# - Language Report -

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

**- Version 1.3 -**

**Copyright 1987 -1998**

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

**Rinus Plasmeijer**

**Marko van Eekelen**

- **Language Report** -

# Preface


- **Introduction**
- **More Information on CLEAN**
- **About this Language Report**
- **Some Remarks on the CLEAN Syntax**
- **Notational Conventions Used in this Report**
- **How to Obtain CLEAN**
- **Current State of the CLEAN System**
- **Copyright, Authors and Credits**
- **Final Remarks**


## Introduction

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

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

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

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

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

CLEAN has many features among which some very special ones. Functional languages are usually implemented using graph rewriting techniques. CLEAN is the only functional languages which basic semantics is defined in terms of Term Graph Rewriting (Barendregt *et al.*, 1987; Sleep *et al.*, 1993, Eekelen *et al.*, 1997) thus providing a better framework for controlling the time space behaviour of functional programs. Of particular importance for practical use is CLEAN's Uniqueness Type System

(Barendsen and Smetsers, 1993a) enabling the incorporation of destructive updates of arbitrary objects within a pure functional framework and the creation of direct interfaces with the outside world.

CLEAN's "unique" features have made it possible to predefine (in CLEAN) a sophisticated and efficient I/O library (Achten and Plasmeijer, 1992 & 1995). The CLEAN I/O library enables a CLEAN programmer to *specify interactive window based I/O applications* on a very high level of abstraction. The library forms a *platform independent interface to window systems*: one can port window based I/O applications written in CLEAN to different platforms without any modification of source code.

In the new 1.0 I/O library call-back functions and I/O definitions can be defined on arbitrary local states thus providing an object-oriented style of programming (Achten, 1996; Achten and Plasmeijer, 1997). Different kind of call-back functions and I/O definitions can be active at the same time. This makes it possible to *combine* different *interactive* CLEAN *programs* into a new application (a kind of multi-tasking within the same application). The applications can be regarded as lightweight processes which can communicate via files, shared state or message passing primitives ((a)synchronous message passing, remote procedure call). All this is provided in a *pure*, *sequential* functional world in which the call-back functions act as indivisible event handlers.

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

## More Information on CLEAN

There is a separate manual in preparation describing the standard libraries (including the I/O library) which are available for CLEAN (Standard Libraries for CLEAN, Achten *et al.*, 1998). The manual will become available on the net (www.cs.kun.nl/~clean).

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

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

There are many papers on the concepts introduced by the CLEAN group (such as *term graph rewriting* (Barendregt *et al.,* 1987), *lazy copying* (van Eekelen *et al.*, 1991), *abstract reduction* (Nöcker, 1993), *uniqueness typing* (Barendsen and Smetsers, 1993, **1996**), CLEAN's *I/O concept* (Achten, 1996 & 1997), *Parallel* CLEAN (Kesseler, 1991 & 1996). For the most recent information on papers and information about CLEAN please check our web pages (www.cs.kun.nl/~clean).

## About this Language Report

In this report the syntax and semantics of CLEAN version 1.3 are explained. We always give a motivation *why* we have included a certain feature. Although the report is not intended as introduction into the language, we did our best to make it as readable as possible. Nevertheless, one sometimes has to work through several sections spread all over the report.

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

## Some Remarks on the CLEAN Syntax

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

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

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

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

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

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

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

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

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

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

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

As is common in modern functional languages, there is a lay-out rule in CLEAN (see 2.3). 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** (Function definition in CLEAN making use of the lay-out rule).

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

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

## Notational Conventions Used in this Report

The following *notational conventions* are used in this report. Text is printed in Garamond 12pts,
the context free syntax descriptions are given in Geneva 9pts,
`examples of CLEAN programs are given in Courier 9pts,`
textual explanation to the examples are given in Garamond 10pts.

- Semantical restrictions are always given in a bulleted (•) list-of-points. When these restrictions are not obeyed they will almost always result in a compile-time error. In very few cases the restrictions

can only be detected at run-time (array index out-of-range, partial function called outside the domain).

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

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

All CLEAN examples given in this report assume that the lay-out dependent mode has been chosen which means that redundant semi-colons and curly braces are left out (see 1.3).

## How to Obtain CLEAN

CONCURRENT CLEAN and the CONCURRENT CLEAN INTEGRATED DEVELOPMENT ENVIRONMENT (IDE) can be used free of charge for *educational purposes **only***. They can be obtained

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

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

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

| **platform** | **Macintosh** | **PowerMac** | **PC** | **PC** | **PC** | **Sun** | **Sun** |
|---|---|---|---|---|---|---|---|
| **oper. sys.** | **MacOS** | **MacOS** | **Windows** | **Linux** | **OS/2** | **Solaris** | **SunOS** |
| at least version | **6.0** | **7.1.2** | **'95 / NT** | **ELF** | **2.0** | **2.0** | **4.1.2** |
| processor | Motorola | PowerPC | Intel | Intel | Intel | Sparc | Sparc |
| process. type | any | any | >= 486 | >= 486 | >= 486 | any | any |
| window system | MacOS | MacOS | Windows '95 / NT | Xview | OS/2 2.0 | Xview/ Open-Look | Xview/ Open-Look |
| Clean compiler | 1.3 | 1.3 | 1.3 | 1.3 | 1.1 | 1.3 | 1.1 |
| Libraries | | | | | | | |
| - Standard Env | yes | yes | yes | yes | yes | yes | yes |
| - 0.8 I/O Lib | yes | yes | yes | yes | yes | yes | yes |
| - 1.0 I/O Lib | yes | yes | *soon* | - | - | - | - |
| IDE | C version | Clean vrs. | Clean vrs. | no | no | no | no |
| assembler | not needed | not needed | not needed | gnu ass. | not needed | Sun ass. | Sun ass. |
| linker | included | included | included | gnu linker | OS2 linker | Sun linker | Sun linker |
| Code gen | Sequential | Sequential | Sequential | Sequential | Sequential | Sequential | Sequential |
| Profiler | No | Yes | Time | No | No | No | No |
| RAM in PC | | | | | | | |
| - minimal | 4 Mb | 8 Mb | 8 Mb | 8 Mb | 8 Mb | 8 Mb | 8 Mb |
| - comfortable | 8 Mb | 16 Mb | 16 Mb | 16 Mb | 16 Mb | 16 Mb | 16 Mb |
| Disk usage | | | | | | | |
| - minimal | 5 Mb | 7 Mb | 6 Mb | 6 Mb | 6 Mb | 7 Mb | 7 Mb |

CONCURRENT CLEAN is available on several platforms. The current situation is described in the Table (please check our WWW-pages regularly to see the latest news). New versions of CLEAN in general appear first on WINDOWS and MAC systems. The other platforms are updated less frequently.

The installation of the CLEAN compiler is rather dependent on the kind of platform one is working on. For each platform there is a `ReadMe` file which should help you to install properly. On the MAC's and WINDOW's systems ('95, NT) there is a dedicated CLEAN INTEGRATED DEVELOPMENT ENVIRONMENT written in CLEAN including dedicated editor, library search facilities and a project manager. For the POWERMAC and Windows there is a time profiler, for the POWERMAC also a space profiler. For the platforms without INTEGRATED DEVELOPMENT Environment one needs to use one of the standard editors available on the platform. In that case a distribution includes the clm (Clean make) frontend which will do the project management. We generate native code for *all* platforms.

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

## Current State of the CLEAN System

*Release 1.3* (April 1998).
- In a record and array update one can directly update any substructure.
- There is a special selector for unique records and arrays which returns the selected element as well as the unique record / array such that the record / array can be easily updated with a value depending on the selected element.
- The compiler and code generator optionally take user defined uniqueness type information into account to perform compile-time garbage collection for unique data structures in simple cases.
- A time profiler (PowerMac and Windows) and space profiler (PowerMac) has been added.
- It is no longer required to import all constructors of an algebraic type, when the type is imported explicitly.
- { and } are no longer treated in a special way by the lay-out rule.
- This manual has been restructured.

The new CLEAN 1.0 I/O library has been publicly released on the net (albeit for the MAC only).
- 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 introduce and combine local states in any I/O object.
- The I/O library provides an object-oriented style of programming.
- The description of the standard libraries has been moved from the language report to a separate document available on the net.

*The current release of the* CLEAN *system has the following limitations:*
- The new CLEAN programming environment is only available for PowerMacs and Window systems and needs improvement.
- The new CLEAN 1.0 I/O library is currently only available for the Mac. A Windows version is under construction. For a description of the 0.8 I/O library we refer to the draft of the new CLEAN book on the net and to the Addison-Wesley book (Plasmeijer and Van Eekelen, 1993).
- The Class mechanism can only have one type class variable which can only be instantiated with a flat type. Due to this restriction we had to define the overloaded array operators in a rather complicated way. This gives rise to a too complex class context for overloaded array operators. We are working on this feature.
- Macros are at this moment substituted in an early stage of the compilation process. This may cause cryptic error messages.
- Only simple variables can be used as array pattern.
- The arrow type constructor `(->)` cannot be used prefixed or used in a curried way.
- Annotations for parallelism are ignored. The distributed code generator is switched off. We are working on it.
- Everything exported in a definition module still has to be repeated in the corresponding implementation module.

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

---

*Release 1.2.3* (June 1997). CLEAN is now also available for Windows '95 and Windows NT 4.0. The compiler generates better error messages for uniqueness errors. Some bugs in the compiler have been removed: Incorrect code was generated for some functions that return a Real; The compiler crashed on some incorrect programs using array comprehensions; 1+0 was compiled as 1 0, instead of 1 + 0; Using both # and a macro in the same function could crash the compiler for some macros; Two overloading bugs have been removed; '#' is now allowed in operator definitions; (ppc) the compiler sometimes generated incorrect code for creating array of records with more than three elements; Arrays were printed as _STRING_ or _ARRAY_, if the string or array was annotated as strict in a constructor or record; sfopen gave inappropriate error message when opening a file for the second time; "toReal" for strings now also allows a '+' before the exponent.; (ppc) fixed bug in sfreadline and freadline; scrolling bug in IO library fixed. Programming Environment bug fixes: The CLEAN IDE for Windows uses less memory during editing and uses less processing time in the background. Directory names saved in the preference file are now interpreted correctly and the editor no longer ignores all keys typed with both Control and Alt shift keys pressed. The error that occurs when the linker cannot overwrite the application file no longer cause the IDE to quit. Shift now reverses direction of searches; responds to quit events added (for example during a shutdown); bug regarding copy/paste from other applications fixed; find error crash (index out of range) fixed; scrolling bug fixed; some small selection bugs fixed; cursor is now obscured when you start typing.

*Release 1.2* (January 1997). For any expression local definitions can be introduced with a let expression. We have introduced a new kind of let expression *before* a guard. Actions which have to be done in sequence can now much more intuitively be written down in such a sequential order. These new special let expressions also have a special scope. It allows to reuse the name for a single threaded parameter. Consequence is that for instance I/O actions can be written down in a more natural style (it looks imperative but it is not, of course). Guards can be nested. The syntax for algebraic types has been changed for existential quantifiers. The type `Void` in no longer a predefined type. Array comprehensions return arrays with unique elements. Multidimensional arrays can now be used in selection patterns and updates. Observation of unique objects is possible albeit for simple cases only (observation of objects of basic type stored in unique structures and observations made by polymorphic projection functions). It makes it more easy to inspect unique data structures before they are updated. It's no longer necessary to place parentheses around lambda expressions when they are used as arguments. This is especially useful when using a monadic programming style. The CLEAN compiler now gives warnings for functions that are not used in a module and are not exported. No code is generated for these functions. The strictness analyser is improved for guarded function alternatives. Some bugs in the compiler have been removed. We generate slightly better code (e.g. for functions that return strict tuples). CLEAN'S native PowerMac (MacOS) version is now released on the net. The old CLEAN I/O examples are rewritten to make use of the new features in CLEAN 1.2. CLEAN'S 1.0 I/O LIBRARY will soon be released on the net for MAC and POWERMAC. New CLEAN I/O examples for the new `1.0` library are made. The CLEAN PROGRAM DEVELOPMENT ENVIRONMENT has been improved. The language report has been updated for version 1.2 including a new chapter on I/O. Still some work as to be done on the chapter about uniqueness typing.

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

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

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

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

*Release 1.0.1* (April 1995). CLEAN 1.0 release on the Mac (Motorola). The most important changes in the language are: CLEAN has been changed from an intermediate language to a functional programming language with a syntax in the style of Miranda, Haskell and the like; so, various small syntactic sugar is added (infix operators, a case construct, local function definitions, lambda-abstractions, list comprehensions, lay-out rule, etcetera); overloaded functions, type classes and type constructor classes can be defined; records and arrays are added as predefined data structure with handy operations (such as an update operator for arrays and records, array comprehensions etc.); a more refined control of strictness is possible (partially strict data structures can be defined for any type, in particular for recursive types, there is strict let construct); the uniqueness typing is refined (now polymorphic and inferred, observation of uniquely typed objects is made easier); existentially quantified types can be defined. The compiler and code generator have been extended and are partly rewritten. Furthermore, the code generator is improved; the code generator is prepared for parallel and distributed evaluation;

*Previous Releases.* The first release of CLEAN was publicly available in 1987 and had version number 0.5 (at that time we thought half of the work was done, ;-) ). At that time, CLEAN was only thought as an intermediate language. Many releases followed. One of them was version 0.8 which is used in the Plasmeijer & Van Eekelen Bible (Adisson-Wesley, 1993).

## Copyright, Authors and Credits

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

**HILT - HIGH LEVEL SOFTWARE TOOLS B.V.,**

The Netherlands.

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

CLEAN, CONCURRENT CLEAN and the CONCURRENT CLEAN DEVELOPMENT SYSTEM, copyright 1987 -1998, HILT B.V., The Netherlands.

CLEAN is a spin-off of the research performed by the research group on FUNCTIONAL PROGRAMMING LANGUAGES, COMPUTING SCIENCE INSTITUTE, at the UNIVERSITY OF NIJMEGEN under the supervision of prof. dr. ir. Rinus Plasmeijer.

*The CONCURRENT CLEAN System is developed by:*

| | |
|---|---|
| Peter Achten: | Sequential and distributed Event I/O, I/O library for the Mac. |
| John van Groningen: | Code generators , CLEAN compiler , Profilers,<br>Low level interfaces, all machine wizarding. |
| Robert Holwerda: | I/O library for Windows '95 & Windows NT. |
| Martin van Hintum: | Integrated DEVELOPMENT ENVIRONMENT (CLEAN version). |
| Marco Kesseler: | Parallel code generator (ParSyTec (Transputer)). |
| Eric Nöcker: | Strictness analyser via abstract reduction, I/O library for OS/2. |

Leon Pillich: I/O library for the Sun.
Sjaak Smetsers: CLEAN compiler,
All type systems (including uniqueness typing and type classes),
Ron Wichers Schreur: Integrated DEVELOPMENT ENVIRONMENT , Testing,
Parser, Support, Porting, CLEAN distribution on the net.
Rinus Plasmeijer &
Marko van Eekelen: CLEAN language design.
Rinus Plasmeijer: Overall design and implementation supervision.

*Special thanks to the following people:*

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

*Many thanks to the following sponsors:*

- the Dutch Technology Foundation (STW);
- the Dutch Foundation for Scientific Research (NWO);
- the International Academic Centre for Informatics (IACI);
- Kropman B.V., Installation Techniques, Nijmegen, The Netherlands;
- Hitachi Advanced Research Laboratories, Japan;
- the Dutch Ministry of Science and Education (the Parallel Reduction Machine project (1984-1987)) who initiated the CONCURRENT CLEAN research;
- Esprit Basic Research Action (project 3074, SemaGraph: the Semantics and Pragmatics of Graph Rewriting (1989-1991));
- Esprit Basic Research Action (SemaGraph II working group 3646 (1992-1995));
- Esprit Parallel Computing Action (project 4106, (1990-1991));
- Esprit II (TIP-M project area II.3.2, Tropics: TRansparent Object-oriented Parallel Information Computing System (1989-1990)).

*A system like CLEAN cannot be produced without an enormous investment in time, effort and money. We would therefore like to thank all commercial CLEAN users who are decent enough to pay the license royalties.*

## Final Remarks

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

*April 1998*

*Rinus Plasmeijer and Marko van Eekelen*

| | | |
|---|---|---|
| Affiliation: | **CSI**<br>**COMPUTING SCIENCE**<br>**INSTITUTE** | **HILT**<br>**HIGH LEVEL SOFTWARE**<br>**TOOLS B.V.** |
| Mail address: | University of Nijmegen,<br>Toernooiveld 100,<br>6525 EC Nijmegen,<br>The Netherlands. | Universitair Bedrijven Centrum,<br>Toernooiveld 1,<br>6525 ED Nijmegen,<br>The Netherlands. |
| e-mail: | rinus@cs.kun.nl<br>marko@cs.kun.nl | rinus@hilt.nl<br>marko@hilt.nl |
| Phone: | +31 24 3652644 | +31 24 3528827 |

| | | |
|---|---|---|
| Fax: | +31 24 3652525 | +31 24 3652525 |
| CLEAN on internet: | www.cs.kun.nl/~clean | www.hilt.nl |
| CLEAN on ftp: | ftp.cs.kun.nl in pub/Clean | |
| Questions about CLEAN: | clean@cs.kun.nl | info@hilt.nl |
| Mailing lists:: | clean-request@cs.kun.nl, subject::help | |

# Table of Contents

**Preface** **i**

Introduction i
More Information on CLEAN ii
About this Language Report ii
Some Remarks on the CLEAN Syntax iii
Notational Conventions Used in this Report iii
How to Obtain CLEAN iv
Current State of the CLEAN System v
Copyright, Authors and Credits vii
Final Remarks viii

**Table of Contents** **xi**

**Basic Semantics** **1**

1.1 Graph Rewriting 1
1.1.1 A Small Example 2
1.2 Global Graphs 4
1.3 Key Features of CLEAN 4

**Defining Modules** **7**

2.1 Identifiers, Scopes and Name Spaces 7
2.1.1 Naming Conventions of Identifiers 7
2.1.2 Scopes and Name Spaces 8
2.1.3 Nesting of Scopes 8
2.2 Modular Structure of CLEAN Programs 9
2.3 Implementation Modules 9
2.3.1 The Main or Start Module 9
2.3.2 Scope of Global Definitions in Implementation Modules 10
2.3.3 Begin and End of a Definition: the Lay-Out Rule 11
2.4 Definition Modules 12
2.5 Importing Definitions 13
2.5.1 Explicit Imports of Definitions 13
2.5.2 Implicit Imports of Definitions 14
2.6 System Definition and Implementation Modules 14

**Defining Functions** **17**

3.1 Defining Functions 17
3.2 Patterns 18
3.2.1 Constructor Patterns 18
3.2.2 Simple Constructor Patterns 19
3.2.3 Variables and Wildcards in Patterns 19

3.2.4 Constant Values of Basic Type as Pattern 20
3.2.5 List Patterns 20
3.2.6 Tuple Patterns 20
3.2.7 Record Patterns 20
3.2.8 Array Patterns 21
3.3 Guards 21
3.4 Expressions 22
3.4.1 Applications 22
3.4.2 Constructor or Function Name 23
3.4.3 Graph Variables 23
3.4.4 Creating Constant Values of Basic Type 23
3.4.5 Creating Lists 24
3.4.6 Creating Tuples 26
3.4.7 Creating Records and Selection of Record Fields 26
3.4.8 Creating Arrays and Selection of Array Elements 28
3.4.9 Lambda Abstraction 31
3.4.10 Case Expression and Conditional Expression 32
3.4.11 Let Expression: Local Definitions for Expressions 32
3.5 Local Definitions 33
3.5.1 Where Block: Local Definitions for a Function Alternative 33
3.5.2 With Block: Local Definitions for a Guarded Alternative 34
3.5.3 Defining Local Functions 34
3.5.4 Defining Local Constants 34
3.6 Special Local Definitions 36
3.6.1 Strict Let Expression: Strict Local Constants 36
3.6.2 Let-Before Expression: Local Constants for a Guard 37

**Defining Types 39**

4.1 Predefined Types 39
4.1.1 Basic Types 40
4.1.2 Predefined Abstract Types 40
4.1.3 List Types 40
4.1.4 Tuple Types 40
4.1.5 Array Types 41
4.1.6 Arrow Types 41
4.2 Defining New Types 41
4.2.1 Defining Algebraic Data Types 42
4.2.2 Defining Record Types 45
4.2.3 Defining Synonym Types 46
4.2.4 Defining Abstract Data Types 46
4.3 Typing Functions 47
4.3.1 Typing Curried Functions 48
4.3.2 Typing Operators 48
4.3.3 Typing Partial Functions 48
4.4 Typing Overloaded Functions 49
4.4.1 Type Classes 49
4.4.2 Functions Defined in Terms of Overloaded Functions 50
4.4.3 Instances of Type Classes Defined in Terms of Overloaded Functions 51
4.4.4 Type Constructor Classes 52
4.4.5 Generic Instances 52
4.4.6 Default Instances 52
4.4.7 Defining Derived Members in a Class 53
4.4.8 A Shorthand for Defining Overloaded Functions 54
4.4.9 Classes Defined in Terms of Other Classes 54
4.4.10 Exporting Type Classes 54
4.4.11 Semantic Restrictions on Type Classes 55
4.4.12 The Costs of Overloading 55
4.5 Defining Uniqueness Types 56

4.5.1 Basic Ideas Behind Uniqueness Typing 56
4.5.2 Attribute Propagation 58
4.5.3 Defining New Types with Uniqueness Attributes 59
4.5.4 Uniqueness and Sharing 61
4.5.5 Combining Uniqueness Typing and Overloading 64
4.5.6 Higher-Order Type Definitions 67
4.5.7 Destructive Updates using Uniqueness Typing 68

**Annotations and Directives 71**

5.1 Annotations to Change Lazy Evaluation into Strict Evaluation 71
5.1.1 Advantages and Disadvantages of Lazy versus Strict Evaluation 71
5.1.2 Strict and Lazy Context 72
5.1.3 Space Consumption in Strict and Lazy Context 72
5.1.4 Time Consumption in Strict and Lazy Context 73
5.1.5 Changing Lazy into Strict Evaluation 73
5.2 Defining Graphs on the Global Level 77
5.3 Defining Macros 77
5.4 Process Annotations 78
5.4.1 Process Creation 79
5.4.2 Process Communication 80
5.5 Efficiency Tips 80

**Context-Free Syntax Description 83**

A.1 CLEAN Program 83
A.2 Import Definition 84
A.3 Function Definition 84
A.4 Macro Definition 86
A.5 Type Definition 86
A.6 Class Definition 87
A.7 Names 87
A.8 Denotations 88

**Lexical Structure 89**

B.1 Lexical Program Structure 89
B.2 Comments 89
B.3 Reserved Keywords and Symbols 90

**Bibliography 91**

**Index 93**

# 1

# Basic Semantics

1.1 Graph Rewriting
1.2 Global Graphs
1.3 Key Features of CLEAN

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

## 1.1 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-ids* 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).

### 1.1.1 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-ids. Using the set of rules with explicit node-ids 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)
```

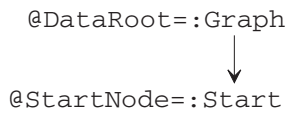
```
x =: Add y z
y =: Succ a          =>   m =: Succ n
                          n =: Add a z              (2)

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-ids appearing in the rewrite rules from the node-ids 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.

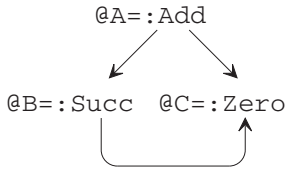
The patterns in rewrite rules contain *formal node-ids*. During the matching these formal node-ids are mapped to the *actual node-ids* of the graph After that the following semantic actions are performed:

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:

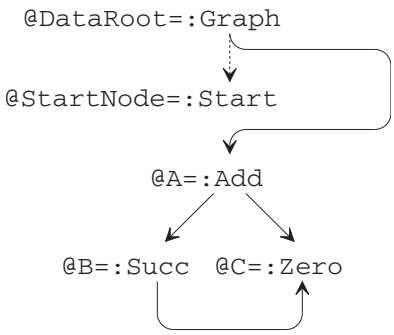



All applied occurrences of `@StartNode` will be replaced by occurrences of `@A`. The graph after rewriting is then:

The graph after rewriting:

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

Pictorial notation:

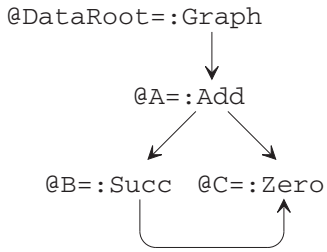



This completes one rewrite. All nodes that are not accessible from `@DataRoot` are garbage and not considered any more in the next rewrite steps. In an implementation once in a while garbage collection is performed in order to reclaim the memory space occupied by these garbage nodes. In this example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

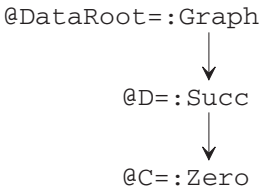



The graph accessible from `@DataRoot` still contains a redex. It matches rule 2 yielding the expected normal form:

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.

## 1.2 Global Graphs

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

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

The root of the initial graph will not only contain the node-id of the start node, the root of the graph to be rewritten, but it will also contain for each *global graph* (see 5.2) a reference to an initial node (initialised with the symbol `Initial`). All references to a specific global graph will be references to its initial node or, when it is rewritten, they will be references to its reduct.

## 1.3 Key Features of CLEAN

On top of the Graph Rewriting System a full featured functional programming language is defined. The most important *features* we added to CLEAN are:

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

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

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

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

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

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

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

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

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

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

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

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

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

# 2

# Defining Modules

2.1 Identifiers, Scopes and Name Spaces
2.2 Modular Structure of CLEAN Programs
2.3 Implementation Modules
2.4 Definition Modules
2.5 Importing Definitions
2.6 System Definition and Implementation Modules

A CLEAN program is composed out of modules. Each module is stored in a file which contains CLEAN source code. There are implementation modules and definition modules, in the spirit of Modula-2 (**Wirth**, 1982). This module system is used for several reasons.
First of all, the module structure is used to control the scope of definitions. The basic idea is that definitions only have a meaning in the implementation module they are defined in unless they are exported by the corresponding definition module.
Having the exported definitions collected in a separate definition module has as advantage that one also obtains a self-contained interface document one can reach out to others. The definition module is a document which defines what can be used by others and how it can be used without revealing uninteresting implementation details.
Furthermore, the module structure enables separate compilation which heavily reduces compilation time. An implementation module can be changed without the need of recompiling other modules. When the contents of a definition module is changed only those modules which are affected by this change need to be recompiled.
In this Chapter we explain the module structure of CLEAN and the influence it has on the scope of definitions. New scopes can also be introduced inside modules. This is further explained in the Chapters 2 and 3
In the pictures in the subsections below nested scopes are indicated by nested boxes.

## 2.1 Identifiers, Scopes and Name Spaces

### 2.1.1 Naming Conventions of Identifiers

In CLEAN we distinguish the following kind of *identifiers*.

```
ModuleName          = LowerCaseId | UpperCaseId | FunnyId
FunctionName        = LowerCaseId | UpperCaseId | FunnyId
ConstructorName     =             UpperCaseId | FunnyId
SelectorVariable    = LowerCaseId
Variable            = LowerCaseId
MacroName           = LowerCaseId | UpperCaseId | FunnyId
FieldName           = LowerCaseId
TypeName            =             UpperCaseId | FunnyId
TypeVariable        = LowerCaseId
UniqueTypeVariable  = LowerCaseId
ClassName           = 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
```

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 can either start with an uppercase or a lowercase character. 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 Appendix A). These two kind of characters cannot be mixed in one identifier. This makes it possible to leave out white space in expressions like `a+1` (same as `a + 1`).

### 2.1.2 Scopes and Name Spaces

The *scope* is the program region in which definitions (e.g. function definition, class definition, macro definition, type definition) with the identifiers introduced (e.g function name, class name, class variable, macro name, type constructor name, type variable name) have a meaning.

It must be clear from the context to which definition an identifier is referring. If all identifiers in a scope have different names than it will always be clear which definition is meant. However, one generally wants to have a free choice in naming identifiers. If identifiers belong to different *name spaces* no conflict can arise even if the same name is used. In CLEAN the following name spaces exist:

- *ModuleName*s form a name space;
- *FunctionName*s, *ConstructorName*s, *SelectorVariable*s, *Variable*s and *MacroName*s form a name space;
- *FieldName*s form a name space;
- *TypeName*s, *TypeVariable*s and *UniqueTypeVariable*s form a name space;
- *ClassName*s form a name space.

So, it is allowed to use the same identifier name for different purposes as long as the identifier belong to different name spaces.

- Identifiers belonging to the same name space must all have different names within the same scope. Under certain conditions it is allowed to use the same name for different functions and operators (overloading, see 4.4).

### 2.1.3 Nesting of Scopes

Reusing identifier names is possible by introducing a new scope level. Scopes can be nested: within a scope a new *nested scope* 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 redefine definitions or reuse names given in a surrounding scope. When a name is reused the old name and definition is no longer in scope and cannot be used in the new 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).

## 2.2 Modular Structure of CLEAN Programs

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 *unless* these definitions are exported by putting them into the corresponding definition module. In that case the definitions also have a meaning in those other modules in which the definitions are imported (see 2.5).

```
CLEANProgram          = {Module}+
Module                = DefinitionModule
                      | ImplementationModule
DefinitionModule      = definition module ModuleName ;
                        {DefDefinition}
                      | system module ModuleName ;
                        {DefDefinition}
ImplementationModule  = [implementation] module ModuleName ;
                        {ImplDefinition}
```

- An executable CLEAN program consists at least of one implementation module, the *main* or *start module*, which is the top-most module (*root module*) of a CLEAN program.
- 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 *.dcl* as suffix, an *implementation* module should have *.icl* as suffix.
- A definition module can have at most one corresponding implementation module.
- Every implementation module (except the main module, see 2.3.1) must have a corresponding definition module.

## 2.3 Implementation Modules

### 2.3.1 The Main or Start Module

- In the main module a `Start` rule has to be defined (see Chapter 1).
- 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 (which makes sense for a top-most module).

**Example** (a very tiny but complete CLEAN program consisting of one implementation module):

```
module hello

Start = "Hello World!"
```

*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 (see Chapter 1). The right-hand side of the `Start` rule is regarded to be the *initial expression* to be computed.

It is allowed to have a `Start` rule in other implementation modules as well. 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.

The definition of the left-hand side of the Start rule consists of the *symbol* `Start` with one optional argument (of type `*World`), which is the environment parameter which is necessary to write interactive applications.

A CLEAN programs can run in two modes.

*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 or debug functions.

One can choose to print the result of a `Start` expression *with* or *without* the data constructors. For example, the initial expression

```
Start:: String
Start = "Hello World!"
```

in mode "show data constructors" will print: `"Hello World!"`, in mode "don't show data constructors" it will print: `Hello World!`

*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* what a functional program needs to interact during execution with the concrete world. The world can be seen as a *state* and modifications of the world can be realised via *state transition functions* defined on the world or a part of the world. By requiring that these state transition functions work on a *unique* world the modifications of the abstract world can directly be realised in the real physical world, without loss of efficiency and without losing referential transparency (see Chapter 4)

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 definition of the I/O library is treated in a separate document (Standard Libraries for CLEAN, Achten *et al.*, 1998).

### 2.3.2 Scope of Global Definitions in Implementation Modules

In an implementation module the following global definitions can be specified in *any* order.

```
ImplDefinition  =  ImportDef        //  see 2.5
                |  FunctionDef      //  see Chapter 3
                |  GraphDef         //  see 3.5.4
                |  MacroDef         //  see Chapter 5
                |  TypeDef          //  see Chapter 4
                |  ClassDef         //  see 4.4
```

*Definitions* on the *global* level (= outermost level in the module,) have in principle the whole implementation module as scope (see Figure 2.1).

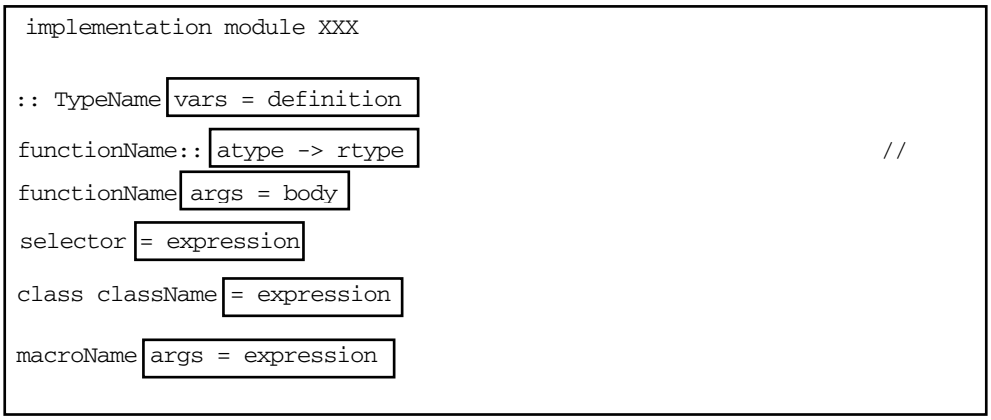



**Figure 2.1** (Scope of global definitions inside an implementation module).

Types can only be defined globally (see Chapter 4 and 6) and therefore always have a meaning in the whole implementation module. 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.

Functions, the type of these functions, constants (selectors) and macros can be defined on the *global* level as well as on a *local* level in nested scopes. When defined globally they have a meaning in the whole implementation module. Arguments introduced on the left-hand side of a definition (formal arguments) only have a meaning in the corresponding right-hand side.

Functions, the type of these functions, constants (selectors) and macros can also be defined locally in a new scope. However, new scopes can only be introduced at certain points. In functional languages local definitions are by tradition defined by using *let*-expressions (definitions given *before* they are used in a certain expression, nice for a bottom-up style of programming) and *where*-blocks (definitions given *afterwards*, nice for a top-down style of programming). These constructs are explained in detail in Chapter 3.

### 2.3.3 Begin and End of a Definition: the Lay-Out Rule

CLEAN modules can be written in two modes: lay-out sensitive mode 'on' and 'off'. The lay-out sensitive mode is switched off when a semi-colon is specified after the module name. In that case each definition has to be ended with a semicolon '`;`'. A new scope has to begin with '`{`' and ends with a '`}`'. This mode is handy if CLEAN code is generated automatically (e.g. by a compiler).

**Example** (example of a CLEAN program not using the lay-out rule).

```
module primes;

import StdEnv;

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

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

Programs look a little bit old fashioned C-like in this way. Functional programmers generally prefer a more mathematical style. Hence, as is common in modern functional languages, there is a lay-out rule

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

In *lay-out sensitive mode* the indentation of the first lexeme after the keywords `let`, `#`, `let!`, `#!`, `of`, `where`, or `with` determines the indentation that the group of definitions following the keyword has to obey. Depending on the indentation of the first lexeme on a subsequent line the following happens. A new definition is assumed (and a semicolon is inserted) if the lexeme starts on the same indentation, except for the following lexemes: `|`, `#`, `#!`, `where` or `with`. 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.

We strongly advise to write programs in lay-out sensitive mode. *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** (same program using the lay-out sensitive mode).

```
module primes

import StdEnv

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

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

## 2.4 Definition Modules

The definitions given in an implementation module only have a meaning in the module in which they are defined. If you want to export a definition, you have to specify the definition in the corresponding definition module. Some definitions can only appear in implementation modules, not in definition modules. The idea is to hide the actual implementation from the outside world. This is good for software engineering reasons while another advantage is that an implementation module can be recompiled separately without a need to recompile other modules. Recompilation of other modules is only necessary when a definition module is changed. All modules depending on the changed module have to be recompiled as well. Implementations of functions, graphs and class instances are therefore only allowed in *implementation* modules. They are exported by only specifying their type definition in the definition module. Also the right-hand side of any type definition can remain hidden. In this way an abstract data type is created (see 4.2.4).

In a definition module the following global definitions can be given in *any* order.

```
DefDefinition   =  ImportDef                    //  see 2.5
                |  FunctionTypeDef              //  see 4.3
                |  MacroDef                     //  see 5.3
                |  TypeDef                      //  see Chapter 4
                |  ClassDef                     //  see 2.5
                |  TypeClassInstanceExportDef   //  see 2.5
```

- The definitions given in an implementation module only have a meaning in the module in which they are defined (see 2.3) *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 2.5).

- The definitions (with exception of TypeClassInstanceExportDef's) given in a definition module have to be repeated in the corresponding implementation module (this restriction will be removed in a future version of CLEAN).
- In the corresponding implementation module all exported definitions have to get an appropriate implementation (this holds for functions, abstract data types, class instances).
- An *abstract data type* is exported by specifying the left-hand side of a type rule in the definition module. In the corresponding implementation module the abstract type *has to be defined again* but the right-hand side has to be defined as well. It can be either an algebraic type, record type or synonym type definition. For such an abstract data type only the name of the type is exported but not its definition.
- A *function,* global *graph* or *class instance* is exported by repeating the type header in the definition module. For optimal efficiency it is recommended also to specify strictness annotations (see 5.1). For library functions it is recommended also to specify the uniqueness type attributes (see Chapter 4). The implementation of the function, graph, class instance has to be given in the implementation module.

**Example** (definition module):

```
definition module Complex

::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 Complex

::Complex :== (!Real,!Real)                //  concrete type, in this case it is a 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)
```

## 2.5 Importing Definitions

Via an *import statement* a definition *exported* by a definition module (see 2.4) can be *imported* into any other (definition or implementation) module. There are two kind of import statements, *explicit* imports and *implicit* imports.

```
ImportDef          =  ImplicitImportDef
                   |  ExplicitImportDef
```

A module *depends on* another module if it imports something 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 $M_1$.

### 2.5.1 Explicit Imports of Definitions

*Explicit imports* are import statements in which the modules to import from as well as the identifiers indicating the definitions to import are explicitly specified.

```
ExplicitImportDef  =  from ModuleName import {Imports}-list ;
Imports            =  FunctionName
                   |  ConstructorName
                   |  SelectorVariable
```

| *FieldName*
| *MacroName*
| *TypeName*
| *ClassName*

All identifiers explicitly being imported in a definition or implementation module will be included in the global scope level (= outermost scope, see 2.3.2) of the module which does the import. Importing identifiers can cause error messages because the imported identifiers may be in conflict with other identifiers in this scope (remember that identifiers belonging to the same name space must all have different names within the same scope, see 2.1). This problem can be solved by renaming the internally defined identifiers or by renaming the imported identifiers (e.g. by adding an additional module layer just to rename things).

**Example** (explicit import):

```
implementation module XXX

from Complex import Complex, re, im, mkcomplex
```

### 2.5.2 Implicit Imports of Definitions

ImplicitImportDef = **import** {*ModuleName*}-*list* ;

*Implicit imports* are import statements in which only the module name to import from is mentioned. In this case *all* definitions that are *exported* from that module are imported as well as *all* definitions that on their turn are *imported* in the indicated 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 Hence, it is meaningful to have definition modules with import statements but without any definitions and without a corresponding implementation module.;

**Example** (implicit import): all (arithmetic) rules which are predefined can be imported easily with one import statement:

```
import MyStdEnv
```

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 MyStdEnv

import
     StdBool, StdChar, StdInt, StdReal, StdString
```

All identifiers implicitly being imported in a definition or implementation module will be included in the global scope level (= outermost scope, see 2.3.2) of the module which does the import. Importing identifiers can cause error messages because the imported identifiers may be in conflict with other identifiers in this scope (remember that identifiers belonging to the same name space must all have different names within the same scope, see 2.1). This problem can be solved by renaming the internally defined identifiers or by renaming the imported identifiers (e.g. by adding an additional module layer just to rename things).

## 2.6 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 call code written in a foreign language from CLEAN programs. 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.

# 3

# Defining Functions

3.1 Defining Functions
3.2 Patterns
3.3 Guards
3.4 Expressions
3.5 Local Definitions
3.6 Special Local Definitions

In this Section *function definitions* are treated (actually: *graph rewrite rules*). *Operator* definitions are regarded as special kind of function definitions (see 3.1 and 4.3). The body of a function consists of a root expression(see 3.4). With the help of patterns (see 3.2) and guards (see 3.3) a distinction can be made between several alternative definitions for a function. Functions and graphs can be defined locally in a function definition (see 3.5). For programming convenience (forcing evaluation, observation of unique objects and threading of sequential operations) special let constructions are provided (see 3.6).

## 3.1 Defining Functions

```
FunctionDef       =  [FunctionTypeDef] DefOfFunction     //  see Chapter 4 for typing functions
DefOfFunction     =  {FunctionAltDef}+
FunctionAltDef    =  Function {Pattern}                  //  see 3.2 for patterns
                     FunctionBody
                     [LocalFunctionAltDefs]              //  see 3.5
Function          =  FunctionName                        //  ordinary function
                  |  (FunctionName)                      //  operator function
FunctionBody      =  [LetBefores]                        //  see 3.6
                     FunctionRhs                         //  see 3.4
                     [LocalFunctionDefs]                 //  see 3.5
FunctionRhs       =  | [StrictLet] Guard                 //  see 3.3
                          FunctionBody
                          [FunctionBody]
                  |  =[>] RootExpression ;               //  see 3.4
```

A *function definition* consist of one or more definitions of *function alternatives* (rewrite rules) which are tried in textual order. On the left-hand side of such a function alternative a *pattern* can be specified which can serve a whole sequence of *guarded function bodies* (called the *rule alternatives*) The root expression (see 3.4) of a particular rule alternative is chosen for evaluation when

+ the pattern on the left-hand side matches the corresponding actual arguments of the function application (see 3.2) *and*
+ the optional *guard* (see 3.3) specified on the right-hand side evaluates to `True`.

A function can be preceded by a definition of its type (see 4.3).

- Function definitions are only allowed in implementation modules (see 2.3).
- It is required that the function alternatives of a function are textually grouped together (separated by semi-colons when the lay-out sensitive 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 2.1). However, it is possible to overload functions and operators (see 4.4).

- 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 3.4.1 and 4.3).
- Each alternative must use the same defining symbol (= or =>).

**Example** (function definition).

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

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

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

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

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

- When an operator is used in infix position *both* arguments have to be present. Operators can be used in a curried way, but then they have to be used as ordinary prefix functions (see also 2.3).

A *constant function definition* is a function defined with arity zero.

**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 `infix`). This is defined in its type (see 4.3.2). See also 3.4.1.

## 3.2 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 described. A *pattern* generally consists of some *data constructor* with its optional arguments which on their turn can contain sub-patterns (see 3.2.1). A *node-id variable* can be attached to a pattern (using the symbol '`=:`') which makes it possible to identify (*label*) the whole pattern as well as its contents *Bracketed patterns* are formal arguments that form a syntactic unit (see 3.2.2 - 3.2.6).

```
BrackPattern  =  (GraphPattern)          //  see 3.2.1
              |  Constructor             //  see 3.2.2
              |  _                       //  see 3.2.3
              |  BasicValuePattern       //  see 3.2.4
              |  ListPattern             //  see 3.2.5
              |  TuplePattern            //  see 3.2.6
              |  RecordPattern           //  see 3.2.7
              |  ArrayPattern            //  see 3.2.8
              |  PatternVariable
              |  Variable =: BrackPattern
```

- It is possible that the specified patterns turn a function into a partial function (see 4.3.3). 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.

### 3.2.1 Constructor Patterns

```
GraphPattern  =  Constructor {Pattern}   //  Constructor pattern
```

```
| GraphPattern ConstructorName GraphPattern   // Constructor operator
| Variable =: GraphPattern                    // named pattern
| Pattern                                     // a pattern in brackets
```

A *constructor pattern* (see above) consists of a constant tag called a *data constructor* (see 3.4.1 and 4.2.1) 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 4.2.1).

**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
```

Data constructors with arity two (see 3.1, see 4.2.1) 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 3.2.1 and 2.3).

**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
```

### 3.2.2 Simple Constructor Patterns

```
Constructor = ConstructorName
            | (ConstructorName)
```

*Constructor symbols* without arguments are just simple zero-arity constant. They form a syntactic unit (for non-operators no brackets are needed in this case). Besides the brackets that can be omitted they behave just like other data constructor patterns (see 3.4.2 and 3.2.1)

### 3.2.3 Variables and Wildcards in Patterns

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

```
PatternVariable = Variable
                | _
```

A *node variable* is a formal argument of a function which matches on *any* concrete value of the corresponding actual argument and therefore it does *not* force evaluation of this argument. A *wildcard* is an *anonymous* node variable ("_") one can use to indicate that the corresponding argument is not used in the right-hand side of the function. The formal arguments of a function and the function body are contained in a new scope. See also 3.4.3.

```
function args = body
```

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

**Example** (use of pattern variables).

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

realpart:: Complex -> Real
realpart (re,_) = re                            //  re and _ are pattern variables
```

### 3.2.4 Constant Values of Basic Type as Pattern

| | | |
|---|---|---|
| BasicValuePattern | = | BasicValue |
| BasicValue | = | *IntDenotation* |
| | \| | *RealDenotation* |
| | \| | *BoolDenotation* |
| | \| | *CharDenotation* |

A *constant value* of predefined *basic type* `Int`, `Real`, `Bool` or `Char` (see 4.1) can be specified as pattern

- The denotation of such a value must obey the syntactic description given in Section 3.4.4.

**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)
```

### 3.2.5 List Patterns

An object of the predefined algebraic type *list* (see 3.4.5 and 4.1.3) can be specified as pattern

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

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

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

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

### 3.2.6 Tuple Patterns

An object of the predefined algebraic type *tuple* (see 3.4.6 and 4.1.4) can be specified as pattern

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

### 3.2.7 Record Patterns

An object of type *record* (see 3.4.7 and 4.2.2) 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 | = | **{**[*TypeName***|**] {*FieldName* [**=** GraphPattern]}-*list***}** |

- The type of the record must have been defined in a record type definition (see 4.2.2).
- The field names specified in the pattern must be identical to the field names specified in the corresponding type.

- When matching a record, the type constructor which can be used to disambiguate the record from other records, can only be left out if there is *at least* one field name specified which is not being defined in some other record.

**Example** (use of record patterns).

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

### 3.2.8 Array Patterns

An object of type *array* (see 3.4.8 and 4.1.5) 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 = **{**{ArrayIndex **=** *Variable*}-*list***}**
| *StringDenotation*

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

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

**Example** (use of array patterns).

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

## 3.3 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 3.1). 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 emphasise 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 4.3.3). When a partial function is applied outside the domain for which the function is defined it will result into a *run-time* error. At compile time this cannot be detected.

**Example** (function definition with guards).

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

**Example** (equivalent definition).

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

Guards can be nested. When a guard on one level evaluates to `True`, the guards on a next level are tried.

- To ensure that at least one of the alternatives of a nested guard will be successful, a nested guarded alternative must always have a 'default' case as last alternative.

**Example** (Nested guard).

```
example arg1 arg2
| predicate11 arg1                                    //   if predicate11 arg1
     | predicate21 arg2 = calculate1 arg1 arg2       //   then (if predicate21 arg2 then ...
     | predicate22 arg2 = calculate2 arg1 arg2       //         elseif predicate22 arg2 then ...
                        = calculate3 arg1 arg2       //         else ...)
| predicate12 arg1      = calculate4 arg1 arg2       //   elseif predicate12 arg1 then ...
```

## 3.4 Expressions

The main body of a function is called the *root expression*. The root expression is a graph expression.

```
RootExpression  =  [StrictLet] GraphExpr
```

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

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

A graph expression generally expresses an application of a function to its arguments or the (automatic) creation of a data structure simply by applying a data constructor to its arguments (see 3.4.1). A case expression and conditional expression are added for notational convenience (see 3.4.10). With a let expression new functions and graphs can be locally defined in an expression (see 3.4.11). One can optionally demand the *interleaved or parallel evaluation* of the expression by another process or on another processor (see Chapter 5)

```
GraphExpr       =  [Process] Application          //  see 3.4.1
                |  [Process] CaseExpr             //  see 3.4.10
                |  [Process] LetExpr              //  see 3.4.11
Application     =  {BrackGraph}+                  //  see 3.4.1
                |  GraphExpr Operator GraphExpr    //  see 3.4.1
BrackGraph      =  SimpleGraph [Selections]       //  see 3.4.7 and 3.4.8 for selections
SimpleGraph     =  (GraphExpr)                    //  see 3.4.1
                |  ConstructorOrFunction          //  see 3.4.2
                |  GraphVariable                  //  see 3.4.3
                |  BasicValue                     //  see 3.4.4
                |  List                           //  see 3.4.5
                |  Tuple                          //  see 3.4.6
                |  Record                         //  see 3.4.7
                |  Array                          //  see 3.4.8
                |  LambdaAbstr                    //  see 3.4.9
```

### 3.4.1 Applications

```
Application     =  {BrackGraph}+                  //  application
                |  GraphExpr Operator GraphExpr    //  operator application
```

```
Operator        =  FunctionName
                |  ConstructorName
```

A (graph) *application* or graph *expression* in principle consists of the application of a *function* or *data constructor* to its (actual) arguments Each function or data constructor can be used in a *curried* way and can therefore be applied to any number (zero or more) of arguments (see 4.3). For convenience and efficiency special syntax is provided to denote values of data structures of predefined type (see 3.4.4 - 3.4.8). A function can only be rewritten if it is applied to a number of arguments equal to the arity of the function (see 3.1).

- All expressions have to be of correct type (see Chapter 4).
- All symbols that appear in an expression must have been defined somewhere within the scope in which the expression appears (see 2.1).

*Operators* are special functions or constructors defined with arity two (see 4.3.2) 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 4.3) 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 3.4.7 and 3.4.8) binds more tightly (`11`). Besides that, due to the priority, brackets can sometimes be omitted, operator applications behave just like other applications (see 3.4.1).

- 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 (applied to less than two arguments), 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.

### 3.4.2 Constructor or Function Name

```
ConstructorOrFunction  =  Constructor
                       |  Function
Function               =  FunctionName
                       |  (FunctionName)
Constructor            =  ConstructorName
                       |  (ConstructorName)
```

Function and constructors applied on zero arguments just form a syntactic unit (for non-operators no brackets are needed in this case). Besides the brackets that can be omitted they behave just like other applications (see 3.4.1)

### 3.4.3 Graph Variables

```
GraphVariable  =  Variable
               |  SelectorVariable
```

There are two kinds of variables which can appear in a graph expression: *variables* introduced as *formal argument* of a function (see 3.1 and 3.2) and *selector variables* (defined in a *selector* to identify parts of a graph expression, see 3.5.4)

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

### 3.4.4 Creating Constant Values of Basic Type

In a graph expression *constant values* of basic type `Int`, `Real`, `Bool` or `Char` can be created. These predefined types introduced for reasons of efficiency and convenience are treated in Section 4.1.1. There is a

special notation to denote a string (an unboxed array of characters, see 3.4.8) as well as to denote a list of characters (see 3.4.5).

```
BasicValue      = IntDenotation
                | RealDenotation
                | BoolDenotation
                | CharDenotation

IntDenotation   = [Sign]~{Digit}+                    // decimal number
                | [Sign]~ 0~{OctDigit}+              // octal number
                | [Sign]~ 0x~{HexDigit}+             // hexadecimal number
Sign            = + | -
RealDenotation  = [Sign~]{Digit~}+.{~Digit}+[~E[~Sign]{~Digit}+]
BoolDenotation  = True | False
CharDenotation  = CharDel~AnyChar/~CharDel.CharDel

AnyChar         = IdChar | ReservedChar | Special
ReservedChar    = (  |  )  |{  |  }  |  [  |  ]  |  ;  |  ,  |  .
Special         = \n  |  \r  |\f  |  \b          // newline,return,formf,backspace
                | \t  |  \\ | \CharDel           // tab,backslash,character delimiter
                | \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

CharDel         = '
```

**Example** (denotations).

```
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'|…
```

### 3.4.5 Creating Lists

Because lists are very convenient and frequently used data structure there are several syntactical constructs in CLEAN for creating lists including *dot-dot expression* and *list comprehensions*. The predefined *type* list is treated in Section 4.1.3.

```
List            = ListDenotation
                | DotDotExpression
                | ListComprehension
```

- A list expression must be of type list (see 4.1.3).
- All elements of a list must be of the same type.

#### *Simple Lists*

```
ListDenotation  = [[{LGraphExpr}-list [: GraphExpr]]]
LGraphExpr      = GraphExpr
                | CharsDenotation
CharsDenotation = CharDel~{AnyChar/~CharDel}+.CharDel
```

One way to create a list is by explicit enumeration of the list elements. List are constructed by adding one or more elements to an existing list. A special notation is provided for the frequently used *list of characters* (see also 3.2).

**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:[3,5,7,9]]
[1,3,5:[7,9]]
```

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

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

### *Dot-dot Expressions*

DotDotExpression = **[**GraphExpr [**,**GraphExpr]**..**[GraphExpr]**]**

With a dot-dot expression the list elements can be enumerated by giving the first element (`n1`), an optional second element (`n2`) and an optional last element (`e`). The generated list is calculated as follows:

```
_from_then_to:: !a !a !a -> .[a] | Enum a
_from_then_to n1 n2 e
| n1 <= n2     = _from_by_to n1 (n2-n1) e
               = _from_by_down_to n1 (n2-n1) e
where
     _from_by_to n s e
     | n<=e    = [n : _from_by_to (n+s) s e]
               = []

     _from_by_down_to n s e
     | n>=e    = [n : _from_by_down_to (n+s) s e]
               = []
```

The step size is `one` by default. If no last element is specified an infinite list is generated.

- Dot-dot expression can only be used if one imports `StdEnum` from the standard library.
- Dot-dot expressions are predefined on objects of type `Int`, `Real` and `Char`, but dot-dots can also be applied to any user defined data structure for which the class enumeration type has been instantiated (see CLEAN's STANDARD LIBRARY).

**Example** (Alternative ways to define a list a dot dot expression).

```
[1,3..9]                                        //   [1,3,5,7,9]
[1..9]                                          //   [1,2,3,4,5,6,7,8,9]
[1..]                                           //   [1,2,3,4,5 and so on…
['a'..'c']                                      //   ['abc']
```

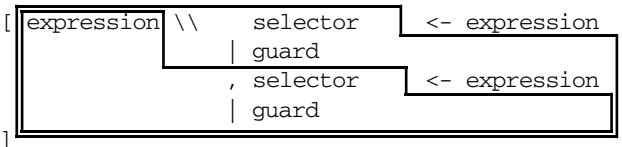
### *List Comprehensions*

```
ListComprehension   =  [GraphExpr \\ {Qualifier}-list]
Qualifier           =  Generators {|Guard}
Generators          =  Generator {& Generator}
Generator           =  Selector <- ListExpr
                    |  Selector <-: ArrayExpr
Selector            =  BrackPattern             //  for brack patterns see 3.2
ListExpr            =  GraphExpr
ArrayExpr           =  GraphExpr
Guard               =  BooleanExpr
BooleanExpr         =  GraphExpr
```

With a list comprehension one can construct a list composed from elements drawn from other lists or arrays. With a *list generator* one can draw elements from a list. With an *array generator* one can draw elements from an array. One can define several generators in a row separated by a comma. The last generator in such a sequence will vary first. One can also define several generators in a row separated by a '`&`'. All generators in such a sequence will vary at the same time but the drawing of elements will stop as soon of one the generators is exhausted. This construct can be used instead of the zip-functions which are commonly used. *Selectors* are simple patterns to identify parts of a graph expression. They are

explained in Section 3.5.4. Only those lists produced by a generator which match the specified selector are taken into account. Guards can be used as filter in the usual way

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




**Example** (list comprehension: `expr1` yields `[(0,0), (0,1), (0,2), (1,0), (1,1), (1,2), (2,0), (2,1), (2,2), (3,0), (3,1), (3,2)]` while `expr2` yields `[(0,0), (1,1), (2,2)]`. `expr3` yields `[(0,0), (1,0), (1,1), (2,0), (2,1), (2,2), (3,0), (3,1), (3,2), (3,3)]`)

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

**Example** (list comprehension: a well-known sort).

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

**Example** (list comprehension: converting an array into a list).

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

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

### 3.4.6 Creating 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 4.1.4.

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

**Example** (tuple).

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

### 3.4.7 Creating Records and Selection of Record Fields

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.

Record = RecordDenotation
| RecordUpdate

#### *Simple Records*

The first way is to create a record is by *explicitly* define a value for *each* of its fields.

RecordDenotation = **{**[*TypeName*|] {*FieldName***=** GraphExpr}-*list***}**

**Example** (Creation of a record ).

```
::Point       =    { x:: Real                        //    record type definition
                   , y:: Real
```

```
                    }
::ColorPoint   =    { p:: Point                        //    record type definition
                    , c:: Color
                    }
::Color        =    Red | Green | Blue                 //    algebraic type definition

CreateColorPoint:: (Real,Real) Color -> ColorPoint    //    type of function
CreateColorPoint (px,py) col =    { c = col      //    function creating a new record
                                  , p = { x = px
                                        , y = py
                                        }
                                  }
```

- A record can only be used if its type has been defined in a record type definition (see 4.2.2); the field names used must be identical to the field names specified in the corresponding type.
- When creating a record explicitly, the order in which the record fields are instantiated is irrelevant, but *all* fields have to get a value; the type of these values must be an instantiation of the corresponding type specified in record type definition. Curried use of records is *not* possible (see 4.2).
- When creating a record, its type constructor can be used to disambiguate the record from other records; the type constructor can be left out if there is *at least* one field name specified which is not being defined in some other record.

***Record Update***

The second way is to construct a new record out of an existing one (a *functional record update*).

| | | |
|---|---|---|
| RecordUpdate | = | **{**[ *TypeName* **\|** ] RecordExpr **&** {*FieldName* {Selection} **=** GraphExpr}-*list***}** |
| Selection | = | **.** [*TypeName***.**] *FieldName* |
| | **\|** | **.** ArrayIndex |
| RecordExpr | = | GraphExpr |

- The record expression must yield a record.

The record written to the left of the **&** (`r & f = v` is pronounced as: `r` with for `f` the value `v`) is the record to be duplicated. On the right from the `&` the structures are specified in which the new record *differs* from the old one. A structure can be any field of the record or a selection of any field or array element of a record or array stored in this record. All other fields are duplicated and created *implicitly*. Notice that the functional update is not an update in the classical, destructive sense since a *new* record is created. The functional update of records is performed very efficient such that we have not added support for destructive updates of records of unique type. The `&`-operator evaluates the existing record to root normal form before the update.

**Example** (Updating a record within a record using the functional update).

```
MoveColorPoint:: ColorPoint (Real,Real) -> ColorPoint
MoveColorPoint cp (dx,dy) = {cp & p.x = cp.p.x + dx, p.y = cp.p.y + dy}
```

***Selection of a Record Field***

| | | | | |
|---|---|---|---|---|
| BrackGraph | = | SimpleGraph [Selections] | | |
| Selection | = | **.** [*TypeName***.**] *FieldName* | | |
| | **\|** | **.** ArrayIndex | // | see 3.4.8 |
| Selections | = | {Selection}+ | | |
| | **\|** | **!** ArrayIndex {Selection}+ | // | see 3.4.8 |
| | **\|** | **!** [*TypeName***.**] *FieldName* {Selection}+ | | |

With a *record selection* (using the '.' symbol) one can select the value stored in the indicated record field. A "unique" selection using the '!' symbol returns a tuple containing the demanded record field and the original record. This type of record selection can be very handy for destructively updating of uniquely typed records with values which depend on the current contents of the record. Record selection binds more tightly (priority `11`) than application (priority `10`). Record selections can be nested and mixed with array selections (see 3.4.8).

**Example** (Record selection).

```
GetPoint:: ColorPoint -> Point
GetPoint cp = cp.p                                  //  selection of a record field

GetXPoint:: ColorPoint -> Real
GetXPoint cp = cp.p.x                               //  selection of a record field

GetXPoint2:: *ColorPoint -> (Real,.ColorPoint)
GetXPoint2 cp = cp!p.x                              //  selection of a record field
```

### 3.4.8 Creating Arrays and Selection of Array Elements

An *array* is a tuple/record-like data structure in which *all* elements are of the *same* type. Instead of selection by position or field name the elements of an array can be selected very efficiently in *constant time* by indexing . The update of arrays is done destructively in CLEAN and therefore arrays have to be unique (see Chapter 4) if one wants to use this feature. Arrays are very useful if time and space consumption is becoming very critical (CLEAN arrays are implemented very efficiently). If efficiency is not a big issue we recommend *not* to use arrays but to use lists instead: lists induce a much better programming style. Lists are more flexible and less error prone: array elements can only be accessed via indices and if you make a calculation error indices may point outside the array bounds. This is detected, but only at run-time. In CLEAN, array indices always start with 0. More dimensional arrays (e.g. a matrix) can be defined as an array of arrays.

For efficiency reasons, arrays are available of several types: there are *lazy arrays* (type `{a}`), *strict arrays* (type `{!a}`) and *unboxed arrays* for elements of basic type and record type (e.g. type `{#Int}`). All these arrays are considered to be of *different* type. By using the overloading mechanism (type constructor classes) one can still define (overloaded) functions which work on any of these arrays. The predefined *type* array is treated in Section 4.1.2.

```
Array           =  ArrayDenotation
                |  ArrayUpdate
                |  ArrayComprehension
```

- All elements of an array need to be of the same type.

#### *Simple Array*

A new array can be created in a number of ways. A direct way is to simply *list* the *array elements*.

```
ArrayDenotation     =  {{GraphExpr}-list}
                    |  StringDenotation                    //  see A.8
StringDenotation    =  StringDel~{AnyChar/~StringDel}~StringDel
StringDel           =  "
```

By default a *lazy* array will be created. Arrays are created *unique* (the `*` or. attribute in front of the type, see Chapter 4) to make destructive updates possible.

A lazy array is a box with pointers pointing to the array elements. One can also create a strict array (explicitly define its type as `{!Int}`), which will have the property that the elements to which the array box points will always be evaluated. One can furthermore create an unboxed array (explicitly define its type as `{#Int}`), which will have the property that the evaluated elements (which have to be of basic value) are stored directly in the array box itself. Clearly the last one is the most efficient representation (see also Chapter 5).

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

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

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

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

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

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

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

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

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

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

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

There are a number of handy functions for the creation and manipulation of arrays predefined in CLEANS STANDARD LIBRARY. These functions are overloaded to be able to deal with any type of array. The class restrictions for these functions express that "an array structure is required" containing "an array element".

**Example** (type of some predefined functions on `Arrays`).

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

### *Array Update*

It is also possible to construct a new array out of an existing one (a *functional array update*).

```
ArrayUpdate          =  { ArrayExpr & {ArrayIndex {Selection} = GraphExpr}-list [\\ {Qualifier}-list] }
ArrayComprehension   =  { GraphExpr \\ {Qualifier}-list}
Selection            =  . [TypeName.] FieldName
                     |  . ArrayIndex
ArrayExpr            =  GraphExpr
```

Left from the **&** (`a & [i] = v` is pronounced as: array `a` with for `a.[i]` the value `v`) the old array has to be specified which has to be of unique type to make destructive updating possible. On the right from the `&` those array elements are listed in which the new array differs from the old one. One can change any element of the array or any field or array element of a record or array stored in the array. The `&`-operator evaluates the array to root normal form before the update.

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

*Important*: For reasons of efficiency we have defined the updates only on arrays which are of *unique* type (`*{…}`), such that the update can always be done *destructively* (!) which is semantically sound because the original unique array is known not to be used anymore (see 4.5)

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

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

One can use an *array comprehension* or a *list comprehension* (see 3.4.5) to list these elements compactly in the same spirit as with a list comprehension.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Adj:: {{#Int}} ->.{.{#Int}}
Adj ma =  {  {ma.[i,j] \\ i <- rowindex}
```

```
          \\ j <- colindex
          }
where
     rowindex = [0..maxindex ma]
     colindex = [0..maxindex ma.[0]]

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

**Example** (updating unique arrays using a unique array selection).

```
MyArray:: .{#Real}
MyArray = {1.5,2.3,3.4}

ScaleArrayElem:: *{#Real} Int Real -> .{#Real}
ScaleArrayElem ar i factor
# (elem,ar) = ar![i]
= {ar & [i] = elem*factor}

Scale2DArrayElem:: {*{#Real}} (Int,Int) Real -> {.{#Real}}
Scale2DArrayElem ar (i,j) factor
# (elem,ar) = ar![i].[j]
= {ar & [i].[j] = elem*factor}

Scale2DArrayElem2:: {*{#Real}} (Int,Int) Real -> {.{#Real}}
Scale2DArrayElem2 ar (i,j) factor
# (elem,ar) = ar![i,j]
= {ar & [i,j] = elem*factor}
```

***Selection of an Array Element***

```
BrackGraph      =  SimpleGraph [Selections]
Selection       =  . [TypeName.] FieldName                    //  see 3.4.7
                |  . ArrayIndex
Selections      =  {Selection}+
                |  ! ArrayIndex {Selection}+
                |  ! [TypeName.] FieldName {Selection}+       //  see 3.4.7
```

With an *array selection* (using the '.' symbol) one can select an array element. When an object `a` is of type `Array`, the $i^{th}$ element can be selected (computed) via `a.[i]`. Array selection is left-associative: `a.[i,j,k]` means `((a.[i]).[j]).[k]`. A "unique" selection using the '`!`' symbol returns a tuple containing the demanded array element *and* the original array. This type of array selection can be very handy for destructively updating of uniquely typed arrays with values which depend on the current contents of the array. Array selection binds more tightly (priority `11`) than application (priority `10`). Array selections can be nested and mixed with record selections (see 3.4.7).

### 3.4.9 Lambda Abstraction

Sometimes it can be convenient to define a tiny function in an expression "right on the spot". For this purpose one can use a *lambda abstraction*. An anonymous function is defined which can have several formal arguments which can be patterns as common in ordinary function definitions (see Chapter 3). However, only simple functions can be defined in this way: no guards, no rule alternatives, no local definitions. Since the dot is already used for record and array selection a '`->`' is used to separate the formal arguments from the function body:

```
LambdaAbstr     =  \ {BrackPattern} -> GraphExpr
```

A lambda expression introduces a new scope (see 2.1).

```
\ args -> body
```

**Example** (lambda expression).

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

### 3.4.10 Case Expression and Conditional Expression

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

```
CaseExpr      =  case GraphExpr of
                 { {CaseAltDef}+ }
              |  if BrackGraph BrackGraph BrackGraph
CaseAltDef    =  Pattern
                 CaseBody
                 [LocalFunctionAltDefs]
CaseBody      =  [LetBefores] CaseRhs
                 [LocalFunctionDefs]
CaseRhs       =  | [StrictLet] Guard CaseBody [CaseBody]
              |  -> RootExpression ;
```

In a *case expression* first the discriminating expression is usually evaluated after which the case alternatives are tried in textual order. Case alternatives are similar to function alternatives. This is not so strange because a case expression is internally translated to a function definition (see the example below). Each alternative contains a left-hand side pattern (see 3.2) which is optionally followed by a *let-before* (see 3.6) and a guard (see 3.3). When a pattern matches and the optional guard evaluates to `True` the corresponding alternative is chosen. A new block structure (scope) is created for each case alternative (see 2.1).

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

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

**Example** (case expression).

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

is semantically equivalent to:

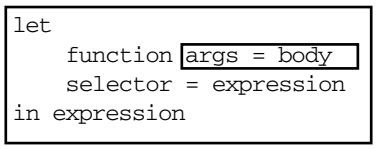
```
h x = mycase (g x)
where
     mycase  [hd:_] = hd
     mycase  []         = abort "result of call g x in h is empty"
```

In a *conditional expression* the first argument is evaluated to a Boolean value, if this value is True, the then-part (second argument) is chosen, otherwise the else-part (third argument) is chosen. The conditional expression can be seen as a simple kind of case expression.

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

### 3.4.11 Let Expression: Local Definitions for Expressions

Sometimes it is convenient to introduce local function definitions (see 3.5.3) or constant (graph) definitions (see 3.5.4) which are only visible for a certain expression. So, a *let* expression is an expressions which introduces a new scope (see 2.1).




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

LetExpresssion = **let** ⦃ {LocalDef}+ ⦄ **in** GraphExpr
LocalDef = GraphDef
| FunctionDef

**Example** (let expression used in a list comprehension).

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

## 3.5 Local Definitions

In a function definition one can locally define functions (see 3.5.3) and constant graphs (see 3.5.4).

LocalDef = GraphDef
| FunctionDef

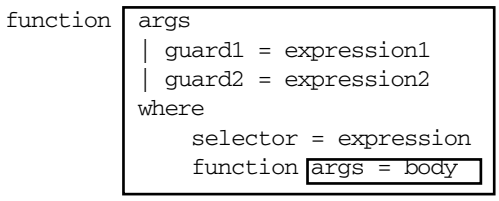
Both kind of local definitions can be introduced by using a *let* expression (see 3.4.11), by using a *where* block (see 3.5.1) or by using a *with* block (see 3.5.2). Constant graph definitions can also be defined by using a *strict let* expression (see 3.6.1), and, in a *let-before* expression(see 3.6.2).

### 3.5.1 Where Block: Local Definitions for a Function Alternative

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

LocalFunctionAltDefs = [ **where**] ⦃ {LocalDef}+ ⦄

Functions and graphs defined in a *where* block can be used anywhere in the corresponding function alternative (i.e. in all guards and rule alternatives following a pattern, see 3.1) as indicated in the following picture showing the scope of a *where* block.




**Example** (`sieve` and `filter` are local functions defined in a where block. They have only a meaning inside primes. At the global level the functions are unknown).

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

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

Notice that the scope rules are such that the arguments of the surrounding function alternative are visible to the locally defined functions and graphs. The arguments can therefore directly be addressed in the local definitions. Such local definitions cannot always be typed explicitly (see 4.3).

**Example** (An alternative definition of `primes`. The function `filter` is locally defined for `sieve`. `filter` can directly access argument `pr` of `sieve`).

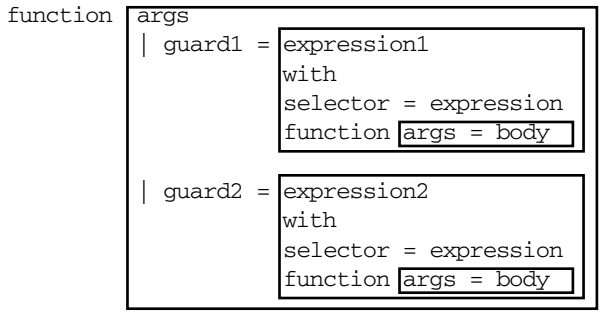
```
primes::[Int]
primes = sieve [2..]
where
     sieve::[Int] -> [Int]                                  //   local function of primes
     sieve [pr:r]   = [pr:sieve (filter r)]
     where
          filter::[Int] -> [Int]                            //   local function of sieve
          filter [n:r]
          | n mod pr == 0     = filter r
          | otherwise         = [n:filter r]
```

### 3.5.2 With Block: Local Definitions for a Guarded Alternative

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

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

Functions and graphs (see 3.5.4) defined in a *with* block can only be used in the corresponding rule alternative as indicated in the following picture showing the scope of a *with* block.




Notice that the scope rules are such that the arguments of the surrounding guarded rule alternative are visible to the locally defined functions and graphs. The arguments can therefore directly be addressed in the local definitions. Such local definitions cannot always be typed explicitly (see 4.3).

### 3.5.3 Defining Local Functions

One can define functions which have a local scope, i.e. which have only a meaning in a certain program region (see 3.4.11, 3.5.1, 3.5.3). Outside the scope the functions are unknown. This locality can be used to get a better program structure: functions which are only used in a certain program area can remain hidden outside that area. Programs can also become more readable because arguments of the surrounding function can directly be accessed in the local function body. Local functions therefore often need less arguments than functions defined on a global level (see 3.5.1). However, such local definitions cannot always be typed explicitly (see 4.3).

### 3.5.4 Defining Local Constants

One can give a name to a constant expression (actually a graph), such that the expression can be used in (and shared by) other expressions. One can also identify certain parts of a constant via a projection function called a selector (see below). Selectors are also used in list comprehensions and array comprehensions (see 3.4.5 and 3.4.8).

```
GraphDef            =  Selector =[:]  GraphExpr  #  [LocalFunctionAltDefs]
```

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

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

When a *graph* is *defined* actually a name is given to (part) of an expression. The definition of a graph can be compared with a definition of a *constant* (data) or a *constant* (*projection*) *function*. However, notice that graphs are constructed according to the basic semantics of CLEAN (see Chapter 1) which means that multiple references to the same graph will result in *sharing* of that graph. Recursive references will result in *cyclic graph structures*. Graphs have the property that they *are computed only once* and that their value is *remembered* within the scope they are defined in.

Graph definitions differ from constant function definitions. A *constant function definition* is just a function defined with arity zero (see 3.1). A constant function defines an ordinary graph rewriting rule: multiple references to a function just means that the same definition is used such that a (constant) function *will be recomputed again for each occurrence of the function symbol made*. This difference can have consequences for the time and space behaviour of function definitions (see 5.2).

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

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

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

Syntactically the definition of a graph is distinguished from the definition of a function by the symbol which separates left-hand side from right-hand side: "**=:**" is used for graphs while "**=>**" is used for functions. However, in general the more common symbol "**=**" is used for both type of definitions. Generally it is clear from the context what is meant (functions have parameters, selectors are also easy recognisable). However, when a simple constant is defined the syntax is ambiguous (it can be a constant function definition as well as a constant graph definition).

To allow the use of the "**=**" whenever possible, the following rule is followed. Locally constant definitions are *by default* taken to be *graph* definitions and therefore shared, globally they are *by default* taken to be *function* definitions (see 3.1) and therefore recomputed. If one wants to obtain a different behaviour one has to explicit state the nature of the constant definition (has it to be shared or has it to be recomputed) by using "`=:`" (on the global level, meaning it is a constant graph which is shared) or "`=>`" (on the local level, meaning it is a constant function and has to be recomputed).

**Example** (Local constant graph versus local constant function definition: `biglist1` and `biglist2` is a *graph* which is computed only once, `biglist3` is a constant *function* which is computed every time it is applied).

```
biglist1  =    [1..10000]                    //   a graph (if defined locally)
biglist2  =:   [1..10000]                    //   a graph
biglist3  =>   [1..10000]                    //   a constant function
```

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 1).

The left-hand side of a graph definition can be a simple name, but is can also be a more complicated pattern called a selector. A *selector* is a pattern which introduces one or more new *selector variables* implicitly defining *projection functions* to identify (parts of) a constant graph being defined One can identify the sub-graph as a whole or one can identify its components. A selector can contain constants (also user defined constants introduced by algebraic type definitions), variables and wildcards. With a *wildcard* one can indicate that one is not interested in certain components.

Selector = BrackPattern // for bracket patterns see 3.2

- 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 1.2).
- To avoid the specification of patterns which may fail at run-time, it is not allowed to test on zero arity constructors. For instance, list used in a selector pattern need to be of form `[a:_]`. `[a]` cannot be used because it stands for `[a:[]]` implying a test on the zero arity constructor `[]`. If the pattern is a record only those fields which contents one is interested in need to be indicated in the pattern
- Arrays cannot be used as pattern in a selector.

Remark: a selector can also appear on the left-hand side of a generator in a list comprehension (see 3.4.5) or array comprehension (see 3.4.8).

**Example** (use of a selectors to locally select tuple elements).

```
unzip::[(a,b)] -> ([a],[b])
unzip []           = ([],[])
unzip [(x,y):xys]  = ([x:xs],[y:ys])
where
     (xs,ys) = unzip xys
```

## 3.6 Special Local Definitions

In addition to ordinary *let* expressions there are also special *let* expressions with which one can locally define graphs (see 3.5.4), but not functions (!). These special let expressions are introduced for very specific reasons.

### 3.6.1 Strict Let Expression: Strict Local Constants

Although CLEAN is by default a lazy language one can force evaluation in several ways. By forcing evaluation one generally obtains a more time- and space-efficient program (see 5.1). Forcing evaluation can influence the termination behaviour of the program (a terminating program may be turned into a non-terminating program). See also Section 5.1.

The nicest way to force evaluation is by defining (partially) strict data structures (see 5.1). But it can also be handy to force evaluation on ad-hoc basis. This can be done by annotating function arguments as being strict (see 5.1.2). Another way to force evaluation is by using a *strict let expression*. The *strict let* expression looks similar to an ordinary *let* expression albeit that only graphs can be defined in a *strict let* expression which will be evaluated to strong root normal form before the root expression is being evaluated (see 3.5). To ensure that evaluation indeed takes place, a strict let expression can only be used before the root expression (which will be evaluated) and it can only contain graph definitions (which can be evaluated). The order in which the graphs in the let expression will be evaluated is undefined.

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

StrictLet = **let!** ⦃ { StrictLetGraphDef } ⦄ **in**
LetGraphDef = Selector **=[:]** GraphExpr # [LocalFunctionDefs]
StrictLetGraphDef = LetGraphDef

| GraphVariable #

**Example** (let! expression forcing evaluation).

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

### 3.6.2 Let-Before Expression: Local Constants for a Guard

Many of the functions for input and output in the CLEAN I/O library are state transition functions. Such a state is often passed from one function to another in a single threaded way (see Chapter 4) to force a specific order of evaluation. This is certainly the case when the state is of unique type. The threading parameter has to be renamed to distinguish its different versions. The following example shows a typical example:

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

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

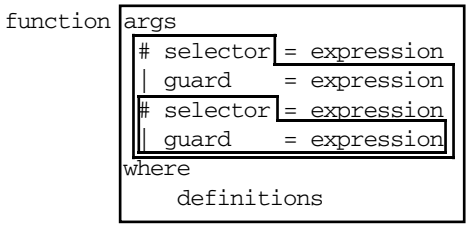
This explicit renaming of threaded parameters not only looks very ugly, these kind of definitions are sometimes also hard to read as well (in which order do things happen? which state is passed in which situation?). We have to admit: an imperative style of programming is much more easier to read when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

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

| | | |
|---|---|---|
| LetBefores | = | {LetBefore}+ |
| LetBefore | = | **#** {LetGraphDef}+ |
| | \| | **#!** {StrictLetGraphDef}+ |

The form with the exclamation mark forces the evaluation of the node-ids that appear in the left-hand sides of the definitions (see strict let-expressions, Section 3.6.1). Instead of the keyword `let` the `#`-symbol is used because it looks nice in combination with the `|`-symbol used for guards.

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




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

**Example** (use of let before expressions, reusing names taking use of the special scope of the let before)

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

**Example** (equivalent definition renaming threaded parameters)

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

A with block (see 3.5.2) may follow a*Let-before* expression. Functions and graphs defined in such a with block can only be used in the *Let-before* expression (and the with block).

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

**Example** (abuse of let before expression)

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

# 4

# Defining Types

**4.1 Predefined Types**
**4.2 Defining New Types**
**4.3 Typing Functions**
**4.4 Typing Overloaded Functions**
**4.5 Defining Uniqueness Types**

CLEAN is a strongly typed language. The basic type system of CLEAN is based on the classical polymorphic Milner/Hindley/Mycroft (Milner 1978; Hindley 1969, Mycroft, 1984) type system. This type system is adapted for graph rewriting systems and extended with *basic types*, (possibly *existentially quantified) algebraic types*, *record types*, *abstract types* and *synonym types*. These types are explained in the Sections 4.1, 4.2 and 4.3.
In CLEAN each classical type is furthermore extended with *uniqueness type attributes*. This very special and important extension is explained in Section 4.5.
CLEAN allows functions and operators to be *overloaded*. *Type classes* and type constructor classes are provided (which look similar to Haskell (Hudak *et al.* , 1992) and Gofer (Jones, 1993) although they have slightly different semantics) with which a restricted context can be imposed on a type variable in a type specification. This is explained in Section 4.4.
Although CLEAN is purely functional, operations with side-effects (I/O operations, for instance) are permitted. To achieve this without violating the semantics, the classical types are supplied with so called uniqueness attributes. This is explained in Section 4.5.

## 4.1 Predefined Types

CLEAN is a *strongly typed* language : every object (graph) and function (graph rewrite rule) in CLEAN has a type. The types of functions can be *explicitly specified* by the programmer or they can be *inferred automatically* (see 4.3.5) Types can be formed by taking instances of type constructors which have been defined explicitly as *algebraic type* (see 4.2.1), *record type* (see 4.2.2), *synonym type* (see 4.2.3), *abstract type* (see 4.2.4) or by taken instances of a *predefined type* (see 4.1.1 - 4.1.6). A *type instance* from a given type is obtained by uniformly substituting a type for a type variable. A type instance can be preceded by a uniqueness type attribute . This is further explained in Section 4.5.1.

```
Type          =  {[Strict] BrackType}+
              |  ArrowType                    //  see 4.1.6
BrackType     =  [UnqTypeAttrib] SimpleType
SimpleType    =  TypeConstructor              //  see 4.2, 4.4
              |  TypeVariable
              |  BasicType                    //  see 4.1.1
              |  PredefAbstrType              //  see 4.1.2
              |  ListType                     //  see 4.1.3
              |  TupleType                    //  see 4.1.4
              |  ArrayType                    //  see 4.1.5
              |  (Type)
```

### 4.1.1 Basic Types

*Basic types* are *algebraic types* (see 4.2) which are predefined for reasons of efficiency and convenience: `Int` (for 32 bits integer values), `Real` (for 64 bit double precision floating point values), `Char` (for 8 bits ASCII character values) and `Bool` (for 8 bits Boolean values). For programming convenience special syntax is introduced to denote constant values (data constructors) of these predefined types (see 3.2). Functions to create and manipulate objects of basic types can be found in the CLEAN library (as indicated below).

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

### 4.1.2 Predefined Abstract Types

As is explained in Section 4.2.4, *Abstract data types* are types of which the actual definition is hidden. In CLEAN the types `World` , `File` and `ProcId` are *predefined abstract data types*. They are recognised by the compiler and treated specially, either for efficiency or because they play a special role in the language. Since the actual definition is hidden it is not possible to denotate constant values of these predefined abstract types. There are functions predefined in the CLEAN library for the creation and manipulation of these predefined abstract data types. Some functions work (only) on unique objects (see Chapter 4).

An object of type **`*World`** (`*` indicates that the world is unique, see 4.5.1) is automatically created when a program is started. This object is optionally given as argument to the `Start` function (see 2.3). With this object efficient interfacing with the outside world (which is indeed unique) is made possible (see Chapter 4).

An object of type **`File`** or **`*File`** can be created by means of the functions defined in `StdFileIO` (see CLEANs Standard Library). It makes direct manipulation of persistent data possible. 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 CLEANs Standard Library). These objects are used in process annotations to allow process creation on an indicated processor (see Chapter 5) in a network topology.

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

### 4.1.3 List Types

A *list* is an algebraic data type predefined just for programming convenience. A list can contain an *infinite number* of elements. All elements must be of the *same type*. Lists are very often used in functional languages and therefore the usual syntactic sugar is provided for the creation and manipulation of lists (dot-dot expressions, list comprehensions) while there is also special syntax for *list of characters*. (see 3.4.5 and 3.2.5)

- 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 5.1.3). We are working on removing this restriction.

```
ListType  =  [Type]
```

### 4.1.4 Tuple Types

A *tuple* is an algebraic data type predefined for reasons of programming convenience and efficiency (see 5.1). 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 3.4.6 and 3.2.5). This is in particular handy for functions that return several values.

The tuple arguments can optionally be annotated as being strict (see 5.1.1). This can be used to increase the efficiency of a program (see 5.1). The compiler will automatically take care of the conversion between lazy and strict tuples where needed (see 5.1.4).

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

### 4.1.5 Array Types

An *array* is an algebraic data type predefined for reasons of efficiency. Arrays contain a *finite number* of elements that all have to be of the *same type*. An array has as property that its elements can be accessed via *indexing* in *constant time*. An *array index* must be an integer value between `0` and the number of elements of the array-`1`. Destructive updates of array elements is possible thanks to uniqueness typing. For programming convenience special syntax is provided for the creation, selection and updating of array elements (array comprehensions) while there is also special syntax for *strings* (i.e. unboxed arrays of characters) (see 3.4.8 and 3.2.8). Arrays have as disadvantage that their use increases the possibility of a run-time error (indices that might get out-of-range). Again, see 3.4.8 and 3.2.8.

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

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

### 4.1.6 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 4.3) to obtain a compact notation. Curried functions applications and types are automatically converted to their uncurried equivalent versions (see 4.3.1)

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

**Example** (of an arrow type).

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

being equivalent with:

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

## 4.2 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 4.2.4 and Chapter 2).

```
TypeDef     = AlgebraicTypeDef      // see 4.2.1 and 4.5.2
            | RecordTypeDef         // see 4.2.2 and 4.5.2
            | SynonymTypeDef        // see 4.2.3 and 4.5.2
            | AbstractTypeDef       // see 4.2.4 and 4.5.2
FunctionDef = [FunctionTypeDef] DefOfFunction   // see 4.3 and 4.5.3
ClassDef    = TypeClassDef          // see 4.4 and 4.5.4
```

| TypeInstanceDef // see 4.4 and 4.5.4

### 4.2.1 Defining Algebraic Data Types

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

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

Types can be preceded by uniqueness type attributes (see 4.5.2). The arguments of a defined data constructor can optionally be annotated as being strict (see 5.1).

AlgebraicTypeDef = **::**TypeLhs **=** [QuantifiedVars] ConstructorDef {|[QuantifiedVars] ConstructorDef} **#**

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

ConstructorDef = *ConstructorName* {[Strict] BrackType}
| **(***ConstructorName***)** [FixPrec] {[Strict] BrackType}
QuantifiedVars = **E.** {[**.**]*TypeVariable*}+**:**
FixPrec = Fix [Prec]
Fix = **infixl**
| **infixr**
| **infix**
Prec = *Digit*

**Example** (algebraic type definition and its use).

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

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

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

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

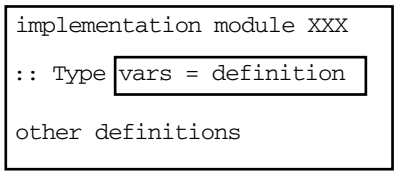



**Figure** (Scope of type definitions).

#### *Defining Infix Data Constructors*

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

default 9). Infix constructors can also be used in prefix position when they are surrounded by brackets (see 3.1).

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

```
::List a  = (<:>) infixr 5 a (List a)
          | Nil

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

### *Using Higher Order Types*

In an algebraic type definition ordinary types can be used (such as a basic type, e.g. `Int`, or a list type, e.g. `[Int]`, or an instantiation of a user defined type, e.g. `Tree Int`), but one can also use *higher order types*. Higher order types can be constructed by curried applications of the type constructors. Higher order types can be applied in the type world in a similar way as higher order functions in the function world. The use of higher order types increases the flexibility with which algebraic types can be defined. Higher order types play an important role in combination with type classes (see 4.4).

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

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

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

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

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

In CLEAN each *top level* type should have kind `X`. A top level type is a type that occurs either as an argument or result type of a function or as argument type of a data constructor (in some algebraic type definition). The rule for determining the kinds of the type variables (which can be of any order) are fairly simple: The kind of a type variable directly follows from its use. If a variable has no arguments, its kind

is X. Otherwise, its kind corresponds to the number of arguments to which the variable is applied. The kind of type variable determines its possible instantiations, i.e. it can only be instantiated with a type which is of the same kind as the type variable itself.

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

```
::Tree2 t = NilTree
          | NodeTree (t Int) (Tree2 t) (Tree2 t)

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

### *Defining Algebraic Data Types with Existentially Quantified Variables*

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

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

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

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

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

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

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

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

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

### *Semantic Restrictions on Algebraic Data Types*

Other semantic restrictions on algebraic data types:

- The name of a type must be different from other names in the same scope and name space (see 2.1).
- All type variables on the left-hand side must be different.
- All type variables used on the right-hand side are bound, i.e. must be introduced on the left-hand side of the algebraic type being defined.
- A data constructor can only be defined once within the same scope and name space. So, each data constructor unambiguously identifies its type to make type inferrencing possible.

- When a data constructor is used in infix position both arguments have to be present. Data constructors can be used in a curried way in the function world, but then they have to be used as ordinary prefix constructors.
- Type constructors can be used in a curried way in the type world; to use predefined bracket-like type constructors (for lists, tuples, arrays) in a curried way they must be used in prefix notation.
- The right-hand side of an algebraic data type definition should yield a type of kind x, all arguments of the data constructor being defined should be of kind x as well.
- A type can only be instantiated with a type that is of the same kind.
- An existentially quantified type variable specified in an algebraic type can only be instantiated with a concrete type (= not a type variable) when a data structure of this type is created.

### 4.2.2 Defining Record Types

A *record type* is basically an algebraic data type in which exactly one constructor is defined. Special about records is
- that a *field name* is attached to each of the arguments of the data constructor;
- that records cannot be used in a curried way.

Compared with ordinary algebraic data structures the use of records gives a lot of notational convenience because the field names enable *selection by field name* instead of *selection by position* When a record is created *all* arguments of the constructor have to be defined but one can specify the arguments in *any* order (see 3.4.7). Furthermore, when pattern matching is performed on a record, one only has to mention those fields one is interested in (see 3.2.6). A record can be created via a functional update (see 3.4.7). In that case one only has to specify the values for those fields which differ from the old record. Matching and creation of records can hence be specified in CLEAN in such a way that after a change in the structure of a record only those functions have to be changed which are explicitly referring to the changed fields.

Existential type variables (see 3.2.1) are allowed in record types (as in any other type). The arguments of the constructor can optionally be annotated as being strict (see 5.1). The optional uniqueness attributes are treated in 4.5.2.

RecordTypeDef = **::** TypeLhs **=** [QuantifiedVars] **{** {*FieldName* **::** [Strict] Type}-*list* **}**

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

**Example** (record definition).

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

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

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

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


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

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

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

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

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

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

    Start:: [Object]
    Start = map (Set 3) [CreateObject1,CreateObject2]
```

### 4.2.3 Defining Synonym Types

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

SynonymTypeDef = **::** TypeLhs **:==** [QuantifiedVars]Type ;

- For the left-hand side the same restrictions hold as for algebraic types (see 4.2.1).
- Cyclic definitions of synonym types (e.g. `::T a b :== G a b; ::G a b :== T a b`) are not allowed.

**Example** (type synonym definition).

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

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

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

### 4.2.4 Defining Abstract Data Types

A type can be exported by defining the type in a CLEAN definition module (see Chapter 2). For software engineering reasons it sometimes better only to export the name of a type but not its concrete definition (the right-hand side of the type definition). The type then becomes an *abstract data type*. In CLEAN this is done by specifying only the left-hand-side of a type in the definition module while the concrete definition (the right-hand side of the type definition) is hidden in the implementation module. So, CLEAN's module structure is used to hide the actual implementation. When one wants to do something useful with objects of abstract types one needs to export functions that can create and manipulate objects of this type as well.

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

AbstractTypeDef = **::** TypeLhs ;

**Example** (abstract data type).

```
definition module stack

::Stack a
```

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

implementation module stack

::Stack a :== [a]

Empty:: (Stack a)
Empty = []

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

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

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

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

## 4.3 Typing Functions

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

```
FunctionDef       =  [FunctionTypeDef] DefOfFunction

FunctionTypeDef   =  FunctionName :: FunctionType ;
                  |  (FunctionName) [FixPrec] [:: FunctionType] ;
FixPrec           =  Fix [Prec]
Fix               =  infixl
                  |  infixr
                  |  infix
Prec              =  Digit
FunctionType      =  Type [ClassContext] [UnqTypeUnEqualities]
```

An explicit specification is *required* when a function is exported, or when the programmer wants to impose additional restrictions on the application of the function (e.g. a more restricted type can be specified, strictness information can be added as explained in Section 5.1, a class context for the type variables can be defined as explained in Section 4.4, uniqueness information can be added as explained in Section 4.5.3). The CLEAN type system uses a combination of Milner/Mycroft type assignment. This has as consequence that the type system in some rare cases is not capable to infer the type of a function (using the Milner/Hindley system) although it will approve a given type (using the Mycroft system; see Plasmeijer and Van Eekelen, 1993; see also the example in 4.2.1).

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

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

**Example** (arity of a function reflected in type).

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

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

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

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

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

### 4.3.1 Typing Curried Functions

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

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

### 4.3.2 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 infix). 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 2.3 and 3.1.

- 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)
```

### 4.3.3 Typing Partial Functions

Patterns and guards imply a condition that has to be fulfilled before a rewrite rule can be applied (see 3.2 and 3.3). This makes it possible to define *partial function s*, functions which are not defined for all possible values of the specified type.

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

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

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

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

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

## 4.4 Typing Overloaded Functions

The names of the functions one defines generally all have to be *different* within the same scope and name space (see 2.1). However, it is sometimes very convenient to *overload* certain functions and operators (e.g. `+`, `-`, `==`), i.e. use *identical* names for *different* functions or operators that perform *similar tasks* albeit on objects of *different types*.

In principle it is possible to simulate a kind of overloading by using records. One simply defines a record (see 4.2.2) in which a collection of functions are stored that somehow belong to each other. Now the field name of the record can be used as (overloaded) synonym for any concrete function stored on the corresponding position. The record can be regarded as a kind of *dictionary* in which the concrete function can be looked up.

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

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

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

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

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

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

### 4.4.1 Type Classes

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

TypeClassDef = **class** *ClassName* [**.**] *TypeVariable* [ClassContext]
[**where** ⦃ {ClassMemberDef}+ ⦄]
| **class** *FunctionName* [**.**] *TypeVariable***::** FunctionType #
| **class (***FunctionName***)** [FixPrec] [**.**] *TypeVariable***::** FunctionType #

ClassMemberDef = FunctionTypeDef #
[MacroDef # ]

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

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

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

TypeClassInstanceDef = **instance** *ClassName* [Type [**default** | ClassContext]]
[**where** ⦃ {DefOfFunction}+ ⦄]

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

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

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

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

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

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

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

### 4.4.2 Functions Defined in Terms of Overloaded Functions

When an overloaded name is encountered in an expression, the compiler will determine which of the corresponding concrete functions/operators is meant by looking at the concrete type of the expression. This type is used to determine which concrete function to apply. All instances of the overloaded type variable of a certain class (with exception of the default instance, see below) must therefore not overlap (being not unifyable) with each other and they all have to be of flat type (see the restrictions mentioned in 4.4.11). If it is clear from the type of the expression which one of the concrete instantiations is meant the compiler will in principle substitute the concrete function for the overloaded one, such that no efficiency is lost.

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

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

will be internally transformed into

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

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

For instance, the function

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

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

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

| | | |
|---|---|---|
| FunctionType | = | Type [ClassContext] [UnqTypeUnEqualities] |
| ClassContext | = | \| *ClassName-list* [**.**]*TypeVariable* {**&** *ClassName-list* [**.**]*TypeVariable* } |

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

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

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

### 4.4.3 Instances of Type Classes Defined in Terms of Overloaded Functions

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

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

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

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

**Example** (Equality class).

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

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

### 4.4.4 Type Constructor Classes

The CLEAN type system offers the possibility to use higher order types (see 4.2.1). This makes it possible to define *type constructor classes* (similar to constructor classes as introduced in Gofer, Jones (1993)). In that case the overloaded type variable of the type class is not of kind `X`, but of higher order, e.g. `X -> X`, `X -> X -> X`, etcetera. This offers the possibility to define overloaded functions which can be instantiated with type constructors of higher order (as usual, the overloaded type variable and a concrete instantiation of this type variable need to be of the same kind). This makes it possible to overload more complex functions like `map` and the like.

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

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

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

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

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

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

### 4.4.5 Generic Instances

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

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

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

### 4.4.6 Default Instances

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

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

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

```
class Read  a:: a -> String
class Write a:: String -> a
instance Read  Int, Bool                                    // export of class instance, see 4.4.10
instance Write Int, Bool

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

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

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

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

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

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

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

### 4.4.7 Defining Derived Members in a Class

The members of a class consists of a set of functions or operators which logically belong to each other. It is often the case that the effect of some members (*derived members*) can be expressed in others. For instance, `<>` can be regarded as synonym for `not (==)`. For software engineering (the fixed relation is made explicit) and efficiency (one does not need to include such derived members in the dictionary record) it is good to make this relation explicit. In CLEAN the existing macro facilities (see Chapter 5) are used for this purpose.

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

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

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

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

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

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

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

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

     max:: a a -> a | Ord a
```

```
max x y :== if (x<y) y x
```

### 4.4.8 A Shorthand for Defining Overloaded Functions

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

| | | |
|---|---|---|
| TypeClassDef | = | **class** *ClassName* [**.**] *TypeVariable* [ClassContext]<br>[**where** ⦃ {ClassMemberDef}+ ⦄]<br>\| **class** *FunctionName* [**.**] *TypeVariable***::** FunctionType ; ;<br>\| **class (***FunctionName***)** [FixPrec] [**.**] *TypeVariable***::** FunctionType ; ; |

**Example** (defining an overloaded function/operator).

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

which is shorthand for:

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

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

**Example** (instantiations of an overloaded function/operator).

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

### 4.4.9 Classes Defined in Terms of Other Classes

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

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

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

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

class Arith a | +,- a
```

### 4.4.10 Exporting Type Classes

To export a class one simply repeats the class definition in the definition module (see Chapter 2). To export an instantiation of a class one simply repeats the instance definition in the definition module, however *without* revealing the concrete implementation (which can only be specified in the implementation module).

**Example** (Exporting classes and instances).

```
definition module example

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

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

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

TypeClassInstanceExportDef
= **export** *ClassName* {BasicType | TypeVariable}-*list*‖

**Example** (Exporting class instances).

```
export Eq Int, Real
```

### 4.4.11 Semantic Restrictions on Type Classes

Semantic restrictions:

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

### 4.4.12 The Costs of Overloading

In Section 4.4 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 4.4).

Overloading can cause code explosion. When in a certain function another overloaded function is applied in such a way that the type system *cannot* deduce which concrete instance of the overloaded function has to be used the system will in principle generate *several versions* of the function: *one* version

is made for *each* of the concrete (combination of) instances possible. In principle special versions will only be generated for instantiations of basic types. Although the system avoids to generate versions that are not being used, code explosion might occur when all versions are being used or when the system simply cannot tell which versions are used. The latter can be the case when such functions are being exported to other modules.

Overloading can cause inefficiency. Instances which are recursively defined in terms of the class itself can lead to an infinite amount of concrete instances. New instances can also be declared in modules that import the overloaded function. To handle all these cases the system will generate one special version of the overloaded function which is parametrised with a type class record (see the introduction of 4.4). 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.

## 4.5 Defining Uniqueness Types

Although CLEAN is purely functional, operations with side-effects (I/O operations, for instance) are permitted. To achieve this without violating the semantics, the classical types are supplied with so called uniqueness attributes. If an argument of a function is indicated as unique, it is guaranteed that at run-time the corresponding actual object is local, i.e. there are no other references to it. Clearly, a destructive update of such a "unique object" can be performed safely.

The uniqueness type system makes it possible to define direct interfaces with an operating system, a file system (updating persistent data), with GUI's libraries, it allows to create arrays, records or user defined data structures that can be updated destructively. The time and space behaviour of a functional program therefore greatly benefits from the uniqueness typing.

Uniqueness types are deduced automatically. Type attributes are polymorphic: attribute variables and inequalities on these variables can be used to indicate relations between and restrictions on the corresponding concrete attribute values.

Sometimes the inferred type attributes give some extra information on the run-time behaviour of a function. The uniqueness type system is a transparent extension of classical typing which means that if one is not interested in the uniqueness information one can simply ignore it.

Since the uniqueness typing is a rather complex matter we explain this type system and the motivation behind it in more detail. The first Section (4.5.1) explains the basic motivation for and ideas behind uniqueness typing. Section 4.5.2 focuses on the so-called uniqueness propagation property of (algebraic) type constructors. Then we show how new data structures can be defined containing unique objects (Section 4.5.3). Sharing may destroy locality properties of objects. In Section 4.5.4 we describe the effect of sharing on uniqueness types. In order to maintain referential transparency, it appears that function types have to treated specially. The last Section (4.5.5) describes the combination of uniqueness typing and overloading. Especially, the subsections on constructor classes and higher-order type definitions are very complex: we suggest that the reader skips these sections at first instance.

### 4.5.1 Basic Ideas Behind Uniqueness Typing

The *uniqueness typing* is an extension of classical Milner/Mycroft typing. In the uniqueness type system *uniqueness type attributes* are attached to the classical types. Uniqueness type attributes appear in the *type specifications of functions* (see 4.5.4) but are also permitted in the definitions of *new data types* (see 4.5.3). A classical type can be prefixed by one of the following uniqueness type attributes:

```
Type          = {[Strict] BrackType}+
              | ArrowType
```

```
ArrowType       =  {BrackType}+ -> Type
BrackType       =  [TypeAttrib] SimpleType
UnqTypeAttrib   =  *                          //  type attribute "unique"
                |  UniqueTypeVariable:        //  a type attribute variable
                |  .                          //  an anonymous type attribute variable
```

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

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

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

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

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

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

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

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

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

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

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

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

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

- Function arguments with no uniqueness attributes added to the classical type are considered as "non-unique": there are no reference requirements for these arguments. The function is only al-

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

lowed to have *read access* (as usual in a functional language) even if in some of the function applications to actual argument appears to have reference count 1.

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

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

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

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

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

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

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

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

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

### 4.5.2 Attribute Propagation

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

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

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

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

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

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

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

would still be valid although it delivers the same object twice. By demanding that the surrounding list becomes unique as well, (so the type of `head` becomes `head:: *[*a] -> *a`) the function `heads` is rejected. In general one could say that uniqueness *propagates outwards.*

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

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

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

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

'non-unique ≤ unique'.

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

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

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

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

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

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

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

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

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

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

### 4.5.3 Defining New Types with Uniqueness Attributes

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

AlgebraicTypeDef = **::** TypeLhs **=** [QuantifiedVars] ConstructorDef {**|** [QuantifiedVars] ConstructorDef} **;**

```
TypeLhs          = [*]TypeConstructor {[*] TypeVariable}
TypeConstructor  = TypeName

ConstructorDef   = ConstructorName {[Strict] BrackType}
                 | (ConstructorName) [FixPrec] {[Strict] BrackType}
QuantifiedVars   = E. {[.]TypeVariable}+:
BrackType        = [UnqTypeAttrib] SimpleType
UnqTypeAttrib    = *
                 | UniqueTypeVariable
                 | .
```

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

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

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

leads to the following types for its data constructors.

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

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

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

**Example** (standard attribuation for data constructors).

For `Cons` the standard attribution leads to the type

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

The type of `Nil` becomes

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

Consider the following Tree definition

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

The type of the data constructor `Node` is

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

Another `Tree` variant.

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

leading to

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

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

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

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

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

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

### 4.5.4 Uniqueness and Sharing

The type inference system of CLEAN will derive uniqueness information *after* the classical Milner/Mycroft types of functions have been inferred (see 4.3). As explained in Section 4.5.1, a function may require a *non-unique* object, a *unique* object or a *possibly unique* object. Uniqueness of the result of a function will depend on the attributes of its arguments and how the result is constructed. Until now, we distinguished objects with reference count 1 from objects with a larger reference count: only the former might be unique (depending on the uniqueness type of the object itself). In practice, however, one can be more liberal if one takes the evaluation order into account. The idea is that multiple reference to an (unique) object are harmless if one knows that only one of the references will be present at the moment it is accessed destructively. This has been used in the following function.

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

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

For this reason, the uniqueness type system uses a kind of *sharing analysis* . This sharing analysis is input for the uniqueness type system itself to check uniqueness type consistency (see 4.5.3). The analysis will label each *reference* in the right-hand side of a function definition as *read-only* (if destructive access might be dangerous) or *write-permitted* (otherwise). Objects accessed via a read-only reference are always non-unique. On the other hand, uniqueness of objects accessed via a reference labelled with *write-permitted* solely depend on the types of the objects themselves.

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

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

The rules used by the sharing analysis to label each reference are the following.
- A reference, say r, to a certain object is labelled with read-only if there exist another reference, say r', to the same object such that either r is observing w.r.t r' or r and r' are in parallel.
- Multiple references to *cyclic structures* are always labelled as read-only.
- All other references are labelled with write-permitted.

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

Consider, for example, the following rule

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

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

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

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

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

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

#### *Higher Order Uniqueness Typing*

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

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

in the application

```
fwritec unifile
```

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

```
WriteAB write_fun = (write_fun 'a', write_fun 'b')
```

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

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

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

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

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

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

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

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

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

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

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

#### *Uniqueness Type Coercions*

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

$$\text{'unique'} \leq \text{'non-unique'}$$

on attributes. Roughly, the validity of $\sigma \leq \sigma'$ depends subtype-wise on the validity of $u \leq u'$ with $u,u'$ attributes in $\sigma,\sigma'$. One has, for example

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

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

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

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

**Example**: a has sign T, b has sign + in

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

This leads to the following coercion rules

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

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

### 4.5.5 Combining Uniqueness Typing and Overloading

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

Consider, for instance, the identity class

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

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

```
instance id L v:[u:a] -> v:[u:a]
```

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

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

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

Another possibility is

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

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

### *Constructor Classes*

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

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

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

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

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

Example

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

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

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

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

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

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

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

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

This type will provide that for an application like

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

indeed yields a spine unique list.

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

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

Consider the following function

```
DoubleMap f l = (map f l, map f l)
```

The type of this function is something like

```
DoubleMap:: (.a ->.b) (m.a) -> (.(m.b),.(m.b))
```

Clearly, `l` is duplicated. However, this does not necessarily mean that `a` cannot be unique anymore. If, for instance, `m` is instantiated with a non-propagating type constructor (like T as defined on the previous page) then uniqueness of a is still permitted. On the other hand, if `m` is instantiated with a propagating type constructor, a unique instantiation of a should be disapproved. In CLEAN, the type system `remembers' sharing of objects (like `l` in the above example) by making the corresponding type attribute non-unique. Thus, the given type for `DoubleMap` is exactly the type inferred by CLEAN's type system. If one tries to instantiate `m` with a propagating type constructor, and, at the same type, `a` with some unique type, this will fail.

The presence of higher-order class variables, not only influences propagation properties of types, but also the coercion relation between types. These type coercions depend on the sign classification of type constructors. The problem with higher-order polymorphism is that in some cases the actual type constructors substituted for the higher order type variables are unknown, and therefore one cannot decide whether coercions in which higher-order type variable are involved, are valid.

Consider the functions

```
double x = (x,x)
dm f l = double (map f l)
```

Here, `map`'s result (of type `(m.a)`) is coerced to the non-unique supertype `(m.a)`. However, this is only allowed if `m` is instantiated with type constructors that have no coercion restrictions. E.g., if one tries to substitute `*WriteFun` for `m`, where

```
WriteFun a = C.(a -> *File)
```

this should fail, for, `*WriteFun` is *essentially* unique. The to solve this problem is to restrict coercion properties of type variable applications `(m σ)` to

$$u{:}(m\ \sigma) \leq u{:}(m\ \tau) \text{ iff } \sigma \leq \tau \ \&\& \ \tau \leq \sigma$$

A slightly modified version of this solution has been adopted in CLEAN. For convenience, we have added the following refinement. The instances of type constructors classes are restricted to type constructors with no coercion restrictions. Moreover, it is assumed that these type constructors are uniqueness propagating. This means that the `WriteFun` cannot be used as an instance for `map`. Consequently, our coercion relation we can be more liberal if it involves such class variable applications.

Overruling this requirement can be done adding the anonymous attribute. the class variable. E.g.

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

Now

```
instance map WriteFun
where
     map..
```

is valid, but the coercions in which (parts of) `map`'s type are involved are now restricted as explained above.

To see the difference between the two indicated variants of constructor variables, we slightly modify map's type.

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

Without overruling the instance requirement for m the type of dm (dm as given on the previous page) becomes.

```
dm:: (.a ->.b) *(m.a) ->.(m b, m b)
```

Observe that the attribute of disappeared due to the fact that each type constructor substituted for m is assumed to be propagating.

If one explicitly indicates that there are no instance restriction for the class variable m (by attributing m with.), the function dm becomes untypable.

### 4.5.6 Higher-Order Type Definitions

We will describe the effect of uniqueness typing on type definitions containing higher-order type variables. At it turns out, this combination introduces a number of difficulties which would make a full description very complex. But even after skipping a lot of details we have to warn the reader that some of the remaining parts are still hard to understand.

As mentioned earlier, two properties of newly defined type constructor concerning uniqueness typing are important, namely, propagation and sign classification. One can probably image that, when dealing with higher-order types the determination on these properties becomes more involved. Consider, for example, the following type definition.

```
::   T m a = C (m a)
```

The question whether T is propagating in its second argument cannot be decided by examining this definition only; it depends on the actual instantiation of the (higher-order) type variable m. If m is instantiated with a propagating type constructor, like [], then T becomes propagating in its second argument as well. Actually, propagation is not only a property of type constructors, but also of types themselves, particularly of `partial types' For example, the partial type [] is propagating in its (only) argument (Note that the number of arguments a partial type expects, directly follows from the kinds of the type constructors that have been used). The type T [] is also propagating in its argument, so is the type T ((,) Int).

The analysis in CLEAN that determines propagation properties of (partial) types has been split into two phases. During the first phase, new type definitions are examined in order to determine the propagation dependencies between the arguments of each new type constructor. To explain the idea, we return to our previous example.

```
::   T m a = C (m a)
```

First observe that the propagation of the type variable m is not interesting because m does not stand for `real data' (which is always of kind *). We associate the propagation of m in T with the position(s) of the occurrence(s) of m's applications. So in general, T is propagating in a higher-order variable m if one of m's applications appears on a propagating position in the definition of T. Moreover, for each higher order type variable, we determine the propagation properties of all first order type variables in the following way: m is propagating in a, where m and a are higher-order respectively first-order type variables of T, if a appears on a propagating position in one of m's applications. In the above example, m is propagating in a, since a is on a propagating position in the application (m a). During the second phase, the propagation properties of (partial) types are determined using the results of the first phase. This (roughly) proceeds as follows. Consider the type T σ for some (partial) type σ, and T as defined earlier. First, determine (recursively) the propagation of σ. Then the type T σ is propagating if (1) σ is propagating, (2) T is propagating in m, and moreover (3) m is propagating in a (the second argument of the type constructor). With T as defined above, (2) and (3) are fulfilled. Thus, for example T [] is propagating and therefore also T (T []). Now define

```
::   T2 a = C2 (a -> Int)
```

The `T T2` is not propagating.

The adjusted uniqueness propagation rule (see also...) becomes:

- Let σ,τ be two uniqueness types. Suppose σ has attribute `u`. Then, if τ is propagating the application (τ σ) should have an attribute `v` such that `v` ≤ `u`.

Some of the readers might have inferred that this propagation rule is a 'higher-order' generalisation of the old 'first-order' propagation rule.

As to the sign classification, we restrict ourselves to the remark that that sign analysis used in CLEAN is adjusted in a similar way as described above for the propagation case.

Example

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

The sign classification of `T` if (-,⊥). Here ⊥ denotes the fact the `a` is neither directly used on a positive nor on a negative position. The sign classification of `m` w.r.t. `a` is +. The partial type `T []` has sign -, which e.g. implies that

```
T [] Int ≤ T [] *Int
```

The type `T T2` (with `T2` as defined on the previous page) has sign +, so

```
T T2 Int ≥ T T2 *Int
```

It will be clear that combining uniqueness typing with higher-order types is far from trivial: the description given above is complex and moreover incomplete. However explaining all the details of this combination is far beyond the scope of the reference manual.

### 4.5.7 Destructive Updates using Uniqueness Typing

So, it is *allowed* to update a uniquely typed function argument (`*`) destructively when the argument does not reappear in the function result. The question is: when does the compiler indeed make use of this possibility.

Destructive updates takes place in some predefined functions and operators which work on predefined data structures such arrays (&-operator) and files (writing to a file). Arrays and files are intended to be updated destructively and their use can have a big influence on the space and time behaviour of your application (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).

Performing destructive updates is only sensible when information is stored in nodes. Arguments of basic type (`Int`, `Real`, `Char` or `Bool`) are stored on the B-stack or in registers and it therefore does not make sense to make them unique.

The CLEAN compiler also has an option to re-use user-defined unique data structures: the space being occupied by a function argument of unique type will under certain conditions be reused destructively to construct the function result. 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 4). This means that one might have to restructure parts of the program to maintain the unicity of objects.

The compiler will do compile-time garbage collection for user defined unique data-structures only in certain cases. In that case run-time garbage collection time is reduced. It might even drop to zero. It

also possible that you gain even more then just garbage collection time, because updating part of a data structure can often be done with fewer instructions than creating a new one, or because cache behaviour improves.

The compiler will reuse uniquely typed data structures under the following conditions:

1. The pattern of the function contains a unique constructor of an algebraic data type, or a boxed record, with at least one argument. A strict record is usually passed unboxed.
2. The function does not contain other references to this constructor.
3. A node is required to construct the result of the function, that has a size less than or equal to the size of the matched unique node in the pattern.
4. This node is not the root node of the result or allocated inside a case expression, lambda expression, comprehension or local function of this function.

For example:

```
reverse_and_append :: *[.a] u:[.a] -> u:[.a];
reverse_and_append [h : t] list = reverse_and_append t [h : list];
reverse_and_append [] list = list;
```

will reuse the `[h : t]` node to construct `[h : list]`, by replacing argument `t` by `list`.

# 5

# Annotations and Directives

5.1 Defining Partially Strict Data Structures and Functions
5.2 Defining Graphs on the Global Level
5.3 Defining Macros
5.4 Process Annotations
5.5 Efficiency Tips

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.
In this Chapter we explain several kinds of annotations and directives which can be defined in CLEAN. These annotations and directives are designed to give the programmer some means to influence the time and space behaviour of CLEAN applications.
CLEAN is by default a *lazy* language: applications will only be evaluated when their results are needed for the final outcome of the program. However, lazy evaluation is in general not very efficient. It is much more efficient to compute function arguments in advance (*strict* evaluation) when it is known that the arguments will be used in the function body. By using strictness annotations in type definitions the evaluation order of data structures and functions can be changed from lazy to strict. This is explained in Section 5.1.
One can define constant graphs on the global level also known as *Constant Applicative Forms* (see Section 5.2). Unlike constant functions, these constant graphs are shared such that they are computed only one. This generally reduces execution time possibly at the cost of some heap space needed to remember the shared graph constants.
Macros (Section 5.3) are special functions which will already be substituted (evaluated) at *compile-time*. This generally reduces execution time (the work has already be done by the compiler) but it will lead to an increase of object code.
By using process annotations (See 5.4) one can express that a CLEAN expression may be evaluated in parallel. This can be used to speed-up CLEAN applications or be used to develop distributed applications.

## 5.1 Annotations to Change Lazy Evaluation into Strict Evaluation

CLEAN uses by default a *lazy evaluation strategy*: a redex is only evaluated when it is needed to compute the final result. Some functional languages (e.g. ML, Harper *et al.*) use a *eager* (*strict*) evaluation strategy and always evaluate all function arguments in advance.

### 5.1.1 Advantages and Disadvantages of Lazy versus Strict Evaluation

Lazy evaluation has the following advantages (+) / disadvantages (-) over eager (strict) evaluation:
+ 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);
+ one can work with infinite data structures (e.g. `[1..]`);
- 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]`);

### 5.1.2 Strict and Lazy Context

Each expression in a function definition 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 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 or strict let-before 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 (manually or automatically) or in the type definition of the data constructor;
+ all the other expressions are lazy.

Evaluation of a function will happen in the following order: patterns, guard, expressions in a strict let (before) expression, root expression (see also 3.1 and 4.5.4).

### 5.1.3 Space Consumption in Strict and Lazy Context

The space occupied by CLEAN structures depends on the kind of structures one is using, but also depends on whether these data structures appear in a strict or in a lazy context. To understand this one has to have some knowledge about the basic implementation of CLEAN (see Plasmeijer and Van Eekelen, 1993).

Graphs (see Chapter 1) are stored in a piece of memory called the heap. The amount of heap space needed highly depends on the kind of data structures which are in use. Graph structures which are created in a lazy context can occupy more space than graphs created in a strict context. Graphs which are not being used are automatically collected by the garbage collector in the run-time system of CLEAN. The arguments of functions being evaluated are stored on a stack. There are two stacks: the A-stack which contains references to graph nodes stored in the heap and the BC-stack which contains arguments of basic type and return addresses. Data structures in a *lazy context* are passed via references on the *A*-stack. 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 graph node (*boxed*) The amount of space occupied is also depending on the arity of the function.

In the table below the amount of space consumed in the different situations is summarised (for the lazy as well as for the strict context). For the size of the elements one can take the size consumed in a strict context.

| *Type* | *Arity* | *Lazy context (bytes)* | *Strict context (bytes)* | *Comment* |
|---|---|---|---|---|
| `Int, Bool` | - | 8 | 4 | |
| `Int (0≤n≤32), Char` | - | - | 4 | node is shared |
| `Real` | - | 12 | 8 | |
| `Small Record` | n | 4 + Σ `size elements` | Σ `size elements` | total length≤12 |
| `Large Record` | n | 8 + Σ `size elements` | Σ `size elements` | |

| | | | | |
|---|---|---|---|---|
| Tuple | 2 | 12 | Σ size elements | |
| | >2 | 8 + 4*n | Σ size elements | |
| 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 | |
| Function | 0,1,2 | 12 | - | |
| | >3 | 4 + 4*n | - | |
| Pointer to node | - | 4 | 4 | |
| {a}, {!a} | n | 20 + 4*n | 12 + 4*n | |
| {#Int} | n | 20 + 4*n | 12 + 4*n | |
| {#Char} | n | 16 + 4*ceil(n/4) | 8 + 4* ceil (n/4) | |
| {#Bool} | n | 20 + 4* ceil (n/4) | 12 + 4* ceil (n/4) | |
| {#Real} | n | 20 + 8*n | 12 + 8*n | |
| {#Record} | n | 20 + Σsize rec elems*n | 12 + Σsize rec elems*n | |

### 5.1.4 Time Consumption in Strict and Lazy Context

Strict arguments of functions can sometimes be handled much more efficiently than lazy arguments, in particular when the arguments are of basic type.

**Example** (functions with strict arguments of basic type are more efficient).

```
Ackermann:: !Int !Int -> Int
Ackermann 0 j = j+1
Ackermann i 0 = Ackermann (i-1) 1
Ackermann i j = Ackermann (i-1) (Ackermann i (j-1))
```

The computation of a lazy version of `Ackermann 3 7` takes 14.8 seconds + 0.1 seconds for garbage collection on an old fashioned MacII (5Mb heap). When both arguments are annotated as strict (which in this case will be done automatically by the compiler) the computation will only 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.

### 5.1.5 Changing Lazy into Strict Evaluation

So, lazy evaluation gives a notational freedom (no worrying about what is computed when) but it might cost space as well as time. In CLEAN the default lazy evaluation can therefore be turned into eager evaluation by adding strictness annotations to types.

```
Strict = !
```

This can be done in several ways:

+ 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 internally as strict (see 5.1.1). In this way lazy arguments are *automatically* turned into strict ones. This optimisation does not influence the termination behaviour of the program. It appears that the analyser can find much information. The analysis itself is quite fast.
+ The strictness analyser cannot find all strict arguments. Therefore one can also *manually* annotate a function as being strict in a certain argument or in its result (see 5.1.1).
+ By using strictness annotations, a programmer can define (partially) strict data structures (Nöcker and Smetsers, 1993; see 5.1.3). Whenever such a data structure occurs in a strict context (see 5.1.1), its strict components will be evaluated.
+ The order of evaluation of expressions in a function body can also be changed from lazy to strict by using a strict let expression or a strict let-before expression (see 3.4).

*One has to be careful though. When a programmer manually changes lazy evaluation into strict evaluation, 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.*

#### *Functions with Strict Arguments*

In the type definition of a function the arguments can optionally be annotated as being strict.

```
FunctionType  =  Type [ClassContext] [UnqTypeUnEqualities]
Type          =  {[Strict] BrackType}+
              |  ArrowType
```

In reasoning about functions it will always be true that the corresponding arguments will be in strong root normal form (see 2.1) before the rewriting of the function takes place.

**Example** (a function with strict annotated arguments).

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

The CLEAN compiler includes a fast and clever strictness analyser which is based on abstract reduction (Nöcker, 1993). The compiler can derive the strictness of the function arguments in many cases, such as for the example above. Therefore there is generally no need to add strictness annotations to the type of a function by hand. When a function is exported from a module (see Chapter 2), its type has to be specified in the definition module. To obtain optimal efficiency, the programmer should also include the strictness information to the type definition in the definition module. One can ask the compiler to print out the types with the derived strictness information and paste this into the definition module.

Notice that strictness annotations are only allowed at the outermost level of the argument type. Strictness annotations inside type instances of arguments are not possible (except for some predefined types). Any (part of) a data structure can be changed from lazy to strict, but this has to be specified in the type definition (see 5.1.3).

#### *Strictness Annotations in Type Definitions*

Functional programs will generally run much more efficient when strict data structures are being used instead of lazy ones. If the inefficiency of your program becomes problematic one can think of changing lazy data structures into strict ones. This has to be done by hand in the definition of the type.

```
AlgebraicTypeDef  =  ::TypeLhs = [QuantifiedVars] ConstructorDef {|[QuantifiedVars] ConstructorDef} //
RecordTypeDef     =  ::TypeLhs = [QuantifiedVars] {{FieldName :: [Strict] Type}-list} //
ConstructorDef    =  ConstructorName {[Strict] BrackType}
                  |  (ConstructorName) [FixPrec] {[Strict] BrackType}
```

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

So, in a type definition one can define a data constructor to be strict in zero or more of its arguments. Strictness is a property of data structure which is specified in its type. In general (with the exceptions of

tuples) one cannot arbitrary mix strict and non-strict data structures because they are considered to be of different type.

**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}
```

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

### *Strictness Annotations on Instances of Predefined Type*

Functions arguments can be annotated as being strict (by hand or automatically, see 5.1.2), new types can be defined as (partially) being strict (see 5.1.3). How about function arguments of predefined type (see 4.1)?

It is important to understand that in CLEAN a data structure with strict components is considered to be of different type than the same data structure with lazy components. For user defined data structures this does not cause any conflicts because the strictness of any instance obeys the strictness properties as specified in the corresponding type definition.

Function arguments of *basic type* or *predefined abstract type* do not contain any (known) substructure and can easily and can without problems be made strict just by annotating the corresponding function argument as being strict. Things are much more complicated for lists, tuples and arrays which *do* contain substructure. How can we change the strictness properties of these substructure since we do not have access to the definition of the predefined type? Well, strictness has to be specified in the type instances instead of the type definition. But, strict versions are of different type than lazy ones. To be able to handle these similar data structures of different type in a uniform way, some conversion has to take place. In the current version of CLEAN strict/lazy version of lists, tuples and arrays are all treated differently (we are working on it).

### *Strictness Annotations on Tuple Instances*

Strictness annotation can be put on any tuple element of any tuple instance (see also 4.14).

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

One can turn a lazy tuple element into a strict one by putting strictness annotations in the corresponding type instance on the tuple elements that one would like to make strict. When the corresponding tuple is put into a strict context the tuple and the strict annotated tuple elements will be evaluated. As usual one has to take care that these elements do not represent an infinite computation.

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

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

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

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

which is equivalent to

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

when for instance G is defined as

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

than the following application is approved by the type system:

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

### *Strictness Annotations on Array Instances*

For reasons of efficiency there are different types of arrays predefined.

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

One can define a lazy array (default, of type `{a}`), a strict array (explicitly type the array as `{!a}`), and an unboxed one (explicitly type the array as `{#a}`, works only on elements of basic value, records and arrays). When put in a strict context, all the elements of a strict array will be evaluated automatically. As usual one has to take care that the elements do not represent an infinite computation.

Lazy, strict and unboxed arrays are regarded to be of different type even if the array elements are of the same type. So, in principle one cannot offer e.g. a strict array to a function demanding a lazy one, and the other way around. Both will give rise to a type error. However, by using the overloading mechanism one can define functions which work on any kind of array (see 2.9).

**Example** (strict and non-strict arrays). `ArrayA` is a strict one and `ArrayB` is a lazy one. The function `Scale` expects a lazy one and can therefore only be applied on a lazy array. `ArrayA` is accepted but `ArrayB` is not accepted as argument of `Scale`. If one wants to define a function which works on any kind of array of Reals, one has to define an overloaded function (see 2.9) like `Scale2`.

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

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

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

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

### *Strictness Annotations on List Instances*

The current version of the CLEAN compiler does not allow to turn the standard lazy lists into strict ones by adding annotations in a type instance. In a future version this will change.

```
ListType   =  [Type]
```

So, if one wants to use a list with strict elements or a spine strict list one has to define a new list using an algebraic data type. This has as disadvantage that one cannot simply use the nice predefined notation for standard lists (list comprehensions and the like).

**Example** (user defined 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** (user defined spine strict list).

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

## 5.2 Defining Graphs on the Global Level

Constant graphs can also be defined on a global level (for local constant graphs see 3.5.4).

| GraphDef = Selector =[:] GraphExpr ; [LocalFunctionAltDefs]

A *global graph definition* defines a global constant (closed) graph, i.e. a graph which has the same scope as a global function definition (see 2.1). 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.

Syntactically the definition of a graph is distinguished from the definition of a function by the symbol which separates left-hand side from right-hand side: "**=:**" is used for graphs while "**=>**" is used for functions. However, in general the more common symbol "**=**" is used for both type of definitions. Generally it is clear from the context what is meant (functions have parameters, selectors are also easy recognisable). However, when a simple constant is defined the syntax is ambiguous (it can be a constant function definition as well as a constant graph definition).

To allow the use of the "**=**" whenever possible, the following rule is followed. Locally constant definitions are *by default* taken to be *graph* definitions and therefore shared, globally they are *by default* taken to be *function* definitions (see 3.1) and therefore recomputed. If one wants to obtain a different behaviour one has to explicit state the nature of the constant definition (has it to be shared or has it to be recomputed) by using "=:" (on the global level, meaning it is a constant graph which is shared) or "=>" (on the local level, meaning it is a constant function and has to be recomputed).

**Example** (Global constant graph versus global constant function definition: `biglist1` is a *graph* which is computed only once, `biglist3` and `biglist2` is a constant *function* which is computed every time it is applied).

```
biglist1  =    [1..10000]                    //   a constant function (if defined globally)
biglist2  =:   [1..10000]                    //   a graph
biglist3  =>   [1..10000]                    //   a constant function
```

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.

## 5.3 Defining Macros

Macros are functions (rewrite rules) which are applied at *compile-time* instead of at *run-time*. Macros can be used to define constants, create in-line substitutions, rename functions, do conditional compilation etc. 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 of a function definition.

At compile-time the right-hand side of the *macro definition* will be substituted for every application of the macro in the scope of the macro definition. This saves a function call and makes basic blocks larger (see Plasmeijer and Van Eekelen, 1993) such that *better* code can be generated. A disadvantage is that also *more* code will be generated. Inline substitution is also one of the regular optimisations performed by the CLEAN compiler. To avoid code explosion a compiler will generally not substitute big functions.

Macros give the programmer a possibility to control the substitution process manually to get an optimal trade-off between the efficiency of code and the size of the code.

| | | |
|---|---|---|
| **MacroDef** | = | [MacroFixityDef] DefOfMacro |
| MacroFixityDef | = | **(***FunctionName***)** Fix [Prec] // |
| DefOfMacro | = | Function {*Variable*} **:==** GraphExpr // [LocalFunctionAltDefs] |

The compile-time substitution process is guaranteed to terminate. To ensure this some restrictions are imposed on Macros (compared to common functions). Only variables are allowed as formal argument. A macro rule always consists of a single alternative. Furthermore,

- Macro definitions are not allowed to be cyclic to ensure that the substitution process terminates.

**Example** (macros):

```
Black     :== 1                                  //  Macro definition
White     :== 0                                  //  Macro definition

:: Color  :== Int                                //  Type synonym definition

Invert:: Color -> Color                          //  Function definition
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                                     //  Fixity of Macro
(?) h t :== [h:t]                                //  Macro definition of operator

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

Notice that macros can contain local function definitions. These local definitions (which can be recursive) will also be substituted inline. In this way complicated substitutions can be achieved resulting in efficient code.

**Example** (example: macros can be used to speed up frequently used functions. See for instance the definition of the function `foldl` in `StdList`).

```
foldl op r l :== foldl r l                       //  Macro definition
where
     foldl r []     = r
     foldl r [a:x]  = foldl (op r a) x

sum list = foldl (+) 0 list
```

After substitution of the macro `foldl` a very efficient function `sum` will be generated by the compiler:

```
sum list = foldl 0 list
where
     foldl r []     = r
     foldl r [a:x]  = foldl ((+) r a) x
```

The expansion of the macros takes place before type checking. Type specifications of macro rules is not possible. When operators are defined as macros, fixity and associativity can be defined.

## 5.4 Process Annotations

There are two ways of creating processes in CLEAN.

One way is by creating interactive applications. These interactive "processes" actually consist of a collection of call-back functions which are applied automatically when certain events occur. The call-back functions are applied by the I/O system sequentially one after another. Hence, scheduling takes place

by the I/O system on the level of call-back functions which perform a state transition in an indivisible action. Interactive processes are explained in Standard Libraries for CLEAN (Achten *et al.*, 1997).

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

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).

CLEAN processes are lightweight processes which run very efficient. Time-slicing, scheduling and communication is controlled by the CLEAN run-time system. Arbitrary process topologies can be created (e.g. cyclic process topologies) beyond the divide (fork) and conquer parallelism generally offered.

*The concurrency features of CLEAN (mail us for information) are currently only supported for a network of Macintosh (Motorola 680x0). We are working on this. There is also a parallel version running on Transputers. See our internet pages.*

### 5.4.1 Process Creation

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

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

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

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

With the `{|P|}` annotation a new *parallel reducer* is created. This reducer is *preferably* located on a *different* processor working on a lazy copy of the corresponding sub-graph. Reducers that are located on different processors run in parallel with each other. The `{|P|}` annotations can be extended with a *location directive* `at location`, where `location` is an expression of predefined type `ProcId` indicating the pro-

cessor on which the parallel process has to be created. In the library `StdProcId` functions are given that yield an object of this type.

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

### 5.4.2 Process Communication

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

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

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

**Example** (hierarchical process topology creation).

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

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

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

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

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

## 5.5 Efficiency Tips

Here are some additional suggestions how to make your program more efficient:

+ Use the CLEAN profiler to find out which frequently called functions are consuming a lot of space and/or time. If you modify your program, these functions are the one to have a good look at.
+ Transform a recursive function to a tail-recursive function.

+ Accumulate results in parameters instead of in right-hand side results.
+ When functions return multiple ad-hoc results in a tuple put these results in a strict tuple instead (can be indicated in the type).
+ It is usually better to use strict records instead of strict tuples in data structures.
+ Arrays can be more efficient than lists since they allow constant access time on their elements and can be destructive updated.
+ Use strict or unboxed data structures whenever possible (see 5.1.5).
+ Export the strictness information to other modules (the compiler will warn you if you don't).
+ Make functions strict in its arguments whenever possible (see. 5.1.5)
+ Use macros for simple constant expressions or frequently used functions.
+ Use CAFs and local graphs to avoid recalculation of expressions.
+ Selections in a lazy context can better be transformed to functions which do a pattern match.
+ Higher order functions are nice but inefficient (the compiler will try to convert higher order functions into first order functions).
+ Constructors of high arity are inefficient.
+ Increase the heap space in the case that garbage collection uses a lot of time.

# A

# Context-Free Syntax Description

**A.1 CLEAN Program**
**A.2 Import Definition**
**A.3 Function Definition**
**A.4 Macro Definition**
**A.5 Type Definition**
**A.6 Class Definition**
**A.7 Names**
**A.8 Denotations**

In this appendix the context-free syntax of CLEAN is given. Notice that the lay-out rule (see 2.3.3) 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 following notational conventions are used in the context-free syntax descriptions:

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

## A.1 CLEAN Program

```
CLEANProgram          =  {Module}+
Module                =  DefinitionModule
                      |  ImplementationModule
DefinitionModule      =  definition module ModuleName ;
                         {DefDefinition}
                      |  system module ModuleName ;
                         {DefDefinition}
ImplementationModule  =  [implementation] module ModuleName ;
                         {ImplDefinition}
```

```
ImplDefinition        =  ImportDef            // see A.2
                      |  FunctionDef          // see A.3
                      |  GraphDef             // see A.3
                      |  MacroDef             // see A.4
                      |  TypeDef              // see A.5
                      |  ClassDef             // see A.6
```

```
DefDefinition         =  ImportDef            // see A.2
                      |  FunctionTypeDef      // see A.3
                      |  MacroDef             // see A.4
                      |  TypeDef              // see A.5
                      |  ClassDef             // see A.6
                      |  TypeClassInstanceExportDef  // see A.6
```

## A.2 Import Definition

```
ImportDef           = ImplicitImportDef
                    | ExplicitImportDef
ImplicitImportDef   = import {ModuleName}-list ,
ExplicitImportDef   = from ModuleName import {Imports}-list ,          // see A.7
Imports             = FunctionName                                     // see A.7
                    | ConstructorName                                  // see A.7
                    | SelectorVariable                                 // see A.7
                    | FieldName                                        // see A.7
                    | MacroName                                        // see A.7
                    | TypeName                                         // see A.7
                    | ClassName                                        // see A.7
```

## A.3 Function Definition

```
FunctionDef         = [FunctionTypeDef] DefOfFunction

FunctionTypeDef     = FunctionName :: FunctionType ,
                    | (FunctionName) [FixPrec] [:: FunctionType] ,
FunctionType        = Type [ClassContext] [UnqTypeUnEqualities]
ClassContext        = | ClassName-list [.]TypeVariable {& ClassName-list [.]TypeVariable }
UnqTypeUnEqualities = ,[{{UniqueTypeVariable}+ <= UniqueTypeVariable}-list]

DefOfFunction       = {FunctionAltDef}+
FunctionAltDef      = Function {Pattern}
                      FunctionBody
                      [LocalFunctionAltDefs]

Function            = FunctionName
                    | (FunctionName)
BrackPattern        = (GraphPattern)
                    | Constructor
                    | BasicValuePattern
                    | ListPattern
                    | TuplePattern
                    | RecordPattern
                    | ArrayPattern
                    | PatternVariable
                    | Variable =: BrackPattern

GraphPattern        = ConstructorName {Pattern}
                    | GraphPattern ConstructorName GraphPattern
                    | Variable =:GraphPattern
                    | Pattern
Constructor         = ConstructorName                                  // see A.7
                    | (ConstructorName)

PatternVariable     = Variable
                    | _

BasicValuePattern   = BasicValue
BasicValue          = IntDenotation                                    // see A.8
                    | RealDenotation                                   // see A.8
                    | BoolDenotation                                   // see A.8
                    | CharDenotation                                   // see A.8

ListPattern         = [[{LGraphPattern}-list [: GraphPattern]]]
LGraphPattern       = GraphPattern
                    | CharsDenotation                                  // see A.8

TuplePattern        = (GraphPattern,{GraphPattern}-list)

RecordPattern       = {[TypeName|] {FieldName [= GraphPattern]}-list}

ArrayPattern        = {{ArrayIndex = GraphPattern }-list}
                    | StringDenotation                                 // see A.8

LetBefores          = {LetBefore}+
```

```
LetBefore           =  # {LetGraphDef}+
                    |  #!  {StrictLetGraphDef}+

GraphDef            =  Selector =[:]  GraphExpr #  [LocalFunctionAltDefs]
LetGraphDef         =  Selector =[:]  GraphExpr #  [LocalFunctionDefs]
Selector            =  BrackPattern
StrictLetGraphDef   =  LetGraphDef
                    |  GraphVariable #

Guard               =  BooleanExpr
BooleanExpr         =  GraphExpr

StrictLet           =  let! {{ { StrictLetGraphDef } }} in

FunctionBody        =  [LetBefores] FunctionRhs
                       [LocalFunctionDefs]
FunctionRhs         =  | [StrictLet] Guard FunctionBody [FunctionBody]
                    |  =[>] RootExpression ;

RootExpression      =  [StrictLet] GraphExpr

GraphExpr           =  [Process] Application
                    |  [Process] CaseExpr
                    |  [Process] LetExpr

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

Application         =  {BrackGraph}+
                    |  GraphExpr Operator GraphExpr
Operator            =  FunctionName                          // see A.7
                    |  ConstructorName                       // see A.7
BrackGraph          =  SimpleGraph [Selections]
SimpleGraph         =  (GraphExpr)
                    |  ConstructorOrFunction
                    |  GraphVariable
                    |  BasicValue
                    |  List
                    |  Tuple
                    |  Record
                    |  Array
                    |  LambdaAbstr

ConstructorOrFunction =  Constructor
                      |  Function

GraphVariable       =  Variable                              // see A.7
                    |  SelectorVariable                      // see A.7

List                =  ListDenotation
                    |  DotDotExpression
                    |  ListComprehension
ListDenotation      =  [[{LGraphExpr}-list [: GraphExpr]]]
LGraphExpr          =  GraphExpr
                    |  CharsDenotation                       // see A.8
DotDotExpression    =  [GraphExpr [,GraphExpr]..[GraphExpr]]
ListComprehension   =  [GraphExpr \\ {Qualifier}-list]
Qualifier           =  Generators { | Guard}
Generators          =  Generator {& Generator}
Generator           =  Selector <- ListExpr
                    |  Selector <-: ArrayExpr
ListExpr            =  GraphExpr
ArrayExpr           =  GraphExpr

Tuple               =  (GraphExpr,{GraphExpr}-list)

Record              =  RecordDenotation
                    |  RecordUpdate
RecordDenotation    =  {[TypeName|] {FieldName= GraphExpr}-list}
```

```
RecordUpdate      = {[TypeName|] RecordExpr & {FieldName {Selection} = GraphExpr}-list}
Selection         = . [TypeName.] FieldName
                  | . ArrayIndex
Selections        = {Selection}+
                  | ! ArrayIndex {Selection}+
                  | ! [TypeName.] FieldName {Selection}+
RecordExpr        = GraphExpr
ArrayExpr         = GraphExpr

Array             = ArrayDenotation
                  | ArrayUpdate
                  | ArrayComprehension
ArrayDenotation   = {{GraphExpr}-list}
                  | StringDenotation                                  // see A.8
ArrayUpdate       = { ArrayExpr & {ArrayIndex {Selection} = GraphExpr}-list [\\ {Qualifier}-list]}
ArrayComprehension= { GraphExpr \\ {Qualifier}-list}
ArrayIndex        = [{IntegerExpr}-list]
IntegerExpr       = GraphExpr

LambdaAbstr       = \ {BrackPattern} -> GraphExpr

CaseExpr          = case GraphExpr of
                    {| {CaseAltDef}+ |}
                  | if BrackGraph BrackGraph BrackGraph
CaseAltDef        = Pattern
                    CaseBody
                    [LocalFunctionAltDefs]
CaseBody          = [LetBefores] CaseRhs
                    [LocalFunctionDefs]
CaseRhs           = | [StrictLet] Guard CaseBody [CaseBody]
                  | -> RootExpression ;

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

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

## A.4 Macro Definition

```
MacroDef          = [MacroFixityDef] DefOfMacro
MacroFixityDef    = (FunctionName) Fix [Prec] #
DefOfMacro        = Function {Variable} :== GraphExpr #
                    [LocalFunctionAltDefs]
```

## A.5 Type Definition

```
TypeDef           = AlgebraicTypeDef
                  | RecordTypeDef
                  | SynonymTypeDef
                  | AbstractTypeDef

AlgebraicTypeDef  = ::TypeLhs = [QuantifiedVars] ConstructorDef {|[QuantifiedVars] ConstructorDef} #
RecordTypeDef     = ::TypeLhs = [QuantifiedVars] {{FieldName :: [Strict] Type}-list} #
SynonymTypeDef    = ::TypeLhs :== [QuantifiedVars]Type #
AbstractTypeDef   = ::TypeLhs #

TypeLhs           = [*]TypeConstructor {[*] TypeVariable}
TypeConstructor   = TypeName                                          // see A.7

QuantifiedVars    = E. {[.]TypeVariable}+:

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

```
Prec          = Digit                                          // see A.8
Strict        = !

Type          = {[Strict] BrackType}+
              | ArrowType
BrackType     = [UnqTypeAttrib] SimpleType
ArrowType     = {BrackType}+ -> Type

UnqTypeAttrib = *
              | UniqueTypeVariable:                            // see A.7
              | .

SimpleType    = TypeConstructor
              | TypeVariable                                   // see A.7
              | BasicType
              | PredefAbstrType
              | ListType
              | TupleType
              | ArrayType
              | (Type)

TypeConstructor = TypeName
                | []
                | ({,}+)
                | {}
                | {!}
                | {#}
BasicType       = Int
                | Real
                | Char
                | Bool
PredefAbstrType = World
                | File
                | ProcId
ListType        = [Type]
TupleType       = ([Strict] Type,{[Strict] Type}-list)
ArrayType       = {[Strict] Type}
                | {#Type}
```

## A.6 Class Definition

```
ClassDef          = TypeClassDef
                  | TypeClassInstanceDef

TypeClassDef      = class ClassName [.] TypeVariable [ClassContext]
                    [where {| {ClassMemberDef}+ |}]
                  | class FunctionName [.] TypeVariable:: FunctionType #
                  | class (FunctionName) [FixPrec] [.] TypeVariable:: FunctionType #

ClassMemberDef    = FunctionTypeDef #
                    [MacroDef #]

TypeClassInstanceDef = instance ClassName [Type [default | ClassContext]]
                       [where {| {DefOfFunction}+ |}]

TypeClassInstanceExportDef
                  = export ClassName {BasicType | TypeVariable}-list #
```

## A.7 Names

```
ModuleName      = LowerCaseId | UpperCaseId | FunnyId
FunctionName    = LowerCaseId | UpperCaseId | FunnyId
ConstructorName =             UpperCaseId | FunnyId
SelectorVariable= LowerCaseId
Variable        = LowerCaseId
MacroName       = LowerCaseId | UpperCaseId | FunnyId
FieldName       = LowerCaseId
TypeName        =             UpperCaseId | FunnyId
TypeVariable    = LowerCaseId
```

```
UniqueTypeVariable = LowerCaseId
ClassName          = 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                                  // see A.8
                   | _ | `
```

## A.8 Denotations

```
IntDenotation      = [Sign]~{Digit}+                        // decimal
                   | [Sign]~ 0~{OctDigit}+                  // octal
                   | [Sign]~ 0x~{HexDigit}+                 // hexadecimal
Sign               = + | -
RealDenotation     = [Sign~]{Digit~}+.{~Digit}+[~E[~Sign]{~Digit}+]
BoolDenotation     = True | False
CharDenotation     = CharDel~AnyChar/~CharDel.CharDel
CharsDenotation    = CharDel~{AnyChar/~CharDel}+.CharDel
StringDenotation   = StringDel~{AnyChar/~StringDel}~StringDel

AnyChar            = IdChar | ReservedChar | Special
ReservedChar       = ( | ) |{ | } | [ | ] | ; | , | .
Special            = \n | \r | \f | \b                    // newline,return,formf,backspace
                   | \t | \\ | \CharDel                    // tab, backslash, character delimiter
                   | \StringDel                            // string delimiter
                   | \{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
Digit              = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

CharDel            = '
StringDel          = "
```

# B

# Lexical Structure

**B.1 Lexical Program Structure**
**B.2 Comments**
**B.3 Reserved Keywords and Symbols**

In this appendix the lexical structure of CLEAN is given. It describes the kind of tokens recognised by the scanner/parser. In particular it summarizes the keywords, symbols and characters which have a special meaning in the language.

## B.1 Lexical Program Structure

In this Section the lexical structure of CLEAN is given. It describes the kind of tokens recognised by the scanner/parser. In particular it summarizes the keywords, symbols and characters which have a special meaning in the language.

```
LexProgram  = { Lexeme | {Whitespace}+ }
Lexeme      = ReservedKeywordOrSymbol      // see Section B.3
            | ReservedChar                 // see Section A.8
            | Literal
            | Identifier
Identifier  = LowerCaseId                  // see A.7
            | UpperCaseId                  // see A.7
            | FunnyId                      // see A.7
Literal     = IntDenotation                // see A.8
            | RealDenotation               // see A.8
            | BoolDenotation               // see A.8
            | CharDenotation               // see A.8
            | CharsDenotation              // see A.8
            | StringDenotation             // see A.8
```

```
Whitespace  = space                        // a space character
            | tab                          // a horizontal tab
            | newline                      // a newline char
            | formfeed                     // a formfeed
            | verttab                      // a vertical tab
            | Comment                      // see Section B.2
```

## B.2 Comments

```
Comment         = // AnythingTillNL newline
                | /* AnythingTill/* Comment AnythingTill*/ */
                | /* AnythingTill*/ */
AnythingTillNL  = {AnyChar/~newline}       // no newline
AnythingTill/*  = {AnyChar/~/*}            // no "/*"
AnythingTill*/  = {AnyChar/~*/}            // no "*/"
AnyChar         = IdChar | ReservedChar | Special   // see A.7
```

## B.3 Reserved Keywords and Symbols

Below the keywords and 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 can be freely used if they are not a reserved character (see A.8). In the comment it is indicated for which context (name space) the symbol is predefined.

**ReservedKeywordOrSymbol** =

// **in *all* contexts:**

```
    /*                  //  begin of comment block
|   */                  //  end of comment block
|   //                  //  rest of line is comment
|   ::                  //  begin of a type definition
|   :==                 //  in a type synonym or macro definition
|   =                   //  in a function, graph, alg. type, rec. field
|   =:                  //  labeling a graph definition
|   =>                  //  in a function definition
|   ;                   //  end of a definition (if no lay-out rule)
|   from                //  begin of symbol list for imports
|   definition          //  begin of definition module
|   implementation      //  begin of implementation module
|   import              //  begin of import list
|   module              //  in module header
|   system              //  begin of system module
|   ->                  //  in a case expression, lambda abstraction
|   [                   //  begin of a list
|   :                   //  cons node
|   ]                   //  end of a list
|   \\                  //  begin of list or array comprehension
|   <-                  //  list gen. in list or array comprehension
|   <-:                 //  array gen. in list or array comprehension
|   {                   //  begin of a record or array, begin of a scope
|   }                   //  end of a record or array, end of a scope
|   .                   //  a record or array selector
|   !                   //  a record or array selector (for unique objects)
|   &                   //  an update of a record or array, zipping gener.
|   {|                  //  begin of process annotations
|   |}                  //  end of process annotations
|   case                //  begin of case expression
|   code                //  begin code block in a syst impl. module
|   if                  //  begin of a conditional expression
|   in                  //  end of (strict) let expression
|   let                 //  begin of let expression
|   #                   //  begin of let expression (for a guard)
|   let!                //  begin of strict let expression
|   #!                  //  begin of strict let expression (for a guard)
|   of                  //  in case expression
|   where               //  begin of local def of a function alternative
|   with                //  begin of local def in a rule alternative
|   infix               //  infix indication in operator definition
|   infixl              //  infix left indication in operator definition
|   infixr              //  infix right indication in operator definition
```

// **in *process* annotations:**

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

// **in *type* specifications:**

```
|   !                   //  strict type
|   .                   //  uniqueness type variable
|   #                   //  unboxed type
|   *                   //  unique type
```

# C

# Bibliography

Achten, P.M. (1996). Interactive Functional Programs - models, methods, and implementations. Ph.D., University of Nijmegen.

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.

Peter Achten and Rinus Plasmeijer (1995). 'The Ins and Outs of CONCURRENT CLEAN I/O'. *Journal of Functional Programming,* 5, 1, pp. 81-110.

Peter Achten and Rinus Plasmeijer (1997). "Interactive Functional Objects in Clean". In: *Proc. of the 1997 Workshop on the Implementation of Functional Languages (IFL'97)*, ed. K. Hammond Davie, T., and Clack, C., St.Andrews, Scotland, pp. 387-406.

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.

Barendregt, H.P. (1984). The Lambda-Calculus, its Syntax and Semantics. North–Holland.

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.

Erik Barendsen and Sjaak Smetsers (1993a). '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.

Erik Barendsen and Sjaak Smetsers (1993b). '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.

Bird, R.S. and P. Wadler (1988). *Introduction to Functional Programming*. Prentice Hall.

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.

Eekelen, M.C.J.D. van, J.W.M. Smetsers, M.J. Plasmeijer (1997). "Graph Rewriting Semantics for Functional Programming Languages". In: *Proc. of the CSL '96, Fifth Annual conference of the European Association for Computer Science Logic (EACSL)*, ed. Marc Bezem Dirk van Dalen, Utrecht, Springer-Verlag, LNCS, **1258**, pp. 106-128.

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.

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.

Kesseler, M.H.G. (1996). The Implementation of Functional Languages on Parallel Machines with Distributed Memory. Ph.D., University of Nijmegen.

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.

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.

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.

Eric Nöcker and Sjaak Smetsers (1993). 'Partially strict non-recursive data types'. *Journal of Functional Programming,* **3**, 2, pp. 191-215.

Rinus Plasmeijer and Marko van Eekelen (1993). *Functional Programming and Parallel Graph Rewriting*. Addison Wesley, ISBN 0-201-41663-8.

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.

Ronan Sleep, Rinus Plasmeijer and Marko van Eekelen (1993). *Term Graph Rewriting - Theory and Practice*. John Wiley & Sons.

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.

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.

# D

# Index

Emboldened terms indicate where a term has been defined in the text. A term starting with an uppercase character generally refers to an identifier in the syntactic description or to a predefined function or operator in the library.


## A

abort, **49**
*abstract data type*, 13, **46**
    predefined, **40**
AbstractTypeDef, **46**, 86
*actual node-id*, **3**
*algebraic data type*, **42**
algebraic data type definition, 19
AlgebraicTypeDef, 42, 59, 74, 86
*anonymous node variable*, 19
AnyChar, **24**, 88, **89**
Application, **22**, **23**, 85
argument
    formal, **18**
arity of a function, 47, **48**
*array*, 21, 24, **28**, **41**, 86
    *comprehension*, **30**
    *generator*, **25**
    index, 29
    *index*, **41**
    pattern, **21**
    *selection*, **31**
ArrayDenotation, 28, 86
ArrayExpr, **25**, **29**, 85, 86
ArrayIndex, 86
ArrayPattern, **21**, 84
ArrayType, **41**, **76**, 87
ArrayUpdate, 29, 86
*arrow type*, **41**
ArrowType, **41**, **57**, **87**
ASCII, 40
**at**, 90

## B

*basic type*, **20**, **40**
BasicType, **40**, 87
BasicValue, **20**, **24**, 84
BasicValuePattern, **20**, 84
block structure, 32
Bool, 20, 23, 40, 87
*BoolDenotation*, **24**, 88
BooleanExpr, **25**, 85
boxing, **72**
BrackGraph, 22, 27, 31, 85
BrackPattern, **18**, 84
BrackType, 39, 43, 57, 60, 87

## C

*CAF*, 77
cartesian product, 41, **47**
**case**, 90
**case**, 32, 86
*case expression*, **32**
CaseAltDef, 32, 86
CaseExpr, 32, 86
Char, 23, 40, 87
Char, 20
CharDel, **24**, 88
*CharDenotation*, **24**, 88
*CharsDenotation*, **24**, 88
class, 50, 54, 87
    enumeration type, 25
ClassContext, 51, 84
ClassDef, **41**, 87
ClassMemberDef, 50, 87
*ClassName*, 7, 88
Clean License
    commercial, iv
    educational, iv
**CleanProgram**, 9, 83
*code*, **15**, 90
**Comment**, **89**
*concurrency*, ii
*conditional expression*, **32**
*console mode*, **10**
constant
    global, 11
    local, 11
*Constant Applicative Form*, 77
*constant function*, **18**
*constant value*, **20**, **42**
Constructor, **19**, **23**, 84
    of zero arity, 19
*constructor operator*, **19**
*constructor pattern*, **19**
*constructor symbol*, **1**
ConstructorDef, 42, 60, 74, 86

*ConstructorName*, 7, 87
ConstructorOrFunction, **23**, 85
context
    lazy, **72**
    strict, **72**
*contractum*, **1**
corresponding module, **9**, 12
*curried application*, **48**
curried constructor application, 19
curried type, 41
currying, 23
*cyclic graph*, **35**

## D

*data constructor*, 18, 19, **42**
*data structure*, 42
DataRoot, **4**
decimal number, 24
**default**, 50, 87
DefDefinition, **12**, 83
**definition**, 9, 83, 90
    global, 11
    local, 11
*definition module*, **9**, **12**, 46
DefinitionModule, 9, 83
DefOfFunction, **17**, 84
DefOfMacro, 78, 86
depending module, **13**
*dictionary*, **49**
Digit, 8, 88
directed arc, **1**
DotDotexpression, **25**, 85

## E

**E.**, **44**
enumeration type, 25
evaluation
    interleaved, 22
    parallel, 22
*existentially quantified variable*, **44**
explicit import, **13**
ExplicitImportDef, **13**, 84
**export**, 55, 87
expression, **23**
    initial, 1

## F

**False**, **24**, 88
field name, 20, 26, 27, **45**
*FieldName*, 7, 87
File, 40, 87
Fix, 42, 47, 86
Fix, 47
fixity, 18, 23, **48**
*flat type*, **55**
*foreign function*, **14**
formal argument, **18**, 19, 23
*formal node-id*, **3**
**from**, **13**, 84, 90
*function*, **1**, **23**, 84
    *alternative*, **17**
    arity of a, **48**
    constant, 18, 35
    curried application of a, **48**
    *definition*, **17**
    global, 11
    local, 11
    partial, 1, 18, **48**
    total, **49**
Function, **17**
*function definition*, **1**
*function object*, **41**
*function symbol*, **1**
function type, **47**
*functional array update*, **29**
*functional record update*, **27**
*functional reduction strategy*, **2**
FunctionAltDef, **17**, 84
FunctionBody, **17**, 32, 85, 86
FunctionDef, **17**, **41**, 84
FunctionDef, 47
*FunctionName*, 7, 87
FunctionType, 47, 51, 74, 84
**FunctionTypeDef**, 47, 84
FunnyId, **8**, 88

## G

garbage collection, 77
garbage collector, 35
generator, **25**, 85
    array, **24**
    list, **26**
Generators, **25**, 85
global definition, 11, 12
*global graph*, **4**
*global graph definition*, 77
global scope, 14
Gofer, i
*graph*, **1**
Graph, 4
graph definition, **35**
*graph rewrite rule*, 1, 17
GraphDef, **35**, 36, 77, 85
GraphExpr, 22, 85
GraphPattern, **18**, 84
GraphVariable, **23**, 85
Guard, **21**, **25**, 26, 85
    nested, 22
*guard*, 17
guarded function body, **17**

## H

head normal form, 10
hexadecimal number, 24
HexDigit, **24**, 88
Hilt B.V., vii
Hindley, 39
Hugs, i

## I

**I**, 90
I/O library, ii, 10
IdChar, **8**, 88
identifier, 8, **89**
identifiers
    renaming, 14

**if**, 90
**if**, 32, 86
ImplDefinition, **10**, 83
**implementation**, 9, 83
**implementation**, 90
ImplementationModule, 9, 83
implicit import.;ib.import
    implicit, **14**
ImplicitImportDef, **14**, 84
import, **14**, 84, 90
    explicit, **13**
*import statement*, 13
ImportDef, **13**, 84
Imports, **13**, 84
**in**, 36, 85, 90
**in**, **33**, 86
**infix**, **42**, **47**, 86, 90
infix constructor, **19**
infix position, 18, 19
**infixl**, 42, 47, 86, 90
**infixr**, **42**, **47**, 86, 90
Initial, 3, 4
initial expression, 9, **10**
**instance**, 50, 87
Int, 20, 23, 40, 87
*IntDenotation*, **24**, 88
IntegerExpr, 86
Intel, iv

## K

keyword, 90

## L

LambdaAbstr, 32, 86
Läufer, 44
lay-out rule, **11**, 12
*lazy context*, **72**
*lazy evaluation*, **71**
*lazy semantics*, **75**
left hand-side of a graph, **1**
**let**, **33**, 86
**let // begin of let expression**, 90
**let!**, 36, 85, 90
LetBeforeExpression, 37, 84, 85
LetExpresssion, **33**, 86
Lexeme, **89**
**LexProgram**, **89**
LGraphExpr, **24**, 85
LGraphPattern, **20**, 84
**Linux**, iv
*list*, 20, **24**, **40**, 85
**list list generator**, **25**
list pattern, **20**
ListDenotation, **24**, 85
ListExpr, **25**, 85
ListPattern, **20**, 84
ListType, **40**, 76, 87
Literal, **89**
local definition, 11
LocalDef, **33**, **34**, 86
LocalFunctionAltDefs, **33**, 86
LocalFunctionDefs, **34**, 86
*loosely coupled parallel architecture*, **79**
LowerCaseChar, **8**, 88
LowerCaseId, **7**, 88

## M

**MacOS**, iv
macro
    global, 11
    local, 11
*macro definition*, **77**
**MacroDef**, 78, 86
MacroFixityDef, 78, 86
*MacroName*, **7**, 87
main module, 9
message passing, ii
Milner, 39
Miranda, i
mode
    console, **10**
    lay-out, **11**
    world, **10**
module, 9, 83, 90
    corresponding, **9**, 12
    definition, **9**, **12**
    depending, **13**
    implementation, **9**
    pass-through, **14**
*ModuleName*, **7**, 87
Motorola, iv

## N

name space, 90
*name spaces*, **8**
nested guards, 22
*nested scope*, **8**
*node*, **1**
*node variable*, **19**
    anonymous, 19
*node-id*, **1**
    actual, **3**
    applied, 1
    formal, **3**
*node-id variable*, **18**
*node-identifier*, **1**
*normal form*, **2**

## O

object oriented programming, 45
octal number, 24
OctDigit, **24**, 88
**of**, 32, 86, 90
*operator*, 18, **23**, **48**, 85
    constructor, 19
**OS/2**, iv
*otherwise*, **21**
*overloaded*, **49**
overloading, 8

## P

**P**, 90
partial function, 1, 18, 21, **48**
partial match, **2**
pass-through module, **14**

*pattern*, **1**, **18**, 19, 84
array, **21**
bracket, **18**
constructor, **19**
list, **20**
of basic type, 20
record, **20**, **36**
tuple, **20**
pattern match, 21
*pattern variable*, **19**
PatternVariable, **19**, **84**
polymorphic algebraic data type, **42**
PowerPC, iv
Prec, **42**, 87
Prec, 47
precedence, 18, **48**
*precedence*, 23
PredefAbstrType, **40**, 87
process, ii, 79, 85
***process annotation***, 90
ProcId, 40, 87
ProcIdExpr, 79, 85
*program*, **1**
*program graph*, **1**
*projection function*, **36**

## Q

Qualifier, **25**, 85
QuantifiedVariables, 42, 60, 86

## R

Real, 20, 23, 40, 87
*RealDenotation*, **24**, 88
recompilation, 12
*record*, 20, 26, 85
record pattern, **20**, **36**
*record selection*, **27**
*record type*, **45**
RecordDenotation, **27**, 85
RecordExpr, **27**, 86
RecordPattern, **20**, 84
RecordTypeDef, 45, 74, 86
RecordUpdate, 27, 86
*redex*, **2**
*redirection*, **1**
*redirection of a node*, **2**
*reducer*, **2**
*reducible expression*, **2**
*reduct*, **2**
*reduction strategy*, **2**
*reference*, **1**
ReservedChar, **24**, 88
**ReservedKeywordOrSymbol =**, **90**
*rewrite of a redex*, **2**
rewrite rules
comparing, 1
right hand-side of a graph, **1**
*root normal form*, **2**, 10
root stable form, 10
RootExpression, 22, 85
*rule*
*alternative*, **17**
rule alternative, 21

## S

*scope*, **8**, 11, 32, 77
nested, **8**
surrounding, **8**
Selection, 27, 29, 31, 86
by field name, **45**
by index, **28**
by position, **45**
Selection of a Record Field, **27**
Selection of an Array Element, **31**
Selector, **25**, **36**, 85
*selector variable*, **23**, **36**
*SelectorVariable*, 7, 87
semantics
lazy, **75**
strict, **75**
*sharing*, **35**
*sharing analysis*, **61**
Sign, **24**, 88
SimpleType, **39**, 43, 87
SML, i
**Solaris**, iv
Special, **24**, 88
SpecialChar, **8**, 88
**Start**, **3**, 4, **9**
*start rule*, **3**
StartNode, **4**
Strict, **73**, 87
*strict context*, **72**
*strict let expression*, **36**
*strict semantics*, **75**
StrictLet, 36, 85
StringDel, **24**, 88
*StringDenotation*, **24**, 88
*strong root normal form*, **2**
strong type system, 39
*strongly typed language*, **39**
sub-graph, **2**
*sub-pattern*, **19**
**SunOS**, iv
surrounding scope, **8**
*symbol*, **1**, 90
arguments of a, **1**
synonym type, **46**
SynonymTypeDef, 46, 86
**system**, 9, 83, 90
*system definition module*, **14**
*system implementation module*, **14**

## T

*Term Graph Rewriting*, 1
*total function*, **49**
tree, 1
**True**, **24**, 88
*tuple*, 20, 26, **40**, 85
tuple pattern, **20**
TuplePattern, **20**, 84
TupleType, 75, 87
TupleType, **41**
type, 39, 43, 56, 74, 87
abstract data, **46**
algebraic data, **42**

array, **41**
arrow, **41**
basic, 40
constructor
lazy array, **43**
list, **43**
strict array, **43**
unboxed array, **43**
context, **50**
curried, 41
existential, **44**
explicitly specified, **39**
flat, **55**
global, 11
inferred, **39**, 47
list, 25, **40**
of a function, 47
of partial function, **48**
record, **45**
synonym, **46**
tuple, **40**
variable, **42**
type class, **49**
definition of, **49**
member of, **49**
*type instance*, **39**
***type specification***, 90
*type variable*, **42**
TypeClassDef, 50, 54, 87
TypeClassInstanceDef, 50, 87
**TypeClassInstanceExportDef**, 55, 87
TypeConstructor, **43**, 86, 87
TypeConstructor, **42**, 60
**TypeDef**, 86
TypeDef, **41**
TypeLhs, **42**, 60, 86
*TypeName*, 7, 87
*TypeVariable*, 7, 87


## U


unboxing, **72**
uniqueness type attribute, 39, **56**
*UniqueTypeVariable*, 7
UnqTypeAttrib, **57**, 60, 87
UnqTypeUnEqualities, 84
update of a record
destructive, **27**
update of an array
destructive, **30**
UpperCaseChar, **8**, 88
UpperCaseId, 7, 88


## V


*Variable*, 7, **23**, 87
existentially quantified, **44**
node-id, **18**
pattern, **19**
selector, **23**, **36**
type, **42**


## W


**where**, **33**, 50, 86, 87, 90
*where block*, **33**
Whitespace, **89**
*wildcard*, **19**, **36**
**Windows**, iv
**with**, **34**, 86, 90
*with block*, **34**
World, 9, 40, 87
abstract, **10**
concrete physical, **10**
*world mode*, **10**


## X


Xview, iv


## Z


zero arity symbol, 23
ZF-expression, **25**, 85

\, 32, 86