The NUT Language Report[1]

**Tarmo Uustalu, Urmas Kopra, Vahur Kotkas,
Michail Matskin, Enn Tyugu**

**Research Report TRITA-IT R 94:14
Dept of Teleinformatics, The Royal Institute of Technology, Stockholm**

**June 1994
Revised December 1996**

Development of the NUT system is carried out in the framework of the project ”Knowledge-Based Software Tools” financed by the Swedish National Board for Industrial and Technical Development (NUTEK) under grant 9303405-2

[1] This document is available by anonymous ftp from `it.kth.se`, file `labs/se/Software/NUT/doc/language.ps.Z`

Contents

1.0 Introduction 1
1.1 Scope and purpose of this document 1
1.2 Assumptions about the reader 1
1.3 Notational conventions 1
1.4 About the NUT system 1
1.5 Other documentation on NUT 2
2.0 NUT packages 4
3.0 Lexical level 5
3.1 Comments 5
3.2 Keywords, operators, separators 5
3.3 Identifiers 5
3.4 Numbers 6
3.5 Strings 7
4.0 Objects and classes 8
4.1 Classes 8
4.2 Predefined classes 9
4.2.1 num 9
4.2.2 bool 9
4.2.3 text 9
4.2.4 prog 10
4.2.5 any 10
4.2.6 array of cl 10
4.2.7 struct 10
4.2.8 row 10
4.3 User classes 10
4.4 Naming of objects 12
5.0 User class texts 14
5.1 super-sections 14
5.2 var- and vir-sections 15
5.3 rel-sections 16
5.4 alias-sections 19
5.5 init-sections 20
5.6 Component declaration prototypes 20
6.0 Procedure texts 23
6.1 Specifications (axioms) 23
6.2 Object contexts of programs 25
6.3 Programs 25
6.3.1 Assignments 26
6.3.2 Procedure calls 26
6.3.3 Subtask calls 29
6.3.4 compute-call 30
6.3.5 produce-call 30
6.3.6 Sequences and the empty statement 31
6.3.7 if-statements 31
6.3.8 for-statements 32
6.3.9 do-statements 32
6.3.10 exit-statements 32
6.4 Expressions 33

6.4.1 Constants....................34
6.4.2 Operator expressions....................34
6.4.3 Function calls....................35
6.4.4 Structure expressions....................35
6.4.5 new-expressions....................36

7.0 Synthesis....................37

1.0 Introduction

1.1 Scope and purpose of this document

This document is a corrected, updated, revised, and extended version of Technical Report TRITA-TCS-SE-9212 *"The NUT Language"*, and replaces it.

NUT is a knowledge-based programming environment, consisting of a windows-based interactive user interface, a language processor, and graphics facilities. The present document is intended as a reference manual of the NUT language for the users of the NUT system. It corresponds to the language of the NUT version 3.0 developed at the Royal Institute of Technology. For the principles and ideas behind the NUT system, and for other documentation of the system, we refer elsewhere (see Sec 1.4).

1.2 Assumptions about the reader

The reader is assumed to be acquainted with the principles of object-oriented (OO) programming, and, recommendably, some OO language (e.g. Smalltalk, C++). Comprehension of Backus-Naur style formal syntactic definitions is required. Some knowledge of logic is beneficial for understanding the automatic synthesis mechanism implemented in the system, but not strictly necessary.

This document is not a crash course in NUT and not even in the NUT language. It is assumed that the reader is somewhat acquainted with both the system in general and with the language in particular. The best document to start with is *"The NUT System"* by Enn Tyugu (see Sec 1.4).

1.3 Notational conventions

Throughout this document, formal syntactic definitions use angle brackets < and > to denote syntactic variables, square brackets [and] to enclose optional items, symbol | to separate alternative items, and ellipses ... to indicate that the preceding item may be repeated any number of times. The metaseparator ::= is a definition symbol.

1.4 About the NUT system

The NUT programming language is based on two paradigms: procedural object-oriented programming and automatic synthesis of programs from declarative specifications. The latter is a technique for automatic construction of programs for unprogrammed procedures out of their specifications and of the programs and specifications of programmed procedures, where a procedure's specification embodies its external view (states the names of its input and output parameters). Automatic synthesis of programs, as practiced in NUT, is based on proof search in intuitionistic propositional logic. It is possible in the language to state constraints on variables in terms of arithmetic equations. Thanks to an equation solver built into the language processor, the system is able to interpret arithmetical equations as multi-way procedures for computing the unknown variable of the equation.

The NUT graphics facilities include the Graphics Editor, a set of graphics functions in the language, and the Scheme Editor. The latter is a tool for visual programming that allows the user to define classes by means of graphical schemes.

The development of the NUT system began and its first versions were implemented by a group of researchers and programmers lead by Prof Enn Tyugu and Dr Michail Matskin at Software Dept. of Institute of Cybernetics, Estonian Academy of Sciences (Tallinn). At the present moment, the system is being further developed by the Software Engineering Group at Dept. of Teleinformatics, The Royal Institute of Technology (Stockholm) in co-operation with the Tallinn group.

The NUT system runs under X Windows System, Version 11, Release 5, and requires the Xaw library (the Athena widget set), and the standard X11 bitmaps.

An installation of the NUT system version 3.0 (executables, libraries, demos, manpage, documentation) for Sun4 machines running SunOS 4.1 is available by anonymous ftp from **it.kth.se**, file **labs/se/Software/NUT/v3.0.tar.Z**.

1.5 Other documentation on NUT

Journal papers on NUT:

- Enn Tyugu. *Knowledge-Based Programming Environments.* Knowledge-Based Systems, 1991, 4(1):4-15.
- Enn Tyugu. *Three New-Generation Software Environments.* Comm. ACM, 1991, 34(6):46-59.
- Enn Tyugu, Michail Matskin, Jaan Penjam, Peep Eomois. *NUT: An Object-Oriented Language.* Computers and Artificial Intelligence, 1986, 5(6):521-542.

Documents on NUT:

- Enn Tyugu. *The NUT system.* June 1994. Available by anonymous ftp from **it.kth.se**, file **labs/se/Software/NUT/doc/syst.ps.Z**.
- Benjamin Volozh, Mari Kōpp, Enn Tyugu. *The NUT Graphics.* Technical Report TRITA-IT-R 93:05, Dept. of Teleinformatics, The Royal Institute of Technology, June 1993. Available by anonymous ftp from **it.kth.se**, file **Reports/TELEINFORMATICS/TRITA-IT-9305.ps.Z**.
- Benjamin Volozh. *Appendix to The NUT Graphics.* March 1994. Available by anonymous ftp from **it.kth.se**, file **labs/se/Software/NUT/doc/graphics-new.ps.Z**.
- *The NUT libraries.* June 1994. Available by anonymous ftp from **it.kth.se**, file **labs/se/Software/NUT/doc/libraries.ps.Z**.
- *Interoperability of NUT with C and UNIX.* June 1994. Available by anonymous ftp from **it.kth.se**, file **labs/se/Software/NUT/doc/interoperab.ps.Z**.
- Bo Andersson, Benjamin Volozh. *User interface of NUT.* June 94. Available by anonymous ftp from **it.kth.se**, file **lab/se/Software/NUT/doc/interface.ps.Z**.

Selected papers on automatic synthesis (structural synthesis of programs):

- Grigori Mints, Enn Tyugu. *Justification of the Structural Synthesis of Programs.* Science of Computer Programming, 1982, 2(3):215-240.
- Grigori Mints. *Propositional Logic Programming.* In: J. E. Hayes, D. Michie, E. Tyugu, eds., Machine Intelligence, Vol. 12, pp 17-37. Clarendon Press, Oxford, 1991.

2.0 NUT packages

Applications written in NUT are arranged into packages.

The text of a package in the NUT language consists of the following:

- one or several main programs,
- the texts of the user-defined classes of the given package.

Main programs are written in the main NUT window. Texts of user-defined classes are written in special class text windows, named after the classes.

3.0 Lexical level

In the NUT language, there are the following kinds of lexical tokens:

- keywords,
- operators,
- separators,
- identifiers,
- numbers,
- strings.

Newline characters and blanks are ignored except as token separators and in strings.

3.1 Comments

Comments begin with a percentage character % and extend to the end of a line or to a next % character. They are ignored completely by the NUT parser.

3.2 Keywords, operators, separators

The following is a complete list of keywords:

```
<keyword> ::=
      nil | true | false
      | super | var | vir | rel | alias | rule | init
      | num | text | bool | prog | any | array | of | all
      | new | spec | c_fun | compute | produce | subtask
      | if | fi | for | step | to | do | od | exit
      | next | curr
```

The following is a complete list of operators:

неравно

```
+   -   *   /   ^   >   <   >=   <=   ==   /=   ~   |   &
```

равно   или   и

The following is a complete list of separators:

```
(  )  [  ]  {  }  .  ,  :  :=  <-  |-  ->  --  ||  ..  #
<blank> <newline>
```

3.3 Identifiers

Identifiers are sequences of letters, digits and underscore symbols beginning with a letter. They are used for naming objects, classes, relations (methods) etc.:

```
<identifier> ::=
      <letter>[<letter>|<digit>|_]...

<digit> ::=
      0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

```
<letter>::=
        a | b | c | d | e | f | g | h | i | j | k | l | m
        | n | o | p | q | r | s | t | u | v | w | x | y | z
        | A | B | C | D | E | F | G | H | I | J | K | L | M
        | N | O | P | Q | R | S | T | U | V | W | X | Y | Z
```

Notes. 1. Keywords should not be used as identifiers (although such usage is, in fact, legal in certain constructions).
2. Standard function names are identifiers that have a certain predefined meaning, and should not be redefined (although this is legal in certain constructions).
2. Identifiers are case-sensitive.
3. Only the first 15 symbols of identifiers are significant.

Examples:

point1, penSize, Double_integral, a1_side

The following is a list of standard function names:

```
<std-function-name> ::=
        sin | cos | tan | asin | acos | atan
        | sqrt | ln | exp | abs | int | mod
        | pos | copy | delete | insert
        | chin | rech | length
        | add_elem | add_pict| del_elem | del_pict
        | reshow | reshow_all | save_elem | save_pict
        | link_name | link_pict | get_ID
        | get_line | get_poly | get_rect | get_oval | get_text
        | get_group | get_type | get_frame | get_name | get_status
        | gr_line | gr_poly | gr_rect | gr_oval | gr_text
        | gr_group | put_frame | put_name
```

3.4 Numbers

Numbers are defined in the usual way:

```
<number> ::=
        <integer> | <real>

<integer> ::=
        [<sign>]<unsigned-integer>

<sign> ::=
        + | -

<unsigned-integer> ::=
        <digit>[<digit>]...

<real> ::=
        <integer>[.[<unsigned-integer>]][<exp-symbol><integer>]
        | [<integer>].<unsigned-integer>[<exp-symbol><integer>]
```

```
<exp-symbol> ::=
        E | e
```

Integers with absolute value greater than 16383 are treated as reals. The highest allowed absolute value for an exponent is 307[1].

Examples:

```
0, 15, 1328
5.1, .28, 12., 0.1e-1, .3E2, 20584
```

3.5 Strings

Strings consist of any sequence of characters surrounded by single quotes (').

```
<string> ::=
        '[<symbol>]...'

<symbol> ::=
        <digit> | <letter>
        | ! | " | # | $ | % | & | ' | ( | ) | * | + | , | - | . | / | : | ; | < | = | > | ?
        | @ | [ | \ | ] | ^ | _ | ` | { | } | ~ | | | <blank> | <newline>
```

The single quote ' is represented in a string by the sequence of two quotes ''. Observe that strings may contain newlines.

Examples:

```
'hello world'
'This is a text'
```

[1] Values of real numbers are restricted as follows: 0.00000000001e-307 < abs(real) < 18.0e307.

4.0 Objects and classes

The notions of object and class lie in the ground of all OO languages.

An object is an entity that possesses an identity, a class, and a value (state). The value of an object can change in time.

Objects can be scalar and compound. The value of a compound object is a structure formed of component objects.

In the sequel, components of an object, components of components, components of components of components, etc., are often called first-level, second-level, third-level etc. components of the object. All these are also called deep components of the object.

A class is a template for creation of objects with similar properties. Such objects are called instances of the class. An object's class determines the following properties of the object:

- the object's domain of values (if the object is scalar), the names and classes of the components (if the object is compound),
- the default (initial) value of the object,
- and the relations (methods) performable on the object.

4.1 Classes

Classes are divided into predefined and user-defined classes. The predefined classes are: the primitive classes, the universal class **any**, the polymorphic classes **array**, **struct**, and **row**. They are described in Sec 4.2. User-defined classes (in short, user classes) are defined by the user in special class definition windows, and they are local to the package. The general properties of user classes are described in Sec 4.3. How to define user classes is explained in Sec 5.0.

Classes can also be divided into scalar and compound classes. The primitive classes are scalar. The classes **array**, **struct**, **row**, and all user classes are compound classes.

All classes, except for classes **struct** and **row** (see Sec 5.2.7, 5.2.8), are named. Syntactically, class names are defined as follows.

```
<class-name> ::=
        <nonarrayclass-name> | <arrayclass-name>

<nonarrayclass-name> ::=
        <primitiveclass-name> | <userclass-name> | any

<primitiveclass-name> :=
        num | bool | text | prog

<arrayclass-name> ::=
        array [<unsigned-integer>] of <nonarrayclass-name>

<userclass-name> ::=
        <identifier>
```

The unsigned integer in an array class name tells the initial length of instances of the given array class.

Classes **struct** or **row** cannot be mentioned in texts in the NUT language. No object can be explicitly declared to have class **struct** or **row**. For closer comments, see Sec 5.2.7, 5.2.8.

Instances of all classes can take value **nil**, representing undefinedness.

On values of all classes, the following operations are predefined: test for identicality, test for non-identicality (implemented as operators, see Sec 6.4.2), and **length** (implemented as a standard function, see Sec 8.4).

A number of functions applicable to object of different classes (e.g. **NutPrint**, **c_getclass**, **self**) are available in the language as C-functions, see separate document *The NUT Libraries*.

4.2 Predefined classes

The predefined classes are: the four primitive classes, the universal class **any**, and the polymorphic classes **array**, **struct**, and **row**.

The domains of values for instances of different predefined classes, as well as basic operations on these domains are described below. The default value for instances of primitive classes and of class **any** is **nil**. No relations are defined on instances of predefined classes.

4.2.1 num

num is a class of numbers. Any number is a **num** value. The basic operations on numeric values are: elementary arithmetical operations and comparison operations (implemented as operators, see Sec 6.4.2), rounding, mod, absolute value, square root, logarithm, exponentiation, trigonometrical functions (implemented as standard functions, see Sec 8.1). Comparison operations yield boolean values.

4.2.2 bool

bool is a class of booleans. **true** and **false** are the two boolean values. Internally, **true** is represented as number **1**, and **false** is represented as number **0**. Therefore, all **bool** objects are internally represented as **num** objects. However, if an object is explicitly declared to have class **bool**, it cannot be used as a **num** object. The basic operations on boolean values are: negation, conjunction, and disjunction (implemented as operators, see Sec 6.4.2).

4.2.3 text

text is a class of strings. Any string is a **text** value. The basic operations on textual values are the usual string-manipulation operations (implemented as standard functions, see Sec 8.2).

4.2.4 prog

prog is a class of procedures. Any programmed procedure (in the NUT language) is a **prog** value. Procedure calling can be regarded as an operation on **prog** objects. Procedures and procedure calls are explained in Sec's 6.0 and 6.3.2, respectively.

4.2.5 any

any is a universal class. The concrete class of an object declared to have class **any** is initially open. It will be determined dynamically when the object (or any of its components) first acquires a non-**nil** value. Arbitrary values are allowed. Since the object may obtain arbitrary class, no assumptions are made about its number of components or their names.

In a sense, the only legal value of class **any** is the constant **nil**, since as soon as a value of an object having class **any** becomes non-**nil**, the object's class is concretized, and it ceases to have class **any**.

4.2.6 array of *cl*

array of *cl* is a class of values that are structures of objects having class *cl*. The initial length for an object having class **array of** *cl* is picked from the class name in the declaration of the object, if indicated there, and is 0 otherwise. During the object's life-time, the length is dynamically extended, and more memory is allocated, if components with higher index than the length are referenced. Dynamic memory allocation is time-consuming and a potential cause for fragmentation of memory. Therefore, explicit initialization of length should be preferred, where possible.

Two-dimensional arrays can be modelled by an arrays of arrays (must be declared as having class **array of any**).

4.2.7 struct

struct is a polymorphic class of values that are structures of objects of arbitrary classes. This class is hidden: no object can be explicitly declared to be of this class; in fact, this class cannot even be mentioned by name in the language. However, group aliases (see Sec 5.4) are of that class, and certain kinds of expressions evaluate to **struct** values (procedure calls, structure expressions, see Sec 6.3.2, 6.4.4).

4.2.8 row

row is a class of values that are structures of objects of some fixed class (and is thus similar to class **array**). This class is hidden: no object can be explicitly declared to be of this class. However, user class instances can have row components (at most one, see Sec 4.3), and row components are always of class **row**. The length of an object's row is derived from the text of the object's class. It is either the length bound given in the declaration of the row, or the maximum over the row element indices mentioned in the class text.

4.3 User classes

User classes are compound classes.

Components of instances of user classes fall into four categories:

- proper components (instance variables),
- virtual components (virtual variables),
- aliases,
- rows (an object can have at most one row; the row does not have a name)

The names and classes (and, optionally, extra properties) of the proper and virtual components of an object are given in the text of the object's class in the form of var- and vir-declarations (see Sec 5.2, 5.6). Besides, any simple name that appears in the text of the object's class undeclared (either alone or as the name to the left of the dot in a dot-name, but not as a receiver in a relation call or a called **prog** object in a **prog** object call) is treated as naming a proper component of the object and having class **any**. Object naming is explained in Sec 4.4.

An alias is an equivalent to some deep component or some group of deep components. The aliases are defined in the text of the object's class in the form of alias definitions (see Sec 5.4).

A row is used for synthesis of loops. The initial length of the row and the class of its elements are given in the text of the object's class in the form of a var-declaration (see Sec 5.2).

The order of an object's components in the value of the object is the following:

1. the proper components, in their order of declaration / first mentioning in the text of the object's class;
2. the virtual components and aliases, in their order of declaration / definition in the text of the object's class;
3. the row of the object (if any).

The default value for an instance of a user class is determined in the class text in the form of initializations and initialization amendments (see Sec 5.4, 5.6). These define default values for deep components of the object. The default value for these deep components for whom no initialization is provided in the class text is **nil**.

The relations performable on an instance of a user class are given in the class text in the form of relation definitions and equivalence amendments (see Sec 5.3, 5.6).

In synthesis (see Sec 7.0), computability of an object can be concluded from computability of the proper components of an object. Note that computability of the virtual components and of the optional row of the object is not required.

If the "Object Reduction" option in the "Options" menu in the NUT main window is "on", then values of a user class are coercible into **num** values, if the class has exactly one proper component, and this proper component has class **num**.

4.4 Naming of objects

The values of compound objects are structures whose components are objects in their own turn. Thus, in every state of the package, we have to do with a hierarchically built-up object world. Wherever the control is in the text of the package, some objects in this hierarchical world are considered as first-level of the hierarchy, and, as a rule, only these objects and their deep components can be referenced. These first-level objects are called the object context. The object context of a main program consists of the global objects of the package. The object context of a procedure consists of the parameters and local variables of the procedure. The object context of a user class text consists of the components of the class.

First-level objects (with the exception for first-level rows) are referenced in any context by simple names that are identifiers.

```
<simplename> ::=
        <identifier>
```

The only contexts where there can be a first-level row are user class texts. In a user class text, the row as a whole cannot be referenced (we say the row, because the class cannot have more than one row). Elements of the row are referenced by row element names:

```
<row-element-name> ::=
        | #<unsigned-integer>
        | #curr | #next
```

The name **#**_n_ refers to the _n_-th element of the first-level row. The relative names **#curr** and **#next** do not name any real objects. The name **#curr** refers to the current component of the row, the name **#next** refers to the next component of the row. These relative names (as well as names containing these relative names) make sense only in relation definitions (see Sec 5.3.). The semantics of **#curr** and **#next** is explained in detail in Sec 7.0.

```
<simplename> ::=
        <identifier>
```

An atomic name is either a simple name or a row element name:

```
<atomicname> ::=
        <simplename> | <row-element-name>
```

The NUT language provides direct addressing to first-level and second-level objects (i.e. until depth two). If a first-level object is a compound object, then the basic way of referencing its component is by the component's name, using the dot-name construction:

```
<dotname> ::=
        <atomicname>[.<simplename>]
```

As one can see from the syntactic definition, the name following the dot in a dot-name can only be simple, it cannot be a row element name. It is not possible to reference elements of the row of a first-level object.

To achieve access to objects of level >2, it is possible in user class texts to define alias names for deep components of the class (see Sec 5.4).

Alternatively to referencing by name, but only in programs (see Sec 6.3) and right-hand sides of initializations (see Sec 5.5), a component of a first-level object can be referenced positionally, by an expression evaluating to the ordinal number (index) of the component in the object.[1] Expressions are explained in Sec 6.1.

```
<indexname> ::=
        <atomicname>[[<expression>]]
```

If an index expression evaluates to a real number, the integer part of the value is used.[2]

Components of **`array`** and **`struct`** objects do not have names, and can be referenced only positionally.

Examples:

```
a, #curr, a.b, #next.x, #3.y, a[8]
```

In the following, we shall use the syntactic notions of lists of names.

```
<simplenames> ::=
        <simplename> [,<simplename>]...

<dotnames> ::=
        <dotname> [,<dotname>]...
```

[1] Also only in programs and right-hand sides of initializations, a certain combination of the two referring modes enables direct access to objects of levels deeper than two:

```
<dotindexname> ::=
        <indexname>[.<simplename>[[<expression>]]]
```

Examples:

```
#2[abs(k)].x, a.b[i+2], a[5].b[2]
```

[2] If the index expression evaluates to a negative number, an unclear situation arises (this error is not caught by the NUT interpreter).

5.0 User class texts

Each user class is defined in its own class text window, named after the class. The text of a class consists of sections specifying the components, relations (methods), and initializations of objects of the class.

```
<userclass-text>[1] ::=
        [<section>] ...

<section> ::=
        <super-section> | <var-section> | <vir-section>
        <rel-section> | <alias-section> | <init-section>
```

The object context of a user class text consists of the components of the class.

Generally, object names appearing in a user class text are interpreted in the object context of the user class text. The rules for naming objects in a given object context were explained in Sec 4.4.

There are four exceptions from this general rule:

- names in the left-hand sides of amendments in the prototype of a component declaration refer to components of the component being declared (cf Sec 5.6);
- names in the left-hand sides of amendments in the prototype of a **new**-expression refer to components of the object being created (cf Sec 6.4.5);
- the denotation of the receiver in a relation call is determined by a special rule outlined in Sec 6.3.2;
- the denotation of the called **prog** object in a **prog** object call is determined by a special rule outlined in Sec 6.3.

5.1 super-sections

A user class can inherit properties from one or more classes, called superclasses of the class. Only user-defined classes can act as superclasses. In the **super**-sections of a user class text, the user indicates the superclasses of the class being defined:

```
<super-section> ::=
        super <super-description>...

<super-description> ::=
        <userclass-name>;
```

The text of a user class can only extend or concretize the properties that the class inherits from the superclasses, e.g. it can add components or methods, concretize the class of a component which was declared to have class **any** in a superclass. It is impossible in a class text to redefine the properties inherited from the superclasses, e.g. overloading of relation definitions is impossible.

[1] Since all sections end with a semicolon, a user class text should always end with a semicolon. It is allowed by the class compiler to omit this very last semicolon in user class texts.

Examples:

```
      super Document;
      super Vehicle;
```

5.2 var- and vir-sections

In the **var**- and **vir**-sections of a user class text, the user declares the names and classes the proper and virtual components and (if the class has a row) the class of the elements of the row of the class he/she is defining:

```
<var-section> ::=
        var <var-declaration>...

<var-declaration> ::=
        <simplenames> : <class-name>;
        | <simplename> : <prototype-in-component-declaration>;
        | <row-specifier> : <class-name>;
        | <row-specifier> : <prototype-in-component-declaration>;

<row-specifier> ::=
        1 .. [<unsigned-integer>]

<vir-section> ::=
        vir <vir-declaration>...

<vir-declaration> ::=
        <simplenames> : <class-name>;
        | <simplename> : <prototype-in-component-declaration>;
```

The simple name(s) to the left of the colon in a component declaration are the name(s) of the component(s) being declared. The class name to the right of the colon in a component declaration is the name of the class of the component(s) being declared. Component declaration prototypes are explained in Sec 5.6.

Each component of a class can be declared only once in the text of the class. If a usage of a simple name (either alone or as the name to the left of the dot in a dot-name, but not as a receiver in a relation call or a called **prog** object in a **prog** object call) in a purely declarative part of a class text (i.e. elsewhere than in procedure texts, see Sec 5.3) precedes its declaration, then it is treated as naming a proper component having class **any**.

A var-declaration which has a row specifier to the left of the colon tells that the class being defined has a row, tells the class of the row elements (-- the class name to the right of the colon), and may tell a bound on the length of the row (-- the unsigned integer following **..**). There can be only one such var-declaration in a class text.

A component or elements of the row may be declared to be of an non-existent class; this does not cause any diagnostics to be displayed. This non-existent class is considered empty (the compound class of structures of length 0, without relations and initializations).

Recursive class definitions are allowed, i.e. a class can have components of the same class. Even mutual recursion is possible.

Examples:

```
var   x, y: Point;
      line: array 5 of Point;
vir   n1, n2, n3: num;
```

5.3 rel-sections

In the **`rel`**-sections of a user class text, the user defines the relations of the class he/she is defining. Relations are procedures performable on instances of the class.

<**`rel`**-section> ::=
  `rel` <relation-definition>...

<relation-definition> ::=
  [<relation-name>**`:`**] <relation-definiens>**`;`**

<relation-name> ::=
  <identifier>

A relation can be defined only once in a class text. To provide a name for a relation is optional.

Relations are procedures. Procedures encode imperative algorithms for object manipulation, and the regular way of defining them is in the form of procedure texts that contain a specification and may contain a program. The specification of a procedure determines the input/output interface (external view) of the procedure: it states parameters of the procedure, defines certain order on them, an tells their roles (input, output, weak). The program of a procedure determines the inner behaviour of the procedure. Procedure texts, specifications, and programs are explained in Sec's 6.0, 6.1, 6.3, respectively.

Relations are used in two ways. First, relations are used as methods: a relation can be called on an object explicitly, if the relation has a name, by sending a message to the object (relation calls are explained in Sec 6.3.2). Second, programmed relations are used in synthesis of values for objects and of programs for procedures and their subtasks. Synthesis of an object value can be prompted by a **`compute`**-statement (explained in Sec 6.3.4), synthesis of programs for a procedure (if it is unprogrammed) and for its subtasks is prompted by calling the procedure (explained in Sec 6.3.2). In planning, which is the difficult part of synthesis, only the specifications of relations are used: they are interpreted as axioms about computability. The mechanism of synthesis is explained in Sec 7.0. Relations without name cannot be called explicitly, they are only used in synthesis.

The names of parameters of relations and of parameters of dependent subtasks of relations are also names of deep components of the class being defined. The names of parameters of independent subtasks of relations are names of deep components of the class mentioned in the specification of the respective subtask.

Both for synthesis of a program for an unprogrammed relation and for a call of (either programmed or unprogrammed) relation, the parameters of the relation are identified with the homonymic deep components of the object who is being asked to perform the relation. ('Homonymic' means 'having same names'.) This means that values are directly written into/read from the state of the object in the course of parameter passing, when the relation is performed.

For synthesis of a program for a subtask, the parameters of the subtask are identified with the homonymic deep components of the class named in the specification of the subtask, if the subtask is independent, and with the homonymic deep components of the object on whom some value or relation program is to be synthesized, if the subtask is dependent. For a call of a subtask of either kind, however, the parameters of the subtask are purely formal.

Besides procedure texts, the language facilitates certain forms of relation definienses that contain a specification and a program in an implicit form (equations, equivalences). It is the task of the class compiler (more exactly, the equation solver) to complete such definienses. Summing up, four kinds of relation definienses are facilitated in the language:

- procedure texts,
- multi-way equations,
- one-way equations,
- equivalences.

```
<relation-definiens> ::=
        <procedure-text>
        | <multi-way-equation> | <one-way-equation> | <equivalence>
```

Procedure texts are explained in detail in Sec 6.0.

Multi-way equations have the following form:

```
<multi-way-equation> ::=
        <expression-in-equation> = <expression-in-equation>
```

where all deep components mentioned in the two expressions have classes coercible to **num**, and the denotations of the two expressions have also class **num** under coercions. Expressions are explained in Sec 6.4. All deep components mentioned in the equation are weak parameters of the relation.

The specification derivable by the class compiler from a multi-way equation *e*1 = *e*2 whose list of names (in the order of first occurrences) is *c*1, ..., *cn* is:

```
-- c1, ..., cn
```

and the program derivable is:

```
if c1 == nil ->
        c1 := ee1
|| ...
|| cn == nil ->
        cn := een
fi
```

where *eei* (*i* = 1, ..., *n*) is an explicit expression for *ci* in terms of *c1*, ..., *ci*-1, *ci*+1, ..., *cn* derived from the equation by the the built-in equation solver of the system.

One-way equations have the following form:

```
<one-way-equation> ::=
        <dotname> = <expression-in-equation>
```

where the deep component named in the left-hand side and the denotation of the expression in the right-hand side have compatible non-numeric classes. The deep component named in the left-hand side is an output parameter, the deep components named in the right-hand side are input parameters of the relation.

The specification derivable by the class compiler from a one-way equation *c* = *e* where the list of names in *e* (in the order of first occurrences) is *c1*, ..., *cn* is:

*c*1, ..., *cn* -> *c*

and the program is:

c := *e*

(i.e. a single assignment statement).

Equivalences have the following form:

```
<equivalence> ::=
        <dotname> = <dotname>
```

where the two named deep components have compatible classes.

The specification derivable by the class compiler from an equivalence *c*1 = *c*2 is:

-- *c*1, *c*2

Equivalence of two deep components is implemented via locating their values at the same place in the memory. Thus, values of deep components that are declared to be equivalent are automatically kept identical.

A relation whose definiens contains names **#curr** and **#next** abbreviates a collection of relations. Their definienses are derived from that of the given relation by substituting **#curr**, **#next** with **#***i*, **#**(*i*+1), respectively, where 0< *i* <length of the row.

Examples:

```
rel
        method1: x + y = 0;
        1 * sin(alpha) = P1.y - P2.y;
        method2: res = in1 & in2;
        how_long: new_text = length('This is a text');
        proceed: [in1, in2 -> res],
              [schema |- old_state -> new_state] x -> y {
                    subtask 1(x, 1, a);
                    subtask 2(x, b);
                    y:= a & b
              };
        symmetric: -- x, y, z {
```

```
if (x == nil) & (y /= nil) & (z /= nil) ->
      x := z - y;
|| (x /= nil) & (y == nil) & (z /= nil) ->
      y := z - x;
|| (x /= nil) & (y /= nil) & (z == nil) ->
      z := x + y
fi
};
```

5.4 alias-sections

In **alias**-sections, the user defines the aliases of the class he/she is defining.

```
<alias-section> ::=
      alias <alias-definition>...

<alias-definition> ::=
      <simplename> = <alias-definiens>;
```

The name to the left of = in an alias definition is the name of the alias being defined.

Aliases provide short names for second-level components and for groups of first- and second-level components:

```
<alias-definiens> ::=
      <name-with-dot> | <group>

<name-with-dot> ::=
      <atomicname>.<simplename>
```

An alias for a second-level component has always the class and the value of this second-level component.

```
<group> ::= (<pseudodotnames>)

<pseudodotnames> ::=
      <pseudodotname> [,<pseudodotname>]...

<pseudodotname> ::=
      <dotname> | all.<simplename>
```

A group alias has class **struct**, and its value is a structure formed of the values of the members of the group. Components of a group alias can be referenced via the alias name positionally.

The **all** construction is used in the definienses of group aliases in the following way. The construction **all.***x* is a short form for *c***1.***x***, ...,** *cn***.***x*, where *c*1, ..., *cn* is the list of all components of the class being defined that possess a component named *x*.

Alias definitions are implemented via common memory, exactly as equivalences (cf Sec 5.3). Therefore, the value of an alias is automatically kept identical to the value its definiens. In synthesis, alias definitions function exactly as equivalences.

Examples:

```
     alias
          Px = P.x;
          all_states = (all.state);
          states = (current_state, next_state);
```

5.5 init-sections

In **init**-sections, the user tells the initializations of the class he/she is defining.

<**init**-section> ::=
  init <initialization>...

Initializations allow the user to provide deep components of instances of his/her class with default (initial) values. Initializations are assignments that are performed, when an instance of the class is created. The order of performing the initializations is the order of their appearance in the class text. Objects are created by means of **new**-expressions, which are explained in Sec 6.4.5. Assignments are explained in Sec 6.3.1.

<initialization> ::=
  <dotname> <assign-symbol> <expression>**;**

<assign-symbol> ::=
  := | **<-**

Though similar to one-way equations, initializations are not used in synthesis in the same way as relations are. But: initializations assign values to deep components of objects at creation-time, and definedness of objects is used in synthesis. For details on the mechanism of synthesis, see Sec 7.0.

Examples:

```
     init P.x := 0;
          structure := [1, 2, 3];
          len <- length('This is a text') + 5;
```

5.6 Component declaration prototypes

Prototypes are used in component declarations of user class texts for giving components more properties than they would acquire just from their classes alone. Component declarations were explained in Sec 5.2.

A prototype for a component consists of the name of the component's class and amendments. Amendments state the extra properties of the component. Amendments are really a shortcut device, the programmer can always do without them. However, class texts with amendments is often more elegant and simpler.

<prototype-in-component-declaration> ::=
  <userclass-name> <amendments>

<amendments> ::=
  <amendment> [**,**<amendment>]...

There are two kinds of amendments:

- initialization amendments,
- equivalence amendments.

```
<amendment> ::=
        <initialization-amendment> | <equivalence-amendment>
```

An *initialization amendment* enriches the component being declared with extra initializations.

```
<initialization-amendment> ::=
        [<simplename> <assign-symbol>[1]] <expression>
```

The name to the left of the assignment symbol refers to a component of the component being declared. The names in the expression to the right of the assignment symbol refer to deep components of the class being defined, according to the general rule about denotations of object names. The class of the value of the expression in the right-hand side must be coercible into the class of the object in the left-hand side. The semantics of initialization amendments is exactly the same as that of initializations, see Sec 5.5.

An *equivalence amendment* enriches the component being declared with extra equivalence relations.

```
<equivalence-amendment> ::=
        [<simplename> =] <dotname>
```

The name to the left of = refers to a component of the component being declared. The name to the right of = refers to a deep component of the class being defined, according to the general rule about denotations of object names. The two objects must be of compatible classes. The semantics of equivalence amendments is exactly the same as that of equivalence relations, see Sec 5.3.

Amendments without left-hand parts are called *positional amendments*, in contrast to "normal" amendments, that are called *amendments by name.*

If the first amendments in a list of amendments are positional, they refer to the first components of the class. The positional amendments that follow an amendment by name refer to the components succeeding the component of the amendment by name.

The type of a positional amendment is determined as follows. If the right-hand side of the positional amendment is a name, then the amendment is an equivalence amendment, otherwise it is an initialization amendment. Positional initialization amendments work as if the assignment symbol were **:=**.

Examples:

```
tr1: triangle side1 = 10;
```

[1] For reasons of backward compatibility, even = is allowed as an assignment symbol in initialization amendments, synonymously to **:=**, if the right-hand side expression of the amendment is not a name. We discourage this usage of = for clarity reasons.

```
    tr2: triangle side1 := 10, 20, alpha := 30;
    tr3: triangle side1 = tr1.side1, side2 := tr2.side2 + 5;
```

When an object of a user class is created, the order of performing the initializations stated in the class text (these appear either in **`init`**-sections or in amendments of component declaration prototypes) is the order of their appearance in the class text.

6.0 Procedure texts

Procedures encode imperative algorithms for manipulating objects.

Procedures can be programmed or unprogrammed:

```
<procedure-text> ::=
        <programmed-procedure-text} | {unprogrammed-procedure-text}
```

The text of a programmed procedure contains a specification-part (also called axiom) and an implementation-part:

```
<programmed-procedure-text> ::=
        <procedure-specification> { <procedure-implementation> }
```

The text of a unprogrammed procedure does not contain a program:

```
<unprogrammed-procedure-text> ::=
        <procedure-specification> { spec }
```

(The keyword **spec** says: “the program has to be synthesized from the specification”.)

The specification-part of a procedure text determines the input/output interface (external view) of the procedure: it states parameters of the procedure, defines certain order on them, an tells their roles (input, output, weak). Specifications are explained in Sec 6.1. The implementation-part of a procedure text determines the inner behaviour of the procedure.

An implementation can be given by a program or by a reference to a C-function:

```
<implementation> ::=
        <program> | c_fun <c-function-name>
```

Programs are explained in Sec 6.3.

In the texts of NUT packages, procedure texts appear in two kinds of positions: as relation definienses in relation definitions (see Sec 5.3) and as **prog** object definienses in **new**-expressions (programmed procedures only, see Sec 6.4.5).

Procedures can be called, see Sec 6.3.2. When a procedure is called, first programs are synthesized for the procedure (if it is unprogrammed) and for its subtasks by the interpreter (more exactly, the synthesizer). Besides, programmed relations are used in synthesis of values for objects and of programs for unprogrammed relations. Synthesis is explained in Sec 7.0.

The main programs of NUT packages are programs that are not part of any procedure text. They do not have parameters. Main programs are written in the NUT main window.

6.1 Specifications (axioms)

The specification of a procedure states the parameters of the procedure, defines certain order on them, an tells their roles (input, output, weak).

Specifications of procedures are used in two ways. First, specifications determine how parameter passing should be carried out in procedure calls and subtask calls (see Sec 6.3.2, 6.3.3). Second, specifications of programmed relations are interpreted as axioms about computability in synthesis (see Sec 7.0).

The specification of a procedure consists of the specifications of the subtasks of the procedure and of a pattern of the procedure.

```
<procedure-specification> ::=
        [<subtask-specifications>] <procedure-pattern>

<subtask-specifications> ::=
        <subtask-specification> [,<subtask-specification>]...
```

Subtasks are a special kind of input parameters of procedures. They have always class **prog**. A subtask of a procedure is specified by a subtask specification in the procedure specification. The program for the subtask of the procedure is to be synthesized by the interpreter (more exactly, the synthesizer), whenever the procedure is needed in synthesis of some object value or relation program.

```
<subtask-specification> ::=
        [ [<userclass-name> |- ] <subtask-pattern> ]
```

The main part of the specification of a subtask is the subtask's pattern. Optionally, also a user class name may be indicated in the specification of a subtask. If this is so, then the subtask is called independent. Otherwise, the subtask is dependent.

Procedure patterns have the following form:

```
<procedure-pattern> ::=
        <dotnames>] [-- <dotnames>] -> [<dotnames>]
        | -- [<dotnames>]
```

The names in the pattern of a procedure are parameters of the procedure.

Subtask patterns have the following form:

```
<subtask-pattern> ::=
        <dotnames> -> <dotnames>
```

The names in subtask pattern are names of the parameters of the subtask.

The names to the left of **->** (and **--**) in the pattern of a procedure or a subtask are names of the input parameters of the procedure or subtask. The names to the right of -> are names of output parameters.

The names to the right of **--**, but not to the right of **->** in the pattern of a procedure are weak parameters of the procedure. In an explicit call of a procedure, its weak parameters function as input-output parameters.

A name can be listed at most once in each of the at most three lists of a pattern. Note that it thus is allowed for a parameter to be an input-output parameter, weak-output parameter etc.

Both for synthesis of a program for an unprogrammed relation and for a call of (either programmed or unprogrammed) relation, the parameters of the relation are identified with the homonymic deep components of the object who is being asked to perform the relation.

For a call of a **prog** object, the parameters of its specification are formal.

For synthesis of a program for a subtask, the parameters of the subtask are identified with the homonymic deep components of the class named in the specification of the subtask, if the subtask is independent, and with the homonymic deep components of the object on whom some value or relation program is to be synthesized, if the subtask is dependent. For a call of a subtask of either kind, its parameters are formal.

Unprogrammed procedures are not allowed to have subtasks or weak parameters.

6.2 Object contexts of programs

The object context for a program consists of its parameters and its local variables. Local variables are not declared in programs The default class for local variables is **any**, and the default value for them is **nil**.

Recall that the parameters of an object's relation are always identified with deep components of the object (see Sec 6.1).

Variables local to main programs are global objects to the package.

Generally, object names appearing in a program are interpreted in the object context of the program. The rules for naming objects in a given object context were explained in Sec 4.4.

There are three exceptions from this general rule:

- names in the left-hand sides of amendments in the prototype of a **new**-expression refer to components of the object being created;
- the denotation of the receiver in a relation call is determined by a special rule outlined in Sec 6.3.2;
- the denotation of the called **prog** object in a **prog** object call is determined by a special rule outlined in Sec 6.3.

6.3 Programs

Programs encode imperative instructions for manipulating objects.

```
<program>[1] ::=
        <statement>
```

[1] A program that is just an empty statement is considered illegal.

```
<statement> ::=
        <assignment>
        | <function-call>
        | <procedure-call>
        | <subtask-call>
        | <compute-call>
        | <produce-call>
        | <sequence>
        | <empty-statement>
        | <if-statement>
        | <for-statement>
        | <do-statement>
        | <exit-statement>
```

Function calls are explained in Sec 6.4.3. Other kinds of statements are explained below.

6.3.1 Assignments

Assignments are used for changing values of objects.

```
<assignment> ::=
        <dotindexname> <assign-symbol> <expression>

<assign-symbol> ::=
        := | <-
```

Expressions are explained in Sec 6.4.

An assignment is performed as follows. The right-hand side expression is evaluated and the result is assigned to the object named in the left-hand side.

How assignment is carried out technically, depends on the assignment symbol. If it is **:=**, then the assignment is done by copying of a value. If it is **<-**, the assignment is done by copying a reference (pointer) to a value. The latter kind of assignment is faster and saves memory, when dealing with large structures.

Generally, the value of the right-hand side expression must be coercible into a value of the class of the object named in the left-hand side. But: if the object named in the left-hand side had class **any** prior to assignment, its class is concretized into that of the assigned value.

Examples:

```
P2.x := 1 * cos(alpha) + P1.x
table[length(table) + 1] <- a.state
string := 'this is a text'
tri <- (new triangle alpha := 0.7)
```

6.3.2 Procedure calls

Procedure calling is one of the main tools of manipulating objects in the NUT system.

Two kinds of entities do have procedural values: relations and **prog** objects. Accordingly, there are two kinds of procedure calls:

- relation calls,
- **prog** object calls.

```
        <procedure-call> ::=
                <relation-call> | <prog-object-call>
```

Relation calling corresponds to message sending in OO languages.

In a relation call, some object, titled the receiver of the call, is prompted to perform one of its relations whereby the actual parameters of the call give values to and receive values from the parameters of the called relation. Recall that, in a relation call, the parameters of the called relation are identified with the homonymic deep components of the receiver, i.e. the relation is performed directly on the receiver (Sec 6.2).

```
        <relation-call> ::=
                [<receiver-name>.] <relation-name> (<expressions-and-holes>)

        <receiver-name> ::=
                <simplename> | <userclass-name>
```

The receiver name in a relation call determines the receiver. If the receiver name is a simple name, then it refers to the first existent among the following alternatives:

- to a first-level object wrt the program where the call appears,
- to a component of the SELF-object, i.e. of the object who is performing the program where the given relation call appears,
- or to a global object,

If the receiver name is a user class name, a temporary object of the mentioned class is created which performs the named relation, and is deleted immediately thereupon. If no receiver name is given in the call expression, the named relation is performed by the SELF-object, i.e. by the object who is performing the program where the given relation call appears.

In a **prog** object call, a **prog** object is prompted to perform itself whereby the actual parameters of the call give values to and receive values from the parameters of the called **prog** object.

```
        <prog-object-call> ::=
                <simplename>(<expressions-and-holes>)
```

The simple name in a **prog** object call determines the **prog** object that has to perform itself. It refers to the first existent among the following alternatives:

- to a first-level object wrt the program where the call appears,
- to a component of the SELF-object, i.e. of the object who is performing the program where the given **prog** object call appears,
- or to a global object,

Lists of expressions and holes as well as expressions are explained in Sec 6.4. The expressions listed in the parentheses in a procedure call represent the actual parameters of the call. The correspondence between actual parameters of the call and the parameters of the called procedure is established by the following rule. Let *n* be the number of subtasks of the called procedure. Then, the *i*-th subtask of the called procedure corresponds to the *i*-th element in the list of actual parameters of the call. Further, the *i*-th parameter[1] in the pattern of the called procedure corresponds to the (*n*+*i*)-th element in the list of actual parameters of the call.

One does not need to use the mechanism of holes for omitting the actual parameters that are the last in the list. Instead, it is allowed just to leave the list of actual parameters shorter.

A procedure call is performed as follows. First, programs are synthesized for the called procedure (if it is unprogrammed) and its subtasks. Failure to synthesize results in a runtime error. Then, the given actual input and weak parameters of the call are evaluated, and their non-**nil** values are copied into the corresponding parameters of the called procedure[2]. Thereafter, the receiver performs the called relation, or the called **prog** object performs itself. Finally, the values of the weak and output parameters of the called procedure are copied into the corresponding given actual parameters of the call. If some actual parameter is not given in the call expression, then no copying is done into or from the corresponding parameter of the called procedure.

Note that, for relation calls, the state of the receiver always provides the default input values for computation. If some actual input parameters are not given in a relation call, no copying is done into the corresponding deep components of the receiver. Thus, the values they happen to possess at the moment of the relation call, serve as the input values for the computation.

The classes of the actual parameters of a relation call must be compatible with those of the corresponding deep components of the receiver. The actual output parameters of a procedure call have to be dot-index-names, since values are copied into them.

The return value of a procedure call is formed from the final values of the weak and output parameters of the called procedure. If the called procedure has only one weak parameter that did not obtain an initial value from an actual parameter of the call, or it has only one output parameter, then the call returns the final value of this weak or output parameter. If the called procedure has several such parameters, then the call returns a **struct** object formed of the final values of these parameters in their order of appearance in the pattern[3] of the called procedure.

Examples:

```
a.put( b.x + 3, , length(b.x) )
c.get( )
```

[1] Exception: the occurrences of input-output parameters among the outputs in the pattern are excluded from the count.

[2] The actual input parameters corresponding to subtasks are neither evaluated nor copied. They are simply ignored.

[3] Exception: if a parameter is an input-output parameter, then, instead of its position among the output parameters in the pattern, its position among the input parameters of the pattern is taken into account.

Besides being used as statements, procedure calls can also be used as expressions, because they do return values, see Sec 6.4.

6.3.3 Subtask calls

Subtask calls may only appear in the programs of procedures (i.e. not in main programs). A subtask call in such a surrounding forces one of the subtasks of the procedure to be performed whereby the actual parameters of the call give values to and receive values from the parameters of the called subtask.

```
<subtask-call> ::=
        subtask <unsigned-integer>(<expressions-and-holes>)
```

The unsigned integer following the keyword **subtask** in a subtask call refers to the subtask to be performed (it is an ordinal number of a subtask specification in the specification of the procedure in whose program the given subtask call appears).

Lists of expressions and holes as well as expressions are explained in Sec 6.4. The expressions listed in the parentheses are actual parameters to the called subtask. The correspondence between the actual parameters of the call and the parameters in the pattern of the subtask is established by the following rule. The *i*-th parameter in the pattern of the called subtask[1] corresponds to the *i*-th element in the list of actual parameters of the call.

One does not need to use the mechanism of holes for omitting the actual parameters that are the last in the list. Instead, it is allowed just to leave the list of actual parameters shorter.

A subtask call is performed as follows. First, the given actual input parameters of the call are evaluated, and their non-**nil** values are copied into the corresponding parameters of the called subtask. Then, the synthetic implementation of the called subtask is performed. Finally, the values of the output parameters of the called subtask are copied into the corresponding given actual parameters of the call. If some actual parameter is not given in the call expression, then no copying is done into or from the corresponding parameter of the called subtask.

The classes of the actual parameters of a subtask call must be compatible with those of the corresponding parameters of the called subtask (if the subtask is independent). The actual output parameters have to be dot-index-names, since values are copied into them.

Subtask calls are similar to procedure calls. The important difference, however, is that they do not return values, i.e. they are not expressions.

Example:

```
subtask 2(in, out)
```

[1] Exception: the occurrences of input-output parameters among the output parameters in the pattern are excluded from the count.

6.3.4 compute-call

A **compute**-call prompts synthesis of fully defined values for given deep components of a given first-level object of the object context. (A value is fully defined if it contains no **nil**'s.)

```
    <compute-call> ::=
            <simplename>.compute([<dotnames>])
```

The name to the left of the dot is the name of the object whose deep components have to be computed. The deep components themselves are referenced by the dot-names listed in the parentheses. In case if the list of dot-names is omitted, then all proper components of the object have to be computed.

compute behaves like a predefined relation common to all user classes, since the syntax of **compute**-call is similar to that of a relation call. There is, nevertheless, one characteristics that distinguishes **compute** from relation calls: the actual input parameters of the call are not evaluated at parameter passing time (they are used as names).

compute returns a `bool` type value which indicates whether the synthesis succeeded or not.

In case if the list of dot-names is omitted (all proper components of the object have to be computed), `false` is returned, if no components can be computed; if at least one component can be computed, `true` is returned.

In case if the list of dot-names is not omitted, `true` is returned, if fully defined values can be computed for all asked deep components; otherwise, `false` is returned.

Example:

```
    tri.compute(alpha, b)
```

6.3.5 produce-call

A **produce**-call prompts that a given applicable first-level object of the object context performs the first of its relations which it can perform and which could possibly help it in producing fully-defined values for its given deep components.

```
    <produce-call> ::=
            <simplename>.produce([<dotnames>])
```

The name to the left of the dot is the name of the object whose deep components serve as computation goals when a performable relation is looked for. The deep components themselves are referenced by the dot-names listed in the parentheses. In case if the list of dot-names is omitted, then all proper components of the object serve as computation goals when an a performable relation is looked for.

produce behaves like a predefined relation common to all user classes, since the syntax of **produce**-call is similar to that of a relation call. There is, nevertheless, one characteristics that distinguishes **compute** from relation calls: the actual input parameters of the call are not evaluated at parameter passing time (they are used as names).

compute returns a `bool` type value which indicates whether any performable relation was found. `true` is returned, if such relation was found (in this case an one-step algorithm is built and executed), otherwise, `false` is returned.

6.3.6 Sequences and the empty statement

A sequence is formed of two statements that have to be performed the first prior to the second.

```
<sequence> ::=
        <statement>; <statement>
```

Example:

```
x := y + 2; b := new array x of num
```

An empty statement is also possible.

```
<empty-statement> ::=
```

6.3.7 if-statements

if-statements encode choices between guarded alternatives.

```
<if-statement> ::=
        [<label>:] if <alternatives> fi

<alternatives> ::=
        <alternative> [|| <alternative>]...

<alternative> ::=
        <expression> -> <statement>

<label> ::=
        <identifier>
```

Expressions are explained in Sec 6.4. The expression to the left of **->** in an alternative is called the guard, and must evaluate to a boolean value. The statement to the right of **->** in an alternative is called the body of the alternative. An **if**-statement is performed as follows. The guards of the alternatives are evaluated consecutively until a true guard is found, whereupon the body of the selected alternative is performed. If all guards evaluate to **false**, nothing happens.

if-statements, as well as **for**- and **do**-statements (see next sections), can be labelled by identifiers.

Example:

```
if
        x[i] > max -> max := x[i]
        || x[i] < min -> min := x[i]
fi
```

6.3.8 for-statements

for-statements encode loops whose iteration is controlled by a numeric control variable.

```
<for-statement> ::=
        [<label>:] for <simplename> [:= <expression>] [step <expression>]
        to <expression> do <statement> od
```

Expressions are explained in Sec 6.4. The control variable of a **for**-statement must have class **num**. The three expressions in a **for**-statement must evaluate to numbers. They give the initial value, the step, and the bound for the value of the control variable. If the initial value expression is omitted, the initial value is 1. If the step expression is omitted, the step is 1.

Example:

```
for i to n do
        tab.x[i] := i + 1
od
```

6.3.9 do-statements

do-statements encode loops whose iteration is controlled by guards of alternatives.

```
<do-statement> ::=
        [<label>:] do <alternatives> od
```

At every round of iteration, the guards of the alternatives are evaluated consecutively until an alternative with true guard is found, whereupon the body of the this alternative is performed. The execution of the loop terminates when no alternative can be selected (i.e. all guards evaluate to **false**).

Example:

```
L: do
        c_button(0) ->
                NutPrint('The left button is pressed')
        || c_button(1) ->
                exit L
        || true  ->
                NutPrint('Nothing happens')
od
```

6.3.10 exit-statements

An **exit**-statement forces exiting from an enclosing **if-**, **for-**, or **do**-statement, or the whole program.

```
<exit-statement> ::=
        exit [<label>]
```

If a label is indicated in an **exit**-statement, then a statement with the given label is exited. Otherwise, the enclosing program is exited.

Examples:

```
exit bst
exit
```

6.4 Expressions

Expressions are constructions of the language that encode values, used in programs (see Sec 6.3) and initializations and initialization amendments (see Sec 5.5, 5.6, 6.4.5). Evaluation of an expression may have side-effects (e.g. change the values of some objects or the like).

```
<expression> ::=
        <constant>
        | <dotindexname>
        | <operator-expression>
        | <function-call>
        | <compute-call>
        | <produce-call>
        | <procedure-call>
        | <structure-expression>
        | <new-expression>
        | (<expression>)
```

Procedure calls were explained in Sec 6.3.2. Other kinds of expressions are explained below.

In equations (see Sec 5.3), expressions of a restricted form, called ‘expressions in equations’ were used. The syntactic definition of <expression-in-equation> can be obtained from that of <expression>, by replacing all occurrences of <dotindexname> in the syntactic definition of <expression> with <dotname>.

Examples:

```
a ^ (b * x) + (-c) ^ 2
-sqrt(a) - (ln(b) / 2 - c)
(a /= nil) & ( b >= 2 | c)
~method(a)
delete(a, pos('This is a text', 'text'), length('text'))
```

The useful syntactic notions of list of expressions and of list of expressions and holes (throughout this report, the word ‘hole’ is used as a technical term meaning ‘omitted item’) are defined as follows:

```
<expressions> ::=
        [<expression> [,<expression>]... ]

<expressions-and-holes> ::=
        [<expression>] [,[<expression>]]...
```

6.4.1 Constants

Constants evaluate to fixed primitive values.

```
<constant> ::=
        <number> | <string> | true | false | nil
```

Numbers have class **`num`**, strings have class **`text`**, **`true`** and **`false`** have class **`bool`**.

`nil` is a constant in every class and represents an undefined value.

6.4.2 Operator expressions

Operators are predefined functions with simplified call syntax. As any functions, they transform input values of certain classes into output values of certain classes. Operator expressions are calls of operators.

```
<operator-expression> ::=
        <unary-operator> <expression>
        | <expression> <binary-operator> <expression>

<unary-operator> ::=
        - | ~

<binary-operator> ::=
        + | - | * | / | ^
        | & | |
        | == | /=
        | > | < | >= | <=
```

The value of an operator expression is the result of applying the operator to the values of the operand expressions.

The operators have the following semantics (and type schemes):

`-`	negation	**`(num -> num)`**
`+, -`	addition, subtraction	**`(num num -> num)`**
`*, /`	multiplication, division	**`(num, num -> num)`**
`^`	power	**`(num, num -> num)`**
`~`	negation	**`(bool -> bool)`**
`&, |`	conjunction, disjunction	**`(bool, bool -> bool)`**
`==, /=`	identical, not identical	**`(x, x ->bool)`**
`<, >, <=, =>`	lt, gt, le, ge	**`(num, num -> bool)`**

The operators `==` and `/=` are polymorphic, they can be used for comparing any two values that are of compatible classes.

The priorities of the operators are the following (starting from the highest):

```
-, ~
*, /, &
+, -, |
==, /=
<, >, <=, =>
```

Examples:

```
(a >= 3) == true
2 ^ 10 - n * 3
```

6.4.3 Function calls

Functions transform input values of certain classes into output values of certain classes.

<function-call> ::=
    <function-name> (<expressions>)

The expressions listed in the parentheses in a function call denote the arguments of the function call. The value of a function call is the result of applying the function to the values of the given arguments.

The NUT language has a number of predefined functions. These are organized into libraries. There is one statically loaded (or, built-in) library and a collection of dynamically loadable libraries. The built-in functions are called standard functions.

<function-name> ::=
    <std-function-name> | <dll-function-name>

The names of the standard functions are listed in Sec 3.3. The descriptions of functions appear in a separate document *The NUT Libraries*[1].

Examples:

```
NutPrint('Hello world!')
gr_text(f, (new Point x := 20, y := 30), 'Cheers!', 1)
asin(x + 0.1)
```

Besides being used as expressions, function calls can also be used as statements, when their return values are not needed, see Sec 6.3.

6.4.4 Structure expressions

Structure expressions are used for forming **struct** values.

<structure-expression> ::=
[<expressions-and-holes>**]**

A structure expression evaluates to a **struct** object formed of the values of its listed expressions. A hole in a structure expression is just a shortcut for constant **nil**.Example:

```
[3, 4, 5, 'Hello', (new square a:=8), , [2, true]]
```

[1] Available by anonymous ftp from **it.kth.se**, file **labs/se/Software/NUT/doc/libraries.ps.Z**.

6.4.5 new-expressions

new-expressions are used for dynamic instantiation of objects from classes or prototypes.

```
<new-expression> ::=
        new <class-name>
        | new <userclass-name> <prototype-in-new-expression>
        | new prog <programmed-procedure-text>
```

When a **new**-expression is evaluated, a new object is created from a given class or prototype (i.e. memory is allocated for the object, and initializations are performed). The value of the expression is the object that was created. This object does not have an identity. To become able to refer to it, the **new**-expression must be assigned to an object with name (typically to an object that previously had class **any** and value **nil**).

The purpose and usage of prototypes in **new**-expressions is the similar to their purpose and usage in component declarations in user class texts, see Sec 5.6. There is just one differences: only initialization amendments are allowed in the prototypes of **new**-expressions.

```
<prototype-in-new-expression> ::=
        <userclass-name> <initialization-amendments>

<initialization-amendments> ::=
        <initialization-amendment> [,<initialization-amendment>]...

<initialization-amendment> ::=
        [<simplename> <assign-symbol>] <expression>
```

The names in the left-hand sides of the initialization amendments of a **new**-expression refers to components of the object being created. The names in the right-hand sides refer to objects of the current object context, according to the general rule about denotations of object names.

When an object is created from a prototype using **new**, the initializations originating from the text of the user class named in the prototype are performed prior to the initializations in the amendments of the prototype.

When creating a **prog** object using **new**, a procedural value must be assigned to it in the prototype by presenting a programmed procedure text in the **new**-expression. Procedure texts are explained in Sec 6.

Examples:

```
new triangle a := 5, beta := 0.8, 0.6
new array 100 of text
new prog x, y -> z {z := sqrt(x^2 + y^2)}
```

7.0 Synthesis

In NUT, synthesis occurs in two kinds of situations:

- synthesis of fully defined values for deep components of an object of the object context can be prompted by a **`compute`**-statement (explained in Sec 6.3.4);
- synthesis of programs for a relation of an object (if its unprogrammed) and for its subtasks is prompted, when the object is asked to perform the relation (relation calls are explained in Sec 6.3.2);

In synthesis of a value for a deep component of an object or of a program for an unprogrammed relation of an object, the following knowledge can be used:

- the states (values) of the fully defined deep components of the object;
- the programmed relations of the object and of its deep components; in synthesis of programs for independent subtasks of relations, also the programmed relations of the classes indicated in subtask specifications.

The difficult stage of synthesis is planning of solution, carried out by the planner. The planner composes the structure of the program that computes the asked value or implements the asked relation.

The planner interprets specifications of programmed relations as axioms about computability. (Computability of an object means that a fully defined value can be computed for the object; computability of a relation or subtask means that a program can be synthesized for the relation or subtask.)

Recall that a relation whose definiens contains names **`#curr`** and **`#next`** abbreviates a collection of relations. Their definienses are derived from that of the given relation by substituting **`#curr`**, **`#next`** with **#***i*, **#**(*i*+1), respectively, where 0< *i* <length of the row.

Planning is based on a formal calculus with the following rules:

- if a deep component of an object has a fully defined value (i.e. a value containing no **`nil`**'s), then this deep component is computable on the object:

  o has deep component *d*   *o.d* has a fully defined value

  ----

  d computable on *o*

- if an object has a programmed relation whose all input parameters (incl. subtasks) and all weak parameters, with exception for at most one weak parameter, are computable on the object, then the remaining weak parameter and all output parameters of the relation are also computable on the object:

  o has prog'd rel'n *r* with spec. *ins* -- *weaks, w* -> *outs*   *ins, weaks* computable on *o*

  ----

  w, outs computable on *o*

- if two deep components of an object are equivalent, and one of them is computable on the object, then the other is also computable on the object;

$$\frac{d1,\ d2 \text{ are equivalent deep components of } o \qquad d1 \text{ computable on } o}{d2 \text{ computable on } o}$$

if two deep component of an object are equivalent, then their corresponding components are also equivalent:

$$\frac{d1,\ d2 \text{ are equivalent deep components of } o}{d1[i],\ d2[i] \text{ are equivalent deep components of } o}$$

- if the constituents of an alias of an object are computable on the object, then the alias is also computable on the object; and, vice versa, if an alias of an object is computable on the object, then its constituents are also computable on the object:

$$\frac{o \text{ has alias } a \text{ with constituents } d1, \ldots, dn \qquad d1, \ldots, dn \text{ computable on } o}{a \text{ computable on } o}$$

$$\frac{o \text{ has alias } a \text{ with constituents } d1, \ldots, \text{dn} \qquad a \text{ computable on } o}{d1, \ldots, dn \text{ computable on } o}$$

- if all proper components of a component of an object are computable on the component, then the component is computable on the object (note that computability of the virtual components and of the row of the component is not required to draw this conclusion); and, if a component is computable an object, then all components of the component are computable on the component:

$$\frac{o \text{ has component } c \qquad c\text{'s proper components are } c1, \ldots, cn \qquad c1, \ldots, cn \text{ computable on } o.c}{c \text{ computable on } o}$$

$$\frac{o \text{ has component } c \qquad c\text{'s components are } c1, \ldots, cn \qquad c \text{ computable on } o}{c1, \ldots, cn \text{ computable on } o.c}$$

- if a deep component of a component of an object is computable on the component, then it is also computable on the object:

$$\frac{o \text{ has component } c \qquad o.c \text{ has deep component } d \qquad d \text{ computable on } o.c}{c.d \text{ computable on } o}$$

- if computability of the output parameters of a dependent subtask of a relation of an object can be proved on the object under the hypothesis that the input parameters of the subtask are computable on the object, then the subtask is computable on the object:

$$\frac{\begin{array}{lr} & [ins \text{ computable on } o] \\ o \text{ has rel'n } r \text{ with dep. subt. } s \text{ with spec. } ins \text{ -> } outs \qquad & outs \text{ computable on } o \end{array}}{s \text{ computable on } o}$$

- if computability of the output parameters of an independent subtask of a relation of an object can be proved on a fresh object[1] of the class mentioned in the specification of the subclass under the hypothesis that the input parameters of the subtask are computable on the same fresh object, then the subtask is computable on the given object:

[*ins* computable on *o'*]

o has rel'n *r* with indep. subtask *s* with spec. *outs* computable on *o'*
cl | - *ins* -> *outs* (*o'* is a fresh object of class *cl*)

----

s computable on *o*

- if computability of the output parameters of an unprogammed relation of an object can be proved on the object under the hypothesis that the input parameters of the relation are computable on the object, then the relation is computable:

[*ins* computable on *o*]

o has unprogrammed relation *r* with spec. *ins* -> *outs* *outs* computable on *o*

----

r computable on *o*

[1] By a fresh object, we mean a new temporary object

8.0 Standard functions

The predefined functions of the NUT language fall into two categories: standard functions and C-functions. The main difference between these two kinds of functions is that there are restrictions on using standard function names as component names etc. (see Sec 3.3), whereas the names of C-functions are not reserved. Also, differently from standard functions, C-functions form a library in the source code of NUT. New C-functions can be added to NUT by the user by modifying the source code. C-functions are described in a separate document, *The NUT Libraries*[1]. How functions are called is explained in Sec 6.1.3.

8.1 Arithmetic, algebra

The following is a brief description of arithmetical/algebraic functions.

`sin`	sine (argument value in radians)	`(num -> num)`
`cos`	cosine (argument value in radians)	`(num -> num)`
`tan`	tangent (argument value in radians)	`(num -> num)`
`asin`	arcsine (return value in radians)	`(num -> num)`
`acos`	arccosine (return value in radians)	`(num -> num)`
`atan`	arctangent (return value in radians)	`(num -> num)`
`sqrt`	square root	`(num -> num)`
`ln`	logarithm base *e*	`(num -> num)`
`exp`	exponent base *e*	`(num -> num)`
`abs`	absolute value	`(num -> num)`
`int`	rounding to integer	`(num -> num)`
`mod`	remainder in integer division	`(num, num -> num)`

A call of an algebraic function causes a run-time error, if the function is undefined for the given argument. A call of `mod` causes a run-time error, if the arguments are not integers.

8.2 String manipulation

pos

find where/whether one string occurs as a substring in another

Synopsis

```
pos(str1, str2) : num
str1, str2 : text
```

Inputs

`str1`	the string where to search
`str2`	the candidate substring

[1] Available by anonymous ftp from `it.kth.se`, file `Software/CSlab/Software-Engineering/NUT/doc/libraries.ps.Z`.

Returns

the start position in `str1` of the first occurrence of `str2` in `str1`, if `str2` occurs in `str1`;
0, if `str2` occurs nowhere in `str1`

Errors

A run-time error arises, if `str1` or `str2` is `nil`.

copy

extract a substring from a string

Synopsis

```
copy(str, pos, len)  : text
str  : text
pos, len  : num
```

Inputs

`str`	the string from where to extract
`pos`	the start position for extraction
`len`	the number of symbols to be extracted

Returns

the result of extracting from `str` `len` symbols (or as many symbols as there remain until the end) starting at position `pos`

Error situations

A run-time error arises, if `str` is `nil`, if `pos` is not a position in `str`, and if `len` is not a positive integer.

delete

delete a substring from a string

Synopsis

```
pos(str, pos, len)  : num
str  : text
pos, len  : num
```

Inputs

`str`	the string from where to delete
`pos`	the start position for deletion
`len`	the number of symbols to be deleted

Returns

the result of deleting from `str` `len` symbols (or as many symbols there remain until the end), beginning at position `pos`

Error situations

A run-time error arises, if `str` is `nil`, if `pos` is not a position in `str`, and if `len` is not a positive integer.

insert

insert a string into another string

Synopsis

```
insert(str1, str2, pos) : num
str1, str2 : text
pos : num
```

Inputs

`str1`	the string where to insert
`str2`	the string to be inserted
`pos`	the position where to insert

Returns

the result of inserting string `str2` into `str1` at position `pos`

Error situations

A run-time error arises, if `str1` or `str2` is `nil`, and if `pos` is not a position in `str1`.

8.3 Interaction with the Graphics window

This set of functions assumes the following user classes to be present in the package:

Class `Frame`:

```
var
     x, y, dx, dy : num;
     penSize : Point;
     penShape, fillPatt, color, mode : num;
     font, fontSize, fontAttr : num;
     orgX, orgY, factorX, factorY : num;
     selectable, locked, protected, react : num;
init                                             √?
     penSize.x := 1;        penSize := 0;
     penShape := 0;
     fillPatt := 0; √?
     color := 0;
     mode := 3;
     font := 8;
     fontSize := 1;         fontAttr := 1;
     orgX := 0;             orgY := 0;
     factorX := 1;          factorY := 1;
     selectable: = 0;       protected := 0;          /
                            react := 0;              !
                            locked := 0;
```

Class `Point`:

```
var
     x, y : num;
```

Class `Line`:

```
var
     p1, p2 : Point;
```

Class `Rect`:

```
var
      x, y, dx, dy : num;
```

get_ID

get the ident of a picture element, given its ordinal number in the picture

Synopsis

```
get_ID(ordno) : num
ordno : num
```

Inputs

`ordno` the ordinal number of the element whose ident is asked

Action

Displays an error message window, if `ord` is not a valid ordinal number for the picture (enumeration of picture elements in pictures is started from 1)

Returns

the ident of the `ordno`-th element in the picture (the normal case);
0, if `ord` is not a valid ordinal number for the picture

get_line

get the geometry of a line in the picture (in the form of a `Line` object)

Synopsis

```
get_line(id) : struct
id : num
```

Inputs

`id` the ident of the line whose geometry is asked

Action

Displays an error message window, if there is no picture element with ident `id`, or the picture element with ident `id` is not a line

Returns

a `Line` object, representing the geometry of the line with ident `id` (the normal case);
`nil`, if there is no picture element with ident `id`, or the picture element with ident `id` is not a line

get_poly

not implemented

get_rect

get the geometry of a rectangle in the picture (in the form of a `Rect` object)

Synopsis

```
get_rect(id) : struct
id : num
```

Inputs

`id` the ident of the rectangle whose geometry is asked

Action

Displays an error message window, if there is no picture element with ident `id`, or the picture element with ident `id` is not a rectangle.

Returns

a `Rect` object, representing the geometry of the rectangle with ident `id` (the normal case);
`nil`, if there is no picture element with ident `id`, or the picture element with ident `id` is not a rectangle

get_oval

get the geometry of an oval in the picture (in the form of a `Rect` object)

Synopsis

```
get_oval(id) : struct
id : num
```

Inputs

`id` the ident of the oval whose geometry is asked

Action

Displays an error message window, if there is no picture element with ident `id`, or the picture element with ident `id` is not an oval.

Returns

a `Rect` object, representing the geometry of the oval with ident `id` (the normal case);
`nil`, if there is no picture element with ident `id`, or the picture element with ident `id` is not an `oval`

get_text

get the text of a textual element in the picture (in the form of a string)

Synopsis

```
get_text(id) : text
id : num
```

Inputs

`id` the ident of the textual element whose text is asked

Action

Displays an error message window, if there is no picture element with ident `id`, or the picture element with ident `id` is not a textual element.

Returns

the text of the textual element with ident `id` (the normal case);
`nil`, if there is no picture element with ident `id`, or the picture element with ident `id` is not a textual element

get_group

from among a group of picture elements, get the element with a certain ordinal number

Synopsis

```
get_group(id, ordno) : num
id : num
ordno : num
```

Inputs

`id`	the ident of the group
`ordno`	the ordinal number in the group of the desired element

Action

Displays an error message window, if there is no picture element with ident `id`, the picture element with ident `id` is not a group, or `ordno` is not a valid ordinal number for the group with ident `id` (enumeration of elements in groups is started from 1).

Returns

the ident of the `ordno`-th element in the group with ident `id` (the normal case);
0, if there is no picture element with ident `id`, the picture element with ident `id` is not a group, or `ordno` is not a valid ordinal number for the group with ident `id`

get_type

get the type of a picture element

Synopsis

```
get_type(id) : text
id : num
```

Inputs

`id`	the ident of the picture element under investigation

Action

Displays an error message window, if there is no picture element with ident `id`.

Returns

the type of the picture element as string (the possible values are: `'line'`, `'poly'`, `'rect'`, `'oval'`, `'text'`, `'group'`) (the normal case);
`nil`, if there is no picture element with ident `id`

get_frame

get the attributes of a picture element (in the form of a `Frame` object)

Synopsis

```
get_frame(id) : struct
id : num
```

Inputs

`id` the ident of the picture element to be read from

Action

Displays an error message window, if there is no picture element with ident `id`.

Returns

a `Frame` object representing the attributes of the picture element with ident `id` (the normal case);
`nil`, if there is no picture element with ident `id`

get_name

get the name of a picture element

Synopsis

```
get_name(id) : text
id : num
```

Inputs

`id` the ident of the picture element whose name is asked

Action

Displays an error message window, if there is no picture element with ident `id`.

Returns

the name of the picture element with ident `id` (the normal case);
`nil`, if there is no picture element with ident `id`, or the picture element with ident `id` does not have a name

get_status

find out whether a picture element is linked to some object or not

Synopsis

```
get_status(id) : num
id : num
```

Inputs

`id` the ident of the picture element under investigation

Action

Displays an error message window, if there is no picture element with ident `id`.

Returns

1, if the picture element with ident `id` is linked to an object;
0, if the picture element with ident `id` is not linked to an object, or if there is no picture element with ident `id`

gr_line

given the attributes and geometry for a line, draw it (and, optionally, link it to the line object)

Synopsis

```
gr_line(frame, line, status) : num
frame : Frame
line : Line
status: num
```

Inputs

`frame` specifies the attributes of the line to be drawn

`line` specifies the geometry of the line to be drawn
[must be given as a name, not expression]

`status` specifies whether to link the object `line` to the line to be drawn or not

Action

Draws a line with attributes specified in object `frame` and geometry specified in object `line`, creates an ident for the drawn line. If status is non-0, links the line to the object `line`.

Returns

the ident of the drawn line

gr_poly

given the attributes and geometry for a polyline, draw it (and, optionally, link it to the polyline object)

Synopsis

```
gr_poly(frame, poly, status): num
frame : Frame
poly : array of Point
status: num
```

Inputs

`frame` specifies the attributes of the polyline to be drawn

`poly` specifies the geometry of the polyline to be drawn
[must be given as a name, not expression]

`status` specifies whether to link the object `poly` to the polyline to be drawn or not

Action

Draws a polyline with attributes specified in object `frame` and geometry specified in object `poly`, creates an ident for the drawn polyline. If status is non-0, links the polyline to the object `poly`.

Returns

the ident of the drawn polyline

gr_rect

given the attributes and geometry for a rectangle, draw it (and, optionally, link it to the rectangle object)

Synopsis

```
gr_rect(frame, rect, status): num
frame : Frame
rect : Rect
status: num
```

Inputs

`frame` specifies the attributes of the rectangle to be drawn

`rect` specifies the geometry of the rectangle to be drawn
[must be given as a name, not expression]

`status` specifies whether to link the object `rect` to the rectangle to be drawn or not

Action

Draws a rectangle with attributes specified in object `frame` and geometry specified in object `rect`, creates an ident for the drawn rectangle. If status is non-0, links the rectangle to the object `rect`.

Returns

the ident of the rectangle drawn

gr_oval

given the attributes and geometry for an oval, draw it (and, optionally, link it to the rectangle object)

Synopsis

```
gr_oval(frame, rect, status): num
frame : Frame
rect : Rect
status: num
```

Inputs

`frame` specifies the attributes of the oval to be drawn

`oval` specifies the geometry of the oval to be drawn
[must be given as a name, not expression]

`status` specifies whether to link the object `rect` to the oval to be drawn or not

Action

Draws a oval with attributes specified in object `frame` and geometry specified in object `rect`, creates an ident for the drawn oval. If status is non-0, links the oval to the object `rect`.

Returns

the ident of the drawn oval

gr_text

given the attributes, upper-left point, and text (string or number) for a textual picture element, draw it (and, optionally, link it to the textual object)

Synopsis

```
gr_~~poly~~text(frame, point, str, status): num
frame : Frame
point : Point
str : text or num
status: num
```

Inputs

`frame` specifies the attributes of the textual element to be drawn

`point` specifies the upper left corner of the textual element

`str` specifies the text of the textual object to be drawn
[must be given as a name, not expression]

`status` specifies whether to link the object `str` to the textual element to be drawn or not, and whether the textual element shall be single-line or multi-line

Action

Draws a textual element with attributes specified in object `frame`, with upper-left point `point`, and text `str` (if `str` is a number, then it is converted into a string); creates an ident for the drawn textual element. If `status` is 0, the textual element is multi-line and linked to the object `str`; if 1, it is multi-line and not linked; if 2, it is single-line and linked to `str`; if 3, it is single-line and not linked.

Returns

the ident of the drawn textual element

gr_group

group a set of picture elements

Synopsis

```
gr_group(ids) : num
ids : array of num
```

Inputs

`ids` the array of the idents of the picture elements to be grouped

Action

Groups the picture elements with idents from the array `ids`, deletes the mentioned idents from the table of idents, and invents an ident for the group. Displays an error message window, if some of the elements of the array `ids` identifies no element of the picture.

Returns

the ident of the created group (the normal case);
0, if some of the elements of the array `ids` identifies no element of the picture

put_frame

change the attributes of a picture element

Synopsis

```
put_frame(frame, id) : num
frame : Frame
id : num
```

Inputs

`frame`	specifies the new values of the attributes of the picture element to be modified
`id`	the picture element to be modified

Action

Modifies the picture element with ident id, according to the attribute values given in `frame`. If `frame` is `nil`, the default frame is used.

Returns

0

put_name

assign a name to a picture element

Synopsis

```
put_name(name, id) : num
elname : text
id : num
```

Inputs

`elname`	the name to be assigned to a picture element
`id`	the ident of the picture element to be assigned a name

Action

Assigns name `elname` to the picture element with ident `id`. Displays an error message window, if there is no picture element with ident `id`.

Returns

0

del_elem

delete a picture element

Synopsis

```
del_elem(id) : num
id : num
```

Inputs

`id`	the ident of the picture element to be deleted

Action

Deletes the picture element with ident `id`. Displays an error message window, if there is no picture element with ident `id`.

Returns

0

del_pict

clear the picture (delete all picture elements)

Synopsis

```
del_pict() : num
```

Inputs

none

Action

Clears the picture.

Returns

0

reshow

reshow a picture element

Synopsis

```
reshow(id) : num
id : num
```

Inputs

`id`	the ident of the picture element to be reshown

Action

Reshows the picture element with ident `id`.

Returns

0

reshow_all

reshow the whole picture

Synopsis

```
reshow_all() : num
```

Inputs

none

Action

Reshows the whole picture.

Returns

0

add_elem

not implemented

add_pict

add a picture from a file to the picture in the Graphics window at a specified position, return the geometry of the added picture

Synopsis

```
add_pict(filename, point) : struct or num
filename : text
point : Point
```

Inputs

`filename`	the name of the file from where a picture is to be read
`point`	the upper left point for placing the picture

Action

Reads a picture from a file named `filename` (default extension `.pic`), and adds it to the picture in the Graphics window, placing its upper left corner at point `point`. If `point` is `nil`, then the picture is placed exactly as given in the file. Displays an error message window, if there is no file named `filename`, or the file named `filename` is in a wrong format.

Returns

a `Rect` object, representing the geometry of the added picture (the area occupied by the added picture) (the normal case);
0, if there is no file named `filename`, or the file named `filename` is in a wrong format

link_pict

add a picture from a file to the picture in the Graphics window at a specified position, return the idents of the elements of the added picture

Synopsis

```
link_pict(filename, point) : array of num or num
filename : text
point : Point
```

Inputs

`filename`	the name of the file from where a picture is to be read
`point`	the upper left point for placing the picture

Action

Reads a picture from a file named `filename` (default extension `.pic`), and adds it to the picture in the Graphics window, placing its upper left corner at point `point`. If `point` is `nil`, then the picture is placed exactly as given in the file. Displays an error message window, if there is no file named `filename`, or the file named `filename` is in a wrong format.

Returns

an array of the idents of the elements of the added picture (the normal case);
0, if there is no file named `filename`, or the file named `filename` is in a wrong format

link_name

link the picture element with a given name to a given object (the picture element must have been created by `link_pict`)

Synopsis

```
link_name(elname, obj) : num
elname : text
```

`obj` : arbitrary graphical class (`Line`, `array of num`, `Rect`, `text`)

Inputs

`elname`	the name of the picture element to be linked to an object
`obj`	the object to be linked to a picture element

Action

Finds a picture element named `elname`, and links it to object `obj`. Displays an error message window, if there is no picture element named `elname`, the type of picture element named `elname` and the class of object `obj` are incompatible, or the picture element named `elname` was not created by `link_pict`.

Returns

the ident of the linked picture element (the normal case);
0, if there was no picture element named `elname`, the type of the picture element named `elname` and the class of object `obj` are incompatible, or the picture element named `elname` was not created by `link_pict`

save_elem

not implemented

save_pict

not implemented

8.4 Miscellaneous

chin

transform a string representation of an integer into a number

Synopsis

```
chin(strrep) : num
strrep: text
```

Inputs

`strrep`	the string representation of a number

Returns

the integer whom the string `strrep` represents (the integer is formed from the digit symbols in `strrep` until the first non-digit symbol; as an exception to this general rule, the symbol + or - in the 1st position counts as a sign)

rech

transform a number (integer or real) into its string representation

Synopsis

```
rech(number) : text
number : num
```

Inputs

`number` the number whose string representation is asked

Returns

the string representation of the number `number`

length

get the length of a string, the number of first-level components of a compound object

Synopsis

```
length(obj) : num
obj : arbitrary class
```

Inputs

`obj` the object whose length is asked

Returns

the number of symbols in `obj`, if `obj` has class `text`;
the number of components in `obj`, if `obj` has a compound class;
0, otherwise