- Language Report -

concurrent
Clean

-

a general purpose, higher order, pure and lazy
functional programming language
based on graph rewriting
designed for the development of
sequential, parallel and distributed
real world applications

-

- Version 1.2 (draft) -

HILT - High Level Software Tools B.V.
and
University of Nijmegen

Rinus Plasmeijer

Marko van Eekelen

- Language Report -

Preface

Introduction

CONCURRENT CLEAN is a *practical applicable general purpose lazy pure functional programming language* suited for *the development of real world applications.*

CLEAN (Brus *et al.*, 1987; Nöcker *et al.*, 1991; Plasmeijer and Van Eekelen, 1993) is well-known for its many features and its fast compiler producing very efficient code.

CLEAN runs on a *Mac*, *PowerMac*, *Sun* and *PC* (*Windows'95*, *WindowsNT*, *OS/2* and *Linux*).

In CLEAN we have incorporated those features we felt people really need to write real world programs (such as records, arrays, higher order types, type classes, type constructor classes and much more) based on our own experience with writing complex applications.

People already familiar with other functional programming languages (such as Miranda (Turner, 1985), SML (Harper *et al.*, 1986), Haskell (Hudak *et al.*, 1992) and Gofer/Hugs (Jones, 1993)) will have no difficulty to program in CLEAN and we hope that they enjoy CLEAN's rich collection of features, CLEAN's compilation speed and the quality of the produced code.

CLEAN has many features among which some very special ones. Of particular importance for practical use is CLEAN's uniqueness type system enabling the incorporation of destructive updates of arbitrary objects within a pure functional framework and the creation of direct interfaces with the outside world.

CLEAN's "unique" features have made it possible to predefine (in CLEAN) a sophisticated and efficient I/O library. The CLEAN I/O library enables a CLEAN programmer to *specify interactive window based I/O applications* on a very high level of abstraction. The library forms a *platform independent interface to* *window systems*: one can port window based I/O applications written in CLEAN to different platforms without any modification of source code.

In the new I/O library version 1.0 different kind of call-back functions and I/O definitions can be active at the same time, thus providing the possibility to *combine* different *interactive* CLEAN *programs* into a new application (a kind of multi-tasking within the same application). The applications can be regarded as lightweight processes which can communicate via files, shared state or message passing primitives ((a)synchronous message passing, remote procedure call). All this is provided in a *pure, sequential* functional world in which the call-back functions act as indivisible event handlers.

CLEAN also has *concurrency* primitives to create functions which can be executed in *parallel*. It is also possible to define *distributed executing interactive applications* running on several PC's/workstations connected in a network. These options are being tested and will become available in future versions of the system.

More information on CLEAN

A book on functional programming in CLEAN is being written in collaboration with the Universities of Utrecht, Leiden and the polytechnical Universities of Arnhem and Leeuwarden. The book contains lots of case studies. The first draft version of this book is available on the net (www.cs.kun.nl/~clean).

The basic concepts behind CLEAN (albeit version 0.8) as well as an exploination of the implementation techniques used can be found in Plasmeijer and Van Eekelen (Adisson-Wesley, 1993).

There are many papers on the concepts introduced by the CLEAN group (such as *term graph rewriting* (Barendregt *et al.*, 1987), *lazy copying* (van Eekelen *et al.*, 1991), *abstract reduction* (Nöcker, 1993), *uniqueness typing* (Barendsen and Smetsers, 1993, 1996), CLEAN's *I/O concept* (Achten, 1996), *Parallel* CLEAN (Kesseler, 1996) An annotated bibliography can be found in the appendix.

For the most recent information on papers and information about CLEAN please check our web pages (www.cs.kun.nl/~clean).

About this language report

In this report the syntax and semantics of CLEAN version 1.2 are explained. We always give a motivation why we have included a certain feature. Although the report is not intended as introduction into the language, we did our best to make it as readable as possible. Nevertheless, one sometimes has to work through several sections spread all over the report. E.g. each predefined type is treated in Chapter 8, information on the use and creation of objects of these type can be found in Chapter 4, explanation on what kind of pattern matching facilities are possible in Chapter 6.

At several places in this report context free syntax fragments of CLEAN are given. We sometimes repeat fragments which are also given elsewhere just to make the description clearer (e.g. in the uniqueness typing chapter we repeat parts of the syntax for the classical types). We hope that this is not confusing. The complete collection of context free grammar rules are summarised in Appendix A.

Some of the features mentioned in this report are still under consideration, design and/or implementation and therefore not yet incorporated in the current release of the CLEAN *system. They should be regarded as "possible future trends". Perhaps they will be kicked out, perhaps we incorporate them in slightly different form, perhaps they will be there in full glory in the next release. We also take the liberty to make small syntactic changes in future versions of the language.*

Note: this manual discusses the new CLEAN 1.0 I/O library which currently is only available on Mac's. For a description of the CLEAN 0.8 I/O system which is available on *all* platforms we support, see the draft version of new book on programming in CLEAN, available on internet.

Some remarks on the CLEAN syntax

The CONCURRENT CLEAN syntax is similar to the notation found in most other modern functional languages. However, there are a couple of small syntactic differences we want to point out here for people who don't like to read language reports.

In CLEAN the arity of a function is reflected in its type. When a function is defined its uncurried type is specified! To avoid any confusion we want to explicitly state here that in CLEAN there is no restriction whatsoever on the curried use of functions. However, we don't feel a need to express this in every type. Actually, the way we express types of functions more clearly reflects the way curried functions are internally treated.

The standard map function (arity 2) is specified in CLEAN as follows:

```
map::(a -> b) [a] -> [b]
map f []       = []
map f [x:xs]   = [f x:map f xs]
```

Each predefined structure such as a list, a tuple, a record or array has its own kind of brackets: lists are *always* denotated with square brackets `[...]`, for tuples the usual parentheses are used `(...,...)`, curly braces are used for records (indexed by field name) as well as for arrays (indexed by number).

In types funny symbols can appear like `.`, `u:`, `*`, `!` which can be ignored and left out if one is not interested in uniqueness typing or strictness.

There are only a few keywords in CLEAN leading to a heavily overloaded use of `:` and `=` symbols:

```
function::argstype -> restype          //    type specification of a function
function pattern | guard = rhs         //    definition of a function

selector = graph                       //    definition of a constant/CAF/graph

function args :== rhs                  //    definition of a macro

::type args =  type                    //    an algebraic type definition
::type args :== type                   //    a type synonym definition
::type args                            //    an abstract type definition
```

Notational conventions

The following *notational conventions* are used in this report. Text is printed in Garamond 12pts,

▌ the context free syntax descriptions are given in Geneva 9pts,
`examples of CLEAN programs are given in Courier 9pts,`
textual explanation to the examples are given in Garamond 10pts.

- Semantical restrictions are always given in a bulleted (•) list-of-points. When these restrictions are not obeyed they will almost always result in a compile-time error. In very few cases the restrictions can only be detected at run-time (array index out-of-range, partial function called outside the domain).

The following notational conventions are used in the context-free syntax descriptions:

[notion]	means that the presence of notion is optional
{notion}	means that notion can occur zero or more times
{notion}+	means that notion occurs at least once
{notion}-*list*	means one or more occurrences of notion separated by comma's
terminals	are printed in **bold 10 pts courier**
symbols	are printed in *italic*
~	is used for concatenation of notions
{notion}/-str	means the longest expression not containing the string str

All CLEAN examples given in this report assume that the lay-out dependent mode has been chosen which means that redundant semi-colons and curly braces are left out (see Section 3.6).

How to obtain CLEAN

Concurrent Clean and the Concurrent Clean Program Development system can be used free of charge for *educational purposes only*. They can be obtained

- via World Wide Web (*www.cs.kun.nl/~clean*) or
- via ftp (*ftp.cs.kun.nl* in directory *pub/Clean*).

It is allowed to copy the system again *for educational purposes only* under the condition that the *whole* distribution for a certain platform is copied, including help files, this language report and the copyright notices.

For any use of Clean in a commercial environment a *commercial license* is required, which is *not* free of charge. Information about commercial licenses can be obtained by contacting Rinus Plasmeijer (rinus@cs.kun.nl). For commercial users we supply additional utility software and give full technical support to enable you to incorporate Clean and Clean applications successfully in your specific environment.

Concurrent Clean is available on several platforms. The current situation is as follows (please check our WWW-pages regularly to see latest news):

platform	**Macintosh**	**PowerMac**	**PC**	**PC**	**PC**	**Sun**	**Sun**
oper. sys.	**MacOS 6.0**	**MacOS 7.1.2**	**OS/2 2.0**	**Windows '95 / NT**	**Linux ELF**	**SunOS 4.1.2**	**Solaris 2.0**
processor	Motorola	PowerPC	Intel	Intel	Intel	Sparc	Sparc
process. type	any	any	>= 486	>= 486	>= 486	any	any
window system	MacOS	MacOS	OS/2 2.0	Windows '95	Xview	Xview/ Open-Look	Xview/ Open-Look
Clean compiler	1.2	1.2	1.1	1.2	1.1	1.1	1.1
Clean I/O lib	1.0/0.8	1.0/0.8	0.8	0.8	0.8	0.8	0.8
Clean PDS	C version	Clean vrs.	make files	Clean vrs.	make files	make files	make files
assembler	not needed	not needed	not needed	not needed	gnu ass.	Sun ass.	Sun ass.
linker	included	included	OS2 linker	included	gnu linker	Sun linker	Sun linker
Code gen	Seq	Seq	Seq	Seq	Seq	Seq	Seq
RAM in PC							
- minimal	4 Mb	8 Mb	8 Mb	8 Mb	8 Mb	8 Mb	8 Mb
- comfortable	8 Mb	16 Mb	16 Mb	16 Mb	16 Mb	16 Mb	16 Mb
Disk usage							
- minimal	5 Mb	7 Mb	6 Mb	6 Mb	6 Mb	7 Mb	7 Mb
Available	now	now	now	*soon*	now	now	now

The installation of the Clean compiler is rather dependent on the kind of platform one is working on. For each platform there is a help file which should help you to install properly. On the Mac's there is a dedicated Clean Programming Development System including dedicated editor, library search facilities and a project manager. There is a new version of the Clean Programming Development System which is entirely written in Clean which we currently are porting to several platforms. For the platforms without Development System one needs to use one of the standard editors available on the platform. In that case a distribution includes make files which will do the project management. We generate native code for *all* platforms.

The Clean compiler is set up to make parallel and distributed evaluation possible. This feature will be made available later.

Current state of the CLEAN System

Some of the features mentioned in this report are still under consideration, design and/or implementation and therefore not yet incorporated in the current release of the Clean *system. They should be regarded as "possible future trends". Perhaps they will be kicked out, perhaps we incorporate them in slightly different* *form, perhaps they will be there in full glory in the next release. We also take the liberty to make small syntactic changes in future versions of the language.*

Release 1.2 (Januari 1997).

- For any expression local definitions can be introduced with a let statement.
- We have introduced a new kind of let statements *before* a guard. Actions which have to be done in sequence can now much more intuitively be written down in such a sequential order. These new special let statements also have a special scope. It allows to reuse the name for a single threaded parameter. Consequence is that for instance I/O actions can be written down in a more natural style (it looks imperative but it is not, of course).
- Guards can be nested.
- The syntax for algebraic types has been changed for existential quantifiers. The type `Void` in no longer a predefined type.
- Array comprehensions return arrays with unique elements.
- Multidimensial arrays can now be used in selection patterns and updates.
- Observation of unique objects is possible albeit for simple cases only (observation of objects of basic type stored in unique structures and observations made by polymorphic projection functions). It makes it more easy to inspect unique datastructures before they are updated.
- It's no longer necessary to place parentheses around lambda expressions when they are used as arguments. This is especially useful when using a monadic programming style.
- The CLEAN compiler now gives warnings for functions that are not used in a module and are not exported. No code is generated for these functions.
- The strictness analyzer is improved for guarded function alternatives.
- Some bugs in the compiler have been removed.
- We generate slightly better code (e.g. for functions that return strict tuples).
- CLEAN'S native PowerMac (MacOS) version is now released on the net.
- The old CLEAN I/O examples are rewritten to make use of the new features in Clean 1.2..
- CLEAN'S 1.0 I/O LIBRARY will soon be released on the net for Mac and PowerMac.
- New CLEAN I/O examples for the new 1.0 library are made.
- The CLEAN Program Development System has been improved.
- The language report has been updated for version 1.2 including a new chapter on I/O. Still some work as to be done on the chapter about uniqueness typing.

The current release of the CLEAN *system has the following limitations:*

- The CLEAN 1.0 I/O library is currently only available for the Mac. The CLEAN 0.8 I/O library (albeit converted to CLEAN 1.0 syntax) is available on all platforms. For a description of the 0.8 I/O library we refer to the draft of the new CLEAN book on the net and to the Addison-Wesley book (Plasmeijer and Van Eekelen, 1993).
- The Class mechanism can only have one type class variable which can only be instantiated with a flat type. Due to this restriction we had to define the overloaded array operators in a rather complicated way. This gives rise to a too complex class context for overloaded array operators. We are working on this feature.
- The code generator is improved taking user defined strictness information into account. It is not optimal yet, under test and therefore not switched on in the current release.
- Macros are at this moment substituted in an early stage of the compilation process. This may cause criptical error messages.
- Only simple variables can be used as array pattern.
- The arrow type constructor `(->)` cannot be used prefixed or used in a curried way.
- Annotations for parallelism are ignored. The distributed code generator is switched off. We are working on it.
- Everything exported in a definition module still has to be repeated in the corresponding implementation module.
- The new CLEAN programming environment is only available for some platforms and needs improvement.

Sorry for all these inconveniences, we are working hard on it.

Release 1.1 (March 1996). The syntax and semantics of classes are improved. The overload declaration is incorporated in the class declaration. It is now also possible to combine uniqueness typing with type (constructor) classes. Arrays can be used as an instantiation of classes. There are different kind of array implementations for optimal efficiency (lazy, strict, unboxed). The class concept makes it possible to define overloaded functions which can deal with all of them (although we are not yet completely happy with the current solution). Uniqueness type attribute equations can now also be specified by the programmer. This allows the definition of higher order functions like '`bind`' such that they can now also be applied to possibly unique arguments without enforcing unnecessary restrictions. A string is not a basic type anymore but has become synonym for an (unboxed) array of character (the type `String` is now defined as type synonym in module `StdString`). Curly braces are used for arrays instead of the ugly '`{:`' '`:}`' pair. Macro definitions can contain local definitions (which are substituted as well). Macros can be applied curried. Constructors for which also functions are defined are kicked out (there were not used very often and it complicated the compiler). The Standard Environment has slightly changed (sorry about this inconvenience). Some operators and functions are moved to other modules to increase orthogonality. The priority of some operators have been changed. We also had to rename some functions (e.g. `#` to `size/length` and `##` to `maxindex`) because these symbols are reserved for a handy syntax extension which will become available in the next release.

CLEAN is ported to PowerMac (MacOS) (a native version which can generate native applications), Sun (Solaris) en PC (Linux). The CLEAN 0.8 I/O library is ported to all these platforms as well. There is a new CLEAN programming environment (written in CLEAN). We will improve this environment (we know it is far from perfect yet) and will port it to all the other supported platforms. Some bugs in the compiler have been removed. Some space leaks have been removed as well. More strictness is found (in local definitions). We generate slightly better code.

Release 1.0.3 (October 1995). Some bugs in the compiler have been removed.

Release 1.0.2 (June 1995). CLEAN is ported to PC (OS/2) and Sun (SunOS). The CLEAN 0.8 I/O library is ported to these platforms as well. Some bugs in the compiler have been removed.

Release 1.0.1 (April 1995). CLEAN 1.0 release on the Mac (Motorola). Compared with the previous public release (0.84b) many important changes have been made (there is a noticeable difference between an intermediate language and a programming language).

The most important changes in the language are:
- CLEAN has been changed from an intermediate language to a functional programming language with a syntax in the style of Miranda, Haskell and the like;
- so, various small syntactic sugar is added (infix operators, a case construct, local function definitions, lambda-abstractions, list comprehensions, lay-out rule, etcetera);
- overloaded functions, type classes and type constructor classes can be defined;
- records and arrays are added as predefined data structure with handy operations (such as an update operator for arrays and records, array comprehensions etc.);
- a more refined control of strictness is possible (partially strict data structures can be defined for any type, in particular for recursive types, there is strict let construct);
- the uniqueness typing is refined (now polymorphic and inferred, observation of uniquely typed objects is made easier);
- existentially quantified types can be defined.

Also the CLEAN I/O library has been changed:
- the I/O library is improved (with respect to orthogonality, modularity, extendibility, portability);
- the I/O library is extended allowing to define interactive processes running interleaved inside one application which can communicate via files, shared data and message passing;
- one can define interactive processes which (in the near future) can run distributed on workstations connected via a network.

This new 1.0 I/O library is not yet made public available in this release (1.0.1). The old 0.8 I/O library (converted to 1.0 syntax) will be made available on all platforms.

The compiler and code generator have been extended and are partly rewritten. Furthermore,

- the code generator is improved;
- the code generator is prepared for parallel and distributed evaluation;

Compared with the 0.84 version we have made a lot of syntactic changes to the language. The complete redesign of CLEAN has as consequence that CLEAN version 1.0 is *not* compatible with its predecessors. A CLEAN application is available which can transform programs written in old CLEAN into new CLEAN.

Copyright, Authors and Credits

CONCURRENT CLEAN and the CONCURRENT CLEAN DEVELOPMENT SYSTEM are a product of

HILT - HIGH LEVEL SOFTWARE TOOLS B.V.,

The Netherlands.

HILT is a Dutch company owned by the CLEAN team founded to ensure excellent technical support for commercial environments. HILT furthermore educates in functional programming and assists in making commercial applications with CLEAN.

CLEAN is a spin-off of the research performed by the research group on functional programming languages, COMPUTING SCIENCE INSTITUTE, at the UNIVERSITY OF NIJMEGEN under the supervision of Rinus Plasmeijer.

The CONCURRENT CLEAN System is developed by:

Peter Achten:	Sequential and distributed Event I/O, I/O library support for the Mac.
John van Groningen:	CLEAN compiler, Code generators (Mac (Motorola, PowerPC), PC (Intel), Sun (Sparc)), Low level interfaces, all machine wizarding.
Rober Holwerda:	I/O library support for Windows '95 & Windows NT.
Martin van Hintum:	Program Development System (CLEAN version).
Marko Kesseler:	Parallel code generator (ParSyTec (Transputer)).
Eric Nöcker:	Strictness analyser via abstract reduction, I/O library support for OS/2.
Leon Pillich:	I/O library support for the Sun.
Sjaak Smetsers:	CLEAN compiler, All type systems (including uniqueness typing and type classes),
Ron Wichers Schreur:	Program Development System (C version), Testing, Parser, Support, Porting, CLEAN 0.8 to 1.0 Conversion program, CLEAN distribution on the net.
Rinus Plasmeijer & Marko van Eekelen:	CLEAN language design.
Rinus Plasmeijer:	Overall design and implementation supervision.

Special thanks to the following people:

Christ Aarts, Steffen van Bakel, Erik Barendsen, Henk Barendregt, Pieter Hartel, Hans Koetsier, Pieter Koopman, Ronan Sleep and all the CLEAN users who helped us to get a better system.

Many thanks to the following sponsors:

- the Dutch Technology Foundation (STW);
- the Dutch Foundation for Scientific Research (NWO);
- the International Academic Centre for Informatics (IACI);
- Kropman B.V., Installation Techniques, Nijmegen, The Netherlands;

- Hitachi Advanced Research Laboratories, Japan;
- the Dutch Ministry of Science and Education (the Parallel Reduction Machine project (1984-1987)) who initiated the CONCURRENT CLEAN research;
- Esprit Basic Research Action (project 3074, SemaGraph: the Semantics and Pragmatics of Graph Rewriting (1989-1991));
- Esprit Basic Research Action (SemaGraph II working group 3646 (1992-1995));
- Esprit Parallel Computing Action (project 4106, (1990-1991));
- Esprit II (TIP-M project area II.3.2, Tropics: TRansparent Object-oriented Parallel Information Computing System (1989-1990)).

Final Remark

We hope that CLEAN *indeed enables you to program your applications in a convenient and efficient way. We will continue to improve the language and the system. We greatly appreciate your comments and suggestions for further improvements.*

Januari 1997

Rinus Plasmeijer and Marko van Eekelen

Affiliation:	HILT High Level Software Tools B.V.	COMPUTING SCIENCE INSTITUTE
Mail address:	Universitair Bedrijven Centrum, Toernooiveld 100, 6525 EC Nijmegen, The Netherlands.	University of Nijmegen, Toernooiveld 1, 6525 ED Nijmegen, The Netherlands.
e-mail:	rinus@cs.kun.nl marko@cs.kun.nl	rinus@cs.kun.nl marko@cs.kun.nl
Phone:	+31 6 54346073	+31 24 3652644
Fax:	+31 24 3652525	+31 24 3652525
CLEAN on internet:	www.cs.kun.nl/~clean	
CLEAN on ftp:	ftp.cs.kun.nl in pub/Clean	
Questions about CLEAN:	clean@cs.kun.nl	
Subscription mailing list::	clean@cs.kun.nl, subject:: subscribe	

Table of contents

1

Introduction

In this section we summarize the key design rules and major features of CLEAN.

1.1 Key design rules for CLEAN

The most important *features* of CLEAN are:

- CLEAN is a *lazy*, *pure*, *higher order functional programming language* with explicit *graph rewriting semantics*; one can explicitly define the *sharing* of *structures* (*cyclic structures* as well) in the language;

- Although CLEAN is *by default* a *lazy language* one can smoothly turn it into a *strict language* to obtain optimal time/space behaviour: *functions* can be defined *lazy* as well as *(partially) strict* in their arguments; any (recursive) *data structure* can be defined *lazy* as well as *(partially) strict* in any of its arguments;

- CLEAN is a *strongly typed* language based on an extension of the well-known Milner / Hindley / Mycroft type inferencing/checking scheme (Milner 1978; Hindley 1969; Mycroft 1984) including the common *polymorphic types*, *abstract types*, *algebraic types*, and *synonym types* extended with a restricted facility for *existentially quantified types*;

- *Type classes* and *type constructor classes* are provided to make *overloaded* use of functions and operators possible.

- CLEAN offers the following *predefined types*: *integers*, *reals*, *Booleans*, *characters*, *strings*, *lists*, *tuples*, *records*, *arrays* and *files*;

- CLEAN's key feature is a *polymorphic uniqueness type inferencing system*, a special extension of the Milner / Hindley / Mycroft type inferencing/checking system allowing a refined control over the *single threaded use of objects*; with this uniqueness type system one can influence the time and space behaviour of programs; it can be used to incorporate *destructive updates of objects within a pure functional framework*, it allows destructive transformation of *state information*, it *enables efficient interfacing* to the non-functional world (to C but also to I/O systems like X-Windows) offering *direct access to file systems and operating systems*;

- CLEAN is a *modular language* allowing *separate compilation* of modules; one defines *implementation modules* and *definition modules*; there is a facility to implicitly and explicitly import definitions from other modules;

- CLEAN offers a sophisticated *I/O library* with which *window based interactive applications* (and the handling of *menus*, *dialogues*, *windows*, *mouse*, *keyboard*, *timers* and *events* raised by sub-applications) can be specified compactly and elegantly on a very high level of abstraction;

- Specifications of window based interactive applications can be *combined* such that one can create several applications (*sub-applications* or *light-weight processes*) inside *one* CLEAN application. *Automatic switching* between these sub-applications is handled in a similar way as under a *multi-finder* (all low level event handling for updating windows and switching between menus is done automatically); sub-applications can exchange information with each other (via *files*, via clipboard copy-paste like actions using *shared state components*, via *asynchronous message passing*) but also with other independently programmed (CLEAN or other) applications running on the *same* or even on a *different* host system;

- Sub-applications can be created on other machines which means that one can define *distributed window based interactive* CLEAN applications communicating e.g. via (*a*)*synchronous message passing* and *remote procedure calls* across a local area network;

- *Dynamic process creation* is possible; processes can run *interleaved* or in *parallel*; *arbitrary process topologies* (for instance cyclic structures) can be defined; the *interprocess communication* is synchronous and is handled *automatically* simply when one function demands the evaluation of its arguments being calculated by another process possibly executing on another processor;

- Due to the strong typing of CLEAN and the obligation to initialise all objects being created *run-time errors can only occur in a very limited number of cases*: when partial functions are called with arguments out of their domain (e.g. dividing by zero), when arrays are accessed with indices out-of-range and when not enough memory (either heap or stack space) is assigned to a CLEAN application;

- Programs written in CLEAN using the 0.8 I/O library can be ported without modification of source code to anyone of the many platforms we support (see the Preface for an overview).

2

Basic semantics

The semantics of CLEAN is based on *Term Graph Rewriting Systems* (Barendregt, 1987a; Plasmeijer and Van Eekelen, 1993). This means that functions in a CLEAN program semantically work on *graphs* instead of the usual *terms*. This enabled us to incorporate CLEAN's typical features (definition of cyclic data structures, lazy copying, uniqueness typing) which would otherwise be very difficult to give a proper semantics for. However, in many cases the programmer does not need to be aware of the fact that he/she is manipulating graphs. Evaluation of a CLEAN program takes place in the same way as in other lazy functional languages. One of the "differences" between CLEAN and other functional languages is that when a variable occurs more than once in a function body, the semantics *prescribe* that the actual argument is shared (the semantics of most other languages do not prescribe this although it is common practice in any implementation of a functional language). Furthermore, one can label any expression to make the definition of cyclic structures possible. So, people familiar with other functional languages will have no problems writing CLEAN programs.
When larger applications are being written, or, when CLEAN is interfaced with the non-functional world, or, when efficiency counts, or, when one simply wants to have a good understanding of the language it is good to have some knowledge of the basic semantics of CLEAN which is based on term graph rewriting. In this chapter a short introduction into the basic semantics of CLEAN is given. An extensive treatment of the underlying semantics and the implementation techniques of CLEAN can be found in Plasmeijer and Van Eekelen (1993).

2.1 Graph rewriting

A CLEAN *program* basically consists of a number of *graph rewrite rules* (*function definitions*) which specify how a given *graph* (the *initial expression*) has to be *rewritten*.

A *graph* is a set of nodes. Each node has a defining *node-identifier* (the *node-id*). A *node* consists of a *symbol* and a (possibly empty) sequence of *applied node-id's* (the *arguments* of the symbol). *Applied node-id's* can be seen as *references* (*arcs*) to nodes in the graph, as such they have a *direction*: from the node in which the node-id is applied to the node of which the node-id is the defining identifier.

Each *graph rewrite rule* consists of a *left-hand side graph* (the *pattern*) and a *right-hand side* (rhs) consisting of a *graph* (the *contractum*) or just a *single* node-id (a *redirection*). In CLEAN rewrite rules are not comparing: the left-hand side (lhs) graph of a rule is a tree, i.e. each node identifier is applied only once, so there exists exactly one path from the root to a node of this graph.

A rewrite rule defines a (*partial*) *function*. The *function symbol* is the root symbol of the left-hand side graph of the rule alternatives. All other symbols that appear in rewrite rules, are *constructor symbols*.

The *program graph* is the graph that is rewritten according to the rules. Initially, this program graph is fixed: it consists of a single node containing the symbol `Start`, so there is no need to specify this graph in the program explicitly. The part of the graph that matches the pattern of a certain rewrite rule is called a *redex* (*reducible expression*). A *rewrite of a redex* to its *reduct* can take place according to the right-hand side of the corresponding rewrite rule. If the right-hand side is a contractum then the rewrite consists of building this contractum and doing a redirection of the root of the redex to root of the right-hand side. Otherwise, only a redirection of the root of the redex to the single node-id specified on the right-hand side is performed. A *redirection* of a node-id n_1 to a node-id n_2 means that all applied occurrences of n_1 are replaced by occurrences of n_2 (which is in reality commonly implemented by *overwriting* n_1 with n_2).

A *reduction strategy* is a function that makes choices out of the available redexes. A *reducer* is a process that reduces redexes that are indicated by the strategy. The result of a reducer is reached as soon as the reduction strategy does not indicate redexes any more. A graph is in *normal form* if none of the patterns in the rules match any part of the graph. A graph is said to be in *root normal form* when the root of a graph is not the root of a redex and can never become the root of a redex. In general it is undecidable whether a graph is in root normal form.

A pattern *partially matches* a graph if firstly the symbol of the root of the pattern equals the symbol of the root of the graph and secondly in positions where symbols in the pattern are not syntactically equal to symbols in the graph, the corresponding sub-graph is a redex or the sub-graph itself is partially matching a rule. A graph is in *strong root normal form* if the graph does not partially match any rule. It is decidable whether or not a graph is in strong root normal form. A graph in strong root normal form does not partially match any rule, so it is also in root normal form.

The default reduction strategy used in CLEAN is the *functional reduction strategy*. Reducing graphs according to this strategy resembles very much the way execution proceeds in other lazy functional languages: in the standard lambda calculus semantics the functional strategy corresponds to normal order reduction. On graph rewrite rules the functional strategy proceeds as follows: if there are several rewrite rules for a particular function, the rules are tried in textual order; patterns are tested from left to right; evaluation to strong root normal form of arguments is forced when an actual argument is matched against a corresponding non-variable part of the pattern. A formal definition of this strategy can be found in (Toyama *et al.*, 1991).

2.1.1 A small example

Consider the following CLEAN program:

```
Add Zero z          =    z                          (1)
Add (Succ a) z      =    Succ (Add a z)             (2)

Start               =    Add (Succ o) o
                         where
                              o = Zero              (3)
```

In CLEAN a distinction is between function definitions (graph rewriting rules) and graphs (constant definitions). A semantic equivalent definition of the program above is given below where this distinction is made explicit ("=>" indicates a rewrite rule whereas "=:" is used for a constant (*sub-*) *graph* definition.

```
Add Zero z          =>   z                          (1)
Add (Succ a) z      =>   Succ (Add a z)             (2)

Start               =>   Add (Succ o) o
                         where
                              o =: Zero             (3)
```

These rules are internally translated to a semantically equivalent set of rules in which the graph structure on both left-hand side as right-hand side of the rewrite rules has been made explicit by adding node-id's. Using the set of rules with explicit node-id's it will be easier to understand what the meaning is of the rules in the graph rewriting world.

```
x =: Add y z
y =: Zero           =>   z                          (1)
```

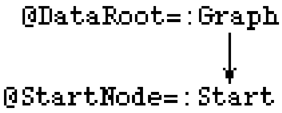
```
x =: Add y z
y =: Succ a       =>   m =: Succ n
                       n =: Add a z               (2)

x =: Start        =>   m =: Add n o
                       n =: Succ o
                       o =: Zero                  (3)
```

The fixed initial program graph that is in memory when a program starts is the following:

The initial graph in linear notation:

```
@DataRoot     =: Graph @StartNode
@StartNode    =: Start
```

The initial graph in pictorial notation:



To distinguish the node-id's appearing in the rewrite rules from the node-id's appearing in the graph the latter always begin with a '`@`'.

The initial graph is rewritten until it is in normal form. Therefore a CLEAN program must at least contain a "*start rule*" that matches this initial graph via a pattern. The right-hand side of the start rule specifies the actual computation. In this start rule in the left-hand side the symbol **`Start`** is used. However, the symbols `Graph` and `Initial` (see next section) are internal, so they cannot actually be addressed in any rule.

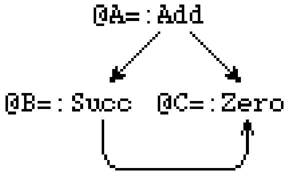
The patterns in rewrite rules contain *formal node-id's*. During the matching these formal nodeid's are mapped to the *actual node-id's* of the graph. After that the following semantic actions are performed:

The start node is the only redex matching rule (3). The contractum can now be constructed:

The contractum in linear notation:

```
@A   =: Add  @B @C
@B   =: Succ @C
@C   =: Zero
```

The contractum in pictorial notation:

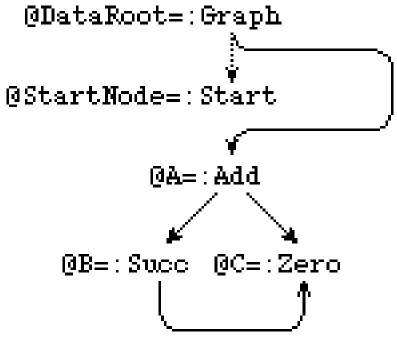


All applied occurrences of `@StartNode` will be replaced by occurrences of `@A`. The graph after rewriting is then:

The graph after rewriting:

```
@DataRoot     =: Graph @A
@StartNode    =: Start
@A   =: Add  @B @C
@B   =: Succ @C
@C   =: Zero
```

Pictorial notation:

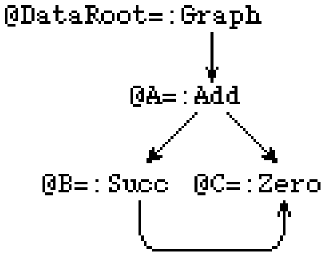


This completes one rewrite. All nodes that are not accessible from `@DataRoot` are garbage and not considered any more in the next rewrite steps. In an implementation once in a while garbage collection is performed in order to reclaim the memory space occupied by these garbage nodes. In this example the start node is not accessible from the data root node after the rewrite step and can be left out.

The graph after garbage collection:

```
@DataRoot      =: Graph @A
@A   =: Add  @B @C
@B   =: Succ @C
@C   =: Zero
```

Pictorial notation :

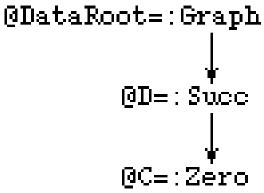


The graph accessible from `@DataRoot` still contains a redex. It matches rule 2 yielding the expected normal form:

The final graph:

```
@DataRoot      =: Graph @D
@D   =: Succ @C
@C   =: Zero
```

Pictorial notation :



The fact that graphs are being used in CLEAN gives the programmer the ability to explicitly share terms or to create cyclic structures. In this way time and space efficiency can be obtained.

2.2 Global graphs

Due to the presence of global graphs in CLEAN the initial graph in a specific CLEAN program is slightly different from the basic semantics. In a specific CLEAN program the initial graph is defined as:

```
@DataRoot      =:   Graph @StartNode @GlobId1 @GlobId2 … @GlobIdn
@StartNode     =:   Start
@GlobId1  =:   Initial
@GlobId2  =:   Initial
…
@GlobIdn  =:   Initial
```

The root of the initial graph will not only contain the node-id of the start node, the root of the graph to be rewritten, but it will also contain for each *global graph* (see 5.1) a reference to an initial node (initialised with the symbol `Initial`). All references to a specific global graph will be references to its initial node or, when it is rewritten, they will be references to its reduct.

3

Lexical structure

In this chapter the lexical structure of CLEAN is explained. It describes the kind of tokens recognised by the scanner/parser (Sections 3.1, 3.2 and 3.3).
In Section 3.4 the symbols are summarised which are used in the context-free syntax description in the chapters hereafter (they are written in italic in the syntax description).
An overview of the scopes induced by the language constructs of CLEAN are given in section 3.5.
As is common in modern functional languages there is a lay-out rule (off-side rule) in CLEAN which permits the omission of braces and semicolons. This lay-out rule is described in Section 3.6. All examples in this report make use of the lay-out rule.

3.1 Lexical program structure

```
LexProgram       =  { Lexeme | {Whitespace}+ }
Lexeme           =  ReservedKeyword                      //  see Section 3.3
                 |  ReservedSymbol                       //  see Section 3.3
                 |  ReservedChar
                 |  Literal                              //  see Section 3.2
                 |  Identifier
Identifier       =  LowerCaseId
                 |  UpperCaseId
                 |  FunnyId
LowerCaseId      =  LowerCaseChar~{IdChar}
UpperCaseId      =  UpperCaseChar~{IdChar}
FunnyId          =  {SpecialChar}+

LowerCaseChar    =  a | b | c | d | e | f | g | h | i | j
                 |  k | l | m | n | o | p | q | r | s | t
                 |  u | v | w | x | y | z
UpperCaseChar    =  A | B | C | D | E | F | G | H | I | J
                 |  K | L | M | N | O | P | Q | R | S | T
                 |  U | V | W | X | Y | Z
SpecialChar      =  ~ | @ | # | $ | % | ^ | ? | !
                 |  + | - | * | < | > | \ | / | | | & | =
                 |  :
IdChar           =  LowerCaseChar
                 |  UpperCaseChar
                 |  Digit
                 |  _ | `
Digit            =  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
CharDel          =  '
StringDel        =  "

Whitespace       =  space                                //  a space character
                 |  tab                                  //  a horizontal tab
                 |  newline                              //  a newline char
```

```
                         |  formfeed                              //  a formfeed
                         |  verttab                               //  a vertical tab
                         |  Comment
   Comment               =  // AnythingTillNL newline
                         |  /* AnythingTill/* Comment AnythingTill*/ */
                         |  /* AnythingTill*/ */
   AnythingTillNL        =  {AnyChar/newline}                     //  no newline
   AnythingTill/*        =  {AnyChar//*}                          //  no "/*"
   AnythingTill*/        =  {AnyChar/*/}                          //  no "*/"
   AnyChar               =  IdChar | ReservedChar | Special
   ReservedChar          =  (  |  )  |  {   |  }  |  [  |  ]  |  ;  |  ,  |  .
   Special               =  \n  |  \r |  \f |  \b                 // newline,return,formf,backspace
                         |  \t  |  \\|  \CharDel                  // tab,backslash,character delete
                         |  \StringDel                            // string delete
                         |  \{OctDigit}+                          // octal number
                         |  \x{HexDigit}+                         // hexadecimal number
   OctDigit              =  0  |  1  |  2  |  3  |  4  |  5  |  6  |  7
   HexDigit              =  0  |  1  |  2  |  3  |  4  |  5  |  6  |  7  |  8  |  9
                         |  A  |  B  |  C  |  D  |  E  |  F
                         |  a  |  b  |  c  |  d  |  e  |  f
```

3.2 Literals

```
   Literal               =  IntegerDenot
                         |  RealDenot
                         |  BoolDenot
                         |  CharDenot
                         |  CharsDenot
                         |  StringDenot
   IntegerDenot          =  [Sign]~{Digit}+              // decimal
                         |  [Sign]~0~{OctDigit}+         // octal
                         |  [Sign]~0x~{HexDigit}+        // hexadecimal
   Sign                  =  + | - | ~
   RealDenot             =  [Sign~]{Digit~}+.{~Digit}+[~E[~Sign]{~Digit}+]
   BoolDenot             =  True | False
   CharDenot             =  CharDel~AnyChar/CharDel.CharDel
   CharsDenot            =  CharDel~{AnyChar/CharDel}+.CharDel
   StringDenot           =  StringDel~{AnyChar/StringDel}~StringDel
```

Example (literals).

```
   Integer (decimal):        0|1|2|…|8|9|10| … |-1|-2| …
   Integer (octal):          00|01|02|…|07|010| … |-01|-02| …
   Integer (hexadecimal):    0x0|0x1|0x2|…|0x8|0x9|0xA|0xB … |-0x1|-0x2| …
   Real:                     0.0|1.5|0.314E10| …
   Boolean:                  True | False
   Character:                'a'|'b'|…|'A'|'B'|…
   String:                   "" | "Rinus"|"Marko"|…
   List of characters:       ['Rinus']|['Marko']|…
```

3.3 Reserved keywords and symbols

Below the symbols are listed which have a special meaning in the language. Some symbols only have a special meaning in a certain context. Outside this context they are ordinary identifiers. In the comment it is indicated for which context (name space) the symbol is predefined.

```
   ReservedKeyword       =                    //   in all contexts:

                            /*                //   begin of comment block
                         |  */                //   end of comment block
                         |  //                //   rest of line is comment
                         |  ::                //   begin of a type definition
                         |  :==               //   in a type synonym or macro definition
                         |  =                 //   in a function, graph, alg. type, rec. field
                         |  =:                //   labeling a graph definition
                         |  =>                //   in a function definition
                         |  ->                //   in a case expression, lambda abstraction
                         |  [                 //   begin of a list
                         |  :                    //  cons node
                         |  ]                    //  end of a list
                         |  \\                   //  begin of list or array comprehension
                         |  <-                   //  list gen. in list or array comprehension
                         |  <-:                  //  array gen. in list or array comprehension
                         |  {                    //  begin of a record or array, begin of a block
                         |  }                    //  end of a record or array, end of a block
                         |  &                    //  an update of a record or array
                         |  {*                   //  begin of process annotations
                         |  *}                   //  end of process annotations
                         |  case                 //  begin of case expression
                         |  class                //  begin of type class definition
                         |  code                 //  begin code block in a syst impl. module
                         |  default              //  to indicate default class instance
                         |  definition           //  begin of definition module
                         |  export               //  to reveal which class instances there are
                         |  from                 //  begin of symbol list for imports
                         |  if                   //  begin of a conditional expression
                         |  implementation       //  begin of implementation module
                         |  import               //  begin of import list
                         |  in                   //  end of (strict) let expression
                         |  infix                //  infix indication in operator definition
                         |  infixl               //  infix left indication in operator definition
                         |  infixr               //  infix right indication in operator definition
                         |  instance             //  def of instance of a type class
                         |  let                  //  begin of let expression
                         |  #                    //  begin of let expression (for a guard)
                         |  let!                 //  begin of strict let expression
                         |  #!                   //  begin of strict let expression (for a guard)
                         |  module               //  in module header
                         |  of                   //  in case statement
                         |  system               //  begin of system module
                         |  where                //  begin of local def of a function alternative
                         |  with                 //  begin of local def in a rule alternative

   ReservedSymbol        =                       //  in type specifications:

                            !                    //  strict type
                         |  .                    //  uniqueness type variable
                         |  #                    //  unboxed type
                         |  *                    //  unique type
                         |  :                    //  in a uniqueness type variable definition
                         |  ->                   //  function type constructor
                         |  []                   //  list type constructor
                         |  (,),(,,),(,,,),...   //  tuple type constructors
                         |  {},{!},{#}           //  lazy, strict, unboxed array type constr.
                         |  Bool                 //  type Boolean
                         |  Char                 //  type character
                         |  File                 //  type file
                         |  Int                  //  type integer
                         |  ProcId               //  type process id
                         |  Real                 //  type real
                         |  World                //  type world


                                                 //  in process annotations:

                         |  at                   //  followed by processor id
                         |  P                    //  a parallel process to normal form
                         |  I                    //  an interleaved process to normal form
```

3.4 Symbols, identifiers and name spaces

In the context-free syntax description given in this language report the symbols listed below are used. The *symbols* are *identifiers* used to name modules, functions, operators, graphs, constructors, (node) variables, field names, macros, types, type variables, uniqueness types, uniqueness type (constructor) variables and type classes. The convention used is that variables always start with a lowercase character while constructors and types always start with an uppercase character. The other identifiers may either start with an uppercase or a lowercase character.

It is allowed to use the same identifier for different purposes as long as the symbols belong to different *name spaces*. Function-, operator-, constructor-, graph-, macro-symbols and node variables form one name space. Type variables and uniqueness type variables form together another name space. All other symbols form a name space on their own.

Under certain conditions it is allowed to use the same name for different functions and operators (overloading, see 8.4).

Notice that for the identifiers names can be used consisting of a combination of lower and/or uppercase characters but one can also define identifiers constructed from special characters like +, <, etc. (see 3.1). These two kind of identifiers cannot be mixed. This makes it possible to leave out white space in expressions like `a+1` (same as `a + 1`). See also 4.3.

ModuleSymb = LowerCaseId | UpperCaseId | FunnyId
FunctionSymb = LowerCaseId | UpperCaseId | FunnyId
ConstructorSymb = UpperCaseId | FunnyId
SelectorVariable = LowerCaseId
Variable = LowerCaseId
MacroSymb = LowerCaseId | UpperCaseId | FunnyId
FieldSymb = LowerCaseId
TypeSymb = UpperCaseId | FunnyId
TypeVariable = LowerCaseId
UniqueTypeVariable = LowerCaseId
ClassSymb = LowerCaseId | UpperCaseId | FunnyId

3.5 Scope of definitions overview

The scope is the program region in which an introduced definition (e.g. function definition, type definition) and corresponding names (e.g. function name, variable name, type name, type variable name) has a meaning. Scopes can be nested: within a scope new scopes can be defined. Within such a nested scope new definitions can be given, new names can be introduced. As usual it is allowed in a nested scope to re-define definitions or re-use names given in a surrounding scope. A definition given or a name introduced in a (nested) scope has no meaning in surrounding scopes. It has a meaning for all scopes nested within it (unless they are redefined within such a nested scope).

In the pictures in the subsections below nested scopes are indicated by nested boxes.

- Within a scope different objects of the same kind (i.e. belonging to the same name space, see 3.4) must have different names.

3.5.1 Scope of definitions given in a definition module

The definitions of a definition module have the widest scope. All symbols that are defined in a definition module are also automatically visible (exported) to *all* other modules. In the latter case imports are required to effectuate the actual scope of a symbol to the other module.

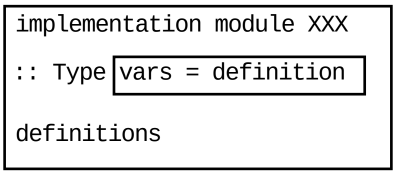
- Within one module a symbol can be defined (see 12.2) only once within the same scope and within the same name space (see 3.4).

3.5.2 Scope of global definitions given in an implementation module

Definitions on the *global* (= outermost) level (see 12.1.1) have in principle as scope the implementation module they are defined in, unless they are exported by the corresponding definition module (see 12.3).

3.5.3 Scope of type definitions

Types can only be defined on the global level (see 3.5.2). Type variables introduced on the left-hand side of a (algebraic, record, synonym, overload, class, instance, function, graph) type definition have the right-hand side of the type definition as scope.

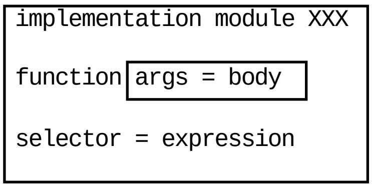


```
implementation module XXX

:: Type vars = definition

definitions
```

Figure (Scope of type definitions).

3.5.4 Scope of function / graph definitions

More complex are the scope rules within function and graph definitions (see the Chapters 5 and 6). Functions and graphs (selectors) can be defined on the *global* level (see 3.52). Variables introduced on the left-hand side of a function definition (formal arguments) have a meaning in the function definition.

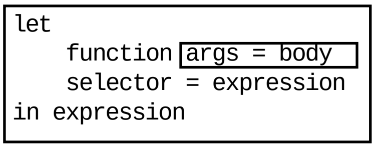


```
implementation module XXX

function args = body

selector = expression
```

Figure (Scope of functions and graphs defined on the global level).

But, functions and graphs (selectors) can be also be defined *locally*. However, there is no general syntactic way to introduce a new scope at any point in the program text.: new scopes can only be introduced at certain points. In functional languages local definitions are by tradition defined by using let-statements (definitions given before they are used in a certain expressio, nice for a bottom-up style of programming) and where-blocks (definitions given afterwards, nice for a top-down style of programming).

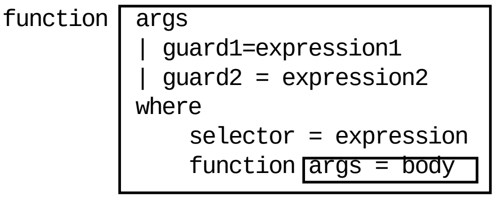
With a *let* statement one can define new functions and graphs which only have a meaning within a certain expression. This is allowed in any expression on the right-hand side of a function or graph definition. The same scope rule applies for a *strict let* expression.



```
let
    function args = body
    selector = expression
in expression
```

Figure (Defining functions and graphs locally for a certain expression).

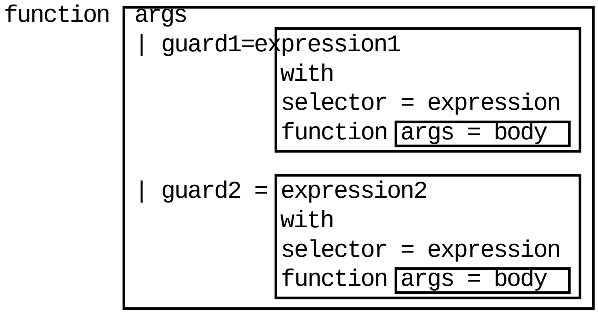
With a *where* block one can define functions and graphs which have meaning within every expression appearing in a *function alternative* (see 6.6). It is not possible to define functions or graphs local to a whole function definition (i.e. in scope of all function alternatives of a function definition).



```
function args
         | guard1=expression1
         | guard2 = expression2
         where
             selector = expression
             function args = body
```

Figure (Defining functions and graphs locally for a function alternative).

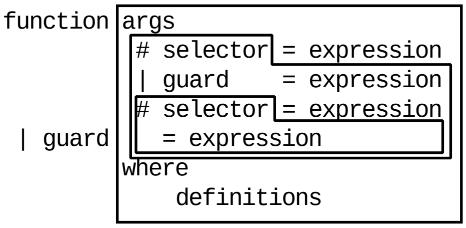
The scope induced by a *where* block can sometimes be too big. In CLEAN one can also restrict the scope of definitions to the body of a *rule alternative* by using a *with* statement.



```
function args
  | guard1=expression1
           with
           selector = expression
           function args = body

  | guard2 = expression2
             with
             selector = expression
             function args = body
```

Figure (Defining functions and graphs locally for a rule alternative).

For expressions which have to be evaluated in a certain sequential order it is very convenient to define these expressions *before* a guard in which they can be tested. This makes it possible to define selectors (graphs) and tests on the contents of these selectors in a textual order which closely corresponds to the order in which they are supposed to be evaluated. A special *let* statement is provided (keyword `let` or `#`) called *let-before* which introduces a very special scope. Each selector defined induces a new scope (excluding the body of the selector, see the picture below) ending at the function body. One cannot use the selectors defined in a *let-before* in a *where* block, the other way around is possible.

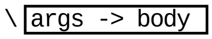


```
function args
          # selector = expression
          | guard    = expression
          # selector = expression
  | guard   = expression
         where
            definitions
```

Figure (Defining local graphs before a guard).

3.5.5 Scope within expressions

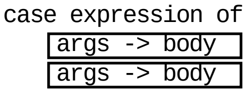
Within an expression new formal parameters can also be introduced. This can happen in a *lambda expression*, which is a nameless function. The formal parameters have a meaning in the corresponding function body.



```
\ args -> body
```

Figure (Introducing formal parameters via lambda expression).

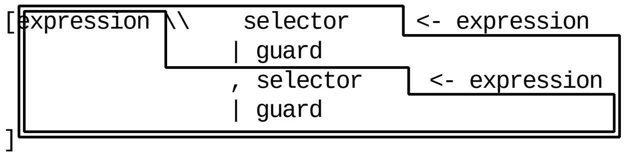
New formal parameters can also be introduced in a *case expression*. They have a meaning in the corresponding case alternative identical to the scope rules of an ordinary function definition(see also section 4.10).



```
case expression of
   args -> body
   args -> body
```

Figure (Introducing formal parameters via a case expression).

In a *list* and in an *array comprehension* new variables can be introduced when *generators* are specified. Each generator can generate a selector which can be tested in a guard and used to generate the next selector and finaly in the resulting expression (see the picture below).



```
[expression \\    selector    <- expression
               | guard
               , selector     <- expression
               | guard
]
```

Figure (Introducing selectors via generators).

3.6 Lay-out rule

As is common in modern functional languages, there is a lay-out rule in CLEAN. When the definition of the module header of a module is not ended by a semicolon a CLEAN program has become lay-out sensitive. The *lay-out rule* assumes the omission of the semi-colon (`';'`) that ends a definition and of the braces (`'{'` and `'}'`) that are used to group a list of definitions. These symbols are automatically added according to the following rules:

In lay-out sensitive mode the indentation of the first lexeme after the keywords `let`, `#`, `let!`, `#!`, `of`, `where`, or `with` determines the indentation that the group of definitions following the keyword has to obey. Depending on the indentation of the first lexeme on a subsequent line the following happens. A new definition is assumed if the lexeme starts on the same indentation (and a semicolon is inserted). A previous definition is assumed to be continued if the lexeme is indented more. The group of definitions ends (and a close brace is inserted) if the lexeme is indented less. Global definitions are assumed to start in column 0.

For reasons of portability it is assumed that a tab space is set to 4 white spaces and that a non-proportional font is used.

Example (use of lay-out rule: same example with and without using the lay-out sensitive mode).

```
primes :: [Int]
primes = sieve [2..]
where
     sieve :: [Int] -> [Int]
     sieve [pr:r]  = [pr:sieve (filter pr r)]

     filter :: Int [Int] -> [Int]
     filter pr [n:r]
     | n mod pr == 0    = filter pr r
     | otherwise        = [n:filter pr r]
```

```
primes :: [Int];
primes = sieve [2..];
where
{    sieve :: [Int] -> [Int];
     sieve [pr:r] = [pr:sieve (filter pr r)];

     filter :: Int [Int] -> [Int];
     filter pr [n:r]
     | n mod pr == 0    = filter pr r;
     | otherwise        = [n:filter pr r];
}
```

4

Expressions

In this chapter it is explained what kind of expressions can be written. In CLEAN, expressions are actually *graph expressions* which define the creation of a *(sub-) graph* (see 2.1).

4.1 Expressions

An expression generally expresses an application of a function or data constructor to its arguments (see 4.2). A case expression and conditional expression are added for notational convenience (see 4.11). With a let expression new functions and graphs can be locally defined (see 4.12). One can optionally demand the *interleaved or parallel evaluation* of the expression by another process or on another processor (see 7.1 and 10.5).

```
Graph              =  [Process] GraphExpr
GraphExpr          =  Application                          //  see 4.2
                   |  CaseExpr                             //  see 4.11
                   |  LetExpr                              //  see 4.12
```

4.2 Applications

```
Application        =  {BrackGraph}+                        //  application
                   |  GraphExpr OperatorSymbol GraphExpr   //  operator application
BrackGraph         =  NodeSymbol                           //  see 4.3
                   |  GraphVariable                        //  see 4.4
                   |  BasicValue                           //  see 4.5
                   |  List                                 //  see 4.6
                   |  Tuple                                //  see 4.7
                   |  Record                               //  see 4.8
                   |  RecordSelection                      //  see 4.8
                   |  Array                                //  see 4.9
                   |  ArraySelection                       //  see 4.9
                   |  LambdaAbstr                          //  see 4.10
                   |  (GraphExpr)                          //  see 4.2
OperatorSymbol     =  FunctionSymb
                   |  ConstructorSymb
```

A (graph) *application* or graph *expression* in principle consists of the application of a *function* or *data constructor* to its (actual) arguments. Each function or data constructor can be used in a *curried* way and can therefore be applied to any number (zero or more) of arguments (see 8.3 and 9.3). For convenience and efficiency special syntax is provided to denote values of data structures of predefined type (see 4.5 -

4.9). A function can only be rewritten if it is applied to a number of arguments equal to the arity of the function (see 6.1).

- All expressions have to be of correct type (see Chapters 8 and 9).
- All symbols that appear in an expression must have been defined somewhere within the scope in which the expression appears (see 3.5).

Operators are special functions or constructors defined with arity two (see 8.3.2) which can be applied in infix position. The *precedence* (`0` through `9`) and *fixity* (`infixl`eft, `infixr`ight, `infix`) which can be defined in the type definition of the operators (see 8.3) determine the priority of the operator application in an expression. A higher precedence binds more tightly. When operators have equal precedence, the fixity determines the priority. In an expression an ordinary function application has a very high priority (`10`). Only selection of record elements and array elements (see 4.8 and 4.9) binds more tightly (`11`). Besides that, due to the priority, brackets can sometimes be omitted, operator applications behave just like other applications (see 4.2).

- It is not allowed to apply operators with equal precedence in an expression in such a way that their fixity conflict. So, when in $a_1\ op_1\ a_2\ op_2\ a_3$ the operators op_1 and op_2 have the same precedence a conflict arises when op_1 is defined as `infixr` implying that the expression must be read as $a_1\ op_1\ (a_2\ op_2\ a_3)$ while op_2 is defined as `infixl` implying that the expression must be read as $(a_1\ op_1\ a_2)\ op_2\ a_3$.
- When an operator is used in infix position *both* arguments have to be present. Operators can be used in a *curried* way, but then they have to be used as ordinary *prefix* functions / constructors. When an operator is used as prefix function c.q. constructor, it has to be surrounded by brackets.

4.3 Node symbols

```
NodeSymbol          = FunctionSymbol
                    | ConstructorSymbol
FunctionSymbol      = FunctionSymb
                    | (FunctionSymb)
ConstructorSymbol   = ConstructorSymb
                    | (ConstructorSymb)
```

Symbols applied on zero arguments just form a syntactic unit (for non-operators no brackets needed in this case). Besides the brackets that can be omitted they behave just like other applications (see 4.2).

4.4 Graph variables

```
GraphVariable       = Variable
                    | SelectorVariable
```

There are two kinds of graph variables that can occur in a graph expression: *variables* (introduced as *formal argument* of a function, see 6.1 and 6.2) and *selector variables* (defined in a *selector* to identify parts of a graph expression, see 5.2).

- There has to be a definition for each node variable and selector variable within in the scope of the graphs expression.

4.5 Constant values of basic type

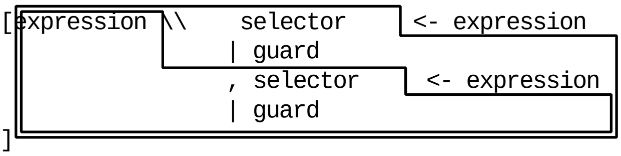
A graph expression can be a *constant value* denoting an object of predefined basic type `Int`, `Real`, `Bool` or `Char`. These predefined types introduced for reasons of efficiency and convenience are treated in Section 8.1.1. There is a special notation to denote a string (an unboxed array of characters, see 4.9) as well as to denote a list of characters (see 4.6). The denotation of constant values must obey the lexical description given in 3.2.

```
BasicValue     =    IntDenot
               |    RealDenot
               |    BoolDenot
               |    CharDenot
```

4.6 Lists and list comprehensions

For programming convenience several ways are offered to create a *list* structure including *list comprehensions* like *dot-dot expression* and *ZF-expressions* (recurrent generators are however not provided). With a *list generator* one can draw elements from a list. With an *array generator* one can draw elements from an array. One can define several generators in a row separated by a comma. The last generator in such a sequence will vary first. One can also define several generators in a row separated by a '`&`'. All generators in such a sequence will vary at the same time but the drawing of elements will stop as soon of one the generators is exhausted. This construct can be used instead of the zip-functions which are commonly used. *Selectors* are simple patterns to identify parts of a graph expression. They are explained in Section 5.3. Only those lists produced by a generator which match the specified selector are taken into account. Guards can be used as filter in the usual way.

The scope of the selector variables introduced on the left-hand side of a generator is such that the variables can be used in the guards and other generators that follow. All variables introduced in this way can be used in the expression before the `\\` (see the picture below).



```
[expression \\    selector   <- expression
                | guard
                , selector    <- expression
                | guard
]
```

A special notation is provided for the frequently used *list of characters* (see also 3.2). The predefined *type* list is treated in Section 8.1.3.

```
List            =  [[{LGraphExpr}-list [: GraphExpr]]]
                |  [GraphExpr [,GraphExpr]..[GraphExpr]]
                |  [GraphExpr \\ {Qualifier}-list]
LGraphExpr      =  GraphExpr
                |  CharsDenot
Qualifier       =  Generators {|Guard}
Generators      =  {Generator}-list
                |  Generator {& Generator}
Generator       =  Selector <- ListExpr
                |  Selector <-: ArrayExpr
Selector        =  BrackPattern                  // for brack patterns see 6.2
ListExpr        =  GraphExpr
ArrayExpr       =  GraphExpr
Guard           =  BooleanExpr
BooleanExpr     =  GraphExpr
```

- A list expression must be of type list.
- A guard must be of type `Bool`.
- Dot-dot expressions are predefined on objects of type `Int`, `Real` and `Char`, but dot-dots can also be applied to any user defined data structure for which the class enumeration type has been instantiated (see `StdClass`).

Example (various ways to define a list with the integer elements `1,3,5,7,9`).

```
[1:[3:[5:[7:[9:[]]]]]]
[1,3,5,7,9]
[1,3..9]
[1:[3,5,7,9]]
[1,3,5:[7,9]]
[n \\ n <- [1..10] | n mod 2 <> 0]
```

Example (ZF-expression: `expr1` yields `[(0,0), (0,1), (0,2), (1,0), (1,1), (1,2), (2,0), (2,1), (2,2), (3,0), (3,1), (3,2)]` while `expr2` yields `[(0,0), (1,1), (2,2)]`). `expr3` yields `[(0,0), (1,0), (1,1), (2,0), (2,1), (2,2), (3,0), (3,1), (3,2), (3,3)]`

```
expr1 = [(x,y) \\ x <- [0..3] , y <- [0..2]]
expr2 = [(x,y) \\ x <- [0..3] & y <- [0..2]]
expr3 = [(x,y) \\ x <- [0..3] , y <- [0..x]]
```

Example (ZF-expression: a well-know sort).

```
sort :: [a] -> [a] | Ord a
sort []       = []
sort [p:ps]   = sort [x\\x<-ps|x<=p] ++ [p] ++ sort [x\\x<-ps|x>p]
```

Example (ZF-expression: converting an array into a list).

```
ArrayA = {1,2,3,4,5}

ListA = [a \\ a <-: ArrayA]
```

Example (various ways to define a list with the characters `'a'`, `'b'` and `'c'`).

```
['a':['b':['c':[]]]]
['a','b','c']
['a'..'c']
['abc']
['ab','c']
```

4.7 Tuples

Tuples can be created that can be used to combine different (sub-)graphs into one data structure without being forced to define a new type for this combination. The elements of a tuple need *not* be of the same type. Tuples are in particular handy for functions that return multiple results. The predefined *type* tuple is treated in Section 8.1.4.

Tuple = **(**GraphExpr**,**{GraphExpr}-*list***)**

Example (tuple).

```
("this is a tuple with",3,['elements'])
```

4.8 Records, record selection and record update

A *record* is a tuple-like algebraic data structure that has the advantage that its elements can be selected by *field name* rather then by position. An update operator for records is provided. The type of a record has to be specified explicitly and curried use is *not* possible (see 8.2).

Record = **{**[*TypeSymb* **|**][RecordExpr **&**][{FieldSymbol **=** GraphExpr}-*list*]**}**
RecordExpr = GraphExpr
RecordSelection = RecordExpr**.**[*TypeSymb* **.**]*FieldSymb*

A new record can be *created* in the two ways. The first way is to *explicitly* define a value for *each* of the fields. The order in which the record fields are specified is irrelevant, but *all* fields have to get a value.

The second way is to construct a new record out of an existing one (a *functional record update*). The record written to the left of the **&** (`r & f = v` is pronounced as: `r` updated with for `f` the value `v`) is the record which is used as blueprint which is of the same type as the new record to be constructed. On the right from the `&` the fields are specified in which the new record *differs* from the old one. The other fields are created *implicitly*. Notice that the functional update is not an update in the classical, destructive sense since a *new* record is created. The functional update of records is performed very efficient such that we have not added support for destructive updates of records of unique type.

With a *record selection* one can select the value stored in the indicated field. Record selection binds more tightly (priority `11`) than application (priority `10`).

- The record expression must yield an object of a record type.
- The type of the record must have been defined (see 8.2.2).
- The field names used in the expression must be the same as the field names defined in the type definition of the record, their types must be an instantiation of the corresponding types.
- When a record is created, *all* fields have to get a value (either implicitly or explicitly). This implies that records cannot be used in a curried way.

- The type symbol of the record being created can only be left out if there is at least one field name is specified which is not being defined in some other record.

Example (record type definition, record creation, selection and update of a record).

```
::Person =     { name        :: String               //   record type definition
               , address     :: String
               , city        :: String
               , cleanuser   :: Bool
               }

SomePerson :: Person                                 //   function creating a new record
SomePerson =   { name        = "Some Body"
               , address     = "Somewhere 17"
               , city        = "Sometown"
               , cleanuser   = False
               }

GetName :: Person -> String                          //   selection of a record fileld
GetName someone = someone.name

SetUser :: Person -> Person                          //   function updating a record
SetUser someone = { someone & cleanuser = True }
```

4.9 Arrays, array selection and array update

An *array* is a tuple/record-like data structure in which *all* elements are of the *same* type. Instead of selection by position or field name the elements of an array can be selected very efficiently in *constant time* by indexing. The update of arrays is done destructively in CLEAN and therefore arrays have to be unique (see Chapter 9) if one wants to use this feature. Arrays are very useful if time and space consumption is becoming very critical (CLEAN arrays are implemented very efficiently). If efficiency is not a big issue we recomment not to use arrays but to use lists instead: lists induce a much better programming style. Lists are more flexible and less error prone: array elements can only be accessed via indices and if you make a calculation error indices may point outside the array bounds. This is detected, but only at run-time. In CLEAN, array indices always start with 0. More dimensional arrays (e.g. a matrix) can be defined as an array of arrays.

For efficiency reasons, arrays are available in several ways: there are *lazy arrays* (type `{a}`), *strict arrays* (type `{!a}`) and *unboxed arrays* for elements of basic type (e.g. type `{#Int}`). All these arrays are considered to be of *different* type. By using the overloading mechanism (type constructor classes) one can still define (overloaded) functions which work on any array. The predefined *type* array is treated in Section 8.1.2.

Array = **{**{GraphExpr}-*list***}**
| **{**ArrayExpr **&** [{ArrayIndex **=** GraphExpr}-*list*] [\\ {Qualifier}-*list*]**}**
| **{**[ArrayExpr **&**] GraphExpr \\ {Qualifier}-*list*}
| *StringDenot*
ArrayExpr = GraphExpr
ArrayIndex = **[**{IntegerExpr}-*list***]**
IntegerExpr = GraphExpr
ArraySelection = ArrayExpr**.**ArrayIndex

A new array can be created in a number of ways. One way is to simply *list* the *array elements*. By default a *lazy* array will be created. Arrays are *unique* (the `*` or . attribute in front of the type, see Chapter 9) to make destructive updates possible.

A lazy array is a box with pointers pointing to the array elements. One can also create a strict array (explicitly define its type as {!Int}), which will have the property that the elements to which the array box points will always be evaluated. One can also create an unboxed array (explicitly define its type as {#Int}), which will have the property that the evaluated elements (which have to be of basic value) are stored directly in the array box itself. Clearly the last one is the most efficient representation (see also Chapter 13).

Example (creating a lazy array, strict and unboxed unique array of integers with elements `1,3,5,7,9)`.

```
MyLazyArray :: .{Int}
MyLazyArray = {1,3,5,7,9}

MyStrictArray :: .{!Int}
MyStrictArray = {1,3,5,7,9}

MyUnboxedArray :: .{#Int}
MyUnboxedArray = {1,3,5,7,9}
```

Example (creating a two dimensional array, in this case a unique array of unique arrays of unboxed integers) .

```
MatrixA :: .{.{#Int}}
MatrixA = {{1,2,3,4},{5,6,7,8}}
```

To make it possible to use operators such as array selection on any of these arrays (of actually different type) a type constructor class has been defined (in `StdArray`) which expresses that "some kind of array structure is created". The compiler will therefore deduce the following general type:

```
Array :: .(a Int) | Array a
Array = {1,3,5,7,9}
```

Important: We are currently making the overloading system more powerful. We hope that this will make it possible in the future to express e.g. overloaded operators on all kinds of arrays in a more convenient way.

There are a number of handy functions for the creation and manipulation of arrays predefined in `StdArray` (see Appendix B). These functions are overloaded (see `StdArray`) to be able to deal with any type of array. The class restrictions for these functions express that "an array structure is required" containing "an array element".

Example (type of some predefined functions in `StdArray`).

```
createArray    :: !Int e -> .(a e) | Array a & ArrayElem e      // size arg1, a.[i] = arg2
size           :: (a e) -> Int     | Array a & ArrayElem e      // number of elements in array
```

Finally one can construct a new array out of an existing one (a *functional array update*). Left from the `&` (`a & [i] = v` is pronounced as: array `a` updated with for `a.[i]` the value `v`) the old array has to be specified which has to be of unique type to make destructive updating possible. On the right from the `&` those array elements are listed in which the new array differs from the old one. The `&`-operator is strict in its arguments.

Important: For reasons of efficiency we have defined the updates only on arrays which are of *unique* type (`*{…}`), such that the update can always be done *destructively* (!) which is semantically sound because the original unique array is known not to be used anymore (see 9.2).

Example (creating an array with the integer elements `1,3,5,7,9` using the update operator) .

```
{CreateArray 5 0 & [0] = 1, [1] = 3, [2] = 5, [3] = 7, [4] = 9}
{CreateArray 5 0 & [1] = 3, [0] = 1, [3] = 7, [4] = 9, [2] = 5}
```

One can use an *array comprehension* or a *list comprehension* (see 4.6) to list these elements compactly in the same spirit as with an list comprehension.

Array comprehensions can be used in combination with the update operator. Used in combination with the update operator the original uniquely typed array is updated destructively. The combination of array comprehensions and update operator makes it possible to selectively update array elements on a high level of abstraction.

Example (creating an array with the integer elements `1,3,5,7,9` using the update operator in combination with array and list comprehensions) .

```
{CreateArray 5 0 & [i] = 2*i+1 \\ i <- [0..4]}
{CreateArray 5 0 & [i] = elem \\ elem <-: {1,3,5,7,9} & i <- [0..4]}
```

```
{CreateArray 5 0 & elem \\ elem <-: {1,3,5,7,9}}
```

Array comprehensions used without update operator automatically generate a whole new array. The size of this new array will be equal to the size of the first array or list generator from which elements are drawn. Drawn elements which are rejected by a corresponding guard result in an undefined array element on the corresponding position.

Example (creating an array with the integer elements `1,3,5,7,9` using array and list comprehensions) .

```
{elem \\ elem <-: {1,3,5,7,9}}
{elem \\ elem <-  [1,3,5,7,9]}
```

A *string* is equivalent to an *unboxed array of character* `{#Char}`. A type synonym is defined in module `StdString`. Notice that this array is *not* unique, such that a destructive update of a string is *not* allowed. There is special syntax to denote strings (see 3.2).

Example (some ways to define a string, i.e. an unboxed array of character).

```
"abc"
{'a','b','c'}
```

With an *array selection* one can select an array element. When an object `a` is of type `Array`, the ith element can be selected (computed) via `a.[i]`. Array selection is left-associative: `a.[i,j,k]` means `((a.[i]).[j]).[k]`. Array selection binds more tightly (priority `11`) than application (priority `10`).

- An array expression must be of type array.
- All elements of an array need to be of same type.
- The array expression on the left of the dot '`.`' and to the left of the update operator '`&`' should yield an object of type unique array.
- An array index must be an integer value between `0` and the number of elements of the array-`1`. An index out of this range will result in a *run-time* error.
- *A unique array of any type created by an overloaded function cannot be converted to a non-unique array.*

Example (array creation, selection, update). The most general types have been defined. One can of course always restrict to a more specific type.

```
MkArray :: !Int (Int -> e) -> .(a e) | Array a & ArrayElem e
MkArray i f = {f j \\ j <- [0..i-1]}

SetArray :: *(a e) Int e -> .(a e) | Array a & ArrayElem e
SetArray a i v = {a & [i] = v}

CA :: Int e -> .(a e) | Array a & ArrayElem e
CA i e = createArray i e

InvPerm :: {Int} -> .{Int}
InvPerm a = {CA (size a) 0 & [a.[i]] = i \\ i <- [0..maxindex a]}

ScaleArray :: e (a e) -> .(a e) | Array a & ArrayElem e & Arith e
ScaleArray x a = {x * e \\ e <-: a}

MapArray:: (a -> b) (ar a) -> .(ar b) | Array ar & ArrayElem a & ArrayElem b
MapArray f a = {f e \\ e <-: a}

inner :: (a e) (a e) -> .(a e) | Array a & ArrayElem e & Arith e
inner v w
| size v == size w = {vi * wi \\ vi <-: v & wi <-: w}
| otherwise        = abort "cannot take inner product"

ToArray :: [e] -> .(a e) | Array a & ArrayElem e
ToArray list = {e \\ e <- list}

ToList :: (a e) -> .[e] | Array a & ArrayElem e
ToList array = [e \\ e <-: array]
```

Example (of operations on 2 dimensional arrays generating new arrays).

```
maxindex n :== size n - 1

Adj:: {{#Int}} -> .{.{#Int}}
Adj ma =  {  {ma.[i,j] \\ i <- rowindex}
          \\ j <- colindex
          }
where
     rowindex = [0..maxindex ma]
     colindex = [0..maxindex ma.[0]]

Multiply:: {{#Int}} {{#Int}} -> .{.{#Int}}
Multiply a b =     {  {sum [a.[i,j]*b.[j,k] \\ j <- colindex] \\ k <- rowindex}
                   \\ i <- rowindex
                   }
where
     rowindex = [0..maxindex a]
     colindex = [0..maxindex a.[0]]
```

4.10 Lambda abstraction

Sometimes it can be convenient to define a tiny function in an expression "right on the spot". For this purpose one can use a *lambda abstraction*. An anonymous function is defined which can have several formal arguments which can be patterns as common in ordinary function definitions (see Chapter 6). However, only simple functions can be defined in this way: no guards, no rule alternatives, no local definitions. Since the dot is already used for record and array selection a '->' is ued to separate the formal arguments from the functionbody:

LambdaAbstr = \ {Pattern} **->** GraphExpr

A lambda expression introduces a new scope (see section 3.5).

```
\ args -> body
```

Example (lambda expression).

```
AddTupleList :: [(Int,Int)] -> [Int]
AddTupleList list = map (\(x,y) -> x+y) list
```

4.11 Case expression and conditional expression

For programming convenience a *case expression* and *conditional expression* are added.

CaseExpr = **case** GraphExpr **of**
{ {CaseAltDef}+ }
| **if** BrackGraph BrackGraph BrackGraph
CaseAltDef = {Pattern}
{LetBeforeExpression}
{{**|** Guard} **->** FunctionBody}+
[LocalFunctionAltDefs]

In a *case expression* first the discriminating expression is evaluated after which the case alternatives are tried in textual order. Case alternatives are similar to functionalternatives. This is not so strange because a case expression is internally translated to a function definition (see the example below). Each alternative contains a left-hand side pattern (see 6.2) which is optionally followed by a *let-before* (see 6.?) and a guard (see 6.3). When a pattern matches and the optional guard evaluates to `True` the corresponding alternative is chosen. A new block structure (scope) is created for each case alternative (see 3.5).

```
case expression of
  args -> body
  args -> body
```

- All alternatives in the case expression must be of the same type.
- When none of the patterns matches a *run-time* error is generated.

Example (case expression).

```
h x = case g x of
          [hd:_]    -> hd
          []        -> abort "result of call g x in h is empty"
```

is semantically equivalent to:

```
h x = mycase (g x)
where
     mycase  [hd:_]     = hd
     mycase  []         = abort "result of call g x in h is empty"
```

In a *conditional expression* first the Boolean expression is evaluated after which either the then- or the else-part is chosen. The conditional expression can be seen as a simple kind of case expression.

- The then- and else-part in the conditional expression must be of the same type.
- The discriminating expression must be of type `Bool`.

4.12 Let expression

Sometimes it is convenient to introduce local function definition or constant (graph) definitions (see also Chapter 6) which are only visible for a certain expression. So, a *let* statement introduces a new scope (see 3.5).

```
let
    function args = body
    selector = expression
in expression
```

Such local definitions can be introduced using a let expression with the following syntax.

LetExpresssion = **let** { {LocalDef}+ } **in** GraphExpr

Example (let expression used in a list comprehension).

```
doublefibs n = [let a = fib i in (a, a) \\ i <- [0..n]]
```

5

Defining constants

One can give a name to a constant expression (actually a graph), such that the expression can be used in (and shared by) other expressions. Constant graphs can be defined on a global level or locally in a function (see 5.1). See also CLEAN's basic semantics in Chapter 2. One can also identify certain parts of a constant via a projection function called a selector (see 5.2).

5.1 Constant graph definitions

```
GraphDef            =   Selector =[:] GraphExpr ;
```

When a *graph* is *defined* a name is given to (part) of an expression (see Chapter 4). The definition of a graph can be compared with a definition of a *constant* (data) or a *constant* (projection) *function*. However, notice that graphs are constructed according to the basic semantics of CLEAN (see Chapter 2) which means that multiple references to the same graph will result in *sharing* of that graph. Recursive references will result in *cyclic graph structures*. Graphs have the property that they *are computed only once* and that their value is *remembered* within the scope they are defined in.

Graph definitions differ from constant function definitions (see 6.1). *Constant function definitions* are a special form of a graph rewriting rule: multiple references to a function just means that the same definition is used such that a (constant) function *will be recomputed again for each occurrence of the function symbol made*.

Syntactically the definition of a graph is distinguished from the definition of a function by the symbol which separates left-hand side from right-hand side: "**=:**" is used for graphs while "**=>**" is used for functions. However, in general the more common symbol "**=**" is used for both type of definitions. Generally it is clear from the context what is meant (functions have parameters). However, constant definitions are ambiguous (they can be a constant function definition as well as a constant graph definition). Locally (i.e. within a function definition, see Chapter 6) constant definitions are *by default* taken to be *graph* definitions (see also 5.1.1) and therefore shared, globally they are *by default* taken to be *function* definitions (see 6.1) and therefore recomputed. If one wants to obtain a different behaviour one has to explicit state the nature of the constant definition (has it to be shared or not) by using "=:" (on the global level) or "=>" (on the local level).

Example (Graph versus constant function definition: `biglist1` is a *graph* which is computed only once, `biglist2` is a constant *function* which is computed every time it is applied).

```
biglist1 =:   [1..10000]                    //   a graph
biglist2 =>   [1..10000]                    //   a constant function
```

5.1.1 Defining graphs in functions

A constant (sub-) graph can be defined locally at several places in a function: one can name any *pattern* (see 6.2), graphs can be defined in any (*strict*) *let* statement, *where* or *with* block (see Chapter 6). Graphs defined locally will be collected by the garbage collector when they are no longer connected to the root of the program graph (see Chapter 2).

Example (graph locally defined in a function: the graph labelled `last` is shared in the function `StripNewline` and computed only once).

```
StripNewline :: String -> String
StripNewline "" = ""
StripNewline string
| string !! last<>'\n'  = string
| otherwise             = string%(0,last-1)
where
     last = maxindex string
```

Example (the Hamming numbers defined using a locally defined cyclic constant graph and defined by using a recursive constant function. The first definition (ham1) is efficient because already computed numbers are reused via sharing. The second definition (ham2) is much more inefficient because the recursive function recomputes everything).

```
ham1 :: [Int]
ham1 = y
where y = [1:merge (map ((*) 2) y) (merge (map ((*) 3) y) (map ((*) 5) y))]


ham2 :: [Int]
ham2 = [1:merge (map ((*) 2) ham2) (merge (map ((*) 3) ham2) (map ((*) 5) ham2 ))]
```

5.1.2 Defining graphs on the global level

Graphs can also be defined on a global level.

```
Definition          =   …
                    |   GraphDef
```

A *global graph definition* defines a global constant (closed) graph, i.e. a graph which has the same scope as a global function definition. The selector variables that occur in the selectors of a global graph definition have a global scope just as globally defined functions.

Special about *global* graphs (in contrast with *local* graphs) is that they are *not* garbage collected during the evaluation of the program. A global graph can be compared with a *CAF* (*constant applicative form*): its value is computed at most once and remembered at run-time. A global graph can save execution-time at the cost of permanent space consumption (see Chapter 13).

5.2 Selectors

A *selector* is a pattern which introduces one or more new *selector variables* implicitly defining *projection functions* to identify (parts of) a graph being defined on a local or global level. One can identify the sub-graph as a whole or one can identify its components. A selector can contain constants (also user defined constants introduced by algebraic type definitions), variables and wildcards. With a *wildcard* one can indicate that one is not interested in certain components.

```
Selector            =   BrackPattern // for bracket patterns see 6.2
```

- When a selector on the left-hand side of a graph definition is not matching the graph on the right-hand side it will result in a *run-time* error.
- The selector variables introduced in the selector must be different from each other and not already be used in the same scope and name space (see 3.5 and 3.4).
- To avoid the specification of patterns which may fail at run-time, it is not allowed to test on zero arity constructors. For instance, list used in a selector pattern need to be of form `[a:_]`. `[a]` cannot

be used because it stands for `[a:[]]` implying a test on the zero arity constructor `[]`. If the pattern is a record only those fields which contents one is interested in need to be in dicated in the pattern.

- Arrays cannot be used as pattern in a selector.

Remark: a selector can also appear on the left-hand side of a generator in a list comprehension (see 4.6) or array comprehension (see 4.8).

Example (use of selectors to select record elements).

```
:: Complex = {re :: Real, im :: Real}

RePart:: Complex -> Real
RePart c = r
where
     {re = r} = c
```

6

Defining functions

In this section *function definitions* are treated (actually: *graph rewrite rules*). *Operator* definitions are regarded as special kind of function definitions (see 6.1 and 8.3). A function can be preceded by a definition of its type (see Chapter 8). The body of a function consists of a root expression(see 6.4). With help of patterns (see 6.2) and guards (see 6.3) a distinction can be made between several alternative definitions for a function. Functions and graphs can be defined locally in a function definition (see 6.5). For programming convenience (forcing evaluation, observation of unique objects and threading of sequencial operations) special let constructions are provided (see 6.6).

6.1 Defining functions

```
FunctionDef        =  [FunctionTypeDef] DefOfFunction     //  see Chapter 8 for typing functions
DefOfFunction      =  {FunctionAltDef}+
FunctionAltDef     =  FunctionSymbol {Pattern}            //  see 6.2 for patterns
                      {LetBeforeExpression}               //  see 6.6
                      {{| Guard} =[>] FunctionBody}+      //  see 6.3 for guards
                      [LocalFunctionAltDefs]              //  see 6.5
FunctionSymbol     =  FunctionSymb                        //  ordinary function
                   |  (FunctionSymb)                      //  operator function
FunctionBody       =  [StrictLet]                         //  see 6.6
                      RootExpression ;                    //  see 6.4
                      [LocalFunctionDefs]                 //  see 6.5
```

A *function definition* consist of one or more definitions of *function alternatives* (rewrite rules) which are tried in textual order. On the left-hand side of such a function alternative a *pattern* can be specified which can serve a whole sequence of *guarded function bodies* (called the *rule alternatives*). The root expression (see 6.4) of a particular rule alternative is chosen for evaluation when

+ the pattern on the left-hand side matches the corresponding actual arguments of the function application (see 6.2) *and*
+ the optional *guard* (see 6.3) specified on the right-hand side evaluates to `True`.

- Function definitions are only allowed in implementation modules (see 12.1).
- It is required that the function alternatives are textually grouped together (separated by semi-colons when the lay-out dependent mode is not chosen).
- Each alternative of a function must start with the same function symbol.
- The function name must in principle be different from other names in the same name space and same scope (see 3.4). However, it is possible to overload functions and operators (see 8.4).
- A function has a fixed arity, so in each rule the same number of formal arguments must be specified. Functions can be applied to any number of arguments though, as usual in higher order functional languages (see 4.2 and 9.3).

Example (function definition).

```
module example                                              //    module header
import StdInt                                               //    implicit import

map :: (a -> b) [a] -> [b]                                  //    definition of the function map
map f list = [f e \\ e <- list]

square :: Int -> Int                                        //    definition of the function square
square x = x * x

Start :: [Int]                                              //    definition of the Start rule
Start = map square [1..1000]
```

Constant definitions on the global level are *by default* taken to be function definitions (see 5.1.2). By using "`=:`" instead of "`=`" one can indicate that a *constant graph* (CAF) is defined instead of a function.

An *operator* is a *function with arity two* which can be used as infix operator (brackets are left out) or as ordinary prefix function (the operator name preceding its arguments has to be surrounded by brackets).

- When an operator is used in infix position *both* arguments have to be present. Operators can be used in a curried way, but then they have to be used as ordinary prefix functions (see also 4.3).

Example (operator definition).

```
(++) infixr 0 :: [a] [a] -> [a]
(++) []       ly   =    ly
(++) [x:xs]   ly   = [x:xs ++ ly]

(o) infixr 9 :: (a -> b) (c -> a) -> (c -> b)
(o) f g = \x -> f (g x)
```

An operator has a precedence (`0` through `9`, default `9`) and a fixity (`infixl`, `infixr` or just `infix`, default `infixl`). This is defined in its type (see 8.3.2). See also Section 4.3.

6.2 Pattern matching

In this section the different kind of *formal arguments* (patterns) that can be specified on the left-hand side of a function definition (rewrite rule definition) are described. A *pattern* generally consists of some *data constructor* with its optional arguments which on their turn can contain sub-patterns (see 6.2.1). A *node-id variable* can be attached to a pattern which makes it possible to identify (*label*) the whole pattern as well as its contents. *Bracketed patterns* are formal arguments that form a syntactic unit (see 6.2.2 - 6.2.6).

```
Pattern         = [Variable =:] BrackPattern
BrackPattern    = ConstructorSymbol                 // see 6.2.2
                | PatternVariable                   // see 6.2.3
                | BasicValuePattern                 // see 6.2.4
                | ListPattern                       // see 6.2.5
                | TuplePattern                      // see 6.2.6
                | RecordPattern                     // see 6.2.7
                | ArrayPattern                      // see 6.2.8
                | (GraphPattern)                    // see 6.2.1
```

- It is possible that the specified patterns turn a function into a partial function (see 8.3.3). When a partial function is applied outside the domain for which the function is defined it will result into a *run-time* error. A compile time *warning* is generated that such a situation might arise.

6.2.1 Constructor patterns

```
GraphPattern    = ConstructorSymbol {Pattern}                   // Constructor pattern
                | GraphPattern ConstructorSymb GraphPattern     // Constructor operator
                | Pattern                                       // a pattern in brackets
```

A *constructor pattern* (see above) consists of a *data constructor* (see 8.2.1) with its optional arguments which on its turn can contain *sub-patterns*. A constructor pattern forces evaluation of the corresponding actual argument to strong root normal form since the strategy has to determine whether the actual argument indeed is equal to the specified constructor.

- the data constructor must have been defined in an algebraic data type definition (see 8.2.1).

Example (algebraic data type definition and constructor pattern in function definition).

```
::Tree a  = Node a (Tree a) (Tree a)
          | Nil

Mirror :: (Tree a) -> Tree a
Mirror (Node e left right)   = Node e (Mirror right) (Mirror left)
Mirror Nil                   = Nil
```

Constructors with arity two (see 6.1, see 8.2.1) can also be defined as *infix constructors* (or *constructor operator*). In a pattern match they can be written down in infix position as well .

- When a constructor operator is used in infix position in a pattern match *both* arguments have to be present. Constructor operators can occur in a curried way, but then they have to be used as ordinary prefix constructors (see also 6.2.1 and 4.3).

Example (algebraic type definition and constructor pattern in function definition).

```
::Tree2 a     = (/\) infixl 0 (Tree a) (Tree a)
              | Value a

Mirror :: (Tree2 a) -> Tree2 a
Mirror (left/\right)    = Mirror right/\Mirror left
Mirror leaf             = leaf
```

6.2.2 Simple Constructor patterns

ConstructorSymbol = *ConstructorSymb*
| **(***ConstructorSymb***)**

Constructor symbols without arguments just form a syntactic unit (for non-operators no brackets needed in this case). Besides the brackets that can be omitted they behave just like other constructor patterns (see 6.2.1).

6.2.3 Variables and wildcards in patterns

A *pattern variable* can be a (node) *variable* or a *wildcard*.

PatternVariable = *Variable*
| _

A *node variable* matches on *any* concrete value of the corresponding actual argument and therefore it does *not* force evaluation of this argument. A *wildcard* is an *anonymous* node variable ("_") one can use to indicate that the corresponding argument is not used in the right-hand side of the rewrite rule. All lower case identifiers in a graph pattern are (node) variables. The formal arguments of a function and the function body are contained in a new scope.

`function`$\boxed{\texttt{args = body}}$

- All variable symbols introduced at the left-hand side of a function definition must have different names.

Example (use of pattern variables).

```
:: Complex :== (!Real,!Real)                                 //    synonym type def

realpart :: Complex -> Real
realpart (re,_) = re
```

6.2.4 Constant values of basic type as pattern

BasicValuePattern = BasicValue

A *constant value* of predefined *basic type* `Int`, `Real`, `Bool`or `Char` (see 8.1) can be specified as pattern.

- The denotation of such a value must obey the syntactic description given in 3.2.

Example (use of basic values as pattern).

```
nfib :: Int -> Int
nfib 0 = 1
nfib 1 = 1
nfib n = 1 + nfib (n-1) * nfib (n-2)
```

6.2.5 List patterns

An object of the predefined algebraic type *list* (see 8.1.3) can be specified as pattern.

ListPattern = **[**[{LGraphPattern}-*list* [**:** GraphPattern]]**]**
LGraphPattern = GraphPattern
| *CharsDenot*

Notice that only simple list patterns can be specified on the left-hand side (one cannot use a dot-dot expression or list comprehension to define a list pattern).

Example (use of list patterns, use of guards, use of variables to identify patterns and sub-patterns; `merge` merges two (sorted) lists into one (sorted) list).

```
merge :: [Int] [Int] -> [Int]
merge f []     = f
merge [] s     = s
merge f=:[x:xs] s=:[y:ys]
| x<y          = [x:merge xs s]
| x==y         = merge f ys
| otherwise    = [y:merge f ys]
```

6.2.6 Tuple patterns

An object of the predefined algebraic type *tuple* (see 8.1.4) can be specified as pattern.

TuplePattern = **(**GraphPattern**,**{GraphPattern}-*list*)

6.2.7 Record patterns

An object of type *record* (see 8.2.2) can be specified as pattern. Only those fields which contents one would like to use in the right-hand side need to be mentioned in the pattern.

RecordPattern = **{**[*TypeSymb* **|**] {FieldSymbol [**=** GraphPattern]}-*list* }

- The type of the record must have been defined in a record type definition (see 8.2.2).
- The field names specified in the pattern must be identical to the field names specified in the corresponding type.
- The type of the record need not to be given if at least one of the field names specified in the pattern unambiguously identifies the type of the record being used.

Example (use of record patterns).

```
::Tree a      =    Node (RecTree a)
              |    Leaf a
::RecTree a   =    { elem   :: a
                   , left   :: Tree a
                   , right  :: Tree a
                   }

Mirror :: (Tree a) -> Tree a
Mirror (Node tree=:{left=l,right=r})  = Node {tree & left=r,right=l}
Mirror leaf                          = leaf
```

Example (the first alternative of function `Mirror` defined in another equivalent way).

```
Mirror (Node tree) = Node {tree & left=tree.right,right=tree.left}
```
or
```
Mirror (Node tree=:{left,right}) = Node {tree & left=right,right=left}
```

6.2.8 Array patterns

An object of type *array* (see 8.1.5) can be specified as pattern. Notice that only simple array patterns can be specified on the left-hand side (one cannot use array comprehensions). Only those array elements which contents one would like to use in the right-hand side need to be mentioned in the pattern.

ArrayPattern = **{**{GraphPattern}-*list***}**
| **{**{ArrayIndex **=** *Variable*}-*list***}**
| *StringDenot*

- All array elements of an array need to be of same type.
- An array index must be an integer value between `0` and the number of elements of the array-`1`. Accessing an array with an index out of this range will result in a *run-time* error.

It is allowed in the pattern to use an index expression in terms of the other formal arguments (of type `Int`) passed to the function to make a flexible array access possible.

Example (use of array patterns).

```
Swap :: !Int !Int !*(a e) -> .(a e) | Array a & ArrayElem e
Swap i j a=:{[i]=ai,[j]=aj} = {a & [i]=aj,[j]=ai}
```

6.3 Guards

Guard = BooleanExpr

A *guard* is a Boolean expression attached to a rule alternative that can be regarded as generalisation of the pattern matching mechanism: the alternative only matches when the patterns defined on the left hand-side match *and* its (optional) guard evaluates to `True` (see 6.1). Otherwise the *next* alternative is tried. Pattern matching always takes place *before* the guards are evaluated.

The guards are tried in *textual order*. The alternative corresponding to the first guard that yields `True` will be evaluated. A right-hand side without a guard can be regarded to have a guard that always evaluates to `True` (the ‘otherwise’ or ‘default’ case). In `StdBool` *otherwise* is predefined as synonym for `True` for people who like to emphasize the default option.

- Only the last rule alternative of a function alternative can have no guard.
- It is possible that the guards turn the function into a partial function (see 8.3.3). When a partial function is applied outside the domain for which the function is defined it will result into a *run-time* error. At compile time this cannot be detected.

Example (function definition with guards).

```
filter :: Int [Int] -> [Int]
filter pr [n:str]
| n mod pr == 0    = filter pr str
                   = [n:filter pr str]
```

Example (equivalent definition).

```
filter :: Int [Int] -> [Int]
filter pr [n:str]
| n mod pr == 0    = filter pr str
| otherwise        = [n:filter pr str]
```

6.4 Function body

The main body of a function is called the *root expression*. It is either a variable (in the case of a *redirection*) or a *graph expression* (in the case a contractum graph is constructed) (see CLEAN's basic semantics in Chapter 2).

```
RootExpression          =   GraphExpr
```

Example (y is the root expression referring to a cyclic graph).

```
ham :: [Int]
ham = y
where y = [1:merge (map ((*) 2) y) (merge (map ((*) 3) y) (map ((*) 5) y))]
```

6.5 Local definitions

In a function definition one can locally define graphs and functions. One way to do this is by using a *let* expression (see 4.12) with which one can introduce graphs and functions before they are used in an expression.

Example (use of a let expression).

```
ham :: [Int]
ham =    let y = [1:merge (map ((*) 2) y) (merge (map ((*) 3) y) (map ((*) 5) y))]
         in y
```

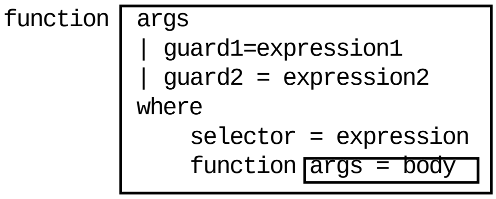
The *let* expression has the disadvantage that the scope of the new definitions is restricted to one specific expression. Sometimes one would like to define *local definitions* with a wider scope. With a *where* block (see 6.5.1) one can define functions and graphs which have meaning within every expression appearing in a *function alternative*. One can also restrict the scope of definitions to the body of one *rule alternative* by using a *with* statement (see 6.5.2).

6.5.1 Where block

At the end of each function alternative one can locally define functions and graphs in a *where block*.

```
LocalFunctionAltDefs    =   [where] { {LocalDef}+ }
LocalDef                =   GraphDef
                        |   FunctionDef
```

Functions and graphs (selectors) defined in a *where* block can be used anywhere in the corresponding function alternative (i.e. in all guards and rule alternatives following a pattern, see 6.1) as indicated in the following picture showing the scope of a *where* block.



```
function args
         | guard1=expression1
         | guard2 = expression2
         where
             selector = expression
             function args = body
```

Example (local functions defined in a where block).

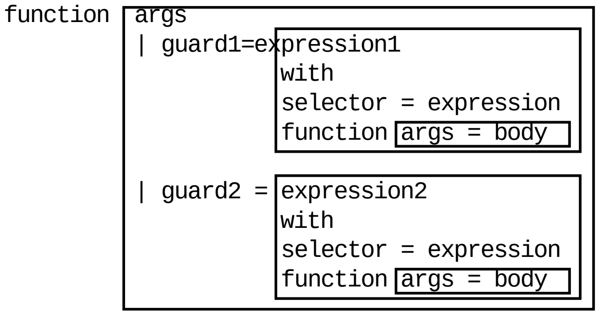
```
primes :: [Int]
primes = sieve [2..]
where
     sieve :: [Int] -> [Int]                          //   local  function def
     sieve [pr:r]   = [pr:sieve (filter pr r)]

     filter :: Int [Int] -> [Int]                     //   local  function def
     filter pr [n:r]
     | n mod pr == 0    = filter pr r
     | otherwise        = [n:filter pr r]
```

6.5.2 With block

One can also locally define functions and graphs at the end of each guarded rule alternative using a *with block*.

```
LocalFunctionDefs   =  [with] { {LocalDef}+ }
LocalDef            =  GraphDef
                    |  FunctionDef
```

Functions and graphs (selectors) defined in a *with* block can only be used in the corresponding rule alternative as indicated in the following picture showing the scope of a *with* block.



```
function args
         | guard1=expression1
                  with
                  selector = expression
                  function args = body

         | guard2 = expression2
                  with
                  selector = expression
                  function args = body
```

6.6 Special let constructions

In addition to ordinary *let* expressions there are also special *let* expressions with which one can locally define graphs (but not functions). These special let expressions are introduced for very specific reasons.

6.6.1 Strict let expression

Although CLEAN is by default a lazy language one can force evaluation in several ways. By forcing evaluation one generally obtains a more time- and space-efficient program (see 13.3). Forcing evaluation can influence the termination behaviour of the program (a terminating program may be turned into a non-terminating program). See also Section 8.5.

The nicest way to force evaluation is by defining (partially) strict data structures (see 8.5). But it can also be handy to force evaluation on ad-hoc basis. This can be done by annotating function arguments as being strict (see 8.5.2). Another way to force evaluation is by using a *strict let expression*. The *strict let* expression looks similar to an ordinary *let* expression albeit that only graphs can be defined in a *strict let* expression which will be evaluated to strong root normal form before the root expression is being evaluated (see 6.5). To ensure that evaluation indeed takes place, strict let expressions can only be used before the root expression. The order in which the graphs in the let expression will be evaluated is undefined. The scope introduced by a *strict let* expression is the same as with an ordinary *let* expression.

Strict let expressions can be used to force unique objects in a strict context such that they can be *observed* before they are *destructively updated* (see 13.6).

```
StrictLet           =  let! { {GraphDef}+ } in
```

Example (let! expression forcing evaluation).

```
SquareArrayElem :: *{Int} Int -> .{Int}
SquareArrayElem a i =   let! e = a.[i]
                        in {a & [i]=e*e}
```

6.6.2 Let-before expression

Many of the functions for input and output in the CLEAN I/O library are state transition functions. Such a state is often passed from one function to another in a single threaded way (see Chapter 10) to force a specific order of evaluation. This is certainly the case when the state is of unique type. The threading parameter has to be renamed to distinghuish its different versions. The following example shows a typical example:

Example (use of state transition functions. The uniquely typed state file is passed from one function to another involving a number of renamings: file, file1, file2)

```
readchars :: *File -> ([Char], *File)
readchars file
| not ok      = ([],file1)
| otherwise   = ([char:chars], file2)
where
     (ok,char,file1)    = freadc file
     (chars,file2)      = readchars file1
```

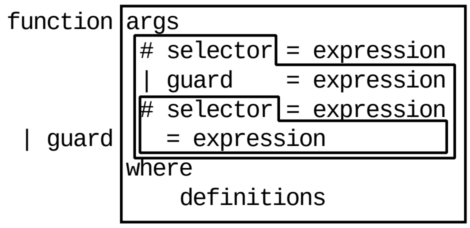
This explicit renaming of threaded parameters not only looks very ugly, these kind of definitions are sometimes also hard to read as well (in which order do things happen? which state is passed in which situation?). We have to admit: an imperative style of programming is much more easier to read when things have to happen in a certain order such as is the case when doing I/O. Thta is why we have introduced *let-before* expressions.

Let-before expressions are special let expressions which can be defined before a guard or function body. In this way one can specify sequencial actions in the order in which they suppose to happen. *Let-before* expressions have the following syntax:

LetBeforeExpression = Lets {GraphDef}+
Lets = **Let** | # | **Let!** | **#!**

The form with the exclamation mark forces the evaluation of the node-ids that appear in the left-hand sides of the definitions (see strict let-statements, section 6.6.1). Instead of the keyword `let` the `#`-symbol is often used because it looks nice in combination with the `|`-symbol used for guards.

Let-before expressions have a special scope rule to obtain an imperative programming look. The variables in the left-hand side of these definitions do not appear in the scope of the right-hand side of that definition, but they do appear in the scope of the other definitions that follow (including the root expression, excluding local definitions in where blocks. This is shown in the following picture:



Note that the scope of variables in the let before expressions does not extent to the definitions in the where expression of the alternative. The reverse is true however: definitions in the where expression can be used in the let before expressions.

Example (use of let before expressions, short notation, re-using names taking use of the special scope of the let before)

```
readchars :: *File -> ([Char], *File)
readchars file
#    (ok,char,file)     = freadc file
|    not ok             = ([],file)
#    (chars,file)       = readchars file
=    ([char:chars], file)
```

Example (equivalent definition renaming threaded parameters)

```
readchars :: *File -> ([Char], *File)
readchars file
#    (ok,char,file1)    = freadc file
|    not ok             = ([],file1)
#    (chars, file2)     = readchars file1
```

```
=    ([char:chars], file2)
```

Example (equivalent definition, using keyword `let` instead of #)

```
readchars :: *File -> ([Char], *File)
readchars file
let  (ok,char,file)      = freadc file
|    not ok              = ([],file)
let  (chars,file)        = readchars file
=    ([char:chars], file)
```

The notation can also be dangerous: the same name is used on different spots while the meaning of the name is not always the same (one has to take the scope into account which changes from definition to definition). However, the notation is rather safe when it is used to thread parameters of unique type. The type system will spot it when such parameters are not used in a correct single threaded manner. We do not recommend the use of let before expressions to adopt a imperative programming style for other cases.

Example (abuse of let before expression)

```
exchange :: (a, b) -> (b, a)
exchange (x, y)
#    temp = x
     x    = y
     y    = temp
=    (x, y)
```

7

Process annotations (DRAFT !)

UNDER CONSTRUCTION. NOT SUPPORTED IN CURRENT RELEASE. SORRY !

There are two ways of creating processes in CLEAN.

One way is by creating interactive applications. These interactive "processes" actually consist of a collection of call-back functions which are applied automatically when certain events occur. The call-back functions are applied by the I/O system sequentially one after another. Hence, scheduling takes place by the I/O system on the level of call-back functions which perform a state transition in an indivisible action. Interactive processes are explained in Chapter 10 on I/O.

In CONCURRENT CLEAN one can also create "real" processes which are executed interleaved in an undefined order or which are executed in parallel on a multi-processor architecture or on a network of processors. These CLEAN processes are generally used to speed-up the program or to obtain a specific distribution of parts of the program across a network of processors (e.g. of the interactive processes !). Interleaved or parallel executing processes can be created by adding process annotations (Plasmeijer and van Eekelen, 1993) to function applications. The annotations only influence the order of evaluation, the program remains a pure functional program, no non-deterministic effects are introduced. The original semantics of the process annotations as explained in the CLEAN book are modified to be able to deal with uniqueness typing (Kesseler, 1995). CLEAN processes are lightweight processes which run very efficient. Time-slicing, scheduling and communication is controlled by the CLEAN run-time system. Arbitrary process topologies can be created (e.g. cyclic process topologies) beyond the divide (fork) and conquer parallelism generally offered.

7.1 Process creation

If an application being evaluated contains an argument which is attributed with an process annotation (`{*I*}` or `{*P*}`) the corresponding argument will be evaluated by a new reduction *process*. This new reducer can run *interleaved* or in *parallel* with the original reduction process. The original process continues with the evaluation in the ordinary reduction order independently. The new reducer will evaluate the expression following the functional strategy until a normal form is reached.

The creation of a new process will in theory not influence the termination behaviour of the program. It will influence the time and space consumption of the program which might cause *run-time* problems when resources are exhausted.

```
Process          =  {* I *}
                 |  {* P [at ProcIdExpr] *}
ProcIdExpr       =  GraphExpr
```

With the `{*I*}` annotation a new *interleaved reducer* is created on the *same* processor that reduces the annotated graph expression to normal form (following the functional strategy). Such an interleaved reducer dies when this normal form is reached. However, during the evaluation of this result other reducers may have been created.

With the `{*P*}` annotation a new *parallel reducer* is created. This reducer is *preferably* located on a *different* processor working on a lazy copy of the corresponding sub-graph. Reducers that are located on different processors run in parallel with each other. The `{*P*}` annotations can be extended with a *location directive* `at location`, where `location` is an expression of predefined type `ProcId` indicating the processor on which the parallel process has to be created. In the library `StdProcId` (see Appendix B.3) functions are given that yield an object of this type.

When there are several local annotations specified in a contractum, the order in which they have to be effectuated is in principle depth-first with respect to the sub-graph structure.

7.2 Process communication

A reducer can demand the evaluation of a sub-graph located on another processor. Such a demand always takes place via a *communication channel* (a *lazy copy node*, see Plasmeijer and Van Eekelen, 1993).

- if the sub-graph the channel is referring to is not in strong root normal form, there will be a reducer process on the other processor (it will be already there or it will be created lazily) that will take care of the evaluation to root normal form. The demanding process is *locked* (suspended) until the root-normal form is reached.
- if the sub-graph the channel is referring to is in strong root normal form, a *lazy copy* of this sub-graph is made on the processor such that it can be inspected by the demanding reducer. Only that part of the graph expression which is in strong root normal form is copied (in one or more chunks) to the demanding processor. Such a copy is an ordinary graph which can contain shared parts, it can be cyclic and it can refer to other parts of the graph stored on another processor. Those parts of the graph which are not in root normal form will not be copied. They are lazy copied in the same way (this might induce the creation of new lazy reduction processes) whenever there is a new demand for them.
- a *reducer* will be *locked* (suspended) if it wants to reduce a redex that is already being reduced by some other reducer. A locked reducer can continue when the redex has been reduced to strong root normal form.

So, process communication takes place automatically and there will always be a serving process that will reduce the demanding information to root normal form before it is shipped.

Example (hierarchical process topology creation).

```
fib :: Int -> Int
fib 0 = 1
fib 1 = 1
fib n
| n>threshold  = fib (n-1) + {*P*} fib (n-2)
| n>2          = fib (n-1) + fib (n-2)
where
     threshold = 10
```

Example (pipeline of processes; the sieve of Eratosthenes is a classical example in which parallel sieving processes are created dynamically in a pipeline).

```
Start :: [Int]
Start = primes
where
     primes :: [Int]
     primes = sieve {*P*} [2..]

     sieve :: [Int] -> [Int]
     sieve []           = []
     sieve [pr:str]     = [pr:{*P*} sieve (filter pr str)]

     filter :: Int [Int] -> [Int]
     filter pr str  = [n \\ n <- str | n mod pr <> 0]
```

8

Defining types

CLEAN is a strongly typed language. The basic type system of CLEAN is based on the classical polymorphic Milner/Hindley/Mycroft (Milner 1978; Hindley 1969, Mycroft, 1984) type system. This type system is adapted for graph rewriting systems and extended with *basic types*, (possibly *existentially quantified) algebraic types*, *record types*, *abstract types* and *synonym types*. These types are explained in the Sections 8.1, 8.2 and 8.3.

In CLEAN each classical type is furthermore extended with *uniqueness type attributes*. This very special and important extension is explained in Chapter 9.

CLEAN allows functions and operators to be *overloaded*. *Type classes* and type constructor classes are provided (which look similar to Haskell (Hudak *et al.* , 1992) and Gofer (Jones, 1993) although they have slightly different semantics) with which a restricted context can be imposed on a type variable in a type specification. This is explained in Section 8.4.

In CLEAN types can be attributed with *strictness information* (see Section 8.5). In this way one can define data structures which (partially) will be evaluated *eager* instead of *lazy* as is by default the case in CLEAN. In this way one can even turn CLEAN into a strict language instead of a lazy one.

8.1 Types

CLEAN is a *strongly typed* language: every object (graph) and function (graph rewrite rule) in CLEAN has a type. The types of functions can be *explicitly specified* by the programmer or they can be *inferred automatically* (see 8.3.5). Types can be formed by taking instances of type constructors which have been defined explicitly as *algebraic type* (see 8.2.1), *record type* (see 8.2.2), *synonym type* (see 8.2.3), *abstract type* (see 8.2.4) or by taken instances of a *predefined type* (see 8.1.1 - 8.1.6). A *type instance* from a given type is obtained by uniformly substituting a type for a type variable. A type instance can be preceded by a uniqueness type attribute. This is further explained in Section 9.1.

```
Type                =  {BrackType}+
BrackType           =  [UnqTypeAttrib] SimpleType
SimpleType          =  TypeConstructor                  // see 8.2, 8.4
                    |  TypeVariable
                    |  BasicType                        // see 8.1.1
                    |  PredefAbstrType                  // see 8.1.2
                    |  ListType                         // see 8.1.3
                    |  TupleType                        // see 8.1.4
                    |  ArrayType                        // see 8.1.5
                    |  ArrowType                        // see 8.1.6
                    |  (Type)
```

8.1.1 Basic types

Basic types are *algebraic types* (see 8.2) which are predefined for reasons of efficiency and convenience: `Int` (for 32 bits integer values), `Real` (for 64 bit double precision floating point values), `Char` (for 8 bits ASCII character values) and `Bool` (for 8 bits Boolean values). For programming convenience special syntax is introduced to denote constant values (data constructors) of these predefined types (see Section 3.2). Functions to create and manipulate objects of basic types can be found in the CLEAN library (as indicated below).

```
BasicType           =  Int                              // see StdInt.dcl
                    |  Real                             // see StdReal.dcl
                    |  Char                             // see StdChar.dcl
                    |  Bool                             // see StdBool.dcl
```

8.1.2 Predefined abstract types

As is explained in Section 8.2.4, *Abstract data types* are types of which the actual definition is hidden. In CLEAN the types `World`, `File` and `ProcId` are *predefined abstract data types*. They are recognised by the compiler and treated specially, either for efficiency or because they play a special role in the language. Since the actual definition is hidden it is not possible to denotate constant values of these predefined abstract types. There are functions predefined in the CLEAN library for the creation and manipulation of these predefined abstract data types. Some functions work (only) on unique objects (see Chapter 9).

An object of type **`*World`** (`*` indicates that the world is unique, see 9.1) is automatically created when a program is started. This object is optionally given as argument to the `Start` function (see 12.2 and 10.1). With this object efficient interfacing with the outside world (which is indeed unique) is made possible (see Chapter 10).

An object of type **`File`** or **`*File`** can be created by means of the functions defined in `StdFileIO` (see Appendix B.5.1). It makes direct manipulation of persistent data possible (see 10.2). The type `File` is predefined for reasons of efficiency: CLEAN `Files` are directly coupled to concrete files.

An object of type **`ProcId`** can be created by means of the functions defined in `StdProcId` (see Appendix B.3.1). These objects are used in process annotations to allow process creation on an indicated processor (see Chapter 7) in a network topology.

```
PredefAbstrType     =  World                            // see StdWorld.dcl
                    |  File                             // see StdFileIO.dcl
                    |  ProcId                           // see StdProcId.dcl
```

8.1.3 List types

A *list* is an algebraic data type predefined just for programming convenience. A list can contain an *infinite number* of elements. All elements must be of the *same type*. Lists are very often used in functional languages and therefore the usual syntactic sugar is provided for the creation and manipulation of lists (dot-dot expressions, list comprehensions) while there is also special syntax for *list of characters*. (see 4.6 and 6.2.5)

Lists cannot be annotated as strict or spine strict. To create such lists a new algebraic data type has to be defined with appropriate strictness annotations (see 8.5.3).

```
ListType            =  [Type]
```

8.1.4 Tuple types

A *tuple* is an algebraic data type predefined for reasons of programming convenience and efficiency (see 13.3). Tuples have as advantage that they allow to bundle a *finite number* of objects of *arbitrary type* into a new object without being forced to define a new algebraic type for such a new object (see 4.7 and 6.2.5). This is in particular handy for functions that return several values.

The tuple arguments can optionally be annotated as being strict (see 8.5.1). This can be used to increase the efficiency of a program (see 13.3). The compiler will automatically take care of the conversion between lazy and strict tuples where needed (see 8.5.4).

```
TupleType                =  ([Strict] Type,{[Strict] Type}-list)
```

8.1.5 Array types

An *array* is an algebraic data type predefined for reasons of efficiency. Arrays contain a *finite number* of elements that all have to be of the *same type*. An array has as property that its elements can be accessed via *indexing* in *constant time*. An *array index* must be an integer value between `0` and the number of elements of the array-`1`. Destructive updates of array elements is possible thanks to uniqueness typing. For programming convenience special syntax is provided for the creation, selection and updating of array elements (array comprehensions) while there is also special syntax for *strings* (i.e. unboxed arrays of characters) (see 4.9 and 6.2.8). Arrays have as disadvantage that their use increases the possibility of a run-time error (indices that might get out-of-range). Again, see 4.9 and 6.2.8.

To obtain optimal efficiency in time and space, arrays are implemented different depending on the concrete type of the array elements. By default an array is implemented as a *lazy array* (type `{a}`), i.e. an array consists of a contiguous block of memory containing pointers to the array elements. The same representation is chosen if *strict arrays* (define its type as `{!a}`) are being used. For elements of basic type an *unboxed array* (define its type as `{#a}`) can be used. In that latter case the pointers are replaced by the array elements themselves. Lazy, strict and unboxed arrays are regarded by the CLEAN compiler as objects of different types. However, most predefined operations on arrays are overloaded such that they can be used on lazy, on strict as well as on unboxed arrays.

```
ArrayType                =  {[Strict] Type}
                         |  {#BasicType}
```

8.1.6 Arrow types

The *arrow type* is used for *function objects* (these functions have at least arity one). One can use the Cartesian product (uncurried version) to denote the function type (see 8.3) to obtain a compact notation. Curried functions applications and types are automatically converted to their uncurried equivalent versions (see 8.3.1).

```
ArrowType                =  ({BrackType}+ -> Type)
```

Example (of an arrow type).

```
((a b -> c) [a] [b] -> [c])
```

being equivalent with:

```
((a -> b -> c) -> [a] -> [b] -> [c])
```

8.2 Defining new types

New types can be defined in an implementation as well as in a definition module. Types can *only* be defined on the global level. Abstract types can only be defined in a definition module hiding the actual implementation in the corresponding implementation module (see 8.2.4 and Chapter 11).

```
Definition               =  ImportDef
                         |  TypeDef
                         |  ClassDef
                         |  FunctionDef
                         |  GraphDef
                         |  MacroDef
TypeDef                  =  AlgebraicTypeDef            //  see 8.2.1 and 9.2.1
                         |  RecordTypeDef               //  see 8.2.2 and 9.2.2
                         |  SynonymTypeDef              //  see 8.2.3 and 9.2.3
                         |  AbstractTypeDef             //  see 8.2.4 and 9.2.4
```

```
FunctionDef           =  [FunctionTypeDef] DefOfFunction          //  see 8.3 and 9.3
ClassDef              =  TypeClassDef                             //  see 8.4 and 9.4
                      |  TypeInstanceDef                          //  see 8.4 and 9.4
                      |  TypeClassInstanceExportDef               //  see 8.4
```

8.2.1 Defining algebraic data types

With an *algebraic data type* one assigns a new *type constructor* (a new type) to a newly introduced *data structure*. The data structure consists of a new *constant value*(called the *data constructor*) which can have zero or more arguments (of any type). Every data constructor must unambiguously have been (pre)defined in an algebraic data type definition. Several data constructors can be introduced in one algebraic data type definition which makes it possible to define alternative data structures of the *same* algebraic data type. The data constructors can, just like functions, be used in a curried way. Also type constructors can be used in a curried way, albeit only in the type world of course.

Polymorphic algebraic data types can be defined by adding (possibly existentially quantified, see below) *type variables* to the type constructors on the left-hand side of the algebraic data type definition. The arguments of the data constructor in a type definition are type instances of types (that are defined or are being defined).

Types can be preceded by uniqueness type attributes (see 9.2). The arguments of a defined data constructor can optionally be annotated as being strict (see 8.5).

```
AlgebraicTypeDef      =  ::TypeLhs = ConstructorDef {|ConstructorDef} ;

TypeLhs               =  [*]TypeConstructor {[*] TypeVariable}
TypeConstructor       =  TypeSymb

ConstructorDef        =  [QuantifiedVariables :] ConstructorSymb {[Strict] BrackType}
                      |  [QuantifiedVariables :] (ConstructorSymb) [Fix][Prec] {[Strict] BrackType}
QuantifiedVariables   =  {E. TypeVariable}+
Fix                   =  infixl
                      |  infixr
                      |  infix
Prec                  =  Digit
```

Example (algebraic type definition and its use).

```
::Day = Mon | Tue | Wed | Thu | Fri | Sat | Sun

::Tree a  = NilTree
          | NodeTree a (Tree a) (Tree a)

MyTree :: (Tree Int)                            //    constant  function yielding a Tree of Int
MyTree = NodeTree 1 NilTree NilTree
```

An algebraic data type definition can be seen as the specification of a grammar in which is specified what legal data objects are of that specific type. All data constructors being defined must therefore have *different* names, to make type inferencing possible. Notice that the other CLEAN types (basic, list, tuple, array, record, abstract types) can be regarded as special cases of an algebraic type.

Defining infix data constructors

Constructors with two arguments can be defined as *infix constructor*, in a similar way as function operators (with fixity (`infixl`, `infixr` or just `infix`, default `infixl`) and precedence (`0` through `9`, default `9`). Infix constructors can also be used in prefix position when they are surrounded by brackets (see 6.1).

Example (algebraic type defining an infix data constructor, function on this type; notice that one cannot use a '`:`' because this character is already reserved).

```
::List a  = (\) infixr 5 a (List a)
          | Nil

Head :: (List a) -> a
Head (x\xs) = x
```

Using higher order types

In an algebraic type definition ordinary types can be used (such as a basic type, e.g. `Int`, or a list type, e.g. `[Int]`, or an instantiation of a user defined type, e.g. `Tree Int`), but one can also use *higher order types*. Higher order types can be constructed by curried applications of the type constructors. Higher order types can be applied in the type world in a similar way as higher order functions in the function world. The use of higher order types increases the flexibility with which algebraic types can be defined. Higher order types play an important role in combination with type classes (see 8.4).

```
Type                = {BrackType}+
BrackType           = [UnqTypeAttrib] SimpleType
SimpleType          = TypeConstructor
                    | TypeVariable
                    | BasicType
                    | PredefAbstrType
                    | ListType
                    | TupleType
                    | ArrayType
                    | ArrowType
                    | (Type)
TypeConstructor     = TypeSymb                    // a user defined type
                    | []                          // list type
                    | ({,}+)                      // tuple type (arity >= 2)
                    | {}                          // lazy array type
                    | {!}                         // strict array type
                    | {#}                         // unboxed array type
                    | (->)                        // an arrow type
```

Predefined types can also be used in curried way. To make this possible all predefined types can be written down in prefix notation as well, as follows:

```
[] a          is equivalent with  [a]
(,) a b       is equivalent with  (a,b)
(,,) a b c    is equivalent with  (a,b,c)  and so on for n-tuples
{} a          is equivalent with  {a}
{!} a         is equivalent with  {!a}
{#} a         is equivalent with  {#a}
(->) a b      is equivalent with  (a -> b)
```

Of course, one needs to ensure that all types are applied in a correct way. To be able to specify the rules that indicate whether a type itself is correct, we introduce the notion of *kind*. A kind can be seen as the `type of a type`. In our case, the kind of a type expresses the number of type arguments this type may have. The kind `X` stands for any so-called *first-order* type: a type expecting no forther arguments ((`Int`, `Bool`, `[Int]`, etcetera). The kind `X -> X` stands for a type that can be be applied to a (first-order) type, which then yields another first-order type, x -> x -> x expecting two type arguments of, and so on.

```
Int, Bool, [Int], Tree [Int]      :: X
[], Tree, (,) Int, (->) a, {}     :: X -> X
(,), (->)                         :: X -> X -> X
(,,)                              :: X -> X -> X -> X
```

In CLEAN each *top level* type should have kind `X`. A top level type is a type that occurs either as an argument or result type of a function or as argument type of a data constructor (in some algebraic type definition). The rule for determining the kinds of the type variables (which can be of any order) are fairly simple: The kind of a type variable directly follows from its use. If a variable has no arguments, its kind is `X`. Otherwise, its kind corresponds to the number of arguments to whch the variable is applied. The kind of type variable determines its possible instantiations, i.e. it can only be instantiated with a type which is of the same kind as the type variable itself.

Example (algebraic type using higher order types; the type variable `t` in the definition of `Tree2` s of kind `X -> X`. Tree2 is instantiated with a list (also of kind `X -> X`) in the definition of MyTree2).

```
::Tree2 t     = NilTree
```

```
                 | NodeTree (t Int) (Tree2 t) (Tree2 t)

MyTree2 :: Tree2 []
MyTree2 = NodeTree [1,2,3] NilTree NilTree
```

Defining algebraic data types with existentially quantified variables

An algebraic type definition may contain *existentially quantified type variables* (or, for short, existential type variables) (Läufer 1992). These special variables are indicated by preceding them with "**E.**". Existential types are useful if one wants to create (recursive) data structures in which objects of *different types* are being stored (e.g. a list with elements of different types).

Example (existential type definitions and theis use). In this example a list-like structure is defined in which functions can be stored. The functions in this structure can be applied one after another in a pipe-line fashion. Each function in the pipeline can yield a result of *arbitrary* type which is exactly of the type required by the next function in the pipe-line. The first function in the pipeline expects type a, the last will yield type `b`. Hence, the function composed in this way is a function of type `a -> b`. The recursive function `ApplyPipe` happens to be an example of a recursive function which type cannot be inferred (with the Milner type system), however its specified type can be checked (with the Mycroft type system).

```
::Pipe a b     =    Direct (a -> b)
               |    E.via:    Indirect (a -> via) (Pipe via b)

ApplyPipe :: (Pipe a b) a -> b
ApplyPipe (Direct func) val          = func val
ApplyPipe (Indirect func pipes) val   = ApplyPipe pipes (func val)

Start = ApplyPipe (Indirect toReal (Indirect exp (Direct toInt))) 3
```

To ensure correctness of typing, there is a limitation imposed on the use of *existentially quantified data structures*.

- Once a data structure containing existentially quantified parts is created the type of these components are forgotten. This means that, in general, if such a data structured is passed to another function it is statically impossible to determine the actual types of those components: it can be of any type. Therefore, a function having an existentially quantified data structure as input is not allowed to make specific type assumptions on the parts that correspond to the existential type variables. This implies that one can only instantiate an existential type variable with a concrete type when the object is created.

Counter Example (Illegal use of an object with existentially quantified components; the concrete type of the components of the `Pipe` are unknown).

```
ApplFunc :: (Pipe Int b) -> ??
ApplFunc (Indirect func pipes) = func 3
```

Semantic restrictions on algebraic data types

Other semantic restrictions on algebraic data types:

- The name of a type must be different from other names in the same scope (see 3.5) and name space (see 3.4).
- All type variables on the left-hand side must be different.
- All type variables used on the right-hand side are bound, i.e. must be introduced on the left-hand side of the algebraic type being defined.
- A data constructor can only be defined once within the same scope and name space. So, each data constructor unambiguously identifies its type to make type inferrencing possible.
- When a data constructor is used in infix position both arguments have to be present. Data constructors can be used in a curried way in the function world, but then they have to be used as ordinary prefix constructors.
- Type constructors can be used in a curried way in the type world; to use predefined bracket-like type constructors (for lists, tuples, arrays) in a curried way they must be used in prefix notation.
- The right-hand side of an algebraic data type definition should yield a type of kind `X`, all arguments of the data constructor being defined should be of kind `X` as well.

- A type can only be instantiated with a type that is of the same kind.
- An existentially quantified type variable specified in an algebraic type can only be instantiated with a concrete type (= not a type variable) when a data structure of this type is created.

8.2.2 Defining record types

A *record type* is basically an algebraic data type in which exactly one constructor is defined. Special about records is

- that a *field name* is attached to each of the arguments of the data constructor;
- that records cannot be used in a curried way.

Compared with ordinary algebraic data structures the use of records gives a lot of notational convenience because the field names enable *selection by field name* instead of *selection by position*. When a record is created *all* arguments of the constructor have to be defined but one can specify the arguments in *any* order (see 4.8). Furthermore, when pattern matching is performed on a record, one only has to mention those fields one is interested in (see 6.2.6). A record can be created via a functional update (see 4.8). In that case one only has to specify the values for those fields which differ from the old record. Matching and creation of records can hence be specified in CLEAN in such a way that after a change in the structure of a record only those functions have to be changed which are explicitly referring to the changed fields.

Existential type variables (see 8.2.1) are allowed in record types (as in any other type). The arguments of the constructor can optionally be annotated as being strict (see 8.5). The optional uniqueness attributes are treated in 9.2.

RecordTypeDef = **::**TypeLhs **= {**{FieldSymbol **::** [Strict] Type}-*list* **}** ;

As data constructor for a record the name of the record type is used internally.

- The semantic restrictions which apply for algebraic data types also hold for record types.
- The field names inside one record all have to be different. It is allowed to use the same field name in different records.

Example (record definition).

```
::Complex     =    { re      :: Real
                   , im      :: Real
                   }
```

The combination of existential type variables in record types are of use for an object oriented style of programming.

Example (using existentially quantified records to create object of same type but which can have different representations).

```
::Object       = E.x:
                   { state   :: x
                   , get     :: x -> Int
                   , set     :: x Int -> x
                   }


CreateObject1 :: Object
CreateObject1 = {state = [], get = myget, set = myset}
where
     myget :: [Int] -> Int
     myget [i:is]   = i
     myget []       = 0

     myset :: [Int] Int -> [Int]
     myset is i = [i:is]

CreateObject2 = {state = 0.0, get = myget, set = myset}
where
     myget :: Real -> Int
     myget r = toInt r
```

```
         myset :: Real Int -> Real
         myset r i = r + toReal i

    Get :: Object -> Int
    Get {state,get} = get state

    Set :: Object Int -> Object
    Set o=:{state,set} i = {o & state = set state i}

    Start :: [Object]
    Start = map (Set 3) [CreateObject1,CreateObject1]
```

8.2.3 Defining synonym types

Synonym types permit the programmer to introduce new type names for an existing type.

SynonymTypeDef = **::**TypeLhs **:==** Type ;

- For the left-hand side the same restrictions hold as for algebraic types (see 8.2.1).
- Cyclic definitions of synonym types (e.g. `::T a b :== G a b; ::G a b :== T a b`) are not allowed.

Example (type synonym definition).

```
::Operator a :== a a -> a

map2 :: (Operator a) [a] [a] -> [a]
map2 op [] []          = []
map2 op [f1:r1] [f2:r2] = [op f1 f2 :map2 op r1 r2]

Start :: Int
Start = map2 (*) [2,3,4,5] [7,8,9,10]
```

8.2.4 Defining abstract data types

A type can be exported by defining the type in a CLEAN definition module (see Chapter 11). For software engineering reasons it sometimes better only to export the name of a type but not its concrete definition (the right-hand side of the type definition). The type then becomes an *abstract data type*. In CLEAN this is done by specifying only the left-hand-side of a type in the definition module while the concrete definition (the right-hand side of the type definition) is hidden in the implementation module. So, CLEAN's module structure is used to hide the actual implementation. When one wants to do something useful with objects of abstract types one needs to export functions that can create and manipulate objects of this type as well.

- Abstract data type definitions are only allowed in definition modules, the concrete definition has to be given in the corresponding implementation module.
- The left-hand side of the concrete type should be identical to (modulo alpha conversion for variable names) the left-hand side of the abstract type definition (inclusive strictness and uniqueness type attributes).

AbstractTypeDef = **::**TypeLhs ;

Example (abstract data type).

```
definition module stack

::Stack a

Empty    ::   (Stack a)
isEmpty  ::   (Stack a) -> Bool
Top      ::   (Stack a) -> a
Push     :: a (Stack a) -> Stack a
Pop      ::   (Stack a) -> Stack a
```

```
implementation module stack

::Stack a :== [a]

Empty :: (Stack a)
Empty = []

isEmpty :: (Stack a) -> Bool
isEmpty [] = True
isEmpty s  = False

Top :: (Stack a) -> a
Top [e:s] = e

Push :: a (Stack a) -> Stack a
Push e s = [e:s]

Pop :: (Stack a) -> Stack a
Pop [e:s] = s
```

8.3 Typing functions and operators

Although one is in general not obligated to explicitly specify the *type of a function* (the CLEAN compiler can *infer* the type) the explicit specification of the type is *highly recommended* to increase the readability of the program.

```
FunctionDef         = [FunctionTypeDef] DefOfFunction

FunctionTypeDef     = FunctionSymb :: FunctionType ;
                    | (FunctionSymb) [Fix][Prec] [:: FunctionType] ;
Fix                 = infixl
                    | infixr
                    | infix
Prec                = Digit
FunctionType        = [{[Strict] BrackType}+ ->] Type [ClassContext] [UnqTypeUnEqualities]
```

An explicit specification is *required* when a function is exported, or when the programmer wants to impose additional restrictions on the application of the function (e.g. a more restricted type can be specified, strictness information can be added as explained in Section 8.5, a class context for the type variables can be defined as explained in Section 8.4, uniqueness information can be added as explained in Section 9.3). The CLEAN type system uses a combination of Milner/Mycroft type assignment. This has as consequence that the type system in some rare cases is not capable to infer the type of a function (using the Milner/Hindley system) although it will approve a given type (using the Mycroft system; see Plasmeijer and Van Eekelen, 1993; see also the example in 8.2.1).

The Cartesian product is used for the specification of the function type. Cartesian product is denoted by juxtaposition of the bracketed argument types. For the case of a single argument the brackets can be left out. In type specifications the binding priority of the application of type constructors is higher than the binding of the arrow ->. To indicate that one defines an operator the function name is on the left-hand side surrounded by brackets.

- The function symbol before the double colon should be the same as the function symbol of the corresponding rewrite rule.
- The arity of the functions has to correspond with the number of arguments of which the Cartesian product is taken. So, in CLEAN one can tell the arity of the function by its type.

Example (arity of a function reflected in type).

```
map :: (a->b) [a] -> [b]                                  //   map  has arity  2
map f []      =    []
map f [x:xs]  =    [f x : map f xs]

domap :: ((a->b) [a] -> [b])                              //   domap  has arity zero
domap = map
```

- The arguments and the result types of a function should be of kind `X`.
- In the specification of a type of a locally defined function one cannot refer to a type variable introduced in the type specification of a surrounding function (there is not yet a scope rule on types defined). The type of *such* a local function can therefore not yet be specified by the programmer. However, the type will be inferred and checked (after it is lifted by the compiler to the global level) by the type system.

Counter example (illegal type specification). The function `g` returns a tuple. The type of the first tuple element is the same as the type of the polymorphic argument of `f`. Such a dependency (here indicated by "`^`" cannot be specified yet.

```
f:: a -> (a,a)
f x = g x
where
     // g :: b -> (^a,b)
     g y = (x,y)
```

8.3.1 Typing curried functions

In CLEAN all symbols (functions and constructors) are defined with *fixed arity*. However, in a application it is of course allowed to apply them to an arbitrary number of arguments. A *curried application* of a function is an application of a function with a number of arguments which is less than its arity (note that in CLEAN the arity of a function can be derived from its type). With the aid of the predefined internal function `_AP` a curried function applied on the required number of arguments is transformed into an equivalent uncurried function application.

The type axiom's of the CLEAN type system include for all `s` defined with arity `n` the equivalence of $s::(t_1\text{->}(t_2\text{->}(\ldots(t_n\text{->}t_r)\ldots))$ with $s::t_1\ t_2\ \ldots\ t_n\ \text{->}\ t_r$.

8.3.2 Typing operators

An *operator* is a *function with arity two* that can be used in infix position. An operator can be defined by enclosing the operator name between parentheses in the left-hand-side of the function definition. An operator has a *precedence* (`0` through `9`, default `9`) and a *fixity* (**`infixl`**, **`infixr`** or just **`infix`**, default `infixl`). A higher precedence binds more tightly. When operators have equal precedence, the fixity determines the priority. In an expression an ordinary function application always has the highest priority (`10`). See also Section 4.3 and 6.1.

- The type of an operator must obey the requirements as defined for typing functions with arity two.
- If the operator is explicitly typed the operator name should also be put between parentheses in the type rule.
- When an infix operator is enclosed between parentheses it can be applied as a prefix function. Possible recursive definitions of the newly defined operator on the right-hand-side also follow this convention.

Example (an operator definition and its type).

```
(o) infix 8 :: (x -> y) (z -> x) -> (z -> y)                    // function composition
(o) f g = \x -> f (g x)
```

8.3.3 Typing partial functions

Patterns and guards imply a condition that has to be fulfilled before a rewrite rule can be applied (see 6.2 and 6.3). This makes it possible to define *partial functions*, functions which are not defined for all possible values of the specified type.

- When a partial function is applied to a value outside the domain for which the function is defined it will result into a *run-time* error.

The compiler gives a warning when functions are defined which might be partial.

With the **`abort`** expression (see StdMisc.dcl) one can change any partial function into a *total function* (the `abort` expression can have any type). The `abort` expression can be used to give a user-defined run-time error message

Example (use of `abort` to make a function total).

```
fac :: Int -> Int
fac 0                  = 1
fac n
    | n>=1             = n * fac (n - 1)
    | otherwise        = abort "fac called with a negative number"
```

8.4 Typing overloaded functions and operators

The names of the functions one defines generally all have to be *different* within the same scope and name space (see 3.4). However, it is sometimes very convenient to *overload* certain functions and operators (e.g. +, -, ==), i.e. use *identical* names for *different* functions or operators that perform *similar tasks* albeit on objects of *different types*.

In principle it is possible to simulate a kind of overloading by using records. One simply defines a record (see 8.2.2) in which a collection of functions are stored that somehow belong to each other. Now the field name of the record can be used as (overloaded) synonym for any concrete function stored on the corresponding position. The record can be regarded as a kind of *dictionary* in which the concrete function can be looked up.

Example (the use of a dictionary record to simulate overloading/type classes). `sumlist` can use the field name `add` as synonym for any concrete function obeying the type as specified in the record definition. The operators `+.`, `+^`, `-.` and `-^` are assumed to be predefined primitives operators for addition and subtraction on the basic types `Real` and `Int`.

```
::Arith a =   {   add       :: a a -> a
              ,   subtract  :: a a -> a
              }

ArithReal = { add = (+.), subtract = (-.) }
ArithInt  = { add = (+^), subtract = (-^) }

sumlist :: (Arith a) [a] [a] -> [a]
sumlist arith [x:xs] [y:ys]  =   [arith.add x y:sumlist arith xs ys]
sumlist arith x y            =   []

Start = sumlist ArithInt [1..10] [11..20]
```

A disadvantage of such a dictionary record is that it is syntactically not so nice (e.g. one explicitly has to pass the record to the appropriate function) and that one has to pay a huge price for efficiency (due to the use of higher order functions) as well. CLEAN's overloading system as introduced below enables the CLEAN system to automatically create and add dictionaries as argument to the appropriate function definitions and function applications. To avoid efficiency loss the CLEAN compiler will substitute the intended concrete function for the overloaded function application where possible. In worst case however CLEAN's overloading system will indeed have to generate a dictionary record which is then automatically passed as additional parameter to the appropriate function.

8.4.1 Type classes

In a *type class definition* one gives a name to a *set of overloaded functions* (this is similar to the definition of a type of the dictionary record as explained above). For each *overloaded function* or *operator* which is a *member* of the class the *overloaded name* and its *overloaded type* is specified. A special *overloaded type variable* indicates how the different instantiations of the class can vary from each other.

```
TypeClassDef        =  class ClassSymb TypeVariable [ClassContext]
                          [[where] { {ClassMemberDef}+ }]
                    |  class FunctionSymb TypeVariable :: FunctionType;
                    |  class (FunctionSymb) [Fix][Prec] TypeVariable :: FunctionType;
```

ClassMemberDef = FunctionTypeDef
[MacroDef]

Example (definition of a type class; in this case the class named `Arith` contains two overloaded operators).

```
class Arith a
where
     (+) infixl 6 :: a a -> a
     (-) infixl 6 :: a a -> a
```

With an *instance declaration* an instance of a given class can be defined (this is similar to the creation of a dictionary record). When the instance is made it has to be specified for which *concrete type* an instance is created. For each overloaded function in the class a *concrete function* or *operator* has to be defined. The type of a concrete function must exactly match the corresponding overloaded type after uniform substitution of the concrete type for the overloaded function type in the type class definition.

TypeClassInstanceDef | **instance** *ClassSymb* [BrackType [**default**] [ClassContext]]
[[**where**] ⦃{DefOfFunction}+ ⦄]

Example (definition of an instance of a type class `Arith` for type `Int`). Notice that the type of the concrete functions can be deduced by substituting the concrete type for the overloaded type variable in the corresponding class definition. One is not obliged to repeat the type of the concrete functions instantiated (nor the fixity or associativity in the case of operators) .

```
instance Arith Int
where
     (+) :: Int Int -> Int
     (+) x y = x +^ y

     (-) :: Int Int -> Int
     (-) x y = x -^ y
```

Example (definition of an instance of a type class `Arith` for type `Real`).

```
instance Arith Real
where
     (+) x y = x +. y
     (-) x y = x -. y
```

One can define as many instances of a class as one likes. Instances can be added later on in any module.

- When an instance of a class is defined a concrete definition has to be given for all the class members.

8.4.2 Functions defined in terms of overloaded functions

When an overloaded name is encountered in an expression, the compiler will determine which of the corresponding concrete functions/operators is meant by looking at the concrete type of the expression. This type is used to determine which concrete function to apply. All instances of the overloaded type variable of a certain class (with exception of the default instance, see below) must therefore not overlap (being not unifyable) with each other and they all have to be of flat type (see the restrictions mentioned in 8.4.11). If it is clear from the type of the expression which one of the concrete instantiations is meant the compiler will in principle substitute the concrete function for the overloaded one, such that no efficiency is lost.

Example (substitution of a concrete function for an overloaded one). given the definitions above the function

```
inc n = n + 1
```

will be internally transformed into

```
inc n = n +^ 1
```

However, it is very well possible that the compiler, given the type of the expression, cannot decide which one of the corresponding concrete functions to apply. The new function then becomes overloaded as well.

For instance, the function

```
add x y = x + y
```

becomes overloaded as well because anyone of the concrete instances can be applied. Consequently, `add` can be applied to arguments of any type as well, as long as addition (+) is defined on them.

This has as consequence that an additional restriction must be imposed on the type of such an expression. A *class context* has to be added to the function type to express that the function can only be applied provided that the appropriate type classes have been instantiated (in fact one specifies the type of the dictionary record which has to be passed to the function in worst case). Such a context can also be regarded as an additional restriction imposed on a type variable, introducing a kind of *bounded polymorphism*.

```
| FunctionType    = [{[Strict] BrackType}+ ->] Type [ClassContext] [UnqTypeUnEqualities]
| ClassContext    = | ClassSymb-list TypeVariable {& ClassSymb-list TypeVariable }
```

Example (use of a class context to impose a restriction on the instantiation of type variable). The function add can be applied on arguments of any type under the condition that an instance of the class `Arith` is defined on them.

```
add :: a a -> a | Arith a
add x y = x + y
```

CLEAN's type system can infer contexts automatically. If a type class is specified as restricted context the type system will check the correctness of the specification (as always a type specification can be more restrictive than is deduced by the compiler).

8.4.3 Instances of type classes defined in terms of overloaded functions

The concrete functions defined in a class instance definition can also be defined in terms of (other) overloaded functions. This is reflected in the type of the instantiated functions. Both the concrete type and the context the class instantiation (and its members) is depending on need to be specified.

Example (instance declaration of which type is depending on the same type class). The function + on lists can be defined in terms of the overloaded operator + on the list elements. With this definition + is defined not only on lists, but also on a list of lists etcetera.

```
instance Arith [a] | Arith a                              //   on lists
where
     (+) infixl 6 :: [a] [a] -> [a] | Arith a
     (+) [x:xs] [y:ys]  =    [x + y:xs + ys]
     (+) _      _       =    []

     (-) infixl 6 :: [a] [a] -> [a] | Arith a
     (-) [x:xs] [y:ys]  =    [x - y:xs - ys]
     (-) _      _       =    []
```

Example (Equality class).

```
class Eq a
where
     (==) infix 2 :: a a -> Bool

instance Eq    [a] | Eq a                                      //   on lists
where
     (==) infix 2 :: [a] [a] -> Bool | Eq a
     (==) [x:xs] [y:ys] = x == y && xs == ys
     (==) []     []     = True
     (==) _      _      = False
```

8.4.4 Type constructor classes

The CLEAN type system offers the possibility to use higher order types (see 8.2.1). This makes it possible to define *type constructor classes* (similar to constructor classes as introduced in Gofer, Jones (1993)).

In that case the overloaded type variable of the type class is not of kind `X`, but of higher order, e.g. `X -> X`, `X -> X -> X`, etcetera. This offers the possibility to define overloaded functions which can be instantiated with type constructors of higher order (as usual, the overloaded type variable and a concrete instantiation of this type variable need to be of the same kind). This makes it possible to overload more complex functions like `map` and the like.

Example (definition of a type constructor class). The class `Functor` including the overloaded function `map` which varies in type variable `f` of kind `X -> X`).

```
class Functor f
where
     map :: (a -> b) (f a) -> (f b)
```

Example (instantiation of a type constructor class). An instantiation of the well-known function `map` applied on lists (`[]` is of kind `X -> X`), and a `map` function defined on Tree's (Tree is of kind `X -> X`).

```
instance Functor []
where
     map :: (a -> b) [a] -> [b]
     map f [x:xs]  =    [f x : map f xs]
     map f []      =    []

::Tree a  =    (/\) infixl 0 (Tree a) (Tree a)
          |    Leaf a

instance Functor Tree
where
     map :: (a -> b) (Tree a) -> (Tree b)
     map f (l/\r)       =    map f l /\ map f r
     map f (Leaf a)     =    Leaf (f a)
```

8.4.5 Generic instances

It is possible to specify a *generic instance* (in that case a type variable is specified as instance for the overloaded type variable in the instance declaration) which will be taken when none of the other defined instances happens to be applicable. Since such a function must work for *any* instance the type of the generic instance must be equivalent to the type of the overloaded function. Therefore it can only perform very general tasks.

Example (defining a generic instance). In this example any two objects of arbitrary type can be compared with each other but they are by default unequal unless specified otherwise.

```
instance Eq    a                                    // generic instance for Eq
where
     (==) infix 2 :: a a -> Bool
     (==) x y = False
```

8.4.6 Default instances

It is possible that a CLEAN expression using overloaded functions is internally *ambiguously overloaded*.

- The problem can occur when an overloaded function is used which has on overloaded type in which the overloaded type variable only appears on the right-hand side of the `->`. If such a function is applied in such a way that the overloaded type does not appear in the resulting type of the application, any of the available instances of the overloaded function can be used. In that case the system cannot determine which instance to take, such that a type error is given.

Counter example (ambiguous overloaded expression). The function body of `f` is ambiguously overloaded which results in a type error. It is not possible to determine whether its argument should be converted to an `Int` or to a `Bool`.

```
class Read  a :: a -> String
class Write a :: String -> a
instance Read  Int, Bool                            // export of class instance, see 8.4.10
instance Write Int, Bool
```

```
f:: String -> String
f x = Write (Read x)     // ! This results in a type error !
```

One can solve such an ambiguity by splitting up the expression in parts that are typed explicitly such that it becomes clear which of the instances should be used.

```
f:: String -> String
f x = Write (MyRead x)
where
     MyRead :: Int -> String
     MyRead x = Read x
```

Another way to solve the ambiguity is to mark one of the instances as the *default instance* (indicated by the keyword `default` in the instance declaration) which will be taken in the case an ambiguously overloaded expression is encountered.

Example (default instance declaration to be used to solve ambiguities). The function body of `f` is ambiguously overloaded. Due to the default instance specified the argument is converted to an `Int`.

```
class Read  a :: a -> String
class Write a :: String -> a
instance Read  Int default, Bool
instance Write Int default, Bool

f:: String -> String
f x = Write (Read x)
```

8.4.7 Defining derived members in a class

The members of a class consists of a set of functions or operators which logically belong to each other. It is often the case that the effect of some members (*derived members*) can be expressed in others. For instance, `<>` can be regarded as synonym for `not (==)`. For software engineering (the fixed relation is made explicit) and efficiency (one does not need to include such derived members in the dictionary record) it is good to make this relation explicit. In CLEAN the existing macro facilities are used for this purpose.

Example (Classes with macro definitions to specify derived members).

```
class Eq a
where
     (==) infix 2 :: a a -> Bool

     (<>) infix 2 :: a a ->  Bool | Eq a
     (<>) x y :== not (x == y)

class Ord a
where
     (<) infix 2 :: a a ->   Bool

     (>) infix 2 :: a a ->   Bool | Ord a
     (>) x y :== y < x

     (<=) infix 2 :: a a ->  Bool | Ord a
     (<=) x y :== not (y<x)

     (>=) infix 2 :: a a ->  Bool | Ord a
     (>=) x y :== not (x<y)

     min :: a a -> a | Ord a
     min x y :== if (x<y) x y

     max :: a a -> a | Ord a
     max x y :== if (x<y) y x
```

8.4.8 A shorthand for defining overloaded functions

A class definition seems sometimes a bit overdone when a class actually only consists of one member. Special syntax is provided for this case.

TypeClassDef	=	**class** *ClassSymb TypeVariable* [ClassContext]
		[[**where**] {{ClassMemberDef}+ }]
	|	**class** *FunctionSymb TypeVariable* **::** FunctionType;
	|	**class (***FunctionSymb***)** [Fix][Prec] *TypeVariable* **::** FunctionType;

Example (defining an overloaded function/operator).

```
class (+) infixl 6 a :: a a -> a
```

which is shorthand for:

```
class + a
where
     (+) infixl 6:: a a -> a
```

The instantiation of such a simple one member class is done in a similar way as with ordinary classes, using the name of the overloaded function as class name (see the syntax definition for instantiation).

Example (instantiations of an overloaded function/operator).

```
instance + Int
where
     (+) x y = x +^ y
```

8.4.9 Classes defined in terms of other classes

In the definition of a class one can optionally specify that other classes which already have been defined elsewhere are included. The classes to include are specified as context after the overloaded type variable. It is not needed (but it is allowed) to define new members in the class body of the new class. In this way one can give a new name to a collection of existing classes creating a hierarchy of classes (cyclic dependencies are forbidden). Since one and the same class can be included in several other classes, one can combine classes in different kinds of meaningful ways. For an example have a closer look at the CLEAN standard library (see e.g. `StdOverloaded` and `StdClass`)

Example (defining classes in terms of existing classes). The class `Arith` consists of the class `+` and `-`.

```
class (+) infixl 6 a :: a a -> a

class (-) infixl 6 a :: a a -> a

class Arith a | +,- a
```

8.4.10 Exporting type classes

To export a class one simply repeats the class definition in the definition module (see Chapter 12). To export an instantiation of a class one simply repeats the instance definition in the definition module, however *without* revealing the concrete implementation. This can only be specified in the implementation module.

Example (Exporting classes and instances).

```
definition module example

class Eq a                                 //   the class Eq is exported
where
     (==) infix 2 :: a a -> Bool

instance Eq   [a] | Eq a                   //   an instance of Eq on lists is exported
instance Eq    a                           //   a generic instance of Eq is exported
```

For reasons of efficiency the compiler will always try to make specialised efficient versions of functions which have become overloaded (see above). In principle one version is made for each possible concrete application. However, when an overloaded function is exported it is unknown with which concrete instances the function will be applied. So, a record is constructed in which the concrete function is stored as is explained in the introduction of this section. This approach can be very inefficient, especially in comparison to a specialised version for instantiations of basic type. The compiler can generate much better code for other modules if it is informed about the instances known in the implementation module. The compiler is unaware of such information (it only inspects definition modules in case of separate compilation). The information should therefore be provided in the corresponding definition module. To make this possible a special export definition is provided. It is recommended to add such an export definition if speed matters, leaf it out when it does not matter or when a small code size matters more. The export definition will only have an effect for instances of basic type (for these types it can really help to have a special version) .

TypeClassInstanceExportDef
= **export** *ClassSymb* BasicType-*list*;

Example (Exporting class instances).

```
export Eq Int, Real
```

8.4.11 Semantic restrictions on type classes

Semantic restrictions:

- When a class is instantiated a concrete definition must be given for each of the members in the class (not for derived members).
- The type of a concrete function or operator must exactly match the overloaded type after uniform substitution of the overloaded type variable by the concrete type as specified in the corresponding type instance declaration.
- The overloaded type variable and the concrete type must be of the same kind.
- A type instance of an overloaded type must be a *flat type*, i.e. a type of the form `T` a_1 … a_n where a_i are type variables which are *all* different.
- All instances other than the default instance of a given overloaded type must differ from each other (be ununifyable with each other).
- It is not allowed to use a type synonym as instance.
- The start rule cannot have an overloaded type.
- If a default instance is specified the type of the corresponding concrete default function must be identical to the type of the overloaded function or operator.
- For the specification of derived members in a class the same restrictions hold as for defining macros.
- A restricted context can only be imposed on one of the type variables appearing in the type of the expression.
- The specification of the concrete functions can only be given in implementation modules.

8.5 Partially strict data structures and functions

CLEAN uses by default a *lazy evaluation strategy*: a redex is only evaluated when it is needed to compute the final result. But it is generally much more efficient to calculate arguments in advance (see 13.3 and Nöcker & Smetsers, 1990, 1993). It gives the possibility to manipulate objects *unboxed* (e.g. in a registers instead of in a nodes of the graph). Therefore it is possible in CLEAN in a type definition to *annotate the arguments of a function* (see 8.3) and *of a data constructor* (see 8.2) to be *strict*. This will force the evaluation the arguments to strong root normal form when the function or data structure is used in a *strict context* (see below). The compiler is capable of deriving strictness information for the arguments of functions, so generally there is no need for the programmer to specify these kind of strictness explicitly.

When a strict annotated argument is put in a strict context while the argument is defined in terms of another strict annotated data structure the latter is put in a strict context as well and therefore also evaluated. So, one can change the default *lazy semantics* of CLEAN into a (*hyper*) *strict semantics* as demanded. The type system will check the consistency of types and ensure that the specified strictness is maintained.

One has to be careful though. When strictness annotations are put on arguments representing infinite computations or infinite data structures the program the termination behaviour of the program might change. It is only safe to put strictness annotations in the case that the function or data constructor is known to be strict in the corresponding argument which means that the evaluation of that argument in advance does not change the termination behaviour of the program. The compiler is not able to check this.

```
Strict          =  !
```

8.5.1 Strict and lazy context

Each graph expression on the right-hand side of a rewrite rule is considered to be either strict (appearing in a *strict context*: it has to be evaluated to strong root normal form) or lazy (appearing in a *lazy context*: not yet to be evaluated to strong root normal form). The following rules specify whether or not a particular graph expression is lazy or strict:

+ a non-variable pattern is strict;
+ an expression in a guard is strict;
+ the expressions specified in a strict let expression are strict;
+ the root expression is strict;
+ the arguments of a function or data constructor in a strict context are strict when these arguments are being annotated as strict in the type definition of that function or data constructor;
+ all the other nodes are lazy.

Evaluation will happen in the following order: patterns, guard, expressions in a strict let expression, root expression (see also 6.1 and 9.3.4).

8.5.2 Functions with strict arguments

In the type definition of a function the arguments can optionally be annotated as being strict. In reasoning about functions it will always be true that the corresponding arguments will be in strong root normal form (see 2.1) before the rewriting of the function takes place.

Example (a function with strict annotated arguments).

```
Acker :: !Int !Int -> Int
Acker 0 j =   inc j
Acker i 0 =   Acker (dec i) 1
Acker i j =   Acker (dec i) (Acker i (dec j))
```

The CLEAN compiler includes a fast and clever strictness analyser which is based on abstract reduction (Nöcker, 1993). The compiler can derive the strictness of the function arguments in many cases, such as for the example above. Therefore there is generally no need to add strictness annotations to the type of a function by hand. When a function is exported from a module (see Chapter 12), its type has to be specified in the definition module. To obtain optimal efficiency, the programmer should also include the strictness information to the type definition in the definition module. One can ask the compiler to print out the types with the derived strictness information and paste this into the definition module.

8.5.3 Defining data structures with strict arguments

It is very hard for a strictness analyser to deduce strictness of data structures since this is highly depending on the way the data structure is being used (the CLEAN compiler will do its best though). Functional programs will generally run much more efficient when strict data structures are being used instead of lazy ones. If the inefficiency of your program becomes problematic one can think of changing lazy data structures into strict ones by hand.

In the type definition of a constructor (in an algebraic data type definition or in a the definition of a record type) the arguments of the data constructor can optionally be annotated as being strict. In reasoning about objects of such a type it will always be true that the annotated argument will be in strong root normal form when the object is examined. Whenever a new object is created in a strict context, the compiler will take care of the evaluation of the strict annotated arguments. When the new object is created in a lazy context, the compiler will insert code that will take care of the evaluation whenever the object is put into a strict context. If one makes a data structure strict in a certain argument, it is better not define infinite instances of such a data structure to avoid non-termination.

So, in a type definition one can define a data constructor to be strict in zero or more of its arguments. Strictness is a property of data structure which is specified in its type. In general (with the exceptions of tuples) one cannot arbitrary mix strict and non-strict data structures because they are considered to be of different type. So, e.g. if one wants to use list with strict elements or a spine strict list one has to define new algebraic data types (with different data constructors). One cannot simply use the predefined notation for lists because these lists are lazy lists.

Example (list with a strict elements). The list element will be evaluated when the Cons node is put in a strict context .

```
::List a       =    Cons !a (List a)
               |    Nil
```

Example (spine strict list).

```
::List2 a      =    Cons2 a !(List2 a)
               |    Nil2
```

Example (a complex number as record type with strict components).

```
::Complex      =    {    re :: !Real,
                         im :: !Real }

(+) infixl 6 :: !Complex !Complex -> Complex
(+) {re=r1,im=i1} {re=r2,im=i2} = {re=r1+r2,im=i1+i2}
```

8.5.4 Strictness annotations on array instances

For reasons of efficiency there are different types of arrays predefined. One can define a lazy array (default, of type `{a}`), a strict array (explicitly type the array as `{!a}`), and an unboxed one (explicitly type the array as `{#a}`, works only on elements of basic value). When put in a strict context, all the elements of a strict array will be evaluated automatically. As usual one has to take care that the elements do not represent an infinite computation. Lazy, strict and unboxed arrays are regarded to be of different type even if the array elements are of the same type. So, in principle one cannot offer e.g. a strict array to a function demanding a lazy one, and the other way around. Both will give rise to a type error. However, by using the overloading mechanism one can define functions which work on any kind of array (see 4.9).

Example (strict and non-strict arrays). `ArrayA` is a strict one and `ArrayB` is a lazy one. The function `Scale` expects a lazy one and can therefore only be applied on a lazy array. If one wants to define a function which works on any kind of array of Reals, one has to define an overloaded function (see 4.9) like `Scale2`.

```
ArrayA :: {Real}
ArrayA = {1.0,2.0,3.0}

ArrayB :: {!Real}
ArrayB = {1.0,2.0,3.0}

Scale :: {Real} Real -> *{Real}
Scale lazy_array factor = {factor * e \\ e <-: lazy_array}

Scale2 :: (a Real) Real -> *(a Real) | Array a
Scale2 any_array factor = {factor * e \\ e <-: any_array}
```

8.5.5 Strictness annotations on tuple instances

Tuples are predefined algebraic data structures that make it possible to combine several results of arbitrary type into one structure. One can define strict tuples, in the same way as defining strict arrays. This can be done by putting strictness annotations in the type instance on the tuple elements that one would like to make strict. When the corresponding tuple is put into a strict context the tuple and the strict annotated tuple elements will be evaluated.

Strictness annotation can be put on any tuple element of any tuple instance. Such an instance can occur in any type definition (also in a synonym type). The meaning of these annotated synonym types can be explained with the aid of a simple program transformation with which all occurrences of these synonym types are replaced by their right-hand sides (of course, annotations included).

As with arrays, strict and lazy tuples are actually regarded to be of different type. However, unlike is the case with arrays, the compiler will automatically convert strict tuples into lazy ones, and the other way around. This is done for programming convenience. Due to the complexity of this automatic transformation, the conversion is done for tuples only! For the programmer it means that he can freely mix strict and lazy tuples. The type system will not complain when a strict tuple is offered while a lazy tuple is required. The compiler will automatically insert code to convert non-strict tuples into strict version and backwards whenever this is needed.

Example (a complex number as tuple type with strict components).

```
::Complex :== (!Real,!Real)

(+) infixl 6 :: !Complex !Complex -> Complex
(+) (r1,i1) (r2,i2) =   (r1+r2,i1+i2)
```

which is equivalent to

```
(+) infixl 6 :: !(!Real,!Real) !(!Real,!Real) -> (!Real,!Real)
(+) (r1,i1) (r2,i2) =   (r1+r2,i1+i2)
```

when for instance G is defined as

```
G :: Int -> (Real,Real)
```

than the following application is approved by the type system:

```
Start = G 1 + G 2
```

9

Defining uniqueness types

Although CLEAN is purely functional, operations with side-effects (I/O operations, for instance) are permitted. To achieve this without violating the semantics, the classical types are supplied with so called uniqueness attributes. If an argument of a function is indicated as unique, it is guaranteed that at run-time the corresponding actual object is local, i.e. there are no other references to it. Clearly, a destructive update of such a “unique object” can be performed safely.
The uniqueness type system makes it possible to define direct interfaces with an operating system, a file system (updating persistent data), with GUI's libraries, it allows to create arrays, records or user defined data structures that can be updated destructively. The time and space behaviour of a functional program therefore greatly benefits from the uniqueness typing (see 13.6).
Uniqueness types are deduced automatically. Type attributes are polymorphic: attribute variables and inequalities on these variables can be used to indicate relations between and restrictions on the corresponding concrete attribute values.
Sometimes the inferred type attributes give some extra information on the run-time behaviour of a function. The uniqueness type system is a transparent extension of classical typing which means that if one is not interested in the uniqueness information one can simply ignore it.

9.1 Uniqueness typing

Since the uniqueness typing is a rather complex matter we explain this type system and the motivation behind it in more detail. The first section (9.1) explains the basic motivation for and ideas behind uniqueness typing. Section 9.2 focusses on the so-called uniqueness propagation property of (algebraic) type constructors. Thenwe show how new data structures can be defined containing unique objects (section 9.3). Sharing may destroy locality properties of objects. In section 9.4 we describe the effect of sharing on uniqueness types. In order to maintain referential transparency, it appears that function types have to treated specially. The last section (9.5) describes the combination of uniqueness typing and overloading. Especially, the subsections on constructor classes and higher-oder type definitions are very complex: we suggest that the reader skips these sections at first instance.

9.2 Basic ideas behind uniqueness typing

The *uniqueness typing* is an extension of classical Milner/Mycroft typing. In the uniqueness type system *uniqueness type attributes* are attached to the classical types (see Chapter 8). Uniqueness type attributes appear in the *type specifications of functions* (see 9.3) but are also permitted in the definitions of *new data types* (see 9.2). A classical type can be prefixed by one of the following uniqueness type attributes:

```
Type                 =  {BrackType}+
BrackType            =  [TypeAttrib] SimpleType
UnqTypeAttrib        =  *                                   //  type attribute "unique"
```

```
| AttribVariable:        // a type attribute variable
| .                      // an anonymous type attribute variable
```

The basic idea behind uniqueness typing is the following. Suppose a function, say F, has a unique argument (an argument with type $*\sigma$, for some σ). This attribute imposes an additional restriction on applications of F.

- It is *guaranteed* F will have private ("unique") access to this particular argument (see Barendsen and Smetsers, 1993; Plasmeijer and Van Eekelen, 1993): the object will have a reference count of 1[1] *at the moment* it is inspected by the function. It is important to know that there can be more than 1 reference to the object before this specific access takes place. If a uniquely typed argument is not used to construct the function result it will become garbage (the reference has dropped to zero). Due to the fact that this analysis is performed statically the object can be garbage collected (see Chapter 2) at compile-time. It is harmless to reuse the space occupied by the argument to create the function result. In other words: *it is allowed to update the unique object destructively without any consequences for referential transparency.*

Example: the I/O library function `fwritec` is used to write a character to a file yielding a new file as result. In general it is semantically not allowed to overwrite the argument file with the given character to construct the resulting file. However, by demanding the argument file to be unique by specifying

```
fwritec :: Char *File -> *File
```

it is guaranteed by the type system that `fwritec` has private access to the file such that overwriting the file can be done without violating the functional semantics of the program. The resulting file is unique as well and can therefore be passed as continuation to another call of e.g. `fwritec` to make further writing possible.

```
WriteABC :: *File -> *File
WriteABC tofile = fwritec 'c' (fwritec 'b' (fwritec 'a' tofile))
```

Observe that a unique file is passed in a single threaded way (as a kind of unique token) from one function to another where each function can safely modify the file knowing that is has private access to that file. One can make these intermediate files more vissible by by writing the `WriteABC` as follows.

```
WriteABC tofile = file3
where
     file1 = fwritec 'a' tofile
     file2 = fwritec 'b' file1
     file3 = fwritec 'c' file2
```

or, alternatively (to avoid the explicit numbering of the files),

```
WriteABC tofile
     #    file = fwritec 'a' tofile
          file = fwritec 'b' file
     =    fwritec 'c' file
```

The type system makes it possible to make no distinction between a CLEAN file and a physical file of the real world: file I/O can be treated as efficiently as in imperative languages.
The uniqueness typing prevents writing while other readers/writers are active. E.g. one cannot apply `fwritec` to a file being used elsewhere

For instance, the following expression is *not* approved by the type system:

```
(file, fwritec 'a' file)
```

- Function arguments with no uniqueness attributes are considered as “non-unique”: there are no reference requirements for these arguments. The function is only allowed to have *read access* (as usual in a functional language) even if in some of the function applications to actual argument appears to have reference count 1.

---

[1] Note that it is very natural in Clean to speak about references due to the underlying graph rewriting semantics of the language: it is always clear when objects are being shared or when cyclic structures are being created.

```
freadc :: File -> (Char, File)
```

The function `freadc` can be applied to both a unique as well as non-unique file. This is fine since the function only wants read access on the file. The type indicates that the result is always a non-unique file. Such as file can be passed for further reading, but not for further writing.

- To indicate that functions don't change uniqueness properties of arguments, one can use *attribute variables*. The most simple example is the identity functions that can be typed as follows:

  ```
  id :: u:a -> u:a
  ```

  Here a is an ordinary type variable, whereas u is an attribute variable. If id is applied to an unique object the result is also unique (in that case u is instantiated with the concrete attribute *). Of course, if id is applied to a non-unique object, the result remains non-unique. As with ordinary type variables, attribute variables should be instantiated uniformly.

  A more interesting example is the function freadc which is typed as

  ```
  freadc :: u:File -> u:(Char, u:File)
  ```

  Again `freadc` can be applied to both unique and non-unique files. In the first case the resulting file is also unique and can, for example, be used for further reading or writing. Moreover, observe that not only the resulting file is attributed, but also the tuple containing that file and the character that has been read. This is due to the so called *uniqueness propagation rule*; see below.

To summarize, uniqueness typing makes it possible to update objects destructively within a purely functional language. For the development of real world applications (which manipulate files, windows, arrays, databases, states etc.) this is an indispensable property.

9.3 Attribute propagation

Having explained the general ideas of uniqueness typing, we can now focus on some details of this typing system.

If a unique object is stored in a data structure, the data structure itself becomes unique as well. This *uniqueness propagation rule* prevents that unique objects are shared indirectly via the data structure in which these objects are stored. To explain this form of hidden sharing, consider the following definition of the function head

```
head :: [*a] -> *a
head [hd:tl] = hd
```

The pattern causes head to have access to the "deeper" arguments hd and tl. Note that head does not have any uniqueness requirements on its direct list argument. This means that in an application of head the list might be shared, as can be seen in the following function heads

```
heads list = (head list, head list)
```

If one wants to formulate uniqueness requirements on, for instance, the hd argument of head, it is not sufficient to attribute the corresponding type variable a with *; the surrounding list itself should also become unique. One can easily see that, without this additional requirement the heads example with type

```
heads :: [*a] -> (*a,*a)
heads list = (head list, head list)
```

is still valid although it delivers the same object twice. By making the list itself unique, (so the type op head becomes head :: *[*a] -> *a) the function is rejected. In general one could say that uniqueness *propagates outwards*.

Some of the readers will have noticed that, by using attribute variables, one can assign a more general uniqueness type to head:

```
head :: u:[u:a] -> u:a
```

The above propagation rule imposes additional (implicit) restrictions on the attributes appearing in type specifications of functions.

Another explicit way of indicating restrictions on attributes is by using *coercion statements.* These statements consist of attribute variable inequalities of the form u <= v. The idea is that attribute substitutions are only allowed if the resulting attribute inequalities are valid, i.e. not resulting in an equality of the form

‘non-unique ≤ unique’.

The use of coercion statements is illustrated by the next example in which the uniqueness type of the well-known append function is shown.

```
append :: v:[u:a] w:[u:a] -> x:[u:a],           [v<=u, w<=u, x<=u,w<=x]
```

The first three coercion statements express the uniqueness propagation for lists: if the elements a are unique (by choosing * for u) these statements force v,w and x to be instantiated with * also. (Note that u <= * iff u = *.) The statement w<=x expresses that the spine uniqueness of append’s result depends only on the spine attribute w of the second argument.

In CLEAN it is permitted to omit attribute variables and attribute inequalities that arise from propagation properties; these will be added automatically by the type system. As a consequence, the following type for append is also valid.

```
append :: [u:a] w:[u:a] -> x:[u:a],        [w<=x]
```

Of course, it is always allowed to specify a more specific type (by instantiating type or attribute variables). All types given below are valid types for append.

```
append :: [u:a] x:[u:a] -> x:[u:a],
append :: *[*Int] *[*Int] -> *[*Int],
append :: [a] *[a] -> *[a].
```

To make types more readable, CLEAN offers the possibility to use *anonymous* attribute variables. These can be used as a shorthand for indicating attribute variables of which the actual names are not essential. This allows us to specify the type for append as follows.

```
append :: [.a] w:[.a] -> x:[.a],           [w<=x]
```

The type system of CLEAN will substitute real attribute variables for the anonymous ones. Each dot gives rise to a new attribute variable except for the dots attached to type variables: type variables are attributed uniformly in the sense that all occurrences of the same type variable will obtain the same attribute. In the above example this means that all dots are replaced by one and the same new attribute variable.

9.4 Defining new types with uniqueness attributes

Although one mostly uses uniqueness attributes in type specifications of functions, they are also allowed in the definition of new data types.

AlgebraicTypeDef = **::**TypeLhs **=** ConstructorDef {**|**ConstructorDef} ;

TypeLhs = [*****]TypeConstructor {[*****] *TypeVariable*}
TypeConstructor = *TypeSymb*

ConstructorDef = [*QuantifiedVariables* **:**] *ConstructorSymb* {[Strict] BrackType}
| [*QuantifiedVariables* **:**] **(***ConstructorSymb***)** [Fix][Prec] {[Strict] BrackType}

QuantifiedVariables = {**E.** *TypeVariable*}+
BrackType = [UnqTypeAttrib] SimpleType
UnqTypeAttrib = **`*`** | **`.`**

As can be inferred from the syntax, the attributes that are actually allowed in data type definitions are '`*`' and '`.`'; attribute variables are not permitted. The (unique) `*` attribute can be used at any subtype whereas the (anonymous) `.` attribute is restricted to non-variable positions.

If no uniqueness attributes are specified, this does not mean that one can only build non-unique instances of such a data type. Attributes not explicitly specified by the programmer are added automatically by the type system. To explain this standard uniqueness attribution mechanism, first remember that the types of data constructors are not specified by the programmer but derived from their corresponding data type definition. For example, the (classical) definition of the `List` type

```
:: List a = Cons a (List a) | Nil
```

leads to the following types for its data constructors.

```
Cons :: a (List a) -> List a
Nil :: List a
```

To be able to create unique instances of data types, the standard attribution of CLEAN will derive appropriate uniqueness variants for the types of the corresponding data constructors. Such a uniqueness variant is obtained via a consistent attribution of all types and subtypes appearing in a data type definition. Here, consistency means that such an attribution obeys the following rules (assume that we have a type definition for some type `T`).

- Attributes that are explicitly specified are adopted.
- Each (unattributed) type variable and each occurrence of `T` will receive an attribute variable. This is done in a uniform way: equal variables will receive equal attributes, and all occurrence of `T` are also equally attributed.
- Attribute variables are added at non-variable positions if they are required by the propagation properties of the corresponding type constructor. The attribute variable that is chosen depends on the argument types of this constructor: the attribution scheme takes the attribute variable of first argument appearing on a propagating position (see example below).
- All occurrences of the `.` attribute are replaced by the attribute variable assigned to the occurrences of `T`.

Examples

For `Cons` the standard attribution leads to the type

```
Cons :: u:a v:(List u:a) -> v:List u:a, [v<=u]
```

The type of `Nil` becomes

```
Nil :: v:List u:a, [v<=u]
```

Consider the following Tree definition

```
:: Tree a     =   Node a [Tree a]
```

The type of the data constructor `Node` is

```
Node :: u:a v:[v:Tree u:a] -> v:Tree u:a, [v<=u]
```

Changing the `Tree` type definition as follows

```
:: Tree a     =   Node a .[Tree a]
```

results in the same type for `Node` (remember that the . attribute stands for the attribute assigned to the occurrences of `Tree`).

Another `Tree` variant.

```
:: Tree *a     =    Node *a [Tree *a]
```

leading to

```
Node :: *a *[*Tree *a] -> *Tree *a
```

Note that, due to propagation, all subtypes have become unique.

Next, we will formalize the notion of uniqueness propagation. We say that an argument of a type constructor, say `T`, is propagating if the corresponding type variable appears on a propagating position in one of the types used in the right-hand side of `T`'s definition. A propagating position are characterized by the fact that they it is not surrounded by an arrow type or by a type constructor with non-propagating arguments. Observe that the definition of propagation is cyclic: a general way to solve this problem is via a fixedpoint construction; see also ???.

Example (propagation rule). Consider the (record) type definition for `Object`.

```
Object a b :: {    state :: a, fun :: b -> a }
```

The argument `a` is propagating. Since `b` does not appear on a propagating position inside this definition, `Object` is not propagating in its second argument.

9.5 Uniqueness and sharing

The type inference system of CLEAN will derive uniqueness information *after* the classical Milner/Mycroft types of functions have been inferred (see 8.3). As explained in Section 9.1, a function may require a non-unique object, a unique object or a possibly unique object. Uniqueness of the result of a function will depend on the attributes of its arguments and how the result is constructed. Until now, we distinguished objects with reference count 1 from objects with a larger reference count: only the former are might be unique (depending on the uniqueness type of the object itself). In practice, however, one can be more liberal if one takes the evaluation order into account. The idea is that multiple reference to an (unique) object are harmless if one knows that only one of the references will be present at the moment it is accessed destructively. This has been used in the following function.

```
AppendAorB:: *File -> *File
AppendAorB file
  | fc == 'a'      = fwritec 'a' file
                   = fwritec 'b' file
where
  (fc,nf)          = freadc file
```

When the right-hand side of `AppendAorB` is evaluated, the guard is determined first (so access from `freadc` to `file` is not unique), and subsequently one of the alternatives is chosen and evaluated. Depending on cond, either the reference from the first `fwritec` application to function `file` or that of the second application is left and therefore unique.

For this reason, the uniqueness type system uses a kind of *sharing analysis*. This sharing analysis is input for the uniqueness type system itself to check uniqueness type consistency (see 9.3.5). The analysis will label each *reference* in the right-hand side of a function definition as *read-only* (if destructive access might be dangerous) or *write-permitted* (otherwise). Objects accessed via a read-only reference are always non-unique. On the other hand, uniqueness of objects accessed via a reference labeled with *write-permitted* solely depend on the types of the objects themselves.

Before describing the labeling mechanism of CLEAN we mention that the “lifetime” of references is determined on a syntactical basis. For this reason we classify references to the same expression in a function definition (say for `f`) according to their estimated run-time use, as *alternative*, *observing* and *parallel*.

- Two references are alternative if they belong to different alternatives of `f`. Note that alternatives are distinguished by patterns (including `case` expressions) or by guards.
- A reference `r` is observing w.r.t. a reference `r`’ if the expression containing `r`’ is either (1) guarded by an expression or (2) preceded by a strict let expression containing `r`.
- Otherwise, references are in parallel.

The rules used by the sharing analysis to label each reference are the following.

- A reference, say `r`, to a certain object is labeled with read-only if there exist another reference, say `r'`, to the same object such that either `r` is observing w.r.t `r'` or `r` and `r'` are in parallel.
- Multiple references to *cyclic structures* are always labeled as read-only.
- All other references are labeled with write-permitted.

Unfortunately, there is still a subtlety that has to be dealt with. Observing references belonging to the second category (strict let expressions) do not always vanish totally after the expression containing the reference has been evaluated: further analysis appears to be necessary to ensure their disappearance. More concretely, Suppose `e[r]` denotes the expression containing `r`. If the type of `e[r]` is a basic type then, after evaluation, `e[r]` will be reference-free. In particular, it does not contain the reference `r` anymore. However, If the type of `e[r]` is not a basic type it is assumed that, after evaluation, `e[r]` might still refer to `r`. But even in the latter case a further refinement is possible. The idea is, depending on `e[r]`, to correct the type of the object to which `r` refers partially in such way that only the parts of this object that are still shared lose their uniqueness.

Consider, for example, the following rule

```
f l =
    let!
        x = hd (hd l)
    in
        (x, l)
```

Clearly, `x` and `l` share a common substructure; `x` is even part of `l`. But the whole “spine” of `l` (of type `[[...]]`) does not contain any new external references. Thus, if `l` was spine-unique originally, it remains spine unique in the result of `f`. Apparently, the access to `l` only affected part of `l`’s structure. More technically, the type of `l` itself is corrected to take the partial access on l into account. In the previous example, `x`, regarded as a function on `l` has type `[[a]] -> a`. Int `f`’s definition the part of `l`’s type corresponding to the variable `a` is mode non-unique. This is clearly reflected in the derived type for `f`, being

```
f :: u:[w:[a]] -> (a,v:[x:[a]]), [w <= x, u <= v]
```

In CLEAN this principle has been generalized: If the strict let expression `e[r]` regarded as a function on `r` has type

```
T (... a ...) -> a
```

Then the a-part of the type of the object to which r refers becomes non-unique; the rest of the type remains unaffected. If the type of `e[r]` is not of the indicated form, `r` is not considered as an observing reference (w.r.t. some reference `r'`), but, instead, as in parallel with `r'`.

9.5.1 Higher order uniqueness typing

Higher-order functions give rise to partial (often called *Curried*) applications, i.e. applications in which the actual number of arguments is less than the arity of the corresponding symbol. If these partial applications contain unique sub-expressions one has to be careful. Consider, for example the following the function `fwritec` with type

```
fwritec :: *File Char -> *File
```

in the application

```
fwritec unifile
```

(assuming that `unifile` returns a unique file). Clearly, the type of this application is of the form `o:(Char -> *File)`. The question is: what kind of attribute is `o`? Is it a variable, is it `*` is it not unique. Before making a decision, one should notice that it is dangerous to allow the above application to be shared. For example, if the expression `fwritec unifile` is passed to a function

```
WriteAB write_fun = (write_fun ‘a’, write_fun ‘b’)
```

Then the argument of `fwritec` is not longer unique at the moment one of the two write operations take place. Apparently, the `fwritec unifile` expression is *essentially* unique: its reference count should never become greater than 1. To prevent such an essentially unique expression from being copied, CLEAN considers the `->` type constructor in combination with the `*` attribute as special: it is not permitted to discard its uniqueness. Now, the question about the attribute `o` can be answered: it is set to `*`. If `WriteAB` is typed as follows

```
WriteAB :: (Char -> u:File) -> (u:File, u:File)
WriteAB write_fun = (write_fun ‘a’, write_fun ‘b’)
```

the expression `WriteAB (fwritec unifile)` is rejected by the type system because it does not allow the argument of type `*(Char -> *File)` to be coerced to `(Char -> u:File)`. One can easily see that it is impossible to type `WriteAB` in such a way that the expression becomes typable.

To define data structures containing Curried applications it is often convenient to use the (anonymous) . attribute. Example

```
:: Object a b = { state :: a, fun :: .(b -> a) }

new :: * Object *File Char
new = { state = unifile, fun = fwritec unifile }
```

By adding an attribute variable to the function type in the definition of `Object`, it is possible to store unique functions in this data structure. This is shown by the funcion `new`. Since `new` contains an essentially unique expression it becomes essentially unique itself. So, `new` can never loose its uniqueness, and hence, it can only be used in a context in which a unique object is demanded.

Determining the type of a Curried application of a function (or data constructor) is somewhat more involved if the type of that function contains attribute variables instead of concrete attributes. Mostly, these variables will result in additional coercion statements. as can be seen in the example below.

```
Prepend :: u:[.a] [.a] -> v:[.a],      [u<=v]
Prepend a b = Append b a

PrependList :: u:[.a] -> w:([.a] -> v:[.a]),         [u<=v, w<=u]
PrependList a = Prepend a
```

Some explanation is in place. The expression (`PrependList some_list`) yields a function that, when applied to another list, say `other_list`, delivers a new list extended consisting of the concatenation of `other_list` and `some_list`. Let’s call this final result `new_list`. If `new_list` should be unique (i.e. `v` becomes `*`) then, because of the coercion statement `u<=v` the attribute `u` also becomes `*`. But, if `u = *` then also `w = *`, for, `w<=u`. This implies that (arrow) type of the original expression (`PrependList some_list`) becomes unique, and hence this expression is cannot be shared. The general rule for determining the uniqueness type of Curried variants of (function or data) symbols can be found in ???

9.5.2 Uniqueness type coercions

As said before, offering a unique object to a function which requires a non-unique argument is safe (unless we are dealing with unique arrow types; see above). The technical tool to express this is via a coercion (subtype) relation based on the ordering

‘unique’ ≤ ‘non-unique’

on attributes. Roughly, the validity of σ ≤ σ’ depends subtype-wise on the validity of `u≤u’` with `u,u’` attributes in σ,σ’. One has, for example

$$u{:}[v{:}[w{:}\mathrm{Int}]] \leq u'{:}[v'{:}[w'{:}\mathrm{Int}]] \text{ iff } u \leq u',\ v \leq v',\ w \leq w'.$$

However, a few refinements are necessary. Firstly, the uniqueness constraints expressed in terms of coercion statements (on attribute variables) have to be taken into account. Secondly, the coercion restriction on arrow types should be handled correctly. And thirdly, due to the so-called *contravariance* of `->` in its first argument we have that

$$u{:}(\sigma \text{ -> } \sigma') \leq u{:}(\tau \text{ -> } \tau') \text{ iff } \tau \leq \sigma,\ \sigma' \leq \tau'$$

Since `->` may appear in the definitions of algebraic type constructors, these constructors may inherit the co- and contravariant subtyping behaviour with respect to their arguments. We can classify the 'sign' of the arguments of each type constructor as + (positive, covariant), - (negative, contravariant) or top (both positive and negative). In general this is done by analysing the (possible mutually recursive) algebraic type definitions by a fixedpoint construction, with basis *sign*(`->`) = (-,+).

Example: `a` has sign ⊤, `b` has sign + in

```
::FunList a b =   FunCons (a, a -> b) (FunList a b)
              |   FunNil
```

This leads to the following coercion rules

- Attributes of two corresponding type variables as well as of two corresponding arrow types must be equal.
- The sign classification of each type constructor is obeyed. If, for instance, the sign of T's argument is negative, then
  $$T\ \sigma \leq T\ \sigma' \text{ iff } \sigma' \leq \sigma$$
- In all other cases, the validity of a coercion relation depends on the validity of $u \leq u'$, where u,u' are attributes of the two corresponding subtypes.

The presence of sharing inherently causes a (possibly unique) object to become non-unique, if it is accessed via a read-only reference. In CLEAN this is achieved by a type correction operation which converts each unique type S to its smallest non-unique supertype, simply by making the outermost attribute of S non-unique. Note that this operation fails if S is a function type.

9.6 Combining uniqueness typing and type classes

An overloaded function actually stands for a collection of real functions. The types of these real functions are obtained from the type of the overloaded function by substituting the corresponding instance type for the class variable. These instance types may contain uniqueness information, and, due to the propagation requirement, the above-mentioned substitution might give rise to uniqueness attributes overloaded type specification.
Consider, for instance, the identity class

```
class id a :: a -> a
```

If we want to define an instance of `id` for lists, say `idL`, which leaves uniqueness of the list elements intact, the (fully expanded) type of `idL` becomes

```
class idL v:[u:a] -> v:[u:a]
```

However, as said before, the type specification of such an instance is not specified completely: it is derived from the overloaded type in combination with the instance type (i.e. `[...]` in this particular example).

In CLEAN we require that the type specification of an overloaded operator anticipates on attributes arising from uniqueness propagation, that is, the uniqueness attribute of the class variable should be chosen in such a way that for any instance type this `class attribute' does not conflict with the corresponding uniqueness attribute(s) in the fully expanded type of this instance. In the above example this means that the type of `id` becomes

```
class id a :: a -> a
```

Another possibility is

```
class id a :: *a -> *a
```

However, the latter version of `id` will be more restrictive in its use, since it will always require that its argument is unique.

9.6.1 Constructor classes

The combination of uniqueness typing and constructor classes (with their higher-order class variables) introduces another difficulty. Consider, for example, the overloaded map function.

```
class map m :: (a -> b) (m a) -> m b
```

Suppose we would add (distinct) attribute variables to the type variables `a` and `b` (to allow `unique instances' of `map`)

```
class map m :: (.a -> .b) (m .a) -> m .b
```

The question that arises is: Which attributes should be added to the two applications of the class variable `m`? Clearly, this depends on the actual instance type filled in for `m`. E.g., if `m` is instantiated with a propagating type constructor (like []), the attributes of the applications of `m` are either attribute variables or the concrete attribute ‘unique’. Otherwise, one can chose anything.

Example

```
instance map []
where
     map f l = [ f x // x <- l ]

::   T a = C (Int -> a)

instance map T
where
     map f (C g) = C (f o g)
```

In this example, the respective expanded type of the both instances are

```
map :: (u:a -> v:b) w:[u:a] -> x:[v:b], w <= u, x <= v

map :: (u:a -> v:b) (T u:a) -> T v:b
```

The type system of CLEAN requires that a possible propagation attribute is explicitly indicated in the type specification of the overloaded function. In order to obtain versions of `map` producing spine unique data structures, its overloaded type should be specified as follows:

```
class map m :: (.a -> .b) .(m .a) -> .(m .b)
```

This type will provide that for an application like

```
map inc [1,2,3]
```

indeed yields a spine unique list.

Observe that if you would omit the (anonymous) attribute variable of the second argument, the input data structure cannot contain unique data on propagating positions, e.g. one could not use such a version of `map` for mapping a destructive write operator on a list of unique files.

In fact, the propagation rule is used to translate uniqueness properties of objects into uniqueness properties of the data structures in which these objects are stored. As said before, in some cases the actual data structures are unknown.

Consider the following function

```
DoubleMap f l = (map f l, map f l)
```

The type of this function is something like

```
DoubleMap :: (.a -> .b) (m .a) -> (.(m .b), .(m .b))
```

Clearly, `l` is duplicated. However, this does not necessarily mean that a cannot be unique anymore. If, for instance, `m` is instantiated with a non-propagating type constructor (like T as defined on the previous page) then uniqueness of a is still permitted. On the other hand, if `m` is instantiated with a propagating type constructor, a unique instantiation of a should be disapproved. In CLEAN, the type system `remembers' sharing of objects (like `l` in the above example) by making the corresponding type attribute non-unique. Thus, the given type for `DoubleMap` is exactly the type inferred by CLEAN's type system. If one tries to instantiate `m` with a propagating type constructor, and, at the same type, `a` with some unique type, this will fail.

The presence of higher-order class variables, not only influences propagation properties of types, but also the coercion relation between types. These type coercions depend on the sign classification of type constructors. The problem with higher-order polymorphism is that in some cases the actual type constructors substituted for the higher order type variables are unknown, and therefore one cannot decide whether coercions in which higher-order type variable are involved, are valid.

Consider the functions

```
double x = (x,x)
dm f l = double (map f l)
```

Here, `map`'s result (of type `.(m .a)`) is coerced to the non-unique supertype `(m .a)`. However, this is only allowed if `m` is instantiated with type constructors that have no coercion restrictions. E.g., if one tries to substitute `*WriteFun` for `m`, where

```
WriteFun a = C .(a -> *File)
```

this should fail, for, `*WriteFun` is *essentially* unique. The to solve this problem is to restrict coercion properties of type variable applications `(m σ)` to

$$u:(m\ \sigma) \leq u:(m\ \tau) \text{ iff } \sigma \leq \tau \ \&\&\ \tau \leq \sigma$$

A slightly modified version of this solution has been adopted in CLEAN. For convenience, we have added the following refinement. The instances of type constructors classes are restricted to type constructors with no coercion restrictions. Moreover, it is assumed that these type constructors are uniqueness propagating. This means that the `WriteFun` cannot be used as an instance for `map`. Consequently, our coercion relation we can be more liberal if it involves such class variable applications.

Overruling this requirement can be done adding the anonymous attribute . the class variable. E.g.

```
class map .m :: (.a -> .b) .(m .a) -> .(m .b)
```

Now

```
instance map WriteFun
where
     map ..
```

is valid, but the coercions in which (parts of) `map`'s type are involved are now restricted as explained above.

To see the difference between the two indicated variants of constructor variables, we slightly modify `map`'s type.

```
class map m :: (.a -> .b) *(m .a) -> .(m .b)
```

Without overruling the instance requirement for `m` the type of `dm` (`dm` as given on the previous page) becomes.

```
dm :: (.a -> .b) *(m .a) -> .(m b, m b)
```

Observe that the attribute of disappeared due to the fact that each type constructor substituted for `m` is assumed to be propagating.

If one explicitly indicates that there are no instance restriction for the class variable `m` (by attributing `m` with .), the function `dm` becomes untypable.

9.6.2 Higher-order type definitions

We will describe the effect of uniqueness typing on type definitions containing higher-order type variables. At it turns out, this combination introduces a number of difficulties which would make a full description very complex. But even after skipping a lot of details we have to warn the reader that some of the remaining parts are still hard to understand.

As mentioned earlier, two properties of newly defined type constructor concerning uniqueness typing are important, namely, propagation and sign classification. One can probably image that, when dealing with higher-order types the determination on these properties becomes more involved. Consider, for example, the following type definition.

```
::  T m a = C (m a)
```

The question whether `T` is propagating in its second argument cannot be decided by examining this definition only; it depends on the actual instantiation of the (higher-order) type variable `m`. If `m` is instantiated with a propagating type constructor, like `[]`, then `T` becomes propagating in its second argument as well. Actually, propagation is not only a property of type constructors, but also of types themselves, particularly of `partial types' For example, the partial type `[]` is propagating in its (only) argument (Note that the number of arguments a partial type expects, directly follows from the kinds of the type constructors that have been used). The type `T []` is also propagating in its argument, so is the type `T ((,) Int))`.

The analysis in CLEAN that determines propagation properties of (partial) types has been split into two phases. During the first phase, new type definitions are examined in order to determine the propagation dependencies between the arguments of each new type constructor. To explain the idea, we return to our previous example.

```
::  T m a = C (m a)
```

First observe that the propagation of the type variable `m` is not interesting because `m` does not stand for `real data' (which is always of kind `*`). We associate the propagation of `m` in `T` with the position(s) of the occurrence(s) of `m`'s applications. So in general, `T` is propagating in a higher-order variable `m` if one of `m`'s applications appears on a propagating position in the definition of `T`. Moreover, for each higher order type variable, we determine the propagation properties of all first order type variables in the following way: `m` is propagating in `a`, where `m` and `a` are higher-order respectively first-order type variables of `T`, if `a` appears on a propagating position in one of `m`'s applications. In the above example, `m` is propagating in `a`, since `a` is on a propagating position in the application `(m a)`. During the second phase, the propagation properties of (partial) types are determined using the results of the first phase. This (roughly) proceeds as follows. Consider the type `T` σ for some (partial) type σ, and `T` as defined earlier. First, determine (recursively) the propagation of σ. Then the type `T` σ is propagating if (1) σ is propagating, (2) `T` is propagating in `m`, and moreover (3) `m` is propagating in `a` (the second argument of the type constructor). With `T` as defined above, (2) and (3) are fulfilled. Thus, for example `T []` is propagating and therefore also `T (T [])`. Now define

```
::  T2 a = C2 (a -> Int)
```

The `T T2` is not propagating.

The adjusted uniqueness propagation rule (see also ...) becomes:

- Let σ,τ be two uniqueness types. Suppose σ has attribute `u`. Then, if τ is propagating the application (τ σ) should have an attribute `v` such that `v` ≤ `u`.

Some of the readers might have inferred that this propagation rule is a ‘higher-order’ generalization of the old ‘first-order’ propagation rule.

As to the sign classification, we restrict ourselves to the remark that that sign analysis used in CLEAN is adjusted in a similar way as described above for the propagation case.

Example

```
::  T m a = C ((m a) -> Int)
```

The sign classification of `T` if (-,). Here denotes the fact the `a` is neither directly used on a positive nor on a negative position. The sign classification of `m` w.r.t. `a` is +. The partial type `T []` has sign -, which e.g. implies that

```
T [] Int ≤ T [] *Int
```

The type `T T2` (with `T2` as defined on the previous page) has sign +, so

```
T T2 Int ≥ T T2 *Int
```

It will be clear that combining uniqueness typing with higher-order types is far from trivial: the description given above is complex and moreover incomplete. However explaining all the details of this combination is far beyond the scope of the reference manual

10

Input / Output handling

In this Chapter the new CLEAN I/O system version 1.0 is described. This system is currently available only on a limited number of platforms (see the Preface).

On other systems the CLEAN I/O system version 0.8 is distributed. On all platforms the CLEAN 0.8 I/O library (albeit converted to CLEAN 1.0 syntax) is available. For a description of the 0.8 I/O library we refer to the draft of the new CLEAN book on the net or to the Addison-Wesley book (Plasmeijer and Van Eekelen, 1993).

CLEAN's Uniqueness Type System makes it possible to update objects destructively. As explained in Section 9.1 one can use this property to create CLEAN functions which have direct read and write access on files. In the same way one can define functions for all communication with the outside world: for file I/O, window based I/O, communication with the operating system, interface with C etc.

Since we want CLEAN programmers to write programs on a high level of abstraction in a declarative style, we wanted to offer more than just an interface to C. We do not want to burden the programmer with the low level details of how I/O is handled on a specific platform. To make this possible a sophisticated I/O system has been predefined in CLEAN. It provides a way for the programmer to specify interactive programs on such a way that window based interactive programs can be developed very easily. *All* low-level event handling and window management is handled *automatically*. The specification is platform independent. Programs can be ported to other machines *without modification of code* while the resulting program will obey the specific look and feel offered by the underlying operating system. Although the I/O system cannot support everything one can imagine, it is powerful enough for most applications. The I/O system can also be extended or modified easily by the (system) programmer to support wishes we did not think of.

I/O handling in CLEAN is done via an *explicit multiple environment passing* scheme to enforce the correct order of evaluation (see 10.1) while destructive updateability in a pure functional language is realised by using uniqueness typing (see Chapter 9).

Files can be *directly* accessed for reading and writing (see 10.2). Graphical User Interfaces can be specified by defining abstract devices using a predefined algebraic data type (see 10.3 and 10.4). Timers can be defined to perform time dependent actions (see 10.5).

The system offers the possibility to *combine* (independently developed) interactive applications (processes) into one new CLEAN application. The different sub-applications are executed in an *interleaved* manner (see 10.7). One can switch between these sub-applications (like in a multi-finder) and exchange information between them. Sub-applications can communicate via *files* (e.g. via copy-pasting), via shared *global states* or via *message passing* (see 10.7).

It is in principle also possible to create sub-applications running on a different processor. In this way distributed applications can be made running in parallel on different machines connected via a network. Such distributed programs can be tested and developed on one processor and with one change in the code and a recompilation turned into the desired distributed version. This is one of the future extensions of the I/O library and is not yet available.

10.1 The world according to CLEAN

CLEAN programs can run in two modes.

10.1.1 I/O using the console

The first mode is a *console mode*. It is chosen when the `Start` rule is defined as a *nullary* function.

```
Start :: TypeOfStartFunction
Start = …                                         //    initial expression
```

In the console mode, that part of the *initial expression* (indicated by the right-hand side of the `Start` rule) which is in *root normal form* (also called the head normal form or root stable form), is printed as soon as possible. The console mode can be used for instance to test functions.

10.1.2 I/O on the unique world

The second mode is the *world mode*. It is chosen when the optional additional parameter (which is of type `*World)` is added to the `Start` rule and delivered as result.

```
Start :: *World -> *World
Start w = …                                       //    initial expression returning a changed world
```

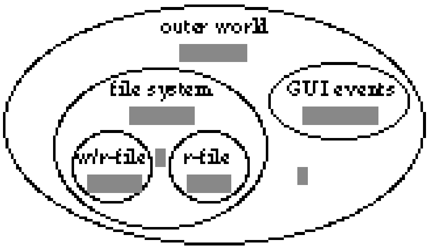
The world which is given to the initial expression is an *abstract data structure*, an *abstract world* of type `*World` which models *the concrete physical world* as seen from the program. The abstract world can in principle contain *anything* we want, anything what is of importance for a functional program to interact during execution with the concrete world. The world can be seen as a *state* and modifications of the world can be realised via *state transition functions* defined on the world or a part of the world. By requiring that these state transition functions work on a *unique* world the modifications of the abstract world can directly be realized in the real physical world, without loss of efficiency and without losing referential transparency (see Chapter 9).

The concrete way in which one can handle the world in CLEAN is determined by the system programmer. One way to handle the world is by using the predefined CLEAN I/O library which can be regarded as a platform independent mini operating system. It makes it possible to do file I/O, window based I/O, dynamic process creation and process communication in a pure functional language in an efficient way.

The program state

For programming convenience the world is further refined in the CLEAN I/O system as follows.

Figure (the world according to CLEAN).



In the CLEAN I/O system the abstract world is divided into unique abstract sub-worlds. Such an *abstract sub-world* models a *part* of the real world which has as property that it can be manipulated independently from another part: one can modify one without influencing another.

- An important sub-world is the *file system* (of type `*Files`) for performing *file I/O* (see 9.2). This idea of sub-worlds can be further refined as required, e.g. one can retrieve a file of type `*File` from the file system. With the *hierarchy of sub-worlds* we can guarantee that things happen in a certain order. E.g. to open a file one first needs the uniquely typed file system, to re-open a file one first has to close it.
- Another important sub-world is the *I/O state* (of type `*IOState l p`) which contains, amongst others, the event queue in which all events intended for the CLEAN application are being stored. The IOState is an abstract data type on which all kinds of operations are defined to handle *event driven* (*window-based*) *I/O*. The abstract data type is parametrised with the type of the *local process state* and the *public process state* (see hereafter).

But, of course, an application does not only manipulate the world, it probably has to manipulate its own data (the *program dependent state*) as well. It is explained in section 10.7 that a CLEAN application can consist of several *interactive processes*. For this reason the program dependent state is split-up into two categories:

- Each interactive process has its own *local process state* containing information which is private for each process.
- With the *public process state* interactive processes which belong to the same group (see 10.7) can exchange information.

Writing an interactive CLEAN program means writing *state transition functions* which manipulate the abstract world and the program dependent states. The four states introduced above are the states on which all top level state transition functions in CLEAN work. These states are collected in one record, the *process state* which is of the following type:

```
::  *PState local public                    // the unique state of an interactive process
     =    {    pLocal    :: !local          // the local (and private) data of the process
          ,    pPublic   :: !public         // the data shared with other processes
                                            // in the same group
          ,    pFiles    :: !*Files         // the unique state of the file system
          ,    pIOState  :: !*IOState local public
                                            // the unique state of the event I/O system
          }
```

Starting and stopping an interactive process

The first thing which generally happens in a CLEAN program is to create an interactive process with the function `OpenIO`. The function `OpenIO` is called a *process control function*. Such a function takes care of all low level event handling in the following way.

- First the *initial process state* is constructed from a specified initial *local process state*, an initial *shared process state* and the initial *world*. The process control function will fetch the initial *event queue* and *file system* from this world.
- The process control function accepts a list of initial state transition functions which are applied one after another on the initial process state. This list typically contains state transition functions with which *abstract devices* are specified (see 10.3) and opened (see 10.4 - 10.6). These descriptions are stored by the process control function in the IOState.
- Now the process control function will repeatedly examine the event queue to see if there is an event on top of the queue matching a description given in one of the stored abstract device specifications. When a matching event is found the corresponding *state transition function* stored in the description (see again 10.3) is applied to the current process state of the program thus yielding a new process state.
- This way of dealing with events continues until finally the predefined state transition function `QuitIO` is applied to the IOState component of the process state after which the process control function (and the interactive process) terminates. From the final process state the final event queue and file system are restored into the world.

So, in an *interactive process* state transition functions defined by the programmer are repeatedly applied to the evolving process state until the final process state has been reached.

The function `OpenIO` is of following type.

```
OpenIO :: (IODef .l .p) (.l,.p) *World -> *World

::   IODef l p
     =    {    ioDefInit      :: InitIO l p  // initial actions of the process
          ,    ioDefAbout     :: String      // name of the process
          }

::   InitIO l p    :==  [(PState l p) -> (PState l p)]
```

QuitIO, the state transition function which stops an interactive process has type:

```
QuitIO :: !(IOState .l .p) -> IOState .l .p
```

Example (a program just starting and stopping an interactive process while doing nothing).

```
module StartAndStop

:: UnUsed = UnUsed                         // Singleton value for unused settings

Start :: *World -> *World
Start world = OpenIO thisprocess (UnUsed,UnUsed) world
where
    thisprocess = { ioDefInit  = [ stop ]
                  , ioDefAbout = "Tiny Process" }
    stop pstate = { pstate & pIOState = QuitIO pstate.pIOState }
```

10.2 File I/O

In the process state the file system of type `*Files` is stored. This *unique* file system gives access to *all* files in the world visible to the program. One can open *writeable files* (they therefore are of type `*File`) or *files* that are *read only* (they have type `File`).

- A file can be opened writeable only if the file is not already open (*run-time* error). A writeable file can be closed and re-opened later on. A file which is opened as read-only can be opened as many times as one likes, but it cannot be closed (and hence it cannot be re-opened as a writeable file). Read-only files are closed automatically by the I/O system when the application terminates or have become garbage.
- When a writeable file becomes shared (loosing its uniqueness property) it can only be used for further reading (it gets the same status as files which are initially opened as read-only).

One can find the predefined functions working on the file system and on the files in this file system in the module `StdFile`.

Example (functions to open and close files). See also `StdFile`.

```
fopen  :: !String !Mode !*Files -> (!Bool,!*File,!*Files)      //   open a writeable file
fclose :: !*File        !*Files -> (!Bool,       !*Files)      //   open a writeable file

sfopen :: !String !Int  !*Files -> (!Bool,!File, !*Files)      //   open a read-only file
```

File I/O is handled very efficiently because the uniqueness typing allows direct access to the actual file. There is no limitation on the kind of file handling which is allowed (e.g. seeks are possible).

```
fwritec :: !Char !*File -> *File               //   directly writes a character into the file
sfreadc :: !File -> (!Bool,!Char,!File)        //   directly reads a character from the file
```

Example (a program that copies a file called `"aap"` to a file called `"noot"`). It uses the file system from the process state. This file system is used to open the source and the destination file. The source file is only being read (indicated by `FReadData`), so it does not have to be unique. The destination file is being written (`FWriteData`) and therefore this file must be unique. The file being written is closed explicitly. Files which are opened read-only are closed automatically by the system (it keeps track of the amount of references to such a file). Notice that the process state is uniquely used everywhere due to the uniqueness of the file system and IOState.

```
module copyfile

import StdEnv, StdEventIO
```

```
::  UnUsed =   UnUsed                               //    Singleton value for unused settings

Start :: *World -> *World
Start world
=   OpenIO { ioDefInit  = [ DoCopyFile ]
           , ioDefAbout = "Copying Process"
           } (UnUsed,UnUsed) world

DoCopyFile :: *(PState .l .p) -> *PState .l .p
DoCopyFile state
=   {state & pFiles = nfiles, pIOState = QuitIO state.pIOState}
where
    nfiles = CopyFile "aap" "noot" state.pFiles

CopyFile :: String String *Files -> *Files
CopyFile source dest files
#   (open,sfile,files) = sfopen source FReadData files
|   not open           = abort "Source file could not be opened.\n"
#   (open,dfile,files) = fopen dest FWriteText files
|   not open           = abort "Destination file could not be opened.\n"
#   (close,files)      = fclose (CopyOneFile sfile dfile) files
|   not close          = abort "Destination file could not be closed.\n"
|   otherwise          = files

CopyOneFile :: File *File -> *File
CopyOneFile sfile dfile
#   (ok,char,sfile)    = sfreadc sfile
|   not ok             = dfile
|   otherwise          = CopyOneFile sfile (fwritec char dfile)
```

10.3 Event based I/O

The *I/O state* (see `StdEventIO`) is an abstract data type which reflects the current state of the event based I/O performed by the program. We already explained that the `IOState` contains the event queue which has been retrieved from the world (see 10.1.2). In this queue all events are being stored that have been generated by the user (by clicking the mouse, pressing keys and buttons etc.) and by the operating system (timer events) while the application is running. Instead of offering low level functions to fetch events from the queue we have chosen to handle all low-level events automatically via CLEAN functions predefined in the CLEAN event I/O library such that a CLEAN programmer only has to deal with the *high-level event handling*.

A CLEAN programmer using the CLEAN I/O system has to define *abstract devices*, an abstraction of the *concrete devices* (such as *Graphical User Interfaces* components) as they can be found on modern computer systems. Examples of abstract devices are: *windows* (including *dialogues*) for window based event I/O (see 10.4), *timers* (for time driven events, see 10.5) and *receivers* (for events generated by using message passing primitives, see 10.7). A device can be composed of *device components* which on their turn can be refined further. For instance, each window can have a *keyboard* (see 10.4.2) and a *mouse* (see 10.4.3) as component and can furthermore have several *controls* (*buttons* and the like, see 10.4.4). A menu is composed out of *sub-menus*, *menu items* and so on.

Specifying abstract devices

Abstract devices are specified in CLEAN by means of predefined *algebraic data types* (see Section 8.2.1) and *type constructor classes* (see Section 8.4). With these types actually a special kind of *declarative device specification language* is offered in which the programmer can define the relevant properties of the concrete devices that are being used on a high level of abstraction. The algebraic specification of a device or device component generally consists of:

- a *constructor* with a meaningful name to indicate the desired device/device component (e.g. **`Menu`** indicates that a menu is wanted);
- the definition of a very limited number of *non-optional attributes* (e.g. each menu must have a `Title` which is of type `String`);

- if applicable the definition of the *sub-components* (e.g. a menu can contain menu-items, a sub-menu etc.) which are defined in the same declarative style;
- a state transition function (called the *call back function*) to be applied on the current process state when the device (component) is triggered by a corresponding event (e.g. one can trigger an abstract menu element by selection of the corresponding concrete menu element with the mouse);
- a list in which one can specify the *optional attributes* (a default value is chosen when an optional parameter is not specified);

The algebraic type instances are glued together to form abstract device (component) definitions. Because the components have different type constructors the glue is provided by type constructor combinators. These have been predefined in the library (see StdIOCommon).

```
::  :+:   t1 t2 c = (:+:) infixr 9 (t1 c) (t2 c)
::  ListNoLS t  c = ListNoLS [t c]
::  NilNoLS     c = NilNoLS
```

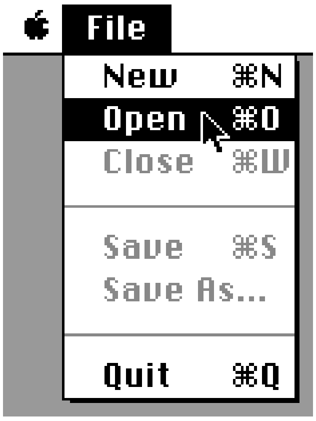
Given two expressions e_1 and e_2 of respective types t_1 and t_2 on some context of type *c*, then $(e_1 \text{ :+: } e_2)$ is an expression of type $(\text{:+: } t_1\ t_2\ c)$. Given an arbitrary number of expressions $e_1 \ldots e_n$ of type *t* on some context of type *c*, then $(ListNoLS\ [e_1 \ldots e_n])$ is an expression of type *(ListNoLS t c)*. Finally, the expression *NilNoLS* fits in every context *c* and has type *(NilNoLS c)*. The following two examples show how algebraic types are used to define and compose abstract device (components).

Example (The predefined algebraic types to define a menu). Constructors are displayed **`bold`**. Note that the variable `m` in `Menu` and `SubMenu` is a type constructor variable. Legal instances of `m` are instances of the class MenuItems (which are `MenuItem` and `MenuSeparator`).

```
::  Menu          m ps = Menu       Title (m ps) [MenuAttribute ps]
::  SubMenu       m ps = SubMenu    Title (m ps) [MenuAttribute ps]
::  MenuItem        ps = MenuItem   Title        [MenuAttribute ps]
::  MenuSeparator ps = MenuSeparator

::  MenuAttribute ps                          // Default:
    =   MenuId           Id                   // no Id
    |   MenuSelectState  SelectState          // menu(item) Able
    |   MenuAltKey       Index                // no AltKey
    //  Attributes ignored by (sub)menus:
    |   MenuShortKey     Char                 // no ShortKey
    |   MenuMarkState    MarkState            // NoMark
    |   MenuFunction     (IOFunction     ps)  // I
    |   MenuModsFunction (ModsIOFunction ps)  // MenuFunction
```

An abstract device specification can be seen as a declarative specification which is interpreted by process control functions like `OpenIO` to generate the demanded action on the screen.

Example (Concrete instantiation of `Menu` and its appearance on the screen of an Apple Macintosh). The call back functions *new*, *open*, *`close`* ,*save*, and *saveAs* need to be declared in the program. As an example we show the call back function of the Quit item. The library function `appPIO` applies an IOState transition function to the IOState component of the process state. Consequently, when "Quit" is chosen from the menu the process will be terminated. This example shows the close relation between the specification and the actual appearance on the screen. Notice that the specification is not static but dynamic: any expression which yields an instance of `Menu` will do.



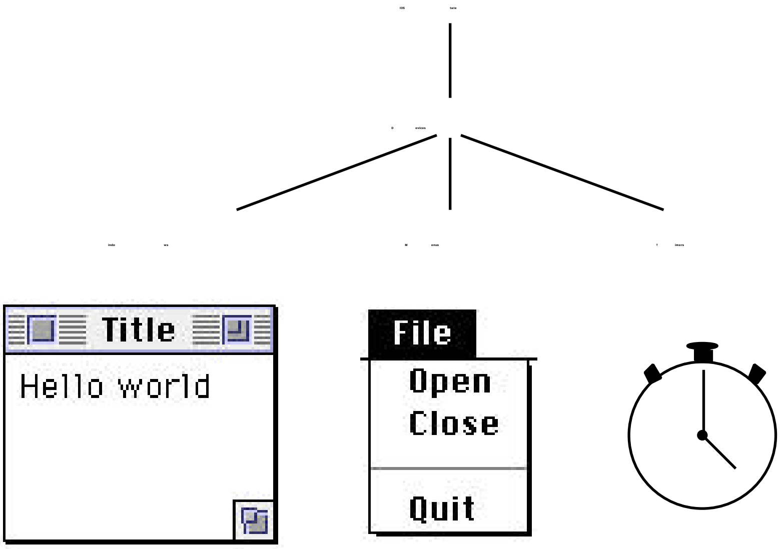
```
Menu "File"
(   ListNoLS
[   MenuItem "New"      [ MenuShortKey 'n', MenuFunction new                              ]
,   MenuItem "Open…"    [ MenuShortKey 'o', MenuFunction open                             ]
,   MenuItem "Close"    [ MenuShortKey 'w', MenuFunction close, MenuSelectState Unable ]
]
:+: MenuSeparator
:+: ListNoLS
[   MenuItem "Save"     [ MenuShortKey 's', MenuFunction save,  MenuSelectState Unable ]
,   MenuItem "Save As…" [ MenuFunction saveAs,                  MenuSelectState Unable ]
]
:+: MenuSeparator
:+: MenuItem "Quit"     [ MenuShortKey 'q', MenuFunction (appPIO QuitIO)                  ]
)   []
```

The specification method is constructed in such a way that not more has to be specified as strictly necessary. Due to the high-level of abstraction the specification can be platform independent. The I/O library can in the future easily be extended by adding more optional attributes and abstract device type constructors without affecting existing programs.

Opening abstract devices and application of call back functions

For each abstract device a special *open function* has been predefined. It is a process or I/O state transition function which stores the algebraic description of the device into the IOState and activates the corresponding *concrete devices* (if they are specified as active) and draws them on the screen (if they have a visual representation).

Figure (what is stored in the IOState).



For instance, with the function `OpenMenu` a menu description like the one given above results in the creation of that menu for the program.

```
class Menus mdef                            // The Menus class to create menus
where
    OpenMenu :: !Int !(mdef (PState .l .p)) !(IOState .l .p) -> IOState .l .p

instance Menus (Menu m) | MenuElements m    // A Menu of MenuElements m is a proper Menus instance
```

The function `OpenIO` will recursively examine the event queue to see if there is an event on top of the queue matching the stored abstract device specifications. If this is the case, the corresponding call back function specified in the algebraic specification is called by applying it to the current process state. A *call back function* is a user-defined *state transition function* defined on the process state, generally of type:

```
CallBackFunc :: Info (PState .l .p) -> PState .l .p
```

The first parameter depends on the kind of call back function. For instance, a call back function invoked by a mouse click will also get information on the current position of the mouse. All call back functions get the actual value of their arguments automatically from the I/O system. When evaluation of a call back function has reached head normal form in every process state component, control (and the process state) is given back such that the next call back function can be determined (with the new process state as returned by the previous call back function) given the next event in the event queue. So, the process states are used to pass information from one call back function to another. A process terminates when the function `QuitIO` is applied on the IO state.

Each call back function can of course change the *process state* but it can also change the definitions or the attributes of the devices and the device components stored in the *I/O state*. Each of them can be modified *dynamically* (that is why they all can have a special label for identification).

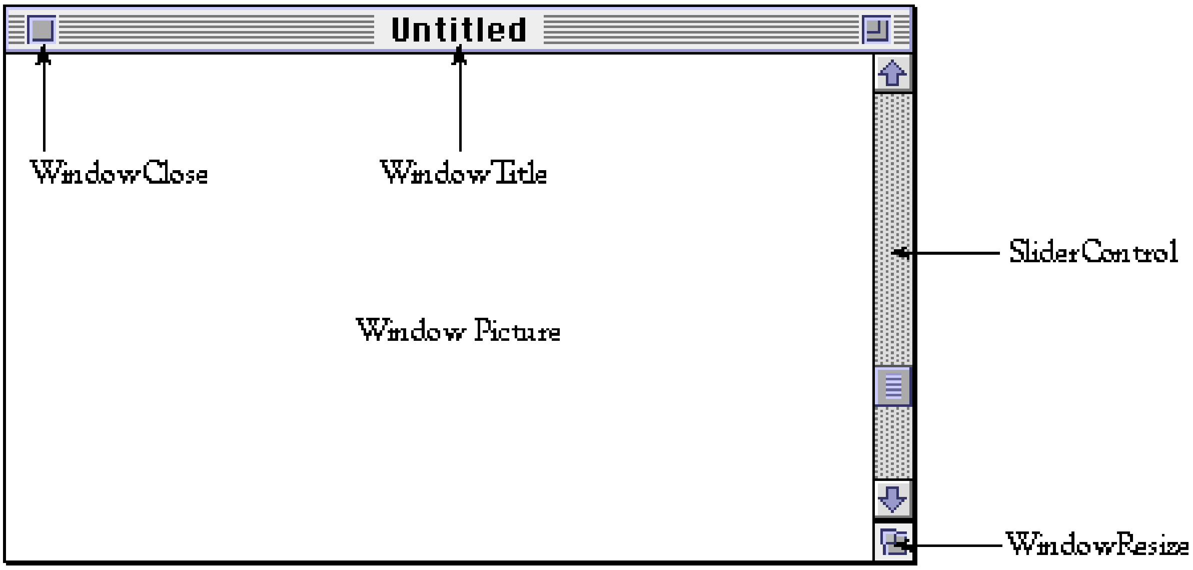
10.4 Graphical user interfaces

In this section we explain how graphical user interfaces like *windows* (including *dialogues*) can be created and manipulated. A *window device* (see `StdWindowDef`) gives a view on a *picture* (again a unique abstract object) on which a set of *drawing and text handling functions* is defined (see 10.4.4). Each window can have a *keyboard* (see 10.4.2) and a *mouse* (see 10.4.3) as component and can furthermore contain several *controls* (*buttons* and the like, see 10.4.5). There is a special lay-out language to control the layout of controls and windows (see 10.4.5).

10.4.1 Windows and dialogues

Windows are the basic medium through which interactive applications and users communicate. An application can have an arbitrary number of open windows. Of these *windows* at most one is *active*. The active window is the window to which all keyboard events are directed. Applications can display anything in a window: a window gives a view on an arbitrary picture. Windows are also used to structure user input to applications: a window may accept keyboard and/or mouse input. Controls (e.g. slider controls) can be put into the window such that very complicated windows can be defined.

Figure (components of a simple window).



Windows can be opened with the function `OpenWindow`. The optional window-id attribute is required to change window settings dynamically (e.g. to close a window, see further `StdWindow` in the appendix).

```
class Windows wdef
where
    OpenWindow      :: !Int !(wdef (PState .l .p)) !(IOState .l .p) -> IOState .l .p
    OpenModalWindow ::      !(wdef (PState .l .p)) !( PState .l .p) ->  PState .l .p

instance Windows (Window       c)    | Controls c
instance Windows (DialogWindow c)    | Controls c

CloseWindow         :: !Id !(IOState .l .p) -> IOState .l .p

::  Window       c ps = Window       Title (c ps) [WindowAttribute ps]
::  DialogWindow c ps = DialogWindow Title (c ps) [WindowAttribute ps]
```

A window is either a basic window (**`Window`**) or a dialogue window (**`DialogWindow`**). Although their definitions can be equal, dialogue windows may adapt their behaviour conforming to the look-and-feel of the current platform. Windows can simply be opened (`OpenWindow`) or modally (`OpenModalWindow`). Modal windows force the user of the program to complete the interaction with that window. Ofcourse, when a modal window is active, new modal windows may be opened for which the same rules apply. Also new none-modal windows may be opened, but they become active only when all modal windows have been closed.

There are a lot of optional window attributes, we only specify a few here (see `StdWindowDef` for a complete list). With the attributes one can attach special call back functions, e.g. to handle mouse clicks in the window, how a window should be positioned (see 10.4.6) how to handle keyboard keys being pressed, what to do when a window is requested to be closed etc. Below we explain keyboard (10.4.2) and mouse handling (10.4.3).

Inside a window controls such as buttons can be positioned (Control Definitions). Controls are treated in section 10.4.6.

10.4.2 Keyboard

A keyboard function is a call back function which can optionally be attached to a window as window attribute.

```
::  WindowAttribute ps                          // Default:
    =   …
    |   WindowKey  SelectState (KeyFunction ps) // no keyboard

::  SelectState    =   Able | Unable
```

In this way one can define a function to handle the response to those keyboard events that are not captured by the window's controls or other interface elements. All keyboard events are directed to the active window (of which there is only one). The `KeyboardState` contains all information needed for the call back function to handle the event: which key was pressed or released and which of the modifier keys were being held down at that occasion.

```
::  KeyFunction ps :== KeyboardState -> ps -> ps

::  KeyboardState
    =   {   keyCode     :: !Char
        ,   keyState    :: !KeyState
        ,   keyModifiers:: !Modifiers
        }
::  KeyState
    =   KeyUp
    |   KeyDown
    |   KeyStillDown
::  Modifiers
    =   {   shiftDown   :: !Bool
        ,   optionDown  :: !Bool
        ,   commandDown :: !Bool
        ,   controlDown :: !Bool
        }
```

10.4.3 Mouse

A `MouseFunction` is a call back function which can optionally be attached to a window as window attribute.

```
::  WindowAttribute ps                              // Default:
    =   …
    |   WindowMouse SelectState (MouseFunction ps)  // no mouse input
```

The `Mouse function` defines the response of the window to mouse events in the window's content (see also controls in 10.3.6 to handle the clicks on buttons and so on). The `MouseState` contains information about the mouse event: the position of the pointer (in picture domain co-ordinates), whether it was a click, a double-click, a triple-click, or the mouse button was still down, or a release of the mouse button and which modifier keys (shift, option etc.) were being held down.

```
::  MouseFunction ps :== MouseState -> ps -> ps

::  MouseState
    =   {   mousePos       :: !Point
        ,   mouseButton    :: !ButtonState
        ,   mouseModifiers :: !Modifiers
        }
::  ButtonState
    =   ButtonUp
    |   ButtonDown
    |   ButtonDoubleDown
    |   ButtonTripleDown
```

```
        |   ButtonStillDown
```

10.4.4 Writing and drawing to a window

To draw in a window the library module `StdWindow` contains the function `DrawInWindow`:

```
DrawInWindow :: Id [DrawFunction] (IOState .l .p) -> IOState .l .p

::  DrawFunction :== *Picture -> *Picture
```

This function takes a list of `DrawFunctions` and applies them to the indicated window in order of appearance in the list. Drawing functions are predefined in the library module `StdPicture`. Like all other interface components Pictures also have optional attributes, and type constructor classes are used to draw (or fill) figures:

```
SetPenPos  :: !Point !*Picture -> *Picture      // Set the current position of the picture's pen
SetPenFont :: !Font  !*Picture -> *Picture      // Set the current font used for drawing text

class Drawables figure
where
    Draw   ::        !figure !*Picture -> *Picture
    DrawAt :: !Point !figure !*Picture -> *Picture

instance Drawables {#Char}
```

Font handling (useful when texts have to be drawn in a window) is performed by functions from the library module `StdFont`.

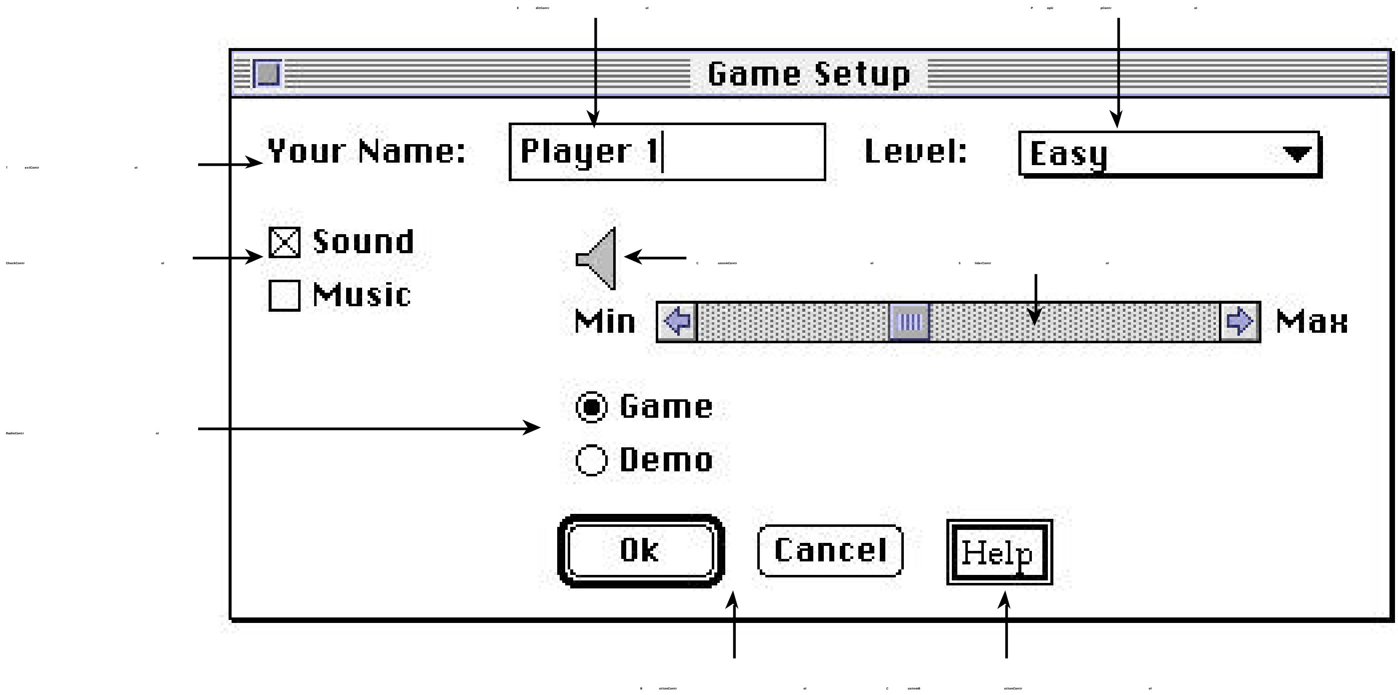
Example (a function that draws a string in a window at a certain position)

```
DrawStringInWindow:: Id Point String (IOState .l .p) -> IOState .l .p
DrawStringInWindow id pos string io = DrawInWindow id [ SetPenPos pos
                                                      , Draw      string
                                                      ] io
```

10.4.5 Controls

Besides that one can draw figures and text in the picture of a window one can also put controls such as buttons into a given window. Controls can be *radio buttons* (a number of options of which only one is valid, all options are displayed), *check boxes* (a number of options which can be turned on and off, all options are displayed), *pop-up menus* (a number of options of which only one is valid and displayed at a time), *slider bars* (a control allowing a user to set a value in a specific range), *text fields* (the text can only be changed by the program during the interaction), *editable text fields* (the displayed text can be changed by the user or the program during the interaction), *buttons* (with or without customised look), and even *user-defined controls* (controls of which the look and feel is specified completely by the programmer, see `StdControls`). Finally, any group of controls can be combined to form a new control using a *CompoundControl*.

Figure (controls in a window).



```
::  RadioControl         ps = RadioControl     TextLine MarkState     [ControlAttribute ps]
::  CheckControl         ps = CheckControl     TextLine MarkState     [ControlAttribute ps]
::  PopUpControl         ps = PopUpControl     [PopUpItem ps] Index   [ControlAttribute ps]
::  SliderControl        ps = SliderControl    Direction Length SliderState
                                               (SliderAction ps)      [ControlAttribute ps]
::  TextControl          ps = TextControl      TextLine               [ControlAttribute ps]
::  EditControl          ps = EditControl      TextLine Width NrLines [ControlAttribute ps]
::  ButtonControl        ps = ButtonControl    TextLine               [ControlAttribute ps]
::  CustomButtonControl  ps = CustomButtonControl Size ControlLook   [ControlAttribute ps]
::  CustomControl        ps = CustomControl       Size ControlLook   [ControlAttribute ps]
::  CompoundControl c    ps = CompoundControl (c ps)                  [ControlAttribute ps]
```

Defining the position of a Control (also applicable for Windows)

```
::  ControlAttribute ps                   // Default:
    …
    |   ControlPos   ItemPos              // (RightTo previous,zero)
```

In the attributes of controls (**`ControlPos`**) or windows (**`WindowPos`**) one can define that they have to be positioned in a certain way. For this purpose a platform independent lay-out language has been defined as follows:

```
::  ItemPos
    :== (   ItemLoc
        ,   ItemOffset
        )
::  ItemLoc
    //  Relative to corner:
    =   LeftTop     |   RightTop     |   LeftBottom |   RightBottom
    //  Relative in next line:
    |   Left        |   Center       |   Right
    //  Relative to other item:
    |   LeftOf Id   |   RightTo Id   |   Above Id   |   Below Id
    //  Relative to previous item:
    |   LeftOfPrev  |   RightToPrev  |   AbovePrev  |   BelowPrev
::  ItemOffset
    :== Vector
```

```
::  Vector
    =   {vx::!Int,vy::!Int}
```

A position consists of a location (`ItemLoc`) and an offset (`ItemOffset`).

- When this location is **`LeftTop`**, **`RightTop`**, **`LeftBottom`** or **`RightBottom`** the item is placed at the indicated corner in the window. When the window is resizable the control will optionally be resized and moved as well.
- When this location is **`Left`**, **`Center`** or **`Right`** the item is placed left-aligned, centred, or right-aligned, *beneath* all items defined earlier in the list.
- When this position is **`LeftOf`**, **`RightTo`**, **`Below`**, or **`Above`** the item is placed relative to the item with the indicated id. When that id is not the id of an item that is defined earlier in the list of items, the item is placed *beneath* the line-oriented items defined earlier, left-aligned. A group of items refering to each other is handled as one new composite item with the layout position of the root item.
- When this position is **`LeftOfPrev`**, **`RightToPrev`**, **`BelowPrev`**, or **`AbovePrev`** the item is placed relative to the previous occuring item in the list of items in the same way as with **`LeftOf`**, **`RightTo`**, **`Below`**, or **`Above`**.
- With the `ItemOffset` an item can be shifted relatively on the indicated position with the (possibly negative) offset specified.

Compound controls, as all controls, can have a layout attribute as well. They are handled as one single layout item. The controls inside a compound control are being layed out locally.

Defining the look of a Control

The look of controls is defined as follows: *system* controls have a predefined look (these are **`Radio-`**, **`Check-`**, **`PopUp-`**, **`Text-`**, **`Edit-`**, **`Button-`**, and **`Slider-Control`**s). *Customised* controls (**`CustomButton-`** and **`Custom-Control`**s) are drawn by their program defined `ControlLook` functions. The `ControlLook` function of a `CompoundControl` is drawn first (if it has one), after which the looks of its control elements in *left-to-right* and *depth-first* order follow. The backgrounds of controls are not erased by the system, thus enabling controls to define local backgrounds.

Defining the size of a Control

When a window is resizeable one would sometimes like to resize the controls in it as well. For this purpose controls can use the **`ControlResize`** attribute.

```
::  ControlAttribute ps                                 // Default:
    |   ControlResize      ControlResizeFunction        // no resize
    |   ControlMinimumSize Size                         // zero

::  ControlResizeFunction
    :== Size ->                                         // current control size
        Size ->                                         // current window size
        Size ->                                         // new window size
        Size                                            // new control size
```

The control resize function determines the new size of a control given the current control size, the current window size, and the new window size. The minimum size of a control can be set with the attribute **`ControlMinimumSize`**. The minimum size of a *compound control* is the minimum surrounding rectangle of its component controls (as a compound is allowed to occupy more space than the total of its component controls).

Computation rules for the new layout are as follows: *Controls* are either *resizeable* or *fixed size*. Resizeable controls are **`Edit-`**, **`Slider-`**, **`Custom-`**, and **`Compound-Controls`**. Fixed size controls are all other controls. If a fixed size control has a **`ControlResize`** attribute then this is ignored. Resizing a window of size *curWSize* to *newWSize* involves the following recomputation of control sizes: fixed size controls do not change in size, as well as resizeable controls without **`ControlResize`** attribute. A resizeable control of size *curCSize* with first ControlResize attribute *f* obtains the new size *f curCSize curWSize newWSize* = *{w,h}*. If the result value is smaller than the minimum size the control will be *hidden*, otherwise its new size will be *{w,h}*. When all sizes have been computed for all controls then the layout will be recalculated. The result of this recalculation is the same as if the window is opened with the new sizes of the controls. This method retains the control layout of a window.

10.5 Timer handling

With the *timer device* (see `StdTimerDef`) a program can be synchronised: a call back function can be evaluated every time a certain time interval has expended. The `TimerInterval` is defined as a number of ticks. The number of ticks per second depends on the operating system. A macro `TicksPerSecond` is available in the library module `StdTimer`. When a time interval is set less than one, a timer event is generated as fast as possible. Several timers can be opened.

```
class Timers tdef
where
    OpenTimer :: !(tdef (PState .l .p)) !(IOState .l .p) -> IOState .l .p

instance OpenTimer Timer

CloseTimer    :: !Id !(IOState .l .p) -> IOState .l .p

::  Timer      ps =   Timer TimerInterval [TimerAttribute ps]
::  TimerInterval :== Int

::  TimerAttribute ps                      // Default:
    =   TimerId       Id                   // no Id
    |   TimerSelect   SelectState          // timer Able
    |   TimerFunction (TimerFunction ps)   // \_ x->x
::  TimerFunction  ps :== NrOfIntervals->ps->ps
::  NrOfIntervals     :== Int
```

The `TimerFunction` is the type of the call back function which is called each time the specified timer interval has passed. The `NrOfIntervals` parameter contains the number of times the interval has passed since the last time the timer function was called. This parameter is required because the program might be busy with the evaluation of some call back function. Each call back function is an indivisible action which turns over control to the process control function when the head normal form is reached on each of the components of the process state. Such an evaluation might of course take more time than one timer interval. In that case the `NrOfIntervals` will be greater than one. It is guaranteed however, that each timer gets its turn some time, provided that no non-terminating event handlers have been defined. When the `TimerInterval` of a timer is less than one, the timer function of this timer will be called as often as possible. The `NrOfIntervals` argument of the timer function will then always be one.

10.6 Incorporation of local state in abstract devices

In the previous sections we have shown how abstract device components can be defined by algebraic types and glued by type constructor combinators. The resulting definitions define components that operate on the same process state context. To enhance modular programming the I/O library allows each component to have its private *local state* of arbitrary type. Again type constructor combinators are used to glue components that operate on the same or different local state, and to encapsulate local state from external access. In this section we first show how algebraic types are used to define components with and without local state, and second we show which type constructor combinators are used to glue these components.

As an example of how local state is incorporated in abstract device component definitions we reconsider the case of menus (Section 10.3). The first four definitions below define menu (components) without local state. For each such type constructor with name `T` we introduce a new type constructor by appending '`LS`' after `T`. So, for the type constructor `Menu` we obtain `MenuLS`. The same is done for the algebraic data constructors. For each new type constructor we increase its arity by inserting the type variable `ls` before the last type variable `ps`. This new variable will correspond with the type of the local state of the component. So, the type constructor `MenuLS m ps` becomes `MenuLS m ls ps`. The types of the call back functions (which are usually collected in the attribute list) change from `ps → ps` to `(ls,ps) → (ls,ps)`. Finally, if a type constructor has a type constructor variable then it is assumed that this variable has the same arity. So the type constructor variable `m` in the types `MenuLS` and `SubMenuLS` is parameterised with `ls` and `ps`.

```
//  Menu(item) definitions without local state:
::  Menu          m    ps = Menu       Title (m    ps) [MenuAttribute      ps ]
::  SubMenu       m    ps = SubMenu    Title (m    ps) [MenuAttribute      ps ]
::  MenuItem           ps = MenuItem   Title           [MenuAttribute      ps ]
::  MenuSeparator      ps = MenuSeparator

//  Menu(item) definitions with local state:
::  MenuLS        m ls ps = MenuLS ls  Title (m ls ps) [MenuAttribute *(ls,ps)]
::  SubMenuLS     m ls ps = SubMenuLS  Title (m ls ps) [MenuAttribute *(ls,ps)]
::  MenuItemLS      ls ps = MenuItemLS Title           [MenuAttribute *(ls,ps)]
::  MenuSeparatorLS ls ps = MenuSeparatorLS
```

To glue type constructors with local state we need to extend the set of type constructor combinators that we introduced in Section 10.3 in a similar way. For each type constructor combinator we introduce a new type constructor combinator with increased arity to represent the local state type. The new type constructor combinators are *:~:*, *ListLS*, and *NilLS* for *:+:*, *ListNoLS*, and *NilNoLS* respectively. With these combinators we can compose arbitrary abstract device component definitions that have local state.

```
::  :+:   t1 t2   c = (:+:) infixr 9 (t1   c) (t2   c)
::  ListNoLS t    c = ListNoLS       [t    c]
::  NilNoLS       c = NilNoLS

::  :~:   t1 t2 l c = (:~:) infixr 9 (t1 l c) (t2 l c)
::  ListLS   t  l c = ListLS         [t  l c]
::  NilLS       l c = NilLS
```

With these combinators we can compose arbitrary abstract device component definitions that either have or do not have local state. To be able to glue components with or without local state, or different types of local state the following type constructor combinators complete the set of combinators.

```
::  LS       t  l c = LS    (t c)
::  NoLS     t    c = E.l : {introLS :: .l,  introDef :: t .l       c}
::  ExtendLS t  l c = E.l1: {extendLS:: .l1, extendDef:: t *(.l1,l) c}
::  ChangeLS t  l c = E.l1: {changeLS:: .l1, changeDef:: t .l1      c}
```

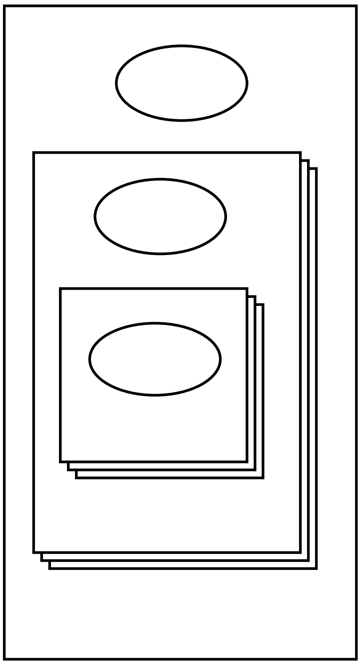
Given an expression *e* of type *t* on some context of type *c*, then *(LS e)* is an expression of type *(LS t l c)* for an arbitrary local state of type *l* and same context of type *c*. Given an expression *e* of type *t* on some local state of type *l* and context of type *c* and a value *v* of type *l*, then the expression *{introLS=v, introDef=e}* encapsulates the local state and is of type *(NoLS t c)*. Given an expression *e* of type *t* on a local state pair of type *(l1,l)* and context of type *c* and a value *v* of type *l1*, then the expression *{extendLS=v, extendDef=e}* encapsulates the first local state component and is of type *(ExtendLS t l c)*. Given an expression *e* of type *t* on a local state of type *l1* and context of type *c* and a value of type *l1*, then the expression *{changeLS=v, changeDef=e}* switches from local state and is of type *(ChangeLS t l c)*.

10.7 Interleaved executing communicating processes

Imagine that one has written two interactive Clean applications, an editor and a compiler for a programming language. Each of these applications will have its own devices (windows, dialogues, menus, timers) and own program state to remember application specific information. Assume that one wants to combine both interactive applications into a new one, for instance to make a programming environment for that language. The Clean I/O system makes this possible and, when applications are structured in the right way, one can even reuse the original source code without any modification.

A Clean program can consist of several interactive processes which can be created dynamically. Each process defines its own user interface, timers and so on with corresponding call back functions. One can switch between these sub-applications and exchange information between them (like in a multi-finder). Again the Clean I/O system will take care of all low-level event handling, activation/de-activation of windows, the switching between menu-bars depending on which application is active and so on. Each call back function remains an indivisible action which will turn over the control to the process control function when the head normal form is reached on each of the components of the process state. So, on one processor the interactive processes will run *interleaved* with each other. The I/O system will call one call back function after another, depending on which application is active and which event is raised.

Figure (process groups and processes and how they can communicate via the process state and file system).



Each interactive application can store its private information in its *local process state*. Several processes can form a *process group*. There can be several groups. Processes in the same group can exchange information via their *shared process state* (see 10.1.2). Since call back functions are indivisible it is guaranteed that only one process at a time can have access to a shared process state. All applications (whether they are in the same group or not) can communicate via files. Since files are uniquely attributed it means that a particular file can only be opened for writing by one (sub-) application at the time. It is good to realise that CLEAN applications can also communicate with other (non-CLEAN) applications running on the computer system in the same way. This means that CLEAN applications can be smoothly incorporated in the real world.

```
::  InitIO l p :== [.IOFunction (PState l p)]

OpenIO     :: !(IODef .l .p) (.l, .p) !*World            -> *World
NewIO      :: !(IODef .l .p) (.l, .p) !(IOState .l` .p`) -> IOState .l` .p`
NewSubIO   :: !(IODef .l .p) (.l, .p) !(IOState .l` .p`) -> IOState .l` .p`
ShareIO    :: !(IODef .l .p) .l       !(IOState .l` .p ) -> IOState .l` .p
ShareSubIO :: !(IODef .l .p) .l       !(IOState .l` .p ) -> IOState .l` .p
```

Initially an interactive CLEAN program has only one group with one process (created with `OpenIO`). Any process can dynamically create new processes using the other process control functions shown above. A *new* interactive process in the *same* process group can be created by applying `ShareIO` on the `IOState`. The process control function `ShareSubIO` also adds a new interactive process to the same process group, but allows the graphical user interface to be *shared*. This means that to the user it seems as if the windows and menus of the newly created process merge with the windows and menus of the parent process. With the function `NewIO` any process can create a new group initially consisting of one interactive process. The function `NewSubIO` does the same, but analogous to `ShareSubIO` also allows the graphical user interface elements to be shared with the windows and menus of the parent process.
]

10.7.1 One-way message passing

We have seen that processes in the same group can communicate via the shared process state and that all processes can communicate via files obtained from the files system. Interactive CLEAN processes can also communicate with each other via message passing.

In the CLEAN system messages are considered to be *abstract events*. Conform the event I/O paradigm of abstract event handling by abstract devices (see 10.3), message events are dealt with by a new abstract device, the *receiver device*. There is no restriction on the type of messages, every typeable expression can be subject to message passing. The type system is applied to enforce type-safe message passing: it is impossible for a correctly typed interactive program to send messages of the wrong type.

```
::  RId mess

class RIds environment
where
    OpenRId :: !*env -> (!(!RId mess,!*RId mess),!*env)   // Create a one-way receiver id

instance RIds World
instance RIds (IOState .l .p)
```

Crucial in achieving type-safe message passing are the special abstract receiver identification values of type `RId`. Receiver ids can be generated from a unique World environment and a unique IOState environment. To create a receiver that accepts messages of some type α one needs to create a *unique* (`RId` α) first. The library function `RIdtoId` coerces a `RId` to `Id`.

```
class Receivers rdef
where
    OpenReceiver   :: !*(RId m) !(rdef m (PState .l .p)) !(IOState .l .p) -> IOState .l .p
    ReopenReceiver :: ! (RId m) !(rdef m (PState .l .p)) !(IOState .l .p) -> IOState .l .p

instance Receivers Receiver

CloseReceiver      :: !Id !(IOState .l .p) -> IOState .l .p

::  Receiver      m ps = Receiver      (ReceiverFunction m      ps )
                                       [ReceiverAttribute       ps ]
::  ReceiverLS ls m ps = ReceiverLS ls (ReceiverFunction m *(ls,ps))
                                       [ReceiverAttribute  *(ls,ps)]

::  ReceiverFunction m ps :== m -> ps -> ps
::  ReceiverAttribute  ps            // Default:
    =   ReceiverSelect SelectState   // receiver Able
```

Interactive processes can dynamically open and close an arbitrary number of receivers with the functions `OpenReceiver` and `CloseReceiver`. When a receiver is created the type of message it can receive is fixed. Sending a message actually means that an event of appropriate type is raised and put into the event queue in the IOState. Due to the message type parameter of `RId` it is guaranteed that only correctly typed messages can be send and that they can only go to the correct receiver.

```
::  SendReport
    =   SendOk
    |   SendUnknownProcess
    |   SendUnknownReceiver
    |   SendUnableReceiver
    |   SendDeadlock

::  SendCont ps :== SendReport -> ps -> ps

ASyncSend :: !(RId mess) mess !(Optional (SendCont (PState .l .p))) !(PState .l .p)
          -> PState .l .p
SyncSend  :: !(RId mess) mess !(Optional (SendCont (PState .l .p))) !(PState .l .p)
          -> PState .l .p
```

Interactive processes can send messages in an *asynchronous* or *synchronous* way (with `ASyncSend` and `SyncSend` respectively). Both functions require the receivers identification value of type `RId m`. Neither function has an effect in case the corresponding receiver does not exist anymore (because the receiver has been closed or the receiving process has been terminated). `ASyncSend` is purely asynchronous. `SyncSend` *blocks* the sending interactive process until the indicated receiver accepts the message. Programmers must be aware that `SyncSend` involves a *context-switch*. In case that there are several interactive processes in the process group it can be possible that after a `SyncSend` the shared process state component may be changed by another process in the group. In case the system detects a potential deadlock situation, message passing is halted and the `SendDeadlock` `SendReport` alternative is applied to the optional `SendCont`inuation.

10.7.2 Two-way message passing

In addition to one-way message passing the Clean I/O library also provides synchronous two-way message passing. This is done analogous to one-way message passing, but now using receivers that accept a message of some type α and respond with a message of some type β. Sending a message is similar to one-way synchronous message passing, except that a response value of type β is returned in the `Send2Cont`inuation function.

```
::  R2Id mess resp

class RIds environment
where
    OpenR2Id :: !*env -> (!(!R2Id m r,!*R2Id m r),!*env)       // Create a two-way receiver id

::  Receiver2      m r ps
=   Receiver2      (Receiver2Function m r      ps)  [ReceiverAttribute      ps]
::  Receiver2LS ls m r ps
=   Receiver2LS ls (Receiver2Function m r *(ls,ps)) [ReceiverAttribute *(ls,ps)]

::  Receiver2Function m r ps :== m -> ps -> (r,ps)

class Receiver2s rdef
where
    OpenReceiver2   :: !*(R2Id m r) !(rdef m r (PState .l .p))
                                    !(IOState .l .p) -> IOState .l .p
    ReopenReceiver2 :: ! (R2Id m r) !(rdef m r (PState .l .p))
                                    !(IOState .l .p) -> IOState .l .p

instance Receiver2s Receiver2

::  Send2Cont r ps :== (SendReport,Optional r) -> ps -> ps

SyncSend2 :: !(R2Id mess resp) mess !(Optional (Send2Cont resp (PState .l .p)))
             !(PState .l .p) -> PState .l .p
```

11

Defining macros

In this chapter macros are treated. Macros are rewrite rules which are applied at compile time which can be used to define constants, create in-line substitutions, rename functions etc.

11.1 Defining Macros

At compile time the right-hand side of the *macro definition* will be substituted for every occurrence of the left-hand side. The substitution process is guaranteed to terminate. With a macro definition one can, for instance, assign a name to a constant such that it can be used as pattern on the left-hand side. Furthermore, the use of macros can speed up a CLEAN program since due to the inline substitution less function calls need to be done. A disadvantage is that more code will be generated when (parameterised) macros are used instead of non-recursive functions.

MacroDef = [MacroFixityDef] DefOfMacro
MacroFixityDef = **(***FunctionSymb***)** [Fix][Prec] ;
DefOfMacro = FunctionSymbol {*Variable*} **:==** FunctionBody ;
[LocalFunctionAltDefs]

The formal arguments of a macro are not allowed to contain constants: only variables are allowed as formal argument. A macro rule always consists of a single alternative.

- Macro definitions are not allowed to be cyclic to ensure that the substitution process terminates.

Example (macros):

```
Black     :== 1
White     :== 0

::Color:== Int

Invert :: Color -> Color
Invert Black = White
Invert White = Black
```

Example (example: macro to write `(a?b)` for lists instead of `[a:b]` and its use in the function `map`).

```
(?) infixr 5
(?) h t :== [h:t]

map :: (a -> b) [a] -> [b]
map f (x?xs)   = f x ? map f xs
map f []       = []
```

Macros can contain local function definitions. These definitions will also be substituted inline. In this way complicated substitutions can be achieved resulting in efficient code.

Example (example: macros can be used to speed up frequently used functions. See for instance the definition of the function `foldl` in `StdList`).

```
foldl op r l :== foldl r l
where
     foldl r []        = r
     foldl r [a:x]  = foldl (op r a) x

sum list = foldl (+) 0 list
```

After substitution of the macro `foldl` a very efficient function `sum` will be generated by the compiler:

```
sum list = foldl 0 list
where
     foldl r []        = r
     foldl r [a:x]  = foldl ((+) r a) x
```

The expansion of the macros takes place before type checking. Type specifications of macro rules is not possible. When operators are defined as macros, fixity and associativity can be defined.

12

Modules

A CLEAN program is composed of modules to enable separate compilation and to provide a facility to hide actual implementations of types and functions.

12.1 Definition and implementation modules

The CLEAN module system is inspired by the module system of Modula-2 (Wirth, 1982). Like in Modula2, a CLEAN program consists of a collection of *definition modules* and *implementation modules*. An implementation module and a definition module *correspond* to each other if the names of the two modules are the same. The basic idea is that the definitions given in an implementation module only have a meaning in the module in which they are defined (see Section 3.5) unless these definitions are exported by putting them into the corresponding definition module (see section 4.4). In that case they also have a meaning in those other modules in which the definitions are imported (see Section 4.3).

```
CLEANProgram          =  {Module}+
Module                =  DefinitionModule
                      |  ImplementationModule
DefinitionModule      =  definition module ModuleSymb ;
                         {Definition}
                      |  system module ModuleSymb ;
                         {Definition}
ImplementationModule  =  [implementation] module ModuleSymb ;
                         {Definition}
```

- Each CLEAN module has to be put in a separate file.
- The name of a module (i.e. the module name) should be the same as the name of the file (minus the suffix) in which the module is stored.
- A *definition* module should have as *.dcl* as suffix, an *implementation* module should have as `.icl` as suffix.
- A definition module can have at most one corresponding implementation module.
- Every implementation module (except the main module, see 11.1.2) must have a corresponding definition module.

12.1.1 Separate compilation

So, if you want to export a definition, you simply repeat the definition in the corresponding definition module. For some kind of definitions in the implementation module it is only allowed to repeat a certain part of it in the definition module (generally the type). The idea is to hide the actual implementation from the outside world. The is good for software engineering reasons while another advantage is that an implementation module can be recompiled separately without a need to recompile other modules. Recompilation of other modules is only necessary when a definition module is changed. All modules depending on the changed module have to be recompiled as well. Implementations of functions, graphs and class instances are therefore only allowed in *implementation* modules. They are exported by only specifying their type definition in the definition module. Also the right-hand side of any type definition can remain hidden. In this way an abstract data type is created (see 8.2.4).

```
Definition          =   ImportDef
                    |   TypeDef
                    |   ClassDef
                    |   FunctionDef
                    |   GraphDef
                    |   MacroDef
```

Example (definition module):

```
definition module ListOperations

::complex                               //  abstract type definition

re :: complex -> Real                   //  type of function taking the real part of a complex number
im :: complex -> Real                   //  type of function taking the imaginary part of a complex
mkcomplex :: Real Real -> Complex       //  type of function making a complex number
```

Example (corresponding implementation module):

```
implementation module ListOperations

::complex :== (!Real,!Real)                     //  type synonym

re :: complex -> Real                           //  type of function followed by its implementation
re (frst,_) = frst

im :: complex -> Real
im (_,scnd) = scnd

mkcomplex :: Real Real -> Complex
mkcomplex frst scnd = (frst,scnd)
```

12.1.2 Special kind of modules

The main or start module

The *main* or *start module* is the top-most module (*root module*) of a CLEAN program.

- Only in the *main* module one can leave out the keyword `implementation` in the module header. In that case the implementation module does not need to have a corresponding definition module.

Evaluation of a CLEAN *program* consists of the evaluation of the application defined in the right-hand side of the **`Start`** *rule* to normal form. The right-hand side of the `Start` rule is regarded to be the *initial expression* to be computed. The definition of the left-hand side consists of the *symbol* **`Start`** with one optional argument (of type `*World`), which is the environment parameter that is necessary to perform I/O actions (see Chapter 10). One can of course add a Start rule to any module. This can be handy for testing functions defined in such a module: to evaluate such a Start rule simply generate an application with the module as root and execute it.

- In the main module a `Start` rule has to be defined.

Example (a very tiny CLEAN program):

```
module hello

Start = "Hello World!"
```

System definition and implementation modules

System modules are special modules. A *system definition module* indicates that the corresponding implementation module is a *system implementation module* which does not contain ordinary CLEAN rules. In system implementation modules it is allowed to define *foreign functions*: the bodies of these foreign functions are written in another language than CLEAN. System implementation modules make it possible to create interfaces to operating systems, to file systems or to increase execution speed of heavily used functions or complex data structures. Typically, predefined function and operators for arithmetic and File I/O are implemented as system modules.

System implementation modules may use machine code, C-code, abstract machine code (PABC-code) or code written in any other language. What exact is allowed is dependent from the CLEAN compiler used and the platform for which code is generated. The keyword *code* is reserved to make it possible to write CLEAN programs in a foreign language. This is not treated in this reference manual.

When one writes system implementation modules one has to be very careful because the correctness of the functions can no longer be checked by the CLEAN compiler. Therefore, the programmer is now responsible for the following:

! The function must be correctly typed.

! When a function destructively updates one of its (sub-)arguments, the corresponding type of the arguments should have the uniqueness type attribute. Furthermore, those arguments must be strict.

12.2 Importing definitions

Via an *import statement* a definition *exported* by a definition module (see 11.3) can be *imported* into a (definition or implementation) module. A *symbol* is said to be *defined* in a module if it either is *implicitly defined* (i.e. *imported* from another module) or when it is *explicitly defined* (i.e. in a definition in the module itself).

```
ImportDef        = import {ModuleSymb}-list ;
                 | from ModuleSymb import {ImportSymbols}-list ;
ImportSymbols    = FunctionSymb
                 | ConstructorSymb
                 | SelectorVariable
                 | FieldSymb
                 | MacroSymb
                 | TypeSymb
                 | ClassSymb
```

There are two kind of import statements, *explicit* imports and *implicit* imports.

Explicit imports are import statements in which the definitions to import are explicitly specified. The symbol names uniquely identifying the definitions to import are listed together with the name of the module to import them from.

Explicit imports can be used to avoid unintended name clashes that can occur via implicit imports.

Implicit imports are import statements in which only the module name to import from is mentioned. In this case *all* symbols that are *exported* from that module are imported and also *all* symbols that on their turn are *imported* in the mentioned definition module, and so on. So, all related definitions from various modules can be imported with one single import. This opens the possibility for definition modules to serve as a kind of *'pass-through'* module (see for instance the definition module `StdEnv` specified below). With such a module one can import a complete environment with one simple statement. Hence, it is meaningful to have definition modules with import statements but without any definitions and without a corresponding implementation module. With an implicit import only those symbols are imported which are not already explicitly defined in the importing module.

Example (implicit import): all (arithmetic) rules which are predefined can be imported easily with one import statement:

```
import StdEnv
```

importing implicitly all definitions imported by the definition module 'StdEnv' which is defined below (note that definition module `'StdEnv'` does not have a corresponding implementation module) :

```
definition module StdEnv

import
    StdBool, StdChar, StdInt, StdReal, StdString
```

- If a symbol is explicitly defined it cannot be imported from another module as well.
- A symbol can be imported more than once only if those imports refer to the same definition.

A module *depends on* another module if it imports a symbol from that other module.

- Cyclic dependencies of definition modules are prohibited, i.e. if a definition module M_1 depends on another definition module M_2 then M_2 is not allowed to depend on M_1.

12.3 Exporting definitions

The definitions given in an implementation module only have a meaning in the module in which they are defined (see Section 3.5) unless these definitions are exported by putting them into the corresponding definition module. In that case they also have a meaning in those other modules in which the definitions are imported (see Section 12.2).

- The definitions given in a definition module have to be repeated in the corresponding implementation module. In the implementation module all definitions have to get an appropriate implementation as well (this holds for functions, abstract data types, class instances).
- An *abstract type definition* is exported by specifying the left-hand side of a type rule in the definition module. In the corresponding implementation module the abstract type *has to be defined again* but then right-hand side has to be defined as well. It can be either an algebraic type, record type or synonym type definition. For such an abstract data type only the name of the type is exported but not its definition.
- A *function*, global *graph* or *class instance* is exported by repeating the type header in the definition module. For optimal efficiency it is recommended also to specify strictness annotations (see 8.5). For library functions it is recommended also to specify the uniqueness type attributes (see Chapter 9). The implementation of the function, graph, class instance has to be given in the implementation module.

13

Time and space efficiency

Programming in a functional language means that one should focus on algorithms and without worrying about all kinds of efficiency details. However, when large applications are being written it may happen that this attitude results in a program which is unacceptably inefficient in time and/or space.

There are several ways in which a CLEAN programmer can improve the time/space behaviour of a program. In this chapter we give some suggestions how this can be done. No new language constructs are introduced here. We just give some additional information about the space and time behaviour of the data structures and language constructs introduced in the previous chapters. Most of this information is highly implementation dependent. So, in reality it might be slightly different. Yet we think that the information given in this chapter might be of practical use for writing big applications.

13.1 Space consumption of CLEAN structures

In this section we give a rough indication of the space occupied by CLEAN data structures. In this context it is important to know that in general the occupied space will depend on whether these data structures appear in a strict or appear in a lazy context (see Section 8.5). Data structures in a *lazy context* are passed via references on the *A*-stack which point to nodes stored in the heap (see Plasmeijer and Van Eekelen, 1993). Data structures of the *basic types* (`Int`, `Real`, `Char` or `Bool`) in a *strict context* are stored on the *B*-stack or in registers. This is also the case for these strict basic types when they are part of a *record* or *tuple* in a strict context. Data structures living on the B-stack are passed *unboxed*. They consume less space (because they are not part of a node) and can be treated much more efficiently. When a function is called in a lazy context its data structures are passed in a node (*boxed*). The amount of space occupied is now also depending on the arity of the function.

In the table below the amount of space consumed in the different situations is summarised (for the lazy as well as for the strict context). For the size of the elements one can take the size consumed in a strict context.

Type	*Arity*	*Lazy context (bytes)*	*Strict context (bytes)*	*Comment*
`Int, Bool`	-	8	4	
`Int (0≤n≤32), Char`	-	-	4	node is shared
`Real`	-	12	8	
`Small Record`	n	4 + Σ size elements	Σ size elements	total length≤12
`Large Record`	n	8 + Σ size elements	Σ size elements	

`Tuple`	`2`	`12`	`Σ size elements`	
	`>2`	`8 + 4*n`	`Σ size elements`	
`{a}`	`n`	`20 + 4*n`	`12 + 4*n`	
`!Int`	`n`	`20 + 4*n`	`12 + 4*n`	
`!Bool,!Char`	`n`	`20 + 4*ciel(n/4)`	`12 + 4*ciel(n/4)`	
`!Real`	`n`	`20 + 8*n`	`12 + 8*n`	
`!Tuple, !Record`	`n`	`20 + size rec/tup*n`	`12 + size rec/tup*n`	
`Hnf`	`0`	`-`	`4+size node`	node is shared
	`1`	`8`	`4+size node`	
	`2`	`12`	`4+size node`	also for `[a]`
	`>2`	`8 + 4*n`	`4+size node`	
`Pointer to node`	`-`	`4`	`4`	
`Function`	`0,1,2`	`12`		
	`>3`	`4 + 4*n`		

13.2 Size limitations

There are some implementation dependent restrictions which play a role when large programs are being written. Here they come:

- the arity of functions and constructors has to be ≤ 32. This also holds for the number of elements in predefined data structures like a tuple or record. There is however no restriction on the number of elements in an array (besides restrictions imposed on the amount of memory on the machine).
- the number of files which can be open at the same time has to be ≤ 16.
- the code size of an implementation module has to be ≤ 32K (Macintosh only caused by limitations of the linker on the Mac).

13.3 Lazy evaluation versus strict evaluation

As one can deduce from the table above strict data structures generally consume less space than lazy data structures. Furthermore, unboxed elements can be put on a stack or kept in registers which also has a positive influence on the evaluation speed. In general one can say that strictness gives a much better space and time behaviour of the program. However, CLEAN is by default a *lazy* functional language because laziness gives notational advantages.

Lazy evaluation has the following advantages (+) / disadvantages (-) compared with eager (strict) evaluation:

+ an expression is only evaluated when its value is needed to produce the result (normal form) of the Start expression ;
+ one can work with infinite data structures (e.g. `[1..]`);
+ only those computations which contribute to the final result are computed (for some algorithms this is a clear advantage while it generally gives a greater expressive freedom);
- it is unknown when a lazy expression will be computed (disadvantage for debugging, for controlling evaluation order);
- strict evaluation is in general much more efficient, in particular for objects of basic types, non-recursive types and tuples and records which are composed of such types;

-/+ in general a strict expression (e.g. 2 + 3 + 4) takes less space than a lazy one, however, sometimes the other way around (e.g. [1..1000]);

Example (functions with strict arguments of basic type are more efficient).

```
Ackerman :: Int Int -> Int
Ackerman 0 j = inc j
Ackerman i 0 = Ackerman (dec i) 1
Ackerman i j = Ackerman (dec i) (Ackerman i (dec j))
```

> The computation of `Ackerman 3 7` takes 14.8 seconds + 0.1 seconds for garbage collection on an old fashion MacII (5Mb heap). When both arguments are annotated as strict it will take 1.5 seconds + 0.0 seconds garbage collection. The gain is one order of magnitude. Instead of rewriting graphs the calculation is performed using stacks and registers where possible. The speed is comparable with a recursive call in highly optimised C or with the speed obtainable when the function was programmed directly in assembly.

So, lazy evaluation gives a notational freedom but it can cost space and time. In CLEAN the default lazy evaluation can therefore be turned into eager evaluation in several ways:

+ One can define (partially) strict data structures (see 8.5.3). Whenever such a data structure occurs in a strict context (see 8.5.1), its strict components will be evaluated. *Warning: infinite data structures thus defined will cause non-termination when put into a strict context.*
+ The CLEAN compiler has a built-in strictness analyser based on *abstract reduction* (Nöcker, 1993) (it can be optionally turned off). The analyser searches for strict arguments of a function and annotate them as strict (see 8.5.1). In this way lazy arguments are automatically turned into strict ones. This optimisation does not influence the termination behaviour of the program. It appears that the analyser can find much information. The analysis itself is quite fast.
+ The strictness analyser cannot find all strict arguments. Therefore one can also manually annotate a function as being strict in a certain argument or in its result (see 8.5.1). *Warning: when the corresponding expression is non-terminating the annotation will invoke a non-terminating evaluation when such a function is being evaluated.*
+ The order of evaluation in a function body can be influenced with a strict let expression (see 6.4). Again this may lead to non-termination.

13.4 Destructive updates using uniqueness typing

In principle it is possible to update a uniquely typed function argument (`*`) destructively when the argument does not reappear in the function result (see Chapter 9). Performing destructive updates is only sensible when information is stored in nodes (and hence not for elements of basic type (`Int`, `Real`, `Char` or `Bool`) in a strict context because they are stored on the B-stack or in registers).

Destructive updates of important predefined data structures such arrays, records and files of course can have a big influence on the space and time behaviour (a new node does not have to be claimed and filled, the garbage collector is invoked less often and the locality of memory references is increased) of programs. So, applications written using these data structures uniquely can run much more efficient in less memory.

In principle it is possible that user-defined unique data structures are also destructively updated by the CLEAN system : the space being occupied by a function argument of unique type can be reused destructively to construct the function result when (part of) this result is of the same type. So, a more space and time efficient program can be obtained by turning heavily used data structures into unique data structures. This is not just a matter of changing the uniqueness type attributes (like turning a lazy data structure into a strict one). A unique data structure also has to be used in a “single threaded” way (see Chapter 9). This means that one might have to restructure parts of the program to maintain the unicity of objects.

The compiler will do compile-time garbage collection for user defined unique data-structures only in certain cases. In that case run-time garbage collection time is reduced. It might even drop to zero. It also possible that you gain even more then just garbage collection time due to better cache behaviour.

Bad news: this feasture is under test and switched of in the current release. You have to wait till the next version of the CLEAN system.

13.5 Graphs versus constant functions versus macros

With a *macro* definition constants and simple functions can be textually substituted at compile time (see Chapter 10). This saves a function call and makes basic blocks larger (see Plasmeijer and Van Eekelen, 1993) such that *better* code can be generated. A disadvantage is that also *more* code will be generated. Inline substitution is also one of the regular optimisations performed by a compiler. To avoid code explosion a compiler will generally not substitute big functions. Macros give the programmer a possibility to control the substitution process to get an optimal trade-off between the efficiency of code and the size of the code.

The difference between a *graph* and a *constant function* is that multiple references to a graph will result in sharing of that graph (see Chapter 5) while multiple reference to a (constant) function will result in equally many function calls (see Chapter 6). Graphs have the property that they *are computed only once* (call by need) and that their value is remembered within the scope they are defined in. A constant function is evaluated each time it is applied. A graph saves execution-time at the cost of space consumption. A constant function saves space at the cost of execution time. So, use graphs when the computation is time-consuming while the space consumption is small and constant functions in the other case.

13.6 The costs of overloading

In Section 8.4 the overloading mechanism of CLEAN is treated. The use of overloading and type classes certainly gives a lot of notational convenience. However, one should be aware of the time and space costs that might be caused by using overloading and type classes.

When an overloaded function is used in such a way that the system can replace the overloaded function by the concrete one, no overhead is introduced (see Section 8.4).

Overloading can cause code explosion. When in a certain function another overloaded function is applied in such a way that the type system *cannot* deduce which concrete instance of the overloaded function has to be used the system will in principle generate *several versions* of the function: *one* version is made for *each* of the concrete (combination of) instances possible. In principle special versions will only be generated for instantiations of basic types. Although the system avoids to generate versions that are not being used, code explosion might occur when all versions are being used or when the system simply cannot tell which versions are used. The latter can be the case when such functions are being exported to other modules.

Overloading can cause inefficiency. Instances which are recursively defined in terms of the class itself can lead to an infinite amount of concrete instances. New instances can also be declared in modules that import the overloaded function. To handle all these cases the system will generate one special version of the overloaded function which is parametrised with a type class record (see the introduction of 8.4). In such cases overloading is implemented by using records as a dictionary in which the concrete function is looked up. This means that the record is used to store higher order functions. Calling such a higher function in this way is much more inefficient than a direct call of the corresponding concrete function. One can avoid unnecessary efficiency loss as follows. When an overloaded function is exported it is advised also to export the concrete instances of the overloaded functions. The concrete names of the functions need not to be exported. The system needs only to know which concrete instances already exist.

13.7 Concurrency

The process annotations of CLEAN are designed to make parallel evaluation on loosely coupled parallel machine architectures possible. A *loosely coupled parallel architecture* is defined as a multi-processor system which consists of a number of self-contained computers, i.e. sparsely connected processors each with private memory. An important property of such systems is that for each processor it is more efficient to access objects located in its own local memory than to use the communication medium to access remote objects. In order to achieve an efficient implementation it is necessary to map the computation graph to the physical processing elements in such a way that the communication overhead due to the exchanging of information is relatively small. Therefore, the graph to be rewritten has to be divided into a number of sub-graphs (*grains*) indicating the parts of the program graph that can be reduced in parallel. A real speed-up on parallel architectures can only be achieved if redexes that yield a sufficient large amount of computation, are evaluated in parallel while the intermediate links are sparsely used (*coarse grain* parallelism).

13.8 Other efficiency issues

Here are some additional suggestions how to make your program more efficient:

+ Transform a recursive function to a tail-recursive function where possible.
+ Accumulate results in parameters instead of in right-hand side results.
+ When functions return multiple results put these results in a strict tuple (can be indicated in the type).
+ Use macros for constant expressions instead of CAF's or functions.
+ Export the strictness information to other modules (the compiler will warn you if you don't).
+ Selections in a lazy context can better be transformed to functions which do a pattern match.
+ Higher order functions are nice but very inefficient, it is much better to use first order functions.
+ Constructors of high arity are inefficient.
+ Increase the heap space in the case that the garbage collector is going bananas.

A

Context-free syntax description

In this chapter the context-free syntax of CLEAN is given. In Section A.1 the construction of a CLEAN program out of definition and implementation modules is given. Hereafter the syntax for, respectively, defining functions (Section A.2), graphs (Section A.3,A.4), macros (Section A.4) and types (Section A.5) is presented. Overloading is treated in Section A.6. These sections have some production rules in common which are collected in Section A.7,A.8 and A.9.
Notice that the lay-out rule (see Section 3.6) permits the omission of the semi-colon (';') which ends a definition and of the braces ('{' and '}') which are used to group a list of definitions. The description of the identifiers and literals can be found in Section 3.4.

The following notational conventions are used in the context-free syntax descriptions:

[notion]	means that the presence of notion is optional
{notion}	means that notion can occur zero or more times
{notion}+	means that notion occurs at least once
{notion}-*list*	means one or more occurrences of notion separated by comma's
terminals	are printed in **bold 10 pts courier**
terminals	that can be left out in lay-out mode are printed in outlined courier
symbols	are printed in *italic* and represent identifiers and literals (see also Section 3.4)
~	is used for concatenation of notions
{notion}/-str	means the longest expression not containing the string str

A.1 CLEAN program

CLEANProgram = {Module}+
Module = DefinitionModule
| ImplementationModule
DefinitionModule = **definition module** *ModuleSymb* ;
{Definition}
| **system module** *ModuleSymb* ;
{Definition}
ImplementationModule = [**implementation**] **module** *ModuleSymb* ;
{Definition}

Definition = ImportDef
| TypeDef
| ClassDef
| FunctionDef
| GraphDef
| MacroDef

ImportDef = **import** {*ModuleSymb*}-*list* ;
| **from** *ModuleSymb* **import** {ImportSymbols}-*list* ;

ImportSymbols = *FunctionSymb*
| *ConstructorSymb*
| *SelectorVariable*
| *FieldSymb*
| *MacroSymb*
| *TypeSymb*
| *ClassSymb*

A.2 Function definition

FunctionDef = [FunctionTypeDef] DefOfFunction

FunctionTypeDef = *FunctionSymb* **::** FunctionType ;
| **(***FunctionSymb***)** [Fix][Prec] [**::** FunctionType] ;
FunctionType = [{[Strict] BrackType}+ **->**] Type [ClassContext] [UnqTypeUnEqualities]
ClassContext = **|** *ClassSymb-list TypeVariable* {**&** *ClassSymb-list TypeVariable* }
UnqTypeUnEqualities = **, [**{{*UniqueTypeVariable}*+ **<=** *UniqueTypeVariable*}-*list* **]**

DefOfFunction = {FunctionAltDef}+
FunctionAltDef = FunctionSymbol {Pattern}
{LetBeforeExpression}
{{**|** Guard} **=**[**>**] FunctionBody}+
[LocalFunctionAltDefs]

Pattern = [*Variable* **=:**] BrackPattern
BrackPattern = ConstructorSymbol
| PatternVariable
| BasicValuePattern
| ListPattern
| TuplePattern
| RecordPattern
| ArrayPattern
| **(**GraphPattern**)**
GraphPattern = ConstructorSymbol {Pattern}
| GraphPattern *ConstructorSymb* GraphPattern
| Pattern

PatternVariable = *Variable*
| _

BasicValuePattern = BasicValue

ListPattern = **[**[{LGraphPattern}-*list* [**:** GraphPattern]]**]**
LGraphPattern = GraphPattern
| *CharsDenot*

TuplePattern = **(**GraphPattern**,**{GraphPattern}-*list***)**

RecordPattern = **{**[*TypeSymb***|**] {FieldSymbol [**=** GraphPattern]}-*list***}**

ArrayPattern = **{**{GraphPattern}-*list***}**
| **{**{ArrayIndex **=** *Variable*}-*list***}**
| *StringDenot*

LetBeforeExpression = Lets {GraphDef}+
Lets = **Let** | **#** | **Let!** | **#!**

Guard = BooleanExpr
BooleanExpr = GraphExpr

FunctionBody = {StrictLet}
RootExpression ;
[LocalFunctionDefs]

StrictLet = **let!** { {GraphDef}+ } **in**

RootExpression = GraphExpr

LocalFunctionDefs = [**with**] { {LocalDef}+ }

LocalDef = GraphDef
| FunctionDef
LocalFunctionAltDefs = [**where**] ⦃ {LocalDef}+ ⦄

A.3 Graph definition and expression

GraphDef = Selector **=**[**:**] GraphExpr ;
Selector = BrackPattern

Graph = [Process] GraphExpr
GraphExpr = Application
| CaseExpr
| LetExpr

Application = {BrackGraph}+
| GraphExpr OperatorSymbol GraphExpr
BrackGraph = NodeSymbol
| GraphVariable
| BasicValue
| List
| Tuple
| Record
| RecordSelection
| Array
| ArraySelection
| LambdaAbstr
| **(**GraphExpr**)**

GraphVariable = *Variable*
| *SelectorVariable*

BasicValue = *IntDenot*
| *RealDenot*
| *BoolDenot*
| *CharDenot*

List = **[**[{LGraphExpr}-*list* [**:** GraphExpr]]**]**
| **[**GraphExpr [**,**GraphExpr]**..**[GraphExpr]**]**
| **[**GraphExpr \\ {Qualifier}-*list* **]**
LGraphExpr = GraphExpr
| *CharsDenot*
Qualifier = Generators {**|**Guard}
Generators = {Generator}-*list*
| Generator {**&** Generator}
Generator = Selector **<-** ListExpr
| Selector **<-:** ArrayExpr
ListExpr = GraphExpr
ArrayExpr = GraphExpr

Tuple = **(**GraphExpr**,**{GraphExpr}-*list* **)**

Record = **{**[*TypeSymb* **|**][RecordExpr **&**][{FieldSymbol **=** GraphExpr}-*list*]**}**
RecordSelection = RecordExpr**.**[*TypeSymb* **.**]*FieldSymb*
RecordExpr = GraphExpr

Array = **{**{GraphExpr}-*list* **}**
| **{**ArrayExpr **&** [{ArrayIndex **=** GraphExpr}-*list*] [\\ {Qualifier}-*list*]**}**
| **{**[ArrayExpr **&**] GraphExpr \\ {Qualifier}-*list*}
| *StringDenot*
ArrayIndex = **[**{IntegerExpr}-*list* **]**
ArraySelection = ArrayExpr**.**ArrayIndex

LambdaAbstr = **** {Pattern} **->** GraphExpr

CaseExpr = **case** GraphExpr **of**
⦃ {CaseAltDef}+ ⦄
| **if** BrackGraph BrackGraph BrackGraph
CaseAltDef = {Pattern}
{LetBeforeExpression}
{{**|** Guard} **->** FunctionBody}+

```
                                [LocalFunctionAltDefs]

LetExpresssion          =       let { {LocalDef}+ } in GraphExpr

Process                 =       {* I *}
                        |       {* P [at ProcIdExpr] *}
ProcIdExpr              =       GraphExpr
```

A.5 Macro definition

```
MacroDef                =       [MacroFixityDef] DefOfMacro
MacroFixityDef          =       (FunctionSymb) [Fix][Prec] ;
DefOfMacro              =       FunctionSymbol {Variable} :== FunctionBody ;
                                [LocalFunctionAltDefs]
```

A.6 Type definition

```
TypeDef                 =       AlgebraicTypeDef
                        |       RecordTypeDef
                        |       SynonymTypeDef
                        |       AbstractTypeDef

AlgebraicTypeDef        =       ::TypeLhs = ConstructorDef {|ConstructorDef} ;
RecordTypeDef           =       ::TypeLhs = {{FieldSymbol :: [Strict] Type}-list} ;
SynonymTypeDef          =       ::TypeLhs :== Type ;
AbstractTypeDef         =       ::TypeLhs ;

TypeLhs                 =       [*]TypeConstructor {[*] TypeVariable}
TypeConstructor         =       TypeSymb

UnqTypeAttrib           =       *
                        |       UniqueTypeVariable:
                        |       .

ConstructorDef          =       [QuantifiedVariables :] ConstructorSymb {[Strict] BrackType}
                        |       [QuantifiedVariables :] (ConstructorSymb) [Fix][Prec] {[Strict] BrackType}
QuantifiedVariables     =       {E. TypeVariable}+
Fix                     =       infixl
                        |       infixr
                        |       infix
Prec                    =       Digit
Strict                  =       !

Type                    =       {BrackType}+
BrackType               =       [UnqTypeAttrib] SimpleType
SimpleType              =       TypeConstructor
                        |       TypeVariable
                        |       BasicType
                        |       PredefAbstrType
                        |       ListType
                        |       TupleType
                        |       ArrayType
                        |       ArrowType
                        |       (Type)

TypeConstructor         =       TypeSymb
                        |       []
                        |       ({,}+)
                        |       {}
                        |       {!}
                        |       {#}
                        |       (->)
BasicType               =       Int
                        |       Real
                        |       Char
                        |       Bool
PredefAbstrType         =       World
                        |       File
                        |       ProcId
```

ListType = **[**Type**]**
TupleType = **(**[Strict] Type**,**{[Strict] Type}-*list***)**
ArrayType = **{**[Strict] Type**}**
| **{**#BasicType**}**
ArrowType = **(**{BrackType}+ **->** Type**)**

A.6 Class definition

ClassDef = TypeClassDef
| TypeClassInstanceDef
| TypeClassInstanceExportDef

TypeClassDef = **class** *ClassSymb TypeVariable* [ClassContext]
[[**where**] ⦃ {ClassMemberDef}+ ⦄]
| **class** *FunctionSymb TypeVariable* **::** FunctionType ;
| **class** **(***FunctionSymb***)** [Fix][Prec] *TypeVariable* **::** FunctionType ;

ClassMemberDef = FunctionTypeDef
[MacroDef] ;

TypeClassInstanceDef | **instance** *ClassSymb* [BrackType [**default**] [ClassContext]]
[[**where**] ⦃ {DefOfFunction}+ ⦄]

TypeClassInstanceExportDef
= **export** *ClassSymb* BasicType- *list* ;

A.7 Symbols

NodeSymbol = FunctionSymbol
| ConstructorSymbol
FunctionSymbol = *FunctionSymb*
| **(***FunctionSymb***)**
ConstructorSymbol
= *ConstructorSymb*
| **(***ConstructorSymb***)**
OperatorSymbol = *FunctionSymb*
| *ConstructorSymb*

ModuleSymb = LowerCaseId | UpperCaseId | FunnyId
FunctionSymb = LowerCaseId | UpperCaseId | FunnyId
ConstructorSymb = UpperCaseId | FunnyId
SelectorVariable = LowerCaseId
Variable = LowerCaseId
MacroSymb = LowerCaseId | UpperCaseId | FunnyId
FieldSymb = LowerCaseId
TypeSymb = UpperCaseId | FunnyId
TypeVariable = LowerCaseId
UniqueTypeVariable = LowerCaseId
ClassSymb = LowerCaseId | UpperCaseId | FunnyId

A.8 Identifiers

LowerCaseId = LowerCaseChar~{IdChar}
UpperCaseId = UpperCaseChar~{IdChar}
.ib.FunnyId; = {SpecialChar}+

LowerCaseChar = **a** | **b** | **c** | **d** | **e** | **f** | **g** | **h** | **i** | **j**
| **k** | **l** | **m** | **n** | **o** | **p** | **q** | **r** | **s** | **t**
| **u** | **v** | **w** | **x** | **y** | **z**
UpperCaseChar = **A** | **B** | **C** | **D** | **E** | **F** | **G** | **H** | **I** | **J**
| **K** | **L** | **M** | **N** | **O** | **P** | **Q** | **R** | **S** | **T**
| **U** | **V** | **W** | **X** | **Y** | **Z**
SpecialChar = **~** | **@** | **#** | **$** | **%** | **^** | **?** | **!**
| **+** | **-** | ***** | **<** | **>** | \ | **/** | **|** | **&** | **=**
| **:**
IdChar = LowerCaseChar
| UpperCaseChar
| Digit

```
                        |  _  |  `
```

A.9 Denotations

```
IntegerDenot    =  [Sign]~{Digit}+                           // decimal
                |  [Sign]~0~{OctDigit}+                      // octal
                |  [Sign]~0x~{HexDigit}+                     // hexadecimal
Sign            =  + | - | ~
RealDenot       =  [Sign~]{Digit~}+.{~Digit}+[~E[~Sign]{~Digit}+]
BoolDenot       =  True | False
CharDenot       =  CharDel~AnyChar/CharDel.CharDel
CharsDenot      =  CharDel~{AnyChar/CharDel}+.CharDel
StringDenot     =  StringDel~{AnyChar/StringDel}~StringDel

AnyChar         =  IdChar | ReservedChar | Special
ReservedChar    =  (   |   )  |   {     |   }  |   [  |   ]  |   ;  |   ,  |   .
Special         =  \n  |   \r |   \f |   \b  // newline,return,formf,backspace
                |  \t  |   \\|    \CharDel               // tab,backslash,character delete
                |  \StringDel                            // string delete
                |  \{OctDigit}+                          // octal number
                |  \x{HexDigit}+                         // hexadecimal number

Digit           =  0  |  1  |  2  |  3  |  4  |  5  |  6  |  7  |  8  |  9
OctDigit        =  0  |  1  |  2  |  3  |  4  |  5  |  6  |  7
HexDigit        =  0  |  1  |  2  |  3  |  4  |  5  |  6  |  7  |  8  |  9
                |  A  |  B  |  C  |  D  |  E  |  F
                |  a  |  b  |  c  |  d  |  e  |  f

CharDel         =  '
StringDel       =  "
```

B

Standard library

The standard library of CLEAN not only contains the well-known functions for arithmetic and manipulation of lists, arrays and the like, but there is also a lot of support for file I/O and window based I/O. The new CLEAN 1.0 I/O library makes the specification and combination of interactive programs possible on a very high level of abstraction. Notice that this new I/O library is not yet available on all platforms. The old CLEAN 0.8 library has been converted to CLEAN 1.0 syntax and is available on all platforms.

In the CLEAN library there are many modules. Modules which names start with `Std`… are the topmost interface modules of the library to be used by CLEAN programmers. In this appendix we have printed the names of types, constructors, functions, type-classes in bold to assist the reader in finding a definition.

The types of the functions in Std... are as general as possible and therefore include uniqueness type information (the funny dots and `u:` etc. in the types). For reasons of efficiency also the strictness information derived by the strictness analyser is exported (the exclamation marks in the types). For most programmers this information will often be of no importance, and if this is the case, simply ignore these funny marks.

B.1 CLEAN's Standard Environment

CLEAN Standard Environment imports all definitions exported by the mentioned modules:

```
definition module StdEnv

import
     StdOverloaded,
     StdClass,

     StdBool,
     StdInt,
     StdReal,
     StdChar,

     StdList,
     StdCharList,
     StdTuple,
     StdArray,
     StdString,
     StdFunc,
     StdMisc,

     StdFile,

     StdEnum
```

Below you find a summary of all the infix operators which are defined in these modules:

Operator	*Associativity*	*Precedence*	*Defined in*	*Description*
\`bind\`	none	0	StdFunc	monadic bind
:=	left	1	StdString	replace
||	right	2	StdBool	Boolean or
&&	right	3	StdBool	Boolean and
<>	none	4	StdClass	Not equal
>	none	4	StdClass	Greater than
<=	none	4	StdClass	Smaller than or equal to
>=	none	4	StdClass	Greater than or equal to
==	none	4	StdOverloaded	Equals
<	none	4	StdOverloaded	Smaller than
++	right	5	StdList	Concatenate lists
+++	right	5	StdOverloaded	Concatenate
+	left	6	StdOverloaded	Add
-	left	6	StdOverloaded	Substract
bitor	left	6	StdInt	Bitwise or
bitxor	left	6	StdInt	Bitwise xor
bitand	left	6	StdInt	Bitwise and
*	left	7	StdOverloaded	Multiply
/	none	7	StdOverloaded	Divide
<<	none	7	StdInt	Shift left
>>	none	7	StdInt	Shift right
mod	none	7	Stdint	Modulo
rem	none	7	StdInt	Remainder
^	right	8	StdOverloaded	Exponent
!	left	9	StdList	List subscript
o	right	9	StdFunc	Function composition
%	left	9	StdOverloaded	Slice

B.1.1 StdOverloaded: predefined overloaded operations

```
definition module StdOverloaded

class (+)  infixl 6     a    :: !a     !a   ->   a             //   Add arg1 to arg2
class (-)  infixl 6     a    :: !a     !a   ->   a             //   Subtract arg2 from arg1
class zero              a    :: a                              //   Zero (unit element for addition)

class (*)  infixl 7     a    :: !a     !a   ->   a             //   Multiply arg1 with arg2
class (/)  infix  7     a    :: !a     !a   ->   a             //   Divide arg1 by arg2
class one               a    :: a                              //   One (unit element for multiplication)

class (^)  infixr 8     a    :: !a     !a   ->   a             //   arg1 to the power of arg2
class abs               a    :: !a          ->   a             //   Absolute value
class sign              a    :: !a          ->   Int           //   1 (pos value) -1 (neg value) 0 (if zero)
class ~                 a    :: !a          ->   a             //   -a1

class (==) infix  4     a    :: !a     !a   ->   Bool          //   True if arg1 is equal to arg2

class (<)  infix  4     a    :: !a     !a   ->   Bool          //   True if arg1 is less than arg2

class toInt             a    :: !a          ->   Int           //   Convert into Int
class toChar            a    :: !a          ->   Char          //   Convert into Char
class toBool            a    :: !a          ->   Bool          //   Convert into Bool
class toReal            a    :: !a          ->   Real          //   Convert into Real
class toString          a    :: !a          ->   String        //   Convert into String

class fromInt           a    :: !Int        -> a               //   Convert from Int
class fromChar          a    :: !Char       -> a               //   Convert from Char
class fromBool          a    :: !Bool       -> a               //   Convert from Bool
class fromReal          a    :: !Real       -> a               //   Convert from Real
class fromString        a    :: !String     -> a               //   Convert from String

class length            m    :: !(m a) -    > Int              //   Number of elements in arg
                                                               //   used for list like structures
                                                               //                       (linear time)

class (%)      infixl 9 a    :: !a !(!Int,!Int) -> a           //   Slice a part from arg1

class (+++)    infixr 5 a    :: !a     !a   -> a               //   Append args
```

B.1.2 StdClass: predefined classes

```
definition module StdClass

import StdOverloaded
from StdBool import not

class PlusMin a    | +, -, zero a

class MultDiv a    | *, /, one a

class Arith a | PlusMin, MultDiv, abs, sign, ~ a

class IncDec a     | + , - , one , zero a
where
     inc :: !a -> a          | + , one a
     inc x :== x + one

     dec :: !a -> a          | - , one a
     dec x :== x - one

class Enum a       | <, IncDec a

class Eq a         | == a
where
     (<>) infix  4:: !a !a   ->   Bool | Eq a
     (<>) x y :== not (x == y)

class Ord a        | < a
where
     (>) infix  4:: !a !a -> Bool | Ord a
     (>) x y   :== y < x

     (<=) infix  4::!a !a -> Bool | Ord a
     (<=) x y  :== not (y<x)

     (>=) infix  4::!a !a -> Bool | Ord a
     (>=) x y  :== not (x<y)

     min::!a !a -> a | Ord a
     min x y   :== if (x<y) x y

     max::!a !a -> a | Ord a
     max x y   :== if (x<y) y x
```

B.1.3 StdBool: operations on Booleans

```
system module StdBool

import    StdOverloaded

instance ==        Bool

instance toBool    Bool
instance toString  Bool

instance fromBool  Bool
instance fromBool  {#Char}                          //   String :== {#Char}

//   Additional Logical Operators:

not                :: !Bool          ->   Bool      //   Not arg1
(||)      infixr 2 :: !Bool Bool  ->   Bool         //   Conditional or  of arg1 and arg2
(&&)      infixr 3 :: !Bool Bool  ->   Bool         //   Conditional and of arg1 and arg2

//   Miscellaneous:

otherwise :== True                                  //   To be used in guards
```

B.1.4 StdInt: operations on Integers

```
system module StdInt

import    StdOverloaded

instance +         Int
instance -         Int
instance zero      Int
instance *         Int
instance /         Int
instance one       Int

instance ^         Int
instance abs       Int
instance sign      Int
instance ~         Int

instance ==        Int

instance <         Int

instance toInt     Int
instance toChar    Int
instance toReal    Int
instance toString  Int

instance fromInt   Int
instance fromInt   Char
instance fromInt   Real
instance fromInt   {#Char}                          //   String :== {#Char}

// Additional functions for integer arithmetic:

(mod)     infix  7      :: !Int !Int   ->   Int       //   arg1 modulo arg2
(rem)     infix  7      :: !Int !Int   ->   Int       //   remainder after division
gcd                     :: !Int !Int   ->   Int       //   Greatest common divider
lcm                     ::  Int !Int   ->   Int       //   Least common multiple

//   Test on Integers:

isEven                  :: !Int        ->   Bool      //   True if arg1 is an even number
isOdd                   :: !Int        ->   Bool      //   True if arg1 is an odd  number

//   Operators on Bits:

(bitor)   infix 6   :: !Int !Int   ->   Int       //   Bitwise Or of arg1 and arg2
(bitand)  infix 6   :: !Int !Int   ->   Int       //   Bitwise And of arg1 and arg2
(bitxor)  infix 6   :: !Int !Int   ->   Int       //   Exclusive-Or arg1 with mask arg2
(<<)      infix 7   :: !Int !Int   ->   Int       //   Shift arg1 to the left arg2 bit places
(>>)      infix 7   :: !Int !Int   ->   Int       //   Shift arg1 to the right arg2 bit places
bitnot              :: !Int        ->   Int       //   One's complement of arg1
```

B.1.5 StdReal: operations on Reals

```
system module StdReal

import    StdOverloaded

instance +         Real
instance -         Real
instance zero      Real

instance *         Real
instance /         Real
instance one       Real

instance ^         Real
instance abs       Real
instance sign      Real
instance ~         Real

instance ==        Real

instance <         Real

instance toInt     Real
instance toReal    Real
instance toString  Real

instance fromReal  Int
instance fromReal  Real
instance fromReal  {#Char}             //   String :== {#Char}

//   Logarithmical Functions:
```

```
ln       :: !Real ->   Real          //   Logarithm base e
log10    :: !Real ->   Real          //   Logarithm base 10
exp      :: !Real ->   Real          //   e to to the power
sqrt     :: !Real ->   Real          //   Square root

//   Trigonometrical Functions:

sin      :: !Real  ->  Real          //   Sinus
cos      :: !Real  ->  Real          //   Cosinus
tan      :: !Real  ->  Real          //   Tangens
asin     :: !Real  ->  Real          //   Arc Sinus
acos     :: !Real  ->  Real          //   Arc Cosinus
atan     :: !Real  ->  Real          //   Arc Tangus

//   Additional conversion:

entier:: !Real     ->  Int           //   Cconvert Real into Int by taking entier
```

B.1.6 StdChar: operations on Characters

```
system module StdChar

import   StdOverloaded

instance +         Char
instance -         Char
instance zero      Char
instance one       Char

instance ==        Char

instance <         Char

instance toInt     Char
instance toChar    Char
instance toString  Char

instance fromChar  Int
instance fromChar  Char
instance fromChar  {#Char}           //   String :== {#Char}

//   Additional conversions:

digtoInt      :: !Char -> Int        //   Convert Digit into Int
toUpper       :: !Char -> Char       //   Convert Char into an uppercase Char
toLower       :: !Char -> Char       //   Convert Char into a  lowercase Char

//   Tests on Characters:

isAscii       :: !Char -> Bool       //   True if arg1 is an ASCII character
isControl     :: !Char -> Bool       //   True if arg1 is a  control character
isPrint       :: !Char -> Bool       //   True if arg1 is a  printable character
isSpace       :: !Char -> Bool       //   True if arg1 is a  space, tab etc
isUpper       :: !Char -> Bool       //   True if arg1 is an uppercase character
isLower       :: !Char -> Bool       //   True if arg1 is a  lowercase character
isAlpha       :: !Char -> Bool       //   True if arg1 is a  letter
isDigit       :: !Char -> Bool       //   True if arg1 is a  digit
isAlphanum    :: !Char -> Bool       //   True if arg1 is an alphanumerical character
```

B.1.7 StdList: operations on Lists

```
definition module StdList

import   StdClass

instance ==   [a] | Eq a

instance <    [a] | Ord a

instance toString   [a] | ToChar a                   // Convert [e to Char] into String
instance fromString [a] | FromChar a                 // Convert String into [Char to e]

instance length    []
instance %         [a]
```

```
//   List Operators:

(!)       infixl 9 :: [.a] Int -> .a                      //   Get nth element of the list
(++)      infixr 5 :: ![.a] u:[.a] -> u:[.a]              //   Append args
flatten            :: ![.[a]] -> [a]                      //   e0 ++ e1 ++ ... ++ en
isEmpty            :: ![.a] -> Bool                       //   [] ?

//   List breaking or permuting functions:

hd             :: ![.a] -> .a                             //   Head of the list
tl             :: !u:[.a] -> u:[.a]                       //   Tail of the list
last           :: ![.a] -> .a                             //   Last element of the list
take           :: !Int [.a] -> [.a]                       //   Take first arg1 elem of the list
drop           :: Int !u:[.a] -> u:[.a]                   //   Drop first arg1 elem from the list
takeWhile      :: (a -> .Bool) !.[a] -> .[a]              //   Take elements while pred holds
dropWhile      :: (a -> .Bool) !u:[a] -> u:[a]            //   Drop elements while pred holds
filter         :: (a -> .Bool) !.[a] -> .[a]              //   Drop all elements not satisfying pred
insert         :: (a a -> .Bool) a !u:[a] -> u:[a]        //   Insert arg2 when pred arg2 elem holds
remove         :: !Int !u:[.a] -> u:[.a]                  //   Remove arg2!arg1 from list
reverse        :: !.[a] -> [a]                            //   Reverse the list
span               :: !(a -> .Bool) !u:[a] -> (.[a],u:[a])    //   (takeWhile list,dropWhile list)
splitAt        :: !Int u:[.a] -> ([.a],u:[.a])            //   (take n list,drop n list)

//   Creating lists:

map            :: (.a -> .b) ![.a] -> [.b]                //   [f e0,f e1,f e2,...
iterate        :: (a -> a) a -> .[a]                      //   [a,f a,f (f a),...
indexList      :: !.[a] -> [Int]                          //   [0..length list - 1]r
epeatn         :: !.Int a -> .[a]                         //   [e0,e0,...,e0] of length n
repeat         :: a -> [a]                                //   [e0,e0,...
unzip          :: ![(a,b)] -> ([a],[b])                   //   ([a0,a1,...],[b0,b1,...])
zip2           :: ![.a] [.b] -> [(.a,.b)]                 //   [(a0,b0),(a1,b1),...
zip            :: !(![.a],[.b]) -> [(.a,.b)]              //   [(a0,b0),(a1,b1),...
diag2          :: !.[a] .[b]      -> [.(a,b)]             //   [(a0,b0),(a1,b0),(a0,b1),...
diag3          :: !.[a] .[b] .[c] -> [.(a,b,c)]           //   [(a0,b0,c0),(a1,b0,c0),...

//   Folding and scanning:

foldl     :: (.a -> .(.b -> .a)) !.a ![.b] -> .a     //   op(...(op (op (op r e0) e1)...en)
foldr     :: (.a -> .(.b -> .b)) !.b ![.a] -> .b     //   op e0 (op e1(...(op r en)...)

// for efficiency reasons, foldl and folr are defined as macros,
// so that applications of these functions will be inlined !

// foldl  :: (.a -> .(.b -> .a)) !.a ![.b] -> .a     //   op(...(op (op (op r e0) e1)...en)
foldl op r l :== foldl r l
where
     foldl r []         = r
     foldl r [a:x]  = foldl (op r a) x

// foldr  :: (.a -> .(.b -> .b)) !.b ![.a] -> .b     //   op e0 (op e1(...(op r en)...)
foldr op r l :== foldr r l
where
     foldr r []         = r
     foldr r [a:x]  = op a (foldr r x)

scan :: (a -> .(.b -> a)) a ![.b] -> .[a]            //   [r,op r e0,op (op r e0) e1,...

//   On Booleans

and            :: ![.Bool] -> Bool                        //   e0 && e1 ... && en
or             :: ![.Bool] -> Bool                        //   e0 || e1 ... || en
any            :: (.a -> .Bool) ![.a] -> Bool             //   True, if ei is True for some i
all            :: (.a -> .Bool) ![.a] -> Bool             //   True, if ei is True for all i

//   When ordering is defined on list elements

maxList        :: !.[a]           -> a          | Ord a   //   Maximum element of list
minList        :: !.[a]           -> a          | Ord a   //   Minimum element of list
sort           :: !u:[a]          -> u:[a]      | Ord a   //   Sort the list
merge          :: !.[a] !u:[a]    -> u:[a]      | Ord a   //   Merge two sorted lists giving a sorted list

//   When equality is defined on list elements

isMember       :: a !.[a]         -> .Bool | Eq a         //   Is element in list
removeMembers  :: u:[a] .[a]      -> u:[a] | Eq a         //   Remove arg2s from list arg1
removeDup      :: !.[a]           -> .[a]  | Eq a         //   Remove all duplicates from list
limit          :: !.[a]           -> a     | Eq a         //   [...,a,a]
```

```
//   Overloaded definition of sum, product, average

sum       :: !.[a] -> a | + , zero  a       //   sum of list elements, sum [] = zero
prod      :: !.[a] -> a | * , one  a        //   product of list elements, prod [] = one
avg       :: !.[a] -> a | / , IncDec a      //   average of list elements, avg [] gives error!
```

B.1.8 StdCharList: operations on lists of characters

```
definition module StdCharList

//   Functions for outlining

cjustify       :: !.Int ![.Char] -> .[Char]      // Center [Char] in field with width arg1
ljustify       :: !.Int ![.Char] -> .[Char]      // Left justify [Char] in field with width arg1
rjustify       :: !.Int ![.Char] ->  [Char]      // Right justify [Char] in field with width arg1n

flatLines      :: ![[u:Char]] -> [u:Char]        // Concatenate by adding newlines
mkLines        :: ![Char] -> [[Char]]            // Split in lines removing newlines
spaces         :: !.Int -> .[Char]               // Make [Char] containing n space characters
```

B.1.9 StdTuple: operations on Tuples

```
definition module StdTuple

import StdClass

instance ==    (a,b)     |    Eq a & Eq b
instance ==    (a,b,c)   |    Eq a & Eq b & Eq c

instance <     (a,b)     |    Ord a & Ord b
instance <     (a,b,c)   |    Ord a & Ord b & Ord c

fst  :: !(!.a,.b) -> .a                                             // t1 of (t1,t2)
snd  :: !(.a,!.b) -> .b                                             // t2 of (t1,t2)

fst3      :: !(!.a,.b,.c) -> .a                                     // t1 of (t1,t2,t3)
snd3      :: !(.a,!.b,.c) -> .b                                     // t2 of (t1,t2,t3)
thd3      :: !(.a,.b,!.c) -> .c                                     // t3 of (t1,t2,t3)

app2      :: !(.(.a -> .b),.(.c -> .d)) !(.a,.c) -> (.b,.d)         // f (a,b) = (f a,f b)
app3      :: !(.(.a -> .b),.(.c -> .d),.(.e -> .f)) !(.a,.c,.e) -> (.b,.d,.f)
                                                                    // f (a,b,c) = (f a,f b,f c)

curry     :: !.((.a,.b) -> .c) .a .b -> .c                          // f a b      = f (a,b)
uncurry   :: !.(.a -> .(.b -> .c)) !(.a,.b) -> .c                   // f (a,b)    = f a b
```

B.1.10 StdArray: operations on Arrays

```
definition module StdArray

import _SystemArray

system module _SystemArray

/*
Warning:
1)   Arrays currently get a special treatment in the CLEAN compiler.
     This means that you shouldn't rename the functions declared here,
     and that you shouldn't make other instances of Array
2)   The structure of this module will change in a future release
*/

class Array a
where

     select         :: ! .(a .e) !Int -> .e                 | select_u e
     uselect        :: ! u:(a e) !Int -> (e, ! u:(a e))     | uselect_u e

     size           :: ! .(a .e)  -> Int                    | size_u e
     usize          :: ! u:(a .e) -> (!Int, ! u:(a .e))     | usize_u e

     update         :: !* (a .e)  !Int .e -> * (a .e)       | update_u e
     createArray    :: !Int e -> * (a e)                    | createArray_u e
```

```
instance Array     {} default, {!}, {#}

class ArrayElem e | select_u, uselect_u, size_u, usize_u, update_u, createArray_u,
                                                                    defaultArrayvalue e

// Operation on unboxed arrays

class select_u e        :: !   { #.e } !Int -> .e
class uselect_u e       :: ! u:{ # e } !Int -> (!e, ! u:{ #e })
class size_u e          :: !   { #.e } -> Int
class usize_u e         :: ! u:{ #.e } -> (!Int, ! u:{ #.e })
class update_u e        :: ! * { #.e } !Int !.e -> * { #.e }
class createArray_u e   :: !Int !e -> *{ #e }


instance select_u       a, Int, Real, Char, Bool, File
instance uselect_u      a, Int, Real, Char, Bool, File
instance size_u              a, Int, Real, Char, Bool, File
instance usize_u             a, Int, Real, Char, Bool, File
instance update_u       a, Int, Real, Char, Bool, File
instance createArray_u  a, Int, Real, Char, Bool, File

class     defaultArrayvalue e :: .e
instance  defaultArrayvalue Int, Real, Char, Bool, File, a
```

B.1.11 StdString: operations on Strings

```
system module StdString

import    StdOverloaded

:: String :== {#Char}

instance ==            {#Char}

instance <             {#Char}

instance toString      {#Char}
instance toInt         {#Char}
instance toReal        {#Char}

instance fromString    {#Char}

instance %             {#Char}

instance +++           {#Char}

//   additional operator

(:=) infixl 1 :: !String !(!Int,!Char) -> String     //   non-destructive update of the i-th element
```

B.1.12 StdFunc: operations on polymorphic functions

```
definition module StdFunc

// Some Classical Functions

I                   :: !.a -> .a                             // Identity function
K                   :: !.a .b -> .a                          // Konstant function
S                   :: .(a -> .(.b -> .(a -> .c))) .b a -> .c  // distribution function
flip                :: .(.a -> .(.b -> .c)) .b .a -> .c      // Flip arguments

(o) infixr  9 ::  u:(.a -> .b) u:(.c -> .a) -> u:(.c -> .b)   // Function composition

twice               :: !(.a -> .a) .a   -> .a                // f (f x)
while               :: !(a -> .Bool) (a -> a) a ->  a        // while (p x) (f x) else x
until               :: !(a -> .Bool) (a -> a) a ->  a        // until (p x) x else (f x)
iter                :: !Int (.a -> .a) .a -> .a              // f (f..(f x)..)

//   Some handy functions for transforming unique states:

::St s a :== s -> (a,s)

seq       :: ![.(.s -> .s)] .s -> .s                         // fn-1 (..(f1 (f0 x))..)
seqList   :: ![St .s .a] .s -> ([.a],.s)                     // fn-1 (..(f1 (f0 x))..)
                                                             // monadic style:
```

```
(`bind`) :: w:(St .s .a) v:(.a -> .(St .s .b)) -> u:(St .s .b), [u <= v, u <= w]
return   :: u:a -> u:(St .s u:a)
```

B.1.13 StdMisc: miscellaneous functions

```
system module StdMisc

abort    :: !String -> .a                        // stop reduction, print argument and core dump
undef    :: .a                                   // fatal error, stop reduction.
```

B.1.14 StdFile: File based I/O

```
system module StdFile

import StdString

//   File modes synonyms

FReadText      :== 0    //   Read from a text file
FWriteText     :== 1    //   Write to a text file
FAppendText    :== 2    //   Append to an existing text file
FReadData      :== 3    //   Read from a data file
FWriteData     :== 4    //   Write to a data file
FAppendData    :== 5    //   Append to an existing data file

//   Seek modes synonyms

FSeekSet       :== 0    //   New position is the seek offset
FSeekCur       :== 1    //   New position is the current position plus the seek offset
FSeekEnd       :== 2    //   New position is the size of the file plus the seek offset

::   *Files

//   Opening and Closing a File from the FileSystem:

openfiles::!*World -> (!*Files,!*World)

closefiles::!*Files !*World -> *World

fopen::!String !Int !*Files -> (!Bool,!*File,!*Files)
/*   Opens a file for the first time in a certain mode (read, write or append, text or data).
     The Boolean output parameter reports success or failure. */

fclose::!*File !*Files -> (!Bool,!*Files)

freopen::!*File !Int -> (!Bool,!*File)
/*   Re-opens an open file in a possibly different mode.
     The Boolean indicates whether the file was successfully closed before reopening. */

//   Reading from a File:

freadc::!*File -> (!Bool,!Char,!*File)
/*   Reads a character from a text file or a byte from a datafile.
     The Boolean indicates succes or failure */

freadi::!*File -> (!Bool,!Int,!*File)
/*   Reads an Integer from a textfile by skipping spaces, tabs and newlines and
     then reading digits, which may be preceeded by a plus or minus sign.
     From a datafile FReadI will just read four bytes (a CLEAN Int). */

freadr::!*File -> (!Bool,!Real,!*File)
/*   Reads a Real from a textfile by skipping spaces, tabs and newlines and then
     reading a character representation of a Real number.
     From a datafile FReadR will just read eight bytes (a CLEAN Real). */

freads:: ! *File !Int -> (!String,!*File)
/*   Reads n characters from a text or data file, which are returned as a String.
     If the file doesn't contain n characters the file will be read to the end
     of the file. An empty String is returned if no characters can be read. */

freadline :: !*File -> (!String,!*File)
/*   Reads a line from a textfile. (including a newline character, except for the last
     line) FReadLine cannot be used on data files. */

//   Writing to a File:
```

```
fwritec :: !Char !*File -> *File
/*   Writes a character to a textfile.
     To a datafile fwritec writes one byte (a CLEAN CHAR). */

fwritei ::!Int !*File -> *File
/*   Writes an Integer (its textual representation) to a text file.
     To a datafile FWriteI writes four bytes (a CLEAN Int). */

fwriter ::!Real !*File -> *File
/*   Writes a Real (its textual representation) to a text file.
     To a datafile FWriteR writes eight bytes (a CLEAN Real). */

fwrites ::!String !*File -> *File
/*   Writes a String to a text or data file. */

//   Testing:

fend ::!*File -> (!Bool,!*File)
/*   Tests for end-of-file. */

ferror ::!*File -> (!Bool,!*File)
/*   Has an error occurred during previous file I/O operations? */

fposition      :: !*File -> (!Int,!*File)
/*   returns the current position of the file poInter as an Integer.
     This position can be used later on for the FSeek function. */

fseek ::!*File !Int !Int -> (!Bool,!*File)
/*   Move to a different position in the file, the first Integer argument is the offset,
     the second argument is a seek mode. (see above). True is returned if successful. */

//   Predefined files.

stdio ::!*Files -> (!*File,!*Files)
/*   Open the 'Console' for reading and writing. */

stderr ::*File
/*   Open the 'Errors' file for writing only. May be opened more than once. */

//   Opening and reading Shared Files:

sfopen ::!String !Int !*Files -> (!Bool,!File,!*Files)
/*   With SFOpen a file can be opened for reading more than once.
     On a file opened by SFOpen only the operations beginning with SF can be used.
     The SF... operations work just like the corresponding F... operations.
     They can't be used for files opened with FOpen or FReOpen. */

sfreadc        :: !File -> (!Bool,!Char,!File)
sfreadi        :: !File -> (!Bool,!Int,!File)
sfreadr        :: !File -> (!Bool,!Real,!File)
sfreads        :: !File !Int -> (!String,!File)
sfreadline     :: !File -> (!String,!File)
sfseek         :: !File !Int !Int -> (!Bool,!File)

sfend          :: !File -> Bool
sfposition     :: !File -> Int
/*   The functions SFEnd and SFPosition work like FEnd and FPosition, but don't return a
     new file on which other operations can continue. They can be used for files opened
     with SFOpen or after FShare, and in guards for files opened with FOpen or FReOpen. */

//   Convert a *File into:

fshare         :: !*File -> File
/*   Change a file so that from now it can only be used with SF... operations. */
```

B.1.15 StdEnum: handling dot-dot expressions

The definitions listed in `StdEnum` are used by the CLEAN compiler to handle dot-dot expressions. Dot-dot expressions can be used for objects of type `Int`, `Char` and `Real`. Dot-dot expressions can also be used of objects of arbitrary user-defined types provided that the indicated classes have been instantiated for objects of that type.

```
definition module StdEnum

import    _SystemEnum
```

```
/*
     This module must be imported if dotdot expressions are used

          [from .. ]         -> _from from
          [from .. to]       -> _from_to from to
          [from, then .. ]   -> _from_then from then
          [from, then .. ]   -> _from_then_to from then to
*/


system module _SystemEnum

from StdClass import Enum
from StdBool import not

from           ::  a         -> [a]    | IncDec , Ord a
from_to        :: !a !a      -> [a]    | Enum a
from_then      ::  a  a      -> [a]    | Enum a
from_then_to   :: !a !a !a   -> [a]    | Enum a

lteq a b  -= not (b < a)
minus a b      :== a - b

implementation module _SystemEnum

import    StdEnv

lteq  a b :== not (b < a)
minus a b :== a - b

from ::  a -> [a] | IncDec , Ord a
from n = [n | _from (inc n)]

from_to :: !a !a -> [a] | Enum a
from_to n e
| n <= e       = [n | _from_to (inc n) e]
| otherwise    = []

from_then ::  a  a -> [a] | Enum a
from_then n1 n2 = [n1 | _from_by n2 (n2-n1)]
where
     from_by :: a a -> [a] | Enum a
     from_by n s = [n | _from_by (n+s) s]

from_then_to : !a !a !a -> [a] | Enum a
from_then_to n1 n2 e
| n1 <= n2     = _from_by_to n1 (n2-n1) e
| otherwise    = _from_by_down_to n1 (n2-n1) e
where
     from_by_to :: !a !a !a -> [a] | Enum a
     from_by_to n s e
          | n <= e       = [n | _from_by_to (n+s) s e]
          | otherwise    = []

     from_by_down_to :: !a !a !a -> [a] | Enum a
     from_by_down_to n s e
          | n >= e       = [n | _from_by_down_to (n+s) s e]
          | otherwise    = []

from_to :: !Int !Int -> [Int]
from_to n e
| n <= e       = [n | from_to (inc n) e]
| otherwise    = []B.1.15
```

B.2 Creating interactive processes

```
definition module StdEventIO


::  InitIO l p
    :== [.IOFunction (PState l p)]
::  IODef l p
    =   {   ioDefInit  :: .InitIO l p           // The initial actions of the process
        ,   ioDefAbout :: String                // The name of the process
        }
```

```
//  Starting an interactive process:

OpenIO :: !(IODef .l .p) (.l,.p) !*World -> *World
/*  OpenIO starts an interactive process specified by the IODef argument. The program state
    argument consisting of a local and public part serves as initial program state. If the
    interactive process has been successfully created, the functions in InitIO are evaluated
    from left-to-right. This is followed by the further evaluation of the interactive
    process. In the cause of the evaluation many new processes can be created and terminated.
    The interactive process created by OpenIO is the root process. OpenIO terminates as
    soon as all interactive processes (including the root process) have terminated. OpenIO
    returns the final world, consisting of the final file system and the event stream. */


NewIO :: !(IODef .l .p) (.l,.p) !(IOState .l` .p`) -> IOState .l` .p`
/*  If the interactive process is active, NewIO starts a new interactive process that will
    run interleaved with the current interactive processes. The new interactive process is
    specified by the IODef argument. Creation of the new interactive process is done as in
    OpenIO. The functions in InitIO are evaluated from left-to-right before any abstract
    event handler of the new interactive process is evaluated. The new interactive process
    becomes the active process (so the current interactive process is deactivated).
    If the interactive process is inactive, NewIO does nothing. */


ShareIO :: !(IODef .l .p) .l !(IOState .l` .p) -> IOState .l` .p

/*  If the interactive process is active, ShareIO starts a new interactive process that will
    run interleaved with the current interactive processes. The new interactive process is
    specified by the IODef argument. Creation of the new interactive process is done as in
    OpenIO. The functions in InitIO are evaluated from left-to-right before any abstract
    event handler of the new interactive process is evaluated. The new interactive process
    becomes the active process (so the current interactive process is deactivated). The new
    interactive process can communicate with all interactive processes by means of the file
    system or by message passing. The new interactive process can communicate with all
    interactive processes of the process group of the interactive process that spawned it by
    means of the public program state component.
    If the interactive process is inactive, ShareIO does nothing. */


NewSubIO :: !(IODef .l .p) (.l,.p) !(IOState .l` .p`) -> IOState .l` .p`

/*  If the interactive process is active, NewSubIO starts a new interactive subprocess that
    will run interleaved with the current interactive processes. The new interactive
    subprocess is specified by the IODef argument. Creation of the new interactive subprocess
    is done as in NewIO.
    A subprocess differs from the interactive processes created above by sharing the user
    interface of its parent process. Logically, all its abstract devices are private to the
    subprocess, but to the user they will appear to be merged with the abstract devices of
    the parent process. Inter-process communication proceeds as usual. Every interactive
    process (including subprocesses) can create further subprocesses. This results in a tree
    of subprocesses, all of which are subordinate to the top interactive process. (See also
    the notes of termination at QuitIO.)
    If the interactive process is inactive, NewSubIO does nothing. */


ShareSubIO :: !(IODef .l .p) .l !(IOState .l` .p) -> IOState .l` .p

/*  If the interactive process is active, NewSubIO starts a new interactive subprocess that
    will run interleaved with the current interactive processes. The new interactive
    subprocess is specified by the IODef argument. Creation of the new interactive subprocess
    is done as in ShareIO.
    A subprocess differs from the interactive processes created above by sharing the user
    interface of its parent process. Logically, all its abstract devices are private to the
    subprocess, but to the user they will appear to be merged with the abstract devices of
    the parent process. Inter-process communication proceeds as usual. Every interactive
    process (including subprocesses) can create further subprocesses. This results in a tree
    of subprocesses, all of which are subordinate to the top interactive process. (See also
    the notes of termination at QuitIO.)
    If the interactive process is inactive, NewSubIO does nothing. */


QuitIO    :: !(IOState .l .p) -> IOState .l .p

/*  QuitIO removes all abstract devices that are held in the interactive process. If the
    interactive process has subprocesses then these will also be quitted recursively. As a
    result evaluation of this interactive process including all subprocesses will terminate. */
```

```
HideIO    :: !(IOState .l .p) -> IOState .l .p
ShowIO    :: !(IOState .l .p) -> IOState .l .p

/*  If the interactive process is active, HideIO hides the interactive process, and ShowIO
    makes it visible. Note that hiding an interactive process does NOT disable the process
    but simply makes it invisible.
    If the interactive process is inactive, HideIO and ShowIO do nothing. */


RequestIO :: !String !(IOState .l .p) -> IOState .l .p

/*  If the interactive process is inactive, RequestIO alerts the user that the interactive
    process needs to become active. If the string argument is not empty, then this alert will
    consist of a notice displaying the string. An interactive process can issue an arbitrary
    number of requests.
    If the interactive process is active, RequestIO does nothing. */
```

B.3 Event based I/O

B.3.1 Windows

StdWindowDef: the window device

```
definition module StdWindowDef

import StdIOCommon
from   StdPicture import DrawFunction, Picture

::  Window         c    ps = Window           Title (c    ps) [WindowAttribute      ps ]
::  WindowLS       c ls ps = WindowLS       ls Title (c ls ps) [WindowAttribute *(ls,ps)]
::  DialogWindow   c    ps = DialogWindow     Title (c    ps) [WindowAttribute      ps ]
::  DialogWindowLS c ls ps = DialogWindowLS ls Title (c ls ps) [WindowAttribute *(ls,ps)]

::  WindowFrame :== Rectangle

::  WindowAttribute ps                                   // Default:
    //  Attributes for all windows:
    =   WindowId          Id                             // no id
    |   WindowPos         ItemPos                        // system dependent
    |   WindowSize        Size                           // screen size
    |   WindowItemSpace   Size                           // system dependent
    |   WindowOk          Id                             // no default (Custom)ButtonControl
    |   WindowCancel      Id                             // no cancel  (Custom)ButtonControl
    |   WindowStandBy                                    // system dependent
    |   WindowHide                                       // initially visible
    |   WindowClose       (IOFunction ps)                // user can't close window
    |   WindowUpdate      (UpdateFunction ps)            // no update actions
    //  Attributes for DialogWindows only:
    |   WindowMargin      Size                           // system dependent
    //  Attributes for Windows only:
    |   WindowSelectState SelectState                    // Able
    |   WindowDomain      PictureDomain                  // {corner1=zero,
                                                         //  corner2={max range,max range}}
    |   WindowOrigin      Point                          // Left top of picture domain
    |   WindowInitDraw    [DrawFunction]                 // No initial drawing in window
    |   WindowMinimumSize Size                           // system dependent
    |   WindowResize      (WindowResizeFunction ps)      // fixed size
    |   WindowActivate    (IOFunction ps)                // I
    |   WindowDeactivate  (IOFunction ps)                // I
    |   WindowMouse       SelectState (MouseFunction ps) // no mouse input
    |   WindowKey         SelectState (KeyFunction   ps) // no keyboard input
    |   WindowCursor      CursorShape                    // no change of cursor

::  WindowResizeFunction ps
    :== WindowFrame ->                                   // old window frame
        WindowFrame ->                                   // new window frame
        ps          -> ps
::  CursorShape
    =   StandardCursor
    |   BusyCursor
    |   IBeamCursor
    |   CrossCursor
    |   FatCrossCursor
    |   ArrowCursor
    |   HiddenCursor
```

StdWindowType: window types

```
definition module StdWindowType

import StdOverloaded


::  WindowType


instance ==       WindowType
instance toString WindowType

WindowType         :: WindowType
WindowLSType       :: WindowType
DialogWindowType   :: WindowType
DialogWindowLSType :: WindowType
```

StdWindow: window handling

```
definition module StdWindow

import StdIOCommon
import StdWindowDef
import StdWindowType
import StdControl
from   iostate    import IOState, PState, Files
from   StdPicture import DrawFunction, Picture


//  Functions applied to non-existent windows or unknown ids have no effect.

class Windows wdef
where
    OpenWindow      :: !Int !(wdef (PState .l .p)) !(IOState .l .p) -> IOState .l .p
    OpenModalWindow ::      !(wdef (PState .l .p)) !( PState .l .p) ->  PState .l .p

instance Windows (Window         c   )    | Controls   c
instance Windows (WindowLS       c ls)    | ControlsLS c
instance Windows (DialogWindow   c   )    | Controls   c
instance Windows (DialogWindowLS c ls)    | ControlsLS c

/*  If the interactive process is active, Open(Modal)Window opens the given window.
    OpenModalWindow:
        always opens a window at the front-most position.
    OpenWindow:
        opens a window behind the window indicated by the integer index.
        Index value 1 indicates the top-most window.
        Index value M indicates the bottom-most modal window, if there are M modal windows.
        Index value N indicates the bottom-most window, if there are N windows.
        If index<M, then the new window is added behind the bottom-most modal window
                    (at index M).
        If index>N, then the new window is added behind the bottom-most window (at index N).
    In case a window with the same Id is already open then that window will be activated.
    In case the window does not have an Id, it will obtain an Id which is fresh with respect
    to the current set of windows. The Id can be reused after closing this window.
    OpenModalWindow terminates when:
        the window has been closed (by means of CloseWindow), or
        the process has been terminated (by means of QuitIO), or
        another window with the same Id has been activated.
    If the interactive process is inactive, Open(Modal)Window does nothing. */


CloseWindow          :: !Id !(IOState .l .p) -> IOState .l .p

/*  If the interactive process is active, and the indicated window is not an inactive
    modal window, then CloseWindow closes the indicated window.
    In case the Id was generated by OpenWindow, it may now become reusable for new windows.
    If the interactive process is inactive, CloseWindow does nothing. */


OpenControls         :: !Id (cdef (PState .l .p)) !(PState .l .p) -> PState .l .p
                                                                   |  Controls cdef
```

```
CloseControls       :: !Id [Id]                 !(PState .l .p) -> PState .l .p

/*  OpenControls adds the given controls argument to the indicated window. The layout of the
    new control structure is the same as when the window would be opened given the argument
    controls appended to its current controls, and its current size.
    CloseControls removes the controls with the given Ids from the indicated window. The
    layout of the remaining control structure does not change. The window is not resized.
*/


ControlSizes        :: !(cdef     .ps) !(Optional Size) !(Optional Size) !(IOState .l .p)
                    -> (![Size],!IOState .l .p)    | Controls   cdef
ControlLSSizes      :: !(cdef .ls .ps) !(Optional Size) !(Optional Size) !(IOState .l .p)
                    -> (![Size],!IOState .l .p)    | ControlsLS cdef
/*  Control(LS)Sizes calculates the sizes of the given control definitions in the size as
    they would be opened as elements of a window.
    The Optional Size arguments are the prefered margins and item spaces respectively (see
    also the (Window/Control)Margin and (Window/Control)ItemSpace attributes). If None is
    specified, the default values for the margins and item spaces are used.    */


HideWindows         :: ![Id]   !(IOState .l .p) ->         IOState .l .p
ShowWindows         :: ![Id]   !(IOState .l .p) ->         IOState .l .p
GetHiddenWindows    ::         !(IOState .l .p) -> (![Id],!IOState .l .p)
GetShownWindows     ::         !(IOState .l .p) -> (![Id],!IOState .l .p)

/*  If the interactive process is active, (Hide/Show)Windows hides/shows the indicated
    windows.
    If the interactive process is inactive, (Hide/Show)Windows does nothing.
    Get(Hidden/Shown)Windows yields the list of currently visible/invisible windows.    */


ActivateWindow      :: !Id     !(IOState .l .p) -> IOState .l .p

/*  If the interactive process is active, ActivateWindow makes the window with the given
    Id the active window. In case the Id is unknown ActivateWindow has no effect.
    If the interactive process is inactive, ActivateWindow does nothing.    */


GetActiveWindow     ::         !(IOState .l .p) -> (!Optional Id, !IOState .l .p)

/*  GetActiveWindow returns the Id of the window that currently has the input focus of
    the interactive process. None is returned if there is no such window.    */


StackWindow         :: !Id !Id !(IOState .l .p) -> IOState .l .p

/*  If the process is active, StackWindow id1 id2 places the window with id1 behind the
    window with id2. If id2 indicates a modal window, then the window with id1 is placed
    behind the last modal window.
    If id1 or id2 is unknown, or id1 indicates a modal window, or the process  is inactive,
    StackWindow does nothing.    */


GetWindowStack      ::         !(IOState .l .p) -> (![(Id,WindowType)],!IOState .l .p)
GetWindowsStack     ::         !(IOState .l .p) -> (![Id],             !IOState .l .p)
GetDialogsStack     ::         !(IOState .l .p) -> (![Id],             !IOState .l .p)

/*  GetWindowStack returns the Ids and WindowTypes of all currently open windows (including
    the hidden windows), in the current stacking order (the head element Id indicates the
    top-most window).
    Get(Windows/Dialogs)Stack is equal to GetWindowStack, restricted to Window(LS)Type and
    DialogWindow(LS)Type.    */


GetDefaultMargins   ::         !(IOState .l .p) -> (!Size,         !IOState .l .p)
GetDefaultItemSpaces ::        !(IOState .l .p) -> (!Size,         !IOState .l .p)
GetWindowMargin     :: !Id     !(IOState .l .p) -> (!Optional Size,!IOState .l .p)
GetWindowItemSpace  :: !Id     !(IOState .l .p) -> (!Optional Size,!IOState .l .p)

/*  GetDefault(Margins/ItemSpaces) yield the default values for the margins and item spaces.
    GetWindowMargin yields the current margin of the indicated window if this window is a
        DialogWindow. In case the window is not a DialogWindow, or does not exist, None is
        yielded.
    GetWindowItemSpace yields the current item space of the indicated window. In case the
        window does not exist, None is returned.    */
```

```
EnableWindow          :: !Id     !(IOState .l .p) -> IOState .l .p
DisableWindow         :: !Id     !(IOState .l .p) -> IOState .l .p
EnableWindowMouse     :: !Id     !(IOState .l .p) -> IOState .l .p
DisableWindowMouse    :: !Id     !(IOState .l .p) -> IOState .l .p
EnableWindowKey       :: !Id     !(IOState .l .p) -> IOState .l .p
DisableWindowKey      :: !Id     !(IOState .l .p) -> IOState .l .p

/*  (En/Dis)ableWindow      (en/dis)ables input handling of mouse and keyboard.
    (En/Dis)ableWindowMouse (en/dis)ables input handling of mouse.
    (En/Dis)ableWindowKey   (en/dis)ables input handling of keyboard.
    For all involved components, disabling a component overrides the SelectStates of its
    elements, which become unselectable. Enabling a disabled component re-establishes the
    SelectStates of its elements.
    The functions have no effect in case of invalid Ids or DialogWindows.    */


GetWindowSelectState      :: !Id !(IOState .l .p) -> (!Optional SelectState,!IOState .l .p)
GetWindowMouseSelectState :: !Id !(IOState .l .p) -> (!Optional SelectState,!IOState .l .p)
GetWindowKeySelectState   :: !Id !(IOState .l .p) -> (!Optional SelectState,!IOState .l .p)

/*  GetWindowSelectState yields the current SelectState of the indicated window.
    GetWindow(Mouse/Key)SelectState
                         yields the current SelectState of the mouse/keyboard of the
                         indicated window.
    The functions return None in case of invalid Ids or DialogWindows.    */


DrawInWindow          :: !Id ![DrawFunction] !(IOState .l .p) -> IOState .l .p

//    Draw in the window (behind all Controls). Invalid Ids have no effect.


SetWindowPos          :: !Id !ItemPos !(IOState .l .p) -> IOState .l .p

/*  If the interactive process is active, SetWindowPos places the window to the indicated
    position. If the ItemPos argument refers to the Id of an unknown window (in case of
    LeftOf/RightTo/Above/Below), SetWindowPos has no effect. SetWindowPos also has no effect
    if the window is moved of the screen, if the Id is unknown, or if the interactive process
    is inactive.    */


GetWindowPos          :: !Id !(IOState .l .p) -> (!Optional ItemOffset,!IOState .l .p)

/*  GetWindowPos returns the current item offset position of the indicated window.
    The corresponding ItemPos is (LeftTop,offset). None is returned in case the window
    does not exist.    */


MoveWindowFrame       :: !Id Vector !(PState .l .p) -> PState .l .p

/*  MoveWindowFrame moves the orientation of the window over the given vector, and updates
    the window if necessary. The window frame is not moved outside the PictureDomain of the
    window.
    In case of unknown Id, or of DialogWindows, MoveWindowFrame has no effect.    */


GetWindowFrame         :: !Id              !(IOState .l .p) -> (!WindowFrame,!IOState .l .p)

/*  GetWindowFrame returns the currently visible frame of the window in terms of the
    PictureDomain. Note that in case of a DialogWindow, GetWindowFrame = ((0,0),size).
    In case the id is unknown, the WindowFrame result = ((0,0),(0,0)).    */


SetPictureDomain       :: !Id PictureDomain !( PState .l .p) ->  PState .l .p

/*  If the interactive process is active, SetPictureDomain resets the current PictureDomain
    of the indicated window, and updates the window if necessary. In case the new
    PictureDomain is smaller than the current WindowFrame, the window is resized to fit the
    new domain exactly. The window frame is moved only if it gets outside the new
    PictureDomain.
    In case of unknown Ids, of DialogWindows, or of inactive processes, SetPictureDomain has
    no effect.    */


GetPictureDomain       :: !Id !(IOState .l .p) -> (!Optional PictureDomain,!IOState .l .p)

/*  GetPictureDomain returns the current PictureDomain of the indicated window.
    None is returned in case the window is a DialogWindow(LS), or if the id is unknown.    */
```

```
SetWindowMinimumSize  :: !Id Size !(PState .l .p) -> PState .l .p

/*  If the interactive process is active, SetWindowMinimumSize sets the minimum size
    of the indicated window as given. The new minimum size is not allowed to exceed the
    dimensions of the current PictureDomain of the window and the screen. The window is
    resized and updated if the current size of either edge of the window is smaller than
    the new minimum size.
    In case of unknown Ids, of DialogWindows, of invalid value, or of inactive processes,
    SetWindowMinimumSize has no effect.    */

GetWindowMinimumSize  :: !Id     !(IOState .l .p) -> (!Optional Size,!IOState .l .p)

/*  GetWindowMinimumSize yields the current value of the minimum size of the indicated
    window. None is returned if the window is a DialogWindow(LS), or does not exist.    */


SetWindowSize        :: !Id Size !(PState .l .p) -> PState .l .p

/*  If the interactive process is active, SetWindowSize sets the size of the indicated window
    as given, and updates the window if necessary. The size is fit between the minimum size
    and the PictureDomain of the window, and the dimensions of the screen.
    In case of unknown Ids, of DialogWindows, or of inactive processes, SetWindowSize has no
    effect.    */


GetWindowSize        :: !Id       !(IOState .l .p) -> (!Size,!IOState .l .p)

/*  GetWindowSize yields the current size of the window frame of the indicated window.
    If the window does not exist, {0,0} is returned.    */


SetWindowTitle       :: !Id Title !(IOState .l .p) -> IOState .l .p
SetWindowOk          :: !Id Id    !(IOState .l .p) -> IOState .l .p
SetWindowCancel      :: !Id Id    !(IOState .l .p) -> IOState .l .p
SetWindowCursor      :: !Id CursorShape
                                  !(IOState .l .p) -> IOState .l .p
GetWindowTitle       :: !Id       !(IOState .l .p) -> (!Title,               !IOState .l .p)
GetWindowOk          :: !Id       !(IOState .l .p) -> (!Optional Id,         !IOState .l .p)
GetWindowCancel      :: !Id       !(IOState .l .p) -> (!Optional Id,         !IOState .l .p)
GetWindowCursor      :: !Id       !(IOState .l .p) -> (!Optional CursorShape,!IOState .l .p)

/*  These functions set the indicated attributes. Invalid Ids have no effect. If the
    indicated window does not have the corresponding attribute then it obtains the new
    attribute.    */
```

B.3.2 Controls

StdControlDef: the control device

```
definition module StdControlDef

import StdIOCommon
from   StdPicture    import    DrawFunction, Picture

::  RadioControl        ps = RadioControl    TextLine MarkState         [ControlAttribute      ps ]
::  RadioControlLS  ls  ps = RadioControlLS  TextLine MarkState         [ControlAttribute *(ls,ps)]
::  CheckControl        ps = CheckControl    TextLine MarkState         [ControlAttribute      ps ]
::  CheckControlLS  ls  ps = CheckControlLS  TextLine MarkState         [ControlAttribute *(ls,ps)]
::  PopUpControl        ps = PopUpControl    [PopUpItem      ps ] Index [ControlAttribute      ps ]
::  PopUpControlLS  ls  ps = PopUpControlLS  [PopUpItem *(ls,ps)] Index [ControlAttribute *(ls,ps)]
::  SliderControl       ps = SliderControl   Direction Length SliderState
                                                (SliderAction      ps ) [ControlAttribute      ps ]
::  SliderControlLS ls  ps = SliderControlLS Direction Length SliderState
                                                (SliderAction *(ls,ps)) [ControlAttribute *(ls,ps)]
::  TextControl         ps = TextControl     TextLine                   [ControlAttribute      ps ]
::  TextControlLS   ls  ps = TextControlLS   TextLine                   [ControlAttribute *(ls,ps)]
::  EditControl         ps = EditControl     TextLine Width NrLines     [ControlAttribute      ps ]
::  EditControlLS   ls  ps = EditControlLS   TextLine Width NrLines     [ControlAttribute *(ls,ps)]
::  ButtonControl       ps = ButtonControl   TextLine                   [ControlAttribute      ps ]
::  ButtonControlLS ls  ps = ButtonControlLS TextLine                   [ControlAttribute *(ls,ps)]
::  CustomButtonControl ps = CustomButtonControl   Size ControlLook     [ControlAttribute      ps ]
::  CustomButtonControlLS ls ps
                           = CustomButtonControlLS Size ControlLook     [ControlAttribute *(ls,ps)]
::  CustomControl          ps = CustomControl      Size ControlLook     [ControlAttribute      ps ]
::  CustomControlLS     ls ps = CustomControlLS    Size ControlLook     [ControlAttribute *(ls,ps)]
::  CompoundControl   c    ps = CompoundControl   (c    ps)             [ControlAttribute      ps ]
::  CompoundControlLS c ls ps = CompoundControlLS (c ls ps)             [ControlAttribute *(ls,ps)]

::  TextLine        :== String
::  NrLines         :== Int
::  Width           :== Int
::  Length          :== Int
::  PopUpItem    ps :== (TextLine, IOFunction ps)
::  ControlLook     :== SelectState -> Size -> [DrawFunction]
::  SliderAction ps :== SliderMove  -> ps   -> ps
::  SliderState
    =   {   sliderMin   :: !Int
        ,   sliderMax   :: !Int
        ,   sliderThumb :: !Int
        }
::  SliderMove
    =   SliderIncSmall
    |   SliderDecSmall
    |   SliderIncLarge
    |   SliderDecLarge
    |   SliderThumb Int
::  Direction
    =   Horizontal
    |   Vertical

::  ControlAttribute ps                                        // Default:
    =   ControlId           Id                              // no id
    |   ControlPos          ItemPos                         // (RightTo previous,zero)
    |   ControlSize         Size                            // system derived/overruled
    |   ControlMinimumSize  Size                            // zero
    |   ControlResize       ControlResizeFunction           // no resize
    |   ControlSelectState  SelectState                     // control Able
    |   ControlHide                                         // initially visible
    |   ControlFunction     (IOFunction                ps) // I
    |   ControlModsFunction (ModsIOFunction            ps) // ControlFunction
    |   ControlMouse        SelectState (MouseFunction ps) // no mouse input/overruled
    |   ControlKey          SelectState (KeyFunction   ps) // no keyboard input/overruled
    //  For CompoundControls only
    |   ControlLook         ControlLook                     // control is transparant
    |   ControlItemSpace    Size                            // system dependent
    |   ControlMargin       Size                            // system dependent

::  ControlResizeFunction
    :== Size ->                                             // current control size
        Size ->                                             // old     window  size
        Size ->                                             // new     window  size
        Size                                                // new     control size
```

StdControlType: control types

```
definition module StdControlType

import StdOverloaded


::  ControlType


instance ==       ControlType
instance toString ControlType

RadioControlType          :: ControlType
RadioControlLSType        :: ControlType
CheckControlType          :: ControlType
CheckControlLSType        :: ControlType
PopUpControlType          :: ControlType
PopUpControlLSType        :: ControlType
SliderControlType         :: ControlType
SliderControlLSType       :: ControlType
TextControlType           :: ControlType
TextControlLSType         :: ControlType
EditControlType           :: ControlType
EditControlLSType         :: ControlType
ButtonControlType         :: ControlType
ButtonControlLSType       :: ControlType
CustomButtonControlType   :: ControlType
```

```
CustomButtonControlLSType :: ControlType
CustomControlType         :: ControlType
CustomControlLSType       :: ControlType
CompoundControlType       :: ControlType
CompoundControlLSType     :: ControlType
```

StdControl: control handling

```
definition module StdControl

/*  Module StdControl specifies all functions on controls.
    Changing controls in a window  requires a *WState.
    Reading the status of controls requires a  WState.    */

import StdIOCommon
import StdControlDef
import StdControlType
from   iostate    import IOState
from   StdPicture import DrawFunction, Picture


::  WState


GetWindow            :: !Id !(IOState .l .p) -> (!Optional WState, !IOState .l .p)
/*  GetWindow returns a read-only WState for the indicated window.
    In case the indicated window does not exist None is returned.    */

SetWindow            :: !Id ![IOFunction *WState] !(IOState .l .p) -> IOState .l .p
/*  Apply the control changing functions to the current state of the indicated window.
    In case the indicated window does not exist nothing happens.    */


//  Functions applied to unknown Ids have no effect.

ShowControls         :: ![Id]                            !*WState -> *WState
HideControls         :: ![Id]                            !*WState -> *WState
EnableControls       :: ![Id]                            !*WState -> *WState
DisableControls      :: ![Id]                            !*WState -> *WState
MarkCheckControls    :: ![Id]                            !*WState -> *WState
UnmarkCheckControls  :: ![Id]                            !*WState -> *WState
SelectRadioControl   :: !Id                              !*WState -> *WState
UnselectRadioControl :: !Id                              !*WState -> *WState
SelectPopUpItem      :: !Id !Index                       !*WState -> *WState
SetControlTexts      :: ![(Id,String)]                   !*WState -> *WState
SetControlLooks      :: ![(Id,Bool,ControlLook)]         !*WState -> *WState
SetSliderStates      :: ![(Id,SliderState->SliderState)] !*WState -> *WState
SetSliderThumbs      :: ![(Id,Int)]                      !*WState -> *WState

/*  Functions that change the state of controls.
    -   SetControlTexts sets the text of the indicated (Radio/Check/Text/Edit/Button)Control(LS)s.
    -   SetControlLooks applied to a CompoundControl turns it into a non-transparant CompoundControl.
        Setting the ControlLook only redraws the indicated controls if the corresponding
        Boolean is True.
    -   SetSliderStates applies the function to the current SliderState of the indicated
        SliderControl and redraws the settings if necessary.
    -   SetSliderThumbs sets the new thumb value of the indicated SliderControl and redraws
        the settings if necessary.    */


DrawInControl        :: !Id ![DrawFunction]              !*WState -> *WState

/*  Draw in a (Custom(Button)/Compound)Control. If the CompoundControl is transparant
    then this operation has no effect.    */


GetControlTypes      ::     !WState -> [(ControlType,Optional Id)]
GetCompoundTypes     :: !Id !WState -> [(ControlType,Optional Id)]

/*  GetControlTypes  yields the list of ControlTypes of the component controls of this
    window.
    GetCompoundTypes yields the list of ControlTypes of the component controls of this
    CompoundControl. For both functions (One id) is yielded if the component control has a
    (ControlId id) attribute, and None otherwise. Component controls are not collected
    recursively through CompoundControls.
    If the indicated CompoundControl is not a CompoundControl, then [] is yielded.    */
```

```
GetControlLayouts       :: ![Id] !WState -> [(Bool,(Optional ItemPos,ItemOffset))]
                                                                    // (None,{0,0})
GetControlSizes         :: ![Id] !WState -> [(Bool,Size)]                 // {0,0}
GetControlSelectStates  :: ![Id] !WState -> [(Bool,SelectState)]          // Able
GetControlShowStates    :: ![Id] !WState -> [(Bool,Bool)]                 // False
GetControlTexts         :: ![Id] !WState -> [(Bool,Optional String)]      // None
GetControlNrLines       :: ![Id] !WState -> [(Bool,Optional NrLines)]     // None
GetControlLooks         :: ![Id] !WState -> [(Bool,Optional ControlLook)] // None
GetControlMinimumSizes  :: ![Id] !WState -> [(Bool,Optional Size)]        // None
GetControlResizes       :: ![Id] !WState -> [(Bool,Optional ControlResizeFunction)] // None
GetPopUpItems           :: ![Id] !WState -> [(Bool,Optional [TextLine])]  // None
GetSelectedPopUpItems   :: ![Id] !WState -> [(Bool,Optional Index)]       // None
GetSelectedRadioControls::       !WState -> [Id]                          // []
GetSelectedCheckControls::       !WState -> [Id]                          // []
GetRadioControlMarks    :: ![Id] !WState -> [(Bool,Optional Bool)]        // None
GetCheckControlMarks    :: ![Id] !WState -> [(Bool,Optional Bool)]        // None
GetSliderDirections     :: ![Id] !WState -> [(Bool,Optional Direction)]   // None
GetSliderStates         :: ![Id] !WState -> [(Bool,Optional SliderState)] // None
GetControlItemSpaces    :: ![Id] !WState -> [(Bool,Optional Size)]        // None
GetControlMargins       :: ![Id] !WState -> [(Bool,Optional Size)]        // None

/*  Functions that return the current state of controls.
    The result list is of equal length as the argument Id list. Each result list element
    corresponds in order with the argument Id list. Of each element the first Boolean result
    is False in case of invalid Ids (if so dummy values are returned - see comment).

    -   GetControlLayouts     yields (One ControlPos) if the indicated control had a
                              ControlPos attribute and None otherwise. The ItemOffset offset
                              is the exact current location of the indicated control
                              (LeftTop,offset).
    -   GetControlShowStates  yields True if the indicated control is visible, and False
                              otherwise.
    -   GetControlTexts       yields (One text) of the indicated (Radio/Check/Text/Edit/
                              Button) Control(LS).
                              If the control is not such a control, then None is yielded.
    -   GetControlNrLines     yields (One nrlines) of the indicated EditControl(LS). If the
                              control is not such a control, then None is yielded.
    -   GetControlLooks       yields the ControlLook of the indicated (Custom/CustomButton/
                              Compound) Control(LS).
                              If the control is not such a control, or it is a transparant
                              CompoundControl(LS), then None is yielded.
    -   GetPopUpItems         yields the TextLines of the indicated PopUpControl. If the
                              control is not such a control, then None is yielded.
    -   GetSelectedPopUpItems yields the Index of the indicated PopUpControl. If the control
                              is not such a control, then None is yielded.
    -   GetRadioControlMarks  yields (One True) if the indicated RadioControl is selected,
                              and (One False) otherwise.
                              If the control is not such a control, then None is yielded.
    -   GetCheckControlMarks  yields (One True) if the indicated CheckControl is checked, and
                              (One False) otherwise.
                              If the control is not such a control, then None is yielded.
    -   GetSliderDirections   yields (One Direction) of the indicated SliderControl(LS).
                              If the control is not such a control, then None is yielded.
    -   GetSliderStates       yields (One SliderState) of the indicated SliderControl(LS).
                              If the control is not such a control, then None is yielded.
    -   GetControlItemSpaces  yields (One Size) of the indicated CompoundControl(LS).
                              If the control is not such a control, then None is yielded.
    -   GetControlMargins     yields (One Size) of the indicated CompoundControl(LS).
                              If the control is not such a control, then None is yielded.

    Important: controls with no ControlId attribute, or illegal ids, can not be found in the
               WState!    */
```

B.3.3 Menus

StdMenuDef: the menu device

```
definition module StdMenuDef

import StdIOCommon

::  Menu          m    ps = Menu       Title (m    ps) [MenuAttribute      ps ]
::  MenuLS        m ls ps = MenuLS ls  Title (m ls ps) [MenuAttribute *(ls,ps)]

::  SubMenu       m    ps = SubMenu    Title (m    ps) [MenuAttribute      ps ]
::  SubMenuLS     m ls ps = SubMenuLS  Title (m ls ps) [MenuAttribute *(ls,ps)]
::  MenuItem          ps = MenuItem   Title           [MenuAttribute      ps ]
::  MenuItemLS     ls ps = MenuItemLS Title           [MenuAttribute *(ls,ps)]
::  MenuSeparator     ps = MenuSeparator
::  MenuSeparatorLS ls ps = MenuSeparatorLS

::  MenuAttribute ps                         // Default:
    =   MenuId           Id                  // no Id
    |   MenuSelectState  SelectState         // menu(item) Able
    |   MenuAltKey       Index               // no AltKey
    //  Attributes ignored by (sub)menus:
    |   MenuShortKey     Char                // no ShortKey
    |   MenuMarkState    MarkState           // NoMark
    |   MenuFunction     (IOFunction     ps) // I
    |   MenuModsFunction (ModsIOFunction ps) // MenuFunction
```

StdMenuType: menu types

```
definition module StdMenuType

::  MenuType


instance ==       MenuType
instance toString MenuType

MenuType   :: MenuType
MenuLSType :: MenuType
```

StdMenu: menu handling

```
definition module StdMenu

import StdMenuDef
import StdMenuType
from   iostate import IOState, PState, Files


//  Operations on unknown Ids are ignored.

class Menus mdef
where
    OpenMenu :: !Int !(mdef (PState .l .p)) !(IOState .l .p) -> IOState .l .p


instance Menus (Menu   m)    | MenuElements   m
instance Menus (MenuLS m ls) | MenuElementsLS m

/*  Open the given menu definition for this interactive process behind the menu indicated by
    the integer index.
    In case a menu with the same Id is already open then nothing happens.
    In case the menu does not have an Id, it will obtain an Id which is fresh with respect to
    the current set of menus. The Id can be reused after closing this menu.
    The index of a menu starts from one for the first present menu.
    If the index is negative or zero, then the new menu is added before the first menu.
    If the index exceeds the number of menus, then the new menu is added behind the last
    menu. */


CloseMenu           :: !Id                        !(IOState .l .p) -> IOState .l .p

/*    Close the given menu (and all of its elements including submenus). */


OpenMenuItems       :: !Id !Int (m (PState .l .p)) !(IOState .l .p) -> IOState .l .p
                                                                   |   MenuElements m
/*  Add menu elements to the indicated (sub)menu.
    OpenMenuItems adds menu elements after the item with the specified index. The index of a
    menu element starts from one for the first menu element in the (sub)menu. If the index is
    negative or zero, then the new menu elements are added before the first menu element of
    the (sub)menu. If the index exceeds the number of menu elements in the (sub)menu, then
    the new menu elements are added behind the last menu element of the (sub)menu.
    No menu elements are added if the indicated (sub)menu does not exist.    */


CloseMenuItems      :: ![Id]                      !(IOState .l .p) -> IOState .l .p
CloseMenuIndexItems :: !Id ![Int]                 !(IOState .l .p) -> IOState .l .p
```

```
/*  Close menu elements.
    CloseMenuItems:
        closes menu elements by their Id.
    CloseMenuIndexItems:
        closes menu elements of the indicated (sub)menu by their indices. Analogous to
        OpenMenuItems, indices range from one to the number of menu elements in a (sub)menu.
        Invalid indices (less than one or larger than the number of menu elements of the
        (sub)menu) are ignored.
    Closing a submenu closes all of its elements including submenus. */


EnableMenuSystem    ::                              !(IOState .l .p) -> IOState .l .p
DisableMenuSystem   ::                              !(IOState .l .p) -> IOState .l .p

/*  Enable/disable the menu system of this process. When the menu system is enabled the
    previously selectable menus and menu items will become selectable again. Enable/disable
    operations on the menu( element)s of a disabled menu system take effect when the menu
    system is re-enabled.
    EnableMenuSystem has no effect in case this process has a modal window. */


EnableMenus         :: ![Id]                        !(IOState .l .p) -> IOState .l .p
DisableMenus        :: ![Id]                        !(IOState .l .p) -> IOState .l .p

/*  Enable/disable individual menus.
    Disabling a menu overrides the SelectStates of its elements, which become unselectable.
    Enabling a disabled menu re-establishes the SelectStates of its elements.
    Enable/disable operations on the elements of a disabled menu take effect when the
    menu is re-enabled.    */


GetMenuSelectState  :: !Id       !(IOState .l .p) -> (!Optional SelectState,!IOState .l .p)

/*  GetMenuSelectState yields the current SelectState of the indicated menu.
    In case the menu does not exist, None is returned.    */


GetMenus            ::           !(IOState .l .p) -> (![(Id,MenuType)],!IOState .l .p)

/*  GetMenus yields the Ids and MenuTypes of the current set of menus of this interactive
    process.    */


GetMenuPos          :: !Id       !(IOState .l .p) -> (!Optional Index,!IOState .l .p)

/*  GetMenuPos yields the index position of the indicated menu in the current list of menus.
    In case the menu does not exist, None is returned.    */


GetMenuAltKey       :: !Id       !(IOState .l .p) -> (!Optional Index,!IOState .l .p)

/*  GetMenuAltKey yields the MenuAltKey index of the indicated menu. In case the menu does
    not exist, or does not have this attribute set, None is returned.    */


SetMenuTitle        :: !Id !Title !(IOState .l .p) -> IOState .l .p
GetMenuTitle        :: !Id       !(IOState .l .p) -> (!Optional Title, !IOState .l .p)

/*  SetMenuTitle sets the title of the indicated menu.
    In case the menu does not exist, nothing happens.
    GetMenuTitle retrieves the current title of the indicated menu.
    In case the menu does not exist, None is returned.    */
```

StdMenuItemType: menu item types

```
definition module StdMenuItemType

::  MenuItemType


instance ==        MenuItemType
instance toString MenuItemType

SubMenuType         :: MenuItemType
SubMenuLSType       :: MenuItemType
MenuItemType        :: MenuItemType
```

```
MenuItemLSType      :: MenuItemType
MenuSeparatorType   :: MenuItemType
MenuSeparatorLSType :: MenuItemType
```

StdMenuItem: menu item handling

```
definition module StdMenuItem

/*  Module StdMenuItem specifies all functions on menu items.
    Changing menu items in a menu    requires a *MState.
    Reading the status of menu items requires a  MState.    */

import StdMenuDef
import StdMenuItemType
from   iostate import IOState, PState, Files


::  MState


GetMenu                :: !Id !(IOState .l .p) -> (!Optional MState, !IOState .l .p)

/*  GetMenu returns a read-only MState for the indicated menu.
    In case the indicated menu does not exist None is returned. */


SetMenu                :: !Id ![IOFunction *MState] !(IOState .l .p) -> IOState .l .p

/*  Apply the menu item changing functions to the current state of the indicated menu.
    Invalid Ids are ignored.        */


EnableMenuItems        :: ![Id]         !*MState -> *MState
DisableMenuItems       :: ![Id]         !*MState -> *MState
MarkMenuItems          :: ![Id]         !*MState -> *MState
UnmarkMenuItems        :: ![Id]         !*MState -> *MState
SetMenuItemTitles      :: ![(Id,Title)] !*MState -> *MState

/*  Enable/disable, mark/unmark, and set titles of menu items (including SubMenuItems).    */


GetMenuItemTypes       ::       !MState -> [(MenuItemType,Optional Id)]
GetSubMenuItemTypes    :: !Id   !MState -> [(MenuItemType,Optional Id)]

/*  GetMenuItemTypes    yields the list of MenuItemTypes of all menu items of this menu.
    GetSubMenuItemTypes yields the list of MenuItemTypes of all menu items of this SubMenu.
    For both functions (One id) is yielded if the item has a (MenuId id) attribute, and None
    otherwise. Ids are not collected recursively through SubMenus.    */

                                                                // Values in case Bool==False
GetMenuItemSelectStates:: ![Id] !MState -> [(Bool,SelectState)] // Able
GetMenuItemMarks       :: ![Id] !MState -> [(Bool,Bool)]        // False
GetMenuItemTitles      :: ![Id] !MState -> [(Bool,String)]      // ""
GetMenuItemShortKey    :: ![Id] !MState -> [(Bool,Char)]        // toChar 0
GetMenuItemAltKey      :: ![Id] !MState -> [(Bool,Index)]       // 0

/*  Functions that return the current state of menu items.
    The  result list is of equal length as the argument Id list.
    Each result list element corresponds in order with the argument Id list.
    Of each element the first Boolean result is False in case of invalid ids
    (if so dummy values are returned - see comment).
*/
```

StdQuit: quit handling

```
definition module StdQuit

from   StdIOCommon import IOFunction
from   iostate     import IOState, PState, Files


::  MenuQuit      ps = MenuQuit   (    ps ->(Bool,    ps))
::  MenuQuitLS ls ps = MenuQuitLS ((ls,ps)->(Bool,(ls,ps)))

/*  MenuQuit and MenuQuitLS define the abstract event handlers that respond to quit events.
    In case of a quit event, the function argument is applied to the current process state.
    If the Boolean result is True, then the interactive process is quitted by the I/O system.
    If the Boolean result is False,then the interactive process is not quitted.    */


//  Opening and closing quit definition (as menu elements):

instance MenuElements   MenuQuit
instance MenuElementsLS MenuQuitLS


//  Enabling, disabling quit event handling:

EnableQuit  :: !(IOState .l .p) -> IOState .l .p
DisableQuit :: !(IOState .l .p) -> IOState .l .p
```

StdClipboardDef: clipboard definition

```
definition module StdClipboardDef

from StdIOCommon import IOFunction


::  MenuCut        ps = MenuCut     (IOFunction ps)
::  MenuCopy       ps = MenuCopy    (IOFunction ps)
::  MenuPaste      ps = MenuPaste   (IOFunction ps)

::  MenuCutLS   ls ps = MenuCutLS   (IOFunction *(ls,ps))
::  MenuCopyLS  ls ps = MenuCopyLS  (IOFunction *(ls,ps))
::  MenuPasteLS ls ps = MenuPasteLS (IOFunction *(ls,ps))
```

StdClipboard: clipboard handling

```
definition module StdClipboard

import StdOverloaded
import StdClipboardDef
from   iostate import IOState, PState, Files


//  Opening and closing clipboard definitions (as menu elements):

instance MenuElements   MenuCut
instance MenuElements   MenuCopy
instance MenuElements   MenuPaste

instance MenuElementsLS MenuCutLS
instance MenuElementsLS MenuCopyLS
instance MenuElementsLS MenuPasteLS


//  Reading and writing the value of the selection to the clipboard:

class Clipboard data
where
    SetClipboard    :: data           !(IOState .l .p) -> IOState .l .p
    GetClipboard    :: !ClipboardType !(IOState .l .p) -> (!Optional data,!IOState .l .p)
    toClipboardType :: !data                           -> ClipboardType

instance Clipboard {#Char}

/*  SetClipboard    sets the given value of the current selection. In case of a cut/copy
                    event this value will be stored (and only then evaluated!) in the
                    clipboard.
    GetClipboard    gets the current content of the clipboard (which may be different from
                    the value 'set' by SetClipboard). The ClipboardType argument indicates
                    the type of the requested clipboard content. If the clipboard is empty or
                    does not contain a data item of the requested type, None is returned.
    toClipboardType yields the ClipboardType of an instance of the Clipboard class.     */

ClearClipboard      :: !(IOState .l .p) ->            IOState .l .p

/*  ClearClipboard removes the selection value set by SetClipboard. */


ClipboardHasChanged :: !(IOState .l .p) -> (!Bool, !IOState .l .p)
```

```
/*  ClipboardHasChanged holds if the current content of the clipboard is different from the
    last access to the clipboard.    */


//  The type tag of clipboard data items:

::  ClipboardType

instance == ClipboardType
```

B.3.4 StdPicture: drawing in windows

```
definition module StdPicture

from   picture import Picture
import StdIOCommon, StdFont, StdPictureDef


/*  Attribute functions.
*/
GetPicture              ::                     !*Picture -> (!Picture,!*Picture)
GetPictureAttributes    ::                     ! Picture -> [PictureAttribute]
SetPictureAttributes    :: ![PictureAttribute] !*Picture -> *Picture

//  Pen position attributes:
SetPenPos               :: !Point              !*Picture -> *Picture
GetPenPos               ::                     ! Picture -> Point

class MovePenPos figure :: !figure             !*Picture -> *Picture
//  Move the pen position as much as when drawing the figure.
instance MovePenPos Point
instance MovePenPos Vector
instance MovePenPos Curve

//  PenSize attributes:
SetPenSize              :: !Int                !*Picture -> *Picture
GetPenSize              ::                     ! Picture -> Int

SetDefaultPenSize       ::                     !*Picture -> *Picture
//  SetDefaultPenSize = SetPenSize 1

//  Colour attributes:
SetPenColour            :: !Colour             !*Picture -> *Picture
GetPenColour            ::                     ! Picture -> Colour

SetDefaultPenColour     ::                     !*Picture -> *Picture
//  SetDefaultPenColour = SetPenColour BlackColour

//  Font attributes:
SetPenFont              :: !Font               !*Picture -> *Picture
GetPenFont              ::                     ! Picture -> Font

SetDefaultPenFont       ::                     !*Picture -> *Picture


/*  Drawing functions.
    These functions are divided into the following classes:
    Drawables: Draw   'line-oriented' figures at the current  pen position.
               DrawAt 'line-oriented' figures at the argument pen position.
    Fillables: Fill   'area-oriented' figures at the current  pen position.
               FillAt 'area-oriented' figures at the argument pen position.
    Clips:     apply a list of drawing functions within a clipping area(s).
               Clip   takes the base point of the area to be the current  pen position.
               ClipAt takes the base point of the area to be the argument pen position.
*/
class Drawables figure
where
    Draw   ::        !figure              !*Picture -> *Picture
    DrawAt :: !Point !figure              !*Picture -> *Picture

class Fillables figure
where
    Fill   ::        !figure              !*Picture -> *Picture
    FillAt :: !Point !figure              !*Picture -> *Picture

class Clips area
where
```

```
    Clip   ::        !area [DrawFunction] !*Picture -> *Picture
    ClipAt :: !Point !area [DrawFunction] !*Picture -> *Picture

::  DrawFunction
    :== *Picture -> *Picture


/*  (Draw/Hilite/Xor)Picture applies the given drawing functions to the given picture in
    left-to-right order. When drawing is done, all picture attributes are set to the
    attribute values of the original picture.
    DrawPicture   is simply the seq of the drawing functions.
    HilitePicture takes care that the drawing functions operate in the appropriate platform
                  'hilite' mode. This mode is usually used for drawing selections.
    XorPicture    takes care that the drawing functions operate in the appropriate platform
                  'xor' mode. When drawing in xor mode it is guaranteed that drawing the same
                  figure twice does not change the picture.
*/
DrawPicture      :: ![DrawFunction] !*Picture -> *Picture
HilitePicture    :: ![DrawFunction] !*Picture -> *Picture
XorPicture       :: ![DrawFunction] !*Picture -> *Picture


/*  (Draw/Hilite/Xor)seqPicture applies the given drawing functions to the given picture in
    left-to-right order.
    Picture attributes are not reset after drawing.
    DrawseqPicture   is simply the seq of the drawing functions.
    HiliteseqPicture takes care that the drawing functions operate in the appropriate
                     platform 'hilite' mode. This mode is usually used for drawing
                     selections.
    XorseqPicture    takes care that the drawing functions operate in the appropriate
                     platform 'xor' mode. When drawing in xor mode it is guaranteed that
                     drawing the same figure twice does not change the picture.
*/
DrawseqPicture   :: ![DrawFunction] !*Picture -> *Picture
HiliteseqPicture :: ![DrawFunction] !*Picture -> *Picture
XorseqPicture    :: ![DrawFunction] !*Picture -> *Picture


/*  Drawing within in a clipping area:
*/
instance Clips Box                     // Clip within a box
instance Clips Rectangle               // Clip within a rectangle (ClipAt _ r p = Clip r p)
instance Clips Polygon                 // Clip within a polygon
instance Clips [figure] | Clips figure // Clip within the union of figures


/*  Point drawing operations:
    DrawPoint   plots a point at the current  pen position p and moves to p+{1,0}
    DrawPointAt plots a point at the argument pen position, but retains the pen position.
*/
DrawPoint        ::                  !*Picture -> *Picture
DrawPointAt      :: !Point           !*Picture -> *Picture


/*  Point connecting drawing operations:
    DrawToPoint   draws a line between the current pen position to the argument point which
                  becomes the new pen position.
    DrawToPointAt draws a line between the two argument points, but retains the pen position.
*/
DrawToPoint      :: !Point           !*Picture -> *Picture
DrawToPointAt    :: !Point !Point    !*Picture -> *Picture


/*  Text drawing operations:
    Picture is an instance of the Fonts class (see StdFont).

    Text is drawn from the baseline at the y coordinate of the pen.
    Draw     text:  Draws the text starting at the current pen position.
                    The new pen position is directly after the drawn text including spacing.
    DrawAt p text:  Draws the text starting at p.
*/
instance Fonts     Picture

instance Drawables Char
instance Drawables {#Char}


/*  Vector drawing operations:
    Draw     v:     Draws a line from the current pen position pen to pen+v.
    DrawAt p v:     Draws a line from p to p+v.
*/
instance Drawables Vector


/*  Oval drawing operations:
    An Oval o is a transformed unit circle
    with    horizontal radius rx    o.oval_rx
            vertical   radius ry    o.oval_ry
    Let (x,y) be a point on the unit circle:
        then (x`,y`) = (x*rx,y*ry) is a point on o.
    Let (x,y) be a point on o:
        then (x`,y`) = (x/rx,y/ry) is a point on the unit circle.
    Draw     o: Draws an oval with the current pen position being the center of the oval.
    DrawAt p o: Draws an oval with p being the center of the oval.
    Fill     o: Fills an oval with the current pen position being the center of the oval.
    FillAt p o: Fills an oval with p being the center of the oval.
    None of these functions change the pen position.
*/
instance Drawables Oval
instance Fillables Oval


/*  Curve drawing operations:
    A Curve c is a slice of an oval o
    with    start angle a    c.curve_from
            end   angle b    c.curve_to
            direction   d    c.curve_clockwise
    The angles are taken in radians (counter-clockwise).
    If d holds then the drawing direction is clockwise, otherwise drawing occurs counter-
    clockwise.
    Draw     c: Draws a curve with the starting angle a at the current pen position.
                The pen position ends at ending angle b.
    DrawAt p c: Draws a curve with the starting angle a at p.
    Fill     c: Fills the figure obtained by connecting the endpoints of the drawn curve
                (Draw c) with the center of the curve oval.
                The pen position ends at ending angle b.
    FillAt p c: Fills the figure obtained by connecting the endpoints of the drawn curve
                (DrawAt p c) with the center of the curve oval.
*/
instance Drawables Curve
instance Fillables Curve


/*  Box drawing operations:
    A Box b is a horizontally oriented rectangle
    with    width  w        b.box_w
            height h        b.box_h
    In case w==0 (h==0),   the Box collapses to a vertical (horizontal) vector.
    In case w==0 and h==0, the Box collapses to a point.
    Draw     b: Draws a box with left-top corner at the current pen position p and
                right_bottom corner at p+(w,h).
    DrawAt p b: Draws a box with left-top corner at p and right-bottom corner at p+(w,h).
    Fill     b: Fills a box with left-top corner at the current pen position p and
                right_bottom corner at p+(w,h).
    FillAt p b: Fills a box with left-top corner at p and right-bottom corner at p+(w,h).
    None of these functions change the pen position.
*/
instance Drawables Box
instance Fillables Box


/*  Rectangle drawing operations:
    A Rectangle r is a horizontally oriented rectangle
    with    width  w   abs (r.corner1.x-r.corner2.x)
            height h   abs (r.corner1.y-r.corner2.y)
    In case w==0 (h==0),   the Rectangle collapses to a vertical (horizontal) vector.
    In case w==0 and h==0, the Rectangle collapses to a point.
    Draw     r: Draws a rectangle with diagonal corners r.corner1 and r.corner2.
    DrawAt p r= Draw r
    Fill     r: Fills a rectangle with diagonal corners r.corner1 and r.corner2.
    FillAt p r= Fill r
    None of these functions change the pen position.
*/
instance Drawables Rectangle
instance Fillables Rectangle
```

```
/*  Polygon drawing operations:
    A Polygon p is a figure
    with    shape    p.polygon_shape
    A polygon p at a point base is drawn as follows:

        DrawPicture [SetPenPos base:map Draw shape]++[DrawToPoint base]

    None of these functions change the pen position.
*/
instance Drawables Polygon
instance Fillables Polygon
```

StdPictureDef: data type definitions

```
definition module StdPictureDef

import StdIOCommon, StdFont

//  The predefined figures that can be drawn:
::  Box // A box is a horizontally oriented rectangle with a width and height
    =   {   box_w           :: !Int       // The width  of the box
        ,   box_h           :: !Int       // The height of the box
        }
::  Oval                                  // An oval is a transformed unit circle
    =   {   oval_rx         :: !Int       // The horizontal radius
        ,   oval_ry         :: !Int       // The vertical   radius
        }
::  Curve                                 // A curve is a slice of an oval
    =   {   curve_oval      :: !Oval      // The source oval
        ,   curve_from      :: !Real      // Starting angle (in radians)
        ,   curve_to        :: !Real      // Ending   angle (in radians)
        ,   curve_clockwise :: !Bool      // Direction: True iff clockwise
        }
::  Polygon                               // A polygon is outlined by a list of vectors
    =   {   polygon_shape   :: ![Vector]  // The shape of the polygon
        }


//  The picture attributes:
::  PictureAttribute                      // Default:
    =   PicturePenSize   Int              // 1
    |   PicturePenPos    Point            // zero
    |   PicturePenColour Colour           // Black
    |   PicturePenFont   Font             // DefaultFont
::  Colour
    =   RGBColour RGB
    |   Black    | White
    |   DarkGrey | MediumGrey | LightGrey // 75%, 50%, and 25% Black
    |   Red      | Green      | Blue
    |   Cyan     | Magenta    | Yellow
::  RGB
    =   {   r :: Int                      // The contribution of red
        ,   g :: Int                      // The contribution of green
        ,   b :: Int                      // The contribution of blue
        }
BlackRGB :== {r=MinRGB,g=MinRGB,b=MinRGB}
WhiteRGB :== {r=MaxRGB,g=MaxRGB,b=MaxRGB}
MinRGB   :== 0
MaxRGB   :== 255

PI       :== 3.1415926535898
```

B.3.5 StdFont: writing in windows

```
definition module StdFont

import StdFontDef
from   font import Font

class Fonts environment
where
    OpenFont        ::           !FontDef  !*environment -> (!(!Bool,!Font),!*environment)
    OpenDefaultFont ::                     !*environment -> (!Font,         !*environment)
    OpenDialogFont  ::                     !*environment -> (!Font,         !*environment)
    GetFontNames    ::                    !*environment -> (![FontName],   !*environment)
    GetFontStyles   ::          !FontName !*environment -> (![FontStyle],  !*environment)
    GetFontSizes    :: !Int !Int !FontName !*environment -> (![FontSize],   !*environment)

instance Fonts World

/*  OpenFont        creates the font as specified by the name, the stylistic
                    variations and size. In case there are no FontStyles ([]), the font
                    is selected without stylistic variations (i.e. in plain style).
                    The Boolean result is True in case this font is available and needn't
                    be scaled. In case the font is not available, the default font is
                    chosen in the indicated style and size.
    OpenDefaultFont returns the default font that applications can use. This
                    font is set in the initial Picture of a new window.
    OpenDialogFont  returns the font used by the system to present information
                    in dialog windows (e.g. TextControls), menus and so on.

    GetFontNames    returns the FontNames  of all available fonts.
    GetFontStyles   returns the FontStyles of all available styles of a particular FontName.
    GetFontSizes    returns all FontSizes of a particular FontName that are available
                    without scaling. The sizes inspected are inclusive between the two
                    Integer arguments. (The Integer arguments need not be ordered; negative
                    values are set to zero.)
    In case the requested font is unavailable, the styles or sizes of the default font
    are returned. */


GetFontDef          :: !Font -> FontDef

/*  GetFontDef returns the name, stylistic variations and size of the
    argument Font. */


GetFontCharWidth    :: ! Char    !Font -> Int
GetFontCharWidths   :: ![Char]   !Font -> [Int]
GetFontStringWidth  :: ! String  !Font -> Int
GetFontStringWidths :: ![String] !Font -> [Int]

/*  GetFontCharWidth(s) (GetFontStringWidth(s)) return the width(s) in terms of pixels
    of given character(s) (string(s)) for a particular Font. */


GetFontMetrics      :: !Font -> FontMetrics

/*  GetFontMetrics yields the metrics of a given Font in terms of pixels.
    FontMetrics is a record which defines the metrics of a font:
        - fAscent   is the height of the top most character measured from the base
        - fDescent  is the height of the bottom most character measured from the base
        - fLeading  is the vertical distance between two lines of the same font
        - fMaxWidth is the width of the widest character including spacing
    The full height of a line is fAscent+fDescent+fLeading. */
```

StdFontDef: data type definitions

```
definition module StdFontDef

::  FontDef
    =   {   fName      :: !FontName
        ,   fStyles    :: ![FontStyle]
        ,   fSize      :: !FontSize
        }
::  FontMetrics
    =   {    fAscent   :: !Int
        ,    fDescent  :: !Int
        ,    fLeading  :: !Int
        ,    fMaxWidth :: !Int
        }
::  FontName  :== String
::  FontStyle :== String
::  FontSize  :== Int
```

B.3.6 Timers

StdTimerDef: the timer device

```
definition module StdTimerDef

import StdIOCommon

::  Timer      ps =   Timer      TimerInterval [TimerAttribute      ps ]
::  TimerLS ls ps =   TimerLS ls TimerInterval [TimerAttribute *(ls,ps)]
::  TimerInterval :== Int

::  TimerAttribute ps                      // Default:
    =   TimerId       Id                   // no Id
    |   TimerSelect   SelectState          // timer Able
    |   TimerFunction (TimerFunction ps)   // \_ x->x
::  TimerFunction  ps :== NrOfIntervals->ps->ps
::  NrOfIntervals     :== Int
```

StdTimerType: timer types

```
definition module StdTimerType

import StdOverloaded


::  TimerType


instance ==       TimerType
instance toString TimerType

TimerType   :: TimerType
TimerLSType :: TimerType
```

StdTimer: timer handling

```
definition module StdTimer

import StdTimerDef
import StdTimerType
from   iostate import IOState, PState, Files


TicksPerSecond :== 60


class Timers tdef
where
    OpenTimer       :: !(tdef (PState .l .p)) !(IOState .l .p) -> IOState .l .p

instance OpenTimer Timer
instance OpenTimer (TimerLS ls)

/*  Open a new timer with or without a local state. This function has no effect in case
    the interactive process already contains a timer with the same Id. Negative
    TimerIntervals are set to zero.
    In case the timer does not have an Id, it will obtain an Id which is fresh with
    respect to the current set of timers. The Id can be reused after closing this timer.
*/


CloseTimer          :: !Id !(IOState .l .p) -> IOState .l .p

/*  Close the timer with the indicated Id. */


GetTimers           ::     !(IOState .l .p) -> ([(Id,TimerType)],     !IOState .l .p)

/*  GetTimers returns the Ids and TimerTypes of all currently open timers.    */


EnableTimer         :: !Id !(IOState .l .p) -> IOState .l .p
DisableTimer        :: !Id !(IOState .l .p) -> IOState .l .p
GetTimerSelectState :: !Id !(IOState .l .p) -> (!Optional SelectState,!IOState .l .p)

/*  (En/Dis)ableTimer (en/dis)able the indicated timer.
    GetTimerSelectState yields the SelectState of the indicated timer. If the timer does not
    exist, then None is yielded.    */
```

```
SetTimerInterval    :: !Id !TimerInterval !(IOState .l .p) -> IOState .l .p
GetTimerInterval    :: !Id !(IOState .l .p) -> (!Optional TimerInterval,!IOState .l .p)

/*  Set the TimerInterval of the indicated timer. Negative TimerIntervals are set to zero.
    GetTimerInterval yields the TimerInterval of the indicated timer. If the timer does not
    exist, then None is yielded.    */
```

StdTime: time related operations

```
definition module StdTime

from iostate import IOState

::  PointOfTime
    =   {   hours   :: !Int // hours       (0-23)
        ,   minutes :: !Int // minutes     (0-59)
        ,   seconds :: !Int // seconds     (0-59)
        }
::  Date
    =   {   year    :: !Int // year
        ,   month   :: !Int // month       (1-12)
        ,   day     :: !Int // day         (1-31)
        ,   dayNr   :: !Int // day of week (1-7, Sunday=1, Saturday=7)
        }


Wait                :: !Int .x -> .x

/*  Wait n x suspends the evaluation of x modally for n ticks.
    If n<=0, then x is evaluated immediately.    */

class Time environment
where
    GetBlinkInterval :: !*environment -> (!Int,         !*environment)
    GetCurrentTime   :: !*environment -> (!PointOfTime, !*environment)
    GetCurrentDate   :: !*environment -> (!Date,        !*environment)

instance Time World
instance Time (IOState .l .p)

/*  GetBlinkInterval returns the time interval in ticks that should elapse between blinks of
    e.g. a cursor. This interval may be changed by the user while the interactive process is
    running!
    GetCurrentTime returns the current PointOfTime.
    GetCurrentDate returns the current Date.
*/
```

B.3.7 Receivers

StdReceiverDef: the receiver device

```
definition module StdReceiverDef

import StdIOCommon


::  Receiver       m      ps
=   Receiver       (ReceiverFunction  m           ps)  [ReceiverAttribute      ps ]
::  ReceiverLS  ls m      ps
=   ReceiverLS  ls (ReceiverFunction  m      *(ls,ps)) [ReceiverAttribute *(ls,ps)]

::  Receiver2      m resp ps
=   Receiver2      (Receiver2Function m resp      ps)  [ReceiverAttribute      ps ]
::  Receiver2LS ls m resp ps
=   Receiver2LS ls (Receiver2Function m resp *(ls,ps)) [ReceiverAttribute *(ls,ps)]

::  ReceiverFunction  m      ps :== m -> ps -> ps
::  Receiver2Function m resp ps :== m -> ps -> (resp,ps)

::  ReceiverAttribute ps               // Default:
    =    ReceiverSelect  SelectState  // receiver Able
```

StdReceiverType: receiver types

```
definition module StdReceiverType

import StdOverloaded


::  ReceiverType


instance ==       ReceiverType
instance toString ReceiverType

ReceiverType    :: ReceiverType
ReceiverLSType  :: ReceiverType
Receiver2Type   :: ReceiverType
Receiver2LSType :: ReceiverType
```

StdReceiver: receiver handling

```
definition module StdReceiver

import StdReceiverDef
import StdReceiverType
from   iostate import PState, IOState, Files

//  The identification of a Receiver:
::  RId  mess
::  R2Id mess resp

class RIds environment
where
    OpenRId  :: !*environment -> (!(!RId  mess,     !*RId  mess),     !*environment)
    // Create a one-way receiver id
    OpenR2Id :: !*environment -> (!(!R2Id mess resp,!*R2Id mess resp),!*environment)
    // Create a two-way receiver id

instance RIds World
instance RIds (IOState .l .p)

instance == (RId  mess)
instance == (R2Id mess resp)

RIdtoId  :: (RId  mess)      -> Id
R2IdtoId :: (R2Id mess resp) -> Id


//  Operations on the ReceiverDevice.

//  Open one-way receivers:

class Receivers rdef
where
    OpenReceiver   :: !*(RId m) !(rdef m (PState .l .p)) !(IOState .l .p) -> IOState .l .p
    ReopenReceiver :: ! (RId m) !(rdef m (PState .l .p)) !(IOState .l .p) -> IOState .l .p

instance Receivers  Receiver
instance Receivers (ReceiverLS ls)

/*  OpenReceiver   opens the given receiver and binds the unique RId to this receiver. The
                   shareable RId has to be used to send messages to this receiver.
    ReopenReceiver first closes the receiver with the given RId, and then opens a new
                   receiver with the given receiver definition. In case the receiver was not
                   found, nothing happens. The new receiver can again be sent messages to via
                   the given RId.
*/


//  Open two-way receivers:

class Receiver2s rdef
where
    OpenReceiver2  ::!*(R2Id m r) !(rdef m r (PState .l .p)) !(IOState .l .p)-> IOState .l .p
    ReopenReceiver2::! (R2Id m r) !(rdef m r (PState .l .p)) !(IOState .l .p)-> IOState .l .p

instance Receiver2s  Receiver2
instance Receiver2s (Receiver2LS ls)

/*  OpenReceiver2   opens the given receiver and binds the unique R2Id to this receiver. The
                    shareable R2Id has to be used to send messages to this receiver.
    ReopenReceiver2 first closes the receiver with the given R2Id, and then opens a new
                    receiver with the given receiver definition. In case the receiver was not
                    found, nothing happens. The new receiver can again be sent messages to
                    via the given R2Id.
*/


CloseReceiver          :: !Id   !(IOState .l .p) -> IOState .l .p

/*  Close the indicated one-way or two-way receiver. Invalid Ids have no effect.    */


GetReceivers           ::       !(IOState .l .p) -> (![(Id,ReceiverType)], !IOState .l .p)

/*  GetReceivers returns the Ids and ReceiverTypes of all currently open one-way and two-way receivers
    of this interactive process.
*/


EnableReceivers        :: ![Id] !(IOState .l .p) ->                         IOState .l .p
DisableReceivers       :: ![Id] !(IOState .l .p) ->                         IOState .l .p
GetReceiverSelectState :: ! Id  !(IOState .l .p) -> (!Optional SelectState,!IOState .l .p)

/*  (En/Dis)ableReceivers (en/dis)able the indicated one-way and two-way receivers.
    Note that this implies that in case of synchronous message passing messages can fail
    (see the comments of SyncSend and SyncSend2 below). Invalid Ids have no effect.
    GetReceiverSelectState yields the current SelectState of the indicated receiver.
    In case the receiver does not exist, None is returned.
*/


//  Inter-process communication:

//  Continuation report for process communication:
::  SendReport
    =   SendOk
    |   SendUnknownProcess
    |   SendUnknownReceiver
    |   SendUnableReceiver
    |   SendDeadlock

::  SendCont    ps :== SendReport             -> ps -> ps
::  Send2Cont r ps :== (SendReport,Optional r) -> ps -> ps

ASyncSend :: !(RId mess) mess !(Optional (SendCont (PState .l .p))) !(PState .l .p)
          -> PState .l .p
/*  ASyncSend posts a message to the receiver indicated by the argument RId. In case the
    indicated receiver belongs to this process, the message is simply buffered.
    ASyncSend is asynchronous: the message will at some point be received by the indicated
    receiver/process.
    The Optional SendCont-inuation is always applied as follows:
    -   SendOk: No exceptional situation has occurred. The message has been sent.
                Note that even though the message has been sent, it cannot be
                guaranteed that the message will actually be handled by the indicated
                receiver because it might become closed, forever disabled, or flooded with
                synchronous messages.
    -   SendUnknownProcess:
                The indicated interactive process does not exist, therefore the
                receiver also does not exist.
    -   SendUnknownReceiver:
                The indicated receiver does not exist, although the interactive
                process that created it does exist.
    -   SendUnableReceiver:
                Does not occur: the message is always buffered, regardless whether the
                indicated receiver is Able or Unable. Note that in case the receiver never
                becomes Able, the message will not be handled.
    -   SendDeadlock:
                Does not occur.
    By default, the SendCont-inuation is \_ ps->ps.
*/

SyncSend :: !(RId mess) mess !(Optional (SendCont (PState .l .p))) !(PState .l .p)
         -> PState .l .p
/*  SyncSend posts a message to the receiver indicated by the argument RId. In case the
```

```
    indicated receiver belongs to the current process, the corresponding ReceiverFunction
    is applied directly to the message argument and current process state.
    SyncSend is synchronous: this interactive process blocks evaluation until the indicated
    receiver has received the message.
    The Optional SendCont-inuation is always applied as follows:
    -   SendOk: No exceptional situation has occurred. The message has been sent and handled
                by the indicated receiver.
    -   SendUnknownProcess:
                The indicated interactive process does not exist, therefore the receiver
                also does not exist.
    -   SendUnknownReceiver:
                The indicated receiver does not exist, although the interactive process that
                created it does exist.
    -   SendUnableReceiver:
                The addressee exists, but its ReceiverSelect attribute is Unable. Message
                passing is halted. The message is not sent.
    -   SendDeadlock:
                The addressee is involved in a synchronous, cyclic communication with
                the current process. Blocking the current process would result in a deadlock
                situation. Message passing is halted to circumvent the deadlock.
                The message is not sent.
    By default, the SendCont-inuation is \_ ps->ps.
*/

SyncSend2::!(R2Id mess resp) mess !(Optional (Send2Cont resp (PState .l .p))) !(PState .l .p)
        -> PState .l .p
/*  SyncSend2 posts a message to the receiver indicated by the argument R2Id. In case the
    indicated receiver belongs to the current process, the corresponding Receiver2Function
    is applied directly to the message argument and current process state.
    SyncSend2 is synchronous: this interactive process blocks until the indicated
    receiver has received the message.
    The Optional Send2Cont-inuation is always applied as follows:
    -   SendOk: No exceptional situation has occurred. The message has been sent and handled
                by the indicated receiver.
    -   SendUnknownProcess:
                The indicated interactive process does not exist, therefore the receiver
                also does not exist.
    -   SendUnknownReceiver:
                The indicated receiver does not exist, although the interactive process that
                created it does exist.
    -   SendUnableReceiver:
                The addressee exists, but its ReceiverSelect attribute is Unable. Message
                passing is halted. The message is not sent.
    -   SendDeadlock:
                The addressee is involved in a synchronous, cyclic communication with
                the current process. Blocking the current process would result in a deadlock
                situation. Message passing is halted to circumvent the deadlock.
                The message is not sent.
    By default, the Send2Cont-inuation is \_ ps->ps. If the SendReport==SendOk,
    (One response) of the receiver is passed as an argument of the Send2Cont-inuation. In all
    other cases, this value is None.
*/
```

B.3.8 StdFileSelect: selecting files

```
definition module StdFileSelect

from StdString   import String
from StdPState   import PState, IOState, Files
from StdIOCommon import Optional, One, None

/*  With the functions defined in this module standard file selector dialogs can be opened,
    which provide a user-friendly way to select input or output files. The lay-out of these
    dialogs depends on the (version of the) operating system.    */

class FileSelect environment
where
    SelectInputFile ::                 !*environment -> (!Optional String,!*environment)
    SelectOutputFile:: !String !String !*environment -> (!Optional String,!*environment)

instance FileSelect World
instance FileSelect (PState .l .p)

/*  SelectInputFile  opens a dialog window in which the user can traverse the file system
                     to select an existing file. If a file has been selected, the String
                     result contains the complete pathname of the selected file. If the
                     user has not selected a file, None is returned.
    SelectOutputFile opens a dialog window in which the user can specify the name of a file
                     to write to in a certain directory. The first argument is the prompt
                     of the dialog (default: "Save As:"), the second argument is the default
                     filename. When a file with the indicated name already exists in the
                     indicated directory a confirm dialog will be opened. The String result
                     contains the complete pathname of the selected file in case a filename
                     is confirmed. If no filename has been confirmed, None is returned.    */
```

B.3.9 StdIOCommon: common definitions

```
definition module StdIOCommon

import StdOverloaded
from   StdString import String
from   id        import Id, toId, ==, toString


/*  General type constructors for composing context-dependent data structures.    */

::  :+:      t1 t2   c = (:+:) infixr 9 (t1   c) (t2   c)
::  :~:      t1 t2 l c = (:~:) infixr 9 (t1 l c) (t2 l c)

::  ListNoLS t       c = ListNoLS [t   c]
::  NilNoLS          c = NilNoLS
::  ListLS   t     l c = ListLS   [t l c]
::  NilLS          l c = NilLS
::  LS       t     l c = LS   (t   c)
::  NoLS     t       c = E.l : {introLS :: .l,  introDef :: t .l       c}
::  ExtendLS t     l c = E.l1: {extendLS:: .l1, extendDef:: t *(.l1,l) c}
::  ChangeLS t     l c = E.l1: {changeLS:: .l1, changeDef:: t .l1      c}


::  Index       :== Int
::  Title       :== String



::  Size        =   {w ::!Int,h ::!Int}

instance        ==   Size   // Equality on Size
instance        zero Size   // Zero ({w=0,h=0})



::  Vector      =   {vx::!Int,vy::!Int}

instance        ==   Vector // Equality on Vector
instance        +    Vector // Add arg1 to arg2        (v1+v2={vx=v1.x+v2.vx,vy=v1.vy+v2.vy})
instance        -    Vector // Subtract arg2 from arg1 (v1-v2={vx=v1.x-v2.vx,vy=v1.vy-v2.vy})
instance        zero Vector // Zero (unit element for addition, {vx=0,vy=0})



::  SelectState =   Able | Unable
::  MarkState   =   Mark | NoMark

Enabled         :: !SelectState  -> Bool      // select == Able
Marked          :: !MarkState    -> Bool      // mark   == Mark
MarkSwitch      :: !MarkState    -> MarkState // Mark -> NoMark; NoMark -> Mark

instance        == SelectState                // Equality on SelectState
instance        == MarkState                  // Equality on MarkState


::  KeyboardState
    =   {   keyCode     :: !Char
        ,   keyState    :: !KeyState
        ,   keyModifiers:: !Modifiers
        }
::  KeyState
    =   KeyUp
    |   KeyDown
    |   KeyStillDown

instance        == KeyState // Equality on KeyState
```

```
::  MouseState
    =   {   mousePos       :: !Point
        ,   mouseButton    :: !ButtonState
        ,   mouseModifiers :: !Modifiers
        }
::  ButtonState
    =   ButtonUp
    |   ButtonDown
    |   ButtonDoubleDown
    |   ButtonTripleDown
    |   ButtonStillDown

instance        == ButtonState // Equality on ButtonState


/*  Modifiers indicates the meta keys that have been pressed (True) or not (False).    */

::  Modifiers
    =   {   shiftDown   :: !Bool
        ,   optionDown  :: !Bool
        ,   commandDown :: !Bool
        ,   controlDown :: !Bool
        }


/*  Frequently occuring data types:                        */

::  Point
    =   {   x       :: !Int
        ,   y       :: !Int
        }
::  Rectangle
    =   {   corner1 :: !Point
        ,   corner2 :: !Point
        }

instance        ==   Point       // Equality on Point
instance        +    Point       // Add arg1 to arg2        (p1+p2={x=p1.x+p2.x,y=p1.y+p2.y})
instance        -    Point       // Subtract arg2 from arg1 (p1-p2={x=p1.x-p2.x,y=p1.y-p2.y})
instance        zero Point       // Zero (unit element for addition, {x=0,y=0})

instance        ==   Rectangle   // Equality on Rectangle

RectangleSize   :: !Rectangle -> Size // The size of the Rectangle

::  PictureDomain :== Rectangle
::  UpdateArea    :== [Rectangle]


/*  The layout language used for windows and controls.    */

::  ItemPos
    :== (   ItemLoc
        ,   ItemOffset
        )
::  ItemLoc
    //   Relative to corner:
    =    LeftTop
    |    RightTop
    |    LeftBottom
    |    RightBottom
    //   Relative in next line:
    |    Left
    |    Center
    |    Right
    //   Relative to other item:
    |    LeftOf   Id
    |    RightTo  Id
    |    Above    Id
    |    Below    Id
    //   Relative to previous item:
    |    LeftOfPrev
    |    RightToPrev
    |    AbovePrev
    |    BelowPrev
::  ItemOffset
    :==  Vector
```

```
instance        == ItemLoc // Equality on ItemLoc


/*  Attributes for interactive processes.    */

::  IOAttribute ps
    =   IOActivate   (IOFunction ps)
    |   IODeactivate (IOFunction ps)
    |   IOHelp       (IOFunction ps)


/*  Frequently used function types.            */

::  IOFunction     ps :==                 ps -> ps
::  ModsIOFunction ps :== Modifiers     -> ps -> ps
::  UpdateFunction ps :== UpdateArea    -> ps -> ps
::  MouseFunction  ps :== MouseState    -> ps -> ps
::  KeyFunction    ps :== KeyboardState -> ps -> ps


/*  Optional type.                           */

::  Optional x
    =   One x
    |   None

hasOption :: !(Optional .x) -> Bool
getOption :: !(Optional .x) -> .x
```

B.3.10 StdPState: access operations on the P State

```
definition module StdPState

from iostate     import PState, IOState, Files
from StdIOCommon import IOFunction
from StdFunc     import St

//  Coercing PState component operations to PState operations.

appListPIO  :: ![.IOFunction (IOState .l .p)] !(PState .l .p) -> PState .l .p
appListPFs  :: ![.IOFunction Files]           !(PState .l .p) -> PState .l .p
appListPLoc :: ![.IOFunction .l]              !(PState .l .p) -> PState .l .p
appListPPub :: ![.IOFunction .p]              !(PState .l .p) -> PState .l .p

appPIO      :: !.(IOFunction (IOState .l .p)) !(PState .l .p) -> PState .l .p
appPFs      :: !.(IOFunction Files)           !(PState .l .p) -> PState .l .p
appPLoc     :: !.(IOFunction .l)              !(PState .l .p) -> PState .l .p
appPPub     :: !.(IOFunction .p)              !(PState .l .p) -> PState .l .p

//  Accessing PState component operations.

accListPIO  :: ![.St (IOState .l .p) .x]      !(PState .l .p) -> (![.x], !PState .l .p)
accListPFs  :: ![.St Files           .x]      !(PState .l .p) -> (![.x], !PState .l .p)
accListPLoc :: ![.St .l              .x]      !(PState .l .p) -> (![.x], !PState .l .p)
accListPPub :: ![.St .p              .x]      !(PState .l .p) -> (![.x], !PState .l .p)

accPIO      :: !.(St (IOState .l .p) .x)      !(PState .l .p) -> (!.x,   !PState .l .p)
accPFs      :: !.(St Files           .x)      !(PState .l .p) -> (!.x,   !PState .l .p)
accPLoc     :: !.(St .l              .x)      !(PState .l .p) -> (!.x,   !PState .l .p)
accPPub     :: !.(St .p              .x)      !(PState .l .p) -> (!.x,   !PState .l .p)
```

B.3.11 StdIOState: global operations on the IO State

```
definition module StdIOState

import StdFont, StdWindowDef
from   iostate import PState, IOState, Files

//  IOState is an environment instance of the class Fonts (see StdFont).

instance Fonts (IOState .l .p)


//  Emit the alert sound.

Beep          ::              !(IOState .l .p) -> IOState .l .p
```

```
/*  If the interactive process is active, Beep emits a sound alert.
    If the interactive process is inactive, Beep does nothing. */


//  Operations on the global cursor:

SetCursor     :: !CursorShape !(IOState .l .p) -> IOState .l .p
/*  Set the shape of the cursor globally. This shape overrules the local cursor shapes of
    windows. */

ResetCursor   ::              !(IOState .l .p) -> IOState .l .p
/*  Undoes the effect of SetCursor. */

ObscureCursor ::              !(IOState .l .p) -> IOState .l .p
/*  ObscureCursor hides the cursor until the mouse is moved. */


//  Operations on the DoubleDownDistance:

SetDoubleDownDistance :: !Int !(IOState .l .p) -> IOState .l .p
/*  Set the maximum distance the mouse is allowed to move to generate a
    ButtonDouble(Triple)Down button state. Negative values are set to zero. */


//  Operations on the attributes of an interaction:

SetIOActivate   :: !(IOFunction (PState .l .p)) !(IOState .l .p) -> IOState .l .p
SetIODeactivate :: !(IOFunction (PState .l .p)) !(IOState .l .p) -> IOState .l .p
SetIOHelp       :: !(IOFunction (PState .l .p)) !(IOState .l .p) -> IOState .l .p
/*  Set the IOActivate, IODeactivate, IOHelp attribute of the interactive process. */
```

B.3.12 StdSystem: platform dependent settings

```
definition module StdSystem

import StdIOCommon

//  Keyboard constants.

UpKey        :== '\036'        // Arrow up
DownKey      :== '\037'        // Arrow down
LeftKey      :== '\034'        // Arrow left
RightKey     :== '\035'        // Arrow right
PgUpKey      :== '\013'        // Page up
PgDownKey    :== '\014'        // Page down
BeginKey     :== '\001'        // Begin of text
EndKey       :== '\004'        // End of text
BackSpKey    :== '\010'        // Backspace
DelKey       :== '\177'        // Delete
TabKey       :== '\011'        // Tab
ReturnKey    :== '\015'        // Return
EnterKey     :== '\003'        // Enter
EscapeKey    :== '\033'        // Escape
HelpKey      :== '\005'        // Help

//  File constants.

DirSeparator :== ':'           // Separator between folder- and filenames in a pathname

//  Constants to check which of the Modifiers is down.

ShiftOnly    :== {shiftDown=True, optionDown=False,commandDown=False,controlDown=False}
OptionOnly   :== {shiftDown=False,optionDown=True, commandDown=False,controlDown=False}
CommandOnly  :== {shiftDown=False,optionDown=False,commandDown=True, controlDown=False}
ControlOnly  :== {shiftDown=False,optionDown=False,commandDown=False,controlDown=True }


/*  The functions HomePath and ApplicationPath prefix the filename given to them with the
    full pathnames of the 'home' and 'application' directory. These functions have been added
    for compatibility with the Sun version of the Clean system. In the 'home' directory
    settings-files (containing preferences, options etc.) should be stored. In the
    application' directory (i.e. the directory in which the application resides) files that
    are used read-only by the application (such as help files) should be stored. On the
    Macintosh these functions just return the filename given to them, which means that the
    file will be stored in the same folder as the application.    */

HomePath        :: !String -> String
```

```
ApplicationPath :: !String -> String


/*  Screen resolution functions.
    h(mm/inch) convert millimeters/inches into pixels, horizontally.
    v(mm/inch) convert millimeters/inches into pixels, vertically.    */

mmperinch       :== 25.4

hmm             :: !Real -> Int
vmm             :: !Real -> Int
hinch           :: !Real -> Int
vinch           :: !Real -> Int

/*  Maximum ranges of window PictureDomains:
    MaxScrollWindowSize yields the range at which scrollbars are inactive.
    MaxFixedWindowSize  yields the range at which the window does not change into a
                        ScrollWindow.
*/

MaxScrollWindowSize :: Size
MaxFixedWindowSize  :: Size
```

B.4 Operations for parallel evaluation

B.4.1 StdProcId: operations for load distribution on ProcIds

C

Annotated CLEAN Bibliography

Below follows an annotated bibliography for people who want to know more about CONCURRENT CLEAN, its underlying concepts and its implementation. Many of these papers are available from our ftp site

General papers on CONCURRENT CLEAN

- Rinus Plasmeijer and Marko van Eekelen (1993). *Functional Programming and Parallel Graph Rewriting*. Addison Wesley, ISBN 0-201-41663-8.

Basic book on CLEAN. Introduction in functional programming using Miranda; CLEAN (version 0.8); Underlying model of computation (lambda-calculus, term rewriting systems, graph rewriting systems); Type systems; Strictness analysis; Implementation techniques using CLEAN as intermediate language; Abstract machines; Code generation for both sequential and parallel architectures.

- Rinus Plasmeijer (1994). 'The CONCURRENT CLEAN Development System'. University of Nijmegen.

Manual on the use of CLEAN's programming environment on the Mac. This information can also be obtained by printing out the help file from the Mac distribution.

- Eric Nöcker, Sjaak Smetsers, Marko van Eekelen, Rinus Plasmeijer (1991). 'CONCURRENT CLEAN'. In Aarts, E.H.L., J. van Leeuwen, M. Rem (Eds.), *Proceedings of the Conference on Parallel Architectures and Languages Europe (PARLE'91)*, Vol II, Eindhoven, The Netherlands, LNCS 505, Springer Verlag, June 1991, 202-219.

Gives a short overview of the features of CONCURRENT CLEAN (version 0.7) as well as of its implementation.

- Tom Brus, Marko van Eekelen, Maarten van Leer, Rinus Plasmeijer (1987). 'CLEAN - A Language for Functional Graph Rewriting'. *Proc. of the Third International Conference on Functional Programming Languages and Computer Architecture (FPCA '87)*, Portland, Oregon, USA, LNCS 274, Springer Verlag, 364-384.

First paper on CLEAN.

Papers on the underlying computational model being used

- Henk Barendregt, Marko van Eekelen, John Glauert, Richard Kennaway, Rinus Plasmeijer, Ronan Sleep (1987). 'Term Graph Rewriting'. *Proceedings of Parallel Architectures and Languages Europe (PARLE)*, part II, Eindhoven, The Netherlands. LNCS 259, Springer Verlag, 141-158.

Basic paper on Term Graph Rewriting, the computational model CLEAN is based upon.

- Ronan Sleep, Rinus Plasmeijer and Marko van Eekelen (1993). *Term Graph Rewriting - Theory and Practice*. John Wiley & Sons.

Collection of theoretical papers by various authors on properties of Term Graph Rewriting systems.

- Yoshihito Toyama, Sjaak Smetsers, Marko van Eekelen and Rinus Plasmeijer (1993). 'The functional strategy and transitive term rewriting systems'. In: *Term Graph Rewriting*, ed. Sleep, Plasmeijer and van Eekelen, John Wiley.

- Marko van Eekelen, Rinus Plasmeijer, Sjaak Smetsers (1991). 'Parallel Graph Rewriting on Loosely Coupled Machine Architectures'. In Kaplan, S. and M. Okada (Eds.) *Proc. of the 2nd Int. Worksh. on Conditional and Typed Rewriting Systems (CTRS'90)*, 1990. Montreal, Canada, LNCS 516, Springer Verlag, 354-370.

Explains parallel Graph Rewriting and the concept of lazy copying.

- Erik Barendsen and Sjaak Smetsers (1993). 'Extending Graph Rewriting with Copying'. In: *Proc. of the Seminar on Graph Transformations in Computer Science*, ed. B. Courcelle, H. Ehrig, G. Rozenberg and H.J. Schneider, Dagstuhl, Wadern, Springer-Verlag, Berlin, LNCS 776, Springer Verlag, pp 51-70.

Formal semantics of (lazy) copying.

- Steffen van Bakel, Simon Brock and Sjaak Smetsers (1992). 'Partial type assignment in left-linear applicative term rewriting systems'. In: *Proc. of the CAAP'92*, ed. J.C. Raoult, Rennes, France, LNCS 581, Springer Verlag, pp. 300-321.

Formal treatment of the "classical" type system of CONCURRENT CLEAN.

- Erik Barendsen and Sjaak Smetsers (1993). 'Conventional and Uniqueness Typing in Graph Rewrite Systems (extended abstract)'. In: *Proc. of the 13th Conference on the Foundations of Software Technology & Theoretical Computer Science*, ed. R.K. Shyamasundar, Bombay, India, LNCS 761, Springer Verlag, pp. 41-51.

Formal treatment of CLEAN's Uniqueness Type System used to guarantee single-threaded use of objects.

Papers on applications written in CLEAN

- Walter de Hoon, Luc Rutten and Marko van Eekelen (1994). 'Implementing a Functional Spreadsheet in CLEAN'. *Journal of Functional Programming*, 5, 3, pp. 383-414.

About a spreadsheet written in CLEAN. As spreadsheet language also a CLEAN-like functional language is chosen which is being interpreted by a theorem prover. One can do symbolic evaluation to verify properties of the spreadsheet.

Papers on advanced I/O

- Peter Achten, John van Groningen and Rinus Plasmeijer (1992). 'High-level specification of I/O in functional languages'. *In: Proc. of the Glasgow workshop on Functional programming*, ed. J. Launchbury and P. Sansom, Ayr, Scotland, Springer-Verlag, Workshops in Computing, pp. 1-17.

Introduction in CLEAN's Event I/O.

- Peter Achten and Rinus Plasmeijer (1995). 'The Ins and Outs of CONCURRENT CLEAN I/O'. *Journal of Functional Programming*, 5, 1, pp. 81-110.

Explains the concepts behind CLEAN's Event I/O and how they can be used to define interactive window-based applications on a high-level of abstraction.

- Peter Achten and Rinus Plasmeijer (1994). 'A framework for Deterministically Interleaved Interactive Programs in the Functional Programming Language CLEAN'. In: *Proc. of the CSN'94, Computing Science*, to appear.

Explains how one can create several interleaved executing interactive CLEAN processes inside one interactive pure functional CLEAN application which can communicate via a shared state as well as via asynchronous message passing.

Papers on the CLEAN to PABC compiler

- Eric Nöcker and Sjaak Smetsers (1993). 'Partially strict non-recursive data types'. *Journal of Functional Programming,* 3, 2, pp. 191-215.

Introduces partially strict data structures as available in CONCURRENT CLEAN and explains why and how they improve efficiency.

- Eric Nöcker (1993). 'Strictness analysis using abstract reduction'. In: *Proc. of the 6th Conference on Functional Programming Languages and Computer Architectures*, ed. Arvind, Copenhagen, ACM Press, pp. 255-265.

Explains the efficient and powerful strictness analysis method incorporated in CLEAN.

Papers on the abstract machine level

- Pieter Koopman, Marko van Eekelen, Eric Nöcker, Sjaak Smetsers, Rinus Plasmeijer (1990). 'The ABC-machine: A Sequential Stack-based Abstract Machine For Graph Rewriting'. *Proc of the Sec. Intern. Workshop on Implementation of Functional Languages on Parallel Architectures,* pp. 297-321, Technical Report no. 90-16, October 1990, University of Nijmegen.

Explains the sequential version of the PABC-machine and gives some information about the compilation of CLEAN to (abstract) ABC-machine code.

Papers on code generation

- Sjaak Smetsers, Eric Nöcker, John van Groningen, Rinus Plasmeijer (1991). 'Generating Efficient Code for Lazy Functional Languages'. In Hughes, J. (Ed.), *Proc. of the Fifth International Conference on Functional Programming Languages and Computer Architecture (FPCA '91),* USA, LNCS 523, Springer Verlag, 592-618.

Explains some optimisations that are used for the generation of efficient machine code.

- John van Groningen , Eric Nöcker and Sjaak Smetsers (1991) 'Efficient heap management in the concrete ABC machine' in Proc. of Third International Workshop on Implementation of Functional Languages on Parallel Architectures, University of Southampton, UK 1991,Technical Report Series CSTR91-07.

Explains the heap management (garbage collection) techniques used for the implementation of CONCURRENT CLEAN on concrete machines.

- Marko Kesseler (1991). 'Implementing the ABC machine on transputers'. In: *Proc. of the 3rd International Workshop on Implementation of Functional Languages on Parallel Architectures*, ed. H. Glaser and P. Hartel, Southampton, University of Southampton, Technical Report 91-07, pp. 147-192.
- Richard Goldsmith, Dave McBurney and Ronan Sleep (1993). 'Parallel execution of CONCURRENT CLEAN on ZAPP'. In: *Term Graph Rewriting*, ed. Sleep, Plasmeijer and van Eekelen, John Wiley.

The papers explain different ways to achieve a parallel implementation of CONCURRENT CLEAN on a MIMD machine with distributed memory.

- Marko Kesseler (1994). 'Reducing Graph Copying Costs - Time to Wrap it up'. In: *Proc. of the First International Symposium on Parallel Symbolic Computation, PASCO '94,* ed. Hoon Hong, Hagenberg/Linz, Austria, World Scientific, Lecture notes Series on Computing, 5, 5, pp. 244-254.

Explains how to generate efficient code for a multi-processor Transputer system.

D

Bibliography

Barendregt, H.P. (1984). The Lambda-Calculus, its Syntax and Semantics. North–Holland.

Bird, R.S. and P. Wadler (1988). *Introduction to Functional Programming.* Prentice Hall.

Harper, R., D. MacQueen and R. Milner (1986). ‘Standard ML’. Edinburgh University, Internal report ECS-LFCS-86-2.

Hindley R. (1969). The principle type scheme of an object in combinatory logic. *Trans. of the American Math. Soc.*, 146, 29-60.

Hudak, P. , S. Peyton Jones, Ph. Wadler, B. Boutel, J. Fairbairn, J. Fasel, K. Hammond, J. Hughes, Th. Johnsson, D. Kieburtz, R. Nikhil, W. Partain and J. Peterson (1992). ‘Report on the programming language Haskell’. *ACM SigPlan notices,* 27, 5, pp. 1-164.

Jones, M.P. (1993). *Gofer - Gofer 2.21 release notes*. Yale University.

Milner, R.A. (1978). ‘Theory of type polymorphism in programming’. *Journal of Computer and System Sciences,* 17, 3, 348-375.

Mycroft A. (1984). Polymorphic type schemes and recursive definitions. In *Proc. International Conference on Programming,* Toulouse (Paul M. and Robinet B., eds.), LNCS 167, Springer Verlag, 217–239.

Turner, D.A. (1985). ‘Miranda: a non-strict functional language with polymorphic types’. In: *Proc. of the Conference on Functional Programming Languages and Computer Architecture,* ed. J.P. Jouannaud, Nancy, France. LNCS 201, Springer Verlag, 1-16.

E

Index

Emboldened terms indicate where a term has been defined in the text. A term starting with an upper-case character generally refers to an identifier in the syntactic description or to a predefined function or operator in the library.