ProSet — Prototyping with Sets
Language Definition[1]

Ernst-Erich Doberkat
Wolfgang Franke
Ulrich Gutenbeil
Wilhelm Hasselbring
Ulrich Lammers
Claus Pahl

University of Essen
Fachbereich Mathematik und Informatik — Software Engineering
Schützenbahn 70, 4300 Essen 1, Germany
pst@informatik.uni-essen.de

April 16, 1992

[1] This is Informatik-Bericht 02-92

Abstract

This document is the defining manual for the programming language ProSet. The name is an acronym for Prototyping with Sets. This language has been defined and is currently being implemented at the University of Essen; it is a descendant of the set-oriented prototyping language SETL.

Contents

List of Tables

1 Introduction

This document is the defining manual for the programming language ProSet. The name is an acronym for Prototyping with Sets. This language has been defined and is currently being implemented at the University of Essen; it is a descendant of the set-oriented prototyping language SETL.

This introductory chapter is intended to provide some thoughts regarding the definition of this language. In particular we will have a brief look at prototyping and how it might influence language design. We will discuss the relationship of ProSet to SETL, and we will discuss several salient features of our language proposal.

1.1 Prototyping

It is well-known that the classical software life cycle has some drawbacks which suggest that it should be complemented by some auxiliary activities. This is true in particular for the early phases. One of the main drawbacks is the lack of support to experimental or exploratory programming. Somewhat related to this problem is the observation that the user's involvement in designing a program is kept to a minimum. Basically the user is only involved during the very early phases when it comes to more or less informally stating the requirements, and at a rather late phase when it comes to acknowledge the functionality of the program. This observation is particularly striking when modeling user interfaces, but it is not restricted to that area. Prototyping tries to find a way out of these problems by assigning the user a more active rôle during requirements elicitation, and by making experimental and exploratory programming part of the activities related to program design. This approach to program construction may complement the life cycle approach by incorporating a prototype subphase between planning and requirements definition during the analytic phase. Boehm's spiral model also takes prototyping into account by proposing prototyping phases to be carried out after risk analysis and assessment.

Having a look at the literature it is difficult to find a concise definition of software prototyping since this is really some sort of umbrella term, covering a multitude of activities more or less related to each other. We stick to the description given by Christiane Floyd: "Prototyping . . . refers to a well-defined phase in the production process where a model is produced in advance, exhibiting all the essential features of the final product for use as test specimen and guide for further production." This description emphasizes that prototyping really means modeling of software, it implies that the model itself should be an executable program. Moreover, it is seen from this description that prototyping should be an activity aiming at the *rapid* production of a piece of software, since otherwise the effects of modeling would be lost. This in turn implies that a language for the support of software prototyping should provide powerful features, in particular versatile data structuring facilities together with convenient control structures operating on these complex data structures.

Consequently we need powerful mechanisms based on a somewhat natural formal calculus. We emphasize a *natural* approach here since it should be possible to express one's thoughts for constructing a program in a programming language rather close to the way one does express things mathematically. Finite set theory provides such a way of cleanly expressing one's thoughts, and our proposal for a prototyping language is based on set theory augmented by bits and pieces from λ-calculus.

1.2 SETL

Using set theory for the purpose of formally describing program designs is by no means new, and the most prominent programming language making finite sets over finite domains available has been SETL. This venerable language was designed during the seventies at New York University's Courant Institute of Mathematical Sciences by J.T. Schwartz and his group. The late seventies, and the early eighties saw implementations of this language on a variety of machines ranging from mainframes to work stations. Subsequently, the language has been used, and has proven the modeling capacities of the language in a convincing way. Highlights are

- the development of the first ADA compiler (certified in April 1983),
- the SETL optimizer (which really was an encompassing prototype of optimization techniques for procedural languages),
- the Rutgers Abstract Program Transformation System RAPTS,
- WAA, a tool for analyzing PASCAL program fragments with respect to their potential for reuse.

The day-to-day use of SETL, however, indicated that the language is not free of problems. First of all, there is not a really satisfying programming environment, and the language itself displayed some very baroque features which sometimes more hindered using the language than supported it. This applies particularly to *programming in the large*, the organization of separately compiled components was felt to be rather awkward. In addition, the arsenal of data structures was considered incomplete since functions as citizens with first class rights are missing, the possibilities of making values persistent are felt as a lack and parallel programming is not possible at all. The programming environment was the subject of the ESPRIT project **SED** during 1986 to 1989. Some progress has been made here, too (for example establishing a component translating SETL programs to a production language like ADA), but regrettably the chance of integrating all the results into a coherent and uniform programming environment was missed, mainly due to problems in the project management.

When we had a look at SETL we decided that we wanted to reimplement it, clean up some of the features and incorporate constructs we felt would be helpful. Reimplementation occurred to be necessary since SETL was originally implemented in a systems implementation language called LITTLE. This language in turn was developed at Courant Institute for the implementation of SETL, it is probably known outside of New York University to some twenty people[1].

When working on the new language design and observing the design of SETL2 proposed by Kirk Snyder of Courant Institute we decided to incorporate some features into this new language. It will become apparent from the rest of this manual that the following is new

- Data abstraction is supported by the new data types `function`, `module`, and `instance`.
- Control abstraction is supported by a variety of constructs for exception handling.
- Data modeling is supported by persistence; each and every value having first class rights in the language may be made persistent.
- Parallel programming is supported by features for generative communication; the control primitives provided by the LINDA model for concurrent programming serve as a basis for some primitive operations in our language.

To avoid confusion between SETL and its variants, and to add a stone to the Tower of Babel we decided to give the language a new name.

1.3 A Brief Overview

This section provides a brief discussion of some of PROSET's features which might be of interest. We will discuss issues pertaining to the type system, to making values persistent, and to generative communication.

[1] including four of them in Europe: Eugenio, Yo, Philippe and Ernst

1.3.1 Data Types

ProSet makes — as its predecessor SETL — the data types from finite set theory available. Before discussing this let us have a brief look at the primitive data types provided by the language: the primitive types `integer`, `real`, `boolean`, `string`, and `atom` are of course available. The first two data types do not offer any surprising properties with the probable exception of prohibiting implicit conversions from integers to reals. Characters are special case of strings, and atoms are uniquely created values permitting the explicit internal representation of external objects in the same way as these things are handled in languages like Lisp. Atoms are not only unique with respect to a particular run of a program, but we have made an attempt to preserve uniqueness across program executions and even machines. This implies that atoms may be exported from and imported to programs. Compound data types include finite sets and finite tuples; these objects have their usual mathematical semantics, in particular we point out that we deal here with value semantics rather than with pointer semantics. Consequently, copying a compound value and modifying the copy will not affect the original. Sets may be described as familiar in mathematics, viz., by enumerating the elements and by describing their elements through properties. The same applies to tuples. Having these types available it is easy to construct mathematical maps and relations by simply forming subsets of a Cartesian product. All these data types are accompanied by the usual operations (intersection, union, concatenation etc.). Thus the convenience of using finite set theory for describing solutions to problems is fully available.

1.3.2 Control Structures

The control structures are rather canonic: we provide the usual arsenal of control structures deriving from e.g. ALGOL and define some operations which take the available compound data types into account. It is for example possible to iterate over a set and perform an operation for each element of this set, or to test whether or not some property is true for each element of a tuple.

1.3.3 Procedures

Procedures are polymorphic and return a value. This is parametric polymorphism in contrast to predefined operators, which are just overloaded. Parameters may be passed by value, by result, and by value/result. This is very similar to SETL, in addition it is possible to define anonymous functions (λs). Procedures and λs may be converted into values of type `function` using a closure operator. The closure of a procedure freezes the value of all non-local objects. Visibility of names is restricted by default to the range in which the name occurs, since procedures and modules may be nested, the declaration of a name as `visible` propagates visibility into local scopes; shielding a name from being visible may be done using a `hidden` declaration. This is somewhat different from the usual model of inheriting visibility from outermost scopes as observed e.g. in Pascal. The usage in SETL has convinced us, however, that it does in fact make sense to handle things in the way described here. Functions (i.e. the respective results of applying the `closure` operator) obtain an identity in a rather straightforward way, consequently these values may be handled as any other value with an identity, in particular these values may be elements of sets, arguments to procedures and functions, and they may be returned by them. This is quite similar to but subtly different from the way things are done in SETL2 programming language.

1.3.4 Exception Handling

In many high level programming languages the occurrence of errors leads to sometimes unintended program termination. Through an exception handling mechanism we integrate a device for dealing gracefully with errors in the program. For dealing flexible with a large class of situations we extend the notion of *error* handling to *exception* handling. An exception is a non-normal situation occurring in the course of executing a program unit which has to be handled by the invoking unit. Thus exception handling is also a device for structuring and modeling, i.e. a device for concisely formulating the algorithm and for separating exceptional conditions and their handling from the algorithm.

An important improvement to early approaches to exception handling is the distinction of exceptions and their handling units. For the sake of flexibility ProSet provides a construct for dynamic association of handlers with exceptions. In handling exceptions ProSet supports both a termination and a resumption model, i.e. the execution of the exception raising unit may be terminated or resumed. This is determined dynamically.

1.3.5 Modules and Instances

We have described so far the facilities and devices for *programming in the small.* Modules and instances are used for the support of *programming in the large.* Modules are templates describing the operation of functions around a common data structure. The objects imported to, and those exported from a module are described in the interface to a module by giving its name and the way the module treats the corresponding object. Modules have to be instantiated yielding instantiations before the services they provide may be used. Note that in accordance with the philosophy of the language we do not specify the type of the imported or the exported values, so the polymorphism of procedures is carried a step further. Modules are somewhat similar to generic packages in Ada. This kind of package has to be instantiated in order to be usable. In a similar way we have to instantiate a module by indicating what the imported values are. The result of such an instantiation is a value of type `instance`. Only after having instantiated a module the values being exported from a module may be used.

Modules provide a data type of their own. The same is true for instances. This has as a consequence that modules may serve as parameters to procedures and may be returned from them as values. Since values of each type may be made persistent it is possible to deal with separate compilation as well as loading and binding of program units in a very flexible way. So a module is separately compiled by making it persistent, and an instance of a separate compiled and instantiated module is used by fetching the instance from the persistent store.

1.3.6 Persistence

Modeling does not only apply to programs, but also to data: in the process of developing an application not only the algorithms have to be explored, but the data and data structures on which the algorithms are to work may emerge from this explorative activity as well. Semantic data models working with objects, attributes and ISA-*relationships* investigate ways of modeling data according to their semantic content. They are used for designing record-oriented schemata with an approach somewhat similar to the one used in software prototyping, but rather than modeling programs high-level representations of data are modeled. This model is mapped into a lower-level structure. Data modeling should accommodate the user by making the representation and manipulation as close as possible to the user's perception of the problem. It is well accepted in the data base research community that data modeling should accommodate the user by making the representation and manipulation as close as possible to the user's perception of the problem. Consequently, it is desirable to

- model data according to the user's needs,
- refine data representations iteratively (which requires access to previously formulated data models),
- re-use patterns or templates of previously formulated data models,
- share data between different users or different prototyping sessions.

We see that there are in fact striking similarities between prototyping programs and modeling data. Both construct a model to be experimented with, and eventually to be transformed into a production version. Thus software prototyping will be most effective and have maximal impact when it caters for program as well as for data modeling, hence when it is supported by a programming language which is able to serve the software engineer (who wants to model programs) as well as the data engineer (who wants to construct a semantic model of his or her data). These considerations suggest introducing a facility for handling persistent data in ProSet. In principle this would be possible through the use of (binary) files, but this sort of repository for data is not powerful enough for sophisticated applications. So we decided to design a special abstract data type called *P-file*, faintly resembling archives under UNIX, which would help preserving structures and which would be more convenient to use. Persistence comes as a property orthogonal to types, so each and every value having a legal type in ProSet may be made persistent.

The guiding metaphor for introducing and handling persistent values is that of an architect having to design a house. She will usually not design the plan from scratch but will rather try to master that task by developing only some feasible solutions from the very beginning on the drawing board and by customizing parts of previously developed designs from formally drawn blueprints. This also includes making mental notes that some pieces of the present design may be of use in other situations. Hence the history of the design will resemble a quilt, new pieces coexisting with modified old ones. Blueprints are stored in archives, each archive being identified somehow, and usually accessible to a whole community of architects. Each blueprint in turn may be fetched from an archive, it is identified by a name and endowed with particular attributes like a date, proprietary notes, material to be used etc. So we see our architect working on the drawing board, accessing and customizing pieces of designs, and progressing towards a complete plan for the house.

1.3.7 Programming Parallel Applications

Since applications which are inherently parallel should be programmed in a parallel way, it is most natural to incorporate parallelism into the process of model building. Opportunities for automatic detection of parallelism in existing programs are limited and furthermore, in many cases the formulation of a parallel program is more natural and appropriate than a sequential one. Most systems in real life are of a parallel nature, thus the intent for integrating parallelism into a prototyping language is not only that of increasing performance. It is intended to provide a tool for prototyping parallel algorithms and modeling parallel systems. However, parallel programming is conceptually harder to undertake and to understand than sequential programming, because a programmer often has to focus on more than one process at a time. Programming in LINDA provides a spatially and temporally unordered bag of processes. Each task in the computation can be programmed (more-or-less) independently of any other task. This enables the programmer to focus on one process at a time thus making parallel programming conceptually the same order of problem-solving complexity as conventional, sequential programming. Process communication and synchronization in LINDA is reduced to concurrent, associative access to a large data pool, thus relieving the programmer from the burden of having to consider all process inter-relations explicitly. Programming parallel applications in ProSet is presented in section 10. The concept for process creation via Multilisp's futures is adapted to set-oriented programming and combined with the concept for synchronization and communication via LINDA's tuple space.

1.4 Acknowledgements

The comments on the preliminary definition by Fritz Henglein as well as the comments on a draft of this paper by H.-G. Sobottka are gratefully acknowledged.

2 Notational conventions

The definition for syntactic constructs in this document is sometimes given by the familiar syntax charts, which are integrated into the regular flow of reading. In these syntax charts, nonterminals are put into a rectangular box whereas ovals are put around terminals. The nonterminal on the left hand side of a grammar production is not being put in any box. Essentially, the syntax charts describe productions of the underlying context free grammar but since they can describe repetition and optional parts, they can combine more rules which conceptually belong together.

Disclaimer Since the main goal of providing syntax charts is to improve readability, we do not claim that the list of all syntax charts is complete and could serve to define the whole language. It should be mentioned, too, that multiple definitions for the same nonterminal are to be read as possible alternatives, not as redefinitions.

In Appendix C the complete grammar is listed in Backus-Naur-form (BNF) decorated with some comments.

3 Lexical conventions

Tokens

The tokens recognized by the language fall into the exactly one of the categories keywords, identifiers, operator identifiers, literals (integer, real, string), and special characters.

Some of the special characters may be combined with others to form single tokens, e.g. `:=` denotes the assignment symbol and is interpreted as a single token.

It might be useful to state informally here what 'token' is supposed to mean: it is a lexical unit to be accepted by the lexical analyser of the language. It cannot be constructed by means of the language — this is important to realize when the macro processor is discussed.

3.1 Comments

Comments allow to drop textual information into a program. Since comments are interpreted by the lexical analyser as white space, see sect. 3.2, they will separate tokens.

In ProSet comments come in two forms. The first one is Ada-like; it begins with a double dash `--` and ends with the end of the line. In the second, Modula-2-like form comments are enclosed in `(*` and `*)`. Everything after reading the two characters `(*` will be ignored until the first occurrence of the characters `*)`. The compound symbols `--`, `(*` and `*)` do not permit intermediate blanks.
Double-dash comments inside `(* ... *)` comments will do no harm as they are ignored — real double-dash comments, however, may hide a `(*` sequence as a beginning of a multi line comment and almost certainly force an error.

3.2 White Space

White space can be used to separate tokens. Comments behave like white space, as mentioned above. Other forms of white space are: blanks, tabs, newlines, returns, vertical tabs and formfeed according to ANSI-C standard.

3.3 Keywords

Keywords are reserved words in ProSet, they are non-empty sequences of lowercase alphabetic characters and should neither be redefined nor used as identifiers. Some keywords contain an underscore character. A complete list of ProSets keywords is given in table 1.

```
abort
and
arb
argv
at
atom
begin
blockiffull
boolean
case
closure
constant
continue
deposit
do
domain
drop
else
elseif
end
endm
eput
eputf
escape
exists
false
fetch
fget
fgetf
for
forall
fput
fputf
from
fromb
frome
function
get
getf
handler
hidden
if
in
include
instance
instantiate
integer
into
is_map
is_smap
lambda
less
lessf
local
loop
macro
max
meet
min
mod
modtype
module
newat
not
notify
notin
om
or
others
pass
persistent
pow
procedure
profile
program
put
putf
quit
random
range
rd
real
repeat
resume
return
rw
self
set
signal
stop
string
subset
then
true
tuple
type
until
use
visible
when
while
whilefound
with
wr
```

Table 1: Reserved words

3.4 Identifiers

Identifiers are sequences of alpha-numeric characters or the underline character _ beginning with an alpha character. Case is significant. The length of identifiers may be restricted by the implementation.

3.5 Literals

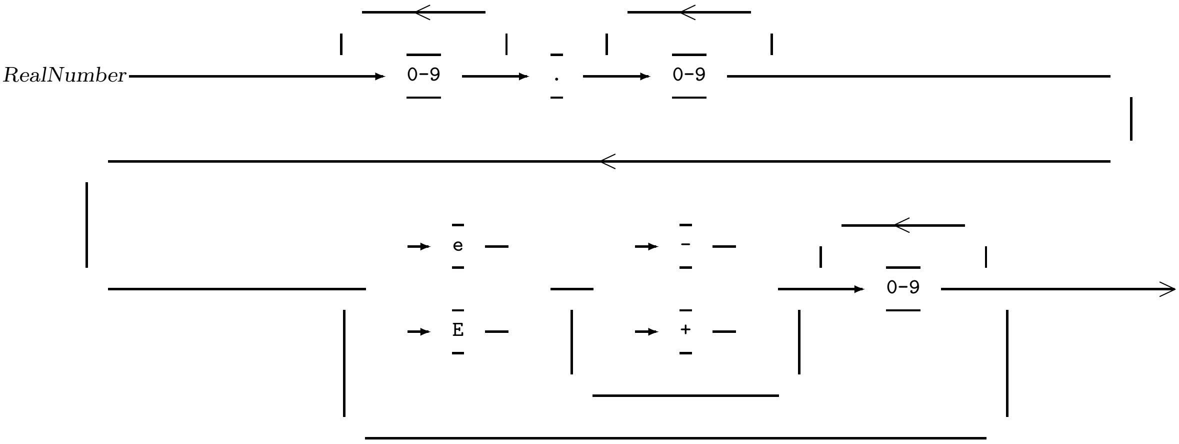
Literals come as numerical and string literals. Numerical literals (numbers) are always given base 10. There are two types of numbers: integer and floating point numbers — size and accuracy depend on the implementation, but we make sure that at least double precision (with respect to C) for floating point arithmetic is used. The syntax charts provided for describing literals differ from the syntax charts used elsewhere in this document in that they do *not* allow intermediate spaces — literals are lexical units.

3.5.1 Integer Literals

Integer literals are represented by nonempty sequences of digits. We make sure that integers have at least the size of an `int` in C. Any implementaion may check arithmetic overflow.

3.5.2 Floating Point Numbers

Floating point numbers (also called real numbers) are written as a nonempty sequence of digits, followed by a period, followed by a nonempty sequence of digits, optionally followed by an `e` (upper- or lowercase), and then an optional sign (+ or −), and then an integer.



Valid floating point numbers are

```
100.0     0.1    1.5E-3  0.05e+3
```

but not

`100     1E10    .1    1.E3    1.0e` (The first number is an integer literal)

Note that digits both before and after the period are required and that all numbers (including integers) are unsigned—unary minus may be used to define negative numbers.

Over- or underflow in a representation of real numbers may result in an error, either at compile time or at run time. Loss of least significant digits is not supposed to be reported.

3.5.3 String Literals

String literals are (possibly empty) sequences of characters. All characters of the character set of the underlying machine are allowed, non-printable characters and some special characters must be represented by *escape sequences*, which are introduced by the character `\`.

We assume that at least all alphabetic characters, all digits and most of the special characters are contained in the character set.

String literals are enclosed by double quotes `"`. They may not be split across lines. The next line shows, as an example, a string containing the escape-sequence `\n` representing a newline character:

```
"This is a string,\n which contains a newline-character"
```

All characters can be represented by escape-sequences similar to the ANSI-C-Standard. The unpopular representation trough trigraphs, which allow to write e.g. `??(` for `[` is not supported.

Table 2 lists all the escape-sequences defined in ProSet. The last line of the table shows a rule allowing for insertion of double quotes into strings: `"\""` is a string literal (of length 1) containing only the double quote character. Since the backslash `\` introduces escape-sequences, it has to be preceded by another backslash to give one in the string: `"\\"` is a string containing one backslash.

Characters do not form a data type on their own—they are represented as strings of size 1 whenever needed.

`\a`	alert signal (bell)
`\b`	backspace: move active position back one character
`\f`	form feed
`\n`	newline
`\r`	return
`\t`	tab (horizontal)
`\v`	vertical tab
\nnn	character, which has the octal representation *nnn* (1–3 octal digits)
`\x`n^+	character represention by arbitrary number of hexadecimal digits, upper or lower case
\c	*c* for all other characters

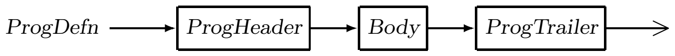
Table 2: Escape sequences

4 Program Structure

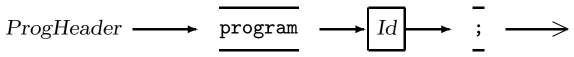
In this section we will discuss the overall structure of ProSet programs. This includes an introduction of those syntactic constructs allowing to divide a program into conceptual units. Next we provide a comprehensive look at the communication of ProSet programs with the environment. This will help the reader to understand the examples in the remainder of the document. We then deal with the basic notions relating to visibility control, the declaration of constants and variables, and the various kinds of procedures.

4.1 Overall Program Structure

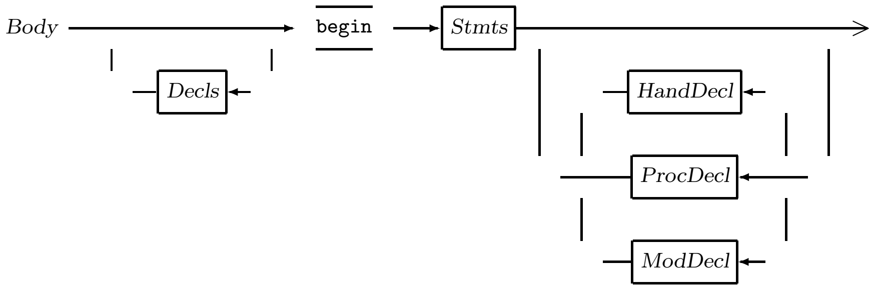
The overall structure of a ProSet program is as follows:



A program begins with a header:

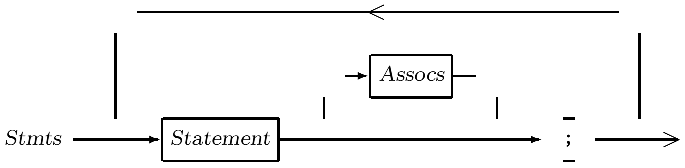


The body of a program has the form:



The body of a procedure (section 4.5), of an exception handler (section 8.5), and of a module (section 11) is constructed according to the same syntax. A body starts with an optional declaration section containing constant and variable declarations. See section 4.4 for more details.

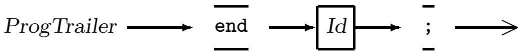
The keyword `begin` separates the declaration section from the list of statements constituting in case of a program body the main procedure of the program:



Note, that the semicolon is used as *terminator*, not as separator as in Pascal. Statements and associations of exception handlers will be explained in section 7 and 8.4, respectively.

A body continues with a sequence of declarations for exception handlers, procedures and modules in any order. In case of a program body the names of those declarations will be visible at the top level of the program and in all enclosed program units (procedures, modules, and exception handlers) not containing a declaration of the same identifier. The definitions of exception handlers, procedures, and modules appear at the end of a body, not at the beginning. This allows to read a program in a top-down manner.

A trailer marks the end of a program. The identifier in the trailer has to be identical to the program name in the header.



4.2 Communication with the Environment

ProSet provides several capabilities for communicating with the operating environment. They include facilities for operating on the persistent store, for input/output, for passing parameters to ProSet programs, and for executing a command by the host environment.

4.2.1 Persistent Store

A persistent value is one whose lifetime extends beyond the termination of the program defining and using it. Persistent values will be stored in containers called *P – Files*. By a persistent declaration a program gets access to a persistent value. The program in figure 1 reads a function object identified by the identifier `demo` from the *P – File* `MyProject` and invokes it.

The mechanisms for handling persistent values will be discussed in section 9.

4.2.2 Input/Output

The input/output operations include the facilities for reading from standard input, for writing to standard output and standard error, resp., and for handling files. The latter ones are not part of the language, but are provided by a predefined library of persistent objects, the *standard library*, which can be accessed via the *P – File* `StdLib`. For more details see section 12 and section 13. The ProSet *hello world* program in figure 2 writes the string `"hello world"` to the output using the operation `put`.

```
program simple;
    persistent constant demo : "MyProject";
begin
    demo();
end simple;
```

Figure 1: Use of a persistent function object.

```
program HelloWorld;
begin
    put("hello world");
end HelloWorld;
```

Figure 2: A simple *hello world* program.

4.2.3 Program Parameters

String parameters to be transmitted to a ProSet program may be provided on the command line of the operating system which initiates the execution of a ProSet program. We do not cater for parameters other than strings. The precise external form in which the parameters should be given depends on the command processor being used. The parameters are collected in the built-in constant `argv` of type `tuple`. The first component of `argv` contains always the name of the executable file. This mechanism is known from C.

4.2.4 The System Function

The `system` function provided by the *standard library* takes one argument and returns an implementation-defined value. The argument must be a string or `om`; otherwise the exception `type_mismatch` is raised. If the argument is equal to `om`, the call is interpreted as a request to see whether a command processor exists. The function returns `false`, if there is no command processor, or `true` to indicate that a command processor exists. If the argument is a string, it is passed to the command processor of the operating system to be executed. The string should be a valid command for the command processor. Note that no error checking is done by ProSet's runtime system on the command string. The program execution will be suspended until the command is completely executed. The call returns the return code of the executed command.

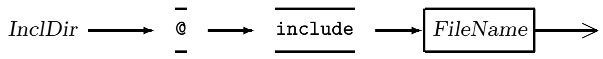
The function should be used carefully: for example the operating system might access open files or even terminate the executing program.

4.2.5 Program Termination

The last statement of the main procedure is an implicit `stop` statement (cf. 7.3.2). Some command processors interpret the return value or *status* of a program. In ProSet the return value of the program may be influenced by the optional argument to the `stop` statement.

4.2.6 Include Directive

Sometimes it is convenient to make code being externally specified available through a single directive. Comparable to the `#include`-directive in C or some Pascal implementations, ProSet supports the inclusion of text files:



After a leading `@` (instead of `#`, because the latter is used as a predefined operator) and the keyword `include` follows an implementation dependent specification of the file to be included. The directive can be used anywhere in the program, but the `@` must be the first *non-whitespace* character on a line. During compilation the `include` line is replaced by the text in the specified file.

4.3 Visibility

This section introduces the basic notions relating to visibility control.

A *declaration* associates an identifier with with a declared entity. This can be either a constant, a variable, a label, an exception, an exception handler, a procedure, a module, or a parameter of an exception handler, of a procedure, or of a module. The occurrence of an identifier in its declaration is called a *defining occurrence* of that identifier. In addition to explicit declarations, ProSet allows implicit declarations (cf. section 4.4.1), i.e. an identifier not declared so far is supposed to be declared upon occurring for the first time. If an identifier is used with its associated meaning its occurrence is called *applied.* Every applied occurrence refers to at most one defining occurrence. The portion of the program text within which we can use an identifier with the associated meaning is called the *scope* of that defining occurrence. In ProSet we can determine the scope of a defining occurrence statically by the syntactic construct in which it is *directly* contained. By a *range* we mean a syntactic construct that may contain declarations. For more details see appendix A (`local` declarations in macros are directives to the macro processor and no longer visible after the macro processor is done). Inner ranges are not part of outer ranges[2]. In ProSet any range may contain at most one defining occurrence of an identifier. The only exception to this rule are labels (section 7.4) and exceptions (section 8).

The scope of an identifier may be restricted to the range within which it is defined, but without the enclosed ranges corresponding to exception handlers, procedures, or modules. We call this a *local* respectively a *hidden* declaration.

Otherwise, if the scope of a defining occurrence includes its range and all enclosed ranges not containing a defining occurrence of the same identifier, we speak of a *visible* declaration.

To control the visibility of identifiers explicitly, ProSet provides the keywords `visible` and `hidden`. The scope rules apply to the visibility of constants, variables, exception handlers, procedures, modules, and formal parameters. The example in figure 3 illustrates the scope rules.

The program produces the output:

```
output: 5
```

The variable `x` declared at the top level of the program is visible in the body of the procedure `q`. The scope of the `hidden` declaration of `x` is restricted to the range associated with `p`.

4.4 Constant and Variable Declarations

In the sequel we discuss the variable declarations and constant declarations in ProSet.

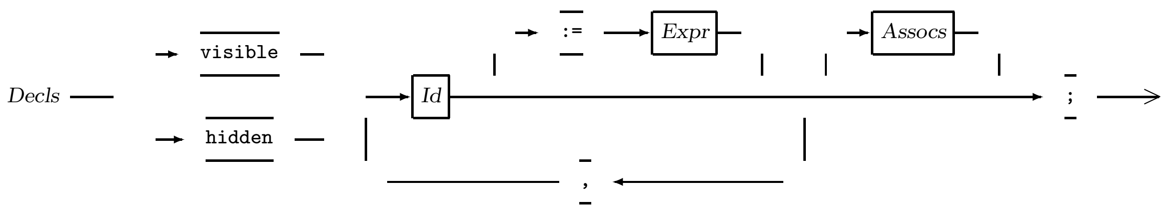
4.4.1 Variable Declarations

The syntax of an explicit variable declaration is as follows:

[2] This is part of ProSet's Setl heritage.

```
program Visibility;
    visible x := 5;
begin
    p();
    procedure p();
            hidden x := 0;
    begin
            q();
            procedure q();
            begin
                putf("output: %d\n", x);
            end q;
    end p;
end Visibility;
```

Figure 3: Example of the visibility of declarations.



An explicit variable declaration must be prefixed by one of the keywords `visible` or `hidden`, which control the visibility of those declarations.

There are two kinds of explicit variable declarations in ProSet. The first one consists of a list of identifiers, which will be defined as variables in the current range (i.e. the main procedure, the procedure, the exception handler, or the module). These variables may be initialized when the corresponding range, in which they are declared, is elaborated. For explaining the meaning of *elaboration* we refer to the definition found in the Ada reference manual[3]: "Elaboration is the process by which a declaration achieves its effect. For example it can associate a name with a program entity or initialize a newly declared variable".

The elaboration of a range generally consists of the following steps:

(a) First, the identifiers of all explicit declared entities (i.e. variables, constants, exception handlers, procedures, modules, or parameters) are introduced. This may hide other identifiers declared in surrounding ranges.

(b) The second step is the creation of the declared entities, i.e. the identifiers of exception handlers, of procedures, and of modules can be used as name of the corresponding entities; constants and variables have the undefined value `om`.

(c) In the next step, implicit associations between handler and exceptions (section 8.4) will be performed, if any.

(d) Finally, constants and variables may be initialized optionally combined with explicit associations between handlers and exceptions (see 8.4). The initializations are performed in the order in which they are written. The initial values may result from the evaluation of the expressions on the

[3] Reference Manual for the Ada Programming Language, United States Department of Defense, Washington D.C., November 1980 (Appendix D)

```
program ImplDecl;
begin
    y := 42;
end ImplDecl;
```

Figure 4: Implicit variable declaration.

right-hand side of the assignment symbol, or they may be loaded from the persistent store (see section 9).

The following example

```
visible x := 5, y := x;
```

is equivalent to

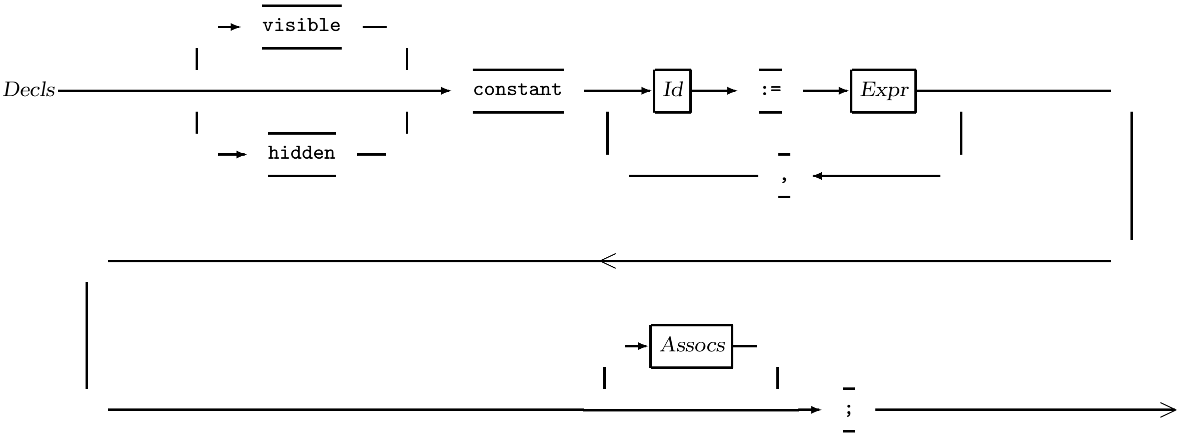
```
visible x := 5;
visible y := x;
```

The second form of an explicit variable declaration is associated with the initialization by values from the persistent store (section 9).

The declaration of variables is not mandatory in ProSet. Variables may be declared implicitly. Their scopes are restricted to the ranges within which they are introduced. The program `ImplDecl` in figure 4 demonstrates the use of an implicitly declared variables. The variable `y` is treated as if it would be declared as `hidden`.

4.4.2 Constant Declarations

A constant declaration has the form:



Constants are declared using the keyword `constant`. Their visibility is controlled by the optional keywords `visible` or `hidden`. *By default* a constant declaration is hidden, i.e.:

```
constant pi := 3.14159;
```

is equivalent to

```
hidden constant pi := 3.14159;
```

In contrast to variable declarations, constants must be initialized. The value of a constant is computed from the expression on the right-hand side. A constant declaration may be optionally combined with explicit associations between handlers and exceptions (see 8.4).

4.4.3 Constants are Dynamic

The constants in ProSet are *dynamic* constants (not *manifest* constants whose value can be determined at compile time). For this reason, constants can be treated by the compiler similiar to variables. The constants are initialized during their declaration, but subsequent assignments are not allowed.

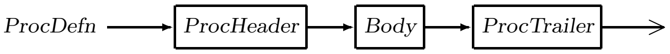
Note, that constants are *elaborated* dynamically. *Dynamic* means that the elaboration takes place when the control flow reaches the corresponding range.

The right-hand side of a constant declaration may be an arbitrary expression.

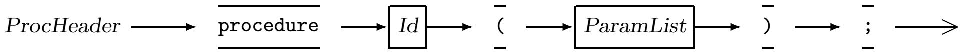
4.5 Procedures

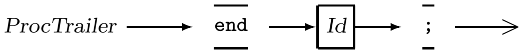
A procedure is a named program unit containing local data and statements which may be executed by calling the procedure.

Syntactically, the definition of a procedure is as follows:

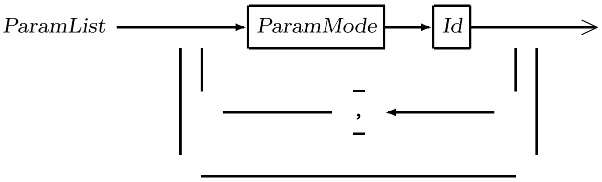


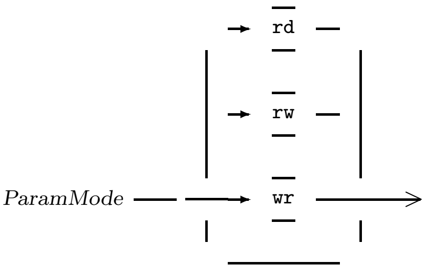
A procedure is enclosed by a header and a trailer. The name of the procedure in the header and in the trailer has to be identical.





The header contains a list of formal parameters:





```
procedure transfer(a, rd b, wr c, rw d);
begin
    c := a + b;
    d := d + c;
    a := 17;
end transfer;
```

Figure 5: Parameter transmission.

The list of the formal parameters is a sequence of names being enclosed in parentheses. Each name may be associated with a mode describing the parameter transmission:

`rd`	read-only parameters
`rw`	read-write parameters
`wr`	write-only parameters

A missing mode indicates `rd`. The example in figure 5 shows the different modes:

The parameters `a` and `b` are read-only parameters. A caller of `transfer` must provide values for `a` and `b`. The parameter transmission is *call by value*, i.e. on entry, the formal parameter will be initialized by a copy of the actual parameter. Within the procedure the parameter is treated just as a visible variable.

In our example, `c` is declared as write-only parameter. Here, the argument must be a valid left-hand side (section 6.10). The parameter transmission is *call by result.* Hence the formal parameter identifies a new visible variable with `om` as an initial value. On exit from the procedure, the value of the variable will be copied into the actual argument.

The last parameter, `d`, is declared as read-write parameter. In this case, the argument must be a valid right-hand side as well as a valid left-hand side (section 6). The parameter is transmitted *call by value/result.* On entry to the procedure the visible variable identified by the formal parameter is initialized with the argument value. When the procedure terminates, the *l*-value of the argument is determined and the value of the variable is assigned to it.

ProSet has *copy semantics.* All parameters are transfered by *copying*, not by *reference.*

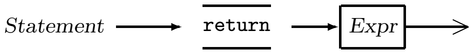
The number of the actual and the formal parameters must agree. We do not allow procedures with a variable number of parameters. Parameterless procedures are defined without a parameter list (but with parentheses). When calling a parameterless procedure the parentheses must be provided.

The following example calls the procedure in figure 5:

```
transfer(1, 2, t(3), z);
```

As stated above the third argument must be a *r*-value and a *l*-value, i.e. `t` must be a `tuple`, `map`, or `string`. The last argument `z` has to be a *l*-value.

The statement



may only appear within a procedure, in an anonymous procedure (section 4.5.1), or in an exception handler (section 8.5). It terminates the execution of a procedure or exception handler. In case of named and anonymous procedures it defines the value that the procedure will return.

Omitting the return value

```
return;
```

is an abbreviation for writing

```
return om;
```

If no `return` statement appears in the body of a procedure, the *default* value `om` will be returned. The procedure `transfer` demonstrates this behavior, i.e. it returns the default value `om` which is usually without significance. To avoid the introduction of unnecessary variables a procedure may be invoked in the form of a statement instead of writing:

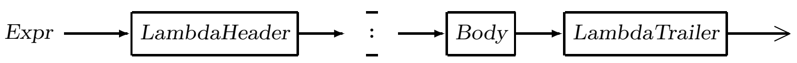
```
junk := transfer(1, 2, t(3), z);
```

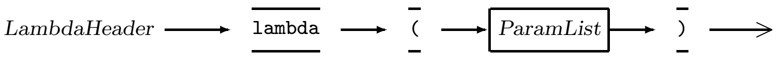
Return values will be discarded when procedures are used as statements.

ProSet procedures may be nested to any depth, as known from Pascal or Ada. This is unlike Setl or C. The name of a procedure and its formal parameters are treated as if they would be declared as `visible`, i.e. they are visible in all its enclosed ranges.

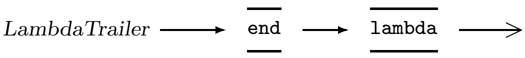
4.5.1 Anonymous Procedures

In addition to named procedures, ProSet provides *anonymous procedures*, which can be seen as a restricted form of λ-expressions in Lisp. The definition of an anonymous procedure looks like a normal procedure definition, except that the name is omitted and that the semicolon in the header is replaced by a colon:





The keywords `end` and `lambda` followed by a semicolon closes the definition of an anonymous procedure.



A definition of an anonymous procedures may be placed in any context, in which a valid expression is expected. The example expression in figure 6 computes for $n \geq 0$

$$(x, y, n) \mapsto F_{n-1}x + F_n y$$

where F_n is the n^{th} Fibonacci-number.

An anonymous procedure can reference itself by the keyword `self`, which is treated like the name of the procedure. The keyword `self` is only allowed within anonymous procedures and refers to the innermost definition of nested anonymous procedures. It is not visible outside an anonymous procedure.

Anonymous procedures may be defined and executed *on the fly*, as shown in the following example:

```
lambda (x, y, n):
begin
    return if n > 0
           then self (y, x + y, n - 1)
           else x
           end if;
end lambda;
```

Figure 6: Recursive call of an anonymous procedure.

```
procedure plus (a, b);
begin
    return a + b;
end plus;
```

Figure 7: A user-defined operator-procedure.

```
lambda(x):  begin return x + 1; end lambda(4);
```

yields the value 5.

4.5.2 User-Defined Operators

Procedures with one or two read-only arguments used in expressions are sometimes more convenient to use in *infix* and *prefix* notation, respectively. This can be done by placing the symbol ! in front of the procedure name. For example the procedure defined in figure 7 may be used as follows:

```
z := x !plus y !plus z;
```

which is apparently more convenient than

```
z := plus(plus(x, y), z);
```

Note that binary operators are left-associative. The precedences of user-defined operator-procedures is discussed in section 6.

Binary operators may be placed in front of the assignment symbol. This applies also to user-defined procedures used as binary operators. Assuming X is the name of a procedure suitable for a binary operator, then the following assignment

```
SomePeoplePreferSelfExplainingVariableNames :=
    SomePeoplePreferSelfExplainingVariableNames !X 1;
```

can be written as

```
SomePeoplePreferSelfExplainingVariableNames !X := 1;
```

5 Data Types

We describe in this section ProSet's data types.

A data type in a programming language may be seen as set of values (the domain of that type) and an associated set of operations defined on this domain (constructors, selectors, predicates, etc.). The values in ProSet can be classified into the following main categories, which are discussed together with the corresponding operations in the following subsections:

- the undefined value `om`. This value is used in ProSet to indicate particular situations like the use of non initialized variables, value extraction from an empty set, access to an undefined tuple component, etc.
- primitive data types
  `integer` (exactly represented whole numbers),
  `real` (floating point numbers),
  `boolean` (the Boolean values `true` and `false`),
  `string` (arbitrary length character strings),
  `atom` (dynamically generated unique values).
- compound data types which are heterogeneous collections of values
  `tuple` (arbitrary length sequences)
  `set` (finite mathematical sets).
- higher order data types
  `function` (first class procedures),
  `modtype` (first class module templates),
  `instance` (module instances).

Since ProSet is a weakly typed language, i.e. a variable name may be used without declaration and may be bound to values of different types during one program execution, it provides some operations which allow dynamic type checking and comparing objects of arbitrary types:

`type x` this unary operator returns a predefined constant of the type `atom` corresponding to the type of `x`. For detail see section 5.1.5.
Note: The type of `om` is undefined, so that the following equation holds:
`type om = om`.

`x = y` equality test: returns `true`, if the types and the values of `x` and `y` are equal.

`x /= y` the negation of the equality.

Apart from these operations, the predefined polymorphic functions and operators in ProSet are defined only for a restricted collection of types. Whenever an argument does not meet the allowed types or the operation is not defined for that particular value of a correct type, an exception will be raised. For an overview of all exceptions which may be raised by predefined operators and functions see Appendix B.

5.1 Primitive Data Types

5.1.1 `integer`

In ProSet exactly representable whole numbers are provided by the data type `integer`. The range of integer values may be restricted and depends on the implementation. Table 3 gives an overview of all integer operations and predefined functions. The unary (binary) operations are used in the usual prefix (infix) notation.

Some operations and functions are discussed in greater detail now:

unary	`+, -, random, type`
binary	`+, -, *, **, /, mod, max, min`
predicates	`=, /=, >, <, >=, <=`
functions provided by the standard library	`abs, even, odd, float, sign, str`

Table 3: Predefined operators and functions: `integer`

`i ** j` computes `i` to the `j`th power. An exception will be raised, if `(j<0)` or `(i=j=0)`.

`i / j` computes the integer part of the quotient of `i` by `j`. An exception will be raised, if `j=0`.

`i mod j` computes the remainder of the integer division. An exception will be raised, if `j=0`.
Thus for `j /= 0` and the settings (`q := i / j`) and (`r := i mod j`) following equation holds: `(i = q * j + r)` with `(0 <= r < abs(j))`.

`random i` returns a uniformly distributed random number in the range from zero up (down) to `i` including both end points.

5.1.2 `real`

ProSet provides floating point numbers through the data type `real`. As usual in programming languages the representation of real numbers is only an approximation to the exact value. It is guaranteed by the implementation, however, that the representation corresponds to C's data type `double` on the same machine.

An overview of the operations on the data type real is given in Table 4:

unary	`+, -, random, type`
binary	`+, -, *, /, **, max, min`
predicates	`=, /=, >, <, >=, <=`
binary functions provided by the standard library	`atan2`
unary functions provided by the standard library	`abs, fix, floor, ceil, exp, log, cos, sin, tan, acos, asin, atan, tanh, sqrt, sign, str`

Table 4: Predefined operators and functions: `real`

Some operations and functions are discussed in greater detail now:

`x ** i` computes the exponentiation of `x` by the integer `i`. An exception will be raised, if `x` and `i` are both zero.

`x / y` computes `x` divided by `y`. An exception will be raised, if `y` is zero.

`random x` returns a randomly selected floating point number in the range from zero up (down) to `x` including zero but excluding `x`.

5.1.3 `boolean`

The Boolean values `true` and `false` are provided by the data type `boolean`. Table 5 gives an overview of the operations on this type.

The operation `and` and `or` will be evaluated as *short circuit-* operations, i.e. they will only partially evaluated until the result is determined. Thus the following equalities hold:

```
a and b = if a then b    else false end if
a or  b = if a then true else b     end if
```

unary	`not, type`
binary	`and, or`
predicates	`=, /=`
constants	`true, false`
predefined functions provided by the standard library	`str`

Table 5: Predefined operators: `boolean`

5.1.4 `string`

The data type `string` represents an arbitrary length sequence of characters.

For strings the usual operations like length of the string (`#`), concatenation (`+`), lexicographic comparison, and extraction of a character or of a slice resp. are defined. Table 6 gives an overview of string operators and functions.

unary	`#, type`
binary	`+`
predicate	`=, /=, >, <, >=, <=`
constant	`""`, *the empty string*
extraction and slicing	`s(i), s(i..j), s(i..)`
unary functions provided by the standard library	`abs, str, char`
string scanning primitives	`span, any, break, len, match, notany, lpad, rspan, rany, rbreak, rlen, rmatch, rnotany, rpad`

Table 6: Predefined operators and functions: `string`

Some operations and functions are discussed in greater detail now:

`s(i)` the extraction `s(i)` takes an integer `i` and returns a one-character string equal to the `i`-th character of `s`. If `i<1` an exception will be raised. If `(i > #s)` the value `om` is returned.

`s(i..j)` the slice `s(i..j)` returns the substring of `s` which extends from its `i`-th to its `j`-th character, inclusive.
The exact definition reads as follows:

$$\texttt{s(i..j)=}\begin{cases} \textit{exception} & , \text{ if } \texttt{(i<1) or (j<i-1) or (j>\#s)} \\ \texttt{"" + s(i) + s(i+1) +...+ s(j)} & , \text{ otherwise} \end{cases}$$

`s(i..)` slice equal to `s(i..#s)`.

The extraction and slice operations may also be used in a left-hand side context:

`s(i) := x;` if `x` is a one-character string, the assignment is equivalent to

`s := s(1..i-1) + x + s(i+1..#s);`

otherwise an exception will be raised.

`s(i..j) := x;` if `x` is a string, the assignment is equivalent to

`s := s(1..i-1) + x + s(j+1..#s);`

otherwise an exception will be raised.

`s(i..)  := x;` if `x` is a string, the assignment is equivalent to

`s := s(1..i-1) + x;`

otherwise an exception will be raised.

For the definition of the string scanning primitives and the operations of the standard libraries see section 13.4.

5.1.5 `atom`

Values of type `atom` in ProSet are generated by a call to the nullary standard function `newat`. There is no other way of obtaining a new atom. The objects thus generated are unique in the sense that the implementation must guarantee that whenever `newat` is called, a new value different from all other calls to `newat` is returned. This is independent of whether they are called

- in one or different program executions,
- on one or different machines,
- at the same or different times.

Apart from the generation of new atoms the only operations on this data type are testing for equality respectively inequality (`=, /=`) and the type-testing operation (`type`).

nullary function	`newat`
unary	`type`
binary	`=, /=`

type constants	`atom, boolean, integer, real, string,`
	`tuple, set, function, modtype, instance`

mode constants	`rd, rw, wr`
predefined functions provided by the standard library	`str`

Table 7: Predefined operators and functions: `atom`

The predefined type constants enumerated in Table 7 are returned by an application of the `type`-operator such that type testings like `(type s = set)` are possible. The mode constants `(rd, rw, wr)` are used to describe the profile of functions and modules (see section 5.3.1 resp. section 11).

Note that atoms can be transformed into a string by the `str`-function. The result depends on the implementation and should only be used for testing and debugging.

5.2 Compound Data Types

In ProSet an arbitrary but finite collection of objects can be structured either in form of a linear `tuple` or an unordered `set`. Both data types are not necessary homogeneous and can be nested to arbitrary depth. With the exception of the undefined value `om`, values of each data type (including `function`, `modtype` and `instance`) may appear as a member of a tuple or set; `om` is prohibited in sets, but allowed as a component of a tuple.

5.2.1 `tuple`

Tuples have their mathematical semantics as ordered sequences of objects; a value may appear multiply in a tuple, the order of components appearing in the tuple is relevant. Conceptually a tuple is an infinite vector with almost all components equal to the value `om`.

The indexing of tuple components starts with the index `1`, the length returned by the `#`-operator is the largest index of a component different from `om`, thus `#[] = 0` holds.

For tuples the usual operations like concatenation (`+`), non-deterministic selection (`random`), insertion (`with`), slicing, testing of tuples for equality or membership (`in, notin`) are defined.

Table 8 describes all tuple operations, for the semantics of `frome` and `fromb` see chapter 7.1.

unary	`#, random, type`
binary	`+, with`
predicates	`=, /=, in, notin`
extraction and slicing	`t(i), t(i..j), t(i..)`
assignments	`frome, fromb`
tuple former expressions	*see below*
constants	`[]`, *the empty tuple*; `argv`, *program-parameter*
functions in the standard library	`str`

Table 8: Predefined operators : `tuple`

There are several kinds of tuple former expressions:[1]

enumeration: The tuple with the objects $\mathtt{x}_1$, ... ,$\mathtt{x}_n$ can be constructed by the tuple former expression [$\mathtt{x}_1$, ... ,$\mathtt{x}_n$]

simple interval: A tuple containing all numbers of the interval `i ... k; i,k` $\in \mathbb{Z}$[2] can be generated by the expression `[i..k]`

interval with step `m`: A tuple containing all numbers of the

$$\substack{\text{increasing}\\ \text{decreasing}} \text{ interval } \{\mathtt{i} + \mathtt{t} * \mathtt{m} : \ \mathtt{t} \geq 0,\ \mathtt{m} \overset{>}{<} 0,\ \mathtt{i} + \mathtt{t} * \mathtt{m} \overset{\leq}{\geq} \mathtt{k}\},\ \ \mathtt{i}, \mathtt{k}, \mathtt{m} \in \mathbb{Z}$$

can be constructed by `[i, i+m ..  k]`.
Note: For (i < k, m ≤ 0) or (i > k, m ≥ 0) this expression results in an endless loop.

descriptive: The tuple former

[e: $\mathtt{x}_1$ in $\mathtt{s}_1$, $\mathtt{x}_2$ in $\mathtt{s}_2$, ..., $\mathtt{x}_n$ in $\mathtt{s}_n$ | C]

generates the tuple of all values to which the expression `e` evaluates, when iteration with $\mathtt{x}_1$ over the compound object $\mathtt{s}_1$, with $\mathtt{x}_2$ over $\mathtt{s}_2$, ..., $\mathtt{x}_n$ over $\mathtt{s}_n$, such that the condition `C` holds. Syntactically the part

`| C`

can be dropped. In this case the condition `true` is assumed.
For a general discussion of iterators, allowed in tuple former expressions, see section 6.7.

The extraction `t(i)` takes an integer `i` and returns the `i`-th component of the tuple `t`. If `i<1` an exception will be raised. For `(i > #t)` the value `om` is returned.

The tuple slices `t(i..j)` and `t(i..)` are defined as :

$$\mathtt{t(i..j)} = \begin{cases} \textit{exception} & , \text{ if } \mathtt{(i<1)\ or\ (j<i-1)} \\ \mathtt{[\ t(x)\ :\ \ x\ in\ [i..j]\ ]} & , \text{ otherwise} \end{cases}$$

$$\mathtt{t(i..)} \ = \mathtt{t(i..\#t)}.$$

For the randomly selected tuple component returned by the operator `random`, the implementation has to guarantee uniform distribution. So the effect of the expression `random t` has to be equivalent to

$$\begin{cases} \mathtt{om} & , \text{ if } \mathtt{\#t\ =\ 0} \\ \mathtt{t(random(\#t} - 1) + 1) & , \text{ otherwise} \end{cases}$$

The extraction and slice operations may also be used in a left-hand side context:

`t(i) := x;` is equivalent to: `t := t(1..i-1) + [x] + t(i+1..#t);`

`t(i..j) := x;` if `x` is a tuple the assignment is equivalent to:

`t := t(1..i-1) + x + t(j+1..#t);`

otherwise an exception will be raised.

[1] Note: In the following descriptions the triple dot ... is a meta language construct which is used as an abbreviation for *and so on*, while the two dots `..` are one token of ProSet's syntax

[2] $\mathbb{Z}$ denotes the set of all integers

`t(i..)  := x;` if `x` is a tuple the assignment is equivalent to:
`t := t(1..i-1) + x;`
otherwise an exception will be raised.

5.2.2 `set`

Sets have their mathematical meaning as unordered collections of objects, for which a value is either a member of the set or not. An element cannot appear more than once in a set, the order of appearing is not relevant. The undefined value `om` must not be inserted into a set, otherwise an exception will be raised.

For sets the usual operations like union (`+`), intersection (`*`), set difference (`-`), cardinality (`#`), insertion (`with`), deletion (`less`), test for membership (`in`, `notin`), test for inclusion (`subset`), and the power set operation (`pow`) are defined.

If a set is not empty, the operations `arb` and `random` return a non-deterministically selected element — otherwise `om` is returned. It depends on the implementation whether each call to the selector `arb` returns the same, a different, or a randomly selected element of a set. In contrast, the implementation has to guarantee uniform distribution for elements randomly selected by the operator `random`.

The predicates `is_map` and `is_smap` test whether the set has the map characteristics (see section 5.2.3).

Table 9 gives an overview of all set operations, for a description of `from` see section 7.1.

unary	`#, pow, arb, random, type`
binary	`+, -, *, mod, with, less`
predicates	`=, /=, in, notin, subset, is_map, is_smap`
assignment	`from`
set former expressions	*see below*
constant	{ }, *the empty set*
functions in the standard library	`npow, str`

Table 9: Predefined operators and functions : `set`

There are several kinds of set former expressions:

enumeration: The set with the objects $\mathtt{x}_1, \ldots, \mathtt{x}_n$ can be constructed by the set former expression $\{\mathtt{x}_1, \ldots, \mathtt{x}_n\}$

simple interval: A set containing all numbers of the interval `i ... k`; `i,k` $\in \mathbb{Z}$ can be generated by the expression $\{\mathtt{i..k}\}$

interval with step `m`: A set containing all numbers of the

$\substack{\text{increasing}\\ \text{decreasing}}$ interval $\{\mathtt{i}+\mathtt{t}*\mathtt{m}:\ \mathtt{t}\geq 0,\ \mathtt{m}\overset{>}{<}0,\ \mathtt{i}+\mathtt{t}*\mathtt{m}\overset{\leq}{\geq}\mathtt{k}\},\ \mathtt{i},\mathtt{k},\mathtt{m}\in\mathbb{Z}$

can be constructed by $\{\mathtt{i,\ i+m\ ..\ k}\}$.
Note: For (i $<$ k, m $\leq$ 0) or (i $>$ k, m $\geq$ 0) this expression results in an endless loop.

descriptive: The set former

$$\{\mathtt{e}:\ \mathtt{x}_1\ \mathtt{in}\ \mathtt{s}_1,\ \mathtt{x}_2\ \mathtt{in}\ \mathtt{s}_2,\ \ldots,\ \mathtt{x}_n\ \mathtt{in}\ \mathtt{s}_n\ \mid\ \mathtt{C}\ \}$$

generates the set of all values to which the expression `e` evaluates, when iteration with $\mathtt{x}_1$ over the compound object $\mathtt{s}_1$, with $\mathtt{x}_2$ over $\mathtt{s}_2$, …, $\mathtt{x}_n$ over $\mathtt{s}_n$, such that the condition `C` holds. Syntactically the part

`| C`

can be dropped. In this case the condition `true` is assumed.
For a general discussion of iterators, allowed in set former expressions, see section 6.7.

5.2.3 Maps

Although maps do not correspond to a separately defined data type in ProSet, there are some operations which support the handling of finite maps and binary relations.

Conceptually a map is defined as a set of pairs, i.e. a set of tuples

(*) $[x, y]$ with $x \neq om$, $y \neq om$.

The domain of a map f is defined by $\{e(1) : e \in f\}$, and its range by $\{e(2) : e \in f\}$.

Since a map is a special kind of set all set operations are valid for maps. The predicate `is_map f` tests whether `f` is a binary relation, i.e. whether the condition (*) holds for all members of `f`. `is_smap f` tests additionally whether `f` is a single valued map, i.e. whether for each x in the domain of `f` there exists *exactly* one y in the range. Note: Both `is_map` and `is_smap` are set operations — an exception will be raised if the argument's type is different from `set`.

For maps the operations enumerated in Table 10 are provided in addition to the operations for sets.

unary	`domain, range`
binary	`lessf`
extraction	`f(i), f{i}, f(`$\mathtt{i}_1$`, ... ,`$\mathtt{i}_n$`), f{`$\mathtt{i}_1$`, ... ,`$\mathtt{i}_n$`}`

Table 10: Additionally defined operators and functions: `Maps`

The unary operator `domain f` returns the domain of the map `f`, the operator `range f` the range of the map, and the binary operator `f lessf x` returns a copy of `f` in which all pairs `[x,y]` with `[x,y] in f` are deleted.

The definition of the extraction operators:

`f{x}` returns the image-set of `f` at the point `x`, i.e. the set of all second components of pairs in `f` whose first component is `x`:

```
f{x} = { y(2) :  y in f | y(1) = x }
```

`f(x)` returns the single image of `f` at the point `x`, i.e. the only element in `f{x}` provided that `(#f{x} = 1)` holds. If `(#f{x} = 1)` fails, `om` is returned.

Similar to strings and tuples the extraction operation may be used in a left-hand side context:

`f(x) := y;` is equivalent to the sequence:

```
f := f lessf x;
if y /= om then f := f with [x, y]; end if;
```

`f{x} := y;` provided that `y` is a set, this assignment is equivalent to the sequence:

```
f := f lessf x;
f := f + { [x, z] :  z in y};
```

otherwise an exception will be raised.

The multiparameter extractions are just abbreviations. Thus the following identities hold:

$$\mathtt{f(i_1, \ldots ,i_n) = f([i_1, \ldots ,i_n])}$$
$$\mathtt{f\{i_1, \ldots ,i_n\} = f\{[i_1, \ldots ,i_n]\}}.$$

5.3 Higher Order Data Types

Objects of the data type `function`, `modtype` or `instance` in ProSet have the rights of first-class citizens. As all other data types introduced so far they have their own identity, and it is allowed

- to assign them to variables,
- insert them into compound objects,

- use them as actual parameters in the call of subroutines,
- return them by the call of a function,
- use them in expressions.

5.3.1 `function`

ProSet provides the possibility to generate a first class data object of the type `function` from a named or anonymous procedure. This is done by applying the `closure` operator to a procedure name or a lambda expression. Thus the statement

```
f := closure p;
```

assigns `f` a function corresponding to the procedure `p`. Similar to the application of the generator `newat` for the data type `atom`, each application of the operator `closure` will result in a new and unique object. Thus assuming in the following code that `g` is a visible procedure name

```
f1 := closure g;
f2 := closure g;
f3 := f1;
```

`f1` is equal to `f3`, but `f1` is not equal to `f2`, because they result from different applications of the `closure` operator. Thus equality resolves to identity.

For a procedure or a lambda expression the static environment in which the procedure is defined resolves the question to which value a name is bound when it is used in the procedure's body. Thus it is possible to allow the procedure to have side effects on non-local objects, i.e. objects declared in an outer range. When a `function` is created through an application to `closure`, the question of binding arises again, because this function may be called outside the defining environment of the corresponding procedure. ProSet solves this question by freezing the bindings to non local objects: At the time when the closure operator is applied to a procedure name or lambda expression, the binding of a name n used inside the procedure to a non local object with the actual value v is stored. This binding associates n with v. At the beginning of each invocation of the resulting function f this binding is restored in such a way that n behaves like an initialized local variable. Thus no side effect of a `function` application to its dynamic environment is possible.

The example in Figure 8 displays this effect. Since `x` is bound to the value `0` at the generation time of function `f`, each application of `f` writes `0` to the standard output. Thus the program produces the output:

```
0
1
```

Note that replacing the lambda expression by the procedure name `p` will not change the output of the example. Note also that the expression

```
(closure p)()
```

causes the function corresponding to the procedure `p` to be executed *on the fly*. The parentheses around `closure p` are necessary because of the operator precedences (see Table 12). Without those parentheses the compiler will produce an error.

Table 11 gives an overview of all operations on the data type `function`.

The expression `profile f` returns a tuple, whose components are chosen from the set {`rd, wr, rw`} and which corresponds to the declaration modes of the formal parameter list of that procedure, `f`

```
  program prog;
      visible x := 0;
  begin
      f := closure lambda():
           begin
               put(x);
           end lambda;
      x := 1;
      f();
      p();

      procedure p();
      begin
          put(x);
      end p;
end prog;
```

Figure 8: Binding in functions versus binding in procedures.

unary	`profile, type`
predicates	`=, /=`

Table 11: Predefined operators: `function`

was derived from. Thus the expression `(# profile f)` returns the number of parameters `f` has to be applied to and `(profile f)(1) = rd` tests whether the first parameter was declared to be a read-only parameter.

The example in Fig. 9 illustrates the use of a function object returned by a procedure call.

5.3.2 `modtype` and `instance`

Modules and instances form data types of their own. Since they are rather complex, and since they are intended to support programming in the large, they are discussed separately (see section 11).

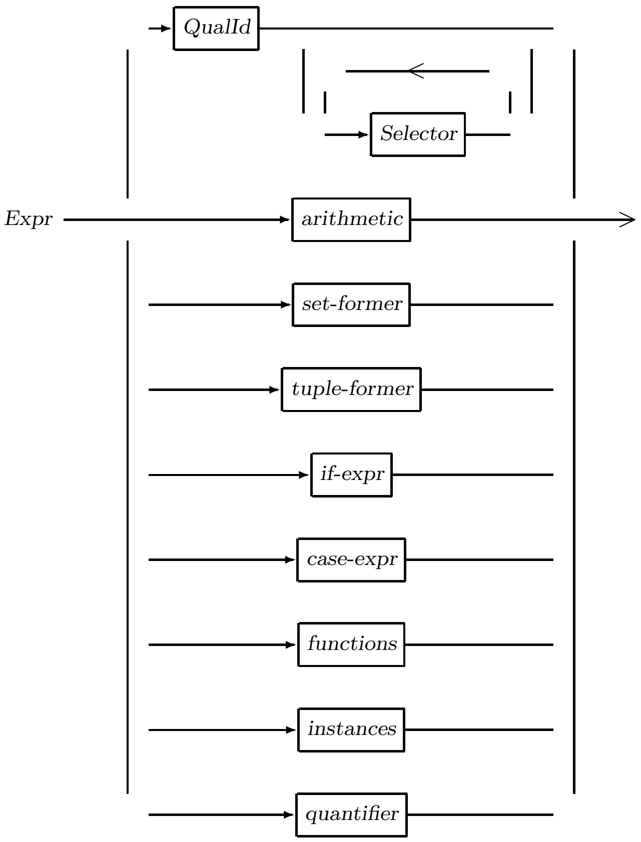
```
program main;
    visible t := 0;
begin
    x := f(t);
    x();                        -- output> g: t = 2
    x();                        -- output> g: t = 2
    putf("main: t = %d\n", t);  -- output> main: t = 1

    procedure f(rw t);
    begin
        t := t + 1;
        return closure g;

        procedure g();
        begin
            t := t + 1;
            putf("g: t = %d\n", t);
        end g;
    end f;
end main;
```

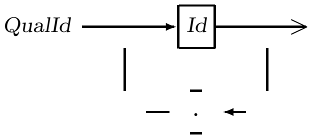
Figure 9: Function returned by a procedure call

6 Expressions

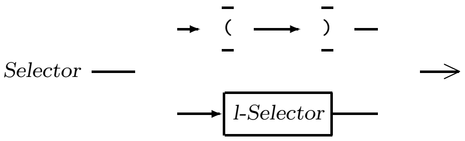
Expression is a syntactical term and it is used for describing the construction of values. This has to be seen in contrast to control structures, for example statements or loops, which may use or contain expressions. E.g. `if`-statements use expressions of type Boolean to determine the flow of control.

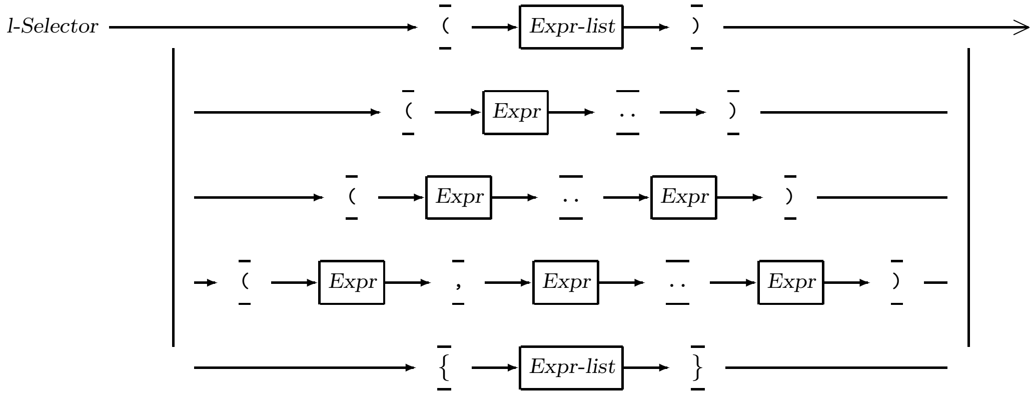
This section gives a reference for expressions. The following syntax chart may serve as a guide:





Most of the ‘nonterminals’ mentioned in the above syntax chart serve as hints for further discussion and are in fact dead ends. Only the upper alternative reflects a syntactic rule.





The first alternative in the rule for *Expr* above declares identifiers standing for variables, constants, procedures and modules as expressions. They may be qualified, i.e. denote components of modules. In this case, it is constructed by a module identifier followed by a dot and another identifier. Applying a selector to these basic objects may yield another expression: the result of a procedure call, the image under a map, an element or a part of a tuple or string, depending on the type of the basic object. Section 4.5 discusses procedure calls, strings are discussed in section 5.1.4, and for tuples and maps the details are given in subsections of 5.2.

The first subsection below reflects the operations on primitive and compound data types as discussed in section 5. This is what the somewhat informal term 'arithmetic' is supposed to describe — addition of numerical values, union of sets, length of a tuple — binary and unary operations on first class objects.

Although tuple- and set-formers have already been discussed along with these data types, the next subsection gives a comprehensive description on a syntactical level and refers to the descriptions provided in section 5.2.1 and section 5.2.2, respectively.

The section continues in discussing conditional expressions and nondeterministic choice, i.e. `if`- and `case`-expressions. They are very similar to control statements of the same name discussed in section 7. The subsequent three subsections deal with compound operations, type tests and higher order objects. The latter denotes first class functions and related objects.

Prior to discussing quantifiers we introduce iterators, which are not expressions by themselves, but their concept is needed on several occasions in ProSet, and the introduction of quantifiers is one of them.

All the language constructs mentioned above (except from iterators) serve to construct values which may be assigned to variables. They are called *r-values* (for right-hand side values) because they may stand on the right-hand side of an assignment. A summary of expressions used as *r*-values is given in the penultimate subsection — the last subsection deals with *l-values.* They are the counterpart of *r*-values and may stand on the left-hand side of an assignment. In ProSet, *l*-values are not restricted to variable names, they can be much more complex so that they are discussed separately.

Syntax diagrams are used to describe those kinds of *r*-values which are discussed in this section. For *l*-values, syntax charts are not as appropriate as they should be since they are not specific in characterizing differences with respect to *r*-values. However, the rules are few and short and listing them will do no harm.

6.1 Arithmetic Operations

The term 'arithmetic' is generally used to describe the four basic operation on numbers — in this section we will use this term in a wider sense. Any operation, unary or binary, on any admissible type of value is regarded as an arithmetic operation. This does neither include the explicit construction of compound values nor other operations including an explicit iteration nor assignments.

Precedence	Operator
8	selectors and handler associations
7	all unary operators (including quantifiers)
6	`**`
5	`*`, `mod`, `div`
4	`+`, `-`, `max`, `min`
3	all binary operators not listed above or below (including user defined binary operators)
2	`=`, `/=`, `<=`, `>=`, `in`, `notin`, `subset`
1	`and`
0	`or`

Table 12: Operators and precedences

Table 12 shows operators and their precedences. Binary operators are left associative without exception, i.e. $a \oplus b \oplus c$ is to be read as $(a \oplus b) \oplus c$ for any binary operator $\oplus$. Making use of precedences may improve readability in obvious cases or with proper grouping and indentation — on the other hand, inserting a (redundant) pair of parentheses may also help reading.

Evaluation Order in Expressions and Expression Lists

The order of evaluation of subexpressions in binary expressions is *not* defined. This not in conflict with associativity — the essence is that the order in which side effects may occur is not predictable by the programmer. This is also true for expression lists which occur e.g. in function calls or constructors for compound values.

Evaluation order is defined, however, in declaration lists (for variables or constants) as stated in section 4.4.

6.2 Set- and Tuple-Formers

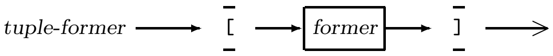
Tuples or sets may be constructed iteratively by adding elements to a variable initialized as the empty tuple or set.

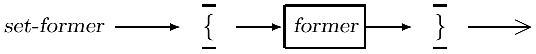
On the other hand, both kinds of compound objects can be constructed by a single language construct without making use of a variable being used in intermediate steps.

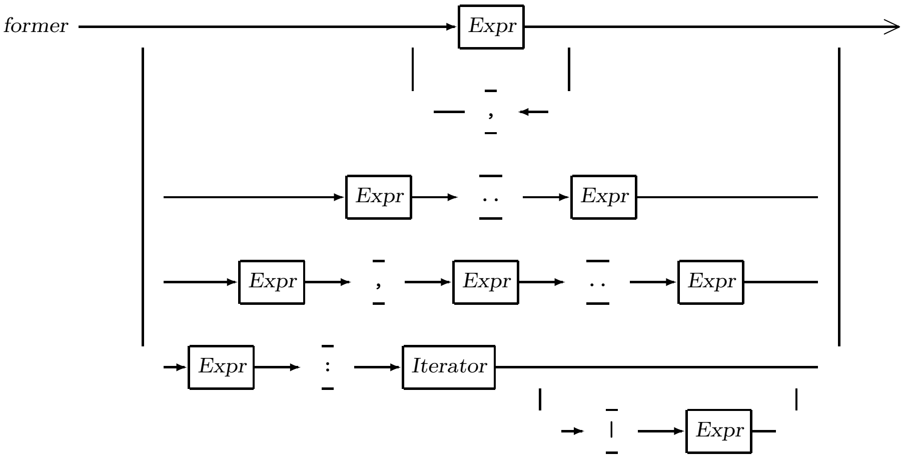
There are four distinct way of doing this:

- enumeration
- simple interval (of integers)
- interval with step
- descriptive

The following charts describe the syntax:







The brackets indicate construction of a tuple whereas curly brackets are used to build sets.

The *former* reflects the list from above:

- *Enumeration* allows arbitrary values to be inserted.
- An *interval* restricts the expressions being involved to evaluate to type integer, otherwise an exception will be raised.
- The last alternative makes use of an *iterator*. It will be described later in this section, at this point a small example may serve as an explanation:

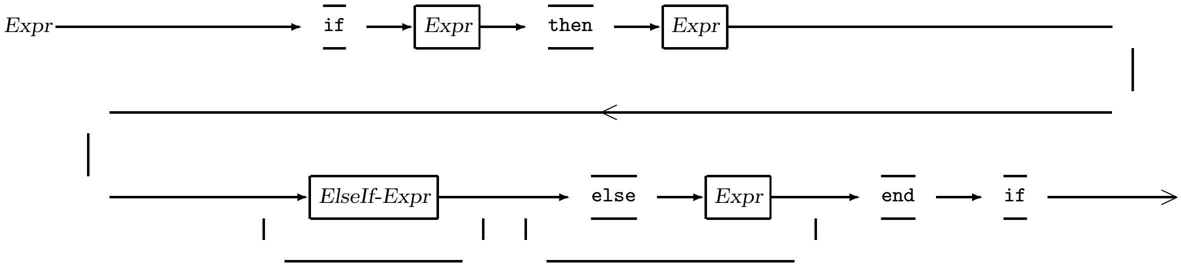
  ```
  {[y,x]:  [x,y] in f | x /= y}
  ```

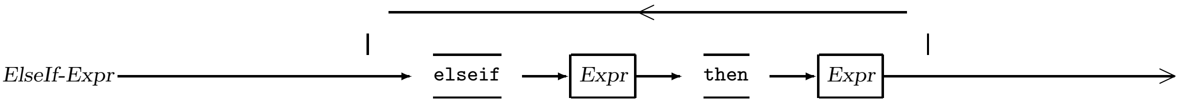
  The set constructed here is the inverted map of `f` where all the pairs `[x,x]` of `f` are not included.

The discussion in sections 5.2 will give further information on the construction of sets and tuples, and section 6.7 will discuss iterators in greater detail.

6.3 `if` Expressions

`If` expressions (conditional expressions) are syntactically close to `if` statements (cf. section 7.5), with the difference that single expressions replace the statement lists. Since only statements are terminated with semicolons, they should not be used inside conditional expressions.





The following example describes using a map for counting:

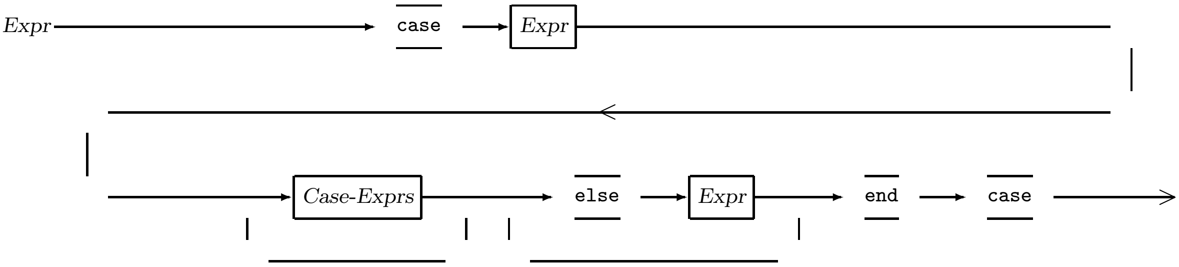
```
count(x) := if count(x) = om then 0 else count(x) end if + 1;
```

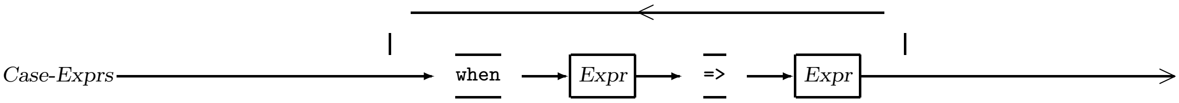
The `if`-expression defines the count of any value `x` not yet seen to be zero.

If the `else` part of the conditional expression is missing, `else om` will implicitly be assumed.

6.4 `case` Expressions

`Case` expressions are derived from `case` statements (cf. section 7.6) in a similar way as `if` expressions are from `if` statements.





The assumption that a missing `else` part means `else om` is valid here, too.

The following example classifies a day, i.e. depending on the value of the variable `day` (assumed to be of type `string` because of the string concatenation used in each case) one branch is chosen — the result is a string:

```
Answer := case day
    when "Monday", "Tuesday", "Wednesday", "Thursday", "Friday"
       => day + " is a work day"
    when "Saturday", "Sunday"
       => day + " is weekend"
    else day + " is not recognized as a day"
  end case;
```

6.5 Compound Operations

Compound operations allow application of binary operators to compound values. The following description is valid for tuples `t = [t1,t2,...,tn]` and may serve as an example:

$\oplus$ `% t = t1` $\oplus$ `t2` $\oplus \cdots \oplus$ `tn`   where $\oplus$ is any binary operator.

For $\oplus$ being the addition operator + and a tuple `t = [2,3,4,5]` this would be `+ % t`, i.e. `2 + 3 + 4 + 5` which results in `14`.

If `#t = 1`, $\oplus$`% t` returns `t(1)`.
If `t = []`, $\oplus$`% t` raises an exception.

For sets s, the following equality can be used as a definition:

$$\oplus \ \texttt{\% s} \quad \equiv \quad \oplus \ \texttt{\% [y: y in s]}$$

Obviously, since the order of iteration over sets is nondeterministic, compound operations on sets should only be used with commutative operators.

Compound operators $\oplus$ `%` can be used both as binary (`x` $\oplus$ `% s`) and as unary operators (`op % s`). The former allows to define the behavior for empty compound objects `s`. The latter will result in an exception, if `s` is empty. For a tuple `s`, the expression `x` $\oplus$ `% s` is equivalent to $\oplus$ `% ([x]+s)`. For a set `s`, a similar rule applies:

$$\texttt{x} \oplus \texttt{\% s} \quad \equiv \quad \oplus \ \texttt{\% ([x]+[y: y in s])}$$

When used as binary operators, $\oplus$ `%` will have the same precedence as $\oplus$. User defined binary operators `!id` (see section 4.5.2) can be used for compound operations, too.

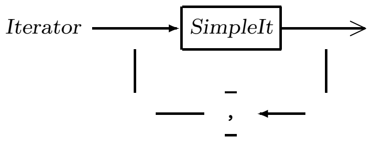
6.6 Higher Order Objects

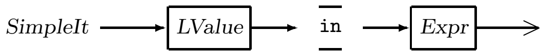
Procedures, `lambda`-expressions and names introduced by a module definition (i.e. the identifier `m` in `module m(...) ...`) are not first class objects — the `closure`-operator must be applied to them to cut off side effects and make them portable (see section 5.3.1). The result of applying the closure operator is a functional object or a value of type `modtype` that may freely be returned as a value or be inserted in any compound object or invoked later on (if it is a function) or instantiated (if it is a module). Instances of modules created by means of `instantiate` (cf. section 11.5) are also regarded as higher order objects.

6.7 Iterators

Iterators are used at several occasions in ProSet. They occur in some kinds of set or tuple constructors, in many loop constructs and in quantified expressions which are described below. Iterators do not form expressions on their own. In fact, they introduce *bindings*: new (local) variable names are bound to values. These bindings are used in the context of expressions and we think that it is a good place to describe them here.

Iterators are defined syntactically by the following diagram:





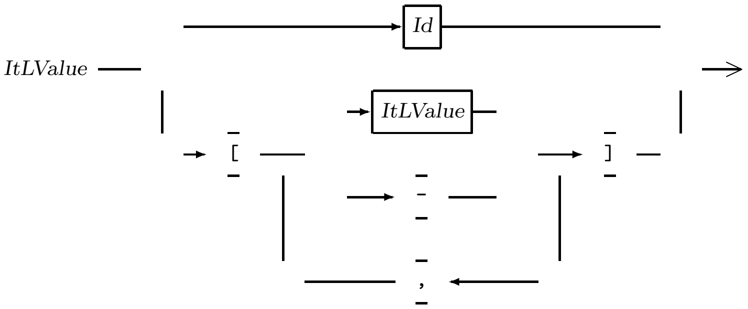
The iterator variables are subsequently bound to values of a set or tuple to be iterated over. In the simplest case

```
x in s
```

there is only one simple iterator containing an identifier `x` as the iterator variable. The expression `s` after the keyword `in` is the compound value to be iterated over. In a loop, `x` is bound to an element of this set or tuple `s` subsequently. With this binding, the enclosing language construct performs some action, and after that, `x` is bound to the next element of `s`.
The iteration ends, after all elements of `s` have been looked at, or if either

- the enclosing construct is a loop and executes a `quit`-statement to terminate the excecution of the loop
- the enclosing construct is a quantifier and the result is determined (see section 6.8).

The l-value in a simple iterator may be a compound l-value instead of a simple variable. In this case, a decomposition is performed as described in section 6.10, but this makes no difference with respect to the iteration process. However, the local environment may use the variables contained in the compound l-value. It should be mentioned, too, that the l-values used here are restricted to identifiers or to a list of l-values of this kind, enclosed in brackets. This restriction is required since iterators introduce new local names. Hence an iterator `x(1) in s` for a new variable `x` is illegal. A multiple l-value will also allow for insertion of dashs as dummy symbol. The following syntax chart describes these l-values:



Another extension is the repetition of simple iterators.

x_1 `in` s_1`,` x_2 `in` s_2`, ... ,` x_n `in` s_n

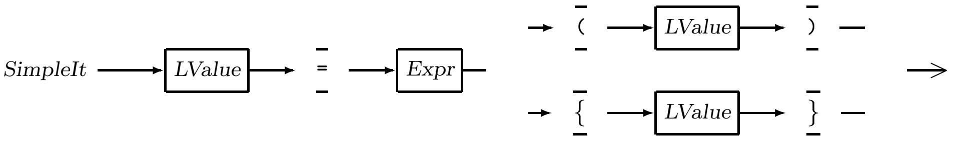
Here, a nested iteration is performed: x_1 is bound to an object of s_1, then x_2 is bound to an object of s_2, and so on, until x_n is bound to an object of s_n. Now, an action is to be performed in a similar way as described with a single iterator.

But there a more differences when compared to a single iterator: Since the order in which the bindings are performed is clearly defined, the expression determining the compound value s_2 may depend on the current value of x_1 and so on, i.e. $s_i, (1 < i \leq n)$ may depend on $x_j, (1 \leq j < i)$. The innermost iteration is performed most frequently, i.e. for each value x_{n-1} of s_{n-1}, x_n is bound to all values of s_n. After the iteration over s_i is exhausted, x_{i-1} is bound to the next value of s_{i-1}, provided $i > 1$. Otherwise, the iteration ends.

In the case of a `for`-loop using a multiple iterator, a `continue`-statement will continue the iteration with x_1 bound the next value of s_1, that is the *outermost* simple iterator. This is for the programmer's convenience: the behavior described is often required and could otherwise only be achieved by formulation of two explicitly nested loops. On the other hand, continuing with the next value x_n of s_n can be achieved by skipping the rest of the loop's body by a conditional statement.

Map Iterators

Map iterators are an abbreviation which may be used to iterate over a single- or multi-valued map:



The upper form with the parentheses is equivalent to an iteration over the set resulting from *Expr* (i.e. a single valued map) and decomposition of the elements which are assumed to be tuples: `y=f(x)` is equivalent to `x in domain f` and assigning `y := f(x)` for another new variable `y`. This form of iterator is also allowed for tuples `f`. Then, `x` is iterated over the tuple of indices `[1..#f]` and `y` is bound to `f(x)`.

For a multi-valued map, the behavior is similar: as above, `x` iterates over the domain of `f` and `y` is bound to `f{x}`.

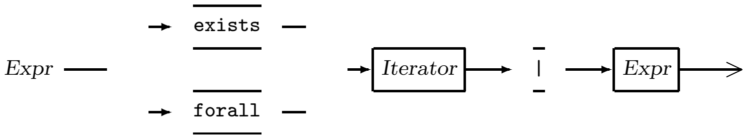
Since these rules are purely syntactical, everything stated above for simple iterators is valid for map iterators, too. Certainly, both kinds of simple iterators may be blended in a multiple iteration as may be deduced from the syntax diagrams.

Order of Iteration

The two types of compound values, namely sets and tuples, show different behavior when being iterated over: since tuples are ordered, the iteration over a tuple preserves this ordering. Embedded `om`-values in tuples will be regarded in iterations, i.e. an iteration over a tuple `t` may be regarded as the iteration over a tuple `[1..#t]` of indices and then selecting the corresponding tuple element of `t` via the index. Sets are unordered and so the elements of a set are used in an order which cannot be predicted by the programmer.

6.8 Quantified Expressions

Quantified expressions allow to formulate predicates on a high level. Two forms are provided in ProSet, which are introduced by the keywords `exists` and `forall`:



The existential quantifier yields `true`, if during the iteration the expression after the bar `|` (the predicate) evaluates to `true` at least once; it yields `false` otherwise.

`Forall` yields `true`, if each iteration step makes the predicate evaluate to `true`, and `false` otherwise.

The first set of bindings produced by the iteration which makes the predicate evaluate to `true` or `false` for `exists` or `forall`, respectively, will terminate the iteration.

Quantifiers share the highest precedence with unary operators. The example

```
exists x in s | p(x) and g
```

should be read as

```
(exists x in s | p(x)) and g
```

Hence it is not to be read as a quantification with predicate `p(x) and g`.

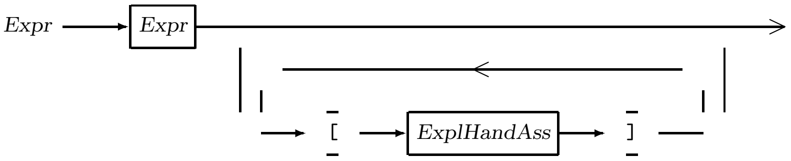
6.9 Summary of Expressions

The following list gives a comprehensive overview over expressions in ProSet returning values. Expressions are

- literals (of type `integer`, `real`, `string`), predefined values as for example `argv`, `atom`, `om`, `true` (see section 5.1)
- unary and binary operations (cf. table 12 in section 6.1) including compound operations. Binary operators are left-associative without any exception
- calls to functions or procedures
- tuple and set formers (see section 5.2 or section 6.2)
- `if`- and `case`-expressions
- functions (see section 5.3.1)
- modules and instances. `instantiate` may be used to create an instance of a module, i.e. a new set of variables local to the module is created and the initialization code is executed. For further discussion, see section 5.3.2.
- quantified expressions `exists`, `forall`

Exceptions

All expressions may be decorated with associations of exceptions to handlers. The following chart describes the syntax for this association.



The concept of exceptions and the association of exceptions with handlers is discussed in depth in section 8.

6.10 L-Values

In the introduction to this section we mentioned expressions not returning any values and defined them to be *l-values* (for *left-hand side values*). This is not the full story about l-values since some expressions returning values may also serve as l-values, for example as `rw`-parameters to procedures and in abbreviated assignments (see 7.1.1).

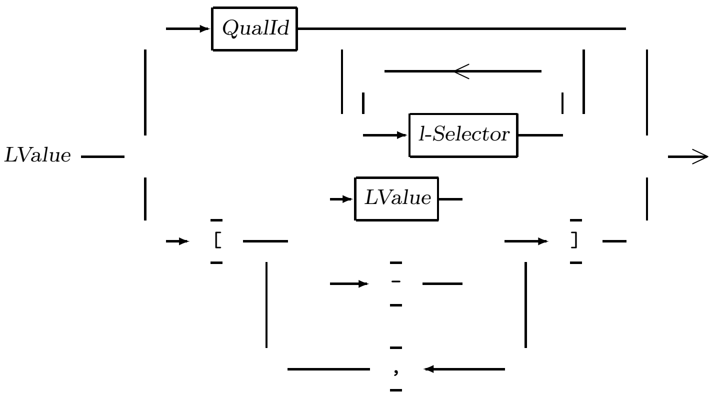
The list below definitely says what l-values are:

- variables are l-values
- for an l-value l, the following expressions are valid l-values, too:
  - if `type` $l =$ `tuple`, then $l(expr_1)$, $l(expr_1\,..)$, and $l(expr_1\,..\,expr_2)$ are valid l-values, if $expr_1 \geq 1$ and $expr_2 \geq expr_1 - 1$ (the expressions should evaluate to an integer).
  - if `type` $l =$ `string`, the rules for tuples apply, but there are further restrictions in that $0 \leq expr_2 \leq \#l$; $l(expr) \equiv l(expr..expr)$.
  - if l is a single valued map, then $l(expr_1, \ldots, expr_n)$ is a valid l-value.
  - if l is a multi valued map, then $l\{expr_1, \ldots, expr_n\}$ is a valid l-value.
- a tuple $[l_1, \ldots, l_n]$ is an l-value (also called *multiple l-value*), if $l_1, \ldots, l_n$ are either l-values or the dummy symbol denoted by the dash `-` .

There is no other way to construct l-values.

Whenever the dummy symbol `-` is contained in a multiple l-value, the resulting expression does not have an r-value. In all other cases, an expression being a valid l-value is also a valid r-value.
Multiple l-values and the dash are discussed in section 7.1 in the context of assignments. They are used to decompose tuple-valued r-values.

Here is a syntax chart describing l-values (it has to be taken with care, see p. 30):



The nonterminal *l-Selector* used above is defined in the beginning of this section. It permits selection of components of compound objects (and strings) but forbids procedure- or function-invocations with an empty parameter list.

Whenever l-values are constructed by means of selectors, the assignments performed on them may be decomposed from left to right and this decomposition defines the semantics:

```
t(1){2}(3) := 17;
```

may be described by

```
    tmp1 := t(1);
    tmp2 := tmp1{2};
    tmp2(3) := 17;
    tmp1{2} := tmp2;
    t(1) := tmp1;
```

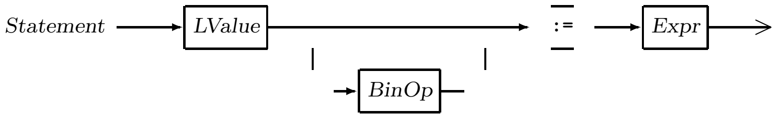
where `tmp1` and `tmp2` are fresh variables.

7 Statements

Statements specify the flow of control and data in programs. Program blocks consist of non-empty statement lists, which are explained in section 4.1. The statements in a list are executed sequentially. Some small control structures are only allowed within certain kinds of program blocks. As opposed to expressions, statements provide no values.

7.1 Assignments

In an assignment an *l*-value is assigned the value of an expression that appears on the right-hand side of the assignment symbol:



For a discussion on expressions which are allowed as *l*-values see section 6.10. The right-hand-side expression has to provide a tuple for multiple *l*-values. The assignment

```
[x1, x2, -, x4] := [1, 2, 3];
```

is equivalent to the series of assignments with `Temp_Tuple` as a fresh name:

```
Temp_Tuple := [1, 2, 3];
x1 := Temp_Tuple(1);
x2 := Temp_Tuple(2);
x4 := Temp_Tuple(4); -- yields om
```

The assignments are done in the specified order only for specified *l*-values. For the dummy symbol `-` (dash) no assignment is performed (cf. section 6.10).

7.1.1 Abbreviated Assignments

Binary operators may be combined with the assignment symbol to abbreviate assignments. Hence, the assignment

```
x := x * (1 + y);
```

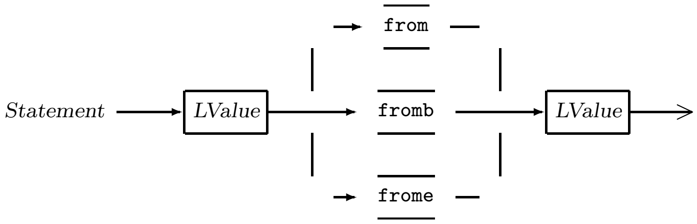
may be abbreviated by

```
x *:= 1 + y;
```

This is allowed for user-defined operators, too (cp. section 4.5.2).

7.1.2 The from Statements

Additionally ProSet provides three statement forms related to assignments:



On the left-hand side any valid *l*-value is allowed, whereas on the right-hand side multiple *l*-values are not allowed. We discuss these statements in greater detail now:

The from Assignment The simple *l*-value should be a set for the `from` assignment. The assignment

```
x from s;
```

is equivalent to the series of assignments:

```
x := arb s;  -- select any element of s
s less:= x;  -- remove it from s
```

See section 5.2 for the selection operator `arb`. Multiple *l*-values on the left-hand side are handled as before. The assignment

```
[x1, -, x3] from s;
```

is equivalent to the series of assignments:

```
Temp_Tuple := arb s;
s less:= Temp_Tuple;
x1 := Temp_Tuple(1);
x3 := Temp_Tuple(3);
```

The fromb and frome assignments The simple *l*-value should be a tuple for the `fromb` and `frome` assignments. The assignment

```
x fromb t;
```

is equivalent to the series of assignments:

```
x := t(1);
t := t(2..);
```

and the assignment

```
x frome t;
```

is equivalent to the series of assignments:

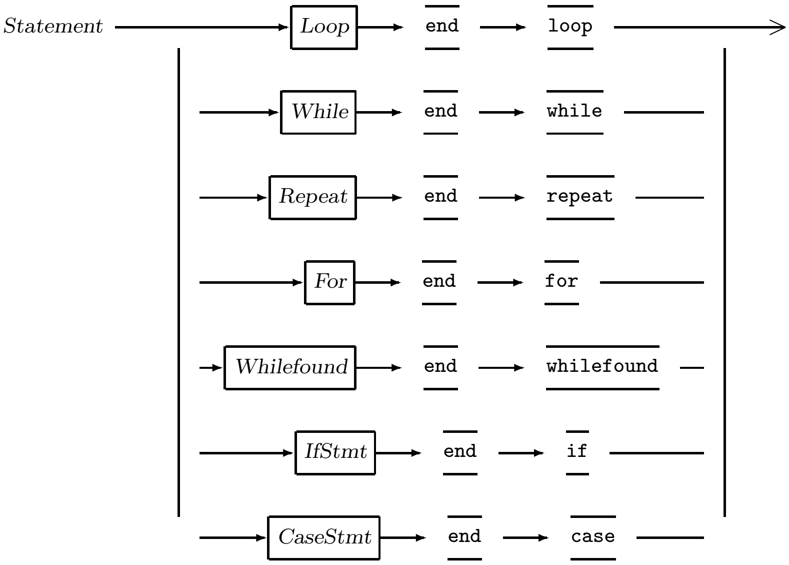
```
x := t(#t);
t := t(1..(#t-1));
```

Remember that no assignment and in general no statement provides a value in ProSet.

7.2 Control Structures in General

Control structures are static structures that control the flow of control in a program at runtime. The remaining subsections of this section will discuss the static control structures in ProSet. Quantified expressions and some set resp. tuple forming expressions contain implicit loops. They are discussed in section 6.

The syntax of the complex control structures shows that the language has Algol as one of its ancestors:

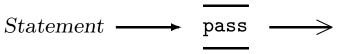


7.3 Small Control Structures

This subsection treats some control structures that do not enclose statements. There exist some additional *small* control structures, which are discussed elsewhere: `return` statements and procedure calls in section 4.5, some exception handling statements in section 8, `quit` and `continue` in section 7.4.6.

7.3.1 `pass`

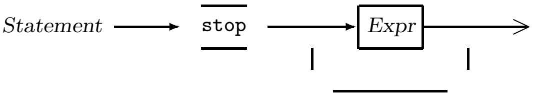
The `pass` statement passes the control to the next statement. It has no additional effect.



This statement may be used to program empty `then` cases for `if` statements (section 7.5) or empty program blocks.

7.3.2 stop

The `stop` statement terminates the execution of a process or of an application program:



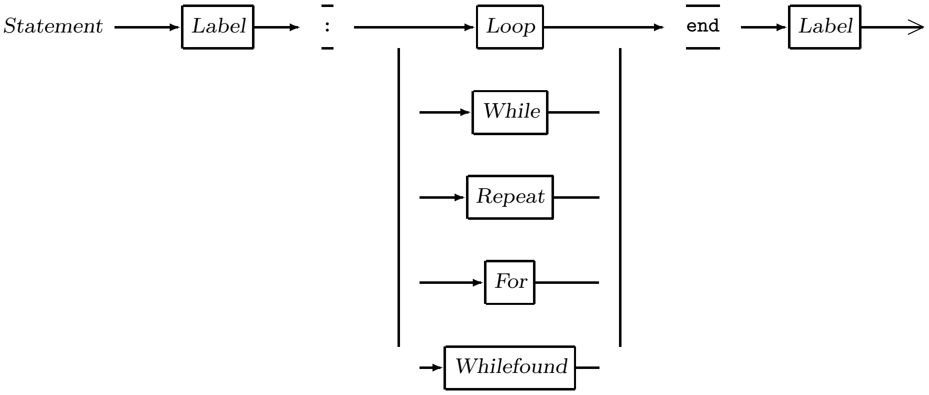
When executed in a spawned process (section 10.2.2), then this process will be terminated in the same way, as if a `return` statement with the same expression had been executed in the main procedure of this process. If no *Expression* is specified, then `om` will be returned as usual.

When executed in a main program, which has been started form the operating system, then the value of the optional *Expression* is passed to the operating system. Its meaning depends on the operating system. If no *Expression* is specified, then `om` is passed to the operating system by default as success code. All spawned processes of this application will be terminated. There exists implicitly a "`stop om;`" statement at the end of each main program.

The runtime system has to perform some work before returning the control to the operating system. This includes closing opened files, and releasing locks on persistent objects (section 9). The values of all still existing persistent variables are written to the persistent store provided that a success code is returned. Persistent values are left unchanged, if an error code is returned.

7.4 Loops

Loops specify repeated execution of enclosed statements. They may optionally be labeled:



Lexically, the label is an identifier. Label name in header and trailer have to match. The label name is only accessible to `quit` and `continue` statements (section 7.4.6). It is not allowed to use any visible names to label loops. However, multiple loops on the *same* level may be labeled with the same name. The following labeling is **not** allowed:

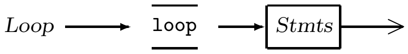
```
label:  loop ...   end label;
label := 1;
```

whereas the following is legal:

```
label:  loop quit label; end label;
label:  loop continue label; end label;
```

7.4.1 `loop`

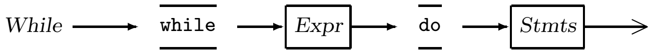
The infinite `loop` repeats the enclosed statements unconditionally:



The first statement following the end of the `loop` statement may only be reached via `quit` statements (section 7.4.6). It is also possible to leave it dynamically e.g. via `return` statements (section 4.5).

7.4.2 `while`

A `while` loop is written as follows:



Execution of such a loop proceeds as usual: the expression is evaluated and has to provide a Boolean value. If this value is `true`, then the enclosed statements are executed. After each execution of these statements, the expression is evaluated anew, and as long as it yields `true`, the statements continue to be executed. As soon as the expression yields `false`, looping ends, and execution proceeds with the first statement that follows the loop. Hence the statements within a `while` loop will be executed zero or more times:

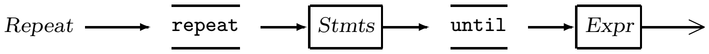
`while` E `do` S `end while`;

satisfies the fixed-point equation

`if` E `then` S; `while` E `do` S `end while`; `end if`;

7.4.3 `repeat`

An `repeat` loop is written as follows:



Execution of such a loop proceeds as usual: the enclosed statements are always executed at least once. Afterwards the expression is evaluated and has to provide a Boolean value. If this value is `false`, then the enclosed statements are executed again. As soon as the expression yields `true`, looping ends, and execution proceeds with the first statement that follows the loop. It follows that the statements within an `repeat` loop will be executed one or more times:

`repeat` S `until` E `end repeat`;

satisfies the fix-point equation

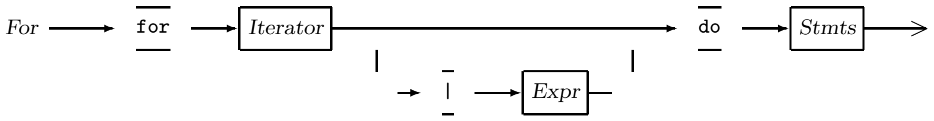
S; `if not` E `then repeat` S `until` E `end repeat`; `end if`;

```
program prog;
begin
    x := 5;
    S := {1, 2, 3};
    for x in S | x < 3  do
        S with:= 10 + x;   -- no effect on the iteration
    end for;
    -- now: x = 5 and S = {1, 2, 3, 11, 12}
end prog;
```

Figure 10: An example for the `for` loop.

7.4.4 `for`

A `for` loop is written as follows:



The *Iterator* is identical to set and tuple forming iterations and to quantified expressions (section 6.7). Forming iterations and quantified expressions imply implicit loops. The following is an *explicit* loop over all elements of the set `S` that satisfy `p(x)`:

`for x in S | p(x) do eat(x); end for;`

The variable `x` in this example is called the *bound variable* of the iterator. The variable `x` is local to the loop construct. This is similar to bound variables which are local to set and tuple formers resp. quantified expressions (sections 5.2 and 6.8). Each time the enclosed statements are executed, the bound variable is assigned the value of another element of `S`, provided `p(x)` yields `true`. The statements are executed exactly as many times as there are elements `x` in `S` for which `p(x)` yields `true`. When all elements of `S` have been dealt with, the program executes the statements following the end of the loop.

Iterations over tuples and strings can be described in exactly the same manner as iteration over sets. One significant difference between set and tuple resp. string iterators is that for the latter two we know the order in which components will be examined by the iteration: they are produced in order of increasing index.

It is **not** allowed to change the values of bound variables. They may be regarded as constants for each iteration.

Iterations proceed over copies, thus side effects on `S` by `eat(x)` in the above example would not influence the iteration over the copy of `S`. Consider the example in Fig. 10: the bound variable `x` is local to the `for` loop. Assignments to `x` are not allowed within the loop. The assignment to `S` has no effect on the iteration.

Multiple *simple* iterators in a `for` loop such as in

`for x in S, y in T | p(x,y) do eat(x,y); end for;`

are equivalent to nested `for` loops:

```
for x in S do
```

```
program prog;
begin
    x := 5;
    S := {1, 2, 3};
    whilefound x in S | x < 4  do
        eat (x);      -- now: x = 1 or x = 2 or x = 3
        S less := x ;
    end whilefound;
    -- now: x = 5 and S = { }
end prog;
```

Figure 11: An example for the `whilefound` loop.

```
    for y in T | p(x,y) do
        eat(x,y);
    end for;
end for;
```

However, `quit` and `continue` statements in a loop controlled by multiple iterators refer to the outermost iterator.

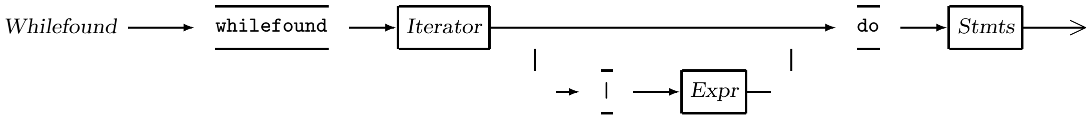
7.4.5 `whilefound`

In SETL the `exists` expression sets its bound variables on exit to the value found or `om` if no value was found. This is sometimes useful in constructs such as

```
(while exists x in { ...  } | condition(x) ) ...
```

In SETL, a found set-element is directly available via `x`, but this bound variable is not local to the loop. In PROSET, bound variables are local to `for` loops (section 7.4.4).

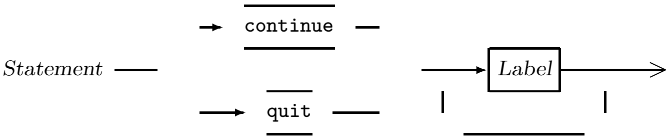
For these reasons we provide the `whilefound` loop:



Consider the example in Fig. 11: the body of the loop is executed provided an `exists` expression with the same iterator would yield `true` (section 6.8). The bound variables are local to the `whilefound` loop as they are in `for` loops. It is **not** allowed to change the values of bound variables. The iterator is reevaluated for each iteration unlike in `for` loops.

7.4.6 `quit` and `continue`

The `quit` and `continue` statements increase the syntactic flexibility of PROSET's loop constructs:



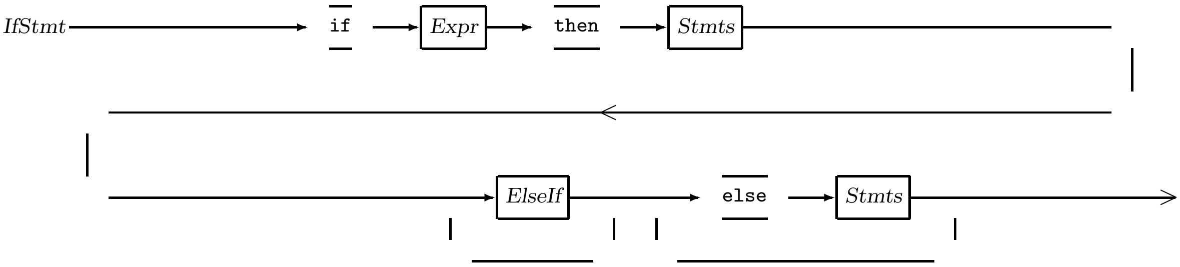
After executing the quit statement, looping ends, and execution proceeds with the first statement that follows the innermost loop provided no label is specified. If a label is specified, then the first statement that follows the respective loop is executed.

The continue statement lets the innermost loop proceed with the next iteration provided no label is specified. If a label is specified, then the corresponding loop proceeds with the next iteration. Only loop loops pass the control directly to the first enclosed statement. In while, repeat, and whilefound loops the condition is evaluated again. Control is passed only to the first statement enclosed, if the condition evaluates to true. In for loops the next iteration is performed provided that there is at least one iteration step left.

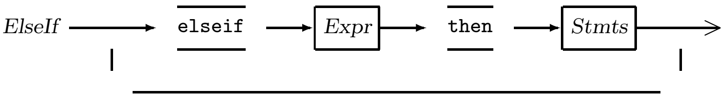
The label has to be a valid loop name. These statement are only allowed within loops.

7.5 if Statements

The if statement is used to direct program control along one of several alternative paths, chosen according to some stated condition:



Execution of the if statement proceeds as usual: the expression after the keyword if is evaluated and has to provide a Boolean value. If this value is true, then the statements following the keyword then are executed. If this value is false and there is no elseif and no else specified, then the first statement that follows the if statement is executed. If it yields false and there is no elseif but else specified, then the statements behind the keyword else are executed. The elseif cases are abbreviations for nested if statements where the respective end if trailers are removed:



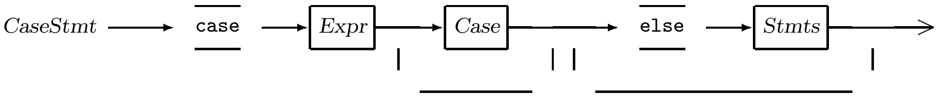
Conditional expressions are discussed in section 6.3.

7.6 case Statements

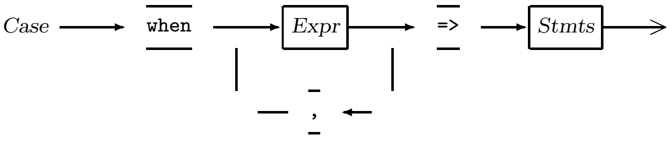
The case statement is a generalization of the if statement. Whereas the if statement controls the flow of execution of a program by choosing sequentially between two alternatives, the case statement allows us to choose among any number of alternative paths of execution:

```
program prog;
begin
    x := 1;
    loop
       case x
           when 1 => pass;
           when 1 => x := 2;
           when 2 => quit;
       end case;
    end loop;
end prog;
```

Figure 12: An example for the `case` statement.



For each branch a list of guarding expressions is specified:



Execution of the `case` statement proceeds as usual: the expression behind the keyword `case` is evaluated and may provide any type. If this value is equal to the value of exactly one of the guarding expressions, then the respective statements are executed. If this value is not equal to any guarding expression, then the statements behind the keyword `else` are executed. If this value is equal to no guarding expression, and there is no `else` specified, then the first statement that follows the `case` statement is executed.

If this value is equal to several guarding expressions, then one of them will be selected nondeterministically, and the respective statements are executed. There are no guarantees given for a fair selection of guarding expressions: it is not guaranteed that the program in Fig. 12 terminates. However, the program may terminate.

The `case` expression is discussed in section 6.4.

8 Exception Handling

In this section we discuss ProSet‘s exception handling mechanism. Having introduced the basic notions in 8.1, we illustrate them in the following subsection 8.2. The next three subsections 8.3 - 8.5 analyse the mechanism in detail. Exception handling and its influence on the treatment of persistent values is the subject of 8.6. Subsection 8.7 presents the most frequently used system-defined exceptions. We conclude this section with a comparison to other concepts.

8.1 Introduction in Exception Handling

Exception handling in ProSet supports the implementation of structured and reliable programs. A program should highlight the idea of the underlying algorithm. Thus, the main part of a program unit should implement the algorithm. *Exceptional conditions* (or in short *exceptions*) and their handling complete the program unit. Consequently, our concept of exception handling is based on the following definition.

An **exception** is a situation, which, once detected, interrupts the execution of the operation by which it is raised and which subsequently has to be communicated to the caller of that operation.

The caller has knowledge of the context in which the operation has been invoked. A flexible response to the activation of an exception is possible. In our approach the idea of *exception* must be understood as an arbitrary non-normal but not necessary an erroneous situation.

The course of actions when handling an exception is as follows (note that we also introduce some notations in boldface):

1. If the exception is detected in an operation, an **exception** is **raised**, i.e. this event is **signalled** to the **caller** of the operation.
2. The caller reacts by invoking a program unit that previously has been **associated** with the exception. These units are called **exception handlers**.
3. The associated handler is executed. The purpose of such handlers is:
    - diagnosing the situation,
    - handling the situation,
    - determining the subsequent flow of control.

A main decision in designing an exception handling model is whether a **signaller** (i.e. the exception raising unit) has to be **terminated** (i.e. the execution of the *caller* continues) or can be **resumed** (i.e. the execution of the *signaller* continues) having handled the exception. ProSet supports both models in a flexible manner.

8.2 Introductory Examples

First we describe a situation in which an operation is unable to satisfy its output assertions. An example is the input routine `my_fget` which is similar to the standard routine `fget`. If an end-of-file is encountered instead of an argument to be read in the input stream, it raises an exception classified as *escape* exception. Hence the operation will be terminated and control returns. Resuming the operation is not possible.

We will invoke `my_fget` as declared in Fig. 13 in a procedure that initializes data (Fig. 14). If it is found that an end-of-file mark read from input is inappropriate, `my_fget` raises the exception `UnexpectedEOF`, which is classified as `escape`.

In the procedure `Init_data` a handler (called `Input`) is associated with the exception `UnexpectedEOF`. The association is done using the construct `when UnexpectedEOF use Input`. This association is only valid for the call of `my_fget`. Thus, if the exception `UnexpectedEOF` is raised the handler is executed.

The handler‘s behaviour is twofold. Depending on a condition which is determined through a parameter to the handler it decides whether

- it is possible to use the subsequent statements of `Init_data` to finish initializing (executing the `return`-ended branch) or
- it is necessary to finish initializing in an alternative way (executing the `abort`-finished branch).

```
procedure my_fget(rd file, wr arg_1, ... ,wr arg_n);
     assigned_args := 0;
begin
     -- ...
     if feof(file) then
        escape UnexpectedEOF(assigned_args); -- raising the exception UnexpectedEOF
     end if;
     -- ...
end fget;
```

Figure 13: An example for handling range faults – signaller

```
procedure Init_data;
begin
     -- ...
     my_fget("a_file",in_1, ...,in_n)
          when UnexpectedEOF use Input; -- associating the exception with a handler
     -- ...

     handler Input(rd assigned_number);
     begin
          if Small(assigned_number) then -- diagnosing the parameter
             return; -- executing the next statement after the call of my_fget
                     -- all parameters are set to om
          else
             Init_substitute_data(); -- satisfying the output assertions of Init_data
                                     -- in an alternative way
             abort; -- regular termination of Init_data
          end if;
     end Input;
end Init_data;
```

Figure 14: An example for handling range faults – caller

```
-- ...
if P-file_not_available then
   signal P_file_incorrect(p_name); -- raising the exception P_file_incorrect
   Retry_with_new_name(p_name);     -- the operation may be resumed with a new name
end if;
-- ...
```

Figure 15: An example for handling domain faults – signaller

```
procedure Modify_persistent_data;
     visible persistent x : "otto"
         when P_file_incorrect use Substitute_p_file; -- associating the exception with
                                                      -- a handler
begin
     -- ...

     handler Substitute_p_file(rw p_file_name);
     begin
          Evaluate_situation(p_file_name,can_be_substituted); -- diagnosing the
          if can_be_substituted then                          -- situation
             p_file_name := "karl";
             resume; -- resuming with a new name
          else
             escape Persistent_error; -- A situation arises which cannot be resolved in
                                      -- this context.
                                      -- An exception is raised, which is signalled to
                                      -- the caller of Modify_persistent_data
          end if;
     end Substitute_p_file;
end Modify_persistent_data;
```

Figure 16: An example for handling domain faults – caller

The handler must terminate the signaller, this is due to the fact that we are dealing with an *escape*-exception.

In this first example we see how to handle *range faults*, which are in general unresolvable. However, *domain faults* may be corrected by the user. An example for such a situation is the *persistent declaration.*

Some lines from a routine similar to our mechanism treating persistent declarations (see section 9.1) are shown in Fig. 15. A part is shown in which an inappropriate $P - File$ name in detected. A persistent declaration is encountered in the procedure in Fig. 16. Here, a *signal*-exception is used (i.e. either termination or resumption is possible, depending on the caller). The signaller is prepared here to resume the execution, i.e. to retry the declaration with a new name. This is indicated by `resume` in the handler‘s body. But the signaller cannot decide whether or not the caller is able to provide a substitute name.

Using the same construct as in the first example for associating, a handler called `Substitute_p_file` is associated to the exception `P_file_incorrect`. The handler tries to provide a new name. If this fails due to the context, another exception (`Persistent_error`) is raised. This is another form of terminating the signaller.

An example using exception handling in expressions is presented in Fig. 17. Here the exception

```
-- ...
if (x/y)[illegal_operand use substitute]/abs(z) < 0 then
    put("zero_divide with y or negative expression");
end if;
-- ...

handler substitute();
    return -1;
end substitute;
```

Figure 17: An example for handling exceptions in expressions

illegal_operand is raised by system when a zero_divide occurs. To prevent the program from terminating if y equals zero a handler is associated which substitutes x/y by -1. This association is only valid for the inner division, nevertheless the program terminates if z equals 0, assuming no other handler is specified.

A last example concerns situations described by *monitoring*. Here, the signalled exception does not indicate failures, but rather a situation the caller wants to supervise. In general, there are two applications for monitoring:

- The caller wants to keep track of the operation‘s progress.
- Under certain, rare situations the operation needs additional informations which are costly to produce or extensive in nature.

Our example deals with the first application. The example is based on a module providing routines to work on a relation *Employee* in a relational database represented by tuples in a B-tree (Fig. 18). The procedure Iterate signals the exception Value and passes each item found in an instance of BTree as a parameter to the handler.

The user of Iterate monitors those items by looking for a special item, all or part of which can be collected. On every item it decides whether to stop evaluating items or to resume and evaluate more items.

In Fig. 19 Empl is an instance of BTree (see 11.3 for instantiation). Three applications of exception handling are presented. For each item, we gave a corresponding *SQL*-query. They are indicated in comments by enclosing them in brackets.

With the first call of Iterate all items are put out to the standard output (by executing the associated handler SelectAll).

With the second call the fact is used that the parameter to the exception (called btree(index) in Iterate and tup in the handler) is a rw-parameter, hence it will be written by the handler. Thus, a new value is passed back to the signaller.

The third use of monitoring starts calling Append which iterates on Empl. If a given value tup is in Empl appending is aborted. If tup is not in Empl, i.e. if all items in Empl are signalled to Append and are compared with the value to be inserted (tup), then the procedure Iterate terminates without raising an exception and the subsequent Insert is executed.

In the last example for resuming exceptions classified as *signal* are raised. Another variant is using *notify*-exceptions. They are expecting the caller to resume.

Another application is using exceptions to classify results (i.e. to provide additional informations using exception parameters) of an operation. Instead of a normal return an exception is raised to pass parameters classifying the evaluated result to the caller. This classification can be examined in a handler. To simulate the regular flow of control the handler executes a return-statement.

```
module BTree(wr Iterate, wr Insert);

...

procedure Iterate();
begin
     for index in [1..TreeSize] loop
         signal Value(btree(index)); -- every tuple in btree is signalled to the caller
     end for;
end Iterate;

procedure Insert();
begin
     -- ...
end Insert;

end BTree;
```

Figure 18: An example for monitoring – exporting module

8.3 Raising Exceptions

With the notion of *raising an exception* it is merely described that the occurrence of an exception is communicated to the caller and that control is handed over. Hence *raise* is a rather technical notion and may hide a rather complex flow of control.

We extend this notion by classifying exceptions into three classes.

notify: The caller obtains a progress report of the evaluation or is requested to carry out some evaluations not implemented by the signaller. This mechanism is akin to co-routines.

signal: The signaller is not able to determine either the handling nor the subsequent flow of control. Further handling depends on the context of the caller. The exceptional condition occurred here cannot be analyzed by the signaller.

escape: In contrast to *signal*-exceptions the signaller is able to realize that further treatment is not possible in the local context. This has to be done by the signaller.

For the sake of unambiguous modeling, each exception should be attached to only one of those classes.

Following that classification the restrictions concerning the subsequent flow of control leaving the handler are obvious, viz. :

notify: The signaller must be resumed.

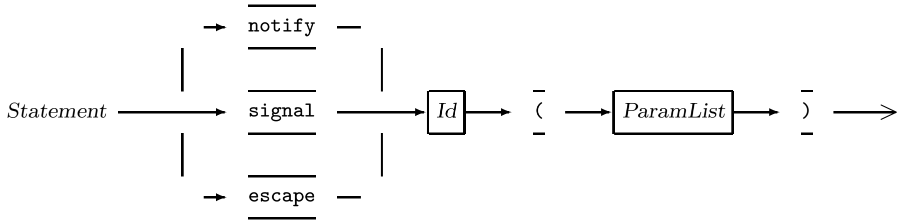
escape: The signaller must be terminated.

signal: The signaller may either be resumed or terminated.

Syntactically, we use a *raise-statement* to raise an exception:

```
visible row := 1;
begin
-- first use: output of all elements                              (select * from Empl)
Empl.Iterate()
     when Value use SelectAll;
-- second use: updating an relation (update Empl set Sal = Sal + 100 where Sal > 1000)
Empl.Iterate()
     when Value use UpdateSal;
-- third use: appending a tuple                       (insert into Empl values (tup) )
Append(tup);

handler SelectAll(rw tup);
begin
     putf("Tuple %d: %x\n",row,tup);
     row +:= 1;
     resume; -- resuming to get the next tuple
end SelectAll;

handler UpdateSal(rw tup);
begin
     if tup(Sal) > 1000 then
        tup(Sal) +:= 100;
     end if;
     resume; -- resuming to update the salaries
end UpdateSal;

procedure Append(rd tup);
begin
     Empl.Iterate()
          when Value use IsIn;
     Empl.Insert(tup);

handler IsIn(rw new);
begin
     if tup = new then
        abort; -- tup is just in Empl, appending is terminated
     else
        resume; -- resuming to get the next tuple
     end if;
end IsIn;
end Append;
```

Figure 19: An example for monitoring – three applications

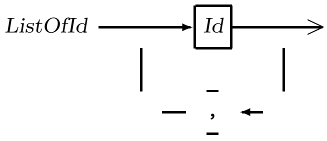


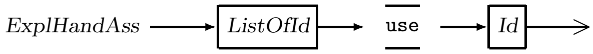
8.4 Associating Exceptions and Handlers

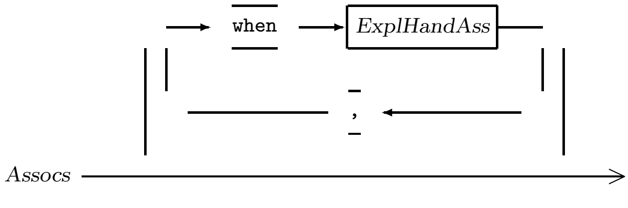
Exception handling takes place at the signaller‘s activation point. The association need not be defined there, it only takes effect at that point. As a consequence we require the handler name to be visible. The association can be bound syntactically to statements, to expressions or to declarations.

There are two forms of association: **explicit associations** for associating at a certain place in the executable part of the program and **implicit associations** for defining a default handler for some exceptions.

Explicit association: Particular exceptions and handler are associated using

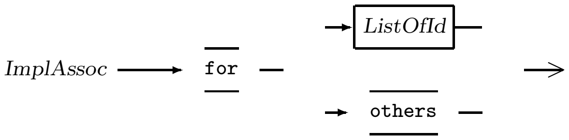






- the notation with square brackets for binding the association to expressions (see section 6.9),
- the `when .. use ..` construct for binding the association to statements (see 4.1) and declarations (see 4.4).

Implicit association: Implicit associations (`handler h (...) for e_1,...,e_n`) are indicated by listing exceptions in the header of those handlers that should be used to handle the exceptions listed in the scope of their visibility.



Explicit associations overwrite implicit associations as well as conflicting explicit associations in enclosing ranges. Associating exceptions to handlers binds syntactically with the same priority as selecting (see section 6.1).

We realize the following properties in our model:

- more than one exception can be associated to one handler,
- an exception raised by a particular operation can be treated differently at different calls of that operation,
- a particular exception that is raised by different operations can be treated differently,
- the association between handler and exception need not be explicit.

Note that binding of exceptions to handlers is done at runtime. Thus there are no static checks.

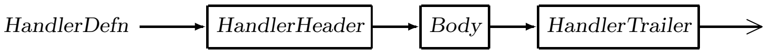
8.5 Handler

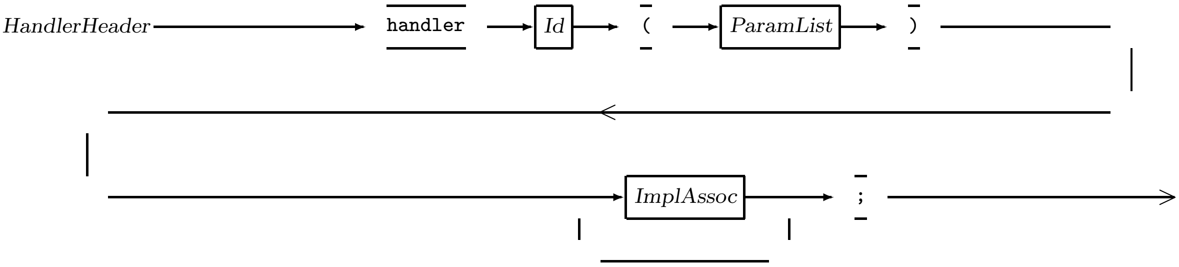
Handlers are similar to procedures, except that

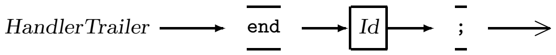
- they are only invoked by raising exceptions,
- they determine the flow of control in a different way,
- they do not have first class rights and it is not possible to generate a first class form from them.

Consequently, handlers are themselves program units in which a complete exception handling can happen. Similar to procedures, handler may have parameters. The list of formal parameters is given when a handler is declared. Actual parameters are supplied when the exception bound to the handler is raised. `rd` is the default parameter mode. `wr`- and `rw`-parameter make sense using the resuming model which allows communication between signaller and caller (the latter is represented by the handler). If the handler has a *return value*, it is used instead of the *return value* of the signaller.

The syntax for defining a handler is that of defining procedures extended by a construct for implicit associations:







Instead of listing particular exceptions a notation `others` may be used. With `others` we define a default handler:

- in explicit associations: `when others use ... ;`
- in implicit associations: `handler h(..) for others;`

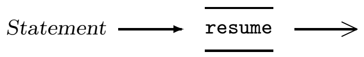
Definitions using `others` are valid for all exceptions not associated to any other handler. The compiler inserts the default handler at the outermost level, i.e. at the program level. This handler works as follows:

- it prints the name and the signaller of the exception,
- it terminates the program (`stop`) with the error code *not successful.*

By specifying own handlers the user can overwrite this handler.

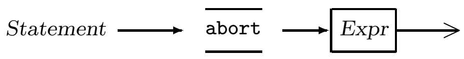
As mentioned above, the handler determines the flow of control. The `return`-statement of procedures is replaced by

resume: to resume the execution of the signaller.



return: to continue the execution of the caller with the next statement following the call of the signaller. Write-parameters of the signaller are given the value `om`. The return value of the handler is passed to the caller (see section 4.5).

abort: to terminate the signaller and the caller. The caller is terminated without raising an exception from the caller level, i.e. the caller terminates normally, and without giving the value `om` to `write`-parameters of the caller, but the actual values.



`return` and `abort` are forms of termination. If the signaller is terminated all locks held by the signaller are released. Note that persistent values are not updated. This is a temporary restriction to our exception handling model, for details see section 8.6.

There must be a correspondence between the *raise*-form (`notify, signal, escape`) and the handler response. Otherwise, the program will be terminated returning to the environment with the error code *not successful*, see section 4.2.5.

For termination of the entire program, the `stop`-command can be used (section 7.3.2).

8.6 Exception Handling and Transactions on Persistent Objects

Future versions of ProSet may integrate a more elaborate transaction mechanism. A `commit` for a transactions on persistent objects is right now realized implicitly by the `return`-statement. But a `transaction_abort` is missing. So far the termination of the signaller having handled any exception substitutes that functionality, although this is a severe restriction to the intuition concerning exceptions. We have to prevent updating modified persistent values in erroneous situations. Erroneous situations may occur using `escape` (errors detected by the signaller) or using `signal` (errors detected by the caller). Thus, updating should be prevented on every termination of the signaller. As a consequence, the following situations cannot be delt with appropriately:

- monitoring, where the caller decides whether to continue evaluating or to terminate. Here, all evaluated persistent results are destroyed,
- letting the handler carry out some evaluations yet satisfying the output assertions. Results may be destroyed.

8.7 Frequently Used Exceptions

Here we describe commonly used exceptions. These exceptions are raised always in the same manner and with the same parameterization.

`memory_full` *escaped* exception, raised whenever there are any problems allocating memory. There are no parameters passed.

`type_mismatch` *escaped* exception, raised whenever there is a mismatch between the expected type and the actual type of a value.

A macro `Predef_Exceptions` yielding all predefined exceptions is available.

8.8 General Remarks

In this last subsection we make some more general remarks on exception handling in ProSet. Readers interested in exception handling beyond these remarks are referred to Goodenough[4] or to Philbrow and Atkinson[5].

The mechanism described in the previous subsections is called *single level mechanism*, i.e. every exception is signalled only to the immediate caller. For every exception, there exists an associated handler. This may be the system-defined default handler, provided the user choose not to directly associate a handler to the exception. Every handler is executed on the caller level. Thus, there is no propagation of exceptions to outer levels. The usefulness of this approach using a default handler is illustrated by the following considerations.

A module exports some routines. These may use local functions. Assume one of these internal functions raises an exception that is not handled in an exported routine which invokes the function that raises the exception. Then the exception must be handled by the user outside the module. Exporting those exceptions

- may violate information hiding,
- may cause propagating uninterpretable exceptions due the increasing level of abstraction on outer levels in the hierarchy of procedure calls,

[4] J. B. Goodenough: Exception Handling: Issues and a Proposed Notation, Communication of the ACM 18(12), 683 - 696, 1975

[5] P. C. Philbrow and M. P. Atkinson: Events and Exception Handling in PS-algol, Computer Journal 33(2), 108 - 125, 1990

- violates ProSet's type system.

So, propagating exceptions in our model must be realized by raising exceptions adapted to the new context in a handler that handles an exception on a lower level of abstraction.

Our approach is in contrast to the so called *multi level mechanism*, which is the basis for exception handling e.g. in Ada or Clu. There exceptions that have been raised are propagated to outer levels until there is an explicitly associated handler. Handling takes place on that level. This is a simpler approach for realizing the handling of exceptions on an appropriate level (i.e. in a context which allows the successful handling). But the disadvantages follow from the discussion above. Information hiding may be violated and propagated exceptions are not adapted to the higher level of abstraction.

Another design decision we made is the distinction of exceptions and handlers. This has been done for the sake of flexibility. We want to provide a means to model and adequately handle exceptional conditions. It should be possible to handle an exception in different contexts in different ways.

An outstanding feature of exception handling in ProSet is the support of a termination and resumption model. In some languages resumption is considered not necessary, because many applications of resumption may be simulated by implementing an explicit retry of the failed operation. But a clear modeling of monitoring or co-routines is difficult without such a feature.

Concluding this section we want to point out that we interprete the notion of exception in a rather wide sense. This promotes exception handling from a device for failure handling to a mechanism dealing with a wider range of situations.

9 Persistence

We describe in this section persistent values and their containers which are represented by an abstract data type called $P-file$.

Prototyping aspects Persistence is interesting when considered in the context of software prototyping. Since prototyping combined with support for a semantic data model allows formulating data on a very high-level for modeling purposes, it is simply a matter of economy to make data persistent: once data are modeled it is not necessary to compute them each time they are used. Hence re-using data in a program does not necessarily mean recomputing them. A related concern for re-using data comes from the observation that more than one program may want to access them. Thus one program may generate data and another may want to access these data. Consequently one may have to face a situation where programs communicate through persistent data. We address the problem of concurrent access to these data only briefly here.

9.1 Persistent values

Persistence of data is characterized by the fact that these data outlive the program that generated them; this is in contrast to volatile data which vanish once the program ceases running. Persistence is an orthogonal property of values. Each value enjoying first class civil rights may be made persistent using the name with which it has been defined as a handle. Making use of a persistent value requires indicating this fact in the scope where this is to happen, and indicating the container from which this value is to be taken as well. Manipulating a persistent value requires an explicit indication of that respective intention. This discussion will make it clear that out mechanism for persistence allows creating and manipulating persistent data, but the the destruction of these data is outside the realm of the user's possibilities; destroying persistent data in the sense of removing a name to which a persistent value is bound (and the value as well) from a repository of persistent data will be a privilege of a tool in the environment.

Persistent values are kept in data structures called $P-files$ which in turn are identified in a program through a string literal. We will discuss $P-files$ in 9.2 in greater detail. Let for now "PFile" be a string literal denoting a $P-file$.

We declare a value accessed through the identifier x as persistent together with an indication that it is to be taken from the $P-file$ denoted by "PFile" using the following device in the declaration section of a range

```
persistent x: "PFile";
```

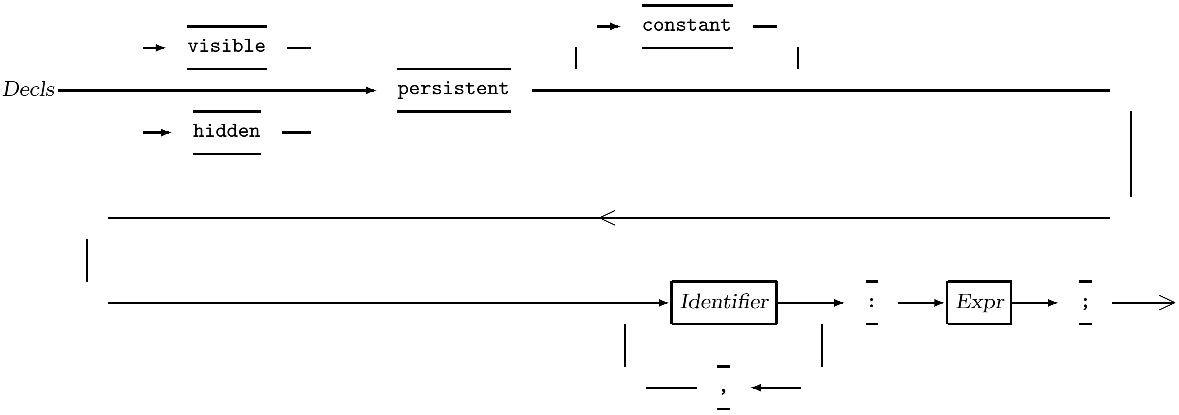
The keyword `persistent` may be used in conjunction with other declarative keywords like `constant` or `visible`, so that a name is declared as a `visible constant` taken from the $P-file$ "PFile" is declared by the mini-novel

```
persistent visible constant x: "PFile";
```

Notice that omitting the keyword `visible` here implies that `x` will be hidden, which is the default.

9.1.1 The Persistence Declaration

We discuss now the effect of declaring an object as persistent in a range. This looks syntactically as follows:



Hence identifiers are associated with a $P-file$; the identifiers may optionally be flagged as constants, indicating that they must not be changed.

Before continuing, we want to remind the reader of a *lock*[6]: locks are being held by *transactions*, i.e. by operations on a $P-file$ for the purpose of manipulating persistent values, and they are associated with these values (which will be called *items* to avoid the all-pervasive term *object*). Transactions themselves are administered by a *transaction manager*, i.e. a program component in the run time system of the ProSet program under consideration. Some of the manager's responsibilities will be outlined below. Returning to locks, we note that for the time being locks come in two flavors, they may be read locks, and write locks, respectively:

read lock a *read lock* on an item prevents the item to be written by any transaction other than the one holding the lock. Consequently holding a read lock on item x means for transaction τ that it may read the item, and that no other transaction may change the value of x while τ holds the lock. More than one transaction may hold a read lock on the same item,

[6] For a detailed discussion, we refer the reader to J. D. Ullman: Principles of Database and Knowledge-Base System, vol. I. Computer Science Press, Rockville, MD, 1988, Chapter 9

write lock a *write lock* on an item permits the transactions holding that lock to read and to write the item. Given an item x, then at most one transaction may hold a write lock on x at any given time, and no other transaction may hold a read lock on x at the same time.

Variable declaration When encountering the declaration

```
persistent x: "PFile";
```

upon entering a range, the steps outlined below are taken.

First the program's environment is searched for the $P - file$ `"PFile"`. It this is not successful, or if the user does not have the proper rights for reading and for writing `"PFile"`, the program aborts. Let us denote by `x."PFile"` the incarnation of `x` we are discussing (other $P - file$s may contain `x`'s, too, given the popularity of that identifier). If the transaction manager finds a lock being set on `x."PFile"` by another program, the execution of the program under consideration is suspended until the value is accessible to it for writing. Now suppose `"PFile"` exists, the user has appropriate rights, and no lock is set on `x."PFile"`. The transaction manager providentially grants a write lock on `x."PFile"` to this transaction, independently of whether or not `x."PFile"` exists, and then checks the existence of `x."PFile"`. If it exists, the value is loaded, and everything is fine: when the range is left, the new value for `x."PFile"` is written to `"PFile"`, and the write lock is revoked. If, however, `x."PFile"` does not exist, we are in trouble. A `notify` exception `p_missing_name` is raised, and the default handler (cp. 8.5) aborts the program. If `p_missing_name` is handled by the user, then `x` is inserted into `"PFile"` as having the value `om`, provided the handler terminates with `resume`. The write lock remains as it is, and execution of the unit containing the range continues. If the handler does not terminate with `resume`, however, `x` is not generated, the write lock is revoked, and the program aborts according to the `notify` restrictions.

Constant declaration When encountering the declaration

```
persistent constant x: "PFile";
```

upon entering a range, *mutatis mutandis* the following steps are taken: The program's environment is searched for the $P - file$ `"PFile"`. It this is not successful, or if the user does not have the proper rights for reading `"PFile"`, the program aborts. Let us denote again by `x."PFile"` the incarnation of `x` we are discussing. If the transaction manager finds a write lock being set on `x."PFile"` by another program, the execution of the program under consideration is suspended until the value is accessible to it for reading. Now suppose `"PFile"` exists, the user has appropriate rights, and no write lock is set on `x."PFile"`. The transaction manager providentially grants a read lock on `x."PFile"` to this transaction, and then checks the existence of `x."PFile"`. If it exists, the value is loaded as a constant, and when the range is left, the read lock is revoked. If, however, `x."PFile"` does not exist the program is aborted.

The example in Fig. 20 demonstrates the use of a simple persistent procedure. The exception `p_missing_name` is handled so that it resumes after having generated the missing persistent value.

When leaving the procedure `demo` we observe as a side effect that the `t_demo` is now a persistent value in the $P-file$ `otto`. Note that we cannot directly make the procedure declared above persistent, since procedures do not have first class rights. Applying the `closure` operator (elevating the procedure to a `function`) enables us to do this. The procedure `user` makes use of `t_demo`, as indicated in Fig. 21. Hence each time invoking *user* we will find `t = 13` in the output.

```
procedure demo;
   visible  t := 13;
   persistent t_demo: "otto" when p\_missing\_name use DoResume;
begin
   t_demo1();
   t_demo := closure t_demo1;

   procedure t_demo1();
   begin
       if t < 17 then
          put("t = ", t);
          t + := 1;
       end if;
   end t_demo1;

   handler DoResume();
   begin
           resume;
   end DoResume;
end demo;
```

Figure 20: Persistent Procedure

```
procedure user();
   persistent constant t_demo: "otto";
begin
   t_demo();
end user;
```

Figure 21: Making use of a persistent procedure

9.2 The ADT $P - file$

Persistent values will usually be stored outside the program defining or using them. The specific kind of location is implementation dependent, e.g. it may happen that cache memory is large enough to hold the contents of a small collection of persistent values at run time, but it may also be the case that primary memory is at premium and that all persistent values have to be swapped to secondary memory. Hence we specify only the operations on the containers for persistent values (called $P-files$), and we leave the realization of these operations together with these specific management of internal or external storage to an implementation.

As a first approximation, $P - files$ may be compared to the well known *archives* under UNIX. An archive consists of a table of contents and of the files (mostly binaries) which are stored in it. Archives may be accessed in a number of ways: one can read the table of contents, one may insert or delete an element from an archive and one may extract named elements from it. In addition, the archive is a UNIX-file, thus it may be identified through an identifier which is admissible under a particular shell.

The abstract data type $P - file$ is represented by a string as an identifier. The string may bear some internal structure e.g. for indicating a hierarchy of $P - file$s. This will be exploited by a suitable tool. The identifier representing a $P - file$ is used to access the $P - file$ in the same way a file is accessed using its name.

The following operations are provided by the tool for $P - file$s:

- creating an initially empty named $P - file$; the name is a valid identifier,
- removing a named $P - file$. The persistent values stored in it are no longer accessible,
- listing the table of contents of a named $P - file$. This indicates which values are stored there by enumerating the names in some order,
- removing a named element from a named $P - file$,
- compressing a named $P - file$ akin to garbage collecting primary memory.

These operations are carried out by tools outside ProSet programs.

10 Multiprocessing

Tuple space communication in ProSet as presented in this section is designed for *multiprocessing* (single application running on multiple processors) as opposed to *multiprogramming* (separate applications). Multiprogramming is done via handling persistent data objects (section 9).

The following subsections will discuss Linda, process creation, and tuple-space communication in ProSet.[7]

10.1 Linda

Linda is a coordination language concept for explicitly parallel programming in an architecture independent way, which has been developed by David Gelernter at Yale University.[8] Communication in Linda is based on the concept of tuple space, i.e. a virtual common data space accessed by an associative addressing scheme.

[7] A more detailed discussion and some examples may be found in W. Hasselbring: "On Integrating Generative Communication into the Prototyping Language ProSet", Informatik-Bericht 05-91, University of Essen, 1991.

[8] For a full account to parallel programming in Linda see N. Carriero and D. Gelernter: "How to write parallel programs", MIT Press, 1990.

Reading access to tuples in tuple space is associative and not based on physical addresses — in fact, the internal structure of tuple space is hidden from the user. Reading access to tuples is based on their expected content described in so-called *templates.* This method is similar to the selection of entries from a data base. Each component of a tuple or template is either an *actual*, i.e. holding a value of a given type, or a *formal*, i.e. a placeholder for such a value. A formal is prefixed with a question mark. Tuples in tuple space are selected by a matching procedure, where a tuple and a template are defined to match iff they have the same structure (corresponding number and type of components) and the values of their actuals are equal to the values of the corresponding tuple fields.

LINDA defines six operators, which may be added to a sequential computation language. These operators enable sequential processes, specified in the underlying computation language, to access the tuple space. The `out` operation evaluates and adds a tuple to the tuple space. The `eval` operation adds an unevaluated (active) tuple to the tuple space. The fields of an `eval` tuple are evaluated concurrently yielding one thread of execution for every field. The `in` operation attempts to withdraw a specified tuple from tuple space. Tuple space is searched for a matching tuple against the template supplied as the operation's argument. When and if a tuple is found, it is withdrawn from tuple space, and the values of its actual fields are bound to any corresponding formals in the template. Tuples are withdrawn *atomically*: a tuple can be grabbed by only one process, and once grabbed it is withdrawn entirely. If no matching tuple exists in tuple space, the process executing the `in` suspends until a matching tuple becomes available. If many tuples satisfy the match criteria, one is chosen arbitrarily. The `rd` operation is the same as `in`, with actuals assigned to formals as before, *except* that the matched tuple remains in tuple space. The predicate operations `inp` and `rdp` attempt to locate a matching tuple and return `0` if they fail; otherwise, they return `1` and perform actual-to-formal assignment as described above. The only difference with `in`/`rd` is that the predicates will not block if no matching tuple is found.

10.2 Process Creation

In this section we will present an adaptation of the approach for process creation known from MULTILISP to set-oriented programming, where new processes may be spawned inside and outside of tuple space.

10.2.1 MULTILISP's Futures

MULTILISP[9] augments SCHEME with the notion of *futures* where the programmer needs no knowledge about the underlying process model, inter-process communication or synchronization to express parallelism. He only indicates that he does not need the result of a computation immediately (but only in the "future") and the rest is done by the runtime system. Instead of returning the result of the computation, a placeholder is returned as result of process spawning. The value for this placeholder is undefined until the computation has finished. Afterwards the value is set to the result of the parallel computation: the future *resolves* to the value. Any process that needs to know a future's value will be suspended until the future resolves thus allowing concurrency between the *computation* of a value and the *use* of that value. The programmer is responsible for ensuring that potentially concurrently executing processes do not affect each other via side effects. An example:

```
(let ((x (future expr1))
      (y expr2))
     ( body ))
```

The value for `x`, which will be the result value of *expr1*, is evaluated concurrently to *expr2* and *body*. The value for `y`, which will be the result value of *expr2*, is evaluated before the evaluation of *body* will

[9] A full account to MULTILISP may be found in R.H. Halstead: "MULTILISP: A language for concurrent symbolic computation", TOPLAS, 7(4), 501–538, 1985

be started. When *body* needs the value of `x`, and `x` is not yet resolved, it *touches* the future of `x` and is suspended until the future resolves. Most operations, e.g. arithmetic, comparison, type checking, etc., touch their operands. This is opposed to simple *transmission* of a value from one place to another, e.g. by assignment, passing as a parameter to a procedure, returning as a result from a procedure, building the value into a data structure, which does **not** touch the value. Transmission can be done without waiting for the value.

10.2.2 Process Creation in ProSet

Futures in Multilisp provide a method for process creation but no means for synchronization and communication between processes, except for waiting for each other's termination. In our approach the concept for process creation via futures is adapted to set-oriented programming and combined with the concept for synchronization and communication using tuple spaces.

Multilisp is based on Scheme, which is a dialect of Lisp with lexical scoping. Lisp and Scheme manipulate pointers. This implies touching in a value-requiring context and transmission in a value-ignoring context. This is in contrast to ProSet that uses value semantics, i.e. a value is never transmitted by reference. However, there are a few cases where we can ignore the value of an expression: if the value of an expression is assigned to a variable, we do not need this value immediately, but possibly in the *future*.

Process creation in ProSet is provided through the unary operator `||`, which may be applied to an expression (preferably a function call). A new process will be spawned to compute the value of this expression concurrently with the spawning process analogously to futures in Multilisp.

If this *process creator* `||` is applied to an expression that is immediately assigned to a variable, the spawning process continues execution without waiting for the termination of the newly spawned process. At any time the *value* of this variable is needed, the requesting process will be suspended until the future resolves (the corresponding process terminates) thus allowing concurrency between the *computation* and the *use* of a value. Consider the following statement sequence to see an example:

```
x := || p();        -- statement 1
...                 -- Some computations without access to x
y := x;             -- statement 2
```

After statement 1 is executed the process `p()` runs in parallel with the spawning process. Statement 2 will be suspended until `p()` terminates, because a copy is needed (value semantics). This is in contrast to Lisp where an assignment would copy the address and ignore the value. If `p()` resolves before statement 2 has started execution, then the resulting value will be assigned immediately.

Also, if a compound data structure is constructed via a set or tuple forming enumeration, and this data structure is assigned immediately to a variable, we do not need the values of the enumerated components immediately, thus the following statement allows concurrency as above:

```
x := { || p(), 123, || q() };
```

If you replace statement 1 in the previously discussed statement sequence by this statement, then concurrency would be achieved as before. Such parallel set or tuple forming expressions may be compared with constructing lists via the function `cons` in Multilisp, where the list components are also not touched.

Conversely, in iterative formers such as `{ || p() .. || q()}` the initial and final values for the iteration are required, and thus the termination of `p()` and `q()` has be awaited.

Compound data structures in ProSet are always touched as a whole. Access to tuple or set components as in

```
x := [ || p(), || q() ];
y := x(1);
```

touches the whole tuple or set, thus both, `p()` and `q()`, have to terminate before `x(1)` is accessible. We decide to touch compound data structures always as a whole to retain consistency for tuples and sets.

The actual parameters for the procedure `doit` in

```
x := doit ( || p(), || q() );
```

are evaluated concurrently to each other. The procedure `doit` is invoked when both processes have terminated their execution. The programmer of `doit` does not have to know that the actual parameters are evaluated concurrently. The runtime system takes care of that. Returning an expression that is prefixed by `||` achieves concurrency according to the context of the corresponding procedure invocation.

In summary: concurrency is achieved only at creation time of a process and maintained on immediately assigning to a variable, storing in a data structure, passing as a parameter to a procedure, returning as a result from a procedure, and depositing in tuple space (this is discussed in section 10.3.1). Every time one tries to obtain a copy one has to wait for the termination of the corresponding process and obtains only then the returned value. The unique moment at which a value is not touched is on creation of the respective process, because this is the unique moment at which no copy is needed.

Analogously to statements, concurrency is achieved in declarations like

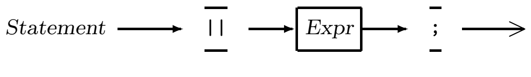
```
constant c := || p();
visible x := [ || p() ];
```

If, after one of such a declaration or similar statement, `x` is assigned a new value, then the corresponding spawned process will be abandoned[10] provided it is still running. Hence, the automatic garbage collection provides a means for explicit process termination outside tuple space. The automatic garbage collection also should take place when the existence of an object terminates. This is the case e.g. on return from a procedure or on program termination. Automatic garbage collection does not apply to processes within tuple space. Process termination within tuple space is discussed in section 10.4.

Process-spawning statements Also the following statement, which spawns a new process, is allowed:

```
|| p();
```

The return value of such a process will not be available and it is not possible to abandon it explicitly. The general form of such a process-spawning statement is



If the process creator `||` is applied in an expression that is an operand to any operator, then this operator will wait for the return value of the created process. Operators **always** touch the values of their operands, and thus have to wait for the termination of processes that compute their operands. For instance, in the following expressions the return values are needed:

[10] Killing processes has to be done with care, especially when such processes are still doing tuple-space operations.

```
   1 + || p()
["x"] + [|| p()]
    - || p()
  type || p()
```

As any other operator, the process creating operator `||` touches the value of its operand. Hence, an expression such as "`|| || p()`" does not make much sense: the leftmost `||` has to wait for the termination of `p()`. However, it is syntactically correct. The following expression spawns three processes:

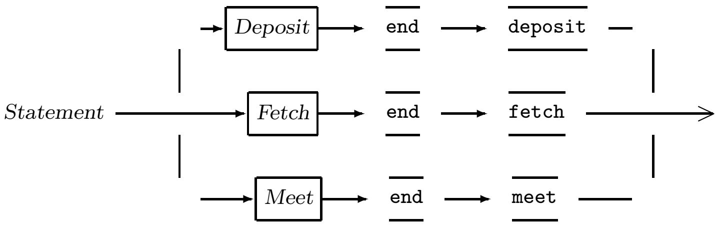
```
|| { || p(), 123, || q() }
```

The set-forming process has to wait for the termination of `p()` and `q()`.

Side effects and write parameters are not allowed for processes. Communication and synchronization is done only via tuple-space operations. However, processes may access common, persistent data objects.

10.3 Tuple-Space Operations

ProSet provides three tuple-space operations:



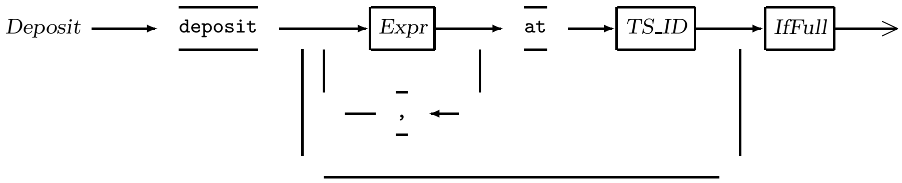
The `deposit` operation deposits new tuples into tuple space, the `fetch` operation fetches and removes a tuple from tuple space, and the `meet` operation meets and leaves a tuple in tuple space. It is possible to change the tuple's value while meeting it.

There is no difference between ProSet-tuples and Linda-tuples. Linda and ProSet both provide tuples thus it is quite natural to combine them on the basis of this common feature. However, a tuple space is a multiset of tuples, whereas the type system of ProSet does not directly provide the notion of multisets or bags. One could model multisets e.g. via maps from tuples to counts, but this would not reflect the matching provided by tuple spaces.

Tuple-space operations are statements that yield **no** values. They should not be confused with operators in expressions that always yield values.

10.3.1 Depositing Tuples

The `deposit` operation deposits tuples into a specified tuple space:



It is possible to deposit several tuples in an expression list into one tuple space and several such expression lists into multiple tuple spaces by one statement, but there are no guarantees made for the chronological order of availability of these tuples for other operations that wait for them (see below). The tuples are handed over to the tuple-space manager, which adds them to the tuple space in an arbitrary order. There is no guarantee given for the time of availability of deposited tuples for matching templates.

All expressions are evaluated in arbitrary order, before any tuple is put into tuple space. The expressions must yield tuples to be deposited in tuple space; if not, an exception will be raised. The identifier *TS_ID* will be discussed in sections 10.4 and 13.8.

We distinguish between passive and active tuples in tuple space. If there are no executing processes in a tuple, then this tuple is added as a passive one (cp. `out` of C-Linda). If there are executing processes in a tuple, then this tuple is added as an active one to tuple space. Depositing a tuple into tuple space does not touch the value. When all processes in an active tuple have terminated their execution, then this tuple converts into a passive one with the return values of these processes in the corresponding tuple fields. Active tuples are invisible to the other tuple-space operations until they convert into passive tuples. The other two tuple-space operations apply only to passive tuples.

Depositing in tuple space does not touch the values of tuples to be deposited thus the following statement deposits an *active* tuple into tuple space:

```
deposit [ || p() ] at TS end deposit;
```

whereas the following statement sequence deposits a *passive* tuple:

```
x := [ || p() ];
deposit x at TS end deposit;
```

The `deposit` operation copies the value of `x`, and thus has to wait for the termination of `p()` (see also section 10.2.2).

Limited Tuple Spaces

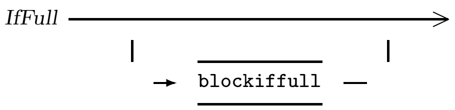
Because every existing computing system has only finite memory, the memory for tuple spaces will also be limited. Pure tuple-space communication does not deal with *full* tuple spaces: there is always enough room available. Thus most runtime systems for Linda hide the fact of limited memory from the programmer. In ProSet-Linda the predefined exception `ts_is_full` will be raised by default when no memory is available for a `deposit` operation. If there are multiple tuples specified in one `deposit` operation, then none of them has been deposited when `ts_is_full` is raised. Conversely, this exception will be raised, if at least one tuple cannot be deposited in any tuple space.

In ProSet it is possible to specify a handler for an exception by annotating a statement with a new binding between exception name and handler name (section 8):

```
deposit [ x ], [ || f(x) ]
     at TS
end deposit when ts_is_full use MyHandler;
```

If not explicitly specified with the `stop` statement, the user-defined handler will not abort the program. If the handler executes a `return` statement, then the statement following the `deposit` will be executed and none of the tuples of the respective `deposit` will be deposited. If the handler executes a `resume` statement, then the `deposit` operation tries again to deposit the tuples. The programmer has to take care not to produce an infinite alternation between raising and resuming. If no user-defined handler is given, then the runtime system will abort the program with an error code.

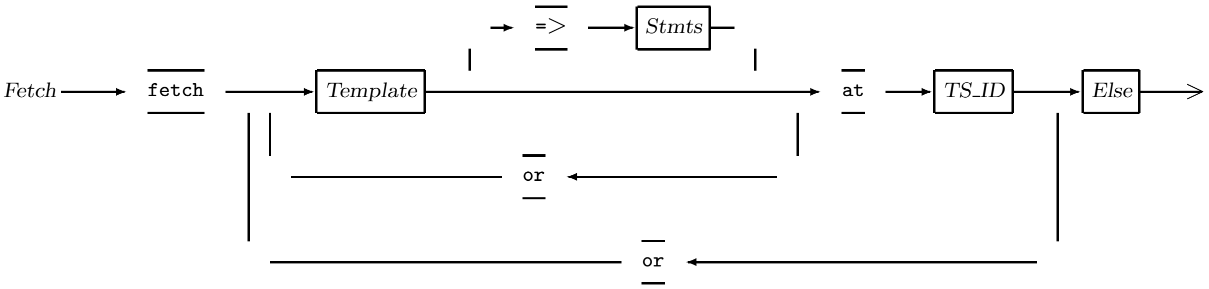
Optionally, the programmer may specify that a `deposit` operation will be suspended until space is available again:



You may specify `blockiffull` as well as a new binding between the predefined exception `ts_is_full` and a handler name, but then `ts_is_full` will never be raised. The compiler will print a warning message in this case.

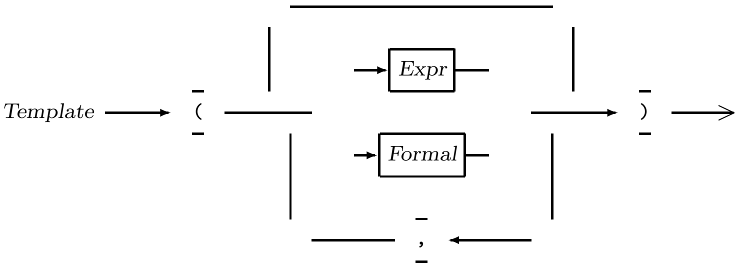
10.3.2 Fetching Tuples

A `fetch` operation fetches and removes one tuple from a tuple space:

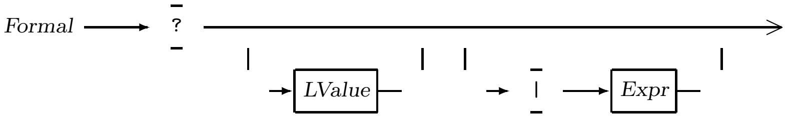


It is possible to specify several templates for multiple tuple spaces in one statement, but only one template may be selected nondeterministically (section 10.3.5). If there are no `else` statements specified (see below) then the statement suspends until a match occurs. The selected tuple is removed from tuple space. If statements are specified for the selected template, these statements are executed (only for this template).

A template consists of a list of ordinary expressions and so-called *formals*:



The expressions are called *actuals*. They are at first evaluated in arbitrary order. The list is enclosed in parentheses and not in brackets in order to set the templates apart from tuples. Note that a template may be empty to match the empty tuple `[]`. The fields that are preceded by a question mark are the *formals* of the template:



As usual | means *such that*. The Boolean expression behind | may be used to customize matching. A tuple and a template match, iff all the following conditions hold:

- the tuple is passive (matching touches the value)
- the numbers of fields are equal
- types and values of actuals in templates are equal to the corresponding tuple fields
- the Boolean expressions behind | in the formals evaluate to `true`. If no such expression is specified in a formal, then this field matches unconditionally

The *l*-values specified in the formals are assigned the values of the corresponding tuple fields provided matching succeeds. If such *l*-values occur in the expressions of formals, then these expressions are evaluated with the corresponding *old* values of these *l*-values. The symbol `$` may be used like an expression as a placeholder for the value of the corresponding tuple field:

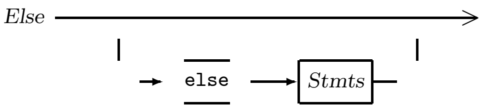
```
fetch ( "x", ? x |(type $ = integer) ) at TS end fetch;
```

This formal only matches integer values in the corresponding tuple field. Conversely, the formal "`? x |(type x = integer)`" would test the value of `x` *before* the assignment of the corresponding tuple value takes place. It is only allowed to use the symbol `$` this way in expressions that are parts of formals. It is not possible to access the values of multiple tuple fields within expressions of formals.

If an *l*-value is specified more than once, it is not determined which of the possible values is assigned. If no *l*-value is specified, then the corresponding value will not be available. You may regard a formal without an *l*-value as a "don't care" or "only take care of the condition" field.

Non-Blocking Matching

It is possible to specify `else` statements to be executed, if none of the templates matches:



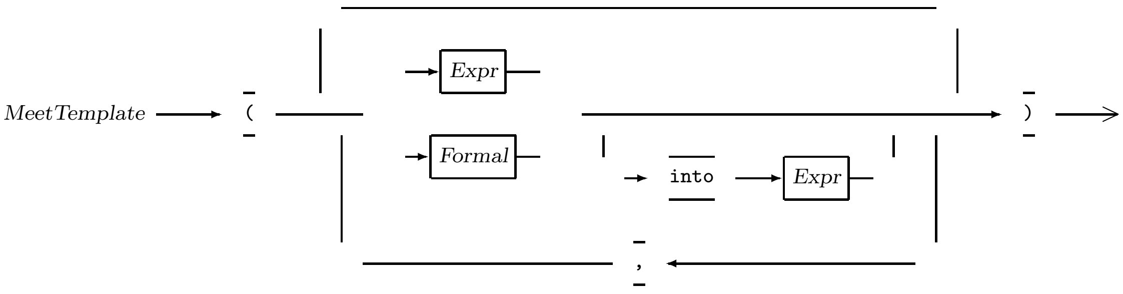
We will use the notion *non-blocking matching* if `else` statements are specified as opposed to *blocking matching* if no `else` statements are specified.

An example for the `fetch` operation:

```
fetch ( "name", ? x |(type $ = integer) ) => put("Integer fetched");
   or ( "name", ? x |(type $ = set) )     => put("Set fetched");
   at TS
 else put("Nothing fetched");
end fetch;
```

10.3.3 Meeting Tuples

The `meet` operation meets and leaves one tuple in tuple space. It is possible to change the tuple while meeting it. Exchanging the keyword `fetch` with `meet` and the nonterminal *Template* with *MeetTemplate* in the first syntax diagram of section 10.3.2, one obtains the syntax for the `meet` operation:



The expressions are evaluated as usual, the formals are used to create templates, which are used for matching as with the `fetch` operation (section 10.3.5). If no `else` case is specified, then the statement suspends until a match occurs. The values of the tuple fields that were fetched for the corresponding formals of the template are assigned to the corresponding *l*-values. If statements are specified for the selected template, these statements are executed (only for this template).

An example for the `meet` operation:

```
meet  ( "name", ? x |(type $ = integer) ) => put("Integer met");
   or ( "name", ? x |(type $ = set) )     => put("Set met");
   at TS
 else put("Nothing met");
end meet;
```

If there are no `into`'s specified as in this example, then the selected tuple is **not** removed from tuple space. This case may be compared with the `rd`/`rdp` operations of C-Linda. Except for the fact that the `meet` operation without `into`'s leaves the tuple it found in tuple space, it works like the `fetch` operation.

Changing Tuples We allow to change tuples while meeting them in tuple space. This is done by specifying expressions `into` which specific tuple fields will be changed. Tuples, which are met in tuple space, may be regarded as shared data since they remain in tuple space; irrespective of changing them or not.

If there are `into`'s specified then the tuple is at first fetched from the tuple space as it would be done with the `fetch` operation. Afterwards a tuple will be deposited into the same tuple space, where all the tuple fields without `into`'s are unchanged and all the tuple fields with `into`'s are updated with the values of the respective expressions. Consider

```
meet ( "x", ? |(type $ = integer) into $+1 ) at TS end meet;
```

which is equivalent to the series of statements with `temp` as a fresh name:

```
fetch ( "x", ? temp |(type $ = integer) ) at TS end fetch;
deposit [ "x", temp+1 ] at TS end deposit;
```

Indivisibility is guaranteed, because fetching the passive tuple at starting and depositing the new passive or active one at the end of the user-defined operation on shared data are atomic operations. Note that another process, which does simultaneously a non-blocking `meet` operation with the same template, may execute its `else` statements.

The symbol `$` may be used like an expression as before. It is not necessary to specify *l*-values for changing tuple fields. However, they may be used:

```
meet ( "string", ? x |(type $ = integer) into $+1 ) at TS end meet;
```

Here `x` is assigned the value of the corresponding tuple field *before* the change takes place. Remember that it is only allowed to use the symbol `$` this way in expressions that are parts of formals.

The `meet` operation will not raise `ts_is_full` when a tuple space is full while depositing a changed tuple, when the number of allowed tuples in this tuple space is exceeded (see also section 10.4). The place for the tuple met will be reserved for the whole operation. However, if the changed tuple exceeds a physical memory limit, this will raise `ts_is_full`.

10.3.4 Comparison with the C-LINDA Operations

The `deposit` operation comprises the `out` and `eval` operations of C-LINDA. You might compare depositing of active tuples with `eval`, but it is not exactly the same, however, because all fields of an `eval` tuple are executed concurrently. This is a noteworthy difference: according to the semantics of `eval` *each* field of a tuple is evaluated concurrently. But probably no system will create a new process to compute e.g. a plain integer constant. The system has to decide, which fields to compute concurrently and which sequentially. Similar problems arise in automatic parallelization of functional languages: here you have to reduce the existing parallelism to a reasonable granularity. In our approach the programmer has to communicate his knowledge about the granularity of his application to the system.

The `fetch` operation merges `select` of ADA and `in` resp. `inp` of C-LINDA. The `meet` operation merges `select` of ADA, `rd` and `rdp` of C-LINDA, and allows for changing tuples in tuple space.

10.3.5 Nondeterminism and Fairness while Matching

There are two sources for nondeterminism while matching:

1. Several matching tuples exist for the templates: one tuple will be selected nondeterministically.
2. The selected tuple matches several templates: one template will be selected nondeterministically.

If in any case there is only one candidate available, this one will be selected. There are several ways for handling fairness while selecting tuples or templates that match if there are multiple candidates available. We assume a fair scheduler to guarantee process fairness, which means that no single process is excluded of CPU time forever. We will now discuss *fairness of choice* which is important for handling the nondeterminism derived from matching. There exist some fairness notions in the literature:

Unconditional Fairness Every process will be selected infinitely often.

Weak Fairness If a process is enabled continuously from some point onwards then it eventually will be selected. Weak Fairness is also called *justice*.

Strong Fairness If a process is enabled infinitely often then it will be selected infinitely often.

In ProSet the following fairness guarantees are given for the two sources for nondeterminism as mentioned above:

1. Tuples will be selected without any consideration of fairness.
2. Templates will be selected in a weakly fair way.

Since deposited tuples are no longer connected with processes, it is reasonable to select them without any consideration of fairness. Linda's semantics do not guarantee tuple ordering — this aspect remains the responsibility of the programmer. If a specific order in selection is necessary, it has to be enforced via appropriate tuple contents.

To specify weakly fair selection of templates by means of temporal logic we introduce the following abbreviations for predicates on a process P and a template T:

$$\begin{array}{rcl}
\mathbf{E} & = & \text{“Process } P \text{ \underline{e}xecutes a blocking \texttt{fetch} or \texttt{meet} operation with template } T\text{”} \\
\mathbf{M} & = & \text{“There is a \underline{m}atching tuple for template } T \text{ in tuple space”} \\
\mathbf{B} & = & \text{“Process } P \text{ is \underline{b}locked with template } T\text{”} \\
\mathbf{S} & = & \text{“Template } T \text{ is \underline{s}elected for a matching tuple provided there exists one”} \\
\mathbf{A} & = & \text{“Process } P \text{ is \underline{a}ctivated (template } T \text{ was selected)”}
\end{array}$$

Now the above-given fairness guarantee may be formulated as follows:[11]

$$\mathbf{E} \wedge \mathbf{M} \wedge \mathbf{S} \;\Rightarrow\; \mathbf{A} \tag{1}$$

$$\mathbf{E} \wedge \neg(\mathbf{M} \wedge \mathbf{S}) \;\Rightarrow\; \mathbf{B} \tag{2}$$

$$\mathbf{B} \wedge \Box\Diamond\mathbf{M} \;\Rightarrow\; \Diamond(\mathbf{S} \wedge \mathbf{A}) \tag{3}$$

Predicates 1 and 2 describe the behavior on executing blocking `fetch` or `meet` operations. Predicate 3 describes the selection of a template that belongs to a suspended process. "$\Box\Diamond\mathbf{M}$" means that there is infinitely often a matching tuple available for the template. **S** (selection) is implied by "$\Box\Diamond\mathbf{M}$".

Weakly fair selection of templates applies only to blocking matching: if a template that is used for non-blocking matching does match immediately then this one is excluded of further matching and the corresponding process is informed of this fact. This applies accordingly to non-blocking matching with multiple templates, too. Templates (resp. processes), which are suspended because no tuple matches them are weakly fair matched with tuples later deposited. You have to be aware that busy waiting with polling methods, which use non-blocking matching operations e.g. in loops, are **not** handled in a fair way.

10.4 Multiple Tuple Spaces

Atoms are used to identify tuple spaces ProSet provides several library functions to handle multiple tuple spaces, which are discussed in section 13.8.

Every ProSet program has its own tuple-space manager. Tuple spaces are not persistent. They exist only until all processes of an application have terminated their execution. A concurrent program terminates when all its sequential processes have terminated. Termination of the main program terminates the entire application and thus all spawned processes (see also section 7.3.2).

[11] In temporal logic "$\Diamond p$" and "$\Box p$" mean that predicate p holds eventually resp. always. See also E.A. Emerson: "Temporal and modal logic", in J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 16, pages 995–1072. Elsevier, 1990.

11 Modules

11.1 General Considerations

The introduction of persistent structures allows introducing modules, thus making separate compilation of larger program units and hence *programming in the large* feasible. Capitalizing on the simplicity of persistent structures for making modules available avoids introducing a separate mechanism for describing modules and separate compilation. That was done e.g. in Ada, where a package is told explicitly which other packages to use, and in SETL, where interactions between modules have to be described separately in a *directory.* Similarly, using packages in Ada usually requires a particular library format and a separate mechanism for binding, all outside the language itself, hence dynamic loading of packages is not possible. In addition, packages suffer from other drawbacks that are implied by this approach, e.g. they must not be circular with respect to their import/export behavior. SETL on the other hand allows circularity, since modules are linked early enough to the programs using them, but there are some other drawbacks, e.g. changing the externally visible behavior of a module requires changes in the directory (where this behavior ist posted) and thus the recompilation of the entire program.

The seemingly straightforward way of using persistent procedures as modules does not work in ProSet since the intent of a module is not fully in accordance with this approach. A module is usually thought of as a collection of routines having access to common data structures. This requires static variables, i.e. variables maintaining their value between different invocations to a routine in the module from the outside.

ProSet makes modules as templates available, using a module requires instantiating the corresponding template. Instantiation requires providing values for the imported parameters. It has the effect of

- executing the template's initialization code,
- making the exported items available,
- returning an instantiation of the template.

Modules and instantiations enjoy first class civil rights, in particular they may be made persistent. Thus we see the following correspondences between our modules and instances, and packages in Ada:

- making a module persistent corresponds to separately compiling a generic package,
- retrieving a module from a P-file corresponds to loading a generic package from a library,
- instantiating a module corresponds to instantiating a generic package and loading the corresponding unit.

Note that all this can be described in the language itself. The interconnection of modules should, however, be supported by a suitable tool which prevents interconnecting modules that do not fit. The construction of such a tool is under consideration and may influence the language constructs presented in this section.

11.2 A Simple Example

Consider the simple module `gensym` for generating symbols given in Fig. 22. The module imports a value called `symb` and exports a value called `generator`. Imports and exports are indicated by `rd` and `wr`, resp., just as formal parameters in procedure declarations; similarly, a parameter imported and exported as well is specified by `rw`. The module has a variable which is visible in all local procedures and which will be initialized to `0` as specified in the initialization part. This part contains code to be

```
module gensym(
               rd symb,
               wr generator
             );
       visible i;

       begin
            i := 0;

       procedure generator();
           persistent constant str : "StdLib";
       begin
           i := i + 1;
           return str(symb) + str(i);
       end generator;

end gensym;
```

Figure 22: A simple Module

```
MyGensym := instantiate closure gensym
                  rd symb := "g.";
                  wr generator;
            end instantiate;
```

Figure 23: Instantiating a Module

executed exactly once at the time the module is instantiated. The procedure `generator` increments `i` and returns a string which is obtained upon concatenating `symb` with `i`, both converted to strings. The module is instantiated as shown in Fig. 23 which executes the initialization code, makes the procedure `MyGensym.generator` available and sets the string `"g."` as the base to the symbols to be generated. So after the instantiation above the assignment

```
x := MyGensym.generator();
y := MyGensym.generator();
```

results in

```
x = "g.1"
y = "g.2"
```

After defining the module, we have

```
type (closure gensym)  = modtype
```

and after instantiating it, we find

```
type MyGensym  = instance
```

and furthermore `profile closure MyGensym.generator = [ ]` holds.

Hence `gensym` is bound to the code of a module, and its closure is of type `modtype`. `MyGensym` is of type `instance`; `profile` is an operator that works on modules and functions. It yields a tuple with components taken from {`rd`, `wr`, `rw`} indicating the way parameters are passed. In the example `MyGensym.h` is a procedure without parameters. The operators `profile` and `closure` are discussed in greater detail in 5.3.1.

11.3 More on Modules and Instances

Modules and instances are values with first class civil rights. Consequently they enjoy an identity, their names may occur on the left or the right hand side of an assignment, and they may be

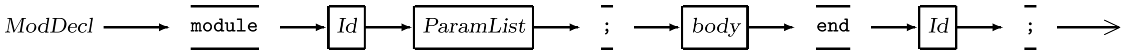
- made persistent,
- transmitted as parameters to and returned as values from procedures, other modules and exception handlers,
- inserted into compound structures like sets, maps and tuples.

Identity The identity of modules is defined in exactly the same way as for procedures, see 5.3.1. Two instances are identical iff their names are identical. Thus the following conditions hold for identical instances:

- their underlying modules, i.e. the modules from which the respective instances are obtained by instantiation are identical,
- the imported values are identical,
- the same objects are exported,
- the respective inner states, i.e. the values to the items visible throughout the underlying module, are identical.

11.4 Modules

As with other types provided by ProSet, the use of modules does not have to be declared explicitly, and checks at run time make sure that a module is used in a proper way (i.e. that parameters are passed and exported values are used according to their specification). Modules may be defined similar to procedures by prefixing an identifier with the keyword `module` followed by a parameter list, the body and a trailer:



Modules form scopes of their own, and their visibility follows the scope rules of the language. This is similar to procedures; in particular the following properties are observed:

- values to which all local procedures, modules and exception handlers have access should be declared as `visible`,
- names declared as `visible` in an enclosing scope are visible to the module, hence to all local procedures, modules and exception handlers, all other names in an enclosing scope are not accessible to the module,

- names local to the module and visible in the module's body retain their respective bindings across invocations of procedures exported from a module.

The latter property represents the characterizing difference between modules and collections of nested procedures.

The `closure` operator (cp. 5.3.1) works with modules, too. It has the effect of freezing the values of non-local names visible to the module and making them visible to the module, and it results in a value of type `modtype`. Note that freezing is in effect a hidden import operation, and that a frozen name behaves exactly like an initialized visible name, i.e. that in particular frozen variables behave in a statical way. Although freezing is very similar to the way procedures are delt with, there are differences due to the different nature of visible variables in procedures, and modules, resp.

Note that defining

```
module m(...); ...
end m;
```

does not bind the identifier `m` to a value that has a type in ProSet's type system. Only after closing it, i.e., after applying the `closure` operator to it will yield an item of type `modtype` which may be used as any other ProSet value.

A module provides provisions for executing initialization code. Syntactically, these statements follow the keyword `begin`; as in procedures, this may be the empty sequence of statements indicated by the keyword `pass`, but `begin` must not be missing. Execution is carried out exactly once each time a module is instantiated.

Visible Persistent Values in Modules Modules form scopes of their own, and dealing with them follows the scope rules of the language. We explicitly do forbid using persistent variables that are visible throughout the module. The reason for this is that persistence and the concept of static variables do not work well together. Suppose that we have the declaration

```
module MM(wr whatever);
visible persistent x:"otto";
begin
    ...
end MM;
```

and that we instantiate the closure of `MM`:

```
M := instantiate closure MM end instantiate;
```

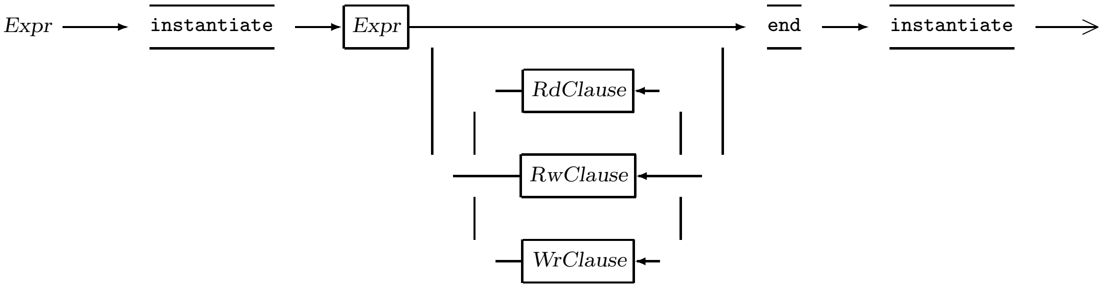
Now everything appears to be just fine: `x` will be fetched from the *P-file* `"otto"` upon execution of the initialization part, and a write lock will be granted to `x`. But now copy `M` to `D`, and suppose `M` becomes no longer accessible. By analogy to procedures one would expect that the write lock on `x` will be revoked, but that does not cater for `x` in `D`. So this is a mess, and it is not clear that we were allowed to copy `M` to `D` at all, given the write lock for `x`. Consequently, invoking a procedure or function is different from instantiating a module, static variables are different from dynamic ones, and declaring persistent variables in modules that are visible throughout the module is not allowed. Note, however, that

- variables may be declared as `visible persistent` in procedures and functions contained in a module, since here the usual mechanism applies,
- variables must not be declared as persistent on the outermost level of a module (i.e. for the declaration section),
- declaring a value as `visible persistent constant` is also not critical, since only a read lock is granted.

11.5 Instances

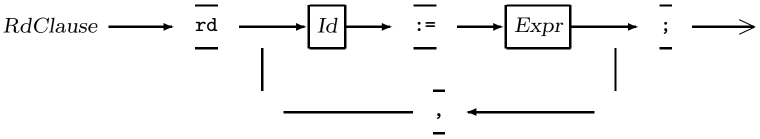
A module is instantiated by giving values to all parameters which are imported by the module, i.e. which are of mode `rd` or `rw`. The instantiation receives values for the exported parameters, i.e. for those parameters of mode `rw`, and for those parameters of mode `wr` that are specified.

Formally, the instantiation of a module is an expression that takes an argument which must evaluate to a value of type `modtype`. It has an instance as value.

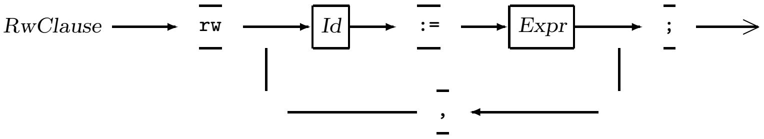


Instantiation proper consists of a sequence of the following clauses in any order:

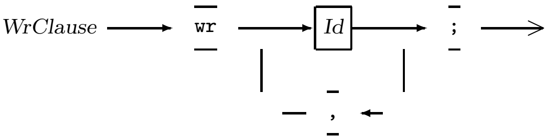
- an `rd` clause indicating the values which are being imported without being exported. Imported values may be computed in this clause.



- an `rw` clause giving the names (lvalues) which are imported and exported as well. Similarly, imported values may be computed, and the may be omitted altogether.



- an `wr` clause indicating the names (lvalues) which are being exported without being imported.



The initialization part may be executed as soon as the last `rd` or `rw` clause has been seen. Note that each identifier qualified with an `rd` or `rw` in the header of the module definition should receive a value in the corresponding clause.

At instantiation time the value of `profile` $\mu.\pi$ (μ being a substitute for the name of the instance) is determined. It is a tuple with values taken from the set {`rd`, `rw`, `wr`} for all π which are imported or exported from the module having `function` or `modtype` as their type. The i^{th} member of this tuple indicates how the i^{th} parameter to π is passed.

11.6 Of Course: Stacks

This subsection deals with the abstract data type *stack*.

We assume the exception `p_missing_name` (cp. 9.1.1) being handled in such a way that it resumes (after having generated the missing persistent value). We omit this in the example. The module `stack` manipulating this data structure is displayed in Fig. 24.

The instantiation is given in Fig. 25. This example is a bit contrived, but never mind. The following equations then hold:

```
profile InstantiateStack.push  = [rd]
profile InstantiateStack.pop  = [wr]
```

Note that `profile InstantiateStack.top` and `profile InstantiateStack.is_empty` are not defined, since they are not exported, albeit the module may export them. Export may, but import must not be selective.

This construction shows that in general not too much can be said about the values imported by or exported from a module at compile time. This is in accordance with the language's philosophy that type checking is done at run time. The interconnection of modules should, however, be supported by a suitable tool which prevents interconnecting modules that do not fit. The construction of such a tool is under consideration. These values are accessible to the module each time it is loaded. The same applies *mutatis mutandis* to functions.

Remark The mechanism for making modules available proposed here has some advantages over the one provided by Ada or Modula-2:

- defining a module, translating and putting it into a library may be done using the mechanisms of the language, once persistent structures are available,
- no separate library format with idiosyncratic access mechanisms is required; maintenance of a library of binaries follows the same rules as maintenance of other persistent structures,
- binding and linking are no longer separate phases in producing an executable program — they are integrated into the general mechanism of maintaining persistent structures.

In particular we need not leave the language level and resort to operating system commands during the entire process from the definition of a module to its use. We pay for this convenience with an added burden on the run time system, possibly slowing down the execution of large ProSet programs. This poses, however, the challenge of developing methods for statically checking as much as possible in the compiler, extending methods of type inference to modules.

```
module   stack  (
                 rd q,
                 rd bottom,
                 wr push,
                 wr pop,
                 wr top,
                 wr is_empty
                );

visible LocalStack;

begin
      q();
      LocalStack := [bottom];

procedure push(x);
begin
          LocalStack := LocalStack with x;
end push;

procedure pop(wr t);
begin
     if LocalStack = [ ] then
        escape ThisIsAnEmptyStack();
     else
        t frome LocalStack;
     end if;
end pop;

procedure top();
begin
     return LocalStack(1);
end top;

procedure is_empty();
begin
     return LocalStack = [ ];
end is_empty;

end stack;
```

Figure 24: Module Maintaining Stacks

```
This_q := closure (lambda():begin put("hello world (what else?)");
                   end lambda);

InstantiateStack := instantiate (closure stack)
                           rd q := This_q, bottom := 42; -- initialization
                                                         -- is performed
                                                         -- exactly here
                           wr push, pop;
                    end instantiate;
```

Figure 25: Instantiating the Module *stack*

12 Input/Output

This section describes the *I/O*-facilities in ProSet. Besides standard routines for handling files and simple input and output routine we have integrated a comfortable format description sublanguage for specifying input and output formats which will be illustrated below.

12.1 Files

Files are denoted in programs by their name, adopting the conventions from the underlying operating system. These names are strings, probably including a path. A file will be opened using `fopen`. This procedure requires two parameters: the file name and the mode in which the file should be opened. These modes are:

- `"r"`: the file will be opened for reading, the file pointer is set to the beginning of file,
- `"w"`: the file will be opened for writing, the previous content is lost,
- `"a"`: the file will be opened for writing, it will be appended at the end, the previous content is preserved.

`fclose` is used for closing files. An unopened file can not be closed. A file must be closed or not yet opened when it is opened. There is no access to an unopened file. The number of simultaneously opened files is finite and system dependent. If any of these conditions are violated, appropriate exceptions are raised (see section 12.4).

For testing on end-of-file two standard functions `eof` and `feof` are provided. `feof` takes a file name as an argument, `eof` implicitly assumes the *standard input*. They both return a boolean value indicating whether or not the end of the given file is reached.

The `stop`-instruction automatically closes all still opened files (for further informations about `stop` see section 7.3.2).

12.2 Formatted Output

The goal of formatted output is the type dependent conversion of values from the internal to an external representation. With a single call to an output routine an arbitrary number of values can be delt with. If the user wants to deviate from standards, he should place a string (called **format specification** below) in front of the argument list in the output routine call describing his own format in preceding argument list in the output routine call. The user can choose between:

- formatted output based on *defaults* or on *user specified* formats.
- output on the *standard output* or into a *specified file*.

Furthermore, the predefined *standard error output* can be used based on both default and user specified formats, respectively.

There exists a function for each combination:

function	output to	format specification
put(arg_1,...,arg_n)	*standard output*	default format
eput(arg_1,...,arg_n)	*standard error*	default format
fput(file,arg_1,...,arg_n)	specified file	default format
putf(format,arg_1,...,arg_n)	*standard output*	user format
eputf(format,arg_1,...,arg_n)	*standard error*	user format
fputf(file,format,arg_1,...,arg_n)	specified file	user format

The functions have the following parameter:

- **file**: a string containing the name of a file opened with mode "w" or "a",
- **format**: a string containing the format specification,
- **arg_1,...,arg_n**: a list of an arbitrary number of arguments (may be empty).

Some simple examples are as follows. All examples presented in the remainder of this section assume files opened in an appropriate mode.

put("Hello world"); puts "Hello world" on the standard output (in general the terminal).
fput("my_file","Hello world"); uses the file my_file instead of the standard output.

```
-- Initialization
var_atom    := newat();
var_bool    := true;
var_integer := 42;
var_real    := 0.045;
var_set     := {123,[4,5]};
var_string  := "An example";
var_tuple   := ["abc",{1,2}];
-- Output to standard output
put(var_atom,var_bool,var_integer,var_real,var_set,
    var_string,var_tuple);
```

yields

```
...[12]  ,#true,42,4.5E-2,{123,[4,5]},"An example",["abc",{1,2}]
```

Here, the defaults are used to put out a value of each supported type (maps are treated as sets).

In the next example, we choose some of those values above and show the result according to a special format specification.

```
our_format := "Integer:%-5d, Set:%{content:$$!5‘*‘d}, String:%s";
putf(our_format,var_integer,var_set,var_string);
```

yields

```
Integer:  42, Set:{content:*123*,[**4**,**5**]}, String:An example
```

Format Specifications The actual format specification in an output routine call describes the constant part, the values of the various arguments describe the variable part of the generated output.

There are **formatters** for each type, i.e. devices for referencing an argument of a particular type. Defaults are defined by the system. They can be used or modified by the user. Modifications in a declarative form or modifications used for a specific reference of a value, i.e. an argument, are possible. Declarations modify the defaults in a scope. They are initiated by a $. Declarations can be used to modify:

- type-specific formats,
- special settings for compound values,
- width of lines and tabulators.

[12] The output generated for atoms depends on the implementation.

By using compound values a block structured output specification is generated. The content of a tuple- or set-formatter forms a new block. Declarations are bound to these blocks. Declarations may have:

- visibility in the actual block only or
- visibility in all subordinate blocks.

The arguments are referred to by using formatters in the format specifications. They are initiated by a %. The external representation of a value is constructed according to the specification given to the formatter and the additional actually valid declarations. The arguments are printed in the order of the references in the format specification, i.e. the n-th item in the format specification refers to the n-th argument in the argument list (but the user may alter this order).

All informations which have to be interpreted must be initiated by one of the symbols `%, $, \` . All other characters will be printed directly. Brackets for tuples or sets which have to be printed, must be given in an *escape form* (e.g. `\{` or `\]`). The same form is to be used to put out the control characters `", $, %, \` (e.g. `\$` or `\\`). The control instructions *backspace, tab, new line, form feed, carriage return* are denoted by `\b, \t, \n, \f` and `\r`, respectively. `\e` is used for the empty word ϵ.

User formatted output does not insert blanks and does not execute *carriage returns* automatically.

Only symbols in the format specification are interpreted, further arguments of type *string* are printed directly.

Default- and User-specifications

Calls to `put`,`eput` and `fput` start their output in new lines (i.e. they carry out a *carriage return*). Single independent arguments are separated by default through blanks. Elements in tuples or sets are separated by commas to emphasize their belonging to one compound value.

```
put("abc",3,[1,2]);            yields            "abc" 3 [1,2]
```

`put` uses the default formats, thus the arguments `"abc", 3` and `[1,2]` are separated through blanks; the elements of the tuple are separated by a comma.

Implicitly `put` uses (in a general form)

```
putf("%x %x ...%x\n",arg_1,...,arg_n);
```

`%x` is the standard formatter for a value. Applied to compound valued arguments `%x` yields

- `[elem_1, ...,elem_n]` for tuples and
- `{elem_1, ...,elem_n}` for sets.

If the non-declarative information, i.e. a constant character or a formatter appears in a compound value formatter, it is interpreted by system as a request to use user formatted output for the content. Here, commas will be suppressed in compound values.

Formatter

A formatter describes the external form to which the corresponding argument should be transformed. The transformation is based on the argument type. The following types are available:

`atom`	only printable by using the standard formatter `%x`
`boolean`	supported by the type descriptor **b**
`function`	not supported
`instance`	not supported
`integer`	supported by the type descriptor **d**
`modtype`	not supported
`real`	supported by the type descriptors **e,f**; **e** is the default; **e** and **f** identify the fixed-point representation and the floating-point representation, respectively.
`set`	supported by the type descriptor {}
`string`	supported by the type descriptor **s**
`tuple`	supported by the type descriptor [ ]

Not supported argument types are causing the exception `type_mismatch` to be raised.

The structure of a formatter for a value (angles are enclosing optional informations, the bar separates alternatives):

`%` ⟨ *i*`(` ⟩ ⟨ `-` | `!` ⟩ ⟨ *m* ⟨'*c*'⟩ ⟨`.`*p* ⟩ ⟩ *format_type* | *positioner* ⟨ `)` ⟩

i Iterator, indicates how often a value of the specified type should be printed.

`-` Left-aligned output, default is right-aligned.

`!` Centered: if the number of characters is even, the output will be centered around the immediately right position from the original center.

m Minimal width for *format_type* = b, d, e, f, s, x, [ ], { }; number of positions for the *format_types* < and >. *m* includes signs, decimal points, and the character `e` that starts the power. Default is 0.

p Maximal width righ-hand side of the point for *format_type* = e,f; default is machine-precision, zeroes at the end are cutted.

c Arbitrary character used for filling up to the width *m*.

`putf("%5‘*‘d",3);` yields `****3`

format type Descriptor for the type of the formatter. In generell, the conversion from internal to external representation of an argument value is described.

positioner An implicit pointer can be positioned in the argument list.

Description of Some Format Convertions

Some of those format types converting an internal representation to an external representation are discussed now.

Real Numbers

For printing real numbers ProSet provides two alternatives format types. **e** identifies the fixed-point representation $(-)n.ppppE \pm eee$, the power has a fixed maximal length of 3, **f** identifies the floating-point representation $(-)nnnn.pppp$. Some examples are presented now:

`putf("%5.4f",1.23456);`	yields	`1.2345`
`putf("%10.2e",1.23456);`	yields	`1.23E+0`
`putf("%10.10e",12345.6);`	yields	`1.23456E+4`

If there is a mismatch between *m* and *p*, the precision *p* has higher priority than the field width *m*.

Strings

`%x` applied for strings prints them in quotes, `%s` omits the quotes.

Compound Values

For compound values the corresponding brackets (`[ ]` and { } for tuples and sets, respectively) are used to describe the format type. The internal structure may be specified inside the brackets.

/abc/`[ ... ]` or */abc/*`{ ... }` for tuples or sets (respectively)

/abc/ is optional, used to substitute the respective opening bracket, the symbol used to separate elements of compound values and the closing bracket. It may appear immediately in front of the *open_bracket*. Instead of *a*, *b* and *c* replacement characters for *open_bracket*, *separation_symbol* and *close_bracket* can be declared (*a*,*b*,*c* can be empty, ϵ must be used in this case).

`putf("%/⟨ | ⟩/[] %/⟨ | ⟩/{}", [1,2,3],{4,5,6})` yields ⟨1|2|3⟩ ⟨4|5|6⟩

In the latter example for tuples and sets the same *open_bracket*, *separation_symbol* and *close_bracket* is used.

Square brackets are used for tuples, braces for sets. These brackets are obligatory. This necessity is illustrated by the following example:

```
putf("%[%[$-2d]%{$3d}]", [ [1,2], {3,4}, [5,6] ]);
```

yields

```
[[1 ,2 ]{  3,  4}[5 ,6 ]]
```

In the example above, integer in tuples are printed in two character width, left-aligned (using `%[$-2d]`). In sets they are printed in three character width and right-aligned (using `%{$3d}`). Those two formatters (`%[$-2d]` and `%{$3d}`) are valid for every tuple and set, resp., in the argument tuple `[ [1,2], {3,4}, [5,6] ]`.

Thus the types *tuple* and *set* are treated differently.

Sets
A format can be specified for every possibly appearing element type only using declarations. Direct references (i.e. formatters) cannot be treated because there is no possibility for unambiguous relations between a formatter and a set element. Strings in the format specification are printed at the beginning of the set. This is valid for all characters that are not control characters and that are inside the brackets.

```
putf("%40‘*‘{$-5s$3d$[$-2d]set:}",{{1,2,3},42, "abc",[1,2]});
```

yields

```
******{set:{1,2,3} 42 abc  [1 ,2 ]}
```

The output has a width of 40 filled with asterisks at the left-hand side (`%40‘*‘{...}`). Only one set is put out. In this set

- strings are printed with a width of 5 and left-aligned (`$-5s`),
- integers are printed with a width of 3 and right-aligned (`$3d`), here the declaration is only valid for outermost level in the set[13],
- integers in tuples are printed with a width of 2 and left-aligned (`$[$-2d]`).

Tuples
The treatment of sets can be applied here. Alternatively, a complete format specification can be done. The content of a tuple formatter is treated as a local argument list.

```
putf("%[first element:%3d, second element:%x]",[1,2]);
```

yields

```
[first element:  1, second element:2]
```

[13] Later we see how to obtain *visible* declarations

Positioning in Argument Lists

Using the positioner `%m>` in forward direction the next m arguments are skipped, in the other direction an implicit argument pointer is set backwards m arguments.

```
putf("%!4‘*‘d,%<%-4‘-‘d,%<%4‘+‘d",7);
```

yields

```
**7* 7--- +++7
```

There is only one argument (the integer value 7) which is used three times by setting backwards two times (using `%<`).

```
if one then
     one_or_two = "%x%>";
else
     one_or_two = "%>%x";
end;
putf(one_or_two,"abc","def");
```

yields

if (`one = true`) : `abc`
and otherwise : `def`

This example shows the use of this construct for the selection of an appropriate format depending on some condition to obtain a value-dependent output.

Using these positioners an implicit pointer to the argument list can be manipulated. This pointer can be set on every arbitrary position, even outside the list. Only if such a referenced but not existing argument is to be put out, an exception is raised (see section 12.4).

Declarations

Declarations address the output of compound values and the general style of output, e.g. the width of lines, tabulators). They are initiated by **$**, brackets are enclosing optional informations. They can appear at every position in a format specification. We discuss the different possibilities now in turn.

$[‘*s*‘]n D: Depth to which levels of compound values are printed (a ‘**&**‘ marks more levels, string *s* may replace ‘**&**‘). Default: no restriction.

```
putf("$‘?‘2D %x",{1,{2,{3,4},5},6,7});          yields          {1,{2,?,5},6,7}
```

$[‘*s*‘]n N: Number of elements of a compound value that should be printed at one level. Further existing elements are indicated by `"..."` (string *s* can replace the dots). Default: no restriction.

```
putf("$2N $2D %x",{1,{2,{3,4},5},6,7});        yields        {1,{2,&,...},...}
```

$n,m T: Setting of tabulators at the positions $n + k * m,\ k \in \mathbb{N}_0, m, n \in \mathbb{N}$. n indicates the first column which should be used in the actual line, m decribes the width of an tabulator column. After printing an element the output pointer is set to the next free column $n + k * m$ (with minimal k, $k \in \mathbb{N}$). Previously, at least one blank is inserted. No *carriage returns* are executed.

The next tabulator position is reached using `\t`. An *end of line* is defined independently. Default: $n = 1, m = 1$.

```
header  := "$5,10T  \t| col_1  \t| col_2  \t| col_3  \t| \n";
content := "$6,10T %-s |\t %8x\t %8x\t %8x\n";
line    := "%40(s)\n";
```

```
putf(header);
putf(line,"-");
putf(content,"1:",t_1_1,t_1_2,t_1_3);
putf(content,"2:",t_2_1,t_2_2,t_2_3);
putf(content,"3:",t_3_1,t_3_2,t_3_3);
```

yields

```
    | col_1   | col_2   | col_3   |
---------------------------------------
 1: | t_1_1     t_1_2     t_1_3
 2: | t_2_1     t_2_2     t_2_3
 3: | t_3_1     t_3_2     t_3_3
```

The `t_i_j` (i,j = 1,..,3) are arbitrary values printed with a minimal width of 8. The width of each tabulator column is 10. The `$n,mT` declaration provides at least one blank inserted between two arguments and each `t_i_j` is prefixed by a blank. Therefore, the maximal width of one argument should be 8.

Multiple tabulator declarations in one scope make sense for designing tables with irregular width of columns.

$n W: Width of line. The maximal number of characters per line is given. If necessary, a *carriage return* is executed. Compound values and strings may be broken, simple values not.

```
putf("$10W abcdefghijklmnopqrstuv$3Wwxyz");
```

yields

```
abcdefghij
klmnopqrst
uvw
xyz
```

The line width is set to 10. Thus, after putting out 10 characters a carriage return is executed, even if an argument is not completely put out. The line width can be changed within a line.

$[$]<Formatter>: Formatters initiated by `$` overwrite the default specification. Initiated by two `$`, the declaration is valid for all subordinate level (e.g. `$$-3d`). If there is only one initial `$`, the declaration is valid only at the actual level, i.e. not in compound arguments of the actual level. In those compound value formatters, the defaults are valid.

```
form_spec := "$$-3‘-‘d%[$3‘+‘d]%d";
putf(form_spec,[1,2,[3,4]],5);
```

yields

```
++1 ++2 3-- 4-- 5--
```

Modifying the positioners ‘<‘ and ‘>‘ does not make sense. Iterators are not allowed. Multiple formatter declarations of one argument in a scope are ambiguous and therefore result in raising an exception.

$‘s‘ OM: Defines a string ‘s‘ to replace the standard output for the undefined value `om`. Default is `#?`.

`putf("%x%[$‘ThisOm‘OM]", om,[om]);` yields `#? [ThisOm]`

$‘s‘ TRUE: Defines a string ‘s‘ to replace the standard output for the boolean value `true`. Default is `#true`.

$‘s‘ FALSE: Defines a string ‘s‘ to replace the standard output for the boolean value `false`. Default is `#false`.

D, **N**, **T**, **W**, **OM**, **TRUE** and **FALSE** are declarations, which are by default valid in all subordinate levels, but also modifiable at every point in those levels (with the corresponding visibility). Declarations are local to the actual level.

Upper and lower case letters are distinguished, for otherwise there would be conflicts (see `D` (Depth) and `d` (decimal)).

12.3 Formatted Input

Input deals with the conversion of values from an external to an internal representation. The user can choose between:

- formatted input based on *defaults* or on *user specified* formats.
- input from the *standard input* or from a *specified file*.

There exists a function for every possible combination:

function	input from	format specification
get(arg_1,...,arg_n)	*standard input*	default format
fget(file,arg_1,...,arg_n)	specified file	default format
getf(format,arg_1,...,arg_n)	*standard input*	user format
fgetf(file,format,arg_1,...,arg_n)	specified file	user format

The functions have the following parameter:

- **file**: a string containing the name of a file opened with mode `"r"` .
- **format**: a string containing the format specification
- **arg_1,...,arg_n**: a list of an arbitrary number of arguments (may be empty).

Some introducing examples are as follows.

`get(s);` assigns on input `"Hello world"` from the standard input the value `"Hello world"` of type *string* to `s`.
On input `Hello world` (without quotes) only `"Hello"` is assigned to `s`.
`fget("my_file",s);` reads a string from the file `my_file` and assigns it to `s`.

`get(var_bool,var_integer,var_real,var_set,var_string,var_tuple);` assigns on input `#true 42 4.5E-2 {123,[4,5]} "An example" ["abc",{1,2}]` as follows:

```
var_bool    := true;
var_integer := 42;
var_real    := 0.045;
var_set     := {123,[4,5]};
var_string  := "An example";
var_tuple   := ["abc",{1,2}];
```

The means for specifying formats are similar to those of output. Thus, there are declarative specifiers and argument referencing ones.

We have default formats for every type. The defaults can be used or modified by the user. The arguments are referred to by formatters in the format specification. The arguments are usually given according to the expected order of input, but the user may alter this order.

All informations which are to be interpreted should be prefixed by one the symbols `%`,`$`, \\. All other characters should be read from input without assigning them to any argument.

`getf("Hello world");` reads `"Hello world"` but does not assign anything. If the strings does not match the exception `io_illegal_input` will be raised.

The values to be read must be separated through *white space* (see section 3.2), in compound values commas are also valid. By default, i.e. using the standard formatter `%x`, only those values are recognized as belonging to a particular type that are producable by the standard output formatter. Strings without quotes are only valid, if they are valid ProSet identifiers.

Formatter

A formatter describes the internal form in which a specific argument should be transformed. The transformation is based on the argument type. The following types are available:

`atom`	not supported
`boolean`	supported by the type descriptor **b**
`function`	not supported
`instance`	not supported
`integer`	supported by the type descriptor **d**
`modtype`	not supported
`real`	supported by the type descriptor **e,f**
`set`	supported by the type descriptor {}
`string`	supported by the type descriptor **s**
`tuple`	supported by the type descriptor [ ]

If an argument has a not supported type the exception `type_mismatch` is raised.

The structure of a formatter for a value (angles are enclosing optional informations, the bar separates alternatives):

$$\%\ \langle\ i(\ \rangle\ \langle\ m\ \rangle\ \mathit{format\_type} \mid \mathit{positioner}\ \langle\ )\ \rangle$$

i Iterator, indicates how often a value of the specified type should be read.
`getf("%[%5(d)]",tup);` assigns on input `1 2 3 4 5 6 7` to the variable `tup` the value `[1,2,3,4,5]`. So, exactly 5 integer values are assigned to `tup`.

m maximal width for *format_type* = b, d, e, f, [ ], { }; exact width for *format_type* = s; number of positions for the *format_types* < and >.

format type descriptor for the type of the formatter. In generell, the conversion from external to internal representation of an argument value is described. Furthermore, a whole line can be assigned and characters in input can be skipped.

positioner An implicit pointer to the argument list can be positioned.

Description of Some Format Convertions

Strings

`%s` reads strings. Strings to be read must be enclosed by quotes, with the exception that valid ProSet identifiers can be recognized even without quotes. If a string width m is given in the formatter (e.g. `%10s`, the width is 10), exactly m characters are read (even blanks). If an end-of-file mark is reached while expecting some more arguments, an exception is raised.

Lines

`%l` reads a line (from the actual position to the end of line).

`fgetf("my_file","\n%l%s",line,string);` assigns from the file `my_file` with content

```
abcdefghij
klmnopqrst
uvwxyz
```

to `line` the value `"klmnopqrs"` and to `string` the value `"uvwxyz"` .

Real
Both external representations of real numbers (*format_types* **e** and **f**) have the same internal representation (see section 12.2), so they are used synonymously.

Compound Values
For compound values the corresponding brackets are used to describe the format type. The internal structure may be specified inside the brackets. Square brackets are used for tuples, braces for sets. They are obligatory. In contrast to output, formatters in sets make sense here. Using `%{%n*x}` it is possible to read n elements of arbitrary type into a set.

Thus `getf("%{%3*x %[%4*x] %2*x}",input);` assigns on input `1 2 abc 3 4 5 6 [1,2] 7` to `input` the value `{1,2,"abc",[3,4,5,6],[1,2],7}`

The next two examples illustrate the use of declarations for limiting the types which can be read. Only those declared inside the brackets are valid.

`getf("%[$d$b$s]",input);` assigns on input `1 abc #true <EOF>`[14] to the variable `input` the value `[1,"abc",true]`

`getf("%[$d$b]",input);` and raises an exception on input `1 abc #true` (`"abc"` corresponds to none of the given types `d` and `b`).

Miscellaneous
`%?` The input pointer skips the next character (*white space* is not skipped, see section 3.2).

`%*` The input pointer skips forward to the next *white space* (excluding it, i.e. the input pointer is set to the *white space*).

`getf("a%?c 1%* %s",string);` assigns on every input of the form `a.c 1.. s` to the variable `s` a string (where `.` represents an arbitrary non-*white space* character and `..` represents an arbitrary number of non-*white space* characters and `s` is a valid string).
Examples: `abc 123 abc` or `a!c 1abc "xyz"`.

Positioning in Argument Lists

In forward direction (`%m`>) the next m arguments are skipped (no assignment is executed), in backward direction an implicit argument pointer is set back m arguments (it may be assigned a number of times).

`getf("%d %< %d %< %d",x);` assigns on input `1 2 3` to `x` sequentially the values `1, 2` and `3`. So, after executing the statement `x` has value `3`.

`getf(" %d %> %d ",a,b,c);` assigns on input `1 2` to `a, c` the values `1` and `2`, respectively.

Using these positioners an implicit pointer to the argument list can be manipulated. This pointer can be set to an arbitrary position, even outside the list. Note, however, that an exception will be raised if an argument is to be read but the input pointer points to a position outside the argument list.

Declarations
Declarations address the output of compound values and the general style of output (e.g. width of lines, tabulators). They can appear at every position in a format specification, brackets are enclosing optional informations.

$n,m T: Setting of tabulators at the positions $n + k * m,\ k \in \mathbb{N}_0, m, n \in \mathbb{N}$. A right-hand side boarder can be obtained by declaring the line width. In opposition to output, here is no need for separating arguments by blanks. The next tabulator position is reached using `\t`. Default: $n = 1, m = 1$.

`form_spec := "$5,4T %[\t%1d\t%2d\n\t%3d\t%4d]";`
`fgetf("my_file",form_spec,tuple);` assigns from the input file `my_file` with content

[14] The end-of-file is operating system dependent.

```
    12345678901234567890
    12345678901234567890
```

to `tuple` the value `[5,90,567,9012]`.

Using multiple tabulator declarations even irregular input structures can be read.

$n W: The maximal number of characters per line which should be read.

$[$]<Formatter>: Formatters initiated by `$` overwrite the default specification. Initiated by two `$`, the declaration is valid for all subordinate levels (e.g. `$$-3d`). If there is only one initial `$`, the declaration is valid only at the actual level, i.e. not in compound arguments of the actual level. In those compound value formatters, the defaults are valid.

Modifying the positioners does not make sense. Iterators are not allowed. Multiple formatter declarations of one argument in one level are ambiguous and therefore result in raising an exception.

$‘s‘P: A prompt which is used for input from the terminal. If declared on the outermost level, the string ‘*s*‘ is used as main prompt. The prompt appears in front of every element on the level on which it is declared.

`getf("%{$‘next element:‘P$d}",my_set);` works as follows (on the terminal, > is the standard prompt):

```
> 1
next element: 2
next element: 3
next element: 4
next element: <EOF>
```

assigns to `my_set` the value {`1,2,3,4`}.

$‘s‘E: End identificaton for compound objects. Default is the closing bracket.

A problem may arise when the default specification for reading set or tuples is used, viz., when the number of elements which have to be read is unknown. We give the possibility of declaring a pattern which indicates the end of input, e.g. `getf("$‘end‘E %{}%{}",set1,set2);` assigns on input `1 2 3 end 4 5 6 end` to `set1` and `set2` the values {`1,2,3`} and {`4,5,6`}, respectively. If `end` is read in input the actual compound value (if there is one) is completely read.

$‘s‘ OM: Defines a string that indicates the undefined value `om`. Default is `#?`.

$‘s‘ TRUE: Defines a string that indicates the boolean value `true`. Default is `#true`.

$‘s‘ FALSE: Defines a string that indicates the boolean value `false`. Default is `#false`.

T, **W**, **P**, **E**, **OM**, **TRUE** and **FALSE** are declarations, which are by default valid in all subordinated levels, but also modifiable at every point in those levels (with the corresponding visibility).

12.4 Exceptions

In this section we enumerate all exceptions that can be raised by I/O-routines. Exceptions are described in section 8.

Using the I/O-routines exceptions are raised in the following situations:

`io_file_error` is `escaped` if the file does not exist or there is no access to it. `io_syntax_error` is `signalled` if a syntactic error in the format string occurs. `Resuming` retries with the default specification. `io_type_mismatch` is `signalled` if there is a type mismatch between the formatter and an argument. Resuming retries with the default specification. io_arguments_missing is escaped if there is no existing argument to a formatter reference. io_illegal_input is escaped, if there is a mismatch between a string to be read and the input.

All exceptions described above are predefined.

An example illustrates the handling of exceptions raised by an I/O-routine.

```
-- ..
form_spec:= "Centered integer $$!3d *%d*%d*%d*\n";
putf(form_spec,1,[2,3],4) when io_type_mismatch use output_handler;
-- ..

handler output_handler();
begin
     resume;
end output_handler;
```

yields

```
1,[2,3],4
```

The exception io_type_mismatch is raised, because the expected type of the second argument is *integer* whereas a *tuple* occurs. Associated with the handler output_handler the execution of putf continues (i.e. putf is resumed). But instead of %!3d the defaults are used, i.e. integers are printed right-aligned with their width 1, whereas something like * 1 * 2 * 3 * 4 * was probably intended by the programmer.

13 Standard Library

We have put some operations considered standard in other languages into a *Standard Library*. These operations manipulate standard ProSet objects. Consequently, the extent of the ProSet kernel decreases and the user is also allowed to redefine predefined operations.

We discuss in this section the contents of the standard library indicating for each operation the intended functionality.

13.1 Preliminaries

The standard library is accessible through ProSet's persistent mechanism (see sections 9.1 and 4.2.1). Each function described below is made accessible by declaring it as

```
               visible persistent constant <function name>:"StdLib" .
```

ProSet facilitates this by making predefined macros available.

```
macro ImportStdLib;
   ImportIntOps; ImportRealOps; ImportStringOps; ImportSetOps; ImportsFurtherOps;
endm ImportStdLib;
```

ImportIntOps, ImportRealOps, ImportStringOps, ImportSetOps and ImportsFurtherOps are macros too.
They all consist of declarations persistent visible constant <function name>:"StdLib";. The function names are those described in the following subsections according to their type specific classification.

All functions described below raise the exception type_mismatch (see section 8.7 for details) if an argument has an inappropriate type.

13.2 Functions on Integers

Functions on integer values:

`abs(i)`	absolute value of an integer argument `i`
`even(i)`	predicate: yields `true`, if an integer argument is even, `false` otherwise
`odd(i)`	predicate: yields `true`, if an integer argument is odd, `false` otherwise
`float(i)`	converts an integer to the corresponding real value
`sign(i)`	yields -1, 0, +1 depending on whether the argument is negative, zero and positive, resp.

13.3 Functions on Reals

Functions on real values:

`acos(x)`	arc cosine
`asin(x)`	arc sine
`atan(x)`	arc tangent
`atan2(x,y)`	arc tangent: $atan2(x, y) = atan(x/y), y \neq 0$
`abs(x)`	absolute value of a real argument
`ceil(x)`	returns the smallest integer which is at least as large as `x`, ceiling: $x \mapsto \lceil x \rceil$
`cos(x)`	cosine
`exp(x)`	e^x
`fix(x)`	converts `x` into the corresponding integer number by dropping the fractional part, $x \mapsto$ `if` $x \geq 0$ `then` $floor(x)$ `else` $ceil(x)$ `end if`
`floor(x)`	returns the largest integer which is not larger than `x`, floor: $x \mapsto \lfloor x \rfloor$
`log(x)`	natural logarithm
`sign(x)`	yields -1, 0, +1 depending on whether the argument is negative, zero and positive, resp.
`sin(x)`	sine
`sqrt(x)`	square root
`tan(x)`	tangent
`tanh(x)`	hyperbolic tangent

13.4 String Scanning Primitives

We discuss here the *string scanning primitives*, that we took from SETL (and therefore from SNOBOL[15]).

Let `s` and `ss` be strings.

`any`: `any(rw s,ss)` yields `s(1)`, if `s(1)` is in `ss`, and then modifies `s` to `s(2 ..)`. If otherwise `s(1)` is not in `ss`, `s` remains unchanged and `om` will be returned.

`break`: `break(rw s,ss)` breaks from `s` the longest initial part that does not include characters from `ss`. This part will be returned, `s` will be modified. Otherwise, if `s(1)` is contained in `ss`, `om` will be returned, `s` remains unchanged.

`len`: `len(rw s,n)`, `n` $\in \mathbb{Z}$ yields `s(1 .. n)` and modifies `s` to `s(n+1 ..)`. If `n` ≤ 0 or `#s` $\leq$ `n` then `s` remains unchanged and `om` will be returned.

`lpad`: `lpad(rw s,n)`, `n` $\in \mathbb{Z}$, returns the string `s` filled with additional blanks at the left-hand side so that a string with length `n` is constructed. If `n` $\leq$ `#s`, then `s` will be returned unchanged.

[15] R. E. Griswold, J. F. Poage and I. P. Polonsky: The SNOBOL4 Programming Language (second edition), Prentice-Hall, Englewood Cliffs, NJ, 1971

`match`: `match(rw s,ss)` returns `ss`, if #ss ≤ #s and `ss = s(1 .. #ss)`. In this case, `s` will be changed to `s(#ss+1 ..)`. Otherwise, `s` remains unchanged, `om` will be returned.

`notany`: `notany(rw s,ss)` yields `s(1)` and modifies `s` to `s(2 ..)` if `s(1)` does not occur in `ss`. If `s(1)` does occur in `ss`, `s` remains unchanged, `om` will be returned.

`span`: `span(rw s,ss)` yields the longest initial part of `s` whose characters all occur in `ss`. This part will be broken off `s`. If such an initial part is not found, i.e. `s(1)` is not in `ss`, `s` remains unchanged, and `om` will be returned.

These functions operate on their argument `s` from left to right, e.g. `any` looks at at leftmost character in `s`, `match` tries to match `ss` and the left-hand side of `s`. For each of them exists a *right-to-left* variant, e.g. `rpad` adds blanks at the right-hand side, `rspan` starts its search for characters from `ss` at the right-hand side of `s`.

`rany`	right-to-left variant of `any`
`rbreak`	right-to-left variant of `break`
`rlen`	right-to-left variant of `len`
`rpad`	right-to-left variant of `pad`
`rmatch`	right-to-left variant of `match`
`rnotany`	right-to-left variant of `notany`
`rspan`	right-to-left variant of `span`

Let us display some examples illustrating the above descriptions.
For this let `ss` be `"abcdefghijklmnopqrstuvwxyz"`.

- If `s` is `"an example"` then `any(s,ss)` yields `"a"` and modifies `s` to `"n example"`.
- If `s` is `"123a456"` then `break(s,ss)` yields `"123"` and modifies `s` to `"a456"`.
- `len(ss,15)` yields `"abcdefghijklmno"` and modifies `ss` to `"pqrstuvwxyz"`.
- If `s` is `"abc"` then `lpad(s,5)` modifies `s` to `"  abc"` and yields `s` as result.
- `match(ss,"abcdef")` yields `"abcdef"` and modifies `ss` to `"ghijklmnopqrstuvwxyz"`.

13.5 Functions on Sets

The most functions on sets are integrated in the language.

`npow`	`k !npow A`: the set of all subsets of `A` with exactly `k` elements

13.6 File Handling

The I/O-functions can be divided in two parts. First the input and output functions `get` and `put` and their variants are integrated in the language and, therefore, are not mentioned here. The second part comprises the file handling facilities. These are `fopen`, `fclose`, `eof` and `feof`. All these functions are discussed in detail in section 12.

`fopen(fn)` tries to open a file specified by the file name `fn`.

`fclose(fn)` tries to close the file `fn`.

`eof()` tests on *end of file* in *standard input*.

`feof(fn)` tests on *end of file* in the file `fn`.

13.7 Further Predefined Functions

`abs(s)` takes a one-character string and return an internal integer code for this character. If #s≠ 1 an exception is raised.

`char(i)` the converse function to `abs`. It takes as its argument an integer value which has to be the internal code for a character and returns the corresponding one-character string. An exception is raised, if for all one-character strings `s` the argument `i` is not equal to `abs(s)`.

`str(x)` converts an arbitrary ProSet object into the corresponding printable string. Note that `x` must not be an argument of the `function, module` or `instance`.

`system(x)` takes one argument. If the argument is `om` it will be tested whether a command processor exists. If the argument is a string, this string is passed to the command processor of the operating system. For details see section 4.2.4.

Note that the values returned by `char` and `abs` depend on the implementation.

13.8 Handling Tuple Spaces

ProSet provides several library functions to handle multiple tuple spaces:

`CreateTS(limit):` Calls the standard function `newat` to return a fresh atom. The tuple-space manager is informed to create a new tuple space represented/identified by this atom. The atom will be returned by `CreateTS`. Thus you can only use atoms that were created by `createTS` to identify tuple spaces.

Since one has exclusive access to a fresh assigned tuple-space identity, `CreateTS` provides information hiding to tuple-space communication.

The integer parameter `limit` specifies a limit on the expected or desired size of the new tuple space. This size limit denotes the total number of passive and active tuples, which are allowed in a tuple space at the same time. `CreateTS(om)` would instead indicate that the expected or wanted size is unlimited regarding user-defined limits, not regarding physical limits.

`ExistsTS(TS):` Provides `true`, if `TS` is an atom that identifies an existing tuple space; else `false`.

`ClearTS(TS):` Removes all active and passive tuples from the specified tuple space. This operation should be indivisible for `TS`.

`RemoveTS(TS):` Calls `ClearTS(TS)` and removes `TS` from the list of existing tuple spaces. Note that after assigning a new value to a variable that contains a tuple-space identity, the respective tuple space is **not** removed. The tuple-space identity is garbage collected, not the tuple space itself. Garbage collection applies only to first class objects.

If these functions are invoked with actual parameters that are not atoms, then the exception `type_mismatch` will be raised. If the functions `ExistsTS`, `ClearTS`, or `RemoveTS` are called with an atom, which is not a valid tuple-space identity, then the exception `ts_invalid_id` will be raised.

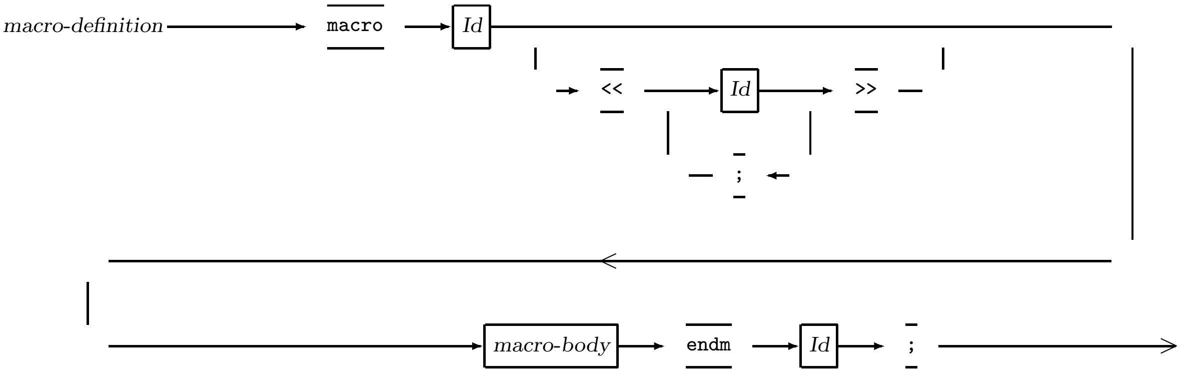
14 Macros

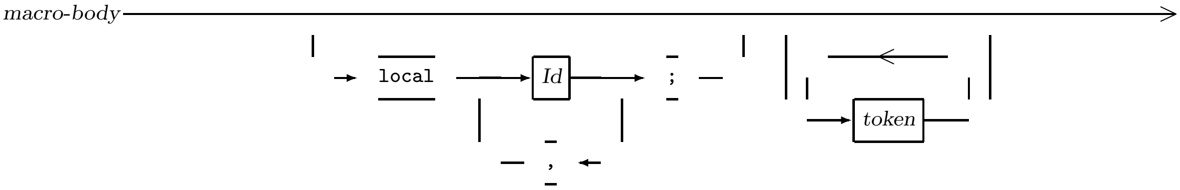
ProSet allows to define parametrized lexical abbreviations for sequences of tokens. These abbreviations are expanded by the macro processor before the lexical analysis phase of the compiler sees the program. The macro processor preserves comments but does not interpret their content.

Although many language designers regard macros as an ancient relict or as a remainder of assembler programming, we consider them as a convenient improvement of readability allowing the abbreviation of lengthy code sequences. Certainly this functionality could be achieved by procedures, but at the cost of inefficiency arising from unnecessary copies and procedure invocations.

14.1 Macro Definition

A macro definition introduces a new macro which is referred to by the identifier following the keyword `macro`. The definition may contain a declaration of local variables which are separated by commas and terminated by a semicolon. It is syntactically described by





The identifier following `endm` must be the macro name. The `local` allows to introduce fresh names for variables, e.g. to hold temporary values or intermediate results. We assert that for each macro invocation a fresh variable is generated.

The idea of having no dependencies on the syntax of the source language led to a somewhat unusual notation of arguments: argument lists are enclosed by double angle quotes (i.e. less- and greater-symbols), arguments are separated by semicolons. The syntactical restriction posed on the actual parameters of a macro application are minimal: a semicolon and `>>` are not allowed. Anything else (any token, including unbalanced parentheses) is allowed as a parameter.

Note that after the opening angle brackets, at least one formal parameter name is required. If a macro is to be defined without formal parameters, the angle brackets should be omitted.

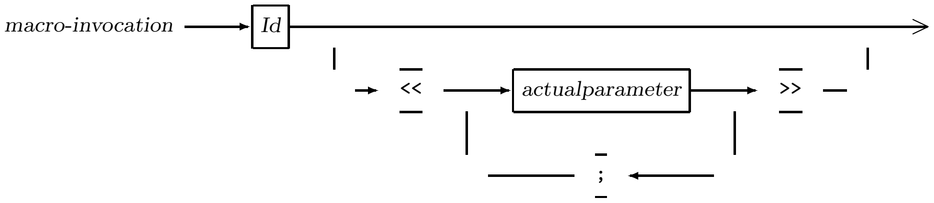
Here is an example which will exchange the values of two variables:

```
macro exchange <<x; y>>
      local temp;
      temp := x;
      x := y;
      y := z;
endm exchange;

  ⋮

if a(i) > a(j) then
    exchange <<a(i); a(j)>>
end if;
```

The `local` variable `temp` is used as a temporary—no user-variables in the same scope are affected.

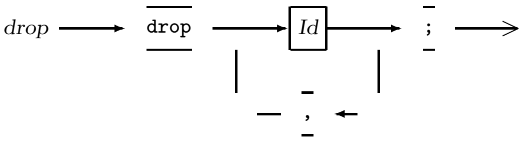
Since ProSet can deal with multiple assignment, too, the macro body could equivalently contain "`[x,y]:=[y,x];`" without the `local`-declaration.

The last line in the above example demonstrates a macro invocation; the general form is:



where *Id* is a macro name and the parameter list contains exactly as many actual parameters are given as formals are declared with the macro.
If the macro is defined without any formal parameters, its invocation will not make use of any argument list provided in angle brackets. However, an argument list in angle brackets occurring after an invocation of a parameter-less macro must not force an error — the arguments may serve as input to another macro being invoked by the expansion of the current macros body.

The scope of macros is given lexically: After defining the macro, the identifier after the keyword `macro` is bound to it until the end of the compiler run or until it is explicitly dropped by a `drop`-directive. This is a directive to the macro processor to undefine a list of macro names. It is constructed as follows:



Using the name of a macro for another macro definition is not allowed unless it is explicitly `drop`ped.

14.2 The Macro Processor

The macro processor works as follows (in a single pass):

- when the definition of a macro is encountered, the name of the macro, its definition and the names of the formal parameters are stored
- when the processor sees an identifier associated with a macro definition, it generates a fresh name for each local variable and substitutes the actual parameters (if any) for the formal parameters in the macro‘s body. This substitution is done exclusively on a textual basis. The text so generated replaces the macro invocation and is being read again by the macro processor to replace further macro applications.
- if a `drop`-directive is read, all the listed macro names are undefined

This stack oriented replacement mechanism allows for macros in macros (where enclosed macros should be dropped when leaving enclosing ones, otherwise two expansions of the enclosing macro will result in a double declaration of the innermost one).
Recursive (direct or indirect) macros, however, will cause infinite loops in the macro processor.

The definition

```
macro DefMacro <<x; y>>
      macro x
            y
      endm x;
endm DefMacro;
```

has the effect that after

```
DefMacro <<one; 1>>
DefMacro <<two; 2>>
```

the macros `one` and `two` are defined and will expand to `1` and `2`, respectively.

A Contour Model

In this appendix we present the *contour model* of ProSet. It helps the reader to understand the execution of a ProSet program and can be seen as an implementation technique for ProSet. The model is informally specified and only some of its major features being of interest for the users of ProSetare pointed out. For the purpose of this introductional specification much detail is glossed over. A more comprehensive specification is available as an internal report.

A.1 Constituents

Every programming language implicitly defines an *abstract machine*. Therefore a 'ProSet maschine' can be thought as an imaginary computer whose memory elements and machine instructions consist of ProSet's data objects, operations and control structures. The machine also has a state. Execution of a program, also called *interpretation*, transforms the *state* of the machine to another by applying the operations to the state of the machine. The state is characterized by the set of data objects and values existing at a given point of time during the execution. Furthermore the *procedure call hierarchy* and the representation of the next operation in the program text belong to the state.

The contour model captures these aspects and furthermore provides a conceptual basis for an implementation of ProSet. The model specifies an abstract machine and describes the execution of programs on this machine. An executing program is called an *application*. In the context of the contour model an application is a sequence of snapshots. Each snapshot consists of a ProSet program and the current state of the record of execution of that program. A snapshot results from the preceeding one by the execution of one statement.

A.1.1 Ranges

One important fact in the choice of an appropriate model for ProSet is the concept of *static* block structure. Nonlocal objects are taken from a static surrounding range. Another point of view is used in Lisp taking nonlocal objects by default from the *dynamic* environment.

The ranges reflect the static block structure of ProSet. As noted in section 4.3 a range is a syntactic construct that may contain declarations. Ranges are nested and inner ranges are not part of outer ones. For ProSet ranges are associated with the following (syntactic) units:

- Each ProSet program is embedded in the *standard environment* containing declarations of all keywords, e.g. `true`, `false`, `integer`, or `om` as well as introducing the default exception handler (section 8.5). These declarations cannot be expressed in ProSet.
- The main program (section 4.1).
- Procedures (section 4.5)
- Modules (section 11)
- Exception handlers (section 8.5)
- Constructs associated with bound variables, i.e. `for`- and `whilefound`-loops (section 7.4.4), set and tuple forming iterations (section 5.2.2 and 5.2.1), and quantified expressions (section **??**).

In the following we take the *standard environment* for granted. Furthermore the main program is treated like a procedure, this does not alter the scope rules.

```
procedure demo;
    visible constant x := 5;
    hidden y;
begin
    y := x + 1;
    z := x + y;
    p();

    procedure p();
    begin
        put(x);
    end p;

end demo;
```

Figure 26: An example for the contour model.

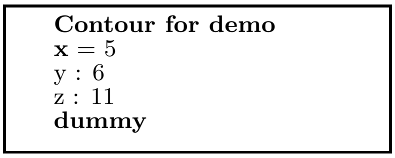


Figure 27: Contour for `demo`

A.1.2 Contours

Contours are the basic building blocks of the state descriptions. For ProSet each contour corresponds to exactly one range and contains the objects declared in that range. We use the graphical representation for contours introduced by Johnston[16]. A contour is represented by a rectangle. The notations used for the representation of data objects will be explained with the example in figure 26 (the line numbers are not part of the program). The range corresponding to the top level of the program `demo` contains four declarations. It serves as a template for the contour corresponding to that range, which is shown in figure 27. The contour represents the state of the four program entities before the procedure `p` is called.

When a new contour is established, all declared entities are entered. This includes also the implicit declared entities. Constants and variables are represented by pairs consisting of a name and a value. To distinguish constants from variables we use the symbol = for the binding of a name identifying a constant to a value and the symbol : in case of variables. Labels, exceptions, procedures, modules, and exception handlers are represented by their name. Integers, reals, booleans, strings, tuples, and sets are represented as usual, e.g. `1`, `3.26`, `true`, `"ProSet"`, `[1,2,3]`, `{3,4}`. For atoms we use arbitrary, but unique representation, e.g. _A2194610371 (probably containing a maschine id, a time stamp, etc.). The names of visible objects are written in boldface.

A.1.3 Nested Contours

One contour represents only a part of the state. As usual for block-structured languages the entire state corresponds to a collection of nested contours. The nesting of the contours corresponds to the structure of the nested ranges. As noted above, one contour corresponds to exactly one range, but one range may correspond to several contours (e.g. in case of recursive procedure calls). If the ranges $\mathcal{A}$ and $\mathcal{B}$ correspond to the contours $\mathcal{A}'$ and $\mathcal{B}'$ and $\mathcal{A}$ is immediately nested within $\mathcal{B}$, then $\mathcal{A}'$ is immediately nested within $\mathcal{B}'$. The contour $\mathcal{B}'$ is called the *static predecessor* of of $\mathcal{A}'$.

A.1.4 Environment Pointer

When executing a program, one contour corresponds to the range which contains the statement being currently performed. This contour is addressed by the *environment pointer ep* and is called the actual or *local* contour.

[16] Johnston, J. B. Contour Model of Block Structured Processes. SIGPLAN Notices 6(2), 55-82, 1971.

A.1.5 Instruction Pointer

Another constituent of this contour model is the *instruction pointer ip* indicating the position in the program text, i.e. *ip* points to the next operation during execution. In this model the instruction pointer is represented by the line number of the next operation.

A.1.6 Processor

The environment pointer *ep* and the instruction pointer *ip* are combined in the *processor* $\mathcal{P}$, which controls the program execution. In this contour model the processor consists of a triple containing a number for the identification of the processor (*pid*), an instruction pointer, and an environment pointer. The use of multiple processors has the advantage that multiprocessing can be modeled in a simple and natural way. A processor is represented the capital letter $\mathcal{P}$ decorated with its *pid* and its *ip*. The environment pointer results from the position of the processor in the graphical representation. For example the processor controling the execution of the program `demo` would be placed inside the contour for demo (fig. 27) as $\mathcal{P}_1^7$. If only one processor is required, the *pid* may be omitted.

A.1.7 Environment

A processor can access entities of its *environment*, consisting of the contour addressed by *ep* and all enclosing contours, by name. The *local* contour is the starting-point if we search an object identified by a given identifier *i*. First we search for *i* in the local contour. If this fails we continue searching in the surrounding contours from inner to outer until we find a definition for *i* being marked as visible.

A.2 State Transitions

Execution of a program on the abstract machine is composed of state transitions. There are several alternatives to change the state:

- Determination of the next operation.

  This can be done either implicitly, e.g. by a sequence of statements, or explicitly by a procedure call or a `quit`-statement.

- Assigning a new value to an existing data object.

  This happens by an assignment or parameter transmissions.

- Changes of the state size or structure:

  - Construction or removal of a value of type `function`, `modtype`, and `instance`.
  - Range entry.
  - Range exit.

In the following we will discuss some state transitions more detailed.

A.2.1 Bound Variables

In ProSet there are some constructs containing bound variables (`for`- and `whilefound`-loops, set and tuple forming iterations, and quantified expressions). Instead of associating them with ranges, the bound variables are *consistently renamed*, i.e. they are replaced by identifiers not appearing elsewhere in the program. The following example

```
x := 5;
for x in {1..10} do
     if x in {x*x:  x in [1..3]} then put(x); end if;
end for;
put(x);
```

is consistently renamed to

```
x := 5;
for t1 in {1..10} do
     if t1 in {t2*t2:  t2 in [1..3]} then put(t1); end if;
end for;
put(x);
```

with `t1` and `t2` as fresh names.

Hence, the introduced identifiers are treated as hidden variables declared in the enclosing range corresponding to an exception handler, a procedure, or a module.

A.2.2 Procedures

Upon call of a procedure p, the `rd`- and `rw`-parameters are computed in the local contour c. Now a new contour c' corresponding to the range of p is established. The new contour is placed immediately within the contour c'' being the local contour when p was declared. The actual parameters are entered in c'' i.e. the formal `rd`- and `rw`-parameters are initialized to the computed values and the `wr`-parameters are initialized to `om`. Then *ep* is set to point to c'' and *ip* is set to the first statement of p to be executed. The contour c is called the *dynamic predecessor* of c$'$. This is indicated by an arrow from c'' to c in the graphical representation.

Upon return of a procedure p, the return value, the `rw`- and `wr`-parameters are written back to the dynamic predecessor c. Then *ep* is set to point to c, *ip* addresses the statement immediately following the procedure call, and the contour c'' is removed.

A.2.3 Exception Handling

When an exception e is raised, we search the dynamic predecessor c'' of the local contour c for an association of a handler h with e. If the search succeeds, a new contour c'' corresponding to the range of h is established. The placement of the new contour as well as the transmission of parameters are analogously to procedures. Of course *ep* now points to c'' and *ip* is set to point to the first statement of h. If the search fails, the *default handler* will be executed, terminating the program. Conceptually, the default handler is established in the contour corresponding to the standard environment.

The actions to be executed upon handler exit depend on the form of termination (we suppose, that it agrees with the raise form; otherwise, the default handler will be executed):

resume: Discard the local contour and let *ep* point to the dynamic predecessor, i.e. the contour for the signaller.

return: Discard the local contour c$'$ and the dynamic predecessor c$'$. Let *ep* point to the the dynamic predecessor c$''$ of c$'$.

abort: Discard c, c$'$, and c$''$ (as used above). The dynamic predecessor of c$''$ becomes the local contour.

A.2.4 Functions

Values of type `function` are created by applying the `closure` operator to a procedure name or lambda expression (section 5.3.1). The representation of a function reflects the semantics of the `closure` operator. Values of type `function` are represented as triples consisting of the following three components:

1. an atom (used for the equality test)
2. a line number representing the corresponding procedure or lambda expression.
3. a pointer to the closure, i.e. a copy of the enclosing contours of the procedure or lambda expression at the time when the `closure` operator was applied.

Upon call of a function, a copy of its closure is established outside any other contours and the contour corresponding to the procedure or lambda expression is placed within the innermost contour of this copy.

A.2.5 Modules and Instances

Modules are represented by their names. Applying the `closure` operator to a module name yields a value of type `modtype`. This value is represented analogously to a value of type `function`. Values of type `instance` are represented in the same way except that their closures additionally contain an innermost contour corresponding to the top levels of the instantiated modules.

A.2.6 Multiple Processes

As noted in section A.1.6 the use of processors allows a simple and natural modeling of multiple processes. The spawning of a new process results in establishing a new, additional processor with a new and unique *pid*. The placeholder returned as result of process spawning is represented by the *pid* of the new processor being put in parentheses. When a processor accesses an object associated with a placeholder, it is suspended until the future resolves to the value.

B Predefined Exceptions

exception	raised by	when	see
`escape type_mismatch()`	[frequently used]		8.7
`escape illegal_operand()`	`i ** j`	`type i = integer, j<0 or j=i=0`	5.1.1
		`type i = real, i=0.0 and j=0`	5.1.2
	`i / j`	`type j = integer, j=0`	5.1.1
		`type j = real, j=0.0`	5.1.2
	`i mod j`	`type j = integer, j=0`	5.1.1
	`s with x`	`type s = set, x=om`	5.2.2
`escape illegal_index()`	`f(i)`	`type f = string, i<0`	5.1.4
		`type f = tuple, i<0`	5.2.1
	`f(i..j)`	`type f = string, i<0 or j<i-1 or j>#f`	5.1.4
		`type f = tuple, i<0 or j<i-1`	5.2.1
	`f(i.. )`	`type f = string, i<0 or #f<i-1`	5.1.4
		`type f = tuple, i<0 or #f<i-1`	5.2.1
`escape illegal_map_operation()`	`f{x}`	`type f = set and ( not is_map f )`	5.2.3
	`f(x)`	`type f = set and ( not is_map f )`	5.2.3
`escape arity_mismatch()`	application of a function object		5.3.1
`escape mode_mismatch()`	application of a function object		5.3.1
`escape memory_full()`	[frequently used]		8.7
`signal ts_is_full()`	`deposit`		10.3.1
`escape ts_invalid_id()`	`ExistsTS`, `ClearTS`, `RemoveTS`		13.8
`escape io_file_error()`	I/O-routines		12.4
`signal io_syntax_error()`	I/O-routines		12.4
`signal io_type_mismatch()`	`.put.`/`.get.`-routines		12.4
`escape io_arguments_missing()`	`.put.`/`.get.`-routines		12.4
`escape io_illegal_input()`	`.get.`-routines		12.4
`notify p_missing_name()`	persistent declaration		9.1.1
`escape p_missing_pfile()`	persistent declaration		9.1.1
`escape p_missing_rights()`	persistent declaration		9.1.1

C Concrete Grammar

```
/******************************************
Program and procedure definition:
******************************************/
xProgDefn ::=   'program' id ';' xProgBody
                'end' id ';' .
xProgBody ::=   xDecls xcBeginStmts xcPHMDefns .
xcPHMDefns ::=   xcPHMDefns xPHMDefn .
xcPHMDefns ::=   .
xPHMDefn ::=   'procedure' id xParamList ';'
               xProgBody 'end' id ';' .
xPHMDefn ::=   'module' id xParamList ';'
               xProgBody 'end' id ';' .
xPHMDefn ::=   'handler' id xParamList xImplAsso
               ';' xProgBody 'end' id ';' .
% Parameter list:
xParamList ::=   '(' ')' .
xParamList ::=   '(' xcParams ')' .
xcParams ::=   xcParams ',' xParamMode id .
xcParams ::=   xParamMode id .
% Parameter mode:
xParamMode ::=   .
xParamMode ::=   'rd' .
xParamMode ::=   'rw' .
xParamMode ::=   'wr' .
% Implicit handler association:
xImplAsso ::=   'for' xIdList .
xImplAsso ::=   'for' 'others' .
xImplAsso ::=   .
xIdList ::=   xIdList ',' id .
xIdList ::=   id .
/******************************************
Declarations:
******************************************/
xDecls ::=   xDecls xDecl .
xDecls ::=   .
xDecl ::=   xDeclKey xcVars xExplAsso ';' .
xcVars ::=   xcVars ',' xSingleVar .
xcVars ::=   xSingleVar .
xSingleVar ::=   id ':=' xExpr .
xSingleVar ::=   id .
xDeclKey ::=   'visible' .
xDeclKey ::=   'hidden' .
xDeclKey ::=   'visible' 'constant' .
xDeclKey ::=   'hidden' 'constant' .
xDeclKey ::=   'constant' .
%
xDecl ::=   xPersDecl xIdList ':' xExpr
            xExplAsso ';' .
xPersDecl ::=   'visible' 'persistent'.
xPersDecl ::=   'hidden' 'persistent' .
xPersDecl ::=   'visible' 'persistent' 'constant'.
xPersDecl ::=   'hidden' 'persistent' 'constant'.
xPersDecl ::=   'persistent' 'constant' .
xPersDecl ::=   'persistent' .
/******************************************
Statements:
******************************************/
xcBeginStmts ::=   'begin' xStmts .
xStmts ::=   xStmts xStmt xExplAsso ';' .
xStmts ::=   xStmt xExplAsso ';' .
%
xExplAsso ::=   xExplAsso 'when' xHandAsso .
xExplAsso ::=   .
%
xHandAsso ::=   xIdList 'use' id .
%
```

```
% Simple Statements:
xStmt ::=   'pass' .
xStmt ::=   'stop' .
xStmt ::=   'stop' xExpr .
xStmt ::=   'return' xExpr .
xStmt ::=   'return' .
xStmt ::=   'resume' .
xStmt ::=   'abort' xExpr .
xStmt ::=   'abort' .
xStmt ::=   'signal' id xActuList .
xStmt ::=   'notify' id xActuList .
xStmt ::=   'escape' id xActuList .
%
xStmt ::=   xStdIO xActuList .
xStdIO ::=   'put' .
xStdIO ::=   'eput' .
xStdIO ::=   'fput' .
xStdIO ::=   'putf' .
xStdIO ::=   'eputf' .
xStdIO ::=   'fputf' .
xStdIO ::=   'get' .
xStdIO ::=   'fget' .
xStdIO ::=   'getf' .
xStdIO ::=   'fgetf' .
%
% Assignments:
xStmt ::=   xLValue ':=' xExpr .
xStmt ::=   xLValue xBinOp ':=' xExpr .
xStmt ::=   xLValue xFrom xcSimpleLV .
xFrom ::=   'from' .
xFrom ::=   'frome' .
xFrom ::=   'fromb' .
%
% Function calls:
xStmt ::=   xcSimpleLV xActuList .
xActuList ::=   '(' xExprList ')' .
xActuList ::=   '(' ')' .
%
% Recursive lambda calls:
xStmt ::=   'self' xActuList .
%
% Conditional statements:
xStmt ::=   'if' xExpr 'then' xStmts xElIfStmts
            xElseStmts 'end' 'if' .
xElIfStmt ::=   'elseif' xExpr 'then' xStmts .
xElIfStmts ::=   xElIfStmts xElIfStmt .
xElIfStmts ::=   .
%
xElseStmts ::=  'else' xStmts .
xElseStmts ::=  .
%
% Case statements:
xStmt ::=   'case' xExpr xCaseStmts  xElseStmts
            'end' 'case' .
xCaseStmts ::=   xCaseStmts xcCaseStmt .
xCaseStmts ::=   xcCaseStmt .
xcCaseStmt ::=    xcCaseList xStmts .
xcCaseList ::=    'when' xExprList '=>' .
%
% Loop statements:
xStmt ::=   xcLoopStmt 'end' 'loop' .
xStmt ::=   xcForStmt 'end' 'for' .
xStmt ::=   xcWhileStmt 'end' 'while' .
xStmt ::=   xcWhilefound 'end' 'whilefound' .
xStmt ::=   xcUntilStmt 'end' 'repeat' .
xStmt ::=   xLabel xLoops 'end' id .
xLabel ::=   id ':' .
xLoops ::=   xcLoopStmt .
xLoops ::=   xcForStmt .
xLoops ::=   xcWhileStmt .
xLoops ::=   xcWhilefound .
xLoops ::=   xcUntilStmt .
xcLoopStmt ::=   ’loop’ xStmts .
xcForStmt ::=   ’for’ xIterator ’do’ xStmts .
xcWhileStmt ::=   ’while’ xExpr ’do’ xStmts .
xcWhilefound ::=   ’whilefound’ xIterator
                   ’do’ xStmts .
xcUntilStmt ::=   ’repeat’ xStmts ’until’ xExpr .
xStmt ::=   ’quit’ .
xStmt ::=   ’quit’ id .
xStmt ::=   ’continue’ .
xStmt ::=   ’continue’ id .
%
% Process spawing statement:
xStmt ::=   ’||’ xExpr .
%
% Tuple-Space Operations:
xStmt ::=   ’deposit’ xDepList ’end’ ’deposit’ .
xStmt ::=   ’deposit’ xDepList ’blockiffull’
            ’end’ ’deposit’ .
xDepList ::=   xDepList xExprList ’at’ xQualId .
xDepList ::=   xExprList ’at’ xQualId .
%
xStmt ::=   ’fetch’ xTempListAt xElseStmts
            ’end’ ’fetch’ .
%
xStmt ::=   ’meet’ xTempListAt xElseStmts
            ’end’ ’meet’ .
%
xTempListAt ::=   xTempListAt ’or’ xcTempList
                  ’at’ xQualId .
xTempListAt ::=   xcTempList ’at’ xQualId .
xcTempList ::=   xcTempList ’or’ xTemplate .
xcTempList ::=   xTemplate .
xTemplate ::=   ’(’ xcEFEmpty ’)’ ’=>’ xStmts .
xTemplate ::=   ’(’ xcEFEmpty ’)’ .
xcEFEmpty ::=   .
xcEFEmpty ::=   xcEFList .
xcEFList ::=   xcEFList ’,’ xExprFormal .
xcEFList ::=   xExprFormal .
xExprFormal ::=   xExpr xInto.
xExprFormal ::=   ’?’ xFormal xInto.
xInto ::=   ’into’ xExpr .
xInto ::=   .
xFormal ::=   xcSimpleLV ’|’ xExpr .
xFormal ::=   ’|’ xExpr .
xFormal ::=   xcSimpleLV .
xFormal ::=   .
%
/*******************************************
Iterators:
*******************************************/
xIterator ::=   xSimpleIts ’|’ xExpr .
xIterator ::=   xSimpleIts .
xSimpleIts ::=   xSimpleIts ’,’ xSimpleIt .
xSimpleIts ::=   xSimpleIt .
xSimpleIt ::=   xLValue ’in’ xExpr .
xSimpleIt ::=   xLValue ’=’ id xMapSel .
/*******************************************
Map Selectors for simple iterators:
*******************************************/
xMapSel ::=   ’(’ xcLValList ’)’ .
xMapSel ::=   ’{’ xcLValList ’}’ .
xcLValList ::=   xcLValList ’,’ xLValue .
xcLValList ::=   xLValue .
/*******************************************
```

```
Left hand side values:
*******************************************/
xLValue ::=   xcSimpleLV .
xcSimpleLV ::=   xQualId .
xcSimpleLV ::=   xcSimpleLV xSelector .
xLValue ::=   ’[’ xcComps ’]’ .
xcComps ::=   xcComps ’,’ xcComp .
xcComps ::=   xcComp .
xcComp ::=   xLValue .
xcComp ::=   ’-’ .
/*******************************************
Selectors:
*******************************************/
xSelector ::=   ’(’ xExprList ’)’ .
xSelector ::=   ’{’ xExprList ’}’ .
xSelector ::=   ’(’ xExpr ’..’ ’)’ .
xSelector ::=   ’(’ xExpr ’..’ xExpr ’)’ .
/*******************************************
Former:
*******************************************/
% sets:
xcSetFormer ::=   xFormer .
xcSetFormer ::=   xExpr ’,’ xExprList .
xcSetFormer ::=   xExpr ’,’ xExpr ’..’ xExpr .
% tuples:
xcTupFormer ::=   xFormer .
xcTupFormer ::=   xcTupComp ’,’ xExpr ’..’ xExpr .
xcTupFormer ::=   xcTupComp ’,’ xcTCList .
xcTCList ::=   xcTCList ’,’ xcTupComp .
xcTCList ::=   xcTupComp .
xcTupComp ::=   xExpr .
xcTupComp ::=   ’-’ .
% general:
xFormer ::=   xExpr .
xFormer ::=   xExpr ’..’ xExpr .
xFormer ::=   xExpr ’:’ xIterator .
/********************************************
Expressions:
*******************************************/
xExprList ::=   xExprList ’,’ xExpr .
xExprList ::=   xExpr .
/*******************************************
Primary Expressions without possible selections:
*******************************************/
xcPrimary ::=   int .
xcPrimary ::=   float .
xcPrimary ::=   ’true’ .
xcPrimary ::=   ’false’ .
xcPrimary ::=   ’om’ .
xcPrimary ::=   ’atom’ .
xcPrimary ::=   ’boolean’ .
xcPrimary ::=   ’integer’ .
xcPrimary ::=   ’real’ .
xcPrimary ::=   ’string’ .
xcPrimary ::=   ’tuple’ .
xcPrimary ::=   ’set’ .
xcPrimary ::=   ’function’ .
xcPrimary ::=   ’modtype’ .
xcPrimary ::=   ’instance’ .
xcPrimary ::=   ’rd’ .
xcPrimary ::=   ’rw’ .
xcPrimary ::=   ’wr’ .
xcPrimary ::=   ’{’ ’}’ .
xcPrimary ::=   ’[’ ’]’ .
%
% Newat:
xcPrimary ::=   ’newat’ ’(’ ’)’ .
%
```

```
% module instantiations:
xcPrimary ::=   'instantiate' xExpr xcClauses
                'end' 'instantiate' .
xcClauses ::=   xcClauses xClause .
xcClauses ::=   xClause .
xClause ::=   'rd' xcImports ';' .
xClause ::=   'rw' xcImports ';' .
xClause ::=   'wr' xIdList ';' .
xcImports ::=   xcImports ',' xImport .
xcImports ::=   xImport .
xImport ::=   id ':=' xExpr .

/******************************************
Primary Expressions with possible selections:
******************************************/
xcPrimary ::=   xcPriSel .
xcPriSel ::=   str .
xcPriSel ::=   '$' .
xcPriSel ::=   'argv' .
xcPriSel ::=   '{' xcSetFormer '}' .
xcPriSel ::=   '[' xcTupFormer ']' .
%
% Identifier:
xcPriSel ::=   xQualId .
xQualId ::=   xQualId '.' id .
xQualId ::=   id .
%
% Lambda Expressions:
xcPriSel ::=   xLambda .
xLambda ::=   'lambda' xParamList ':' xProgBody
              'end' 'lambda' .
%
% Recursive lambda calls:
xcPriSel ::=   'self' xActuList .
xcPowTerm ::=   'self' .
%
xcPriSel ::=   xcPriSel xSelector .
xcPriSel ::=   xcPriSel '(' ')' .
%
% Explicit handler associations:
xcPriSel ::=   xcPriSel '[' xHandAsso ']' .
%
% Quantifiers:
xExpr ::=   xQuantifier .
xQuantifier ::=   xQualifier xSimpleIts
                  '|' xcPowTerm .
xQualifier ::=   'exists' .
xQualifier ::=   'forall' .
%
% Conditional Expressions:
xcPriSel ::=   'if' xExpr 'then' xExpr xElIfExprs
               xElseExpr 'end' 'if' .
xElIfExprs ::=   xElIfExprs xElIfExpr .
xElIfExprs ::=   .
xElIfExpr ::=   'elseif' xExpr 'then' xExpr .
%
xElseExpr ::=   'else' xExpr .
xElseExpr ::=   .
%
% Case Expressions:
xcPriSel ::=   'case' xExpr xCaseExprs xElseExpr
               'end' 'case' .
xCaseExprs ::=   xCaseExprs xcCaseExpr .
xCaseExprs ::=   xcCaseExpr .
xcCaseExpr ::=   xcCaseList xExpr .
%
xcPriSel ::=   '(' xExpr ')' .
/******************************************
```

```
Binary operations:
******************************************/
xExpr ::=   xExpr xcOrOp xcOrTerm / xcOrTerm .
xcOrOp ::=   'or' .
xcOrOp ::=   'or' '%' .
xcOrTerm ::=  xcOrTerm xcAndOp xcAndTerm /
              xcAndTerm .
xcAndOp ::=   'and' .
xcAndOp ::=   'and' '%' .
xcAndTerm ::=  xcAndTerm xcBoolOp xcBoolTerm /
               xcBoolTerm .
xcBoolOp ::=   '=' .
xcBoolOp ::=   '/=' .
xcBoolOp ::=   '<' .
xcBoolOp ::=   '<=' .
xcBoolOp ::=   '>' .
xcBoolOp ::=   '>=' .
xcBoolOp ::=   'in' .
xcBoolOp ::=   'notin' .
xcBoolOp ::=   'subset' .
xcBoolOp ::=   '=' '%' .
xcBoolOp ::=   '/=' '%' .
xcBoolOp ::=   '<' '%' .
xcBoolOp ::=   '<=' '%' .
xcBoolOp ::=   '>' '%' .
xcBoolOp ::=   '>=' '%' .
xcBoolOp ::=   'in' '%' .
xcBoolOp ::=   'notin' '%' .
xcBoolOp ::=   'subset' '%' .
xcBoolTerm ::=   xcBoolTerm xcSetOp xcSetTerm /
                 xcSetTerm .
xcSetOp ::=   'with' .
xcSetOp ::=   'less' .
xcSetOp ::=   'lessf' .
xcSetOp ::=   '!' id .
xcSetOp ::=   'with' '%' .
xcSetOp ::=   'less' '%' .
xcSetOp ::=   'lessf' '%' .
xcSetOp ::=   '!' id '%' .
xcSetTerm ::=  xcSetTerm xcAddOp xcAddTerm /
               xcAddTerm .
xcAddOp ::=   '+' .
xcAddOp ::=   '-' .
xcAddOp ::=   'max' .
xcAddOp ::=   'min' .
xcAddOp ::=   '+' '%' .
xcAddOp ::=   '-' '%' .
xcAddOp ::=   'max' '%' .
xcAddOp ::=   'min' '%' .
xcAddTerm ::=   xcAddTerm xcMulOp xcMulTerm /
                xcMulTerm .
xcMulOp ::=   '*' .
xcMulOp ::=   '/' .
xcMulOp ::=   'mod' .
xcMulOp ::=   '*' '%' .
xcMulOp ::=   '/' '%' .
xcMulOp ::=   'mod' '%' .
xcMulTerm ::=  xcMulTerm xcPowOp xcPowTerm /
               xcPowTerm .
xcPowOp ::=   '**' .
xcPowOp ::=   '**' '%' .
xcPowTerm ::=   xUnOp xcPowTerm .
xcPowTerm ::=   xcPrimary .
%
xBinOp ::=   xcOrOp .
xBinOp ::=   xcAndOp .
xBinOp ::=   xcBoolOp .
xBinOp ::=   xcSetOp .
xBinOp ::=   xcAddOp .
xBinOp ::=   xcMulOp .
xBinOp ::=   xcPowOp .
/******************************************
Unary Operators:
******************************************/
xUnOp ::=   ’+’ .
xUnOp ::=   ’-’ .
xUnOp ::=   ’#’ .
xUnOp ::=   ’||’ .
xUnOp ::=   ’not’ .
xUnOp ::=   ’pow’ .
xUnOp ::=   ’arb’ .
xUnOp ::=   ’random’ .
xUnOp ::=   ’domain’ .
xUnOp ::=   ’range’ .
xUnOp ::=   ’type’ .
xUnOp ::=   ’profile’ .
xUnOp ::=   ’closure’ .
xUnOp ::=   ’is_map’ .
xUnOp ::=   ’is_smap’ .
xUnOp ::=   ’!’ id .
xUnOp ::=   ’or’ ’%’ .
xUnOp ::=   ’and’ ’%’ .
xUnOp ::=   ’=’ ’%’ .
xUnOp ::=   ’/=’ ’%’ .
xUnOp ::=   ’<’ ’%’ .
xUnOp ::=   ’<=’ ’%’ .
xUnOp ::=   ’>’ ’%’ .
xUnOp ::=   ’>=’ ’%’ .
xUnOp ::=   ’in’ ’%’ .
xUnOp ::=   ’notin’ ’%’ .
xUnOp ::=   ’subset’ ’%’ .
xUnOp ::=   ’with’ ’%’ .
xUnOp ::=   ’less’ ’%’ .
xUnOp ::=   ’lessf’ ’%’ .
xUnOp ::=   ’!’ id ’%’ .
xUnOp ::=   ’+’ ’%’ .
xUnOp ::=   ’-’ ’%’ .
xUnOp ::=   ’max’ ’%’ .
xUnOp ::=   ’min’ ’%’ .
xUnOp ::=   ’*’ ’%’ .
xUnOp ::=   ’/’ ’%’ .
xUnOp ::=   ’mod’ ’%’ .
xUnOp ::=   ’**’ ’%’ .
```

Keyword index

Nonterminal index

General index