}  else  {
TBprintf(stderr, "Ignored: %t\n", e);
return NULL;
}
}

/* Check input signature for tool: repres
 * returns: NULL (requested input signature is subset of tool's input signature);
 *          non-null term: first element not in tool's input signature.
 * NOTE:    In the signature all terms have as first argument <repres>, but this
 *          does NOT appear in the actual terms that will be sent to the tool.
 */

term *repres_check_in_sign(term_list *reqs){
char *in_sign[3];
int i;

in_sign[0] = "rec-eval(<repres>,input(<str>))";
in_sign[1] = "rec-eval(<repres>,output(<term>))";
in_sign[2] = "rec-terminate(<repres>,<term>)";

for( ; reqs; reqs = next(reqs)){
for(i = 0; i < 3; i++){
if(TBmatch(first(reqs), in_sign[i])) goto found;
}
```

```
return first(reqs);
found:;
}
return NULL;
}
```

## C.13 repres.c

```
#include "repres.tif.c"
#include "Representations.c"

term *input(char *s)
{
  return TBmake("snd-value(%t)", read_str(s));
}

term *output(term *trm)
{ term *t;
  T4 b;

  if (TBmatch(trm, "none(%d)", &b))
    return TBmake("snd-value(%s)", represb_n(b));
  else if (TBmatch(trm, "none(%t)", &t))
    return TBmake("snd-value(%s)", represl_n(t));
  else if (TBmatch(trm, "dot(%d)", &b))
    return TBmake("snd-value(%s)", represb_d(b));
  else if (TBmatch(trm, "dot(%t)", &t))
    return TBmake("snd-value(%s)", represl_d(t));
  else if (TBmatch(trm, "space(%d)", &b))
    return TBmake("snd-value(%s)", represb_s(b));
  else if (TBmatch(trm, "space(%t)", &t))
    return TBmake("snd-value(%s)", represl_s(t));
  else if (TBmatch(trm, "nfd(%d)", &b))
    return TBmake("snd-value(%s)", represb_nfd(b));
  else if (TBmatch(trm, "nfd(%t)", &t))
    return TBmake("snd-value(%s)", represl_nfd(t));
  else return TBmake("snd-value(repres-error(%t))", trm);
}

void rec_terminate(term *t)
{
  exit(0);
}

void main(int argc, char *argv[])
{
  TBinit("repres", argc, argv, repres_handler, repres_check_in_sign);

  init();
  initDigitDA();

  TBeventloop();
}
```

# D ToolBus scripts

## D.1 One-tool version

```
process UI is
  let Tid : ui
  in
     execute(ui, Tid?) .
     ( CALC-REC-BUTTON(Tid)
       +
       CALC-LST-BUTTON(Tid)
       +
       QUIT-BUTTON(Tid)
     ) * delta
  endlet

tool ui is {command = "wish-adapter -script ui-sda.tcl"}


process CALC-REC-BUTTON(Tid : ui) is
  let  D : str , E : str, U : term , V : term, W : term, S : str
  in
     rec-event(Tid, button(calc-rec)) .
     snd-eval(Tid, get-expr-dialog) .
     ( rec-value(Tid, add(D?, E?)) .
       snd-msg(make-input, D) .
       rec-msg(make-input, D, U?) .
       snd-msg(make-input, E) .
       rec-msg(make-input, E, V?) .
       snd-msg(compute-rec, add(U, V)) .
       rec-msg(compute-rec, add(U, V), W?)
       +
       rec-value(Tid, sub(D?, E?)) .
       snd-msg(make-input, D) .
       rec-msg(make-input, D, U?) .
       snd-msg(make-input, E) .
       rec-msg(make-input, E, V?) .
       snd-msg(compute-rec, sub(U, V)) .
       rec-msg(compute-rec, sub(U, V), W?)
       +
       rec-value(Tid, mult(D?, E?)) .
       snd-msg(make-input, D) .
       rec-msg(make-input, D, U?) .
       snd-msg(make-input, E) .
       rec-msg(make-input, E, V?) .
       snd-msg(compute-rec, mult(U, V)) .
       rec-msg(compute-rec, mult(U, V), W?)
       +
       rec-value(Tid, div(D?, E?)) .
       snd-msg(make-input, D) .
       rec-msg(make-input, D, U?) .
       snd-msg(make-input, E) .
       rec-msg(make-input, E, V?) .
       snd-msg(compute-rec, div(U, V)) .
       rec-msg(compute-rec, div(U, V), W?)
```

```
      +
      rec-value(Tid, less(D?, E?)) .
      snd-msg(make-input, D) .
      rec-msg(make-input, D, U?) .
      snd-msg(make-input, E) .
      rec-msg(make-input, E, V?) .
      snd-msg(compute-rec, less(U, V)) .
      rec-msg(compute-rec, less(U, V), W?)
    ) .
    snd-eval(Tid, get-repres-dialog) .
    ( rec-value(Tid, none) .
      snd-msg(make-output, none(W)) .
      rec-msg(make-output, none(W), S?) .
      snd-do(Tid, display(S))
      +
      rec-value(Tid, dot) .
      snd-msg(make-output, dot(W)) .
      rec-msg(make-output, dot(W), S?) .
      snd-do(Tid, display(S))
      +
      rec-value(Tid, space) .
      snd-msg(make-output, space(W)) .
      rec-msg(make-output, space(W), S?) .
      snd-do(Tid, display(S))
      +
      rec-value(Tid, nfd) .
      snd-msg(make-output, nfd(W)) .
      rec-msg(make-output, nfd(W), S?) .
      snd-do(Tid, display(S))
    ) . snd-ack-event(Tid, button(calc-rec))
  endlet

process CALC-LST-BUTTON(Tid : ui) is
  let  D : str , E : str, U : term , V : term, W : term, S : str
  in
    rec-event(Tid, button(calc-lst)) .
    snd-eval(Tid, get-expr-dialog).
    ( rec-value(Tid, add(D?, E?)) .
      snd-msg(make-input, D) .
      rec-msg(make-input, D, U?) .
      snd-msg(make-input, E) .
      rec-msg(make-input, E, V?) .
      snd-msg(compute-lst, add(U, V)) .
      rec-msg(compute-lst, add(U, V), W?)
      +
      rec-value(Tid, sub(D?, E?)) .
      snd-msg(make-input, D) .
      rec-msg(make-input, D, U?) .
      snd-msg(make-input, E) .
      rec-msg(make-input, E, V?) .
      snd-msg(compute-lst, sub(U, V)) .
      rec-msg(compute-lst, sub(U, V), W?)
      +
      rec-value(Tid, mult(D?, E?)) .
      snd-msg(make-input, D) .
```

```
      rec-msg(make-input, D, U?) .
      snd-msg(make-input, E) .
      rec-msg(make-input, E, V?) .
      snd-msg(compute-lst, mult(U, V)) .
      rec-msg(compute-lst, mult(U, V), W?)
      +
      rec-value(Tid, div(D?, E?)) .
      snd-msg(make-input, D) .
      rec-msg(make-input, D, U?) .
      snd-msg(make-input, E) .
      rec-msg(make-input, E, V?) .
      snd-msg(compute-lst, div(U, V)) .
      rec-msg(compute-lst, div(U, V), W?)
      +
      rec-value(Tid, less(D?, E?)) .
      snd-msg(make-input, D) .
      rec-msg(make-input, D, U?) .
      snd-msg(make-input, E) .
      rec-msg(make-input, E, V?) .
      snd-msg(compute-lst, less(U, V)) .
      rec-msg(compute-lst, less(U, V), W?)
    ) .
    snd-eval(Tid, get-repres-dialog) .
    ( rec-value(Tid, none) .
      snd-msg(make-output, none(W)) .
      rec-msg(make-output, none(W), S?) .
      snd-do(Tid, display(S))
      +
      rec-value(Tid, dot) .
      snd-msg(make-output, dot(W)) .
      rec-msg(make-output, dot(W), S?) .
      snd-do(Tid, display(S))
      +
      rec-value(Tid, space) .
      snd-msg(make-output, space(W)) .
      rec-msg(make-output, space(W), S?) .
      snd-do(Tid, display(S))
      +
      rec-value(Tid, nfd) .
      snd-msg(make-output, nfd(W)) .
      rec-msg(make-output, nfd(W), S?) .
      snd-do(Tid, display(S))
    ) . snd-ack-event(Tid, button(calc-lst))
  endlet

process QUIT-BUTTON(Tid : ui) is
  rec-event(Tid, button(quit)) .
  snd-ack-event(Tid, button(quit)) .
  shutdown("Bye!")

process SDA-REC is
  let Tid : sda-rec, E : term, V : term
  in
    execute(sda-rec, Tid?) .
    ( rec-msg(compute-rec, E?) .
```

```
      snd-eval(Tid, expr(E)) .
      rec-value(Tid, V?) .
      snd-msg(compute-rec, E, V)
    ) * delta
  endlet

tool sda-rec is {command = "./sda-rec"}


process SDA-LST is
  let Tid : sda-lst, E : term, V : term
  in
     execute(sda-lst, Tid?) .
     ( rec-msg(compute-lst, E?) .
       snd-eval(Tid, expr(E)) .
       rec-value(Tid, V?) .
       snd-msg(compute-lst, E, V)
     ) * delta
  endlet

tool sda-lst is {command = "./sda-lst"}


process REPRES is
  let Tid : repres, E : str, V : term
  in
     execute(repres, Tid?) .
     ( rec-msg(make-input, E?) .
       snd-eval(Tid, input(E)) .
       rec-value(Tid, V?) .
       snd-msg(make-input, E, V)
       +
       rec-msg(make-output, V?) .
       snd-eval(Tid, output(V)) .
       rec-value(Tid, E?) .
       snd-msg(make-output, V, E)
     ) * delta
  endlet

tool repres is {command = "./repres"}


toolbus(UI, SDA-REC, SDA-LST, REPRES)
```

## D.2 Two-tool version

```
process UI is
  let Tid : ui
  in
     execute(ui, Tid?) .
     ( CALC-REC-BUTTON(Tid)
       +
       CALC-LST-BUTTON(Tid)
       +
       QUIT-BUTTON(Tid)
     ) * delta
  endlet

tool ui is {command = "wish-adapter -script ui-sda.tcl"}


process CALC-REC-BUTTON(Tid : ui) is
  let  D : str , E : str, U : term , V : term, W : term, S : str
  in
     rec-event(Tid, button(calc-rec)) .
     snd-eval(Tid, get-expr-dialog) .
     ( rec-value(Tid, add(D?, E?)) .
       snd-msg(make-input, D) .
       rec-msg(make-input, D, U?) .
       snd-msg(make-input, E) .
       rec-msg(make-input, E, V?) .
       snd-msg(compute-rec, add(U, V)) .
       rec-msg(compute-rec, add(U, V), W?)
       +
       rec-value(Tid, sub(D?, E?)) .
       snd-msg(make-input, D) .
       rec-msg(make-input, D, U?) .
       snd-msg(make-input, E) .
       rec-msg(make-input, E, V?) .
       snd-msg(compute-rec, sub(U, V)) .
       rec-msg(compute-rec, sub(U, V), W?)
       +
       rec-value(Tid, mult(D?, E?)) .
       snd-msg(make-input, D) .
       rec-msg(make-input, D, U?) .
       snd-msg(make-input, E) .
       rec-msg(make-input, E, V?) .
       snd-msg(compute-rec, mult(U, V)) .
       rec-msg(compute-rec, mult(U, V), W?)
       +
       rec-value(Tid, div(D?, E?)) .
       snd-msg(make-input, D) .
       rec-msg(make-input, D, U?) .
       snd-msg(make-input, E) .
       rec-msg(make-input, E, V?) .
       snd-msg(compute-rec, div(U, V)) .
       rec-msg(compute-rec, div(U, V), W?)
       +
       rec-value(Tid, less(D?, E?)) .
       snd-msg(make-input, D) .
```

```
      rec-msg(make-input, D, U?) .
      snd-msg(make-input, E) .
      rec-msg(make-input, E, V?) .
      snd-msg(compute-rec, less(U, V)) .
      rec-msg(compute-rec, less(U, V), W?)
     ) .
     snd-eval(Tid, get-repres-dialog) .
     ( rec-value(Tid, none) .
      snd-msg(make-output, none(W)) .
      rec-msg(make-output, none(W), S?) .
      snd-do(Tid, display(S))
      +
      rec-value(Tid, dot) .
      snd-msg(make-output, dot(W)) .
      rec-msg(make-output, dot(W), S?) .
      snd-do(Tid, display(S))
      +
      rec-value(Tid, space) .
      snd-msg(make-output, space(W)) .
      rec-msg(make-output, space(W), S?) .
      snd-do(Tid, display(S))
      +
      rec-value(Tid, nfd) .
      snd-msg(make-output, nfd(W)) .
      rec-msg(make-output, nfd(W), S?) .
      snd-do(Tid, display(S))
     ) . snd-ack-event(Tid, button(calc-rec))
  endlet

process CALC-LST-BUTTON(Tid : ui) is
  let  D : str , E : str, U : term , V : term, W : term, S : str
  in
     rec-event(Tid, button(calc-lst)) .
     snd-eval(Tid, get-expr-dialog).
     ( rec-value(Tid, add(D?, E?)) .
      snd-msg(make-input, D) .
      rec-msg(make-input, D, U?) .
      snd-msg(make-input, E) .
      rec-msg(make-input, E, V?) .
      snd-msg(compute-lst, add(U, V)) .
      rec-msg(compute-lst, add(U, V), W?)
      +
      rec-value(Tid, sub(D?, E?)) .
      snd-msg(make-input, D) .
      rec-msg(make-input, D, U?) .
      snd-msg(make-input, E) .
      rec-msg(make-input, E, V?) .
      snd-msg(compute-lst, sub(U, V)) .
      rec-msg(compute-lst, sub(U, V), W?)
      +
      rec-value(Tid, mult(D?, E?)) .
      snd-msg(make-input, D) .
      rec-msg(make-input, D, U?) .
      snd-msg(make-input, E) .
      rec-msg(make-input, E, V?) .
```

```
      snd-msg(compute-lst, mult(U, V)) .
      rec-msg(compute-lst, mult(U, V), W?)
      +
      rec-value(Tid, div(D?, E?)) .
      snd-msg(make-input, D) .
      rec-msg(make-input, D, U?) .
      snd-msg(make-input, E) .
      rec-msg(make-input, E, V?) .
      snd-msg(compute-lst, div(U, V)) .
      rec-msg(compute-lst, div(U, V), W?)
      +
      rec-value(Tid, less(D?, E?)) .
      snd-msg(make-input, D) .
      rec-msg(make-input, D, U?) .
      snd-msg(make-input, E) .
      rec-msg(make-input, E, V?) .
      snd-msg(compute-lst, less(U, V)) .
      rec-msg(compute-lst, less(U, V), W?)
    ) .
    snd-eval(Tid, get-repres-dialog) .
    ( rec-value(Tid, none) .
      snd-msg(make-output, none(W)) .
      rec-msg(make-output, none(W), S?) .
      snd-do(Tid, display(S))
      +
      rec-value(Tid, dot) .
      snd-msg(make-output, dot(W)) .
      rec-msg(make-output, dot(W), S?) .
      snd-do(Tid, display(S))
      +
      rec-value(Tid, space) .
      snd-msg(make-output, space(W)) .
      rec-msg(make-output, space(W), S?) .
      snd-do(Tid, display(S))
      +
      rec-value(Tid, nfd) .
      snd-msg(make-output, nfd(W)) .
      rec-msg(make-output, nfd(W), S?) .
      snd-do(Tid, display(S))
    ) . snd-ack-event(Tid, button(calc-lst))
  endlet

process QUIT-BUTTON(Tid : ui) is
  rec-event(Tid, button(quit)) .
  snd-ack-event(Tid, button(quit)) .
  shutdown("Bye!")

process CALC-REC is
  let Tid : calc-rec, E : term, K : term, L : term, V : term
  in
    execute(calc-rec, Tid?) .
    ( rec-msg(compute-rec, E?) .
      snd-eval(Tid, expr(E)) .
      ( rec-event(Tid, op(K?)) .
        snd-msg(list-op, K) .
```

```
        rec-msg(list-op, K, L?) .
        snd-ack-event(Tid, op(K, L))
      ) * rec-value(Tid, V?) .
          snd-msg(compute-rec, E, V)
    ) * delta
  endlet

tool calc-rec is {command = "./calc-rec"}


process CALC-LST is
  let Tid : calc-lst, E : term, K : term, L : term, V : term
  in
    execute(calc-lst, Tid?) .
    ( rec-msg(compute-lst, E?) .
      snd-eval(Tid, expr(E)) .
      ( rec-event(Tid, op(K?)) .
        snd-msg(list-op, K) .
        rec-msg(list-op, K, L?) .
        snd-ack-event(Tid, op(K, L))
      ) * rec-value(Tid, V?) .
          snd-msg(compute-lst, E, V)
    ) * delta
  endlet

tool calc-lst is {command = "./calc-lst"}


process DIGIT-LIST is
  let Tid : digit-list, E : term, V : term
  in
    execute(digit-list, Tid?) .
    ( rec-msg(list-op, E?) .
      snd-eval(Tid, expr(E)) .
      rec-value(Tid, V?) .
      snd-msg(list-op, E, V)
    ) * delta
  endlet

tool digit-list is {command = "./digit-list"}


process REPRES is
  let Tid : repres, E : str, V : term
  in
    execute(repres, Tid?) .
    ( rec-msg(make-input, E?) .
      snd-eval(Tid, input(E)) .
      rec-value(Tid, V?) .
      snd-msg(make-input, E, V)
      +
      rec-msg(make-output, V?) .
      snd-eval(Tid, output(V)) .
      rec-value(Tid, E?) .
      snd-msg(make-output, V, E)
```

```
    ) * delta
  endlet

tool repres is {command = "./repres"}


toolbus(UI, CALC-REC, CALC-LST, DIGIT-LIST, REPRES)
```

# E The C program adapted for the two-tool version

## E.1 event.c

```
#include "init.c"

term *result;

void initResult()
{
   result = NULL;
   TBprotect(&result);
}


term *S(term *dt)
{
  TBsend(TBmake("snd-event(op(S(%t)))", dt));
  TBreceive();
  return result;
}


term *P(term *dt)
{
  TBsend(TBmake("snd-event(op(P(%t)))", dt));
  TBreceive();
  return result;
}


T4 Z(term *dt)
{ T4 b;

  TBsend(TBmake("snd-event(op(Z(%t)))", dt));
  TBreceive();
  if (TBmatch(result, "%d", &b))
    return b;
  else {
    TBmsg("panic in event: %t\n", result);
    return M;
  }
}


T4 eq1_D4(term *dt1, term *dt2)
{ T4 b;

  TBsend(TBmake("snd-event(op(eq1D4(%t,%t)))", dt1, dt2));
  TBreceive();
  if (TBmatch(result, "%d", &b))
    return b;
  else {
    TBmsg("panic in event: %t\n", result);
    return M;
```

```
  }
}


T4 True_D4(term *dt)
{ T4 b;

  TBsend(TBmake("snd-event(op(TrueD4(%t)))", dt));
  TBreceive();
  if (TBmatch(result, "%d", &b))
    return b;
  else {
    TBmsg("panic in event: %t\n", result);
    return M;
  }
}


term *t2d(T4 b)
{
  TBsend(TBmake("snd-event(op(t2d(%d)))", b));
  TBreceive();
  return result;
}


T4 lt2(term *dt1, term *dt2)
{ T4 b;

  TBsend(TBmake("snd-event(op(lt2(%t,%t)))", dt1, dt2));
  TBreceive();
  if (TBmatch(result, "%d", &b))
    return b;
  else {
    TBmsg("panic in event: %t\n", result);
    return M;
  }
}


term *carry1(term *dt1, term *dt2)
{
  TBsend(TBmake("snd-event(op(carry1(%t,%t)))", dt1, dt2));
  TBreceive();
  return result;
}


term *borrow2(term *dt1, term *dt2)
{
  TBsend(TBmake("snd-event(op(borrow2(%t,%t)))", dt1, dt2));
  TBreceive();
  return result;
}
```

```
term *compl(term *dt)
{
  TBsend(TBmake("snd-event(op(compl(%t)))", dt));
  TBreceive();
  return result;
}


term *modsum1(term *dt1, term *dt2)
{
  TBsend(TBmake("snd-event(op(modsum1(%t,%t)))", dt1, dt2));
  TBreceive();
  return result;
}


term *modsub2(term *dt1, term *dt2)
{
  TBsend(TBmake("snd-event(op(modsub2(%t,%t)))", dt1, dt2));
  TBreceive();
  return result;
}


term *EMPTY(void)
{
  TBsend(TBmake("snd-event(op(EMPTY()))"));
  TBreceive();
  return result;
}


term *Ulist(term *dt)
{
  TBsend(TBmake("snd-event(op(Ulist(%t)))", dt));
  TBreceive();
  return result;
}


T4 True_LD4(term *y)
{ T4 b;

  TBsend(TBmake("snd-event(op(TrueLD4(%t)))", y));
  TBreceive();
  if (TBmatch(result, "%d", &b))
    return b;
  else {
    TBmsg("panic in event: %t\n", result);
    return M;
  }
}
```

```
term *Lappend(term *y1, term *y2)
{
  TBsend(TBmake("snd-event(op(Lappend(%t,%t)))", y1, y2));
  TBreceive();
  return result;
}


T4 E(term *y)
{ T4 b;

  TBsend(TBmake("snd-event(op(E(%t)))", y));
  TBreceive();
  if (TBmatch(result, "%d", &b))
    return b;
  else {
    TBmsg("panic in event: %t\n", result);
    return M;
  }
}


term *LeftHead(term *y)
{
  TBsend(TBmake("snd-event(op(LeftHead(%t)))", y));
  TBreceive();
  return result;
}


term *LRTail(term *y)
{
  TBsend(TBmake("snd-event(op(LRTail(%t)))", y));
  TBreceive();
  return result;
}


T4 eq1_LD4(term *y1, term *y2)
{ T4 b;

  TBsend(TBmake("snd-event(op(eq1LD4(%t,%t)))", y1, y2));
  TBreceive();
  if (TBmatch(result, "%d", &b))
    return b;
  else {
    TBmsg("panic in event: %t\n", result);
    return M;
  }
}


term *inv(term *y)
{
  TBsend(TBmake("snd-event(op(inv(%t)))", y));
```

```
  TBreceive();
  return result;
}


term *RightHead(term *y)
{
  TBsend(TBmake("snd-event(op(RightHead(%t)))", y));
  TBreceive();
  return result;
}


term *RLTail(term *y)
{
  TBsend(TBmake("snd-event(op(RLTail(%t)))", y));
  TBreceive();
  return result;
}


term *Rappend(term *y1, term *y2)
{
  TBsend(TBmake("snd-event(op(Rappend(%t,%t)))", y1, y2));
  TBreceive();
  return result;
}
```

## E.2 digit-list.tif.c

```
/* GENERATED AUTOMATICALLY, DO NOT MODIFY */
/* Tool Interface File for tool: digit-list */

#include "TB.h"

/* Prototypes for functions used in event handler */
term * expr(term *);
void   rec_terminate(term *);

/* Event handler for tool: digit-list */

term *digit_list_handler(term *e){
term *term_arg[1];

if(TBmatch(e, "rec-eval(expr(%t))", &term_arg[0])){
return expr(term_arg[0]);
} else if(TBmatch(e, "rec-terminate(%t)", &term_arg[0])){
rec_terminate(term_arg[0]);
return NULL;
}  else  {
TBprintf(stderr, "Ignored: %t\n", e);
return NULL;
}
}
```

```
/* Check input signature for tool: digit-list
 * returns: NULL (requested input signature is subset of tool's input signature);
 *          non-null term: first element not in tool's input signature.
 * NOTE:    In the signature all terms have as first argument <digit-list>, but this
 *          does NOT appear in the actual terms that will be sent to the tool.
 */

term *digit_list_check_in_sign(term_list *reqs){
char *in_sign[2];
int i;

in_sign[0] = "rec-eval(<digit-list>,expr(<term>))";
in_sign[1] = "rec-terminate(<digit-list>,<term>)";

for( ; reqs; reqs = next(reqs)){
for(i = 0; i < 2; i++){
if(TBmatch(first(reqs), in_sign[i])) goto found;
}
return first(reqs);
found:;
}
return NULL;
}
```

## E.3 digit-list.c

```
#include "digit-list.tif.c"
#include "LDigitDA.c"

term *expr(term *trm)
{ term *t, *t1, *t2;
  T4 b;

  if(!trm)
    return TBmake("snd-value(digit-list-error(%t))", trm);
  else if (TBmatch(trm, "S(%t)", &t))
    return TBmake("snd-value(%t)", S(t));
  else if (TBmatch(trm, "P(%t)", &t))
    return TBmake("snd-value(%t)", P(t));
  else if (TBmatch(trm, "Z(%t)", &t))
    return TBmake("snd-value(%t)", TBmake("%d", Z(t)));
  else if (TBmatch(trm, "eq1D4(%t, %t)", &t1, &t2))
    return TBmake("snd-value(%t)", TBmake("%d", eq1_D4(t1, t2)));
  else if (TBmatch(trm, "TrueD4(%t)", &t))
    return TBmake("snd-value(%t)", TBmake("%d", True_D4(t)));
  else if (TBmatch(trm, "t2d(%d)", &b))
    return TBmake("snd-value(%t)", t2d(b));
  else if (TBmatch(trm, "lt2(%t, %t)", &t1, &t2))
    return TBmake("snd-value(%t)", TBmake("%d", lt2(t1, t2)));
  else if (TBmatch(trm, "carry1(%t,%t)", &t1, &t2))
    return TBmake("snd-value(%t)", carry1(t1, t2));
  else if (TBmatch(trm, "borrow2(%t,%t)", &t1, &t2))
    return TBmake("snd-value(%t)", borrow2(t1, t2));
  else if (TBmatch(trm, "compl(%t)", &t))
    return TBmake("snd-value(%t)", compl(t));
```

```
  else if (TBmatch(trm, "modsum1(%t,%t)", &t1, &t2))
    return TBmake("snd-value(%t)", modsum1(t1, t2));
  else if (TBmatch(trm, "modsub2(%t,%t)", &t1, &t2))
    return TBmake("snd-value(%t)", modsub2(t1, t2));
  else if (TBmatch(trm, "EMPTY()"))
    return TBmake("snd-value(%t)", EMPTY());
  else if (TBmatch(trm, "Ulist(%t)", &t))
    return TBmake("snd-value(%t)", Ulist(t));
  else if (TBmatch(trm, "TrueLD4(%t)", &t))
    return TBmake("snd-value(%t)", TBmake("%d", True_LD4(t)));
  else if (TBmatch(trm, "Lappend(%t,%t)", &t1, &t2))
    return TBmake("snd-value(%t)", Lappend(t1, t2));
  else if (TBmatch(trm, "E(%t)", &t))
    return TBmake("snd-value(%t)", TBmake("%d", E(t)));
  else if (TBmatch(trm, "LeftHead(%t)", &t))
    return TBmake("snd-value(%t)", LeftHead(t));
  else if (TBmatch(trm, "LRTail(%t)", &t))
    return TBmake("snd-value(%t)", LRTail(t));
  else if (TBmatch(trm, "eq1LD4(%t, %t)", &t1, &t2))
    return TBmake("snd-value(%t)", TBmake("%d", eq1_LD4(t1, t2)));
  else if (TBmatch(trm, "inv(%t)", &t))
    return TBmake("snd-value(%t)", inv(t));
  else if (TBmatch(trm, "RightHead(%t)", &t))
    return TBmake("snd-value(%t)", RightHead(t));
  else if (TBmatch(trm, "RLTail(%t)", &t))
    return TBmake("snd-value(%t)", RLTail(t));
  else if (TBmatch(trm, "Rappend(%t,%t)", &t1, &t2))
    return TBmake("snd-value(%t)", Rappend(t1, t2));
  else return TBmake("snd-value(digit-list-error(%t))", trm);
}

void rec_terminate(term *t)
{
  exit(0);
}

void main(int argc, char *argv[])
{
 TBinit("digit_list", argc, argv, digit_list_handler, digit_list_check_in_sign);

 init();
 initDigitDA();

 TBeventloop();
}
```

## E.4 Nat1DA.c

```
#include "event.c"
#include "initDigitDA.c"

term *rlz(term *n)
{ term *t;

  TBprotect(&n);
```

```
  t = cond_trm(n, or1(E(n), not(Z(LeftHead(n)))), rlz(LRTail(n)));
  TBunprotect(&n);
  return t;
}

term *lS(term *n)
{ term *t;

  TBprotect(&n);
  t = rlz(cond_trm(Ulist(d[1]),
                   E(n),
                   cond_trm(Lappend(lS(RLTail(n)), Ulist(d[0])),
                            eq1_D4(RightHead(n), base),
                            Lappend(RLTail(n), Ulist(S(RightHead(n)))))));
  TBunprotect(&n);
  return t;
}

term *lP(term *n)
{ term *t;

  TBprotect(&n);
  t = rlz(cond_trm(Lappend(lP(RLTail(n)), Ulist(base)),
                   eq1_D4(RightHead(n), d[0]),
                   Lappend(RLTail(n), Ulist(P(RightHead(n))))));
  TBunprotect(&n);
  return t;
}

T4 lZ(term *n)
{ T4 b;

  TBprotect(&n);
  b = E(rlz(n));
  TBunprotect(&n);
  return b;
}

term *length(term *n)
{ term *t;

  TBprotect(&n);
  t = cond_trm(EMPTY(), E(n), lS(length(LRTail(n))));
  TBunprotect(&n);
  return t;
}
```

## E.5 Nat2DA.c

```
#include "Nat1DA.c"

T4 less2(term *n1, term *n2)
{ T4 b;

  TBprotect(&n1);
```

```
  TBprotect(&n2);
  b = cond(F, lZ(n2), cond(T, lZ(n1), less2(lP(n1), lP(n2))));
  TBunprotect(&n1);
  TBunprotect(&n2);
  return b;
}

term *add2(term *n1, term *n2)
{ term *t;

  TBprotect(&n1);
  TBprotect(&n2);
  t = cond_trm(rlz(n1), lZ(n2), lS(add2(n1, lP(n2))));
  TBunprotect(&n1);
  TBunprotect(&n2);
  return t;
}

term *sub2(term *n1, term *n2)
{ term *t;

  TBprotect(&n1);
  TBprotect(&n2);
  t = cond_trm(rlz(n1), lZ(n2), lP(sub2(n1, lP(n2))));
  TBunprotect(&n1);
  TBunprotect(&n2);
  return t;
}

term *mult2(term *n1, term *n2)
{ term *t;

  TBprotect(&n1);
  TBprotect(&n2);
  t = cond_trm(EMPTY(), lZ(n2), add2(n1, mult2(n1, lP(n2))));
  TBunprotect(&n1);
  TBunprotect(&n2);
  return t;
}

term *div2(term *n1, term *n2)
{ term *t;

  TBprotect(&n1);
  TBprotect(&n2);
  t = cond_trm(Mtrm,
               lZ(n2),
               cond_trm(EMPTY(), less2(n1, n2), lS(div2(sub2(n1, n2), n2))));
  TBunprotect(&n1);
  TBunprotect(&n2);
  return t;
}
```

## E.6 Nat3DA.c

```
#include "Nat1DA.c"

T4 Dless2(term *n1, term *n2)
{ T4 b;

  TBprotect(&n1);
  TBprotect(&n2);
  b = cond(F, E(n2), cond(Dless2(LRTail(n1), LRTail(n2)),
                          eq1_D4(LeftHead(n2), LeftHead(n1)),
                          lt2(LeftHead(n1), LeftHead(n2))));
  TBunprotect(&n1);
  TBunprotect(&n2);
  return b;
}

T4 less2(term *n1, term *n2)
{ T4 b;

  TBprotect(&n1);
  TBprotect(&n2);
  b = cond(F, lZ(n2), cond(T,
                           lZ(n1),
                           cond(Dless2(rlz(n1), rlz(n2)),
                                eq1_LD4(length(rlz(n1)), length(rlz(n2))),
                                less2(length(rlz(n1)), length(rlz(n2))))));
  TBunprotect(&n1);
  TBunprotect(&n2);
  return b;
}

term *add2(term *n1, term *n2)
{ term *t;

  TBprotect(&n1);
  TBprotect(&n2);
  t = cond_trm(rlz(n1),
               lZ(n2),
               cond_trm(rlz(n2),
                        lZ(n1),
                        Lappend(add2(add2(RLTail(n1), RLTail(n2)),
                                     Ulist(carry1(RightHead(n1),
                                                  RightHead(n2)))),
                                Ulist(modsum1(RightHead(n1),
                                              RightHead(n2))))));
  TBunprotect(&n1);
  TBunprotect(&n2);
  return t;
}

term *sub2(term *n1, term *n2)
{ term *t;

  TBprotect(&n1);
  TBprotect(&n2);
```

```
  t = cond_trm(rlz(n1),
               lZ(n2),
               cond_trm(Mtrm,
                        lZ(n1),
                        Lappend(sub2(sub2(RLTail(n1), RLTail(n2)),
                                     Ulist(borrow2(RightHead(n1),
                                                   RightHead(n2)))),
                                Ulist(modsub2(RightHead(n1),
                                              RightHead(n2))))));
  TBunprotect(&n1);
  TBunprotect(&n2);
  return t;
}

term *Dmult2(term *n, term *dt)
{ term *t;

  TBprotect(&n);
  TBprotect(&dt);
  t = cond_trm(EMPTY(), Z(dt), add2(n, Dmult2(n, P(dt))));
  TBunprotect(&n);
  TBunprotect(&dt);
  return t;
}

term *mult2(term *n1, term *n2)
{ term *t;

  TBprotect(&n1);
  TBprotect(&n2);
  t = cond_trm(EMPTY(),
               or2(lZ(n1), lZ(n2)),
               add2(Lappend(mult2(n1, RLTail(n2)), Ulist(d[0])),
                    Dmult2(n1, RightHead(n2))));
  TBunprotect(&n1);
  TBunprotect(&n2);
  return t;
}

term *Ddiv2(term *n1, term *n2)
{ term *t;

  TBprotect(&n1);
  TBprotect(&n2);
  t = cond_trm(Mtrm,
               lZ(n2),
               cond_trm(d[0], less2(n1, n2), S(Ddiv2(sub2(n1, n2), n2))));
  TBunprotect(&n1);
  TBunprotect(&n2);
  return t;
}

term *fill2(term *n1, term *n2)
{ term *t;
```

```
  TBprotect(&n1);
  TBprotect(&n2);
  t = cond_trm(Mtrm, lZ(n2), cond_trm(rlz(n2),
                                      less2(n1, n2),
                                      fill2(n1, Lappend(n2, Ulist(d[0])))));
  TBunprotect(&n1);
  TBunprotect(&n2);
  return t;
}

term *dr2(term *n1, term *n2, term *n3)
{ term *t;

  TBprotect(&n1);
  TBprotect(&n2);
  TBprotect(&n3);
  t = cond_trm(EMPTY(),
               less2(n3, n2),
               Lappend(Ulist(Ddiv2(n1, n3)),
                       dr2(sub2(n1, Dmult2(n3, Ddiv2(n1, n3))),
                           n2,
                           RLTail(n3))));
  TBunprotect(&n1);
  TBunprotect(&n2);
  TBunprotect(&n3);
  return t;
}

term *div2(term *n1, term *n2)
{ term *t;

  TBprotect(&n1);
  TBprotect(&n2);
  t = dr2(n1, n2, RLTail(fill2(n1, n2)));
  TBunprotect(&n1);
  TBunprotect(&n2);
  return t;
}
```

## E.7 calc-rec.tif.c

```
/* GENERATED AUTOMATICALLY, DO NOT MODIFY */
/* Tool Interface File for tool: calc-rec */

#include "TB.h"

/* Prototypes for functions used in event handler */
term * expr(term *);
void   rec_ack_event(term *);
void   rec_terminate(term *);

/* Event handler for tool: calc-rec */

term *calc_rec_handler(term *e){
term *term_arg[1];
```

```
if(TBmatch(e, "rec-eval(expr(%t))", &term_arg[0])){
return expr(term_arg[0]);
} else if(TBmatch(e, "rec-ack-event(%t)", &term_arg[0])){
rec_ack_event(term_arg[0]);
return NULL;
} else if(TBmatch(e, "rec-terminate(%t)", &term_arg[0])){
rec_terminate(term_arg[0]);
return NULL;
}  else  {
TBprintf(stderr, "Ignored: %t\n", e);
return NULL;
}
}

/* Check input signature for tool: calc-rec
 * returns: NULL (requested input signature is subset of tool's input signature);
 *          non-null term: first element not in tool's input signature.
 * NOTE:    In the signature all terms have as first argument <calc-rec>, but this
 *          does NOT appear in the actual terms that will be sent to the tool.
 */

term *calc_rec_check_in_sign(term_list *reqs){
char *in_sign[3];
int i;

in_sign[0] = "rec-eval(<calc-rec>,expr(<term>))";
in_sign[1] = "rec-ack-event(<calc-rec>,<term>)";
in_sign[2] = "rec-terminate(<calc-rec>,<term>)";

for( ; reqs; reqs = next(reqs)){
for(i = 0; i < 3; i++){
if(TBmatch(first(reqs), in_sign[i])) goto found;
}
return first(reqs);
found:;
}
return NULL;
}
```

## E.8 calc-rec.c

```
#include "calc-rec.tif.c"
#include "Nat2DA.c"

term *expr(term *trm)
{ term *t, *t1 = NULL, *t2 = NULL;

  TBprotect(&trm);
  TBprotect(&t1);
  TBprotect(&t2);
  if (TBmatch(trm, "add(%t, %t)", &t1, &t2))
    t = TBmake("snd-value(%t)", add2(t1, t2));
  else if (TBmatch(trm, "sub(%t, %t)", &t1, &t2))
    t = TBmake("snd-value(%t)", sub2(t1, t2));
```

```
  else if (TBmatch(trm, "mult(%t, %t)", &t1, &t2))
    t = TBmake("snd-value(%t)", mult2(t1, t2));
  else if (TBmatch(trm, "div(%t, %t)", &t1, &t2))
    t = TBmake("snd-value(%t)", div2(t1, t2));
  else if (TBmatch(trm, "less(%t, %t)", &t1, &t2))
    t = TBmake("snd-value(%t)", TBmake("%d", less2(t1, t2)));
  else t = TBmake("snd-value(calc-error(%t))", trm);
  TBunprotect(&trm);
  TBunprotect(&t1);
  TBunprotect(&t2);
  return t;
}

void rec_ack_event(term *trm)
{ term *t1, *t2;

  if (TBmatch(trm, "op(%t,%t)", &t1, &t2))
    result = t2;
  else TBprintf(stderr, "calc-error: %t\n", trm);
}

void rec_terminate(term *t)
{
  exit(0);
}

void main(int argc, char *argv[])
{
  TBinit("calc_rec", argc, argv, calc_rec_handler, calc_rec_check_in_sign);

  init();
  initDigitDA();
  initResult();

  TBeventloop();
}
```

## E.9 calc-lst.tif.c

```
/* GENERATED AUTOMATICALLY, DO NOT MODIFY */
/* Tool Interface File for tool: calc-lst */

#include "TB.h"

/* Prototypes for functions used in event handler */
term * expr(term *);
void   rec_ack_event(term *);
void   rec_terminate(term *);

/* Event handler for tool: calc-lst */

term *calc_lst_handler(term *e){
term *term_arg[1];

if(TBmatch(e, "rec-eval(expr(%t))", &term_arg[0])){
```

```
return expr(term_arg[0]);
} else if(TBmatch(e, "rec-ack-event(%t)", &term_arg[0])){
rec_ack_event(term_arg[0]);
return NULL;
} else if(TBmatch(e, "rec-terminate(%t)", &term_arg[0])){
rec_terminate(term_arg[0]);
return NULL;
}  else  {
TBprintf(stderr, "Ignored: %t\n", e);
return NULL;
}
}

/* Check input signature for tool: calc-lst
 * returns: NULL (requested input signature is subset of tool's input signature);
 *          non-null term: first element not in tool's input signature.
 * NOTE:    In the signature all terms have as first argument <calc-lst>, but this
 *          does NOT appear in the actual terms that will be sent to the tool.
 */

term *calc_lst_check_in_sign(term_list *reqs){
char *in_sign[3];
int i;

in_sign[0] = "rec-eval(<calc-lst>,expr(<term>))";
in_sign[1] = "rec-ack-event(<calc-lst>,<term>)";
in_sign[2] = "rec-terminate(<calc-lst>,<term>)";

for( ; reqs; reqs = next(reqs)){
for(i = 0; i < 3; i++){
if(TBmatch(first(reqs), in_sign[i])) goto found;
}
return first(reqs);
found:;
}
return NULL;
}
```

## E.10 calc-lst.c

```
#include "calc-lst.tif.c"
#include "Nat3DA.c"

term *expr(term *trm)
{ term *t, *t1 = NULL, *t2 = NULL;

  TBprotect(&trm);
  TBprotect(&t1);
  TBprotect(&t2);
  if (TBmatch(trm, "add(%t, %t)", &t1, &t2))
    t = TBmake("snd-value(%t)", add2(t1, t2));
  else if (TBmatch(trm, "sub(%t, %t)", &t1, &t2))
    t = TBmake("snd-value(%t)", sub2(t1, t2));
  else if (TBmatch(trm, "mult(%t, %t)", &t1, &t2))
    t = TBmake("snd-value(%t)", mult2(t1, t2));
```

```
  else if (TBmatch(trm, "div(%t, %t)", &t1, &t2))
    t = TBmake("snd-value(%t)", div2(t1, t2));
  else if (TBmatch(trm, "less(%t, %t)", &t1, &t2))
    t = TBmake("snd-value(%t)", TBmake("%d", less2(t1, t2)));
  else t = TBmake("snd-value(calc-error(%t))", trm);
  TBunprotect(&trm);
  TBunprotect(&t1);
  TBunprotect(&t2);
  return t;
}

void rec_ack_event(term *trm)
{ term *t1, *t2;

  if (TBmatch(trm, "op(%t,%t)", &t1, &t2))
    result = t2;
  else TBprintf(stderr, "calc-error: %t\n", trm);
}

void rec_terminate(term *t)
{
  exit(0);
}

void main(int argc, char *argv[])
{
  TBinit("calc_lst", argc, argv, calc_lst_handler, calc_lst_check_in_sign);

  init();
  initDigitDA();
  initResult();

  TBeventloop();
}
```