concurrent

Clean

- version 1.0 -

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

Concurrent Clean Language Report

Rinus Plasmeijer

Marko van Eekelen

April 1995
University of Nijmegen

Clean Report

Preface

History of Clean

The pure, lazy functional language **Concurrent Clean** (**version 1.0**) defined in this reference manual is a *major* upgrade of the previous release (0.84b) of Clean (Brus *et al.*, 1987; Nöcker *et al.*, 1991; Plasmeijer & van Eekelen, 1993).

Clean was originally designed as an experimental *intermediate* language and deliberately kept syntactically as poor as possible to be able to focus on the essential language and implementation issues. This strategy enabled us to study new concepts (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) without too much implementation effort. The ideas were tested in the Clean compiler which could be used on small machines and produced state-of-the-art code (Smetsers *et al.*, 1991).

The consequence was that people started to use Clean to construct large applications even though Clean was actually not intended as a programming language. So, it became necessary to turn the experimental *intermediate* language into a proper **practical applicable general purpose functional *programming* language** suited for **the development of real world applications**.

What is special about Concurrent Clean 1.0

Compared with the previous version of Clean we have added a lot of new features based on our own experience with writing complex applications (such as the new Clean I/O system). Many of the added language constructs are similar to those commonly found in other modern lazy functional languages (such as Miranda (Turner, 1985), SML (Harper *et al.*, 1986), Haskell (Hudak *et al.*, 1992) and Gofer (Jones, 1993)). People familiar with these languages will have no difficulty to program in Clean and we hope that they enjoy the compilation speed and quality of the produced code.

In addition Clean offers a couple of very special features. Of particular importance for practical use is Cleans' uniqueness typing enabling the incorporation of **destructive updates** of arbitrary objects **within a *pure* functional framework** and the creation of ***direct* interfaces with the outside world.**

Cleans' "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 which makes it possible to port window based I/O applications written in Clean without modification of source code.

In Clean it is possible to create **processes**. The new Clean I/O library takes advantage of this feature such that it is now also has become possible to develop **distributed executing interactive applications** running on several PC's/workstations connected in a network. The applications can communicate via **asynchronous as well as synchronous message passing**. Such a distributed application can be developed on one processor on which the processes will run in an **interleaved** fashion. This is very handy for testing.

The new Clean compiler still combines **fast compilation** with the generation of **efficient code** and is available on an increasing number of platforms (Mac, PC, Sun).

About this report

In this report the syntax and semantics of Clean 1.0 are explained. Although the report is not intended as introduction into the language, we did our best to make it as readable as possible. People already familiar with functional programming will have no problems to develop their applications in Clean. A quick introduction in functional programming, in Clean (albeit version 0.8) as well as in the underlying implementation techniques can be found in Plasmeijer and Van Eekelen (Adisson-Wesley, 1993). For people who are primarily interested in functional programming we recommend Bird & Wadler (1988).

In this report we always motivate *why* we have included a certain feature. These explanations occur at several locations. E.g. for each predefined data structure a basic motivation for its introduction is given 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 1.0 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). The complete collection of context free grammar rules are summarized in Appendix A.

Some of the features mentioned in this report are still under consideration, design and/or implementation and therefore not incorporated in the current release of the system (see Current State and Final Remarks). They should be regarded as "possible future trends".

Major differences with the previous release

Compared with the previous 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:

- Cleans' 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.

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

- the code generator is improved;
- we support more platforms;
- the code generator is prepared for parallel and distributed evaluation;

Some remarks on the new Clean syntax

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 to transform programs written in old Clean into new Clean.

The new Clean syntax is similar to the notation found in most other modern functional languages. So people familiar with these languages will have no difficuluties with programming in Clean. However, there are a couple of small 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).

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]
```

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.

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 `{…}`, and arrays look like this `{: … :}`.

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

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

macro args :== rhs                   // a macro definition (function synonym)
```

notational conventions

The following **notational conventions** are used in this report. Text is printed in Helvetica 11pts,
the context free syntax descriptions are given in Geneva 9pts,
`examples of Clean programs are given in Courier 10pts,`
textual explanation to the examples are given in Helvetica 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 12 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 Conurrent Clean Program Development system 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 commercial use of Clean 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 assist you to incorporate Clean and Clean applications in your specific environment.

Versions of Concurrent Clean are available for several platforms. The current situation is as follows:

Platform	Macintosh	Macintosh	PC	PC	PC	Sun	ParSyTec
Processor	Motorola	PowerPC	Intel	Intel	Intel	Sparc	Transputer
Proc. type	any	any	>386	>386	>386		T800
OS	MacOS vrs 7	MacOS vrs 7	>OS2 2.0	Linux	Windows 95	SunOS/Solaris	-
Window Sys	MacOS	MacOS	>OS2 2.0	Xview	Windows 95	Open-Look	-
Clean version	1.0	1.0	1.0	1.0	1.0	1.0	0.8
Clean IO lib	1.0 /0.8	1.0 /0.8	0.8	0.8	0.8	0.8	-
prim,sec mem	3,10	3,10	3,10	3,10	3,10	3,10	4 * n,-
assembler	not needed	not needed	gnu	gnu	gnu	Sun system	
linker	included	included	gnu	gnu	gnu	Sun system	
Code gen	Seq	Seq	Seq	Seq	Seq	Seq	Seq, Par
Available	now	soon	soon	later	planned	now	now

How to get started

How to install a Clean system 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. There are make files which will do the project management. We generate native code for all platforms (also for the PowerPC).

On the Mac there is a dedicated Clean Programming Development System. We are working on a platform independent Clean Programming Development System (yes, written in Clean).

Authors and Credits

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.
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 OS2.
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).

Parser, Support, Clean 0.8 to 1.0 Conversion program.

Rinus Plasmeijer &
Marko van Eekelen: Clean language design.
Rinus Plasmeijer: Overall design and implementation supervision.

Concurrent Clean and the Concurrent Clean System are 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.

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 Foundation for Applied Technical Sciences (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)).

Current State and Final Remarks

IMPORTANT

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 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.

Array comprehensions are not optimized yet and always generate lazy arrays. Arrays cannot be used as instantiations of type classes / type constructor classes. Only simple variables can be used as array pattern.

For derived members of type classes Clean currently macros are being used which are at this moment substituted in an early stage of the compilation process. This may cause criptical error messages. It is also annoying since macros cannot be used curried yet.

Observation of unique objects is switched off (eg in strict let expressions). This means that currently handling of unique objects is sometimes more tedious than is strictly necessary. It is not allowed to have multiple let statements in a row.

Strings are not yet equivalent to an array of Char.

Constructors for which functions are defined do not work.

Annotations for parallelism are ignored. The distributed code generator is switched off.

Everything exported in a definition module still has to be repeated in the corresponding implementation module.

The code generator is not yet optimal. User defined uniqueness type information is not yet exploitated (no destructive updates yet).

The Clean 1.0 library is only available for the Mac. Some of the primitives for message passing (synchroneous send, remote procedure call) are not yet implemented. On the other platforms still only the Clean 0.8 library (albeit converted to Clean 1.0 syntax) is available. For a description of the 0.8 system we refer to the Addison-Wesley book (Plasmeijer and Van Eekelen, 1993).

Sorry for all this inconvenience, we are working on it.

We hope that Clean 1.0 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.

Rinus Plasmeijer and Marko van Eekelen

April 1995

Computer Science Institute
University of Nijmegen
Toernooiveld 1
6525 ED Nijmegen
The Netherlands

email: rinus@cs.kun.nl
marko@cs.kun.nl

Clean Report

Contents

1

Introduction

In this section we summarize the key design rules and features of Clean 1.0.

1.1 Key design rules for Clean 1.0

Key design rules for Clean 1.0 have been:

- ❏ The language must be **purely functional**, **higher order** and **lazy**;
- ❏ The semantics of the language must be based on **graph rewriting systems**;
- ❏ The language must be **suitable for writing real world applications** in a very compact and readable style;
- ❏ It must be possible to create programs with an **efficiency comparable with C**;
- ❏ **Direct and efficient interfacing with the non-functional world** must be possible;
- ❏ One must be able to **control** the **time and space efficiency** of the program;
- ❏ **Parallel and distributed evaluation** of programs must be possible;
- ❏ Program components must be **re-usable**;
- ❏ A program (including window based interactive programs) must be **fully portable.**

1.2 Short summary of the features of Clean 1.0

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 type inferencing scheme (Milner 1978; Hindley 1969) 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**;

- Cleans' key feature is a **polymorphic uniqueness type inferencing system**, a special extension of the Milner/Hindley type inferencing 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 initialize 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;

- Clean 1.0 is supported on several platforms (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*. 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. The main "difference" between Clean and other 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.

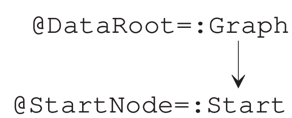
```
x  =: Add y z
y  =: Zero               => z                            (1)
x  =: Add y z
y  =: Succ a             => m  =: Succ n
                            n  =: Add a z                (2)

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.

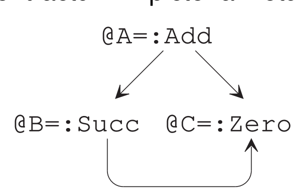
The patterns in rewrite rules contain **formal node-id’s**. During the matching these formal nodeid’s are mapped to the **actual node-id’s** of the graph. After that the following semantic actions are performed:

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:

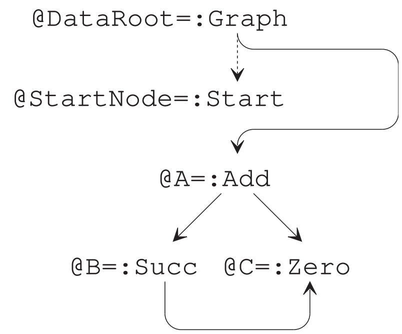


All applied occurrences of `@StartNode` will be replaced by occurrences of `@A`. The graph after rewriting is then:

The graph after rewriting:

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

Pictorial notation:

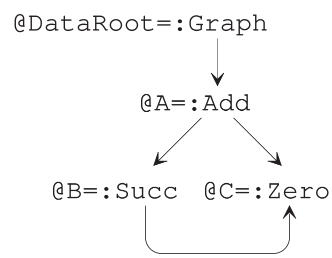


This completes one rewrite. All nodes that are not accessible from `@DataRoot` are garbage and not considered any more in the next rewrite steps. In an implementation once in a while garbage collection is performed in order to reclaim the memory space occupied by these garbage nodes. In this example 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 :

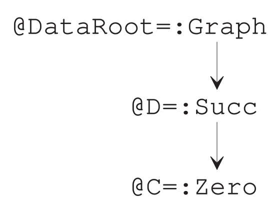


The graph accessible from `@DataRoot` still contains a redex. It matches rule 2 yielding the expected normal form:

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 1.0 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 (initialized 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 recognized by the scanner/parser (Sections 3.1, 3.2 and 3.3).

In Section 3.4 the symbols are summarized which are used in the context-free syntax description in the chapters hereafter (they are written in italic in the syntax description).

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 or type synonym
                   | :==                               // in a synonym type or macro definition
                   | =                                 // in a function, graph, alg.  type, rec. field
                   | =:                                // in 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
                   | <-                                // in list gen. in list or array comprehension
                   | <-:                               // in array gen. in list or array comprehension
                   | {                                 // begin of a record, begin of a block
                   | }                                 // end of a record, end of a block
                   | &                                 // an update of a record or array
                   | {:                                // begin of an array
                   | :}                                // end of an 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
                   | definition                        // begin of definition module
                   | 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 overloaded function
                   | instances                         // def of known instances
                   | let!                              // begin of strict let expression
                   | module                            // in module header
                   | of                                // in case statement
                   | overload                          // def of type scheme of overl. function
                   | 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
                   | .                                 // unique type variable
                   | *                                 // unique type
                   | :                                 // between a class type from uniq type var
                   | ->                                // function type constructor
```

```
| []                        // list type constructor
| (,),(,,),(,,,),...        // tuple type constructors
| {::}                      // array type constructor
| Bool                      // type boolean
| Char                      // type character
| File                      // type file
| Int                       // type integer
| ProcId                    // type process id
| Real                      // type real
| String                    // type string
| Void                      // type void
| 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 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				

3.5 Scope of definitions

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 (**program regions**) 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).

- 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 3.6, 12.1.1) have in principle as scope the module they are defined in, unless they are exported (see 12.3).

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.

3.5.3 Scope within functions and graphs

More complex are the scope rules within function and graph definitions (see the Chapters 5 and 6). In the picture below the general appearance of a function definition is given and the scope that is introduced by the different kind of definitions that can appear.

One can deduce from the picture that a new scope is introduced for each function alternative (see 6.1). Within such a function alternative new functions and graphs can be defined locally within a where block (see 6.6). Each of the guarded rule alternatives also introduce a new scope. Within such a rule alternative new functions and graphs can be defined locally within a with block (see 6.6). Each strict let clause introduces a new nested scope (see 6.4).

```
module scope_example

function args

         |guard =    let! selector = expr
                     …
                     in
                      rootexpression

                   with
                     selector = expr
                     …

                     function args = body
                     …
         |guard =      …
                     …

          where
            selectors = expr
            …
            function args = body
            …
```

New scopes are also introduced by list and array comprehensions and by case expressions. In a list and in an array comprehension new variables can be introduced when generators are specified. These variables introduces a new scope in the comprehension (from left to right). Each case alternative definition invokes a new scope which is identical to the scope rules of an ordinary function definition as indicated in the picture above (see also section 4.10).

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!`, `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
                        =  [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; = [n:filter pr r];
}
```

4

Expressions

In Clean, expressions are actually *graph expressions* which define the creation of a *(sub-) graph* (see 2.1). An expression generally expresses an application of a function or constructor to its arguments (see 4.2 - 4.4). For convenience and efficiency special syntax is provided to denote values of data structures of predefined type (see 4.5 - 4.9). A case clause, conditional expression and lambda abstraction is added for convenience (see 4.10).

4.1 Expressions

An expression can be labelled with a *selector* (see 5.2) which can be used to identify (parts of) the sub-graph expressed. 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 4.10
                | LambdaAbstr                     //  see 4.11
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
                | (GraphExpr)                     //  see 4.2
```

4.2 Applications

```
Application     = {BrackGraph}+                             //  application
                | GraphExpr OperatorSymbol GraphExpr        //  operator application
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 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). Rewriting can only take place if a symbol is applied to a number of arguments equal to the arity of the symbol (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) bind 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 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 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`, `Char` or `String`. These predefined types introduced for reasons of efficiency and convenience are treated in Section 8.1.1. notice that a string has to be regarded as an array and not as a list of characters. There is a special notation for a list of characters (see 4.6). The denotation of a constant value must obey the lexical description given in 3.2.

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

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 **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 seperated by a comma. The last generator in such a sequence will vary first. One can also define several generators in a row seperated 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. A special notation is provided for the frequently used *list of characters*. 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
```

- All list elements of a list must be of the same type.
- A list expression must be of type list.
- A guard must be of type `Bool`.
- Dot-dot expressions are perdefined on objects of type `Int`, `Real` and `Char`, but they can also be applied on any user defined object of enumeration type (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)]`)

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

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. Destructive updates of unique 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 following 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**). Left from the **&** (`r & f = v` is pronounced as: `r` updated with for `f` the value `v`) the old record that is used as blueprint must be of the same type as the 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.

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 and update):

```
::Person       = {   name         :: String,
                     address      :: String,
                     city         :: String,
                     cleanuser    :: Bool      }

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

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 . Destructive updates of unique arrays are provided. In Clean, array indices always start with 0. More dimensional arrays (e.g. a matrix) can be defined as an array of arrays. The predefined *type* array is treated in Section 8.1.2.

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

A new array can be created in a number of ways.

- First of all there are a number of handy functions for the creation and manipulation of arrays which are predefined in StdArray (see Appendix B.2.1).

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

```
createArray :: !Int a -> *{:a:}              // length arg1, a.[i] = arg2
mkArray     :: !Int (Int->a) -> *{:a:}       // length arg1, a.[i] = arg2 i
#           :: {:a:} -> Int                  // number of elements in array
##          :: {:a:} -> Int                  // upb of array  (#array - 1)
```

- Another way is to simply **list** the **array elements**.
- 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. On the right from the `&` the indices are listed in which the new array differs from the old one. One can use an **array comprehension** to list these elements compactly in the same spirit as with an list comprehension (see 4.6).
- The `&`-operator is strict in its arguments. Notice that the functional update is not an update in the classical, destructive sense since a *new* array is created. However, for reasons of efficiency the updates are only defined on arrays which are of unique type, such that the update can be done **destructively** (!) which in this particular case is safe since the original array is known not to be used anymore (see 9.2).
- One can even leave out the indices. The size of the array generated is then 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 (various ways to define an array with the integer elements `1,3,5,7,9`:

```
MkArray 5 (\i -> 2*i+1)
{: 1,3,5,7,9 :}
{: 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 :}
{: 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 :} :}
{: elem \\ elem <-: {: 1,3,5,7,9 :} :}
{: elem \\ elem <-  [  1,3,5,7,9  ] :}
```

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`).

- 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 '`&`' should yield an object of type 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.
- **It is assumed that arrays are mainly used to achieve optimal speed. A functional update is therefore for the time being *only* defined on unique arrays such that these updates can take place destructively!**
- An array expression must be of type array.

Example (array creation, selection, update):

```
SetArray :: *{:a:} Int a -> *{:a:}
SetArray a i v = {:a & [i] = v:}

CA :: Int a -> *{:a:}
```

```
CA m d = CreateArray m d

Fib :: Int -> {:Int:}
Fib m = fib
where
   fib = {:CA m 0 & [0]=1,[1]=1,[n]=fib.[n-1] + fib.[n-2] \\ n <- [2..m-1]:}

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

ScaleArray :: a {:a:} -> *{:a:} | Arith a
ScaleArray x a = {:x * e \\ e <-: a:}

MapArray :: (a -> b) {:a:} -> *{:b:}
MapArray f a = {:f e \\ e <-: a:}

inner :: {:a:} {:a:} -> *{:a:} | Arith a
inner v w
   | #v == #w  =  {:vi * wi \\ vi <-: v & wi <-: w :}
   | otherwise =  abort "cannot take inner product"

ToArray :: [a] -> *{:a:}
ToArray list = {: e \\ e <- list :}

ToList :: {:a:} -> *[a]
ToList array = [e \\ e <-: array]
```

4.10 Case expression and conditional expression

For programming convenience a *case expression* and *conditional expression* is included to make a choice between several alternatives (one can obtain the same effect by defining additional functions and use the pattern matching mechanism).

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

In the **case expression** first the discriminating expression is evaluated after which the patterns are tried in textual order. When a pattern matches the expression the corresponding alternative is chosen. One can define the same kind of patterns as one can also define on the left-hand side of a function definition, inclusive optional guard (see Chapter 6). A new block structure (scope) s created with a case expression (see 3.5).

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

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 special case of the case expression.

- The then- and else-part in the conditional expression must be of the same type.
- The boolean expression must yield an object of type `Bool`.

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

4.11 Lambda abstraction

If one wants to define a tiny function it can sometimes be convenient to use a lambda abstraction. Since the dot is already used for record and array selection the syntax for lambda abstraction is as follows:

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

Example (lambda expression):

```
TupleList :: [a] -> [(a,a)]
TupleList = map (\x -> (x,x)) list
```

5

Defining graphs

Graph definitions can appear on the global level and on the right-hand side of a function definition (see 5.1) to construct the contractum (see Cleans' basic semantics in Chapter 2). Graphs can be expressed by a graph expression (see Chapter 4). A selector is a right-hand-side pattern used to identify sub-graphs of a graph (see 5.2).

5.1 Graph definitions

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

When a **graph** is **defined** a name is given to (part) of a constant graph expression (see Chapter 4). The definition of a graph can be compared with a definition of **constants** (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 a 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 occurence 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 ment. Constant definitions are ambiguous. Locally they are *by default* taken to be *graph* definitions (see 5.1.1), globally they are *by default* taken to be *function* definitions (see 6.1).

Example (Graph versus constant function definition:`biglist1` is a *graph* which is computed only once but which consumes a lot of space. `biglist2` is a constant *function* which is computed every time it is applied but it consumes only a modest amount of space):

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

5.1.1 Defining graphs in functions

The contractum graph specified on the right-hand side of a rewrite rule (function) which has as root the graph defined in the root expression (see 6.5) can be composed out of **locally defined** (**sub-**) **graphs** (see Cleans' basic semantics in Chapter 2). A (sub-) graph can be defined in a *strict let expression* (see 6.4), as local definition in a *function body* (see 6.6), and as local definition for a *rule alternative* (see also 6.6).

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 definition 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 = #string-1
```

5.1.2 Defining graphs on the global level

Graphs can also be defined on a global level.

```
ImplModDef          = …
                    | 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).

5.3 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. 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).

Remark: a selector can also appear on the left-hand side of a generator in a list comprehension (see 4.5) or array comprehension (see 4.8).

Example (use of selectors to select record elements).

```
::Rectangle =  {  toplft   :: (Int,Int),
                  botrgt   :: (Int,Int)
               }

WindowTopAndBottom :: IOState s -> (Int,Int,IOState s)
WindowTopAndBottom io
=  (top,bot,io`)
   where
      ({toplft = (_,top),botrgt = (_,bot)},io`) = ActiveWindowGetFrame io
```

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. This is explained in Chapter 8.

6.1 Defining functions

```
FunctionDef        = [FunctionTypeDef] DefOfFunction
DefOfFunction      = {FunctionAltDef}+
FunctionAltDef     = FunctionSymbol {Pattern}
                     {[| Guard] =[>] FunctionBody}+
                     [LocalFunctionAltDefs]
FunctionSymbol     = FunctionSymb                    // ordinary function
                   | (FunctionSymb)                  // operator function
FunctionBody       = {StrictLet}+
                     RootExpression ;
                     [LocalFunctionDefs]
```

A **function definition** consist of one or more definitions of **function alternatives** (rewrite rules) which are tried in textual order. The left-hand side of such a function alternative can serve a whole sequence of **guarded function bodies** (called the **rule alternatives**). Each rule alternative is semantically considered as an individual rewrite rule alternative. A particular rule alternative is chosen for evaluation when

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

When a rule is evaluated, first the sub-graphs defined in the *strict let expression* will be evaluated (see 6.4). Hereafter the *root expression* is evaluated (see 6.5). In each *rule alternative* one can optionally specify *local definitions* containing the definitions of sub-graphs and functions that have a meaning for that rule alternative only (the *with* block, see 6.6). The last rule alternative in a sequence can optionally also be followed by local definitions that have a meaning for *all* preceding rule alternatives belonging to the same left-hand side (the *where* block, see 6.6 as well).

- 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 graph.

An **operator** is a **function with arity two** that can be used in infix notation. Operators can be applied in infix notation (brackets are left out) or as ordinary prefix function (the name 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 Left-hand side patterns

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 explained. A **pattern** in general 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 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). It is allowed to put redundant brackets on patterns.

```
Pattern          = [Variable =: ] BrackPattern
BrackPattern     = SimplePattern                 // 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 patterns 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. A compile time *warning* is generated that such a situation might arise.

6.2.1 Constructor patterns

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

A **constructor pattern** (see above) in general 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
```

A symbol occurring in a pattern can also be a symbol that is *both* defined as a data constructor in an algebraic data type and defined as function symbol of a function (see 8.3.4).

6.2.2 Simple patterns

```
SimplePattern       = ConstructorSymbol
```

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 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.

- All variable symbols appearing 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`, `Char` or `String` (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 list patterns).

- All list elements of a list must be of the same type.

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
             =  [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 = GraphPattern}-list:}
```

- 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.

Example (use of array patterns):

```
Swap :: !*{:a:} !Int !Int -> *{:a:}
Swap a=:{:[i]=ai,[j]=aj:} i j = {: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 emphesis 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. A 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 Strict let expression

Although Clean is a lazy functional language one can force evaluation of sub-graphs by using a **strict let expression**. This can be used to put unique objects in a preliminary context such that they can be *observed* before they are *destructively updated* (see 13.6). By forcing evaluation one can obtain 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.

| StrictLet = **let!** ⟦ {GraphDef}+ ⟧ **in**

Before the root expression is evaluated (see 6.5), first the sequence of sub-graphs specified in the strict let expression is evaluated to strong root normal form. The order in which the sub-graphs specified in the strict let expression themselves are evaluated is undefined. One may repeat a strict let expression thus sequentialising evaluation order.

- If there is a sequence of strict let expressions it is not allowed that a graph expression defined in one let expression refers to a graph expression defined in a next let expression (since such a graph expression should be evaluated after the first one). The other way around is allowed. See also 3.5.

Example (let! expression forcing evaluation):

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

6.5 Root expression

The expression defining the root of the contractum is called the **root expression**. Such a right-hand side of a rule alternative is either a variable (in the case of a *redirection*) or a *graph expression* (in the case a contractum graph is constructed) (see Cleans' basic semantics in Chapter 2). not the whole contractum need to be specified in the root expression. Parts of the contractum (sub-graphs of the contractum) can be refined after the root expression (in a local definition) or before (in the strict let expression).

| RootExpression = GraphExpr

Example (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))]
```

6.6 Local definitions

For the purpose of introducing definitions with a limited scope both graphs as well as functions can be defined locally. *Local functions* are always lifted to the *global level* by the compiler whereas *local graphs* are not lifted and stay on a local level (see 13.5). There are two kinds of **local definitions**.

- Firstly, each *rule alternative* can contain local definitions. They are defined in a **with block** introducing a new scope (see 3.5). The definitions given in such a block have only a meaning for the *corresponding rule alternative*.

- Secondly, as usual in functional languages, one can also define local definitions in a **where block** (a block preceded by the keyword `where`) that have a meaning for the *whole sequence of rule alternatives* belonging to the same left-hand side (see 6.1).

When the sequence of rule alternatives only consists of one alternative the scope and meaning of both kind of local definitions is the *same* and it therefore does not make any difference which kind of block is used (in that case the keywords can even be left out if the curly braces are used).

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

Example (local function definition):

```
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
                        =  [n:filter pr r]
```

7

Process annotations

In Clean one can create processes to influence the efficiency of the program or to obtain a specific distribution of parts of the program across a network of processors. In this chapter the creation interleaved or parallel executing processes are discussed. This done by adding process annotations (Plasmeijer and van Eekelen, 1993) to function applications. In Clean one can also create distributed executing interactive applications in Clean. This is discussed in Chapter 10.

NOT SUPPORTED IN CURRENT RELEASE.

8

Defining types

Clean is a strongly typed language. The basis of the type system of Clean is the classical polymorphic Milner/Hindley (Milner 1978; Hindley 1969) type system. This type system is adapted for graph rewriting systems and extended with *basic types*, *algebraic types*, *existential types*, *record types*, *synonym types* and *abstract types*. This more or less standard type system is 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) but they work slightly different) such that a restricted context can be assigned to an overloaded function. This is explained in Section 8.4.

In Clean types can be attributed with *strictness information* (see Section 8.5). With these strictness information one can turn the default lazy evaluation strategy of Clean into a (partially) strict one.

8.1 Types

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

```
Type            = [UnqTypeAttrib] TypeConstructor {BrackType}      // see 8.3, 9.1, 9.2, 9.3, 8.5
                | BrackType
BrackType       = [UnqTypeAttrib] SimpleType
SimpleType      = 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), `Bool` (for 8 bits boolean values) and `String` (for an array of characters). 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 these predefined abstract 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
                | String           // see StdString.dcl
```

8.1.2 Predefined abstract types

As is explained in Section 8.2.4, *Abstract data types* are algebraic data types of which the actual definition is hidden. In Clean the types `World`, `File`, `ProcId` and `Void` are ***predefined* abstract data types**. This means that they are recognized by the compiler and that they are treated special, either for efficiency or because these objects 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 `File`s 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.

The predefined type **`Void`** is introduced for don't care usage, e.g. for instantiating an existentially quantified type variable. There is no special denotation for an object of type `Void`. One can use e.g. an `abort` statement (see `StdMisc`) to create an object of type `Void`.

```
PredefAbstrType = World            // see StdWorld.dcl
                | File             // see StdFileIO.dcl
                | ProcId           // see StdProcId.dcl
                | Void             // see StdMisc.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 (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]
```

- All elements of a list must be of the same 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 (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.

An array can optionally be annotated as being strict (see 8.5.5). This can be used to increase the efficiency of a program (see 13.3). **Strict arrays** are considered of being of different type then **lazy** ones. The compiler does not perform conversions between lazy and strict arrays (see 8.5.5).

```
ArrayType           = {:[Strict] Type:}
```

- All elements of an array must be of the same type.

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 to:

```
((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).

```
ImplModDef        = TypeDef
                  | OverloadingDef
                  | 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
OverloadingDef    = OverloadedFunctionTypeDef            // see 8.4 and 9.4
                  | TypeInstanceDef                      // see 8.4 and 9.4
                  | TypeClassDef                         // see 8.4 and 9.4
FunctionDef       = [FunctionTypeDef] DefOfFunction      // see 8.3 and 9.3
DefModDef         = TypeDef
                  | AbstractTypeDef                      // see 8.2.4 and 9.2.4
                  | OverloadingDef
                  | FunctionTypeDef
                  | MacroDef
```

8.2.1 Defining algebraic data types

An **algebraic data structure** consist of a **constant** (called **data constructor**) followed by zero or more algebraic data structures (of any kind) as argument. Every user defined data structure and data constructor must unambiguously have been (pre)defined in an **algebraic data type** definition. An algebraic data type declaration introduces a *new type* and *new data constructors* over that type. Several constructors can be introduced in one type definition which makes it possible to define alternative data structures of the *same* algebraic data type. Constructors of an algebraic data structure can, just like functions, be used in a curried way.

Polymorphic algebraic data types can be defined by using **type variables**. The arguments of the constructor in a type definition are type instances of types that are defined or are being defined recursively. A type instance of a type `T` can be obtained by uniformly substituting types for the type variables in `T`.

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

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

TypeLhs           = [UnqTypeAttrib] TypeSymb {TypeVarDef}
TypeVarDef        = [E.] [UnqTypeAttrib] TypeVariable
```

```
ConstructorDef      = ConstructorSymb {[Strict] BrackType}
                    | (ConstructorSymb) [Fix][Prec] {[Strict] BrackType}
Fix                 = infixl
                    | infixr
                    | infix
Prec                = Digit
```

Example (algebraic type definition):

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

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

Constructors with two arguments can, in a similar way as function operators (with fixity (`infixl`, `infixr` or just `infix`, default `infixl`) and precedence (`0` through `9`, default `9`), be defined as **infix constructor**. 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):

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

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

- The name of a type must be different from other names in the same scope (see 3.5) and name space (see 3.4).
- Types can only be defined on the global level.
- 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.
- When a constructor is used in infix position both arguments have to be present. Constructors can be used in a curried way, but then they have to be used as ordinary prefix constructors .

Defining algebraic data types with existentially quantified variables

Clean incorporates the extension of the Hindley/Milner type system with the possibility of algebraic types to be existential (Läufer 1992). An **existential algebraic data type definition** is an algebraic type definition in which **existentially quantified variables** are used. These special variables are marked 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). Such kind of data structures are for instance used internally in Cleans' I/O library to store (program and I/O) states of different types and state transition functions defined on these states in one data structure (see 10.3).

Example (existential algebraic type definition and its use):

```
::Tree E.a  = NilTree
            | NodeTree (a, a, a a -> a) (Tree Void) (Tree Void)
```

```
F :: Tree Void
F = NodeTree (3,5,(+)) (NodeTree (3.5,5.4,(+)) NilTree NilTree) NilTree

G :: (Tree Void) -> Tree Void
G (NodeTree (obj1,obj2,func) l r) = NodeTree (func obj1 obj1,obj2,func) l r
```

There are severe limitations imposed on the use of data structures of **existential types**. Once a data structure of existential type is created and is passed to another function it is generally statically not possible to determine what the actual type is of those components of the constructor that correspond to the existential quantified variables.

- Therefore, it is *not* allowed to pass such objects to other functions as argument or result *if* these functions either require or deliver this actual type. In other words, for the type inference system an existentially quantified type variable is treated as a type variable that can be unified with a concrete type (= not a type variable) *only* at the explicit creation of a data structure of this type with its defining data constructor. In all other contexts an existentially quantified type variable can only be unified with non concrete types (type variables).
- For software engineering reasons it is required that an existentially quantified type variable is instantiated with the predefined type `Void` (see the example above).

Components that correspond to the *same* existentially quantified type variable will have the *same* type. So, it is allowed to apply these components in expressions that yield an ordinary type. It is also allowed to use the components to create a new object of existential type. Furthermore, it is allowed to pass the existentially quantified type variable to polymorphic functions.

Apart from the restrictions mentioned above existential algebraic types are not different from standard algebraic types. They can be used e.g. as the basis of record types, synonym types and abstract types.

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 they 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). Existentially quantified type variables (see 8.2.1) are allowed in record types. 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 existentially quantified type variables in record types are of use for an object oriented style of programming.

Example (using 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  }

::MyObject  :== Object Void

CreateObject1 :: MyObject
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 :: MyObject -> Int
Get {state,get} = get state

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

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

8.2.3 Defining synonym types

Synonym types permit the programmer to introduce meaningful type names for a type instance of any type. It also enables the type system to give shorter and more understandable error messages.

```
SynonymTypeDef     = ::TypeLhs :== Type ;
```

- For the left-hand side the same restrictions hold as for algebraic types (see 8.2.1).
- All type variables used on the right-hand side must have been defined on the left-hand side.
- Mutual dependencies of synonym type definitions are 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

Abstract data type definitions are type definitions appearing in definition modules of which the actual definition (the right-hand side of the type definition) is hidden in the implementation module. So, Cleans' 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. The concrete implementation of an abstract type can be given in the corresponding implementation module but it is also possible to define an abstract data structure based on a concrete type as defined elsewhere. This feature enables the definition of operations on abstract data types over several definition modules which is very convenient for the creation of *library packages* (such as Cleans' Event I/O library). All implementation modules of such a library package can use the concrete definition the abstract data type is based on.

- Abstract data type definitions are only allowed in definition modules (see Chapter 11), their concrete implementation has to be given either in the corresponding implementation module or it has to be explicitly imported from another module. The concrete implementation remains invisible for other modules built on top of the definition module in which the abstract type is defined.
- In the concrete type either an algebraic type or a record type or a synonym type should be defined with the same left-hand side as given in the abstract type in the definition module.

```
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 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]

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 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.

- Local functions can only be typed if its type does not contain type variables introduced in the type specification of a surrounding function (there is not a scope rule on types defined).
- 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
```

8.3.1 Typing curried functions

In Clean all symbols (functions and constructors) are defined with **fixed arity**. However, in a application it is 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** because the **`abort`** expression can be unified with any type. This can be used to give a slightly better 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)
               = abort "fac called with a negative number"
```

8.3.4 Typing functions with functions on constructor position

A symbol can be defined both as function as well as a constructor in an algebraic type (see also 8.3.5). A symbol which is defined in this way can behave like a function (it is rewritten when it matches a function definition) but it can also behave like a constructor (when it does not match the function definition). Such a function can therefore be used as ordinary constructor (**functions used on constructor position**), e.g. as pattern in the left-hand side of a function definition. One can also regard it the other way around: for a constructor a function is defined as well (**constructor for which a function is defined**).

NOT SUPPORTED IN CURRENT RELEASE.

8.3.5 Type consistency

The type specifications of graphs and functions are **inferred** at compile time by a **type inference mechanism**. However, for software engineering reasons it is highly recommended to always specify types explicitly. **Explicit type specifications** are allowed to be more *restrictive* than the inferred types. In that case the restrictive type is used in all contexts.

A **type specification** must be **correct**. In Clean a **classical type specification** for a graph or a function is defined to be **correct** if it satisfies each of the restrictions given below:

- for each argument of each symbol the **demanded type** (i.e. the type that is indicated in the corresponding type specification) is **unifyable** (i.e. there exists a type that is a type instance of both types) with the **offered type** (i.e. the actual type of the argument in the application);
- for each right-hand side the offered type is unifyable with the right-hand side of the corresponding type specification;
- an existentially quantified type variable can be unified with a concrete type (= not a type variable) *only* at the explicit creation of a data structure of this type with its defining data constructor. In all other contexts an existentially quantified type variable can only be unified with non concrete types (type variables);
- a function symbol which occurs on a constructor position in a pattern must be defined as constructor with the same type as the function.

A correctly typed graph rewriting system specified in Clean will always be **safe** (Van Bakel *et al.*, 1991) i.e. if an expression has type τ then all reducts of the expression can be typed with τ.

8.4 Typing overloaded functions and operators

The names of identifiers in the same scope and name space all have to be *different* (see 3.4). However, sometimes it is very convenient to *overload* 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*. One way to simulate overloading is by using records (see 8.2.2). A **type class record** is an ordinary record of which functions are stored which somehow belong to each other. Functions defined on such a record can use the field name of such a record as (overloaded) synonym for any concrete function stored in the corresponding field. An instantiation of a type class record can be regarded as a kind of **dictionary** in which the concrete function can be looked up.

Example (the use of records to simulate 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=:{add} [x:xs] [y:ys]  =  [add x y:sumlist arith xs ys]
sumlist arith x y                   =  []

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

A disadvantage of this method is that it is syntactically not so nice (e.g. one explicitly has to pass the class 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. The overloading mechanism in Clean introduces syntactic sugar and tries to avoid the efficiency penalty. The syntax for the definition and creation of this specific use of records enables the Clean system to automatically add them 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 the system will indeed have to generate a class record structure which is passed as additional parameter as explained above.

8.4.1 Defining overloaded function types and concrete instances

With an **overloaded function type definition** one defines the **type scheme** of the overloaded function. The type of a concrete function must be an instance of this type scheme. The special **type scheme variable** defines in which variable the scheme can vary. With an **instance** declaration one defines an overloaded function c.q. operator name to be a *synonym* for some **concrete function** or **operator**. In the instance definition it is specified which concrete function is ment and for which **concrete type** an instance of the overloaded function is created. The type of the concrete function must be equal to the overloaded type after uniform substitution of the specified concrete type for the type scheme variable. For a concrete function one can refer to a function which has already been defined elsewhere or one can define a new function right on the spot.

```
OverloadedFunctionTypeDef
                  = overload FunctionSymb TypeVariable ::  FunctionType;
                  | overload (FunctionSymb) [Fix][Prec] TypeVariable ::  FunctionType;
OverloadedFunctionInstanceDef
                  = instance FunctionSymb [BrackType[ClassContext]][=FunctionSymb];
                  | instance FunctionSymb [BrackType[ClassContext]]
                    DefOfFunction;
```

One can define as many instances as one like. Instances can be added in any module. One and the same concrete function can be used as instance for different overloaded functions (as long as the types match).

Example (defining an overloaded operator and instantiations with existing concrete operators): The types of the concrete operators `+^` and `+.` need to be instances of the type scheme of `+` (take for the type scheme variable `a` of the overloaded operator `+` respectively `Int` and `Real`)

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

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

instance + Real
(+) x y = x +. y
```

When an overloaded name is encountered in an expression, the compiler will determine which of the corresponding concrete functions/operators is ment by looking at the concrete type of the expression. From this type the concrete function to apply is determined. All concrete functions/operators of an overloaded function/operator must therefore be defined on *different* instances of the type scheme (with exception of the default instance, see below). If it is clear from the type of the expression which one of the concrete function is ment the compiler will in principle substitute the concrete function application 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
```

If it is not clear from the type of the expression which concrete function is ment (more than one of the concrete functions fit type technically) the compiler will make specialized versions of the function of which the expression is part of. In principle one version is made for each concrete possible substitution. However, the compiler will avoid making versions which are not being used or which are not important in terms of efficiency and in any case it will avoid code explosion.

Example (creating specialized versions for overloaded functions): assume the following function definition and `Start` rule:

```
sumlist [x:xs] [y:ys]   =  [x + y:sumlist xs ys]
sumlist x y             =  []

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

From the context of `sumlist` it is not clear which concrete instantiation of `+` to be used. So the compiler will in principle generate a special version of `sumlist` for all possible versions which are needed. In this example `sumlist` is only applied to integer lists. So the compiler will only generate a version of `sumlist` for `Int`s as follows:

```
sumlistI [x:xs] [y:ys]  =  [x +^ y:sumlistI xs ys]
sumlistI x y            =  []

Start = sumlistI [1..10] [11..20]
```

It is possible to specify a function as **default instance** (no concrete type instance is specified for the type scheme variable in the instance declaration in that case) 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 default function must be equivalent to the type of the overloaded function. The default function provides the possibility to define a standard interpretation for an overloaded function.

Example (defining a default instance indicating that objects are by default unequal unless specified otherwise)

```
overload (==) infix 2 a :: a a -> Bool

instance ==                          // default function for ==
(==) infix 2 :: a a -> Bool
(==) x y = False
```

The following restrictions apply:

- The type of a concrete function or operator must exactly match the overloaded type scheme after uniform substitution of the type scheme variable by the type as specified in the corresponding type instance declaration.
- 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.
- It is not allowed to use a type synonym as instance.
- All instances other than the default instance of a given overloaded type must differ from each other (be ununifyable with each other).
- 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.
- If the concrete function or operator is not specified in a definition module, it has to be defined in the corresponding implementation module.
- The start rule cannot have an overloaded type.
- Ambiguously overloading is not permitted.

8.4.2 Type classes

When a function is defined in terms of an overloaded function it can occur that the type system cannot decide which one of the corresponding concrete functions to apply. The new function then becomes overloaded as well. This has as consequence that an additional restriction must be imposed on the use of such a function. This is reflected in the type of the function.

For instance, the function

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

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

In a **type class definition** one gives a name to a **set of overloaded functions** (this set actually defines a type class record, see above). The definition of the overloaded functions themselves can be directly specified in the type class definition itself but one can also refer to overloaded functions type definitions declared elsewhere. One and the same overloaded function can be a **member** of different type classes. Instances of the overloaded functions are created as described above. There is no hierarchy in type classes.

TypeClassDef	=	**class** *ClassSymb TypeVariable*
		[**where**] ⦃ {ClassMemberDef}+ ⦄
		[**instances** {[BrackType[ClassContext]]}-*list* ;]
ClassMemberDef	=	FunctionTypeDef [MacroDef]
	|	*FunctionSymb* ;
TypeClassInstanceDef		
	=	**instance** *ClassSymb* [BrackType[ClassContext]]
		[**where**] ⦃ {DefOfFunction}+ ⦄

Example (definition of a type class):

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

Example (equivalent definition):

```
class Arith a
where +; -

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

In the definition of the type of a function that has become overloaded one can now refer to a type class to impose a restricted **context** on the instantiation of a type variable of the function (see 8.3). Such a context imposes a condition (predicate) under which type variables is allowed to be instantiated. So one obtains a kind of **bounded polymorphism**. The function can only be applied if for the corresponding concrete type the indicated type classes have been instantiated.

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

Example (use of a type class to impose a restriction on the instantiation of type variable): the function `add` can only be applied on arguments for which an instance of the class `Arith` is defined.

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

Example (instance declaration of which type is depending on same type class): the function `sumlist` itself can also be defined as an instance for the overloaded operator + working on arbitrary list for which the type class `Arith` is defined on the list elements. `sumlist` must now be defined as operator because this is expected from an instance of +. With this definition + is defined on lists, and list of lists etc.

```
instance + [a] | Arith a
(+) [x:xs] [y:ys] =  [x + y:xs + ys]
(+) x y           =  []
```

Cleans' type system will infer contexts automatically. It will however *not* express this in terms of type classes but simple summarize the collection of functions on which the overloaded function is depending. This is caused by the fact that it is allowed that one function is defined as a member of several type classes. 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).

- For an overloaded function which is exported the type (including the context) has to be defined explicitly by the programmer.
- The type checker will complain if a concrete application cannot be applied due to the fact that certain instances of type classes have not been declared.

Example (Equality class).

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

```
instance == [a] | Eq a                              // on lists
(==) [x:xs] [y:ys]   = x == y && xs == ys
(==) []     []       = True
(==) _      _        = False

instance == (a,b) | Eq a & Eq b                     // on tuples of arity two
(==) (x1,x2) (y1,y2) = x1 == y1 && x2 == y2
```

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 seen as synonym for `not (==)`. For software engineering (the fixed relation is made explicit) and efficiency (one does not need to include derived members in the class record) reasons it is good to make this relation explicit. In Clean macro definitions can be used for this purpose.

Example (Classes with derived members).

```
definition  module  overloaded

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
```

When an overloaded function is exported a type class record has to be constructed as explained in the introduction. However, for efficiency reasons it would be nice to know which instances of the type class are known in the implementation module. When a type class is exported one can explicitly define the minimal **set of instances** which exist for this type class. For any function defined on such a type class one can now deduce that at least these instances are known.

Example (Classes with derived members).

```
definition  module  overloaded
```

```
overload == a :: a a -> Bool
```

definition module StdInt

```
import overloaded

instance == Int
```

definition module StdReal

```
import overloaded

instance == Real
```

definition module example

```
import StdInt, StdReal

class Eq a
where ==
instances Int, Real
```

8.4.2 Constructor classes

Clean offers **constructor classes** (see Gofer, Jones (1993)) offering the possibility to define an overloaded function for concrete functions on arbitrary types which are of the same **kind**. The similarity is expressed by allowing the variable in an overloaded type scheme to stand for type symbols (type constructors) as well. The type variable can be instantiated with any concrete type (a user defined type symbol, a predefined **type constructor** like the arrow type constructor, list type constructor,, tuple type constructor, and array type constructor,) as long as the instantiation of the type scheme results in a "normal" type after uniform substitution.

TypeConstructor	= *TypeSymb*	// an algebraic type (including records)
	| **(->)**	// an arrow type
	| **[]**	// list type
	| **(**{**,**}+**)**	// tuple type
	| **{::}**	// array type

Example (functions with similar behaviour): the well-known function `map` is generally applied on lists, but a similar function can also be defined on trees.

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

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

mapT :: (a -> b) (Tree a) -> (Tree b)
mapT f (l/\r)      =  mapT f l /\ mapT f r
mapT f (Leaf a)    =  f a
```

On overloaded function `map` with as concrete instantiations `mapL` and `mapT` can be defined as follows:

```
overload map t :: (a -> b) (t a) -> (t b)
```

```
instance map []   =  mapL
instance map Tree =  mapT
```

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).

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 strictness analyser of the Clean System based on abstract reduction (see Nöcker, 1993) is able to derive the strictness of the function arguments for many cases, such as for the example above. Generally there is no need to add strictness annotations for functions by hand.

8.5.3 Algebraic data structures and records with strict arguments

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.

Strictness is a property of a the constructor of a certain type. Generally (with the exception of tuples) one cannot create strict versions for an instance of a certain non-strict (predefined) type. For instance, to create a spine strict list one has to define a new algebraic data type (see below). To be able to create strict and lazy version of an array, a lazy as well as a strict abstract data type has been predefined.

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).

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

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}
```

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. One therefore needs to specify strictness annotations for data constructors by hand and one has to ensure that no infinite computations occur.

8.5.4 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. A tuple is the only type for which it is allowed to make a strict instance.

This can be done by putting strictness annotations in the type instance on the tuple elements that one would like to make strict. The tuple itself has to be annotated as strict as well. 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).

notice that strictness is not a property of a tuple type but of a particular instance of the tuple type. This means that the type system will consider strict and non-strict versions of a tuple as being equal. The compiler will insert code to convert non-strict tuples into strict version and backwards whenever this is required.

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

A special feature of Clean is that the classical types can be extended with *uniqueness type attributes*. Uniqueness typing forms a key feature of Clean (Plasmeijer and Van Eekelen, 1993; Barendsen *et al.*, 1993; Barendsen and Smetsers, 1993). It is related to linear type systems (special is the subtype relation on unique types and the input taken from a reference count analyser) with gives the opportunity to interface Clean programs in a direct and efficient way with the non-functional world while the resulting program still remains a *pure* functional program. For instance, the type system makes it possible to directly interface 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). The uniqueness type system is polymorphic.

The uniqueness type system in Clean 1.0 is an inferencing system capable of deducing uniqueness types automatically. Also in the case of uniqueness typing we advise the programmer to specify types explicitly. The type system will infer uniqueness type attributes for *all* types, not only for the types corresponding to the vital objects one is primarily interested in. The deduced types often give a lot of extra information on the actual behaviour of a function.

9.1 Uniqueness typing

Since the uniqueness type system is a rather new phenomenon we explain this type system and the motivation behind it in more detail.

9.1.1 Basic ideas behind uniqueness typing

The **uniqueness type system** is an extension on top of the classical type system. In the uniqueness type system a **uniqueness type attribute** is attached to each classical type (see Chapter 8). Uniqueness type attributes appear both in the definitions of *new types* as (see 9.2) well as in the *type specification of a function* (see 9.3). A classical type can be prefixed by one of the following uniqueness type attributes:

```
Type            = [UnqTypeAttrib]  TypeConstructor  {BrackType}
                | BrackType
BrackType       = [UnqTypeAttrib]  SimpleType
UnqTypeAttrib   = *                                   // type attribute "unique"
                | UniqueTypeVariable:                 // a type attribute variable
```

```
                    | .                                  // an anonymous type attribute variable
```

The basic idea behind uniqueness typing is the following. A uniqueness type attribute imposes an additional restriction on the use of the corresponding object.

- When in the type specification of a function an argument of type `T` is attributed with the type attribute **unique** ("*****") it is *guaranteed* by the type system that the function will have private ("unique") access to this particular argument (see Barendsen and Smetsers[1], 1993; Plasmeijer and Van Eekelen, 1993): the object will have a reference count of one[2] *at the moment* it is inspected by the function. It is important to know that there can be many references to the object before this specific access takes place. If a uniquely typed argument is not returned again in the function result it has become garbage (the reference has dropped to zero). Due to the fact that uniqueness typing is static the object can be garbage collected (see Chapter 2) at compile time. In particular it is perfectly safe for the compiler 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))
```

So, 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. The type system makes it possible to let a Clean file and a physical file of the real world be one and the same: 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` with a shared file being used elsewhere (it is not possible that both the original as well as the modified file exist at the same time).

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

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

- When there is no uniqueness type attribute attached to a classical type in a function type definition, it is assumed that no uniqueness is required: the corresponding object is assumed to be attributed as "non-unique". Private access to a non-unique argument cannot be guaranteed, the object can be

[1] Compared with the theoretical system as described in Barendsen and Smetsers, 1993), Cleans offers a slightly restricted (these restrictions do not impose serieus restrictions for practical applications) simplyfied type system which is reasonably efficient as well.

[2] 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.

shared by others. The function is only allowed to have **read access** (as usual in a functional language) even if the function happens to be applied with a unique attributed actual argument.

```
freadc :: Char File -> File
```

The function `freadc` accepts 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 anymore.

- One can also define functions which are **polymorphic** in their **uniqueness type attribute**. Such functions will accept both unique as well as non-unique objects (called **possibly unique** objects). **Uniqueness type variables** will be uniformly substituted as is the case with ordinary type variables.

```
freadc :: Char u:File -> u:File
```

The function freadc above now accepts both a unique (and then it also yields a unique file) as well as a non-unique file (and then it will also yield a non-unique file). In other words this function is polymorphic in its uniqueness type attribute indicated by the uniqueness type variable `u` attached to the ordinary type `File`. If in a concrete application the function is applied to a non-unique file only further reading is possible, if it is applied to a unique file one can either continue with reading as well as with writing.

The **anonymous uniqueness type variable** is used as a shorthand when the names of the uniqueness type variables do not matter introduced to keep type specifications as readable as possible (see 9.3).

So, uniqueness typing makes it possible to update objects destructively within a pure functional language. For the development of real world applications (which manipulate files, windows, arrays, databases, states etc.) this is a very desirable property.

9.2 Defining new types with uniqueness attributes

Uniqueness type attributes generally only appear in a type of a function when destructively updateable objects are being manipulated. However, uniqueness type attributes are actually attached to *every* classical type, both in the definition of an algebraic type as well as in a function type. The default values for the uniqueness attributes are chosen in such a way that one can leave out the attributes if one don't care about unicity. The classical notation is also a valid type for Cleans' type system extended with uniqueness type attributes. As a consequence the uniqueness typing can remain a *hidden* feature when it is of no concern. Even when a classical data type has been defined with no uniqueness in mind, a unique variant can still be instantiated from it.

```
AlgebraicTypeDef   = ::TypeLhs = ConstructorDef {: ConstructorDef} ;
TypeLhs            = [UnqTypeAttrib] TypeSymb {TypeVarDef}
TypeVarDef         = [E.] [UnqTypeAttrib] TypeVariable
ConstructorDef     = ConstructorSymb {[Strict] BrackType}
                   | (ConstructorSymb) [Fix][Prec] {[Strict] BrackType}
BrackType          = [UnqTypeAttrib] SimpleType
```

When uniqueness type attributes are not specified explicitly in the definition of a type, a fresh uniqueness type *variable* is internally added as attribute to each classical type (is does not matter whether an algebraic, record, synonym or abstract type is being defined), obeying the following general rules:

- All occurrences of a classical type variable must be attributed with the same uniqueness type attribute within its scope.
- All identical instantations of a recursive algebraic type must have the same uniqueness type attribute within the recursive definition.

Example (uniqueness expansion of a classically defined type): The classical algebraic definition of a list:

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

is internally automatically expanded to:

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

For the instantiation of a uniqueness type variable certain restrictions hold. One of the most important restriction is the *propagation rule* stating that unique objects can only be stored in data structures which are unique themselves (see Barendsen and Smetsers, 1993; Plasmeijer and Van Eekelen, 1993). An argument of a type constructor is **propagating** if in the corresponding algebraic data type definition the corresponding variable occurs in the right-hand side on a position which is not part of a arrow type nor (part of) an instantiation of a non-propagating variable of another data type.

- The **propagation rule**: when an argument of a type is *propagating* and its uniqueness type variable is being instantiated with type `*`, the whole type must be attributed `*` as well.

Example (propagation rule): in the type `List a` given above the argument `a` is propagating. Now consider the following function:

```
hd (Cons x xs) = x
```

The type system will deduce the following most general type for `hd`

```
hd :: v:(List u:a)   -> u:a
```

The function hd can be called with a possibly spine unique list with can furthermore have unique elements. The resulting element is unique if the list had unique elements. The following more restricted types are also allowed (to understand the meaning of these types simply substitute the type attributes for the corresponding type variables in the definition above in an uniform way):

```
hd :: *(List  a)  ->  a          // a spine unique list with non-unique elements is demanded
hd :: *(List *a)  -> *a          // a spine unique list with unique elements is demanded
hd ::  (List  a)  ->  a          // the common (non-unique) list of (non-unique) elements
```

Counter example (invalid uniqueness type instance not obeying the propagation rule described above):

```
hd ::  (List *a)  -> *a          // a unique object is not allowed inside a non-unique one
```

So, one can define a function on a unique version via a proper instantiation of the automatically attached uniqueness type variables. not all imaginable unique instantiations of a classically defined data structures can be obtained in this way. Sometimes one has to slightly modify the classical type definition to make the proper unique instantiation possible.

Example (changing a type definition to make it possible to obtain certain unique instances): assume the following algebraic type definition and assume that one would like to have an instantiation of this data structure where only the left list is a spine unique one:

```
:: T a      =  C [a] [a]
```

There is now way to express this since the left list is not an argument of the type. So, one has to redefine `T` e.g.:

```
:: T l r    =    C l r
```

and instantiate this type with `T *[a] [a]`.

If one would like to define algebraic types with even more complicated or restricted uniqueness structures one has to explicitly specify the uniqueness type attributes in the algebraic data type definition. The semantic restrictions mentioned above have to be obeyed. Furthermore:

- All uniqueness type variables introduced on the left-hand side of a type definition must have different names.
- Uniqueness type variables used on the right-hand side of a type definitions must have been introduced on the left-hand side.
- All uniqueness type variables must have at least one occurrence on the right-hand side which is not part of a recursive instantiation of the algebraic type being defined.
- To simplify the type system it is *not* allowed to attribute arrow (function) types in type definitions with a uniqueness type.
- For record types, synonym types and abstract types the same restrictions hold as for algebraic types.
- When an abstract data type is specified its arguments are assumed to be propagating with a positive sign (see below).

Example (algebraic type with user defined uniqueness property):

```
:: u:T a    =  C u:[a] [a]
```

and for instance instantiate this type with `*T a` to obtain the required structure explained above.

The uniqueness type variable "`.`" and the uniqueness type "`*`" have a special meaning when being used in an algebraic data type definition. The "`.`" is introduced for notational convenience and is always uniformly instantiated with the attribute of the type itself. The "`*`" is reserved to define **essentially unique objects** which means that the type system will never allow such an object to be shared in any context.

Example (algebraic type with user defined uniqueness types):

```
:: T a      =  C .[a] [a]
```
which is equivalent to
```
:: u:T a    =  C u:[a] [a]
```
which is automatically expanded to
```
:: u:T v:a =   C u:[v:a] w:[v:a]
```

9.3 Typing functions and graphs with uniqueness attributes

The type system of Clean will infer and check the uniqueness type attributes *after* the classical types of functions and graphs have been inferred and approved (see 8.3). The uniqueness types inferred by the type system will in general add detailed uniqueness type information to the classical types.

Example (inferred uniqueness type). Consider the standard definition for the append function:

```
(++) infixr 0                                                // infix operator
(++) [hd:tl] list = [hd:tl ++ list]
(++) nil list        = list
```

The following type will be inferred:
`(++) infixr 0 :: [.a] u:[.a] -> u:[.a]`
Which is shorthand for:
`(++) infixr 0 :: x:[w:a] u:[w:a] -> u:[w:a]`
Indicating that the new list being constructed is unique if both arguments lists are, the result is spine unique only if the second argument list is spine unique.

As explained in Section 9.1, a function can demand a non-unique object, a unique object or a possibly unique object. Non-unique objects always stay non-unique. Unique or possibly unique objects can remain unique or possibly unique but they can also loose their unicity depending on how these objects are being used inside the function body. The uniqueness attributes of the result of a function will depend on the attributes of its arguments and how the result is constructed.

9.3.1 Uniqueness and sharing

The uniqueness type system uses a kind of reference count analysis called **sharing analysis**. The sharing analysis performed by the Clean system is input for the uniqueness type system to check uniqueness type consistency (see 9.3.5). The sharing analysis will mark each *reference* to an object (see 2.1) in a right-hand side of a function definition as **not-shared** (if this could be shown) or **shared** (otherwise). When a reference is marked shared by the sharing analysis the corresponding object will be typed non-unique. So, objects marked *shared* by the sharing analysis *cannot be typed unique*. When a reference is marked not-shared the value of the type attribute can become `*` (if the reference refers to a unique attributed object) but it can also become not unique or a uniqueness type variable (otherwise).

Multiple references to unique objects

The sharing analysis is a liberal system to make manipulation of unique objects as easy as possible. Unique objects do not have to be used in a pure "linear" way. Multiple references are possible to allow easy inspection (read access) of destructively updateable objects.

- When there is only *one reference* in the right-hand side to a certain object (the reference count of the object is locally one) the reference is marked as *not-shared*.
- *Cyclic structures* are marked as *shared*.
- A certain reference to an object will be marked as *not-shared* although there are temporally more references to this object *if* it can be shown at compile time that the *evaluation order* is such that the other references will vanish before the object is accessed via this particular reference. Otherwise such a reference will be marked shared.

So, the sharing analysis takes the evaluation order into account as far as this is fixed in a functional language. Another reference is not counted by the sharing analysis to determine if a certain reference is shared or not if it is an *alternative reference* or if it is an *observing reference*.

Example (multiple references to a unique object): The function F demands a spine unique list. And although there are three references to such a list in G the type system will approve the expression. The first reference to the list in the conditional is an observing reference. The other two are alternative references. Whenever F is called (in either then or else branch) only one reference to the list will be left.

```
F :: *[Bool] -> ...

G *[Bool] -> ...
G list = if (hd list) (F list) (F (tl list))

hd :: *[u:a] -> u:a
tl :: *[u:a] -> *[u:a]
```

An **alternative reference** is not taken into account because the reference will no longer exist when the choice between the alternatives has been made. An alternative reference is a reference to the object that will vanish because:

- it belongs to a different *rule* alternative then the reference to be marked;
- it belongs to a different *case* alternative then the reference to be marked;
- it belongs to the other part of a *conditional* (`if`) (the *then* part while the reference to be marked is part of the *else* part (or the other way around)).

An **observing reference** is not taken into account because it can be shown that the reference will no longer exist after the observation has taken place. Observing references are very important because they enable to *inspect values* stored in a *unique object* in such a way that one can still obtain a unique reference to the object (such that it can be destructively updated). An observing reference is a reference to the object that will vanish because:

a. the observation is performed *before* the reference to be marked as non-shared while
b. it is guaranteed that the reference to the object will *vanish*.

In Clean the following **property of the order of evaluation** is taken into account to determine observing references (see also 6.1). For each *rule* or *case* alternative:

+ for each non-variable part of a *pattern* the corresponding argument is evaluated to strong root normal form *before*
+ the *guard* (if present) is evaluated to a boolean *before*
+ the graphs of the topmost *strict let expression* are evaluated to strong root normal form *before*
+ the graphs of the *next* strict let expression are evaluated to strong root normal form *before*
+ in this way all strict let blocks are handled *before*
+ the *root expression* is evaluated to strong root normal form in which the first argument (the boolean expression) of a *conditional* (`if`) is evaluated *before* the *then* or *else* part.

A reference to an object `e::Te` in an observing expression `o::To` will *vanish* if one can deduce from the type `To` that e after evaluation cannot contain references to `e` anymore. This is the case if

- `To` is of polymorphic type `a` (as is the case with projection functions created via a pattern match, tuple/record/array selectors etc.) or
- `To` is a **plain basic type**, i.e. of type `Int`, `Real`, `Char` or `Bool` or
- `To` is of **plain algebraic type** T $a_1 \ldots a_n$, i.e. an algebraic type which only contains constructors, objects of plain basic type or objects of the same algebraic type T $a_1 \ldots a_n$.
    + `To` must furthermore be **hyper strict**, i.e. all its elements must be strict and
    + `To` must not be unifyable with any type in `Te`.

Example (observing a unique state): the contents of the unique state `abc` is observed in the pattern match before the state is overwritten in the function body.

```
Jsr :: InstrId *ABCState -> *ABCState
Jsr address abc=:{pc,cs}
   =  ABCState {abc & cs = CS_push pc cs, pc = address}
```

Example (observing a spine unique list): the unique list is observed in the let-block in which the nth element is taken out of the list.

```
Select :: n *[a] -> (*[a],a)
Select list n  =  let! y = NthElem n list
                  in (list,y)
                  where
                     NthElem :: Int [.a] -> a
                     NthElem 1 [x:xs]  =  x
                     NthElem n [x:xs]  =  NthElem (n-1) xs
```

Counter Example: suppose that one wants to extract two elements from the spine unique list. NthElem2 is not observing because one cannot tell from the type of NthElem2 whether a new spine unique list is returned or part of the original spine unique list. To make NthElem2 observing it should return the elements e.g. a strict tuple instead.

```
Select2 :: n *[a] -> ([a],a,a)
Select2 list n =  let! [y1,y2] = NthElem2 n list
                  in (list,y1,y2)
                  where
                     NthElem2 :: Int [.a] -> [.a]
                     NthElem2 1 [x,y:xs]  =  [x,y]
                     NthElem2 n [x:xs]    =  NthElem2 (n-1) xs
```

9.3.2 Meaning of the type attributes in the specification of a function type

The uniqueness type system is a **subtyping** system: the attribute unique (`*`) is regarded as being a sub-type of the attribute non-unique (`*` ≤ non-unique). So, the type system can **coerce** unique to non-unique. As with general subtyping, coercions are **contravariant** with respect to function types. Contravariance is indicated by the sign. The **sign** of an argument is determined by the **position** on which the argument occurs in the right-hand side of the type definition. For an algebraic data type, say `T` a_1 … a_n, that does not contain function types, all a_i are on a **positive position**. For an arrow/function type, say `a -> b`, however, `a` is on a **positive position** and `b` is on a **negative position**. For algebraic data types, say `T` a_1 … a_n, in which function types are contained each a_i can be on a positive position (the *sign* is **+**), on a negative position (the *sign* is **–**) or on a position which is both negative as well as positive (the *sign* is **+/–**) depending on the use of the corresponding variable in the right-hand sides of the algebraic data type. note that an argument which has a negative sign cannot be propagating.

- The coercion from unique to non-unique is only possible if the sign of the corresponding argument is *positive*.

Example (sign and propagation: `a` has sign + and is propagating, `b` has sign – and is not propagating):

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

The meaning of the uniqueness type attributes in the type specification of a function is different for positive (in a function argument) and negative (in a function result) positions.

When no uniqueness type attribute is specified explicitly the ***default*** **attribute non-unique** is taken by the type system (notice that there is no explicit notation for this attribute in Clean).

- For a (sub-) arguments on a positive position no type attribute means that the corresponding (sub-) argument is expected to be non-unique. When the function is actually applied with a unique typed (sub-) argument it will be **coerced** by the type system to a non-unique type.
- On a negative position the absence of a type attribute indicates that uniqueness of the corresponding result cannot be guaranteed. Such a result might be shared and therefore it cannot be given to a function demanding it as unique argument.

Example (the uniqueness type attribute of a classical type is by default not unique):

```
I :: a -> a
I x = x
```

When the **uniqueness type attribute** `*` is derived or specified explicitly the following holds.

- On a positive position it means that the corresponding (sub-) argument is required to be unique. The function can *only* be applied with a unique (`*`) typed (sub-) argument[1].
- On a negative position it indicates that uniqueness of the corresponding result is guaranteed.

Another valid restricted type for the identity function is the following type which indicates that the function can only be applied on a unique argument and that it will return a unique object:

```
I :: *a -> *a
I x = x
```

When a **uniqueness type *variable*** is derived or specified explicitly the following holds.

- On a positive position it means that the corresponding (sub-) argument can be unique *as well as* non-unique (this is also called **possibly unique**). Uniqueness type variables in the type specification of functions are very useful for the specification of uniqueness consistency between the arguments and the result of polymorphic functions.
- On a negative position a ***bound*** **uniqueness type variable** (i.e. a uniqueness variable which is introduced on a positive position) means that the uniqueness will depend on the uniqueness attribute of the corresponding argument(s) in the actual function call (see below).
  When ***free*** **uniqueness type variables** appear in the result type of a function it means that the corresponding result can be regarded as unique *as well as* non-unique. So, one can deduce from this that a *unique result* is returned.

Example (inferred uniqueness types of functions). The inferred most general type of the identity function is given below. It indicates that, when a (non-)unique argument is given to the identity function, the result will be (non-)unique as well (and of the same classical type). The previous types for I are restricted versions of this most general inferred type (uniformly instantiate `u` with non-unique or with `*`).

[1] Remark: In the classical type system sources of non-polymorphism are a user specified restricted type in the type specification of a function or denotations of objects that are predefined or user-defined in an algebraic data type. For uniqueness types the only source of non-polymorphism (read: `*`) is the specification of a restrictive type by the user or in a library (e.g. `FWriteC`).

```
I :: u:a -> u:a
I x = x
```

Shorthand notation in function type specifications

The **anonymous uniqueness type variable** "`.`" can be used in a function type instead of a uniqueness type variable in cases the name does not matter. Since each classical type variable has to be attributed uniformly, one can prefix each of them with a dot to indicate that they are actually attributed with a uniqueness type variable. For notational convenience one furthermore need only to specify the (possible) uniqueness of the innermost structure. The system will infer the propagation consequences for the surrounding structures.

Example (use of the anonymous uniqueness type variable).

```
I :: .a -> .a
I x = x

map :: (.a -> .b) [.a] -> [.b]
```
which is equivalent with
```
map :: (u:a -> v:b) w:[u:a] -> x:[v:b]
```

9.3.3 Typing curried functions

When a *curried function* is applied to a (possibly) unique element the resulting function type must be unique (a curried function in this aspect acts like a constructor). This behaviour is reflected in the **propagation property for curried functions**: the arrow type of a curried application of a function with at least one argument is attributed as follows:

- if one of the arguments has uniqueness attribute not being non-unique then the resulting arrow type must have attribute `*`, otherwise the resulting arrow type is attributed as being non-unique.
- it is not allowed to ever share on object of arrow type which is attributed with `*` (it is said to be **essentially unique**). This restriction implies that an essentially unique function cannot be coerced to a non-unique function type. *This has as consequence that an essentially unique function cannot be passed as argument to a function that will share the essentially unique function inside its function body.*

The propagation property reflects the fact that a curried function can be regarded as a kind of data structure.

Example (essentially unique functions cannot be coerced to a non-unique function). Consider again the inferred type of the function `map`.

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

The type specification shows that a non-unique function is demanded. This means that map cannot be called with an essentially unique function (i.e. a curried function which has "eaten" unique arguments). And indeed, one can see in the definition of `map` that its first argument becomes shared in the function body.

9.3.4 Typing functions with functions on constructor positions

NOT SUPPORTED IN CURRENT RELEASE.

9.3.5 Type consistency

The (uniqueness) type specifications of graphs and functions/operators are **inferred** at compile time by the **uniqueness type inferencing mechanism**. Explicit **uniqueness type specifications** are allowed to be more *restrictive* than the inferred types. In that case the restrictive type is used in all contexts.

A **uniqueness type** specification must be **correct**. It is assumed that the underlying classical type specification is already correct (see 8.3.5). So, only the attached uniqueness type attributes have to be taken into consideration for uniqueness type consistency. Each occurrence of a uniqueness type attribute is assumed to have a sign (see 9.3.2) and is propagating or not (see 9.2).

Each node is typed with respect to uniqueness by *correctly* instantiating uniqueness variables:

- All occurrences of a classical type variable must be attributed with the same uniqueness type attribute within its scope.
- For each argument of the node symbol, the type attribute has to be **sharing consistent** i.e. its attribute must be non-unique if the argument has been marked shared by the sharing analysis (see 9.3.1).
- The instantiated type (the **offered type**) must be **coercible** (with `*` ≤ u: ≤ non-unique) to the environment type of the node symbol (the **demanded type**).
  Coercions are defined as follows:
  - a non-arrow type `offered` is coercible to a type `demanded` if all the arguments, sub-arguments and results of the type constructors are coercible with respect to the sign of their position; for sign + this means that `offered` ≤ `demanded`, for sign - this means that `demanded` ≤ `offered` and for sign ± it means that both `offered` ≤ `demanded` and `demanded` ≤ `offered` (so, `demanded` = `offered`);
  - an arrow type `offered` is coercible to a type `demanded` if the type attributes of `offered` and `demanded` (so, `demanded` = `offered`) are the same and the arguments, subarguments and the result are coercible as well. Consequently, unique arrow types can never be shared in a context.
- When an argument of a type is **propagating** and its uniqueness type variable is being instantiated with type `*`, the whole type must be attributed `*` as well (see 9.2).
- Furthermore, if the node contains a curried application of a function symbol then the **propagation rule for curried functions** has to be taken into account (see 9.3.3).

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. For a description of the 0.8 system we refer to the Addison-Wesley book (Plasmeijer and Van Eekelen, 1993) or to the appendix of the "Gentle Intro in Clean" as included with the Clean 1.0 distribution.

Cleans' 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 easily be extended or modified 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 realized 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 10.6). One can switch between these applications (like in a multi-finder) and exchange information between them. Sub-applications can communicate via *files* (e.g. via copy-pasting), via *global states* interactions can have in common or via *message passing* (see 10.6).

It is in principle also possible to create sub-applications running on a different processor (see 10.7). 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.

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
```

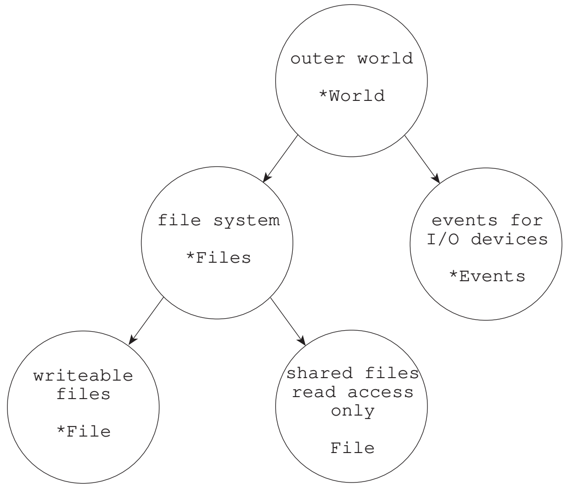
The world which is given to the initial expression is an *abstract data structure*, an **abstract world** of type `*World` which models **the concrete physical world** as seen from the program. The abstract world can in principle contain *anything* we want, anything what is of importance for a functional program to interact during execution with the concrete world. The world can be seen as a *state* and modifications of the world can be realized 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 independent 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 **event I/O system** (of type `*IOState local share`) in which 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 parametrized with the type of the *local process state* and the *shared process state* (see hereafter) because state transition functions working on these sates are stored in the `IOState` as well.

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 10.6 and 10.7 that a Clean application can consists 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 **shared process state** interactive processes which belong to the same group (see 10.6) can exchange information (e.g. to realize inter process communication via shared data such as clipboard copy-pasting between processes).

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 share                    // the unique state of an interactive process
   = { pLocal     :: local,              // the local (and private) data of the process
       pPublic    :: share,              // the data shared with other processes in the same group
       pFiles     :: !*Files,            // the unique state of the file system
       pIOState   :: !*IOState local share    // the unique state of the event I/O system
```

```
        }
```

Starting and stopping an interactive process

The first thing which generaly 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 - 10.7). These descriptions are stored by the process control function in the IOState.
- Now the process control function will repeatedly examen 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 on 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 on the process state after which the process control function (and the intercative process) terminates yielding the final process state.

So, in an **interactive process** state transition functions defined by the programmer are repeatedly applied to the initial process state until the final process state has been reached.

The function `OpenIO` is of following type.

```
OpenIO :: (IODef *local *share) (.l,.s) *World -> *World

:: IODef .l .s
   = {  ioDefInit     :: InitIO .l .s,  // initial actions process
        ioDefAbout    :: String         // name of the process
     }

:: InitIO .l .s  :==   [(PState .l .s) -> (PState .l .s)]
```

QuitIO, the state transition function which stops an interactive process has type:

```
QuitIO :: !*(IOState .l .s) -> *IOState .l .s
```

Example (a program just starting and stopping an interactive process doing nothing).

```
module StartAndStop

:: UnUsed = UnUsed

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 program 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 it 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 like, 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 these files have become garbage.
- When a writeable file becomes shared (loosing its uniqueness attribute) 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 `StdFileIO` (see ??).

Example (functions to open and close files). See also `StdFileIO`.

```
fopen   :: !String !Mode !*Files -> (!Bool,!*File,!*Files)  // writeable file
fclose  :: !*File !*Files -> (!Bool,!*Files)                // writeable file

sfopen  :: !String !Int !*Files -> (!Bool,!File,!*Files)    // 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 `"source"` to a file called `"dest"`). 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                   // Used to initialize unused settings

:: Local    :== UnUsed                  // Local program state not used
:: Share    :== UnUsed                  // Data sharing not used
:: *State   :== PState  Local Share     // Synonym type for State
:: *IO      :== IOState Local Share     // Synonym type for IOState

Start :: *World -> *World
Start world = OpenIO
                 {  ioDefInit    =  [DoCopyFile],
                    ioDefAbout   =  "Copying Process"} (UnUsed,UnUsed) world
```

```
DoCopyFile :: *State -> *State
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
   |  not sopen    =  abort "Source file could not be opened.\n"
   |  not dopen    =  abort "Destination file could not be opened.\n"
   |  not dclose   =  abort "Destination file could not be closed.\n"
   |  otherwise    =  files3
where
   (sopen,sfile,files1)        = sfopen source FReadData files
   (dopen,dfile,files2)        = fopen dest FWriteText files1
   (dclose,files3)             = fclose (CopyOneFile sfile dfile) files2

CopyOneFile :: File *File -> *File
CopyOneFile sfile dfile
   | not ok     =  dfile
   | otherwise =  CopyOneFile nsfile (fwritec char dfile)
where
   (ok,char,nsfile)   =  sfreadc sfile
```

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 10.6). A device can be composed of **device components** which on their turn can be refined further. For instance, each window can have **menu** (see 10.4.1), 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**. With such a predefined algebraic type 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:

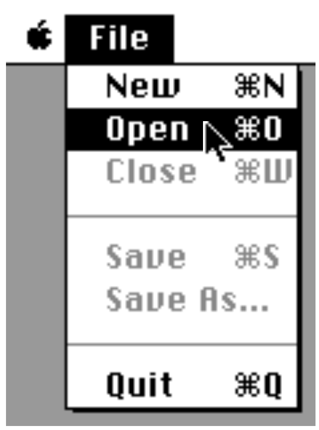
- a **constructor** with a meaningful name to indicate the desired device/device component (eg **`PullDownMenu`** indicates that a pull-down menu item is wanted);
- the definition of a very limited number of **non-optional attributes** (eg each pull-down menu must have a `Title` which is of type `String`);
- if applicable the definition of the **sub-components** (eg a menu can contain menu-items, a submenu 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 (eg 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);

Example (The predefined algebraic type MenuDef with which one can define a menu). Constructors are displayed **`bold`**, call back functions are in *`italic`*.:

```
:: MenuDef       *ps   =  Menu          Title [MenuElement ps]   [MenuAttribute ps]
:: MenuElement   *ps   =  SubMenuItem   Title [MenuElement ps]   [MenuAttribute ps]
                       |  MenuItem      Title                    [MenuAttribute ps]
                       |  MenuSeparator

:: MenuAttribute  *ps                              // Default:
   =  MenuId            Id                         // no Id
   |  MenuSelectState   SelectState                // menu(item) Able
   |  MenuShortKey      KeyCode                    // no KeyCode
   |  MenuAltKey        Index                      // no AltKey
   |  MenuMarkState     MarkState                  // NoMark

   // Attributes ignored by (sub)menus:

   |  MenuFunction      (ps -> ps)                 // Identity function
   |  MenuModsFunction  (Modifiers -> ps -> 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 `MenuDef` 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. Notice that `QuitIO` is used as call-back function. 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 yield an instance of `MenuDef` will do.



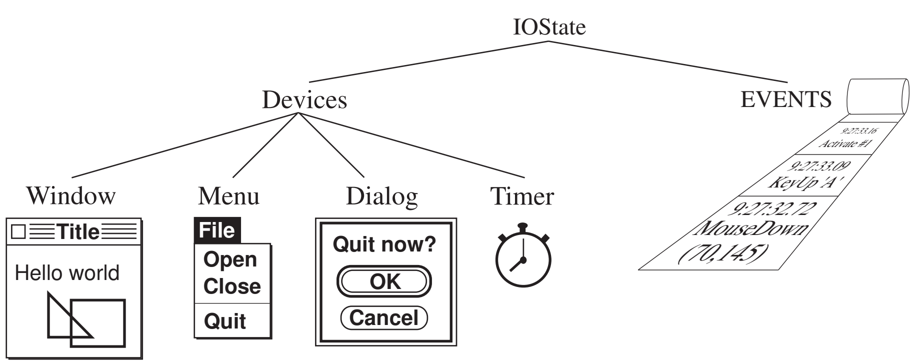
```
Menu "File"
[  MenuItem "New"         [  MenuFunction    new,
                             MenuShortKey    'n'   ],
   MenuItem "Open…"       [  MenuFunction    open,
                             MenuShortKey    'o'   ],
   MenuItem "Close"       [  MenuFunction    close,
                             MenuShortKey    'w',
                             MenuSelectState Unable   ],
   MenuSeparator,
   MenuItem "Save"        [  MenuFunction    save,
                             MenuShortKey    's',
                             MenuSelectState Unable   ],
   MenuItem "Save As…"    [  MenuFunction    saveAs,
                             MenuSelect      Unable   ],
   MenuSeparator,
   MenuItem "Quit"        [  MenuFunction    quit
                             MenuShortKey    'q'   ]
] []
```

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 without effecting existing programs.

Opening abstractdevices and application of call-back functions

For each abstract device a special *open function* has been predefined. It is a state transition function (defined on the process state) which, when it is applied (on the current process state), stores the algebraic description of the device into the IOState and it activates the correponding *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 can be attached to a given window. Each menu has a private state again which can be used to remember specific menu settings. So each device / device component can be regarded as a kind of object containing information which is relevant for that object and which can only be accessed by that object.

```
OpenMenus :: ![MenuDef (PState local share)]
                                !(IOState local share) -> IOState local share
```

The function `OpenIO` will recursively examen 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 in on 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 .s) -> *PState .l .s
```

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 a call-back function has reached head normal form, control (and the states) is given back such that the next call-back function can be determined (with the new states as returned by the previous call-back function called) 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).

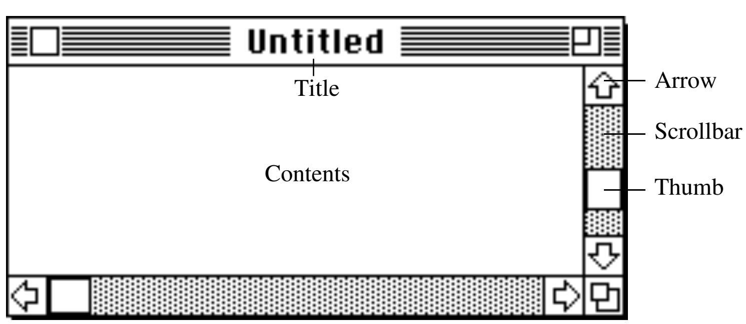
10.4 Graphical user interfaces

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 **menu** (see 10.4.2), 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 lay-out of controls and windows (see 10.4.6).

10.4.1 Windows, dialogues and notices

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`. Each window has a private state which can be used to remember specific window settings. The window-id which is returned can be used to change window settings dynamically (e.g. to close a window, see further `StdWindow` in the appendix).

```
OpenWindow        :: !(WindowDef (PState local share))
                          !(PState local share) -> PState local share
OpenModalWindow   :: !(WindowDef (PState local share))
                          !(PState local share) -> PState local share
CloseWindow       :: !Id !(IOState local share) -> IOState local share

:: WindowDef ps
   =  DialogWindow Title Id             [ControlDef ps] [WindowAttribute ps]
   |  Window       Title PictureDomain  [ControlDef ps] [WindowAttribute ps]
```

Windows can be of fixed size (**`DialogWindow`**) or of variable size (**`Window`**). There are two special windows which can be opened. When a **modal window** is opened (`OpenModalWindow`) one is forced to perform interactions with that window until this window is 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) keyboard keys being pressed when the window is active, what to do when a window is closed etc. Below we explain keyboard (10.4.2) and mouse handling (10.4.3).

Inside a window controls like buttons and the like 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:
   = …
   | WindowKeys   SelectState (KeysFunction  ps)  // no keyboard

:: SelectState  =  Able | UnAble
```

In this way one can define a function to handle the response to keyboard events. 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 time.

```
:: KeysFunction ps :==   KeyboardState -> ps -> ps

:: KeyboardState   :==   (!KeyCode, !KeyState, !Modifiers)
:: KeyCode         :==   Char
:: KeyState        =     KeyUp | KeyDown | KeyStillDown
:: Modifiers       :==   (Bool,Bool,Bool,Bool)
                                       //(Shift,Option,Command,Control)
```

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 `Mousefunction` defines the response of the window to mouse events in the window's contents (see also controls in 10.3.6 to handle the clicks on bottons 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 a release of the mouse button and which modifier keys (shift, option etc.) were being held down.

```
:: MouseFunction ps   :==   MouseState -> ps -> ps

:: MouseState      :==   (!MousePosition, !ButtonState, !Modifiers)
:: MousePosition   :==   (!Int, !Int)
:: ButtonState     =     ButtonUp | ButtonDown | ButtonDoubleDown |
                         ButtonTripleDown | ButtonStillDown
```

10.4.4 Writing and drawing to a window

The library module `StdWindow` contains several functions with which one can draw in a given window, for example:

```
DrawInWindow       :: Id [DrawFunction] *(IOState .l .s) -> *IOState .l .s

:: DrawFunction    :==   Picture -> Picture
```

These functions take a list of `DrawFunction`s and apply these to the indicated window in order of appearance in the list. A large set of functions has been predefined in the module `StdPicture`, such as:

```
MovePenTo    :: !Point  !Picture -> Picture
DrawString   :: !String !Picture -> Picture
```

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) : `MovePenTo` and `DrawString` are predefined functions defined in `StdPicture`.

```
DrawStringInWindow:: Id Point String  *(IOState .l .s) -> *IOState .l .s
DrawStringInWindow id pos string io
   = DrawInWindow id [  MovePenTo   pos,
                        DrawString  string   ] io
```

10.4.5 Menus

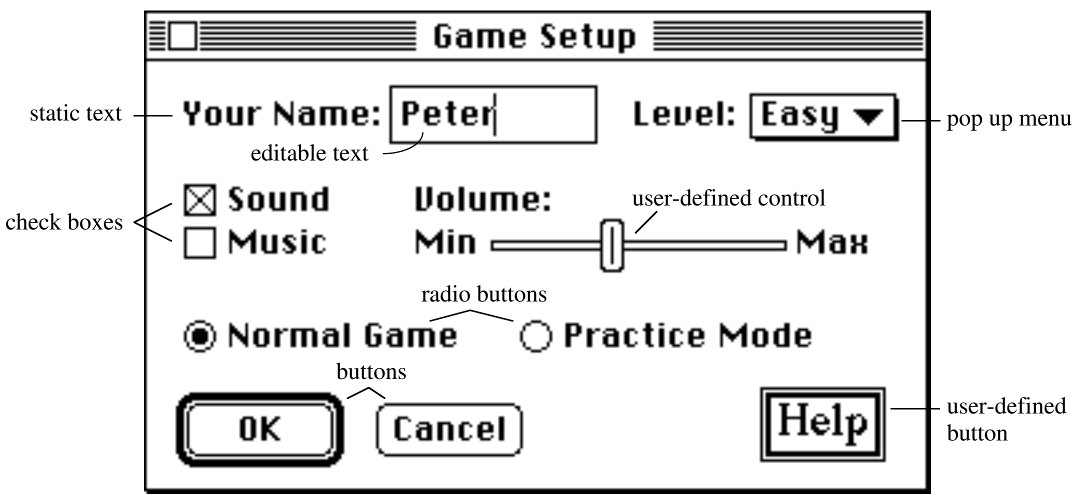
With the functions defined in library module `StdMenu` one can attach a menu to a given window. The **menu device** (see `StdMenuDef`) conceptualizes choosing from a menu of available commands. A menu device can contain components like **pull-down menus** each containing sub-components like a **menu-item**, **menu radio-item** or a **sub-menu**. See further the intro of 10.3.

```
OpenMenus :: ![MenuDef (PState local share)]
                              !(IOState local share) -> IOState local share
```

10.4.6 Controls

Besides that one can draw pictures and write text in a window one can put controls like bottons and the like into a given window. Controls can be **editable text fields** (the displayed text can be changed by the user or the program during the interaction), **static text fields** (the text cannot be changed), **pop-up menus** (a number of options of which only one is valid and displayed at a time), **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), **buttons**, **final buttons** (default bottons) and even **user-defined controls** (controls of which the look and feel can be specified by the programmer, see `StdControls`). A group of controls can be combined to form a unit which is called a **CompoundControl**.

Figure (controls in a window).




```
:: ControlDef ps
   =  RadioControl     TextLine MarkState                     [ControlAttribute ps]
   |  CheckControl     TextLine MarkState                     [ControlAttribute ps]
   |  PopUpControl     [PopUpItem ps] Index                   [ControlAttribute ps]
   |  SliderControl    Direction Length SliderState (SliderAction ps)
                                                              [ControlAttribute ps]
   |  TextControl      TextLine                               [ControlAttribute ps]
   |  EditControl      TextLine Width NrLines                 [ControlAttribute ps]
   |  ButtonControl    TextLine                               [ControlAttribute ps]
   |  CustomButtonControl  Size (ControlLook ps)              [ControlAttribute ps]
   |  CustomControl    Size (ControlLook ps) CustomState      [ControlAttribute ps]
   |  CompoundControl [ControlDef ps]  (ControlLook ps)       [ControlAttribute ps]
```

Defining the position of a Control (also applicable for Windows)

```
:: ControlAttribute ps                                    // Default:
```

```
…
  | ControlPos          ItemPos                    // RightTo previous
```

In the attributes of a 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 is created as follows:

```
:: ItemPos    :==   (ItemLoc, ItemOffset)
:: ItemLoc    =     LeftTop | RightTop    | LeftBottom | RightBottom
              |     Left    | Center      | Right
              |     LeftOf  Id   | RightTo  Id
              |     Above   Id   | Below    Id
:: ItemOffset :==      (!Int,!Int)
```

An 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, right-aligned or centred, *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 items defined earlier, left-aligned. When an item is placed below an item that is centred resp. right-aligned this item will also be centred resp. right-aligned. When an item is placed right to a centred item, these items are centred together.
- With the `ItemOffset` an item can be shifted relatively on the indicated position with the (possibly negative) offset specified.

Defining the look of a Control

The look of controls is defined as follows: *system* controls have a predefined look (these are **Radio**-, **Check**-, **PopUpList**-, **Text**-, **EditText**-, *system* button look **Button**-, and **SliderControl**s). *Customized* controls (*custom* button look **Button**- and **CustomControl**s) are drawn by their program defined `ControlLook` functions. The `ControlLook` function of a `CompoundControl` is drawn first, after which the looks of its control elements in *left-to-right* and *depth-first* order follow. The backgrounds of controls is *not erased* by the system, thus enabling controls to define local background textures.

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                                // (0,0)

:: 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 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 **`EditText`**-, **`Slider`**-, **`Custom`**-, and **`CompoundControls`**. Fixed size controls are all other controls. The **`ControlResize`** attribute is ignored for fixed size controls. 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 for which no **`ControlResize`** attribute has been defined. A resizeable control of size *curCSize* with attribute **`ControlResize`** `f` obtains the new size `f curCSize curWSize newWSize = (x,y)`. If the `ControlResizeFunction` yield a value smaller than the minimim size the control will be **hidden**, otherwize its new size will be `(x,y)`. When all sizes have been recomputed for all controls then the layout will be recalculated. The result of this recalculation should be 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 synchronized: a call-back function can be evaluated every time a certain time interval has passed. 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 whenever no other event is generated. Several timers can be installed. Each timer has a private state which can be used to remember timer specific settings.

```
OpenTimer    :: !(TimerDef (PState local share))
                          !(IOState local share) -> IOState local share
CloseTimer   :: !Id !(IOState local share) -> IOState local share

:: TimerDef ps      =  Timer TimerInterval [TimerAttribute ps]
:: TimerInterval    :== Int

:: TimerAttribute *ps                                  // Default:
   =  TimerId          Id                              // no Id
   |  TimerSelect      SelectState                     // timer Able
   |  TimerFunction    (NrOfIntervals -> ps -> ps)     // f _ x = x
:: NrOfIntervals    :== Int
```

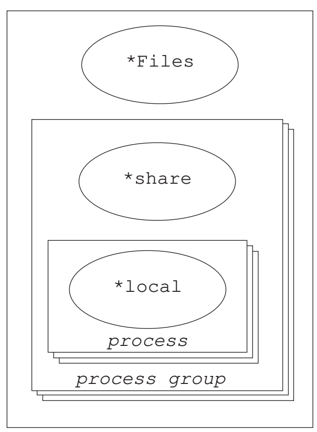
The `TimerFunction` is the type of the call-back function which is called each time the specified timer interval has passed. The `NrOfIntervals` parameter contains the number of times the interval has passed since the last time the timer function was called. The problem is the program might be busy with the evaluation of a call-back function. Each call-back function is 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. 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 `TimerState` argument of the timer function will then always be one.

10.6 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 consists of several interactive processes which can be created dynamically. Each process defines its own user interface, timers etcetera with correponding 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 (but see also 107). 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).



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 realize that Clean applications can in 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 ps :== [ps -> ps]

OpenIO  :: !(IODef *local *share) (*local,*share) !*World -> *World
NewIO   :: !(IODef *local *share) (*local,*share)
              !(IOState local` share`) -> IOState local` share`
ShareIO :: !(IODef *local *share) *local
```

```
                !(IOState local` *share) -> IOState local` *share
```

Initially a 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`. With the function `NewIO` any process can create a new group initially consisting of one interactive process.

The following functions are planned: The operations `NestNewIO` and `NestShareIO` define the *nested form* of `NewIO` and `ShareIO`. In both cases the currently active process will be *hidden* and replaced by the argument process. Other processes will still continue to be evaluated. The new process can spawn new processes as well. Only when this process has terminated, its parent process is *shown* again, and the function terminates with the final value of the local program state of the terminated process. Observe that it can be the case that there are processes around that have been created by the child process. So, the difference between the two operations is that `NestNewIO` creates a new group of processes for the new process, while `NestShareIO` adds the new process in the group of the father process.

10.6.1 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**. Receivers can be created and disposed of dynamically. Message passing is *polymorphic*: the content of a message can be *any* typeable expression. 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.

```
OpenReceiver       :: !(ReceiverDef mess (PState local share))
                      !(IOState local share) -> (!IOId mess, !IOState local share)
CloseReceiver      :: !(IOId mess) !(IOState local share) -> IOState local share

  :: ReceiverDef        mess *ps
     =  Receiver [ReceiverAttribute mess ps]

  :: ReceiverAttribute mess *ps                      // Default:
     =  ReceiverSelect     SelectState               // receiver Able
     |  ReceiverFunction  (mess -> ps -> ps)         // f _ x = x
```

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 (the message type `m` can of course be of any (polymorphic) type). The returned `MessageId` is important because one needs it when one wants to send a message to this receiver. Sending a message actually means that an event of appropriate type is raised and put into the event queue in the IOState. Thanks to the MessageId it is guaranteed that only correctly typed messages can be send and that they can only go to the correct receiver.

```
ASyncSend    :: !(IOId mess) mess !(IOState local share) -> IOState local share
```

Interactive processes can send messages in a *asynchronous* or *synchronous* way (with `ASyncSend` and `SyncSend` (*not yet implemented*) respectively). Both functions require the receivers identification value of type `MessageId 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.

Using `SyncSend` the sending interactive process *blocks* 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. One should also be aware that processes can send messages to blocked processes such that a **deadlock** is possible.

10.6.3 Remote procedure calls

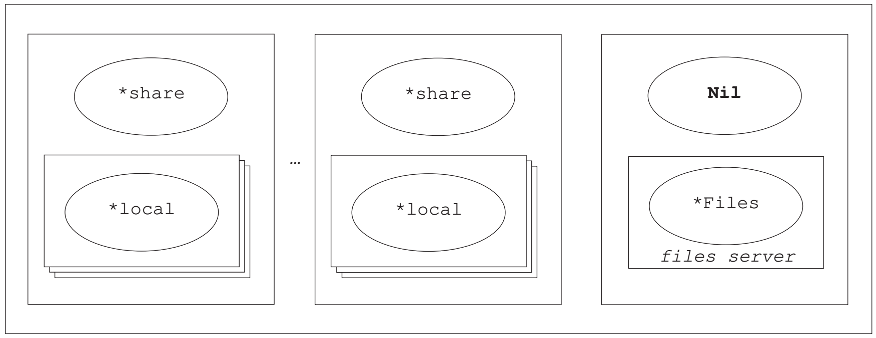
It is *not yet possible* (since it is not yet implemented in the current release) to define a special kind of interactive process, the **Remote Procedure Call process** (**RPC process**). A new RPC process be the first process of new process group (`NewRPC`) or it can be a member of the current process group (`ShareRPC`). Special about the process is that it contains a special receiver, which can be addressed via a **Remote Procedure Call** (`SendRPC`). A RPC process of type `in out` is an interactive process that for every `in` message generates one `out` message. Analogous to opening a receiver, the creation of a RPC process yields an identification value which is type parameterized with the `in` and `out` message types.

```
:: RPCDef in out ps  =  {  rpcFunction :: (in,ps) -> (out,ps),
                           rpcInitIO   :: InitIO ps  }
:: RPCId in out

NewRPC  :: (RPCDef in out *(PState .l .s)) (.l,.s)
                           *(IOState .m .t)  -> (RPCId in out, *IOState .m .t)
ShareRPC:: (RPCDef in out *(PState .l .s)) .l
                           *(IOState .m .s)  -> (RPCId in out, *IOState .m .s)
SendRPC :: (RPCId in out) in
                           *(PState .l .s) -> (SuccessOrFail out,*PState .l .s)
```

Using SendRPC the sending interactive process *blocks* until the indicated receiver accepts the message. Programmers must be aware that SendRPC involves a **context-switch**. In case that there are several interactive processes in the process group it can be possible that after a SendRPC the shared process state component may be changed by another process in the group. One should also be aware that processes can remotely call processes which are blocked such that a **deadlock** is possible.

10.7 Distributing executing communicating processes

In a distributed implementation interactive processes are implemented as parallel reduction processes. In Clean process belonging to the same process group can be created on a different processor. Data sharing is not possible.



NOT SUPPORTED IN CURRENT RELEASE.

11

Defining macros

11.1 Defining Macro's

In this chapter macros are treated. Macro's 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 Macro's

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 (parameterized) macros are used instead of non-recursive functions.

MacroDef	=	*FunctionSymb* {*Variable*} **:==** GraphExpr ;
	|	**(** *FunctionSymb* **)** [Fix][Prec] ;
		(*FunctionSymb* **)** {*Variable*} **:==** GraphExpr ;

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 mutually dependent nor recursive.
- Macro applications are not allowed to be curried.

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 []        = []
```

The expansion of the macros takes place before type checking. Type specifications of macro rules is not possible.

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            = DefModule
                  | ImplModule
DefModule         = DefModuleHdrDef {ImportDef} {DefModDef}
ImplModule        = ImplModuleHdrDef {ImportDef} {ImplModDef}

DefModuleHdrDef   = definition module ModuleSymb ;
                  | system module ModuleSymb ;
ImplModuleHdrDef  = [implementation] module ModuleSymb ;
```

- 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 Kind of definitions allowed in modules

Most definitions can be given both in implementation as well as in definition modules. However, *functions* and *graphs* can only be defined in *implementation* modules whereas *abstract types* can only be defined in *definition* modules.

```
ImplModDef          = TypeDef
                    | OverloadingDef
                    | FunctionDef
                    | GraphDef
                    | MacroDef
DefModDef           = TypeDef
                    | AbstractTypeDef
                    | OverloadingDef
                    | FunctionTypeDef
                    | 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*	|	*SelectorVariable*	|	*MacroSymb*
	|	*TypeSymb*	|	*ConstructorSymb*	|	*FieldSymb*

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 of a definition module that are not abstract type definitions can be repeated in the corresponding implementation module without any harm or change in meaning. In that case the specifications in the implementation module have to be exactly the same as the ones in the corresponding definition module, including annotations and attributes.
- An *abstract type definition* in a definition module is indicated by specifying the left-hand side graph of its type rule. In the corresponding implementation module the abstract type *has to be defined again* but then the definition has to 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* symbol can explicitly be defined in a definition module by the specification of its type. The types of the arguments and of the result of the function symbol must have been defined in the definition module as well.
- A *graph* symbol can explicitly be defined in a definition module by the specification of its type. Its type must have been defined in the definition module as well.

- All symbols used in a *macro* definition must be defined as well.
- Definitions introduced in a definition module have to be repeated in the corresponding 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 summarized (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	
`String`	n	16 + 4*ciel(n/4)	16 + 4*ciel(n/4)	
`Small Record`	n	4 + Σsize elements	Σsize elements	total length≤12
`Large Record`	n	8 + Σsize elements	Σsize elements	
`Tuple`	2	12	Σsize elements	
	>2	8 + 4*n	Σsize 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 a string or 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 optimization 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. We plan to incorporate this a one of the next versions 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 optimizations performed by a compiler. To avoid code explosion a compiler will generally not substitute big functions. Macro’s 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 parametrized 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 reults 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 1.0 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.5) and types (Section A.6-8) is presented. Overloading is treated in Section A.8. These sections have some production rules in common which are collected in Section A.10.
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** chicago
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 their definition in Section 3.4)

A.1 Clean program

CleanProgram	=	{Module}+
Module	=	DefModule
	|	ImplModule
DefModule	=	DefModuleHdrDef {ImportDef} {DefModDef}
ImplModule	=	ImplModuleHdrDef {ImportDef} {ImplModDef}
DefModuleHdrDef	=	**definition module** *ModuleSymb* ;
	|	**system module** *ModuleSymb* ;
ImplModuleHdrDef	=	[**implementation**] **module** *ModuleSymb* ;
ImportDef	=	**import** {*ModuleSymb*}-*list* ;
	|	**from** *ModuleSymb* **import** {ImportSymbols}-*list* ;
ImportSymbols	=	*FunctionSymb*

```
                     | SelectorVariable
                     | MacroSymb
                     | TypeSymb
                     | ConstructorSymb
                     | FieldSymb

DefModDef            = TypeDef
                     | AbstractTypeDef
                     | OverloadingDef
                     | FunctionTypeDef
                     | MacroDef
ImplModDef           = TypeDef
                     | OverloadingDef
                     | FunctionDef
                     | GraphDef
                     | MacroDef
```

A.2 Function definition

```
FunctionDef          = [FunctionTypeDef] DefOfFunction

DefOfFunction        = {FunctionAltDef}+
FunctionAltDef       = FunctionSymbol {Pattern}
                       {[| Guard] =[>] FunctionBody}+
                       [LocalFunctionAltDefs]

Pattern              = [Variable =: ] BrackPattern
BrackPattern         = SimplePattern
                     | PatternVariable
                     | BasicValuePattern
                     | ListPattern
                     | TuplePattern
                     | RecordPattern
                     | ArrayPattern
                     | (GraphPattern)
GraphPattern         = ConstructorSymbol {Pattern}
                     | GraphPattern ConstructorSymb GraphPattern
                     | Pattern

SimplePattern        = ConstructorSymbol

PatternVariable      = Variable
                     | _

BasicValuePattern    = BasicValue

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

TuplePattern         = (GraphPattern, {GraphPattern}-list)

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

ArrayPattern         = {:{GraphPattern}-list:}
                     | {:{ArrayIndex = GraphPattern}-list:}

Guard                = BooleanExpr

FunctionBody         = {StrictLet}+
                       RootExpression ;
```

```
                        [LocalFunctionDefs]

StrictLet             = let! { {GraphDef}+ } in

RootExpression        = GraphExpr

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

A.3 Graph definition

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

A.4 Graph expression

```
Graph                 = [Process] GraphExpr

GraphExpr             = Application
                      | CaseExpr
                      | LambdaAbstr

Application           = {BrackGraph}+
                      | GraphExpr OperatorSymbol GraphExpr

BrackGraph            = NodeSymbol
                      | GraphVariable
                      | BasicValue
                      | List
                      | Tuple
                      | Record
                      | RecordSelection
                      | Array
                      | ArraySelection
                      | (GraphExpr)

GraphVariable         = Variable
                      | SelectorVariable

BasicValue            = IntDenot
                      | RealDenot
                      | BoolDenot
                      | CharDenot
                      | StringDenot

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

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

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

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

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

LambdaAbstr         = \ {Pattern}+ -> GraphExpr

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

A.5 Macro definition

```
MacroDef            = FunctionSymb {Variable} :== GraphExpr ;
                    | (FunctionSymb) [Fix][Prec] ;
                      (FunctionSymb) {Variable} :== GraphExpr ;
```

A.6 Type definition

```
TypeDef             = AlgebraicTypeDef
                    | RecordTypeDef
                    | SynonymTypeDef

AlgebraicTypeDef    = ::TypeLhs = ConstructorDef {| ConstructorDef} ;
RecordTypeDef       = ::TypeLhs = {{FieldSymbol :: [Strict]Type}-list} ;
SynonymTypeDef      = ::TypeLhs :== Type ;

TypeLhs             = [UnqTypeAttrib] TypeSymb {TypeVarDef}
UnqTypeAttrib       = *
                    | UniqueTypeVariable:
                    | .

TypeVarDef          = [E.] [UnqTypeAttrib] TypeVariable

ConstructorDef      = ConstructorSymb {[Strict] BrackType}
                    | (ConstructorSymb) [Fix][Prec] {[Strict] BrackType}
Fix                 = infixl
                    | infixr
                    | infix
Prec                = Digit
Strict              = !

Type                = [UnqTypeAttrib] TypeConstructor {BrackType}
                    | BrackType
TypeConstructor     = TypeSymb
                    | (->)
                    | []
                    | ({,}+)
                    | {::}
BrackType           = [UnqTypeAttrib] SimpleType
SimpleType          = TypeVariable
```

```
                   | BasicType
                   | PredefAbstrType
                   | ListType
                   | TupleType
                   | ArrayType
                   | ArrowType
                   | (Type)

BasicType          = Int
                   | Real
                   | Char
                   | Bool
                   | String
PredefAbstrType    = World
                   | File
                   | ProcId
                   | Void
ListType           = [Type]
TupleType          = ([Strict] Type, {[Strict] Type}-list)
ArrayType          = {:[Strict] Type:}
ArrowType          = ({BrackType}+ -> Type)
```

A.7 Abstract type definition

```
AbstractTypeDef    = ::TypeLhs ;
```

A.8 Type specification of function

```
FunctionTypeDef    = FunctionSymb :: FunctionType ;
                   | (FunctionSymb) [Fix][Prec] [:: FunctionType ] ;
FunctionType       = [{[Strict] BrackType}+ ->] Type [ClassContext]
ClassContext       = | ClassSymb TypeVariable {& ClassSymb TypeVariable }
```

A.9 Defining overloaded functions, type classes, constructor classes

```
OverloadingDef     = OverloadedFunctionTypeDef
                   | TypeClassDef
                   | OverloadedFunctionInstanceDef
OverloadedFunctionTypeDef
                   = overload FunctionSymb TypeVariable :: FunctionType ;
                   | overload (FunctionSymb) [Fix][Prec] TypeVariable :: FunctionType ;
OverloadedFunctionInstanceDef
                   = instance FunctionSymb [BrackType[ClassContext]][=FunctionSymb] ;
                   | instance FunctionSymb [BrackType[ClassContext]]
                     DefOfFunction ;
TypeClassDef       = class ClassSymb TypeVariable
                     [where] { {ClassMemberDef}+ }
                     [instances {[BrackType[ClassContext]]}-list ;
ClassMemberDef     = FunctionTypeDef [MacroDef]
                   | FunctionSymb ;
TypeClassInstanceDef
                   = instance ClassSymb [BrackType[ClassContext]]
                     [where] { {DefOfFunction}+ }
```

A.10 Common definitions

```
NodeSymbol         = FunctionSymbol
                   | ConstructorSymbol
```

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

ArrayExpr           = GraphExpr
BooleanExpr         = GraphExpr
IntegerExpr         = GraphExpr
ListExpr            = GraphExpr
ProcIdExpr          = GraphExpr
RecordExpr          = GraphExpr
```

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 funtions 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 Cleans' Standard Environment

```
system module StdEnv

import
   StdClass,         // imports StdOverloaded and the defs for basic types

   StdList,
   StdClassList,
   StdTuple,
   StdArray,
   StdCharList,
   StdFunc,
   StdMisc,
   StdFile
```

B.1.1 Operations on values of basic type

StdClass: predefined classes

```
definition module StdClass

import StdOverloaded
import StdBool, StdInt, StdReal, StdChar, StdString

class PlusMin a  where +; -; zero

class MultDiv a  where *; /; one

class Arith a    where +; -; zero; *; /; one; abs; sign; ~

class Enum a     where <; inc; +; -; zero

class Eq a
where

   ==

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


class Ord a
where

   <

   (>)  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

class ToInt          a  where toInt
class ToChar         a  where toChar
class ToBool         a  where toBool
class ToReal         a  where toReal
class ToString       a  where toString

class FromInt        a  where fromInt
class FromChar       a  where fromChar
class FromBool       a  where fromBool
class FromReal       a  where fromReal
class FromString     a  where fromString
```

StdOverloaded: predefined overloaded operations

```
definition module StdOverloaded

overload (+)  infixl 6  a  :: !a !a -> a          // Add arg1 to arg2
overload (-)  infixl 6  a  :: !a !a -> a          // Subtract arg2 from arg1
overload zero           a  :: a                   // Zero (unit element for addition)

overload (*)  infixl 7  a  :: !a !a -> a          // Multiply arg1 with arg2
overload (/)  infix  7  a  :: !a !a -> a          // Divide arg1 by arg2
overload one            a  :: a                   // One (unit element for multiplicati

overload (^)  infixr 8  a  :: !a !a -> a          // arg1 to the power of arg2
```

```
overload inc            a  :: !a    -> a            // Increment with 1
overload dec            a  :: !a    -> a            // Decrement with 1
overload abs            a  :: !a    -> a            // Absolute value
overload sign           a  :: !a    -> Int          // 1 (pos value) -1 (neg value) 0 (if
overload ~              a  :: !a    -> a            // -a1

overload (==) infix  2  a  :: !a !a -> Bool         // True if arg1 is equal to arg2

overload (<)  infix  2  a  :: !a !a -> Bool         // True if arg1 is less than arg2

overload toInt          a  :: !a    -> Int          // Convert into Int
overload toChar         a  :: !a    -> Char         // Convert into Char
overload toBool         a  :: !a    -> Bool         // Convert into Bool
overload toReal         a  :: !a    -> Real         // Convert into Real
overload toString       a  :: !a    -> String       // Convert into String

overload fromInt        a  :: !Int  -> a            // Convert from Int
overload fromChar       a  :: !Char -> a            // Convert from Char
overload fromBool       a  :: !Bool -> a            // Convert from Bool
overload fromReal       a  :: !Real -> a            // Convert from Real
overload fromString     a  :: !String -> a               // Convert from String

overload #              a  :: !a                -> Int   // Number of elements in arg
overload ##             a  :: !a                -> Int   // Upperbound of arg
overload (%)   infixl 9 a  :: !a !(!Int,!Int)   -> a     // Slice a part from arg1

overload (+++) infixr 0 a  :: !a !a -> a            // Append args
```

StdBool: operations on Booleans

```
system  module  StdBool

import   StdOverloaded

instance ==          Bool

instance toBool      Bool
instance toString    Bool

instance fromBool    Bool
instance fromBool    String

// Additional Logical Operators:

not            :: !Bool       -> Bool  // Not arg1
(||)  infixr 0 :: !Bool Bool  -> Bool  // Conditional or  of arg1 and arg2
(&&)  infixr 1 :: !Bool Bool  -> Bool  // Conditional and of arg1 and arg2

// Miscellaneous:

otherwise  :== True                    // To be used in guards
```

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 inc        Int
instance dec        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    String

// 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 0 :: !Int !Int   -> Int   // Bitwise Or of arg1 and arg2
(bitand) infix 1 :: !Int !Int   -> Int   // Bitwise And of arg1 and arg2
(bitxor) infix 0 :: !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
```

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 inc        Real
instance dec        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   String

// 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
```

StdChar: operations on Characters

```
system  module  StdChar

import   StdOverloaded

instance ==         Char

instance <          Char

instance toInt      Char
instance toChar     Char
instance toString   Char

instance fromChar   Int
instance fromChar   Char
instance fromChar   String

// Additional convertions:

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
```

StdString: operations on Strings

```
system module StdString

import   StdOverloaded

instance ==           String

instance <            String

instance toString     String
instance toInt        String
instance toReal       String

instance fromString   String

instance #            String
instance ##           String
instance %            String

instance +++          String

// additional operator

(!!) infixl 9 :: !String !Int  -> Char            // select i-th element (0 <= i <= ##a
(:=) infixl 9 :: !String !(!Int,!Char) -> String  // update i-th element
```

B.1.2 Operations on predefined abstract types

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 #    [a]
instance ##   [a]
instance %    [a]

// List Operators:

(!)   infixl 9 :: [.a] Int -> .a                  // Get nth element of the list
(++)  infixr 0 :: ![.a] u:[.a] -> u:[.a]          // Append args
flatten        :: ![.[a]] -> [a]                  // e0 ++ e1 ++ ... ++ e##
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 lis
drop       :: Int !u:[.a] -> u:[.a]               // Drop first arg1 elem from the l
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 satisfyir
insert     :: (a a -> .Bool) a !u:[a] -> u:[a]    // Insert arg2 when pred arg2 elem
```

```
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..##list]
repeatn    :: !.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)...e##)
foldr      :: (.a -> .(.b -> .b)) !.b ![.a] -> .b // op e0 (op e1(...(op r e##)...)
scan       :: (a -> .(.b -> a)) a ![.b] -> .[a]   // [r,op r e0,op (op r e0) e1,...

// On Booleans

and        :: ![.Bool] -> Bool                      // e0 && e1 ... && e##
or         :: ![.Bool] -> Bool                      // e0 || e1 ... || e##
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
```

StdClassList: operations on Lists requiring classes

```
definition module StdClassList

from StdClass import Eq, Ord, PlusMin, Arith, FromInt

// 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 to a sortec

// 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]

// When addition is defined on list elements

sum              :: !.[a] -> a  | PlusMin a          // sum, sum [] = zero

// When multiplication and addition is defined on list elements

prod             :: !.[a] -> a  | Arith a            // product, prod [] = one
avg              :: !.[a] -> a  | Arith a            // average, avg [] gives error!
```

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
```

StdArray: operations on Arrays

```
system  module  StdArray

import StdOverloaded

instance #     {:a:}
instance ##    {:a:}
instance %     {:a:}

createArray  :: !Int a            -> *{:a:}   // {: arg2   \\ i <- [0..##arg1] :}
mkArray      :: !Int (Int -> a)   -> *{:a:}   // {: arg2 i \\ i <- [0..##arg1] :}
```

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 arg
rjustify   :: !.Int ![.Char] -> [Char]   // Right justify [Char] in field with width ar

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
```

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 (f x)
until      :: !(a -> .Bool) (a -> a)   a ->  a             // f (f x) until (p 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`) infix 0 :: u:(St .s .a) u:(.a -> .(St .s .b)) -> u:(St .s .b);
return            :: u:a -> u:(St .s u:a)
```

StdMisc: miscellaneous functions

```
system module StdMisc

abort   :: !String -> .a            // stop reduction, print argument and core dump
undef   :: .a                       // fatal error, stop reduction.
```

B.2 Operations for parallel evaluation

B.2.1 StdProcId: operations for load distribution on ProcIds

B.3 Creating interactive processes

```
definition module StdEventIO

// Definition of IOState: the environment on which all GUI I/O functions operate.

:: IOState local share

:: *PState local share
   = { pLocal     :: local,   // the local (and private) data of an interaction
       pPublic    :: share,   // the shared data (in a group) of an interaction
       pFiles     :: !*Files, // the current state of the file system
       pIOState   :: !*IOState local share   // the IOState environment of this proce
     }

:: InitIO *local *share
   :==   [IOFunction (PState local share)]

:: IODef *local *share
   = { ioDefInit    :: InitIO local share,  // The initial actions of the process
       ioDefAbout   :: String               // The name of the process
     }

// Starting an interaction:

OpenIO :: !(IODef *local *share) (*local,*share) !*World -> *World

/* OpenIO starts an interaction specified by the IODef argument.
   Of each device only the first occurrence is taken into account. The
   program state argument consisting of a local and shared part serves
   as initial program state. If the interaction has been successfully
   created, the functions in InitialIO are evaluated from left-to-right.
   This is followed by the actual evaluation of the interaction. In the
   cause of the evaluation many new sub interactions can be created and
   terminated. The interaction created by OpenIO is the root interaction.
   OpenIO terminates as soon as all sub interactions (including the root
   interaction) have terminated.
   OpenIO returns the final file system and the resulting event stream. */
```

```
ParallelIO :: !ProcId !(IODef local share) !share
      !(IOState mess' local' share') -> (!IOId mess, !IOState mess' local' share')

/* ParallelIO applies only to the root interaction. If the root interaction
   is active, ParallelIO starts a new sub interaction that will run in
   parallel with the current sub interactions on the processor with the
   given PROCID. If the PROCID equals CurrentP (see StdProcId) then
   ParallelIO proceeds as NewIO applied to the remaining arguments.
   The new sub interaction is specified by the IODef argument. Creation of
   the new sub interaction is done as in OpenIO. The functions in InitialIO
   are evaluated from left-to-right before any abstract event handler of the
   new sub interaction is evaluated.
   If the interaction is inactive, ParallelIO does nothing. */

NewIO :: !(IODef *local *share) (*local,*share)
               !(IOState local` share`) -> IOState local` share`

/* If the interaction is active, NewIO starts a new sub interaction
   that will run interleaved with the current sub interactions.
   The new sub interaction is specified by the IODef argument. Creation of
   the new sub interaction is done as in OpenIO. The functions in
   InitialIO are evaluated from left-to-right before any abstract event
   handler of the new sub interaction is evaluated.
   The new sub interaction becomes the active sub interaction (so the
   current sub interaction is deactivated).
   If the interaction is inactive, NewIO does nothing. */

ShareIO :: !(IODef *local *share) *local
               !(IOState local` *share) -> IOState local` *share

/* If the interaction is active, ShareIO starts a new sub interaction that
   will run interleaved with the current sub interactions. The new sub
   interaction is specified by the IODef argument. Creation of the new sub
   interaction is done as in OpenIO. The functions in InitialIO are
   evaluated from left-to-right before any abstract event handler of the new
   sub interaction is evaluated.
   The new sub interaction becomes the active sub interaction (so the
   current sub interaction is deactivated).
   The new sub interaction can communicate with all sub interactions by
   means of the file system or by message passing. The new sub interaction
   can communicate with all sub interactions of the interaction group of the
   sub interaction that spawned it by means of the share program state
   component.
   If the interaction is inactive, ShareIO does nothing. */

QuitIO  :: !(IOState local share) -> IOState local share

/* QuitIO removes all devices that are held in the sub interaction.
   As a result evaluation of this sub interaction will terminate.
   QuitIO is the only function that causes OpenIO to terminate. */

HideIO  :: !(IOState local share) -> IOState local share
ShowIO  :: !(IOState local share) -> IOState local share

/* If the interaction is active, HideIO hides the sub interaction,
   and ShowIO makes it visible. Note that hiding a sub interaction does
   NOT disable the sub interaction, but simply makes it invisible.
   If the interaction is inactive, HideIO and ShowIO do nothing. */

ActivateIO :: !(IOState local share) -> IOState local share

/* If the interaction is active, ActivateIO activates the sub interaction.
   As a result, all open windows and dialogs of the sub interaction will
   be moved top-most on the desktop.
   If the interaction is inactive, ActivateIO does nothing. */

RequestIO :: !String !(IOState local share) -> IOState local share
```

```
/* If the interaction is inactive, RequestIO alerts the user that the
   interaction needs to become active. If the string argument is not empty,
   then this alert will consist of a Notice displaying the string.
   An interaction can issue an arbitrary amount of requests.
   If the interaction is active, RequestIO does nothing. */
```

B.4 Event based I/O

B.4.1 StdWindow: window handling

```
definition  module  StdWindow

import   StdIOCommon, StdWindowDef, StdControlDef,StdPicture

// Module StdWindow specifies all functions on windows.
// Functions applied to non-existent windows or unknown ids are ignored.

OpenWindow         :: !(WindowDef (PState local share)) !(PState local share)
            -> PState local share
OpenModalWindow    :: !(WindowDef (PState local share)) !(PState local share)
            -> PState local share

/* If the interaction is active, Open(Modal)Window opens the given window.
   In case a window with the same Id is already open then that window will
   be activated.
   OpenModalWindow terminates when the window has been closed (by means of
   CloseWindow).
   If the interaction is inactive, Open(Modal)Window does nothing. */

CloseWindow        :: !Id   !(IOState local share) -> IOState local share

/* If the interaction is active,   CloseWindow closes the indicated window.
   If the interaction is inactive, CloseWindow does nothing. */

HideWindows        :: ![Id] !(IOState local share) -> IOState local share
ShowWindows        :: ![Id] !(IOState local share) -> IOState local share

/* 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.   */

ActivateWindow     :: !Id !(IOState local share) -> IOState local share

/* If the interaction 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 interaction is inactive, ActivateWindow does nothing.   */

GetActiveWindow    :: !(IOState local share) -> (Bool, !Id, !IOState local share)

/* GetActiveWindow returns the Id of the currently frontmost and visible window of
   the interactive process. The Boolean result reports whether such a window exists.
   In case it is False, Id = 0.  */

StackWindow        :: !Id !Id !(IOState local share) -> IOState local share

/* If the process is active, StackWindow id1 id2 places the window with id1
   behind the window with id2. If id2 is unknown, the window becomes the active
   window.
   If id1 is unknown, or the process is inactive, StackWindow does nothing.   */

SetWindowPos       :: !Id !ItemPos !(IOState local share) -> IOState local share

/* 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 local share) -> (!Bool, !ItemOffset, !IOState local shar

/* GetWindowPos returns the current item offset position of the indicated window.
   The corresponding ItemPos is (LeftTop,offset). In case the window does not exist
   the Boolean result is False and ItemOffset=(0,0).  */

GetWindowFrame  :: !Id !(IOState local share) -> (!WindowFrame, !IOState local share)

/* 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)).   */

MoveWindowFrame   :: !Id Vector !(PState local share) -> PState local share

/* 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. */

SetPictureDomain     :: Id !PictureDomain !(PState local share) -> PState local share

/* 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.  */

SetWindowMinimumSize   :: Id Size           !( PState local share) ->  PState local

/* If the interactive process is active, SetWindowMinimumSize sets the minimum size
   of the indicated window as given. The new minimum size is set to be smaller than
   the current PictureDomain of the window. 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, or of inactive processes,
   SetWindowMinimumSize has no effect. */

SetWindowSize        :: Id Size           !( PState local share) ->  PState local sha

/* 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.
   In case of unknown Ids, of DialogWindows, or of inactive processes,
   SetWindowSize has no effect.  */

SetWindowTitle       :: !Id !Title        !(IOState local share) -> IOState local sha
SetWindowCursor      :: !Id !CursorShape  !(IOState local share) -> IOState local sha
SetWindowUpdate      :: !Id !(UpdateFunction (PState local share))
                              !(IOState local share) -> IOState local share
SetWindowClose       :: !Id !(IOFunction (PState local share))
                              !(IOState local share) -> IOState local share
SetWindowResize      :: !Id !(WindowResizeFunction (PState local share))
                              !(IOState local share) -> IOState local share
SetWindowActivate    :: !Id !(IOFunction (PState local share))
                              !(IOState local share) -> IOState local share
SetWindowDeactivate  :: !Id !(IOFunction (PState local share))
                              !(IOState local share) -> IOState local share

/* These functions set the indicated attributes. Invalid Ids are ignored.
   If the indicated window does not have the corresponding attribute then in case of
   SetWindow(Close/Resize) the new attribute is not set. In the other cases the window
   obtains the new attribute. */
```

```
EnableWindowMouse       :: !Id   !(IOState local share) -> IOState local share
DisableWindowMouse      :: !Id   !(IOState local share) -> IOState local share
EnableWindowKeys        :: !Id   !(IOState local share) -> IOState local share
DisableWindowKeys       :: !Id   !(IOState local share) -> IOState local share
SetWindowMouseFunction  :: !Id !(MouseFunction   ( PState local share))
                               !(IOState local share) -> IOState local share
SetWindowKeysFunction   :: !Id !(KeysFunction    ( PState local share))
                               !(IOState local share) -> IOState local share

// These functions change the state of mouse and keyboard input. Invalid Ids are ignorec

DrawInWindow      :: !Id ![DrawFunction] !(IOState local share) -> IOState local shar

// Draw in the window (behind all Controls). Invalid Ids are ignored.
```

StdWindowDef: the window device

```
definition  module  StdWindowDef

// Window definitions.

import   StdControlDef, StdFont

:: WindowDef *ps
   =  DialogWindow Title Id             [ControlDef ps] [WindowAttribute ps]
   |  Window       Title PictureDomain  [ControlDef ps] [WindowAttribute ps]

:: WindowFrame  :==   Rectangle

:: WindowAttribute   *ps                                  // Default:

   // Attributes for all windows:

   =  WindowId          Id                              // no id
   |  WindowPos         ItemPos                         // system dependent
   |  WindowSize        Measure Measure                 // screen size
   |  WindowItemSpace   Measure Measure                 // system dependent
   |  WindowStandBy                                     // system dependent
   |  WindowHide                                        // initially visible
   |  WindowClose       (IOFunction    ps)              // user can't close window
   |  WindowUpdate      (UpdateFunction   ps)           // update by system

   // Attributes for DialogWindows only:

   |  WindowMargin    Measure Measure                   // system dependent

   // Attributes for Windows only:

   |  WindowMinimumSize Size                            // system dependent
   |  WindowResize      (WindowResizeFunction      ps)// fixed size
   |  WindowActivate    (IOFunction ps)                 // I
   |  WindowDeactivate  (IOFunction ps)                 // I
   |  WindowMouse       SelectState (MouseFunction ps)// no mouse input
   |  WindowKeys        SelectState (KeysFunction  ps)// no keyboard input
   |  WindowCursor      CursorShape                     // no change of cursor

:: WindowResizeFunction *ps   :==   Size ->             // old window size
                                    Size ->             // new window size
                                    ps  -> ps
:: CursorShape
   =  StandardCursor
   |  BusyCursor
   |  IBeamCursor
   |  CrossCursor
   |  FatCrossCursor
   |  ArrowCursor
```

```
    | HiddenCursor

MMPerInch :== 25.4

IsAllWindowsAttribute        :: !(WindowAttribute .ps) -> Bool
IsWindowOnlyAttribute        :: !(WindowAttribute .ps) -> Bool
IsDialogWindowOnlyAttribute  :: !(WindowAttribute .ps) -> Bool

eqWindowAttribute    :: !(WindowAttribute .ps) !(WindowAttribute .ps) -> Bool

IsWindowId           :: !(WindowAttribute .ps) -> Bool
IsWindowResize       :: !(WindowAttribute .ps) -> Bool
IsWindowSize         :: !(WindowAttribute .ps) -> Bool
IsWindowMinimumSize  :: !(WindowAttribute .ps) -> Bool
IsWindowActivate     :: !(WindowAttribute .ps) -> Bool
IsWindowDeactivate   :: !(WindowAttribute .ps) -> Bool
IsWindowMouse        :: !(WindowAttribute .ps) -> Bool
IsWindowKeys         :: !(WindowAttribute .ps) -> Bool
IsWindowClose        :: !(WindowAttribute .ps) -> Bool
IsWindowUpdate       :: !(WindowAttribute .ps) -> Bool
IsWindowStandBy      :: !(WindowAttribute .ps) -> Bool
IsWindowHide         :: !(WindowAttribute .ps) -> Bool

GetWindowDefAttributes     :: !(WindowDef .ps) -> [WindowAttribute .ps]
SetWindowDefAttributes     :: ![WindowAttribute .ps] !(WindowDef .ps) -> WindowDef .p
GetWindowId                :: !(WindowAttribute .ps) -> Id
GetWindowResizeFunction    :: !(WindowAttribute .ps) -> WindowResizeFunction .ps
GetWindowSize              :: !(WindowAttribute .ps) -> Size
GetWindowMinimumSize       :: !(WindowAttribute .ps) -> Size
GetWindowMouseInfo         :: !(WindowAttribute .ps) -> (!Bool, !MouseFunction .ps)
GetWindowKeysInfo          :: !(WindowAttribute .ps) -> (!Bool, !KeysFunction  .ps)
GetWindowUpdateFunction    :: !(WindowAttribute .ps) -> UpdateFunction .ps
GetWindowActivateFunction  :: !(WindowAttribute .ps) -> IOFunction .ps
GetWindowCloseFunction     :: !(WindowAttribute .ps) -> IOFunction .ps

DialogFont                 :: Font
HorMeasureToPixels         :: !Measure -> Int
VerMeasureToPixels         :: !Measure -> Int
```

B.4.2 StdControlDef: the control device

```
definition  module  StdControl

import   StdIOCommon, StdControlDef

// Module StdControl specifies all functions on controls.

// Functions applied to unknown ids are ignored.

EnableControls       :: !Id ![Id]         !( PState local share) ->  PState local sha
DisableControls      :: !Id ![Id]         !( PState local share) ->  PState local sha
MarkCheckControls    :: !Id ![Id]         !(IOState local share) -> IOState local sha
UnmarkCheckControls  :: !Id ![Id]         !(IOState local share) -> IOState local sha
SelectRadioControl   :: !Id !Id           !(IOState local share) -> IOState local sha
ChangeEditText       :: !Id !Id !String   !(IOState local share) -> IOState local sha
ChangeDynamicText    :: !Id !Id !String   !(IOState local share) -> IOState local sha
ChangeControlLook    :: !Id !Id !(ControlLook  (PState local share))
                             ! (PState local share) ->  PState local share

// These functions change the state of controls. Invalid Ids are ignored.

ChangeCustomState :: !Id !Id !CustomState !(PState local share) -> PState local share

// Change the state of a CustomControl.

ChangeSliderState :: !Id !Id !SliderState !(IOState local share) -> IOState local sha
ChangeSliderThumb :: !Id !Id !Int         !(IOState local share) -> IOState local sha
```

```
// ChangeSlider(State/Thumb) set the SliderState/Thumb and redraw the settings of the sl

ChangeEventHandler   :: !Id !Id !(ControlAttribute (PState local share))
                           !(IOState local share) -> IOState local share

/* Set the abstract event handler of a (Radio/Check/Text/Edit/
   (Custom)Button/Custom/Compound)Control.
   The ControlAttribute argument must be a Control(Resize/Function/ModsFunction/Mouse/Ke
   attribute.  */

DrawInControl       :: !Id !Id ![DrawFunction] !(IOState local share) -> IOState local

// Draw in a (Custom(Button)/Compound)Control.

GetWindowInfo       :: !Id !(IOState local share)
               -> (!Bool, !WindowInfo, !IOState local share)
/* GetWindowInfo returns the WindowInfo for the indicated window.
   The Boolean result indicates whether the indicated window exists.
   In case it is False a dummy WindowInfo is returned. */

GetEditText            :: !Id   !WindowInfo -> (Bool,String)     // False -> ""
GetDynamicText         :: !Id   !WindowInfo -> (Bool,String)     // False -> ""
GetSelectedPopUpItem     :: !Id   !WindowInfo -> (Bool,Index)    // False -> 0
GetSelectedRadioControls  ::    !WindowInfo -> [Id]              //
GetSelectedCheckControls  ::    !WindowInfo -> [Id]              //
RadioControlMarked   :: !Id   !WindowInfo -> (Bool,Bool)         // False -> False
CheckControlMarked   :: !Id   !WindowInfo -> (Bool,Bool)         // False -> False
GetCustomState      :: !Id   !WindowInfo -> (Bool,CustomState)   // False -> BoolCS Fal
GetSliderState      :: !Id   !WindowInfo -> (Bool,SliderState)   // False -> {sliderMir
                                                                 // sliderMax   = 0,
                                                                 // sliderThumb= 0}

/* Functions that return the current contents of controls that
   can be changed by the user. The first Boolean result is False in
   case of invalid ids (if so dummy values are returned - see comment).

   The id passed to GetSelectedPopUpItem must be the id of a PopUpControl.

   Important: controls with no ControlId attribute, or illegal ids, can
              not be found in the WindowInfo!   */
```

StdControlDef: the control device

```
definition  module  StdControlDef

// Definition of controls

import   StdIOCommon

:: ControlDef *ps
   =  RadioControl    TextLine MarkState                      [ControlAttribute ps]
   |  CheckControl    TextLine MarkState                      [ControlAttribute ps]
   |  PopUpControl    [PopUpItem ps] Index                    [ControlAttribute ps]
   |  SliderControl   Direction Length SliderState (SliderAction ps)  [ControlAttribut
   |  TextControl     TextLine                                [ControlAttribute ps]
   |  EditControl     TextLine Width NrLines                  [ControlAttribute ps]
   |  ButtonControl   TextLine                                [ControlAttribute ps]
   |  CustomButtonControl  Size (ControlLook ps)              [ControlAttribute ps]
   |  CustomControl   Size (ControlLook ps) CustomState       [ControlAttribute ps]
   |  CompoundControl [ControlDef ps] (ControlLook ps)        [ControlAttribute ps]

:: TextLine           :==   String
:: NrLines            :==   Int
:: Width              :==   Int
:: Length             :==   Int
```

```
:: PopUpItem    *ps   :==   (TextLine, ps -> ps)
:: ControlLook  *ps   :==   SelectState -> UpdateFunction ps
:: SliderAction *ps   :==   SliderMove  -> ps -> ps
:: SliderState
   =  {  sliderMin    :: !Int,
         sliderMax    :: !Int,
         sliderThumb  :: !Int
      }
:: SliderMove
   =  SliderIncSmall
   |  SliderDecSmall
   |  SliderIncLarge
   |  SliderDecLarge
   |  SliderThumb Int
:: Direction
   =  Horizontal
   |  Vertical

:: CustomState
   =  BoolCS Bool | IntCS Int | RealCS Real | StringCS String
   |  PairCS CustomState CustomState
   |  ListCS [CustomState]

:: ControlAttribute *ps                                   // Default:
   =  ControlId            Id                             // no id
   |  ControlPos           ItemPos                        // (RightTo previous, (0,0))
   |  ControlSize          Size                           // system derived/overruled
   |  ControlMinimumSize   Size                           // (0,0)
   |  ControlResize        ControlResizeFunction          // no resize
   |  ControlSelectState   SelectState                    // control Able
   |  ControlFunction      (IOFunction    ps)             // I
   |  ControlModsFunction  (ModsIOFunction ps)            // ControlFunction
   |  ControlMouse         SelectState (MouseFunction ps)// no mouse input/overruled
   |  ControlKeys          SelectState (KeysFunction  ps)// no keyboard input/overrul

:: ControlResizeFunction
   :==   Size ->                                   // current  control size
         Size ->                                   // old      window  size
         Size ->                                   // new      window  size
         Size                                      // new      control size

eqControlAttribute      :: !(ControlAttribute .ps) !(ControlAttribute .ps) -> Bool

isControlId             :: !(ControlAttribute .ps) -> Bool
isControlPos            :: !(ControlAttribute .ps) -> Bool
isControlSize           :: !(ControlAttribute .ps) -> Bool
isControlMinimumSize    :: !(ControlAttribute .ps) -> Bool
isControlResize         :: !(ControlAttribute .ps) -> Bool
isControlSelectState    :: !(ControlAttribute .ps) -> Bool
isControlFunction       :: !(ControlAttribute .ps) -> Bool
isControlModsFunction   :: !(ControlAttribute .ps) -> Bool
isControlMouse          :: !(ControlAttribute .ps) -> Bool
isControlKeys           :: !(ControlAttribute .ps) -> Bool

getControlAttributes    :: !(ControlDef .ps) -> [ControlAttribute .ps]
setControlAttributes    :: [ControlAttribute .ps] !(ControlDef .ps) -> ControlDef .ps

getControlId            :: !(ControlAttribute .ps) -> Id
getControlPos           :: !(ControlAttribute .ps) -> ItemPos
getControlSize          :: !(ControlAttribute .ps) -> Size
getControlMinimumSize   :: !(ControlAttribute .ps) -> Size
getControlResize        :: !(ControlAttribute .ps) -> ControlResizeFunction
getControlSelectState   :: !(ControlAttribute .ps) -> SelectState
getControlFunction      :: !(ControlAttribute .ps) -> IOFunction .ps
getControlModsFunction  :: !(ControlAttribute .ps) -> ModsIOFunction .ps
getControlMouseInfo     :: !(ControlAttribute .ps) -> (SelectState, MouseFunction .ps
getControlKeysInfo      :: !(ControlAttribute .ps) -> (SelectState, KeysFunction  .ps
```

B.4.3 StdMenu: menu handling

```
definition module StdMenu

import   StdMenuDef

// Operations on menus.

// Operations on unknown Ids are ignored.

OpenMenus :: ![MenuDef (PState local share)]
      !(IOState local share) -> IOState local share

/* Open the given menu definitions for this interaction. */

EnableMenuSystem  ::
      !(IOState local share) -> IOState local share
DisableMenuSystem ::
      !(IOState local share) -> IOState local share

/* Enable/disable the MenuSystem. When the menu system is enabled the
   previously selectable menus and menu items will become selectable
   again. Operations on a disabled menu system take effect when the
   menu system is re-enabled. */

EnableMenus  :: ![Id]
      !(IOState local share) -> IOState local share
DisableMenus :: ![Id]
      !(IOState local share) -> IOState local share

/* Enable/disable PullDownMenus. Disabling a menu overrides the
   SelectStates of its menu elements, which become unselectable.
   Enabling a disabled menu re-establishes the SelectStates of
   the menu elements. */

InsertMenuItems   :: !Id !Int ![MenuElement (PState local share)]
      !(IOState local share) -> IOState local share
AppendMenuItems   :: !Id !Int ![MenuElement (PState local share)]
      !(IOState local share) -> IOState local share

/* Adding menu elements in a (sub)menu.
   InsertMenuItems adds menu elements before the item with the
   specified index, and AppendMenuItems adds menu elements after that
   item. 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 group. If the index exceeds the number of menu elements in
   the group, then the new menu elements are added behind the last
   menu element of the group. Only MenuItems and MenuSeparators
   can be added to (sub)menus. */

RemoveMenuItems      :: ![Id]
      !(IOState local share) -> IOState local share
RemoveMenuIndexItems:: !Id ![Int]
      !(IOState local share) -> IOState local share

/* Removing menu elements from the indicated (sub)menu(s).
   RemoveMenuItems removes the menu elements by their Id.
   RemoveMenuIndexItems removes menu elements of the indicated
   (sub)menu by their indices.
   As in the previous two functions, 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 group)
   are ignored. */

EnableMenuItems            :: ![Id]
      !(IOState local share) -> IOState local share
```

```
DisableMenuItems            :: ![Id]
      !(IOState local share) -> IOState local share
MarkMenuItems               :: ![Id]
      !(IOState local share) -> IOState local share
UnmarkMenuItems             :: ![Id]
      !(IOState local share) -> IOState local share
ChangeMenuItemTitles        :: ![(Id, Title)]
      !(IOState local share) -> IOState local share
ChangeMenuItemFunctions     ::
      ![(Id, MenuAttribute (PState local share))]
      !(IOState local share) -> IOState local share

/* Enable/disable, mark/unmark, and change titles/functions of
   MenuElements (including SubMenuItems). */
```

StdMenuDef: the menu device

```
definition  module  StdMenuDef

// MenuDefinitions:


:: MenuDef       *ps   =  Menu          Title [MenuElement ps]  [MenuAttribute ps]
:: MenuElement   *ps   =  SubMenuItem   Title [MenuElement ps]  [MenuAttribute ps]
                       |  MenuItem      Title                   [MenuAttribute ps]
                       |  MenuSeparator

:: MenuAttribute *ps                              // Default:
   =  MenuId            Id                        // no Id
   |  MenuSelectState   SelectState               // menu(item) Able
   |  MenuShortKey      KeyCode                   // no KeyCode
   |  MenuAltKey        Index                     // no AltKey
   |  MenuMarkState     MarkState                 // NoMark

   // Attributes ignored by (sub)menus:

   |  MenuFunction      (ps -> ps)                // I
   |  MenuModsFunction  (Modifiers -> ps -> ps)   // MenuFunction

MenuDefGetMenuId         :: !(MenuDef ps) -> Optional Id
MenuDefGetSelectState    :: !(MenuDef ps) -> SelectState
MenuDefGetElements       :: !(MenuDef ps) -> [MenuElement ps]
MenuDefSetAbility        :: !(MenuDef ps) !SelectState    -> MenuDef ps
MenuDefSetElements       :: !(MenuDef ps) ![MenuElement ps]  -> MenuDef ps

eqMenuAttribute          :: !(MenuAttribute .ps) !(MenuAttribute .ps) -> Bool
```

B.4.4 StdPicture: drawing in windows

```
definition  module  StdPicture

// Drawing functions and other operations on Pictures.

import   StdFont

:: *Picture

// The predefined figures that can be drawn:
:: Point          :== (!Int, !Int)
:: Line           :== (!Point, !Point)
:: Curve          :== (!Oval, !Int, !Int)
:: Rectangle      :== (!Point, !Point)
:: Oval           :== Rectangle
:: Polygon        :== (!Point, !PolygonShape)

:: PolygonShape :== [Vector]
```

```
:: Vector        :== (!Int, !Int)

// The pen attributes which influence the way figures are drawn:
:: PenSize     :== (!Int, !Int)
:: PenMode     =  CopyMode      | OrMode        | XorMode       | ClearMode     | Hilite
               |  NotCopyMode   | NotOrMode     | NotXorMode    | NotClearMode
:: PenPattern    =  BlackPattern
                 |  DkGreyPattern
                 |  GreyPattern
                 |  LtGreyPattern
                 |  WhitePattern

// The colours:
:: Colour      =  RGB Real Real Real
               |  BlackColour   | RedColour
               |  WhiteColour   | GreenColour
               |  BlueColour    | YellowColour
               |  CyanColour    | MagentaColour

MinRGB         :== 0.0
MaxRGB         :== 1.0

// Rules setting the attributes of a Picture:

/* SetPenSize (w,h) sets the PenSize to w pixels wide and h pixels high.
   SetPenMode sets the interference how new figures 'react' to drawn ones.
   SetPenPattern sets the way new figures are drawn.
   SetPenNormal  sets the SetPenSize to (1,1), the PenMode to CopyMode and
      the PenPattern to BlackPattern. */

SetPenSize       :: !PenSize    !Picture -> Picture
SetPenMode       :: !PenMode    !Picture -> Picture
SetPenPattern    :: !PenPattern !Picture -> Picture
SetPenNormal     ::             !Picture -> Picture

/* Using colours:
   There are basically two types of Colours: RGB and basic colours.
      An RGB colour defines the amount of red (r), green (g) and blue (b)
      in a certain colour by the tuple (r,g,b). These are Real values and
      each of them must be between MinRGB and MaxRGB (0.0 and 1.0).
      The colour black is defined by (MinRGB, MinRGB, MinRGB) and white
      by (MaxRGB, MaxRGB, MaxRGB).
      Given a RGB colour, all amounts are adjusted between MinRGB and
      MaxRGB.
   Only FullColour windows can apply RGB colours. Applications that use
   these windows may not run on all computers (e.g. Macintosh Plus).
   A small set of basic colours is defined that can be used on all systems.

   SetPenColour   sets the colour of the pen.
   SetBackColour  sets the background colour. */

SetPenColour     :: !Colour !Picture -> Picture
SetBackColour    :: !Colour !Picture -> Picture

/* Using fonts:
   The initial font of a Picture is 12 point Chicago in PlainStyle.
   SetFont      sets a new complete Font in the Picture.
   SetFontName  sets a new font without changing the style or size.
   SetFontStyle sets a new style without changing font or size.
   SetFontSize  sets a new size without changing font or style.
                The size is always adjusted between MinFontSize and
                MaxFontSize (see deltaFont.dcl).
   PictureCharWidth   (PictureStringWidth) yield the width of the given
                      Char (String) given the current font of the Picture.
   PictureFontMetrics yields the FontInfo of the current font. */

SetFont               :: !Font     !Picture -> Picture
SetFontName           :: !FontName   !Picture -> Picture
```

```
SetFontStyle         :: ![FontStyle]   !Picture -> Picture
SetFontSize          :: !FontSize   !Picture -> Picture

PictureCharWidth     :: !Char    !Picture -> (!Int,      !Picture)
PictureStringWidth   :: !String     !Picture -> (!Int,      !Picture)
PictureFontMetrics   ::          !Picture -> (!FontInfo, !Picture)

/* Drawing within in a polygonal clipping area:
   The Polygon argument defines the shape of the clipping area in
   which the drawing functions will be applied. */

DrawClip :: !Polygon [DrawFunction] !Picture -> Picture

// Determine the position of the pen:

GetPenPos :: !Picture -> (!Point, !Picture)

// Rules changing the position of the pen:

// Absolute and relative pen move operations (without drawing).

MovePenTo        :: !Point      !Picture -> Picture
MovePen          :: !Vector     !Picture -> Picture

// Absolute and relative pen move operations (with drawing).

LinePenTo        :: !Point      !Picture -> Picture
LinePen          :: !Vector     !Picture -> Picture

/* DrawChar (DrawString) draws the Char (String) in the current font.
   The baseline of the characters is the y coordinate of the pen.
   The new position of the pen is directly after the Char (String)
   including spacing.
*/

DrawChar         :: !Char    !Picture -> Picture
DrawString       :: !String     !Picture -> Picture

// Rules not changing the position of the pen after drawing:

/* Non plane figures:
      DrawPoint   draws the pixel in the Picture.
      DrawLine draws the line  in the Picture.
      DrawCurve   draws the curve in the Picture.
                  A Curve is part of an Oval o starting from angle a
                  upto angle b (both of type Int in degrees modulo 360):
                     (o, a, b).
                  See Wedges for further information on the angles. */

DrawPoint        :: !Point      !Picture -> Picture
DrawLine         :: !Line    !Picture -> Picture
DrawCurve        :: !Curve      !Picture -> Picture

/* A Rectangle is defined by two of its diagonal corner Points (A,B)
   with  A = (Ax, Ay),
         B = (Bx, By)
   such that Ax <> Bx and Ay <> By.
   In case either Ax = Bx or Ay = By, the Rectangle is empty.

   DrawRectangle  draws the edges of the rectangle.
   FillRectangle  draws the edges and interior of the rectangle.
   EraseRectangle erases the edges and interior of the rectangle.
   InvertRectangle   inverts the edges and interior of the rectangle.

   MoveRectangle  moves the contents of the rectangle over the given vector.
   CopyRectangle  copies the contents of the rectangle over the given vector. */

DrawRectangle       :: !Rectangle  !Picture -> Picture
```

```
FillRectangle      :: !Rectangle  !Picture -> Picture
EraseRectangle     :: !Rectangle  !Picture -> Picture
InvertRectangle    :: !Rectangle  !Picture -> Picture

MoveRectangle      :: !Rectangle !Vector !Picture -> Picture
CopyRectangle      :: !Rectangle !Vector !Picture -> Picture

/* Ovals: an Oval is defined by its enclosing Rectangle.
   Note : the Oval of a square Rectangle is a Circle.
*/

DrawOval           :: !Oval    !Picture -> Picture
FillOval           :: !Oval    !Picture -> Picture
EraseOval          :: !Oval    !Picture -> Picture
InvertOval         :: !Oval    !Picture -> Picture

/* Polygons: a Polygon is a figure drawn by a number of lines without
   taking the pen of the Picture, starting from some Point p.
   The PolygonShape s (a list [v1,...,vN] of Vectors) defines how the
   Polygon is drawn:
      MoveTo p, DrawLine from v1 upto vN, DrawLineTo p to close it.
   So a Polygon with s = [] is actually the Point p.

   ScalePolygon   by scale k sets shape [v1,...,vN] into [k*v1,...,k*vN].
                  Negative, as well as 0 are valid scales.
   MovePolygonTo  changes the starting point into the given Point and
   MovePolygon    moves the starting point by the given Vector.   */

DrawPolygon        :: !Polygon    !Picture -> Picture
FillPolygon        :: !Polygon    !Picture -> Picture
ErasePolygon       :: !Polygon    !Picture -> Picture
InvertPolygon      :: !Polygon    !Picture -> Picture

ScalePolygon       :: !Int        !Polygon -> Polygon
MovePolygon        :: !Vector     !Polygon -> Polygon
```

B.4.5 StdFont: writing in windows

```
definition module StdFont

// Operations on Fonts.

:: Font

:: FontName  :== String
:: FontStyle :== String
:: FontSize  :== Int
:: FontInfo
   =  {  fontAscent   :: !Int,
         fontDescent  :: !Int,
         fontLeading  :: !Int,
         fontMaxWidth :: !Int
      }

MinFontSize     :== 6
MaxFontSize     :== 128

SelectFont :: !FontName ![FontStyle] !FontSize -> (!Bool, !Font)

/* SelectFont 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 size is always adjusted between MinFontSize and MaxFontSize.
   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. */
```

```
DefaultFont :: (!FontName, ![FontStyle], !FontSize)

/* DefaultFont returns name, style and size of the default font. */

FontNames    :: [FontName]
FontStyles   :: !FontName   -> [FontStyle]
FontSizes    :: !FontName   -> [FontSize]

/* FontNames   returns the FontNames of all available fonts.
   FontStyles  returns the FontStyles of all available styles.
   FontSizes   returns all FontSizes of a font that are available without scaling.
   In case the font is unavailable, the styles or sizes of the default font
   are returned. */

FontCharWidth      :: !Char    !Font -> Int
FontCharWidths     :: ![Char]     !Font -> [Int]
FontStringWidth    :: !String     !Font -> Int
FontStringWidths   :: ![String]   !Font -> [Int]

/* FontCharWidth(s) (FontStringWidth(s)) return the width(s) in terms of pixels
   of given character(s) (string(s)) for a particular Font. */

FontMetrics     :: !Font -> FontInfo

/* FontMetrics yields the FontInfo in terms of pixels of a given Font. The FontInfo
   is a four-tuple (ascent, descent, max width, leading) which defines the metrics
   of a font:
      - ascent    is the height of the top most character measured from the base
      - descent   is the height of the bottom most character measured from the base
      - max width is the width of the widest character including spacing
      - leading   is the vertical distance between two lines of the same font.
   The full height of a line is the sum of the ascent, descent and leading. */
*/
```

B.4.6 TimerDef: timer handling

```
definition  module  StdTimer

// Operations on the TimerDevice.

import   StdTimerDef

TicksPerSecond :== 60

:: CurrentTime
   =  {  hours   :: !Int, // hours (0-23)
         minutes :: !Int, // minutes  (0-59)
         seconds :: !Int  // seconds  (0-59)
      }
:: CurrentDate
   =  {  year    :: !Int, // year
         month   :: !Int, // month       (1-12)
         day     :: !Int, // day         (1-31)
         dayNr   :: !Int  // day of week (1-7, Sunday=1, Saturday=7)
      }

OpenTimer           :: !(TimerDef (PState local share))
                          !(IOState local share) -> IOState local share
/* Open a new timer. This function has no effect in case the interaction
   already contains a timer with the same Id. Negative TimerIntervals
   are set to zero. */

CloseTimer            :: !Id !(IOState local share) -> IOState local share
/* Close the timer with the indicated Id. */

EnableTimer           :: !Id !(IOState local share) -> IOState local share
DisableTimer          :: !Id !(IOState local share) -> IOState local share
```

```
ChangeTimerFunction  :: !Id !(NrOfIntervals -> (PState local share)
                                          -> (PState local share))
                        !(IOState local share) -> IOState local share
SetTimerInterval     :: !Id !TimerInterval
                        !(IOState local share) -> IOState local share
/* Enable/disable, and change the TimerFunction and TimerInterval
   of the indicated timer. Negative TimerIntervals are set to zero. */

Wait               :: !TimerInterval .x -> .x

/* Wait suspends the interaction for TimerInterval ticks. */

GetTimerBlinkInterval:: !(IOState local share) -> (!TimerInterval, !IOState local sh
/* Returns the TimerInterval that should elaps between blinks of e.g.
   a cursor. This interval can change during the interaction! */

GetCurrentTime  :: !(IOState local share) -> (!CurrentTime, !IOState local share)
GetCurrentDate  :: !(IOState local share) -> (!CurrentDate, !IOState local share)
/* GetCurrentTime and GetCurrentDate return the current time and date. */
```

StdTimerDef: the timer device

```
definition  module  StdTimerDef

// TimerDefinitions:

:: TimerDef     *ps   =  Timer TimerInterval [TimerAttribute ps]
:: TimerInterval      :== Int
:: NrOfIntervals      :== Int

:: TimerAttribute  *ps                              // Default:
   =  TimerId         Id                            // no Id
   |  TimerSelect     SelectState                   // timer Able
   |  TimerFunction   (NrOfIntervals -> ps -> ps)   // f _ x = x

TimerDefAttributes   :: !(TimerDef *ps)
             -> (Optional Id, SelectState, TimerInterval, NrOfIntervals -> *ps -> *ps)
TimerDefSetAbility   :: !(TimerDef *ps) !SelectState                   -> TimerDef *
TimerDefSetInterval  :: !(TimerDef *ps) !TimerInterval                 -> TimerDef *
TimerDefSetFunction  :: !(TimerDef *ps) !(NrOfIntervals -> *ps -> *ps)  -> TimerDef *
```

B.4.7 StdReceiver: receiver handling

```
definition  module  StdReceiver

// Operations on the ReceiverDevice.

import   StdReceiverDef

OpenReceiver       :: !(ReceiverDef mess (PState local share))
                      !(IOState local share) -> (!IOId mess, !IOState local share)
CloseReceiver      :: !(IOId mess) !(IOState local share) -> IOState local share

EnableReceiver     :: !(IOId mess) !(IOState local share) -> IOState local share
DisableReceiver    :: !(IOId mess) !(IOState local share) -> IOState local share
/* Enable/Disable receiving events that have been raised to this
   interaction. If there is no receiver device, nothing happens. */

//ChangeReceiverFunction
//          :: !(IOId mess) !(ReceiverFunction mess (PState local share))
//             !(IOState local share) -> IOState local share
/* Change the ReceiverFunction of the receiver device. If there
   is no receiver device, nothing happens. */

ASyncSend     :: !(IOId mess) mess !(IOState local share) -> IOState local share
/* ASyncSend posts an event in the event stream environment that is
```

```
   addressed to the interaction with the given IOId. */
```

StdReceiverDef: the receiver device

```
definition  module  StdReceiverDef

// ReceiverDefinitions:

import   StdIOCommon

:: ReceiverDef        mess *ps
   =  Receiver [ReceiverAttribute mess ps]

:: ReceiverAttribute mess *ps                    // Default:
   =  ReceiverSelect     SelectState          // receiver Able
   |  ReceiverFunction   (mess -> ps -> ps)   // f _ x = x

ReceiverDefAttributes  :: !(ReceiverDef mess *ps) -> (SelectState, mess -> *ps -> *p
ReceiverDefSetAbility  :: !SelectState
                              !(ReceiverDef mess *ps) -> ReceiverDef mess *ps
ReceiverDefSetFunction :: !(mess -> *ps -> *ps)
                              !(ReceiverDef mess *ps) -> ReceiverDef mess *ps
```

B.4.8 StdIOCommon: common definitions

```
definition  module  StdIOCommon

// Common types for the event I/O system and their access rules:

import   StdPicture

:: Id               :==   Int
:: Index            :==   Int
:: Title            :==   String
:: Size             :==   (!Int,!Int)          // (width, height) in pixels

:: SelectState      =     Able | Unable
:: MarkState        =     Mark | NoMark

:: KeyboardState    :==   (!KeyCode, !KeyState, !Modifiers)
:: KeyCode          :==   Char
:: KeyState         =     KeyUp | KeyDown | KeyStillDown

:: MouseState       :==   (!MousePosition, !ButtonState, !Modifiers)
:: MousePosition    :==   Point
:: ButtonState      =     ButtonUp | ButtonDown
                    |     ButtonDoubleDown | ButtonTripleDown | ButtonStillDown

/* Modifiers indicates the meta keys that have been pressed (True) or
   not (False): (Shift, Option, Command, Control). */

:: Modifiers     :==   (!Bool,!Bool,!Bool,!Bool)

:: PictureDomain    :==   Rectangle
:: UpdateArea       :==   [Rectangle]

/* The layout language used for windows and controls. */

:: ItemPos
   :==   (  ItemLoc,
            ItemOffset
         )
:: ItemLoc
   =  LeftTop      | RightTop     | LeftBottom | RightBottom
   |  Left         | Center       | Right
```

```
    |  LeftOf  Id | RightTo  Id
    |  Above   Id | Below Id
:: ItemOffset
   :==   (!Int,!Int)

:: Measure
   =  MM       Real
   |  Inch     Real
   |  Pixel    Int

/* 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
:: KeysFunction    *ps   :==   KeyboardState  -> ps -> ps

EqualSelectState   :: !SelectState   !SelectState   -> Bool
EqualMarkState     :: !MarkState  !MarkState        -> Bool
EqualButtonState   :: !ButtonState   !ButtonState   -> Bool
Enabled            :: !SelectState                  -> Bool
Checked            :: !MarkState                    -> Bool
MarkSwitch         :: !MarkState                    -> MarkState
```

B.4.9 StdIOState: global operations on the IO State

```
definition  module  StdIOState

// Operations on the IOState that have a global effect.

import   StdEventIO, StdWindowDef

// Emit the alert sound.

Beep :: !(IOState local share) -> IOState local share
/* If the interaction is active, Beep emits a sound alert.
   If the interaction is inactive, Beep does nothing. */

// Operations on the global cursor:

SetGlobalCursor :: !CursorShape !(IOState local share) -> IOState local share
/* Set the shape of the cursor globally. This shape overrules the
   local cursor shapes of windows. */

ResetCursor :: !(IOState local share) -> IOState local share
/* Undoes the effect of SetGlobalCursor. */

ObscureCursor :: !(IOState local share) -> IOState local share
/* ObscureCursor hides the cursor until the mouse is moved. */

// Operations on the DoubleDownDistance:

SetDoubleDownDistance :: !DoubleDownDist !(IOState local share)
   -> IOState local share
/* 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:
```

```
ChangeIOActivate :: !(IOFunction (PState local share))
      !(IOState local share) -> IOState local share
ChangeIODeactivate :: !(IOFunction (PState local share))
      !(IOState local share) -> IOState local share
ChangeIOHelp :: !(IOFunction (PState local share))
      !(IOState local share) -> IOState local share
/* Change the IOActivate, IODeactivate, IOHelp attribute of the interaction. */
```

B.4.10 StdSystem: platform dependent settings

```
definition module StdSystem

// Platform dependent constants and functions (the values are given for the Macintosh).

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      :== (True,False,False,False)
OptionOnly     :== (False,True,False,False)
CommandOnly    :== (False,False,True,False)
ControlOnly    :== (False,False,False,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.

MMToHorPixels       :: !Real -> Int
MMToVerPixels       :: !Real -> Int
InchToHorPixels     :: !Real -> Int
```

```
InchToVerPixels   :: !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.5 File based I/O

```
system module StdFile

// 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 dat
   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. */
```

StdFileSelect: selecting files

```
definition module StdFileSelect

import   StdFile, StdEventIO

/* 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.  */

SelectInputFile :: !Files !(IOState local share)
            -> (!Bool, !String, !Files, !IOState local share)

/* SelectInputFile opens a dialog in which the user can traverse the
   file system to select an existing file. The boolean result indicates
   whether the user pressed the Open button (True) or the Cancel button
   (False). The String result contains the complete pathname of the
   selected file. When Cancel was pressed an empty string will be
   returned. */

SelectOutputFile:: !String !String !Files !(IOState local share)
            -> (!Bool, !String,!Files,! IOState local share)

/* SelectOutputFile opens a dialog 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. The boolean result indicates
   whether the user pressed the Save button (True) or the Cancel button
   (False). The String result contains the complete pathname of the
   selected file. When Cancel was pressed an empty string will be
   returned. When a file with the indicated name already exists in the
   indicated directory a confirm dialog will be opened. */
```

B.6 Built-in functions

Semantics (dot-dot expression):

$$\begin{aligned}
\mathrm{Tr}\,[\![\ [e_1..e_2]\ ]\!] &= _from_to\ e_1\ e_2\\
\mathrm{Tr}\,[\![\ [e_1..]\ ]\!] &= _from\ e_1\\
\mathrm{Tr}\,[\![\ [e_1,e_2..e_3]\ ]\!] &= _from_then_to\ e1\ e2\ e3\\
\mathrm{Tr}\,[\![\ [e_1,e_2..]\ ]\!] &= _from_then
\end{aligned}$$

B.6.1 The built-in functions

```
definition module system_functions

import   StdEnv

instance + Char
instance - Char
instance inc Char

_from           ::  a       -> [a] | Enum a
_from_to        :: !a !a    -> [a] | Enum a
_from_then      ::  a  a    -> [a] | Enum a
_from_then_to   :: !a !a !a -> [a] | Enum a
```

```
_inc a          :== inc a
_lt a b         :== a < b
_lteq a b       :== not (b < a)
_minus a b      :== a - b
```

implementation module system_functions

```
implementation module system_functions

import   StdEnv

_inc a      :== inc a
_lt a b     :== a < b
_lteq a b   :== not (b < a)
_minus a b  :== a - b

instance + Char      = plusChar
(plusChar) infixl 6 :: !Char !Char -> Char
(plusChar) a b
   = code {
      .inline ++c
         addI
      .end
   }

instance - Char      = minusChar
(minusChar) infixl 6 :: !Char !Char -> Char
(minusChar) a b
   = code {
      .inline ++c
         subI
      .end
   }

instance inc Char    = incChar
incChar :: !Char -> Char
incChar a
   = code {
      .inline inc+c
         incI
      .end
   }

_from :: a -> [a] | Enum 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 = []
```

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.

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 Deveopment System'. University of Nijmegen.

Manual on the use of Cleans' 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 Cleans' 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). 'FunSheet: a functional spreadsheet'. In: *Proc. of the 6th International Workshop on the Implementation of Functional Languages, UEA*, ed. J. Glauert, Norwich,UK, pp 11/1-11/17. An updated version will also appear in: *Journal of Functional Programming*.

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 Cleans' Event I/O.

❏ Peter Achten and Rinus Plasmeijer (1995). 'The Ins and Outs of Concurrent Clean I/O'. *Journal of Functional Programming,* 5, 1, to appear.

Explains the concepts behind Cleans' 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.

A

B

C

Z