ECOLE NORMALE SUPERIEURE

Introduction to the CLAIRE

programming language

François LABURTHE
Yves CASEAU

Département de Mathématiques et Informatique

Introduction to the CLAIRE
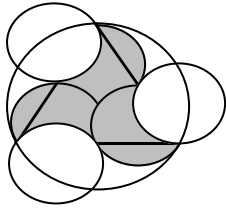# programming language

François LABURTHE
Yves CASEAU

Laboratoire d'Informatique de l'Ecole Normale Supérieure
45 rue d'Ulm 75230 PARIS Cedex 05

Tel : (33)(1) 44 32 00 00
Adresse électronique : ... @dmi.ens.fr

CLAIRE

Introduction to the CLAIRE Programming Language

Yves Caseau
François Laburthe
with the help of H. Chibois, A. Demaille, S. Hadinger,
C. Le Pape, A. Linz , T. Kökeny and L. Segoufin

Departement Mathématiques et Informatique
Ecole Normale Supérieure - Paris - FRANCE

Table of Contents

0. INTRODUCTION

CLAIRE is a high-level functional and object-oriented language with advanced rule processing capabilities. It is intended to allow the programmer to express complex algorithms with fewer lines and in an elegant and readable manner.

To provide a high degree of expressivity, CLAIRE uses

- A rich type system including type intervals and second-order types (with static/dynamic typing),
- Parametric classes and methods,
- An object-oriented logic with set extensions,
- Dynamic versioning that supports easy exploration of search spaces.

To achieve its goal of readability, CLAIRE uses

- set-based programming with an intuitive syntax,
- simple-minded object-oriented programming,
- truly polymorphic and parametric functional programming,
- a powerful-yet-readable extension of DATALOG to express logical conditions,
- an entity-relation approach with explicit relations, inverses and unknown values.

CLAIRE was designed for advanced applications that involve complex data modeling, rule processing and problem solving. CLAIRE was meant to be used in a C++ environment, either as a satellite (linking CLAIRE programs to C++ programs is straightforward) or as an upper layer (importing C++ programs is also easy). The key set of features that distinguishes CLAIRE from other programming languages has been dictated by our experience in solving complex optimization problems using a complex hybrid prototype language called LAURE. In order to implement complex algorithms with programs that are still easy to read (and thus, to maintain) we have found ourselves to make heavy use of two key features:

- production rules: CLAIRE supports rules that binds a CLAIRE expression (the conclusion) to a logical condition. Whenever the condition's value (a boolean) changes for a given object or set of objects, the conclusion is evaluated. This allows to trigger the same conclusion from many different types of changes and makes programs simpler. In addition, rules can be added to any class of object without any code modifications. The object logic used for conditions has been carefully crafted so that it combines expressive power with inference compilation techniques that rival hand code writing. Thus, CLAIRE is a choice language for artificial intelligence applications such as expert systems.

- versioning: CLAIRE supports versioning of a user-selected view of the entire system. The view can be made as large (for expressiveness) or as small (for efficiency) as is necessary. Versions are created linearly and can be viewed as a stack of snapshots of the system. CLAIRE supports very efficient creation/rollback of versions, which constitutes the basis for powerful backtracking, a key feature for problem solving. Contrary to most logic programming languages, this type of backtracking covers any user-defined structure, not simply a set of logic variables.

CLAIRE also provides automatic memory allocation/de-allocation which would have prevented an easy implementation as a C++ library. A less compelling feature for a new language is that set-oriented programming is much easier with a set-oriented language like CLAIRE than with C++ libraries.

CLAIRE is a high-level language, that can be used as a complete development language, since it is a general purpose language, but also as a pre-processor to C++, since a CLAIRE program can be naturally translated into a C++ program. CLAIRE is a set-oriented language in the sense that sets are first-class objects, typing is based on sets and control structures for manipulating sets are parts of the language kernel. Similarly, CLAIRE makes manipulating lists easy since lists are also first-class objects. CLAIRE can also be seen as a functional programming language, with full support for lambda abstraction, where functions can be passed as parameters and returned as values, and with powerful parametric polymorphism. Last, CLAIRE is meant to be reflective (i.e., implemented as a reflective

system, where each program is represented as an object). This aspect is out of scope of this book which is only concerned with the language itself, but makes CLAIRE very extensible, in a way similar to LISP.

CLAIRE is an object-oriented language with single inheritance. As in SMALLTALK, everything that exists in CLAIRE is an object. Each object belongs to a unique class and has a unique identity. Classes are the corner stones of the language, from which methods (procedures), slots and arrays (relations) are defined. Classes belong themselves to an single inheritance hierarchy. However, classes may be grouped using set union operators, and these unions may be used in most places where a class would be used, which offers an alternative to multiple inheritance. In a way similar to Modula-3, CLAIRE is a modular language that provides recursively embedded modules with associated namespaces. This is a clear departure from C++, since the module decomposition can either be parallel to the class organization (mimicking C++ encapsulation) or orthogonal (e.g., encapsulating one service among multiple classes).

CLAIRE is a typed language, with full inclusion polymorphism. This implies that one can use CLAIRE with a variety of type disciplines ranging from weak typing in a manner that is close to SMALLTALK up to a more rigid manner close to C++. This flexibility is useful to capture programming styles ranging from prototyping to production code development. The more typing information available, the more CLAIRE's compiler will behave like a statically-typed language compiler. This is achieved with a rich type system, based on sets, that goes beyond types in C++. This type system provides functional types (second-order types) similar to ML, parametric types associated to parametric classes and many useful type constructors such as unions or intervals. Therefore, the same type system supports the naive user who simply wishes to use classes as types and the utility library developer who needs a powerful interface description language.

As the reader will notice, CLAIRE draws its inspiration from a large number of existing languages. A non-exhaustive list would include SMALLTALK for the object-oriented aspects, SETL for the set programming aspects, OPS5 for the production rules, LISP for the reflection and the functional programming aspects, ML for the polymorphism and C for the general programming philosophy. As far as its ancestors are concerned, CLAIRE is very much influenced by LORE, a language developed in the mid 80s for knowledge representation. It was also influenced by LAURE but is much smaller and does not retain the original features of LAURE such as constraints or deductive rules. CLAIRE is also closer to C in its spirit and its syntax than LAURE was. Another major difference is that CLAIRE was designed to be simple and easy to teach, which this user's manual will try to demonstrate.

This document is organized as follows. The first chapter is a short tutorial on the main aspects of CLAIRE. A few selected examples are used to gradually introduce the concepts of the language without worrying about completeness. These are running programs that can be used to practice with the interpreter and the compiler. Our hope is that a reader familiar with other object-oriented languages should be able to start programming with CLAIRE without further reading. Chapter 2 gives a description of objects, classes and basic expressions in CLAIRE. It explains how to define a class (including a parameterized class) and how to read a slot value, call a method or do an assignment.

Chapter 3 deals with the control structures of the language. These include block and conditional structures, loops and object instantiation. It also describes the set-oriented aspects of CLAIRE and set iteration. Chapter 4 covers methods and types. It explains how to define a method, how to define and use a type. Types being set expressions and first-class objects can be used in many useful ways. This chapter also covers more advanced polymorphism in CLAIRE.

Chapter 5 is the most important chapter of the book since it covers the aspects that are the most original in CLAIRE, namely rules and versions. It introduces the notion of generalized array (binary relation) and the logic sub-language for writing conditions. It then describes rules and versioning, with its use for hypothetical reasoning and search for problem solving. Chapter 6 covers the remaining topics, namely input/output, modules and global variables.

In addition, three appendices are included. The first appendix focuses on the external syntax of the CLAIRE language (includes lexical conventions and a formal grammar). The second appendix is the description of the application programming interface. It is a description of the methods that are part of the standard CLAIRE system library. The last appendix is a very short description of the standard CLAIRE system (compiler & interpreter) that has been made available through ftp.

1. TUTORIAL

1.1 Loading a Program

This first chapter is a short tutorial that introduces the major concepts gradually. It contains enough information for a reader familiar with other object-oriented language to start practicing with CLAIRE. Each aspect of the language will be detailed in a further chapter. All the examples that are shown here should be available as part of the standard CLAIRE system so that you should not need to type the longer examples.

The first step that must be mastered to practice with CLAIRE is to learn how to invoke the compiler or the interpreter. Once you have found this out, you are ready to try our first program. This program simply prints the release number of the CLAIRE system that you are using.

```
[main() -> printf("claire release ~S\n", release() ]
```

You must first save this line on a file, using your favorite text editor (e.g. emacs). Let us now assume that this one-line program is in a file *release.cl*. Using a file that ends with .cl is not mandatory to load a CLAIRE file but it is necessary for the compiler to work properly so it is a good habit to take.

When you invoke the CLAIRE executable, you enter a loop called a top-level[1]. This loop prompts for a command with the prompt "CLAIRE>" and returns the result of the evaluation with a prompt "[..]". The number inside the brackets can be used to retrieve previous results (this is explained in the last appendix). Here we assume that you are familiar with the principle of a top-level loop; otherwise, you may start by reading the description of the CLAIRE top-level in the Appendix C. To run our program, we enter two commands at the top-level. The first one `load("release")` loads the file that we have written and returns true to say that everything went fine. The second command `main()` invokes the method (in CLAIRE a procedure is called a method) that is defined in this file.

```
% claire
claire> load("release")
[1] true
claire> main()
[2] claire release 1.0.
claire> q
%
```

Each CLAIRE programs is organized into blocks, that are surrounded by parentheses, and definitions that are surrounded by brackets for class and method definition. Our program has only one definition of the method *main*. The declaration main() tells that this method has no parameters, the expression after the arrow -> is the definition of the method. Here it is just a printf statement, that prints its first argument (a format string) after inserting the other arguments at places indicated by the control character ~ (followed by an option character which can be A,S,I). This is similar to a C printf, except that the place where the argument *release()* must be inserted in the control string is denoted with ~S. There is no need to tell the type of the argument for printf, CLAIRE knows it already. We also learn from this example that there exist a pre-defined method release() that returns some version identification and that you exit the top-level by typing q (^D also works).

---

[1] In the following we assume that CLAIRE is invoked in a workstation environment using a command shell. You must first find out how to invoke the CLAIRE system in your own environment. This should be explained in you installation documents.

In this example, release() is a system-defined method[2]. The list of such methods is given in the second appendix. When we load the previous program, it is interpreted (each instruction is a CLAIRE object that is evaluated). It can also be compiled (through the intermediate step of C code generation). Here is how we would compile the previous program and generate the executable "test".

```
% claire -cf release -o test
% test
claire> main()
claire release 1.0.
test2> q
```

We can also notice that the main() method is not executed automatically as it would with a C compiler, but that we enter a standard CLAIRE top level. This behavior, as well as the use of parameters in the command line (i.e., the -cf option), is OS (operating system) dependent. The compiler generates an equivalent C file that is linked with a default *main.c* file which is provided by the host system. This file can easily be customized or replaced to achieve different behaviors (cf. Appendix C).

Let us now write a second program that prints the first 10 Fibonacci numbers. We will now assume that you know how to load and execute a program, so we will only give the program file. The following example defines the *fib(n)* function, where fib(n) is the nth Fibonacci number.

```
[fib(n:integer) : integer
    -> if (n < 2) 1 else fib(n - 1) + fib(n - 2)]

MAX :: 10               // global constant (cf. 6.4)

[main() -> for i in (1 .. MAX) printf("fib(~S) = ~S\n",i,fib(i)) ]
```

From this simple example, we can notice many interesting rules for writing method in CLAIRE. First, the range of a method is introduced by the "typing" character "**:**". The range is almost mandatory, since the default range is *void*, which will imply that the method does *not* return a value (e.g., main). Conditional in CLAIRE use a traditional **if** construct (Section 3.3), but the iteration construct "for" is a set iteration. The expression **for x in S e(x)** evaluates the expression **e(x)** for all values **x** in the set **S**. There are many kinds of set operators in CLAIRE (Section 3.1); for instance **n .. m** is the interval of integers between **n** and **m**.

Obviously, this program is very naive and is not the right way to print a long sequence of Fibonacci numbers, since the complexity of *fib(n)* is exponential. We can compute the sequence using two local variables to store the previous values of *fib(n - 1)* and *fib(n - 2)*. The next example illustrates such an idea and the use of **let**, which is used to introduce a list of local variables. Notice that they are local variables, whose scope is only the instruction after the keyword **in**. Also notice that a variable assignment uses the symbol **:=**, as in PASCAL, and the symbol **=** is left for equality.

```
[main()
 -> let n := 2, f_n-1 := 1, f_n-2 := 1 in
    ( printf("fib(0) = 1 \nfib(1) = 1\n"),
      while (n < 10)
         let f_n := f_n-1 + f_n-2 in
            ( printf("fib(~S) = ~S \n",n,f_n),
```

---

[2] The release is a float number X.Y, where X is the version number and Y the release number. The version number in this book should be the same as the one obtained with your system. Changes among different release numbers should not affect the correctness of this documentation (Note: X.99 ≈ X+1.00)

```
          n :+ 1,            // short cut for n := n + 1
        f_n-2 := f_n-1, f_n-1 := f_n)  ) ]
```

Note that we used f_n-1 and f_n-2 as variable names. Almost all characters are allowed within identifiers (all characters but separators, '/', '#' and @). Hence, x+2 can be the name of an object whereas the expression denoting an addition is x + 2. Blank spaces are always mandatory to separate identifiers.

Warning: *CLAIRE's syntax is intended to be fairly natural for C programmers, with expressions that exist both in CLAIRE and C having the same meaning. There are two exceptions to this rule: the choice of := for assignment and = for equality, and the absence of special status for characters +, *, -, etc. Minor differences include the use of & and | for boolean operations and % for membership.*

A more elegant way is to use an array *fib[n]*, as in the following version of our program.

```
[fib[n:integer] : integer -> 1]


[main()
 -> for i in (2 .. MAX) fib[i] := fib[i - 1] + fib[i - 2],
    for i in (0 .. MAX) printf("fib(~S) = ~S\n",i,fib[i]) ]
```

An interesting feature of CLAIRE is that the domain of an array is not necessarily an interval of integers. It can actually be any type, which means that arrays can be seen as "extended dictionaries" (Section 5.1). On the other hand, when the domain is a finite set, CLAIRE allows to define an "initial value" using the **->** keyword, as for a method. For instance, the ultimate version of our program could be written as follows (using the fact that intervals are enumerated from small to large).

```
[fib[n:(0 .. 10)] : integer -> if (n < 2) 1 else fib[n - 1] + fib[n - 2]]

[main() ->    for i in (0 .. 10) printf("fib(~S) = ~S\n",i,fib[i])]
```

Let us now write a file copy program. We use two system functions *getc*(p) and *putc*(p) that respectively read and write a character c on an input/output port p. A port is an object usually associated with a file from the operating system. A port is open with the system function *fopen(s1,s2)* where s1 is the name of the file (a string) and s2 is another string that controls the way the port is used (cf. Section 6.1; for instance "w" is for writing and "r" is for reading).

```
[copy(f1:string,f2:string)
 -> let   p1 := fopen(f1,"r"),
          p2 := fopen(f2,"w"),
          c := ' ' in
      ( use_as_output(p2),
        while (c != EOF) (c := getc(p1), putc(c,p2)),
        fclose(f1), fclose(f2) )]
```

Let us now write a program that copies a program and automatically indents it. Printing with indentation is usually called pretty-printing, and is offered as a system method in CLAIRE: *pretty(x)* pretty-prints on the output port. All CLAIRE instructions are printed so that they can be read back. In the previous example, we have used two very basic read/write methods (at the character level) and thus we could have written a very similar program using C. Here we use a more powerful method called *read(p)* that reads one instruction on the port p (thus, it performs the lexical & syntactical analysis and generate the CLAIRE objects that represents instructions). Surprisingly, our new program is very similar to the previous one.

```
[copy&indent(f1:string,f2:string)
    -> let   p1 := fopen(f1,"r"),
             p2 := fopen(f2,"w"),
             c := unknown in
         ( use_as_output(p2),
           while (c != eof)
             pretty_print(c := read(p1)),
               fclose(p1), fclose(p2) )]
```

In the next example, we will illustrate how to use modules to obtain different namespaces. All identifiers in CLAIRE belong to a namespace, represented by a module. Modules are organized into a tree, the top of which is the default *CLAIRE* module. All the previous examples have used this default namespace implicitly. Our next program is a very simplified phone directory. The public interface for that program is a set of two methods store(name,phone) and dial(name). We want all other objects and methods to be in a different namespace, so we create a new module called phone_application. We also use comments that are defined in CLAIRE as anything that in on the same line after the character;.

```
;; definition of the module
phone_application :: module(part_of = claire)
open(phone_application)

;; value is an extended array that stores the phone #
[private/value[s:string] : string]

;; lower returns the lower case version of a strin
;; (i.e. lower("aBcD") = "abcd")
[lower(s:string) : string
-> let s2 := copy(s) in
    ( for i in (1 .. length(s))
        (if (s2[i] % 'A' .. 'Z')
            s2[i] = char!(integer!(s2[i]) - 32)))
    s2)]

[claire/store(name:string,phone:string)
  -> value[lower(name)] := phone]

[claire/dial(name:string) : string  ; returns the phone #
  -> value[lower(name)] ]

end(phone_application)
```

This example illustrates many important features of modules: they are classes just like any other objects: once the module x has been defined, we can use the associated namespace with the *open*(x) method, and later return to the initial namespace with *end*(x). The statement *part_of(x)=y* defines x as a new child of the module y. When open(x) has been executed, any new identifier that is read will belong to the new namespace associated with x. This has an important consequence on the visibility of the identifier, since an identifier *lower* defined in a module *phone_application* is only visible (i.e. can be used) in the module x itself or its descendants. Otherwise, the identifier must be qualified (phone_application/lower) to be used. There are two ways to escape this rule: first, an identifier can be associated to any module above the open module, if it is declared with the qualified form. Second, when an identifier is declared with the prefix private/, it becomes impossible to access the identifier using the qualified form. For instance, we used *private/value* to forbid the use of the array (in the CLAIRE extended sense) anywhere but in the descendants of the module *phone_application*.

Module organization is a key aspect of software development and should not be mixed with the code. The previous example is not the preferred way to use modules. It is better to put module definitions in a project file, and to load a file inside a module's namespace using the *load(m:module)* method. For instance, we could remove the first three lines and the last two in the previous example and put the result in the file *phone.cl*; then we write a init.cl project file as follows.

```
;; modules definitions
phone_application :: module(    part_of = claire,
                                made_of = list("phone"))
phone_database :: module(part_of = phone_application)
```

We could have given any name to the project file, but calling it init.cl will make CLAIRE load it automatically when it start. We can then call *load(phone_application)* to load the file in the

phone_application namespace. This is achieved through the slot *made_of* that contains the list of files that we want to associate with the module (cf. Section 6)

1.2 Objects and Classes

Our next example is a small pricing software for HiFi Audio components[3]. The goal of the program is to manage a small database of available material, to help build a system by choosing the right components (according to some constraints) and compute the price.

We start by defining our class hierarchy according to the following figure.
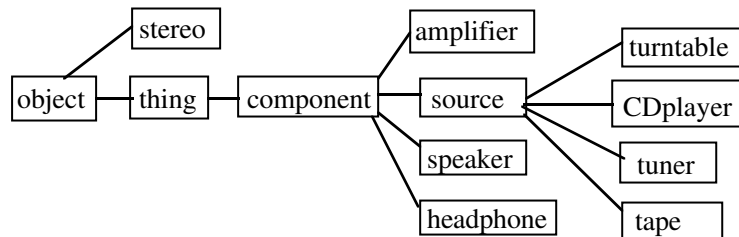

```
[component <: thing(price:integer, brand:string)]
[amplifier <: component(   power:integer, input:integer,
                           ohms:set[{4,8}])]
[speaker <: component(maxpower:integer, ohms:{4,8})]
[headphone <: component(maxpower:integer,   ohms:{4,8})]
[musical_source <: component(sensitivity:integer)]
[CDplayer <: musical_source(laser_beams:(1 .. 3))]
[turntable <: musical_source]
[tuner <: musical_source]
 B :: thing()     C :: thing()    nodolby :: thing()
[tape <: musical_source(dolby:{nodolby,B,C})]
[stereo <: object(    sources:set[musical_source],
                      amp:amplifier,
                      out:set[speaker U headphone],
                      warranty:boolean = false)]
```

Now that we have defined the taxonomy of all the objects in our HiFi world, we can describe the set of all models actually carried by our store. These are defined by means of instances of those classes.

```
amp1 :: amplifier(    power = 120, input = 4, ohms = {4,8},
                      price = 400, brand = "Okyonino")
amp2 :: amplifier(    power = 40, input = 2, ohms = {4},
                      price = 130, brand = "Cheapy")
tuner1 :: tuner(sensitivity = 10, price = 200, brand = "Okyonino")
tuner2 :: tuner(sensitivity = 30, price = 80, brand = "Cheapy")
CD1 :: CDplayer( sensitivity = 3, price = 300,
                 laser_beams = 3, brand = "Okyonino")
CD2 :: CDplayer( sensitivity = 7, price = 180,
                 laser_beams = 2, brand = "Okyonino")
CD3 :: CDplayer( sensitivity = 15, price = 110,
                 laser_beams = 1, brand = "Cheapy")
t1 :: tape( sensitivity = 40, price = 70,
            dolby = nodolby, brand = "Cheapy")
```

---

[3] All brands and product names are totally fictitious.

```
s1 :: speaker(   ohms = 8, maxpower = 150,
                 price = 1000, brand = "Magisound")
s2 :: speaker(   ohms = 8, maxpower = 80,
                 price = 400, brand = "Magisound")
s3 :: speaker(   ohms = 4, maxpower = 40,
                 price = 150, brand = "Cheapy")
ph :: speaker(ohms = 4, maxpower = 40, price = 50, brand = "Okyonino")
   etc ...
```

Now that we have defined some HiFi components with their technical features, we can manipulate them and define some methods. For example, we can compute the total price of a stereo as the sum of the prices of all its components. We first need an auxiliary method that computes the sum of a list of integers.

```
[sum(s:list[integer]) : integer
   -> let n := 0 in (for y in s n :+ y, n) ]

[price(s:stereo) : integer
     -> sum(list{price(x) | x in sources(s) U {amp(s)} U out(s)}) ]

InventoryTotal:integer :: 0           // global variable with range integer
```

Note here the use of set image (we consider the list of all *price(x)* for all x in the following set: the union of *sources(s), {amp(s)}* and *out(s)*). Also, we introduce a global variable *InventoryTotal*, of range integer and value 0. If we want to keep some "specials" which are sets of components for which the price is less than the sum of its components, we may use an (extended) array to store them:

```
[discount[s:set[component]] : integer -> 0]
discount[{amp1,s1}] := 1200
discount[{amp1,CD1}] := 600
```

To find the best price of a set of components, we now write a more sophisticated method that tries to identify the best subsets that are on sale. This is a good example of CLAIRE's programming style (if we assume that size(s) is small and that *discount* contains thousands of tuples).

```
[price(s:set[component]) : integer
-> let p := 100000 in
     (for s2 in set[s]                    ;; decompose s = s2 U ...
       let   x := size(s2),
             p2 := (   if (x > 1) discount[s2]
                       else if (x = 1) price(s2[1])
                       else 0)  in
          (if (p2 > 0) p :min (p2 + price(difference(s,s2))))),
       p) ]
```

Notice that we use some syntactical sugar here: `p :min x` is equivalent to `p := (p min x)`. This works with any other operation (such as +).

1.3 Rules

We now want to do some reasoning about stereo systems. We start by writing down the rules for matching components with one another. We want a signal to be raised whenever one of these rules is violated. Hence we create the following exception:

```
[technical_problem <: exception(s:string) ]
event(speaker,amp,sources,out)
```

The event declaration (cf. Section 5) is necessary to tell CLAIRE when the rule should be checked. A rule is defined by a condition (following the :: keyword) and a conclusion (following the => keyword). The condition is checked whenever an event occurs, that is whenever a relation `r` that was declared with `event(r)` is modified. Then, for any set of objects that now satisfy the condition, the

conclusion expression is evaluated. Here are some simple rules that will raise exceptions when some technical requirements are not met.

```
[compatibility1(sp:speaker, a:amplifier) ::
   exists(s:stereo, o = speaker(s) & a = amp(s) &
               not(ohms(o) % ohms(a)) )
   => technical_problem(s = "conflict speakers-amp") ]

[compatibility2(s:stereo) ::
   size(sources(s)) > inputs(amp(s))
   => technical_problem(s = "too many sources") ]

[compatibility3(s:stereo) ::
   exists(o:speaker, o % out(s) & maxpower(o) < power(amp(s)))
   => technical_problem(s = "amp to strong for the speakers") ]
```

We can now use our system (applying the rules on the small database) to look for consistent systems. For example, assume I want to buy speakers that fit my amp (for instance, amp1): we will try several possibilities to fill the slot *out* of my stereo and will watch whether they raise an exception or not. In order for the rule to be triggered, we need to tell which changes in the database are relevant to rule triggering. Here, modifications on the relation *out* should trigger the evaluation of the concerned rules. This was achieved by the command *event(out)* (which could be disabled by the command *noevent(out).*)

```
my_system :: stereo(amp = amp1)
( for sp in instances(speaker)
   try (out(my_system) :add sp, break(my_system))
   catch technical_problem out(my_system) :delete sp )
```

If we want to successively choose the speakers, the CD player, the tape, etc..., we cannot guarantee that if a choice does not immediately raise an exception, there will always exist a solution in the end. Thus, we need to make some hypothetical reasoning: we suppose one branch of the choice contains a solution, and we backtrack on failure. The conclusions that had been drawn during the hypothesis need to be undone. For this, we have the possibility to declare that some relations in the database are stored in a special way such that one can go back to a previous state. Such states of the database (versions) are called worlds. The methods *world+()* and *world-()* respectively create a new world and return to the previous one. The command *store(out)* means that the graph of the relation *out* will be stored in that special way adapted to the world mechanism. In this example, we create the list of all possible (bringing no conflict according to the rules) stereos with two different musical sources.

```
store(out)

[all_possible_stereos() : list[stereo]
 -> let solutions:list[stereo] := list() , syst:stereo := unknown in
    (for a in amplifier
        (amp(syst) := a,
         for sp in speakers try
            (world+(), out(syst) := {sp},
           for h in headphones try
            ( world+(), out(syst) :add h,
              for s1 in source try
                ( world+(), sources(syst) := {s1},
                  for s2 in {s in source | owner(s) != owner(s1)}
                  try (world+(),
                       sources(syst) :add s2,
                       solutions :add copy(syst) )
                  catch any world-() )
             catch any world-() )
          catch any world-() )
      catch any world-()), solutions)    ]
```

This method explores the tree of all possibilities for stereos and returns the list of all the valid ones.

Following are more examples of rules that address the problem of proposing a warranty or proposing additional equipment when the system is homogeneous. These rules will however not be triggered if they are declared after the objects are created or modified or if no triggering relation has been declared as an event.

```
[warranty1(s:stereo) :: price(s) > 10000 => warranty(s) := true]

[warranty2(s:stereo) ::
      brand(amp(s)) = "Okyonino" => warranty(s) := true]

[warranty3(s:stereo) ::
    exists(sp:speaker, sp % out(s) & brand(sp) = "Magisound")
 => warranty(s) := true]

[homogeneity(s:stereo, c:component) ::
    size({so in sources(s) | brand(so) != brand(amp(s))}) = 0
    & brand(c) = brand(amp(s))
    & not(type(c) % {type(x) | x in sources(s)})
 => propose(c,s)]
```

Here is a last example of a method that returns the list of all possible stereos, classified by increasing prices. The same thing could be done with other criteria of choice.

```
[price_order(s1:stereo, s2:stereo) : boolean
 -> price(s1) <= price(s2) ]



[cheapest() : list[stereo] ->
 let l := all_possible_stereos() in sort(l,price_order @ stereo) ]
```

2. OBJECTS, CLASSES AND SLOTS

2.1 Objects and Entities

A program in CLAIRE is a collection of entities communicating one with another (everything in CLAIRE is an entity). Some entities are already created when you start running CLAIRE and you will create new ones when writing programs. Entities are organized into three sorts: primitive entities, bags and objects. The set (class) of all entities is called `any` and the set (a class also) of all objects is called `object`.

Primitive entities are already created before you start writing anything: they consist of integers, floats, symbols, strings, streams and functions. The most common operations on them are already built in, but you can add yours.

Bags are made of lists, printed (a,b,c,d) and sets, printed {a,b,c,d}. Members of the lists and sets can be anything.

Objects can be seen as tuples (lists of a given length and structure), with named fields (called slots) and unique identifiers. Two objects are distinct even if they represent the same tuple. The tuple structure and the associated slot names is represented by a class. An object is uniquely an instance of a class which describes the tuple structure (ordered list of slots). CLAIRE comes with a collection of structures (classes) as well as with a collection of objects (instances).

Each entity in CLAIRE belongs to a special class called its *owner*, which is the smallest class to which the entity belongs. The *owner* relationship is the extension to `any` of the traditional *isa* relationship between objects and classes.

2.2 Classes

A class is a formal name for a set of objects: a class corresponds to the set of all its instances. Classes are organized into a tree, each class being the subclass of another one, called its superclass. This relation of being a subclass (inheritance) corresponds to set inclusion: each class denotes a subset of its superclass. So, in order to identify instances of a class as objects of its superclass, there has to be some correspondence between the structures of both classes: all slots of a class must be present in all its subclasses. Subclasses are said to *inherit* the structure (slots) of their superclass (while refining it with other slots). The root of the class tree is the class `any` since it is the set of all objects. Formally, a class is defined by its superclass and a list of additional slots. Two types of classes can be created: those whose instances will have a name and those whose instances will be unnamed. Named objects must inherit (not directly, but they must be descendants) of the class `thing`.

Each slot is given as <name>:<range>=<default>. The range is a type and the optional default value is an object which type is included in <range>. The range must be defined before it is used, thus recursive class definitions use a forward definition principle (e.g., person).

```
[person <: thing]                  // forward definition
[person <: thing(age:integer = 0, father:person)]
[woman <: person]                  // another forward definition
[man <: person(wife:woman)]
[woman <: person(husband:man)]
[child <: person(school:string)]
[complex <: object(re:float,im:float)]
```

A class inherits from all the slots of its superclasses, so they need not be recalled in the definition of the class. For instance, here, the class *child* contains the slots *age* and *father*, because it inherited them from *person*.

A default value is used to place in the object slot during the instantiation (creation of a new instance) if no explicit value is supplied. The default value must belong to the range and will trigger rules or inverses in the same way an explicit value would. The only exception is the "*unknown*" value

which represents the absence of value. *unknown* is used when no default value is given (the default default value). Note that the default value is a real entity that is shared by all instances and not an expression that would be evaluated for each instantiation.

A class is the set union of all the instances of its descendants (itself, its subclasses, the subclasses of its subclasses, etc...). A class can create new members of itself (see section 3.5), its instances.

In some cases, it may be useful to "freeze" the data representation at some point: for, this, two mechanisms are offered: a class can be declared to have no instances with

```
abstract(person)
```

A class can also be declared to have no more descendants with

```
defined(colors)
```

A class can be declared to instantiate ephemeral objects, in which case its extension (the list of its instances) is not kept. An important consequence is that ephemeral objects may be garbage collected when they are no longer used. For this behavior, the class must inherit from ephemeral_object.

```
[action <: ephemeral_object(on:any, performed_by:object)]
```

A class definition can contain no slot definition, in which case the parenthesis may be dropped. This would be for instance the case for a forward definition, which is necessary in the case of recursive class definitions. Here is a simple example.

```
[parent <: thing]
[child <: thing(father:parent)]
[parent <: thing(son:child)]
```

A class cannot be defined twice in a different manner. It is necessary to restart the CLAIRE session if you want to change a class.

2.3 Parametric Classes

A class can be parameterized by a subset of its slots. This means that subsets of the class that are defined by the value of their parameters can be used as types. This feature is useful to describe parallel structures that only differ by a few points: parametrization helps describing the common kernel, provides a unified treatment and avoids redundancy.

A parameterized class is defined by giving the list of slot names into brackets. Parameters can be inherited slots, and include necessarily inherited parameters.

```
[stack[of] <: object(of:type,content:list[any],index:integer = 0)]
```

```
[complex[re,im] <: object(re:float = 0.0,im:float = 0.0)]
```

The set of real numbers is then

```
complex[re:float, im:{0.0}]
```

When the range of a parameter is a singleton, a shorthand notation is allowed: one can replace the previous declaration by **complex[re : float, im = 0.0]**. The set of stacks with range integer and the set of stacks which contain integers are respectively[4]

```
stack[of:{integer}]   stack[of : subtype[integer]]
```

---

[4]Here again, intervals of types are mentionned. {} denotes the smallest possible type (contained in any other type, so it is the empty set). Hence, {} .. integer denotes the set of all types contained in integer (such as the interval 1 .. 10).

2.4 Calls

Calls are the basic building blocks of a CLAIRE program. A call is a polymorphic function call (a *message*) with the usual syntax : a selector followed by a list of parameters between parentheses. A call is used to invoke a method or to read a slot. As we saw in the tutorial, slots and methods can be mixed to define one given selector (e.g., price), which is called a property.

```
eval(x), f(x,y,z), price(x), name(y)
```

If a slot is read before being defined (its value being unknown), an error is raised. To read a slot that may not be defined, one must use the `get(r:property;x:object)` method.

```
father(John)           ;; may provoke an error if father(John) is unknown
get(father,john)       ;; may return unknown
```

When the selector is an operation, such as +,-,%,etc... (% denotes set membership) an infix syntax is allowed (with explicit precedence rules). Hence the following expressions are valid.

```
1 + 2, 1 + 2 * 3
```

Note that new operations may be defined (Section 4.5). This syntax extends to boolean operations (and:& and or:|). However, the evaluation follows the usual semantic for boolean expression (e.g., (x & y) does not evaluate y if x evaluates to false).

```
(x = 1) & ((y = 2) | (y > 2)) & (z = 3)
```

The values that are combined with and/or do not need to be boolean values (although boolean expressions always return the boolean values `true` or `false`). Using a philosophy similar to LISP, all values are assimilated to true, except for false, nil and {}. The special treatment for the empty list and the empty set (cf. Conditionals, Section 3.3) yields a simpler programming style when dealing with lists or sets.

A dynamic functional call where the selector is evaluated can be obtained using the *call* method. For instance, call(+,1,2) is equivalent to +(1,2) and call(show,x) is equivalent to show(x). The difference is that the first parameter to call can be any expression. Notice that the compiler will detect if the first parameter is a constant property and substitute the "call" with the equivalent functional call. This is the key for writing parametric methods using the inline capabilities of CLAIRE (cf. Section 4.1). This also mean that using *call* is not a safe way to force dynamic binding, this should be done using the property *extensible*. Notice that call takes a variable number of arguments. A similar method named *apply* can be used to apply a property to an explicit list of arguments.

Since the use of *call* is somehow tedious, CLAIRE supports the use of variables (local or global) as selectors in a function call and re-introduce the call implicitly. For instance,

```
[compose(f:property, g:property, x:any) => f(g(x)) ]
```

is equivalent to

```
[compose(f:property, g:property, x:any) => call(f, call(g,x)) ]
```

The functional syntax for reading slots is elegant but not necessarily most readable. The dotted syntax (i.e., x.age versus age(x)) is an interesting alternative. CLAIRE supports this syntax for calls with one arguments. Its intended use is for handling slots (it may become the preferred way to use slots in future versions) although it actually works for any call with one argument. In the rest of the document we shall use both syntactical forms to illustrate their use.

2.5 Updates

Assigning a value to a variable is always done with the operator **:=** . This applies to local variables but also to the slots of an object. The value returned by the assignment is always the value that was assigned.

```
age(x) := 10,  john.father := mary
```

When the instantiation is not made from scratch, but depends on the former value of the variable, an implicit syntax ":op" can be used to combine the previous value with a new one using the

operation op. This can be done with any (built-in or user-defined) operation (An operation is a method with arity 2 with same type for the range and the first parameter that has been explicitly declared as an operation).

```
x.age :+ 1,   friends(john) :add mary, x.price :min 100
```

Note that the use of :op is pure syntactical sugar: `A(x) :op y` is equivalent to `A(x) := (A(x) op y)`. The receiving expression should not, therefore, contain side-effects as in the dangerous following example `A(x :+ 1) :+ 1`.

3. LISTS, SETS AND INSTRUCTIONS

3.1 Lists, Sets and Set expressions

CLAIRE provides two easy means of manipulating collections of objects: sets and lists. Lists are ordered heterogeneous collections. To create a list, one must use the list(...) instruction : it admits any number of arguments and returns the list of its arguments. Each argument to the list(...) constructor is evaluated.

```
list(a,b,c,d)    list(1,2 + 3)    list()
```

Sets are collections without order and without duplicates. Sets are created similarly with the set(...) constructor

```
set(a,b,c)    set(1,2 + 3)
```

Constant sets are valid CLAIRE types and can be build directly as

```
{a,b,c,d}    {3, 8}
```

A set can also be formed by selection. The result can either be a set with {x in a | P(x)}, or a list with list{x in a | P(x)}, when one wants to preserve the order of *a* and keep the duplicates if *a* was a list.

```
{x in class | (thing % x.ancestors) }
list{x in (0 .. 14) | x mod 2 = 0}
```

Also, the image of a set via a function can be formed. Here again, the result can either be a set with {f(x) | x in a} or a list with list{f(x) | x in a}, when one wants to preserve the order of *a* and the duplicates.

```
{(x ^ 2) | x in (0 .. 10)}
list{size(slots(x)) | x in class}
```

For example, we have the traditional average_salary method:

```
[av_sal(s:set[man]) : float -> sum(list{sal(m) | m in s}) / size(s)]
```

Last, two usual constructions are offered in CLAIRE to check a boolean expression universally (*forall*) or existentially (*exists*). A member of a set that satisfies a condition can be extracted (a non-deterministic choice) using the exists construct: exists(x in a | f(x)). For instance, we can write :

```
exists(x in (1 .. 10) | x > 2)        ;; returns 3 in most implementations
exists(x in class | length(x.ancestors) > 10)
```

Conversely, the boolean expression forall(x in a | f(x)) returns true if and only if f(x) is true for all members of the set a. The following example returns false (because of 1) :

```
forall(x in (1 .. n) | exists( y in (1 .. x) | y * y > x))
```

3.2 Blocks

Parentheses can be used to group a sequence of instructions into one. In this case, the returned value is the value of the last instruction.

```
(x := 3, x := 5)
```

Parentheses can also be used to explicitly build an expression. In the case of boolean evaluation (for example in an if), any expression is considered as true except false, the empty set {} and the empty list list().

```
(1 + 2) * 3    if (x = 2 & 1)
```

Local variables can be introduced in a block with the let construct. These variables can be typed, but it is LETnot mandatory. A *let* instruction contains a sequence of variable definitions and, following the *in* keyword, a body (another instruction). The scope of the local variable is exactly that body and the value of the *let* instruction is the value returned by this body.

```
let x := 1, y := 3 in (z := x + y, y := 0)
```

The value of local variables can be changed with the same syntax as an update to an object: the syntax :op is allowed for all operations op.

```
x := x + 1,   x :+ 1,   x :/ 2,   x :^ 2
```

When is a special form of the *let* which only evaluates the body if the value of the local variable (unique) is not *unknown* (otherwise, the returned value is unknown). This is convenient to use slots that are not necessarily defined as in the following example

```
when f := get(father,x) in printf("his father is ~S\n",f)
```

Local variables can also be introduced as pattern, that is a list of variables. In that case, the initial value must be a list of the right length. For instance, one could write :

```
let (x,y,z) := list(1,2,3) in x + y + z
```

The list of variable is simply introduced as a sequence of variables surrounded by two parenthesis. The most common use of this form is to assign the multiple values returned by a function with range tuple, as we shall see in the next section. If we suppose that f is a method that returns a tuple with arity 2, then the two following forms are equivalent :

```
let (x1,x2) := f() in ...
```

```
let l := f(), x1 := l[1], x2 := l[2] in ...
```

List of variables can also be assigned directly within a block as in the following example

```
(x1,x2) := list(x2,x1)
```

Although this mostly used for assigning the result of tuple-valed functions without any useless allocation, it is interesting to note that the previous example will be compiled into a nice value-exchange intruction without any allocation (the compiler is smart enough to determine that the list " list(x2,x1) " is not used as such).

The key principle of lexical variables is that they are local to the " let " in which they are defined. CLAIRE supports another similar type of block, which is called a temporary slot assignment. The idea is to change the value of a slot but only locally, within a given expression. This is done as follows :

```
let r(x) := y in e
```

changes the value of r(x) to y, executes e and then restore r(x) to its previous value. It is strictly equivalent to

```
let old_v := r(x) in (r(x) := y, let result := e in (r(x) := old_v, result)
```

3.3 Conditionals

if statements have the usual syntax (if <test> x else y) with implicit nestings (else if). The <test> expression is evaluated and the instruction x is evaluated if the value is different from false, nil or {} (cf. Section 2.4). Otherwise, the instruction y is evaluated, or the default value nil is returned if no *else* part was provided.

```
if (x = 1) x := f(x,y)
else if (x > 1) x := g(x,y)
else (x := 3, f(x,y))

if (let y := 3 in x + y > 4 / x) print(x)
```

If statements must be inside a block, which means that if they are not inside a sequence surrounded by parenthesis they must be themselves surrounded by parenthesis (thus forming a block).

case is a set-based switch instruction: CLAIRE tests the branching sets one after another, executes the instruction associated with the first set that contains the object and exits the case instruction without any further testing. Hence, the default branch is associated with the set *any*. As for a *if*, the returned value is nil if no branch of the *case* is relevant.

```
case x ({1} x + 1, {2,3} x + 2, any x + 3)
case x (integer (x := 3, print(x)), any error("~I is no good\n",x))
```

Note that the compiler will not accept a modification of the variable that is not consistent with the branch of the case (such as case x ({1} x := 2))

3.4 Loops

CLAIRE supports two types of loops: iteration and conditional loops (*while* and *until*). Iteration is uniquely performed with the *for* statement, it can be performed either on a list or on a set:

```
for x in (1 .. 3) a[x] := a[x + 3]
for x in list{x in class | size(x.ancestors) >= 4} printf("~S \n",x)
```

A set here is taken in a very general sense, that is an object that can be seen as a set through the enumeration method *set!*. This includes all CLAIRE types but is not restricted since this method can be defined on new classes. For instance, set!(n:integer) returns the subset of (0 .. 29) that is represented by the integer n taken as a bit-vector.

The iteration control structure plays a major role in CLAIRE. It is possible to optimize its behavior by telling CLAIRE how to iterate a new class (C) of set. This is done through adding a new restriction of the property *iterate* for this class C, which tells how to apply a given expression to all members of an instance of C. This may avoid the explicit construction of the equivalent set which is performed through the *set!* method. This optimization aspect is described in Section 4.6.

Conditional loops are also standard (the exiting condition is executed before each loop in a *while* and after each loop in a *until*),

```
while (x > 0) x :+ 1
until (x = 12) x :+ 1
while not(i = size(l)) (l[i] := 1, i :+ 1)
```

The value of a loop is {}. However, loops can be exited with the *break*(x) instruction, in which case the return value is the value of x.

```
for x in class (if (x % subtype[integer]) break(x))
```

3.5 Instantiation

Instantiation is the mechanism of creating a new object of a given class; instantiation is done by using the class as a selector and by giving a list of "<slot>=<value>" pairs as arguments.

```
complex(re = 0.0, im = 1.0)
person(age = 0, father = john)
```

Recall that the list of instances of a given class is only kept for classes that do not inherit from the *ephemeral_object* class. In fact, the creation of a new instance of a class corresponds to a function call to the method *close*. Objects with a name are represented by the class *thing*, hence descendants of *thing* (classes that inherit from thing) can be given a name with the definition operation ::. These named objects can later be accessed with their name, while objects with no name offer no handle to manipulate them after their creation outside of their block (objects with no name are usually attached to a local variable with a *let* whenever any other operation other than the creation itself is needed)

```
paul :: person(age = 10, father = peter)
```

Additionally, there is a simpler way of instantiating parameterized classes by dropping the slot names. All values of the parameter slots must be provided in the exact order that was used to declare the list of parameters. For instance, we could use :

```
complex(0.0,1.0), stack(integer)
```

3.6 Exception Handling

Exceptions are a useful feature of software development: they are used to describe an exceptional or wrong behavior of a block. Exception can be raised, to signal this behavior and are caught by exception handlers that surround the code where the exceptional behavior happened. Exceptions are CLAIRE objects (a descendent from the class *exception*) and can contain information in slots. The class *exception* is a descendent from *ephemeral_object*, so the list of instances is not kept. In fact, raising an exception *e* is done by creating an instance of the class *e*. Then, the method *close* is called: the normal flow of execution is aborted and the control is passed to the previously set dynamic handler. A handler is created with the following instruction.

```
try <expression> catch <class> <expression>
```

For instance we could write

```
try 1 / x catch any (printf("1/~A does not exists",x),0)
```

A handler "try e catch c f", associated with a class c, will catch all exceptions that may occur during the evaluation of e as long as they belong to c. Otherwise the exception will be passed to the previous dynamic handler (and so on). When a handler "catches" an exception, it evaluates the "f" part and its value is returned. The last exception that was raised can be accessed directly with the *exception!()* method.

The most common exceptions are errors and there is a standard way to create an error in CLAIRE using the error(s:string,l:listargs) instruction. This instruction creates an error object that will be printed using the string s and the arguments in l, as in a printf statement (cf. Section 6). Here are a few examples.

```
error("stop here")
error("the value of price(~S) is ~S !",x,x.price)
```

Another very useful type of exception is the *contradiction*. CLAIRE provides a class *contradiction* and a method contradiction!() for creating new contradictions. This is very commonly used for hypothetical reasoning with forms like

```
try ( world+(),    ; create a new world
     ...           ; performs an update that may cause a contradiction
catch contradiction (world-(),  ; return to previous world
               ...
```

4. METHODS AND TYPES

4.1 Methods

A method is the definition of a property for a given signature. A method is defined by the following pattern : a selector (the name of the property represented by the method), a list of typed parameters (the list of their types forms the domain of the method), a range expression and a body (a list of instructions introduced by ->).

$$[<\text{selector}>(<\text{typed parameters}>) : <\text{range}>^{\text{opt}} ->|=> \quad <\text{body}>]$$

```
[fact(n:{0}) : integer -> 1]
[fact(n:integer) : integer -> n * fact(n – 1)]
[print_test() : void -> print("Hello"), print("world\n") ]
```

If two methods have intersecting signatures and the property is called on objects in both signatures, the definition of the method with the smaller domain is taken into account. If the two domains have a non-empty intersection but are not comparable, a warning is issued and the result is implementation-dependent. The set of methods that apply for a given class or return results in another can be found conveniently with *methods*.

```
methods(integer,string) ; returns (string!@integer, make_string@integer)
```

The range declaration can only be omitted if the range is any. In particular, this is convenient when using the interpreter.

```
[loadMM() -> open(my_module), load("f1"), load("f2"), end(my_module)]
```

If the range is void (unspecified), the result cannot be used inside another expression (a type-checking error will be detected at compilation).

CLAIRE supports methods with a variable number of arguments using the *listargs* keyword. The arguments are put in a list which is passed to the (unique) argument of type listarg. For instance, if we define

```
[f(x:integer,y:listargs) -> x + size(y)]
```

A call f(1,2,3,4) will produce the binding x = 1 and y = list(2,3,4) and will return 4.

The body of a method is either a CLAIRE expression (the most common case) or an external (C) function. In the first case, the method can be seen as defined by a lambda abstraction. This lambda can be created directly through the following:

$$\text{lambda}[(<\text{typed parameters}>), <\text{body}>]$$

Defining a method with an external function is the standard way to import a C/C++ function in CLAIRE. This is done with the *function(...)* constructor, as in the following.

```
[f(x:integer,y:integer) -> function(my_version_of_f)]
```

```
[cos(x:float) -> function(cos_for_claire)]
```

The integration of external functions is detailed in Section 6.5.

CLAIRE also provides *inline* methods, that are defined using the => keyword before the body instead of ->. An inline method behaves exactly like a regular method. The only difference is that the compiler will use in-line substitution in its generated code instead of a function call when it seems more appropriate[5]. Inline methods can be seen as polymorphic macros, and are quite powerful because of the combination of parametric function calls (using *call*(...)) and parametric iteration

---

[5] The condition for substitution is implementation-dependent. For isntance, the compiler checks that the expression that is substituted to the input parameter is simple (no side-effects and a few machine instructions) or that there is only one occurence of the parameter.

(using *for).* Let us consider the two following examples, where subtype[integer] is the type of everything that represents a set of integers:

```
[sum(s:subtype[integer]) : integer
 => let x := 0 in (for y in s x :+ y, x) ]

[min(s:subtype[integer], f:property) : integer
 => let x := 0, empty := true in
      (for y in s
         (if empty (x := y, empty := false)
          else if call(f,y,x) x := y),
       x) ]
```

For each call to these methods, the compiler performs the substitution and optimizes the result. For instance, the optimized code generated for sum({age(x) | x in person}) and for   min({x in 1 .. 10| f(x) > 0}, >) will be

```
let x := 0 in
  (for %v in person.instances
     let y := %v.age in x :+ y, x)

let x := 0, empty := true, y := 1, max := 10 in
      (while (y <= max)
         (if (f(y) > 0)
             (if empty (x := y, empty := false)
              else if (y > x) x := y),
          y :+ 1),
       x)
```

Notice that in these two cases the construction of temporary sets is totally avoided. The combined use of inline methods and functional parameters provides an easy way to produce generic algorithms that can be instantiated as follows.

```
[mymin(l:list[integer]) : integer -> min(l, my_order) ]
```

The code generated for the definition of mymin @ list[integer] will use a direct call to *my_order* (with static binding) and the efficient iteration pattern for lists, because *min* is an inline method. In that case, the previous definition of min may be seen as a pattern of algorithms.

4.2 Types

CLAIRE uses an extended type system that is built on top of the set of classes. As a class, a type denotes a set of objects, but it can be much more precise than a class. Since methods are attached to types (by their signature), this allows to attach methods to complex sets of objects.

Any class (even parameterized) is a type. Finite sets of objects can also be used as types. For example, {john, jack, mary} and {1,4,9} are types.

$$<type> \equiv \quad <class> \mid <class>[<parameter>:<type>^{seq}] \mid \{<item>^{seq} \} \mid$$
$$(<item> .. <item>) \mid (<type> \; U \; <type>) \mid (<type> \; ^\wedge \; <type>) \mid$$
$$set[<type>] \mid list[<type>] \mid subtype[<type>] \mid$$
$$tuple(<type>^{seq})$$

Extended intervals (x .. y) are accepted for items x and y in some classes that are ordered by <=. The set of such classes is integer, float, symbols and strings. For example, (1 .. 10) and ('a' .. 'z'), the set of lower-case characters are types. Moreover, CLAIRE offers type constructors for union, intersection, typed sets, typed lists and tuples. For example, integer U float denotes the set of numbers and (1 .. 100) ^ (-2 .. 5) denotes the intersection of both integer intervals, i.e. (1 .. 5). set[(1 .. 10)] denotes the type of sets of integers between 1 and 10 and list[float] denotes all lists of floats. Finally, the last constructor is used to denote tuples. For instance, tuple(integer, char) denotes the set of pairs with an integer as first element and a character as second.

Classes are sorted with the inheritance order. This order can be extended to types with the same intuitive meaning that a type t_1 is a subtype of a type t_2 if the set represented by t_1 is a subset of that represented by t_2. The relation "t_1 is a subtype of a type t_2" is noted $t_1 <= t_2$. This order supports the introduction of the " subtype " constructor: subtype[t] is the type of all types that are less than t. For example, `subtype[ list[integer U float]]` denotes the set of types are subtypes of `list[integer U float]`, which are types that contain lists of numbers. For instance, `list[integer]` and `tuple(integer U float, integer U float)` are both members of this type.

A syntactical remark: CLAIRE also accepts the expression C[p = v] as a shortcut for C[p:{v}]. For instance, `stack[of = integer]` is the set of stacks whose parameter "of" is exactly integer, whereas `stack[of:subtype[integer]]` is the set of stacks whose parameter (a type) is a subset of integer.

The tuple types play an important role since they are used to represent functions that returns multiple values. A function that should return multiple values is represented in CLAIRE with a method with range tuple that returns the list of these values. For instance, here is a method that returns the minimum and the " second smallest " members of a list :

```
[min2(l:list[integer]) : tuple(integer,integer)

  -> let x1 := INT+, x2 := INT+ in
            (for x in l (   if (x < x1) (x2 := x1, x1:= x)

                                   else if (x < x2) (x2 := x)),
              list(x1,x2)) ]
```

By declaring the range of this method as tuple, we allow the compiler to perform a few optimizations. Especially, if min2 is used in a multi-variable assignment such as

```
 let (x,x') := min2(l) in ...
```

no useless list allocation will be perfomed, which enables the use of tuple-valued methods as the righteous implementation of multi-valued functions.

4.3 Polymorphism

In addition to the traditional "objet-oriented" polymorphism, CLAIRE also offers two forms of parametric polymorphism, which can be considered as advanced features and skipped by a novice reader.

(1) There often exists a relation between the types of the arguments of a method. Capturing such a relation is made possible in CLAIRE through the notion of an "extended signature". For instance, if we want to define the operation "push" on a stack, we would like to check that the argument y that is being pushed on the stack s belongs to the type of(s), that we know to be a parameter of s. The value of this parameter can be introduced as a variable and reused for the typing of the remaining variables (or the range) as follows.

```
[push(s:stack[of = X], y:X) -> ( content(s) :add y, index(s) :+ 1)]
```

The declaration `s:stack[of = X]` introduced X as a type variable with value `of(s)`, since `stack[of]` was defined as a parameterized class. using X in `y:X` simply means that y must belong to the type of(s). Such intermediate type variables must be free identifiers (the symbol is not used as the name of an object) and must be introduced with one of the following templates:

```
<class>[pi=vi,...,], tuple(...,vi,...)
```

The use of type variables in the signature can be compared to pattern matching. The first step is to bind the type variable. If (p = V) is used in `c[ ...]` instead of `p:t`, it means that we do not put any restriction on the parameter p but that we want to bind its value to V for further use. Note that this is only interesting if the value of the parameter is a type itself. Similarly, the use of type variables inside `tuple(...)` only applies to tuple of types, since the variable will be bound to the i-th component of the tuple (*not to the type of this i-th component*). Once a type variable V is defined, it can be used to form a pattern (called a `<type with var>` in the CLAIRE syntax in Appendix A) as follows:

```
<type with var> ≡ <type> │ <var> │ {<var>} │
                  tuple(<type with var>^seq+ ) │
                  <class>[⟨ <var>:<type with var> │ <var>=<var> ⟩^seq+]
```

For instance, here is the definition of a method that pushes a pair (y,0) on the stack, assuming that each stack member is a pair whose second item is an integer :

```
[push(s:stack[of = tuple(X,integer)], y:X) -> push(s, list(y,0)) ]
```

(2) The second advanced typing feature of CLAIRE is designed to capture the fine relationship between the type of the output result and the types of the input arguments. When such a relationship can be described with a CLAIRE expression e(x1,...,xn), where x1, ..., xn are the types of the input parameters, CLAIRE allows to substitute type[e] to the range declaration. It means that the result of the evaluation of the method should belong to e(t1,...,tn) for any types t1,...,tn that contain the input parameters.

For instance, the identity function is known to return a result of the same type as its input argument (by definition !). Therefore, it can be described in CLAIRE as follows.

```
[id(x:any) : type[x] -> x]
```

In the expression that we introduce with the *type[e]* construct, we can use the types of the input variables directly through the variables themselves. For instance, in the "type[x]" definition of the Identity example, the "x" refers to the type of the input variable. Notice that the types of the input variables are not uniquely defined. Therefore, the user must ensure that her "prediction" e of the output type is valid for any valid types t1, ..., tn of the input arguments.

The expression e may use the extra type variables that were introduced earlier. For instance, we could define the "top" method for stacks as follows.

```
[top(s:stack[of = X]) : type[X] -> content[s.index] ]
```

The "second-order type" e (second-order means that we type the method, which is a function on objects, with another function on types) is built using the basic CLAIRE operators on types such as U, ^ and some useful operations such as "member". If c is a type, member(c) is the minimal type that contains all possible members of c. For instance, member({c}) = c by definition. This is useful to describe the range of the enumeration method set!. This method returns a set, whose members belong to the input class c by definition. Thus, we know that they must belong to the type member(X) for any type X to whom c belongs (cf. definition of member). This translates into the following CLAIRE definition.

```
set!(c:class) : type[set[member(c)]] -> instances(c)
```

For instance, if c belongs to (A .. B) then set!(c) belongs to set[B].

To summarize, here is a more precise description of the syntax for defining a method:

```
[<property> (<v_i>:<t_i>, i ∈ (1 .. n)) : <range> -> <exp> ]
```

Each type t_i for the variable v_i is an "extended type" that may use type variables introduced by the previous extended types t_1, t_2 ... t_{i-1} . An extended type is defined as follows.

```
<et> ≡  <class> │ <set> │ <var> │ (<et>  ^ │ U <et>) │ (<obj> .. <obj>)│
        set[<var>] │ list[<var>] │ class[<var>] │ tuple(<et>^seq)
```

The <range> expression is either a regular type or a "second order type", which is a CLAIRE expression e introduced with the "type[e]" syntactical construct .

```
<range> ≡    <type> │ type[<expression>]
```

Here is a last interesting example, which illustrates the power of extended signature to capture polymorphic functions. We want to import a method sort from C that sorts a list according to an order specified as a parameter (a method). To use an optimized C implementation, we want to make sure that the method will apply correctly to members of the list and that no type error will occur inside the C function. Here is the correct CLAIRE definition.

```
[sort(m:method[domain:tuple(X,X), range = boolean], l:list[X])
 : type[l]  -> function(sort_list)]
```

4.4 Escaping Types

There are two ways to escape type checking in CLAIRE. The first one is casting, which means giving an explicit type to an expression. The syntax is similar to C

          `<cast>` ≡   `(`<expression> **as** <type>`)`

This will tell the compiler that <expression> should be considered as having type <type>. Casting is ignored by the interpreter and should only be used as a compiler optimization. The second type escaping mechanism is the non-polymorphic method call, where we tell what method we want to use by forcing the type of the first argument. This is equivalent to the *super* message passing facilities of many object-oriented language.

          `<super>` ≡   <selector>`@`<type>`(`<exp>^{seq}`)`

The instruction f@c(...) will force CLAIRE to use the method that it would use for f(...) if the first argument was of type c (CLAIRE only checks that this first argument actually belongs to c).

CLAIRE supports side-effects on lists and sets which are not type safe. What this means is that these operations may cause further type error that will not be caught by the compiler. The cause is the fact that it is possible, under certain occasions (such as a cascade of function calls passing a list as an argument), to add a new member *x* to a list *l* that currently belongs to *list[t]*, while *x* does not belong to *t*. Although rare, this possibility (due to the nature of the *list[t]* type) means that the compiler will only use these kinds of type when it can be sure that no special side-effects can be performed (fortunately, this is often the case). The rest of the time, the compiler will generate code that relies on dynamic typing (type checking at run time).

4.5 Selectors, Properties and Operations

As we previously said, CLAIRE supports two syntaxes for using selectors, f(...) and (.... f ....). The choice only exists when the associated methods have exactly two arguments. The ability to be used with an infix syntax is attached to the property f and not to a particular restriction of it (a method attached to a given signature).

```
f :: operation()
```

Once f has been declared as an operation, CLAIRE will check that it is used as such subsequently. Restrictions of f can then be defined with the usual syntax

```
[f(x:integer, y:integer) : ... ]
```

Note that declaring f as an operation can only be done when no restriction of f is known. If the first appearance of f is in the declaration of a method, f is considered as a normal selector and its status cannot be changed thereafter. Each operation is an object (inherits from `property`) with a *precedence* slot that is used by the reader to produce the proper syntax tree from expressions without parentheses.

```
gcd :: operation(precedence = precedence(/))
12 + 3 gcd 4              ;; same as 12 + (3 gcd 4)
```

So far we have assumed that any method definition is allowed, provided that inheritance conflict may cause warning. Once a property is compiled, CLAIRE uses a more restrictive approach since only new methods that have an empty intersection with existing methods (for a given property) are allowed. This allows the compiler to generate efficient code. It is possible to keep the "open" status of a property when it is compiled through the *abstract* declaration.

```
abstract(f)
```

Such a statement will force CLAIRE to consider **f** as an "abstract" parameter of the program that can be changed at any time. In that case, any re-definition of f (any new method) will be allowed.

When defining a property parameter, one should keep in mind that another user may redefine the behavior of the property freely in the future.

It is sometimes useful to model a system with redundant information. This can be done by considering pairs of relations inverse one of another. In this case the system maintains the soundness of the database by propagating updates on one of the relations onto the other. For example if husband is a relation from the class man onto the class woman and wife a relation from woman to man, if moreover husband and wife have been declared inverse one of another, each modification (addition or retrieval of information) on the relation husband will be propagated onto wife. For example husband(mary) := john will automatically generate the update wife(john) := mary. Syntactically, relations are declared inverses one of another with the declaration

```
inverse(husband) := wife
```

This can be done for any relation: slots and arrays (cf. Section 5). Inverses introduce an important distinction between multi-valued relations and mono-valued relations. A relation is multi-valued in CLAIRE when its range is a subset of bag (i.e. a set or a list). In that case the slot *multivalued* of the relation is set to true[6] and the set associated with an object x is supposed to be the set of values associated with x through the relation.

This has the following impact on inversion. If r and s are two mono-valued relations inverse one of another, we have the following equivalence :

s(x) = y $\Leftrightarrow$ r(y) = x

In addition, the range of r needs to be included in the domain of s and conversely. The meaning is different if r is multi-valued since the inverse declaration now means :

s(x) = y $\Leftrightarrow$ x $\in$ r(y)

Two multi-valued relations can indeed be declared inverses one of another. For example, if parents and children are two relations from person to set[person] and if inverse(children) = parents, then

children(x) = {y in person | x $\in$ parents(y)}

Modifications to the inverse relation are triggered by updates (with :=) and creations of objects (with filled slots). Since the explicit inverse of a relation is activated only upon modifications to the database (it is not retroactive), one should always set the declaration of an inverse as soon as the relation itself is declared, before the relation is applied on objects. This will ensure the soundness of the database. To escape the triggering of updates to inverse relations, the solution is to fill the relation with the method put instead of :=. For example, the following declaration

```
let john := person() in (put(wife,john,mary), john)
```

does the same as

```
john :: person(wife = mary)
```

without triggering the update husband(mary) := john.

4.6 Iterations

We just saw that CLAIRE will produce in-line substitution for some methods. This is especially powerful when combined with parametric function calls (using *call*(...)) taking advantage of the fact that CLAIRE is a functional language. There is another form of code substitution supported by CLAIRE that is also extremely useful, namely the iteration of set data structure.

Any object s that understands the *set!* method can be iterated over. That means that the construction *for x in s e(x)* can be used. The actual iteration over the set represented by s is done by constructing explicitly the set extension. However, there often exists a way to iterate the set structure without constructing the set extension. The simplest example is the integer interval structure that is iterated with a while loop and a counter.

It is possible to define iteration in CLAIRE through code substitution. This is done by defining a new inline restriction of the property *iterate*, with signature (x:X,v:Variable,e:any). The principle is

---

[6] This slot can be reset to false in the rare case when the relation should actually be seen as mono-valued.

that CLAIRE will replace any occurence of (for v in x e) by the body of the inline method as soon as the type of the expression x matches with X (v is assumed to be a free variable in the expression e). For instance, here is the definition of iterate over integer intervals :

```
[iterate(x:interval[min:integer,max:integer],v:Variable,e:any)
  =>  let v := min(x), %max := max(x) in
          (while (v <= %max) (e, v :+ 1)) ]
```

Here is a more interesting example. We can define hash tables as follows. A table is defined with a list (of size $2^n - 3$, which is the largest size for which a chunk of size 2^n is allocated), which is full of "unknown" except for these objects that belong to the set. Each object is inserted at the next available place in the table, starting at a point given by the hashing function (a generic hashing function provided by CLAIRE: *hash*).

```
[htable <: object(    count:integer = 0,
                      index:integer = 4,
                      arg:list = nil)]
[set!(x:htable) -> {y in x.arg | known?(y)}]
[insert(x:htable,y:any)
  -> let l := x.arg in
        (if (x.count >= length(l) - 3)
          (x.arg := make_list(^2(x.index - 3), unknown),
           x.index :+ 1, x.count := 0,
           for z in {y in l | known?(y)} insert(x,z),
           insert(x,y))
        else let i := hash(l,y) in
          (until (l[i] = unknown | l[i] = y)
                  (if (i = length(l)) i := 1 else i :+ 1),
           if (l[i] = unknown)
               (x.count :+ 1, l[i] := y)))]
```

To iterate over such tables without computing set!(x) we define

```
[iterate(s:htable, v:Variable, e:any)
  => for v in s.arg (if known?(v) e) ]
```

Thus, CLAIRE will replace

```
let s:htable := ... in sum(s)
```

by

```
let s:htable := ... in
  (let x := 0 in
    (for v in s.arg
       (if known?(v) x :+ v),
     x))
```

The use of *iterate* will only affect the compiled code unless one uses oload, that calls the optimizer for each new method. *iterate* is a convenient way to extend the set of CLAIRE data structure that represent sets with the optimal efficiency. Notice that, for a compiled program, we could have defined *set!* as follows (this definition would be valid for any new type of set).

```
[set!(s:htable) -> {x | x in s}]
```

5. ARRAY, RULES AND HYPOTHETICAL REASONING

5.1 Generalized Arrays

Arrays can be defined in CLAIRE with the following syntax :

```
[ <name>[var:(<integer> .. <integer>)] : <type> -> <expression(var)> ]
```

The <type> is the range of the array and <expression> is an expression that is used to fill the array. This expression may either be a constant or a function of the variables of the array (i.e., an expression in which the variables appear). If the expression is a constant, it is implicitly considered as a default value, the domain of the array may thus be infinite. If the default expression is a function, then the array is filled when it is created, so the domain needs to be finite. When one wants to represent incomplete information, one should fill this spot with the value unknown. For instance, we can define

```
[square[x:(0 .. 20)] : integer -> x * x]
```

Arrays can be accessed through square brackets and can be modified with assignment expressions as for local variables.

```
square[1], square[2] := 4, square[4] :+ 5,
```

Arrays have been extended in CLAIRE by allowing to use any type instead of an integer interval for their domain. They are thus useful to model relations, when the domain of a relation is more complex than a class (in which case a slot should rather be used to model the relation).The syntax for such a definition is, therefore,

```
<array> ≡      [<name>[var:<type>] : <type> -> <expression(var)>]
```

This is a way to represent many sorts of complex relations and use them as we would with arrays. Here are some examples.

```
[creator[x:class] : string -> "who created that class" ]
[maximum[x:set[0 .. 10]] : integer
       -> (if x min(x,> @ integer) else 0) ]
[color[x:{car,house,table}] : colors -> unknown ]
```

We can also define two-dimensional arrays such as

```
[distance[x:tuple(city,city)] : integer -> 0]
[cost[x:tuple(1 .. 10,1 .. 10)] : integer -> 0]
```

The proper way to use such an array is distance[list(denver,miami)] but CLAIRE also supports distance[denver,miami]. CLAIRE also supports a more straightforward declaration such as :

```
[cost[x:(1 .. 10), y:(1 .. 10)] : integer -> 0]
```

As for properties, arrays can have an explicit inverse, which is either a property or an array. Notice that this implies that the inverse of a property can be set to an array.

5.2 Logical Assertions

The purpose of the two next sections is to describe how to write logical rules. A rule in CLAIRE is made by associating a logical condition to an expression. Each time that the condition becomes true for one or two entities, the expression will be evaluated for these entities. To define a logical condition, we use the logic language defined by the following grammar.

```
<assertion> ≡ <expression> <comp> <expression> |
              exists(<var>, <assertion>) |
              not(<assertion>) |
              if (<expression><comp><expression>) <assertion>
              else <assertion> |
```

```
                        <assertion> & <assertion> |
                        <assertion> | <assertion>

   <expression> ≡ <variable> | <entity> | <property>(<expression>) |
                  <array>[<expression>] | <expression> <operation> <expression> |
                  {<var> in <expression> | assertion} |
                  list{<var> in <expression> | assertion}
```

The basic building block of a logical condition is a logical expression. The set of logical expressions is a subset of CLAIRE expressions with the same semantics. For instance, here are some logical expressions.

```
age(Paul), Paul, size[x], x + y, x + (y * z),
{x in person| age(x) > 10}
```

The value of a logical expression is a CLAIRE entity. Logical assertions are made by combining expressions. The most common type of assertion is obtained by comparing two expressions with a comparison operation. A comparison operation is an operation that returns a boolean value. For instance, =, <, <=, % are very commonly used comparisons. Here are some logical assertions:

```
age(Paul) = 10,
size({x in person | Paul % friend(x)}) < 2,
y % integer
```

Existential quantification (there exists a ... such that ...) is introduced with the *exists* construction. For instance, here is how we say that there exists a common friend (z) to two persons x and y:

```
exists(z, x % friend(z) & y % friend(z) )
```

Existential variables can be typed, as in

```
exists(z:woman, x % friend(z) & y % friend(z) )
```

Notice that this form of *exists* in only permitted inside logical conditions because there is an implicit iteration over any (i.e., exists(x, P(x)) ⇔ exists(x in any | P(x))). Variables that are not introduced in the logical condition by an "exist" or an "{.. in .. | ...}" are called free variables, they can be used within their scope to form expressions. Last, assertions can be formed as a conditional expression using if. The test in a logic conditional is necessarily of the form (expression comp expression) and it must have two branches. Here is an example that one could use to compute taxes:

```
if (r < 10000) x = 0
else if (r < 20000) x = r * 0.1
else x = r * 0.2 − 2000
```

The value of a logical assertion is always a boolean, thus logical assertions can be combined with & (and) and | (or). In addition to pre-defined operations such as <= or +, it is possible to use new properties inside logical assertions but they need to be described using the *description* array and a set of keyword including *comparison, bijection, binary_operation, monoid, group_operation* and *mapping*. For instance, to use the operation *less_than* inside a logical assertion, we need to declare :

```
description[less_than] := comparison
```

A complete description of *description* can be found in Appendix B.

5.3 Rules

A rule in CLAIRE is made by associating a logical condition to an expression. The rule is attached to one or two entities: each time that the condition becomes true, the expression will be evaluated for these entities. The interest of rules is to attach an expression not to a functional call (as with methods) but to a logical condition : the programmer needs not to explicitly place method calls wherever the condition might become true, the system automatically evaluates the expression whenever the condition becomes true.

To define a rule, we must define one or two free variables, that are introduced as parameters of the rule (these free variables can be seen as universally quantified variables), a condition, that is given as an assertion using the previously defined variables and a conclusion that is preceded by =>. Here is a classical transitive closure example:

```
[r1(x:person, y:person) ::
   exists(z, x % friend(z) & z % friend(y) )
   => friend(y) :add x]
```

Rules are named (for easier debugging) and can use any CLAIRE expression as a conclusion, using both free and existentially quantified variables, such as in:

```
[r2(x:person, y:person) ::
   exists(z, age(x) + age(y) = age(z) )
   => printf("~S ~S ~S\n",x,y,z)]
```

It is now important to understand how a rule works. Rules are checked (efficiently) each time that an *event* occurs. Events are update to slots or arrays that are declared as event-generating with the *event* statement.

<events definition> ≡ **event** │ **noevent(** < <array> │ <property> >^{seq}**)**

These declarations can be grouped together with the following syntax :

```
event(age,friend,fib)   ⇔  (event(age), event(friend), event(fib) )
```

Rules are associated to a logical condition, which uses certain relations. One can foresee that the condition is liable to become true when these relations are updated. The control offered to the user is the ability to choose which of the relations (arrays and slots) that appear in the condition of the rule should trigger an evaluation of that condition (and in the case of a fulfilled requirement, evaluate the conclusion of the rule). Hence, a rule is triggered only upon updates to relations that are relevant to the rule (that may change the evaluation of the condition) and that are declared events. Rule triggering can be traced using *trace(if_write)*, as shown in Appendix C. Note that a rule like:

```
[r1(x:(0 .. 20)) :: x mod 2 = 0 => printf("~S is even\n",x)]
```

will never be fired since no relation appears in its condition. Since rules are triggered by events, a rule is never applied at the time it is created to objects that would already satisfy its condition. For instance, let us define the following rule to fill the array *fib* with the Fibonacci sequence.

```
[r3(x:(0 .. 100), y:integer) ::
   if (x < 2) y = 1 else y = fib[x − 1] + fib[x − 2] =>  fib[x] := y]
```

This rule will *not* compute fib[x] for all x in (0 .. 100). However, as soon as we give two consecutive values fib[x] and fib[x+1] for some x, the rule will compute all further values. Thus the right CLAIRE statement should be

```
[r3(x:(0 .. 100), y) ::
   y = fib[x − 1] + fib[x − 2] =>  fib[x] := y ]

(fib[0] :=1, fib[1] := 1)
```

or, more simply

```
fib[0] :=1, fib[1] := 1,
for x in (2 .. 100)  fib[x] := fib[x − 1] + fib[x − 2]
```

A rule considers as events all slots or arrays that have been declared as such *before* the rule has been declared, so the event declaration needs to precede the rule declaration, in order to create the "demons" that will watch over the given relation and fire the rule when needed. The declaration noevent(...) is usually used to correct mistakes at the top-level, but it can also be used to prevent explicitly a rule to react to some relations that were declared previously as events for other rules. By spreading the *event* declarations (and occasionally *noevent* declarations) before and between the rules, one has a complete control over the triggering of the rules. Updates that are considered as events are:

- r(x) := y, where r is a slot of x and event(r) has been declared.
- a[x] := y, where a is an array and event(a) has been declared.
- r(x) :add y, where r is a multi-valued property of x (with range bag) and event(r) has been declared.
- a[x] :add y, where a is a multi-valued array and event(a) has been declared.

Note that an update of the type r(x) :delete y (resp. a[x] :delete y), where r is a slot of x (resp. a is an array), will never be considered an event if r is multi-valued. However, one can always replace this declaration by r(x) := delete(r(x), y) which is an event, but which costs a memory allocation for the creation of the new r(x).

Each time such an event occur, the conclusion of each rule is evaluated for each set of objects that satisfy the condition because of the event. Although this usually means that the condition was not satisfied before, this is not always the case. For instance, consider the rule:

```
[r4(x:integer) :: f[x] = 1 | f[x] = 2 => print(x)]
```

if f[x] is 1 and is changed to 2, the rule will be triggered. This is the desired behavior in most cases, but sometimes it is needed that the rule is triggered only once. The proper solution is to guard the conclusion with a flag:

```
[flag[x:integer] : boolean -> false]
[r4'(x:integer) ::
   f[x] = 1 | f[x] = 2 => if not(flag[x]) (flag[x] := true, print(x))
```

It is possible to perform an update without creating any triggering event with the method *put*. It is possible to perform the rule propagation later with the method *propagate*. This allows the user to keep a precise control over when and how rule propagation is performed. Also, since the logical variables may be used in the conclusion, it is *necessary* not to introduce new local variables in the conclusion with the same name.

5.4 Hypothetical Reasoning

In addition to rules, CLAIRE also provides the ability to do some hypothetical reasoning. It is indeed possible to make hypotheses on part of the knowledge (the database of relations) of CLAIRE, and to change them whenever we come to a dead-end. This possibility to store successive versions of the database and to come back to a previous one is called the world mechanism (each version is called a world). The slots or arrays x on which hypothetical reasoning will be done need to be specified with the declaration store(x). This command has the same syntax as *events* . For instance, the declaration

```
store(age,friend,fib)   ⇔   store(age), store(friend), store(fib)
```

Each time we ask CLAIRE to create a new world, CLAIRE saves the status of arrays and slots declared with the store command. Worlds are represented with numbers, and creating a new world is done with *world+()*. Returning to the previous world is done with *world-()*. Returning to a previous world n is done with *world=(n)*. Worlds are organized into a stack (sorry, you cannot explore two worlds at the same time) so that save/restore operations are *very* fast. The current world that is being used can be found with *world?()*, which returns an integer.

In addition, you may accept the hypothetical changes that you made within a world while removing the world and keeping the changes. This is done with the *world!-* and *world!=* methods. *world!-()* decreases the world counter by one, while keeping the updates that were made in the current world. It can be seen as a collapse of the current world and the previous world. *world!=(n)* repeats *world!-()* until the current world is *n*.

For instance, here is a simple program that solves the n queens problem (the problem is the following : how many queens can one place on a chessboard so that none are in situation of chess, given that a queen can move vertically, horizontally and diagonally in both ways ?)

```
[column[n:(1 .. 8)] : (1 .. 8) -> unknown]
[possible[x:(1 .. 8), y:(1 .. 8)] : boolean -> true]
event(column), store(column, possible)
```

```
[r1(x,y) :: exists(z, column[z] = y)
   => possible[x,y] := false]
[r2(x,y) :: exists(z, column[z] + z = x + y)
   => possible[x,y] := false]
[r3(x,y) :: exists(z, column[z] - z = x - y)
   => possible[x,y] := false]

[queens(n:(0 .. 8)) : boolean
      -> if (n = 0) true
         else exists(p in (1 .. 8) |
             (possible[n,p] &
                ( (world+(),
                   column[n] := p,
                   queens(n - 1)) |
                  world-())) ]

for p in (1 .. 8)
   (if possible[n,p]
         (world+(),
          column[n] := p,
          if queens(n - 1) break(true),
          world-(),
          false )) ]

  queens(8)
```

In this program queens(n) returns true if it is possible to place n queens. Obviously there can be at most one queen per line, so the purpose is to find a column for each queen in each line : this is represented by the column array. So, we have eight levels of decision in this problem (finding a line for each of the eight queens). The search tree (these imbricated choices) is represented by the stack of the recursive calls to the method queens. At each level of the tree, each time a decision is made (an affectation to the array column), a new world is created, so that we can backtrack (go back to previous decision level) if this hypothesis (this branch of the tree) leads to a failure.

Note that the array possible which tells us whether the nth queen can be set on the pth line is filled by means of rules triggered by possible (declared event) and that both possible and column are declared store so that the decisions taken in worlds that have been left are undone (this avoids to keep track of decisions taken under hypotheses that have been dismissed since).

Updates on lists can also be "stored" on worlds so that they become defeasible. Instead of using the nth= method, one can use the method *store(l,x,v,b)* that places the value v in l[x] and stores the update if b is true. In this case, a return to a previous world will restore the previous value of l[x]. Here is a typical use of store:

```
store(l,n,y,l[n] != y)
```

This is often necessary for arrays with range list or set. For instance, consider the following :

```
[A[i:integer] : tuple(integer,integer,integer) -> list(0,0,0)]
(let l := A[x] in
   (l[1] := 3, l[3] := 3))
```

even if store(A) is declared, the manipulation on l won't be recorded by the world mechanism. You would need to write :

```
A[x] := list(3,A[x][2],3)
```

Using store, you can use the original (and more space-efficient) pattern and write:

```
(let l := A[x] in
   (store(l,1,3,true), store(l,3,3,true)))
```

Last, it is worth noticing that hypothetical reasoning makes a heavy use of contradictions and their associated handlers.

5.5 Additional tuning

CLAIRE supports some additional tuning of its rules through the *mode* declaration. First, we noticed that a rule is only applied to future events. There are some cases, however, where it is convenient to apply a rule to all existing objects, which we call *enforcing* the rule. The enforcing mode can be turned on and off with the *mode* declaration.

$$\texttt{<modes>} \equiv \textbf{mode(}\ \texttt{<\ default\ |\ break\ |\ set\ |\ <boolean>\ |\ <integer>\ >}^{seq}\textbf{)}$$

mode(true) will set the enforcing mode, which will cause any new rule to be applied to all possible objects, until *mode(false)* is invoked. Enforcing rules may be expensive and it is usually preferable to define rules before the objects that they designate, rather than using this feature.

Rules should not, in general, be written in such a way that the result is order-dependent. The order in which they are triggered depend on the event (the propagation pattern) and the order in which the rules were entered. If it becomes necessary to have a more precise control over this order, priorities may be used. A priority is an integer attached to the rule using the *mode* declaration. *mode(i)* sets the current level of priority to *i*. This level will be attached to all newly created rules (the default level is 10). CLAIRE will ensure that rules with higher priority will be triggered first for each new event. Notice that there is no implicit stack structure for triggering rules: the events generated by the application of a first rule may cause a new rule to be evaluated before a second rule is applied to the original event.

The conslusion is applied to any pair of object that is obtained through a logical derivation of the the conclusion and the update. This assumes that the conclusion can be fired more than once when the logical expression is redundant (multiple derivation paths for the same pair). However, it may be wrong to apply the conclusion twice even if a pair is obtained from two different paths. Consider the following example :

```
[strange1(x:person) :: x.age = 18 | x.age > 10  => give(x, $1000)]
```

An update "John.age := 12" may cause the rule to be fired twice. The solution in CLAIRE is to use the *mode(set)* declaration before the rule which will force the computation of the set of pairs before firing the conclusion (thus eliminating duplicates). The regular mode is obtained with mode*(default)*. Similarly, rules should have a monotonous behaviour which means that their conclusion should not invalidate the condition. Non-monotonous rules are supported by CLAIRE but the user must be aware of possible difficulties. Let us consider a second example :

```
[strange2(x:person, y:person) :: x.age = 18 & y % x.friends
     => (x.age := 19, invite(x,y)) ]
```

Should the rule invite one friend (which one) or all friends when the age of John is set to 18. The default behavior as well as the "set" behavior will invite all friends. The *mode(break*) declaration will ensure that the conclusion is fired only once for each update event. Using non-monotonic rules is tricky, but it is sometimes very useful.

Last, CLAIRE supports rules with no conclusions, which are only defined by a condition. These rules are called queries and can be used to manipulate a logical condition. A query can be used as a method: R(x) will return the set of objects y such that (x,y) satisfy the condition. For instance, suppose that we have defined :

```
[r5(x:person, y:integer) ::
   y = size({z in person | z % children(x)})
   =>  printf("~S has ~A children\n") ]
```

We can define a similar query as follows :

```
[r5'(x:person, y:integer) :: y = size({z in person | z % children(x)})]
```

This query can be evaluated directly, such as in r5'(john) or implicitly as in inverse(r5',2). Queries are not necessarily optimized and are provided as a programming convenience. Here is a last example, which illustrates the use of queries together with the use of bijections inside a logic condition. The goal is to reason about lines and points (that belong to lines) using logic rules (that are not easy to represent with a "pure" object-oriented model).

```
[point <: object]                       // naive class
line :: tuple(point,point)              // a line is defined by two points
[line!(a:point,b:point) : line -> list(a,b)]    // constructor
[p1(l:line) : point -> l[1]]            // first point
[p2(l:line) : point -> l[2]]            // second point

description[line!] := bijection         // tell CLAIRE about the
projection1[line!] := p1                // bijection:
projection2[line!] := p2                // LINE <-> POINT x POINT
[holds(l:line) : set[p:point] -> {}]    // set of points on the line

[reflexivity(l:line,p:point) :: p = l.p1 => l.holds :add p]
[symmetry(l:line,p:point) :: p % l.holds
    => line!(l.p2,l.p1).holds :add p  ]
[transitivity(l:line,p:point) ::
    exist(x:point, x % l.holds, p % line!(p2(l),x).holds) ]
```

the last rule is defined as a query so that we can evaluate the result of the logic condition and check that it correspond to our intention. *transitivity(line!(A,B))* will return the set of points p that belong to a line (B,C) such that C belongs to the line (A,B). Once we are satisfied that the condition is correct, we can transform the rule into :

```
[transitivity(l:line,p:point) ::
    exist(x:point, x % holds(l), p % holds(line!(p2(l),x)))
    => holds(l) :add p]
```

6. I/O, Modules and System Interface

6.1 Printing

There are several ways of printing in CLAIRE. Any entity may be printed with the function *print*. When print is called for an object that does not inherit from thing (an object without a name), it calls the method *self_print* of which you can define new restrictions whenever you define new classes. If *self_print* was called on an object x owned by a class toto for which no applicable restriction could be found, it would print <toto>

In the case of bags (sets or lists), strings, symbols or chars, the standard method is *princ*. It formats its argument in a somewhat nicer way than print. For example

```
print("john")      gives "john"
princ("john")      gives john
```

Finally, there also exists a *printf* macro as in C. Its first argument is a string with possible occurrences of the control sequences ~S, ~I and ~A. The macro requires as many arguments as there are tilde forms in the string, and pairs in order of appearance arguments together with tildes. These control sequences do not refer to the type of the corresponding argument but to the way you want it to be printed. The macro will call *print* for each argument associated with a ~S form, *princ* for each associated with a ~A form and will print the result of the evaluation of the argument for each ~I form. A mnemonic is A for alphanumeric, S for standard and I for instruction. Hence the command

```
printf("~S is ~A and here is what we know\n ~I",john,23,show(john) )
```

will be expanded into

```
(print(john), princ(" is "), princ(23),
 princ(" and here is what we know\n"), show(john) )
```

Output may also be directed to a file or another device instead of the screen, using a port. A port is an object bound to a physical device or a file. The syntax for creating a port bound to a file is very similar to that of C. The two methods are *fopen* and *fclose*. Their use is system dependent and may vary depending on which C compiler you are using. However, fopen always requires a second argument : a control string most often formed of one or more of the characters 'w', 'a', 'r': 'w' allows to (over)write the file, 'a' ('a' standing for append) allows to write at the end of the file, if it is already non empty and 'r' allows to read the file. The method fopen returns a port. The method *use_as_output* is meant to select the port on which the output will be written. Following is an example:

```
(let p:port := fopen("agenda-1994","w") in
        ( use_as_output(p), write(agenda), fclose(p) ) )
```

Note that for the sake of rapidity, communications through ports are buffered, so it may happen that the effect of printing instructions is delayed until other printing instructions for this port are given. To avoid problems of synchronization between reading and writing, it is sometimes useful to ensure that the buffer of a given port is empty. This is done by the command *flush(p:port). flush(p)* will perform all printing (or reading) instructions for the port p that are waiting in the associated buffer.

Two ports are created by default when you run CLAIRE : *stdin* and *stdout* . They denote respectively the standard input (the device where the interpreter needs to read) and the standard output (where the system prints the results of the evaluation of the commands). Because CLAIRE is interpreted, errors are printed on the standard output. The actual value of these ports is interface dependent.

It is sometimes also convenient to have access to the text printed directly as a string. Two methods are offered to do this: *print_in_string* and *end_of_string. print_in_string()* starts redirecting all printing statements towards the string being built. *end_of_string()* returns the string formed by all the printing done between these two instructions.

Last, CLAIRE also provides a special port which is used for tracing: *trace_output*(). This port can be set directly or through the trace(_) macro (cf. Appendix C). All trace statements will be directed to this port. A *trace* statement is either obtained implicitly through tracing a method or a rule, or explicitly with the *trace* statement. the statement *trace(n, <string>, <args> ...)* is equivalent to *printf(<string>, <args> ..)* with two differences: the string is printed only if the verbosity level *verbose()* is higher than n and the output port is *trace_output().*

To avoid confusion, the following hierarchy is suggested for verbosity levels:

1 - error: this message is associated with an error situation

2 - warning: this message is a warning which could indicate a problem

3 - note: this message contains useful information

4 - debug: this message contains additional information for debugging purposes

This hierarchy is used for the messages that the CLAIRE system send to the user (which are all implemented with trace).

6.2 Reading

Ports offer the ability to direct the output to several files or devices. The same is true for reading. Ports just need to be opened in reading mode (there must be a 'r' in the control string when fopen is called to create a reading port). The basic function that reads the next character from a port is *getc(p : port). getc(p)* returns the next characters read on p. When there is nothing left to be read in a port, the method returns the special character EOF. As in C, the symmetric method for printing a character on a port also exists: *putc(c : char, p : port)* writes the character c on p.

There are however higher-level primitives for reading . Files can be read one expression at a time : *read(p : port)* reads the next CLAIRE expression on the port *p* or, in a single step, *load(s : string)* reads the file associated to the string s and evaluates it. It returns *true* when no problem occurred while loading the file and *false* otherwise. A variant of this method is the method *sload(s : string)* which does the same thing but prints the expression read and the result of their evaluation. Another variant is the method *oload(s : string)* which does the same thing but substitute an optimized form to each method's body. This may hinder the inspection of the code at the toplevel, but it will increase the efficiency of the interpreter.

Files may contain comments. A comment is anything that follows a ; until the end of the line. When reading, the CLAIRE reader will ignore comments (they will not be read and hence not evaluated). For instance

```
    x :+ 1,    ;;  increments x by 1
```

To increase the compatibility with C++, CLAIRE also recognizes lines that begin with // as comments. These comments are not allowed inside expressions and are passed to C++ through compilation (see Appendix C).

The second type of special instructions are immediate conditionals. An immediate conditional is defined with the same syntax as a regular conditional but with a #if if instead of an if

```
     #if <test> <expression> <else <expression> >opt
```

When the reader finds such an expression, it evaluates the test. If the value is true, then the reader behaves as if it had read the first expression, otherwise it behaves as if it had read the second expression (or nothing if there is no else). This is useful for implementing variants (such as debugging versions). For instance

```
  #if debug printf("the value of x is ~S",x)
```

6.3 Modules

Organizing software into modules is a key aspect of software engineering : modules separate different functionalitites as well as different level of abstractions for a given task. To avoid messy designs and to encourage modular programming, programs can be structured into modules which all have their own identifiers and may hide them to other modules. A *module* is thus a namespace that can be visible or hidden for other modules. CLAIRE supports multiple namespaces, organized into a hierarchy similar to the UNIX file system. The root of the hierarchy is the module *CLAIRE*, which is implicit. A module is defined as a usual CLAIRE object with two important slots: *part_of* which contains the name of the father module, and a slot *uses* which gives the list of all modules that can be used inside the new module. For instance,

```
interface :: module(part_of = library, uses = list(claire))
```

defines *interface* as a new sub-module to the *library* module that uses the module *CLAIRE* (which implies using all the modules). All module names belong to the *CLAIRE* namespace (they are shared) for the sake of simplicity. Note that the slot *uses* may also contain the name of C++ libraries that are used for imported functions (cf. § 6.5).

Identifiers always belong to the namespace in which they are created (*CLAIRE* by default). The instruction *current_module()* returns the module currently opened. To change to a new module, one may use *open(m : module)* and *end(m : module)*. The instruction *open(m)* makes m the current module. Each newly created identifier (symbol) will belong to the module m, until *end(m)* resumes to the original module. For instance, we may define

```
open(interface)
window <: object(...)
end(interface)
```

This creates the identifier *interface/window*. Each identifier needs to be preceded by its module, unless it belongs to the current module or one of its descendent, or unless it is private (cf. visibility rules). We call the short form "window" the unqualified identifier and the long one "interface/window" the qualified identifier.

The visibility rules among name spaces are as follows:

- unqualified identifiers are visible if and only if they belong to a descendent of the current module,
- all qualified identifiers that are *private* are not visible.
- other qualified identifiers are visible everywhere, but the compiler will complain if their module of origin does not belong to the list of allowed modules of the current modules.

Any identifier can be made private when it is defined by prefixing it with *private/*. For instance, we could have written

```
open(interface)
claire/window <: object(...)
private/temporary <: window(...)
end(interface)
```

The declaration *private/temporary* makes "temporary" a private identifier that cannot be accessed outside the module interface (or one of its descendants). The declaration CLAIRE/window makes window an identifier from the CLAIRE module (thus it is visible everywhere), which is allowed since CLAIRE belongs to the list of usable modules for interface.

In practice, there are almost always a set of files that we want to associate with a module, which means that we want to load into the module's namespace. CLAIRE allows an explicit representation of this through the slots *made_of* and *source*. made_of(m) is the list of files (described as strings) that we want to associate with the module and source(m) is the common directory (also described as a string). The benefit are the load/sload methods that provide automatic loading of the module's files into the right namespace and the module compiling features (cf. Appendix C).

6.4 Global Variables and Constants

CLAIRE offers the possibility to define global variables; they are named objects from the following class :

```
[global_variable <: thing(range:type,value:any)]
```

For instance, one can create the following

```
tata :: global_variable(range = integer, value = 12)
```

However, there is a shorthand notation:

```
tata:integer :: 12
```

Variables can be used anywhere, following the visibility rules of their identifiers. Their value can be changed directly with the same syntax as local variables.

```
tata := 12, tata :+ 1, tata :- 10
```

The value that is assigned to a global variable must always belong to its range, with the exception of the unknown value which is allowed. If a variable is re-defined, the new value replaces the old one, with the exception still of unknown, which does not replace the previous value. This is useful for defining flags, which are global_variables that are set from the outside (e.g., by the compiler). One could write, for instance,

```
talk:boolean :: unknown
 (#if talk printf( ....
```

The value of talk may be set to false before running the program to avoid loading the *printf* statements. When the value does not change, it is simpler to just assign a value to an identifier. For instance,

```
toto :: 13
```

binds the value 13 to the identifier *toto*. This is useful to define aliases, especially when we use imported objects from other modules. For instance, if we use *Logic/Algebra/exp*, we may want to define a simpler alias with :

```
exp :: Logic/Algebra/exp
```

6.5 System Integration

Methods are usually defined within CLAIRE. However, it is also possible to define a method through a C function, since most entities in CLAIRE can be shared with C. The C function must accept the method's parameters with the C types that correspond to the CLAIRE types of the parameters and return accordingly a result of the type associated with the range. The ability to exchange entities with the "outside world" was a requirement for CLAIRE and is a key feature.

To understand how C and CLAIRE can share entities, we must introduce the notion of "sort", which is a class of entities that share the same physical representation. There are five sorts in CLAIRE: object, integer, char, imported and any which cover all other entities. Objects are represented as pointers to C structures: to each class we associate a structure with the same name where each slot of the object becomes a field in the structure. Integers share the same representation with C and characters are also represented with integers. Imported objects are "tagged pointers" and are represented physically by this associated pointer. For instance, a CLAIRE string is the association of the tag *string* and the "char*" pointer which is the C representation of the string. Imported objects include strings, floats (where the pointer is of type "double*"), ports (pointer of type "FILE *") and external functions. Last, the sort any contains all other entities (such as symbols or bags) that have no equivalent in C and are, therefore, represented in the same way, with an object identifier with C type "OID" (OID is a system-dependent macro).

The method c_interface(c) (cf. Appendix C) can be used to obtain the C type used for the external representation of entities from the class c.

```
claire> c_interface(float)
eval[1]> "double *"
```

Now that we understand the external representation of entities in CLAIRE, we can define, for instance, the *cos* method for floats. The first part goes in the CLAIRE file and stands as follows.

```
[cos(x:float) : float -> function(cos_for_claire)]
```

We then need to define in the proper C file the C function *cos_for_CLAIRE* as follows.

```
double *cos_for_claire(double *y)
        {double *x;
            x = malloc(size_of(double));
            *x = cos(*y);
            return x;}
```

When the two files are compiled and linked together, the method cos is defined on floats and can be used freely. The linking is either left to the user when a complex integration task is required, or it can be done autoimatically by CLAIRE when a module *m* is compiled. The slot (m) may contain a string such as "XX", which tells CLAIRE that the external functions can be found in a library file XX.lib and that the header file with the proper interface definitions is XX.h.

There is one special case when importing an external function if this external function makes use of CLAIRE memory allocation either directly or through a call back to CLAIRE. In this case, the compiler must be warned to insure proper protection from garbage collection. This is done with the additional argument true in the *function*(...) constructor. Note that this cannot be the case unless the external function make explicit use of CLAIRE's API. Here is a simple example.

```
[mycopy(x:bag) : bag -> function(mycopy,true)]

OID mycopy(OID x)
     {count++; return (copy_bag(x)); }
```

When a method is defined within CLAIRE and compiled later, the compiler produces an equivalent C function that operates on the external representation of the parameters. This has two advantages: on the one hand, the C code generated by the compiler is perfectly readable (thus we can use the compiler as a code generator or modify its output by hand); on the second hand, the compiled methods can be invoked very easily from another C file, making the integration between compiled CLAIRE module and C programs reasonably simple (especially when compared with the LAURE language).

The only catch is the naming convention due to polymorphism and extensibility. The default strategy is to generate the function *m_c* for the method *m* defined on the class *c* (i.e. a method which is a restriction of the property *m* and whose first type in the signature is the class *c*). When this first type *t* is not a class, the class *class!(t)* is used instead. However, this is ambiguous in two cases: either there are already multiples definition of *m* on *c*, or the property *m* is open and further definitions are allowed. In the first case a number is added to the function name; in the second case, the name of the module is added to the function name. Therefore, the preferred strategy is to avoid overloading for methods that are used as interfaces for other programs, or to look at the generated C code otherwise to check the exact name.

For instance, in the previous example with the fib method, the generated C function will simply be (as it will appear in the generated header file) :

```
int fib_integer(int x);
```

Another interesting consequence is that all the library functions on strings can be used within any C program that is linked with the compiled CLAIRE code. Since these functions use the same "char *" type as other string functions in C, we can freely use the following (as they appear in the header files):

```
char * copy_string(char *s);
char * substring_string(char *s, int n1, int n2);
```

The API with CLAIRE is not limited to the use of functions associated with methods. It also includes access to all the objects which are seen as C structures. When a CLAIRE file is compiled, the structure definitions associated with the classes are placed in a header file. This header file allows the C user to manipulate the C pointers obtained from CLAIRE in a very natural way (see Appendix C). The pointers that represent objects can be obtained in two ways: either as a parameter of a function that is invoked from CLAIRE, or through a C identifier when the object is a named object. The compiler generates a C global variable (L_x) for each named object x whose value is the representation of the object[7]. For instance, if John is an object from the class person, the following declaration will be placed into the header file:

```
extern struct person *L_John;
```

The set of primitive classes (symbol, boolean, char, bag) is fixed once for all and trying to add a new one will provoke an error. On the other hand, the set of imported object can be enriched with new classes. More details about the integration between CLAIRE and C code will be given in the Appendix C, where we examine the CLAIRE compiler and its output.

---

[7] As for external functions, special characters (e.g., +, /) are dealt with through a transformation described in the last Appendix.

APPENDIX A: CLAIRE DESCRIPTION

In the following summary of the grammar, we have used the following conventions:

    <a>$^{\text{seq}}$ denotes a (possibly empty) list of <a> separated by commas

    <a>$^{\text{seq+}}$ denotes a non empty list of <a> separated by commas

    <a>$^{\text{opt}}$ denotes <a> or nothing

keywords of CLAIRE are printed boldface. ⟨ and ⟩ are simply used for grouping. │ is used for choices and **|** is used for the CLAIRE character '|'

A1. Lexical Conventions

a. Identifiers in the CLAIRE language

A name expression in the CLAIRE language is called an *identifier*. It is used to designate a named object or a variable inside a CLAIRE expression. Each identifier belongs to a namespace. When it is not specified, the namespace is determined by the current reading environment, the identifier is unqualified. A qualified identifier contains its namespace as a qualification, designed by another identifier (the name of the namespace), followed by a slash '/', itself followed by the unqualified form of the identifier.

An unqualified identifier in CLAIRE is a non-empty sequence of basic characters, beginning with a non-numerical character. A basic character is any character, with the exception of '[', ']', '{', '}', '(', ')', ' '(space), EOL (end of line), '**;**', '**"**', '**"**', '**/**', '**.**', '**@**' and '**:**' that play a special role in the grammar. The first six are used to define expressions. Spaces and EOL are meaningless, but are not allowed inside identifiers (therefore they are separators characters). The characters '**;**', '**'**', '**"**', '**@**', '**/**' and '**:**' are reserved to the CLAIRE system. The point character (.) is special because it is used for slot access in expressions such as *x.age*. An identifier should not start with the character #. Each sequence of characters defines one unique identifier, inside a given namespace.

        < ident> ≡ <unqualified identifier> │ <qualified identifier>
        <qualified identifier> ≡ <identifier>**/**<unqualified identifier>
        <unqualified identifier> ≡ <'a' .. 'Z'><basic character>*

    Implementation note: in CLAIRE 1.0, the length of an unqualified identifier is limited to 50 characters.

b. Symbols

Identifiers are represented in CLAIRE with entities called symbols. Each identifier is represented by a *symbol*. A symbol is defined by a namespace (where the identifier belongs), a name (the sequence of character from the unqualified symbol) and a status. The status of a symbol is either *private* or *export* (the default status). When the status of an identifier is private, the reader will not recognize the qualified form from a module that is not a submodule of that of the identifier. Therefore, the only way to read the identifier is inside its own namespace. When the status of the identifier is export, the qualified form gives access to the designated object, if the sharing declarations of namespaces have been properly set (Section 6.1).

Each symbol may be bound to the object which it designates. The object becomes the *value* of the symbol. The name of the object must be the symbol itself. In addition, the symbol collects another piece of useful information: the module where its value (the named object) is defined. If the symbol is defined locally (through a *private* or *export* definition), this *definition* module is the same as the owner of the symbol itself. If the symbol is shared (if it was defined as an identifier of an above module), this module is a subpart of the owner of the symbol.

c. Characters and Strings

Characters are CLAIRE objects, that can be expressed with the following syntax:

```
<CLAIRE character>  ≡   ' <character> '
```

Implementation note: A character is an ASCII representation of a 8bits integer. The ASCII value for the character 'x' may be obtained directly with #/x. The end-of-file character (ascii value -1) is stored in the global variable EOF.

Strings are objects defined as a sequence of characters. A string expression is a sequence of characters beginning and ending with ' " '. Any character may appear inside the string, but the character ' " '. Should this character be needed inside a string, it must be preceded by the ' \ ' character.

```
< string> ≡   "  ⟨ <character – ' "'  ⟩ * "
```

The empty string, for instance, is expressed by: "". Note that the "line break" character can be either a line break (*new line*) or the special representation '\n'.

d. Integer and Floats

Numbers in CLAIRE can *either* be integers or floating numbers. Only the decimal representation of integers and floats is allowed. The syntax for integer is straightforward:

```
<integer> ≡  ⟨– ⟩opt  <positive integer>
<positive integer> ≡  <'0' .. '9'>+
```

If the integer value is too large, and overflow error is produced. The syntax for floating numbers is also very classical:

```
<float > ≡   <integer>.<positive integer>    |
             <integer>⟨.<positive integer>⟩opt e ⟨+|– ⟩opt <positive integer>
```

Implementation note: in CLAIRE 1.0, integers are coded on 30 bits and floats on 64 bits.

e. Booleans and External Functions

The two boolean values of CLAIRE are **false** and **true**:

```
<boolean> ≡   false │ true
```

External functions may be represented inside the CLAIRE system. An external function is defined with the following syntax:

```
<external_function> ≡    function(<unqualified identifier> ⟨,<boolean>  ⟩opt  )
```

The identifier must be the name of a function defined elsewhere. Therefore, it is an unqualified identifier.

Implementation note: in most implementations of CLAIRE, external functions can only be used for a function call when the CLAIRE program has been compiled and linked to the definition of the external function. However, this is not true, for instance, of a LISP implementation.

f. Spaces, Lines and Comments

Spaces and end_of_lines are not meaningful in CLAIRE. However they play a role of separator:

```
<separator> ≡     │ [  │  ] │ { │ } │  ( │  ) │  : │  " │  ' │ ; │
                 @ │      │ EOL │  /
<basic character> ≡ <character – separator>
```

Comments may be placed after a ';' on any line of text. Whatever is between a ';' and a EOL character is considered as a comment and ignored:

```
<comment> ≡ ;'<character - EOL>* EOL
```

CLAIRE also supports the use of // to define a comment, as in C++, but only between blocks in a file. These comments have a special status since they are passed into the code generated by the compiler.

Named objects (also called things) are also designated entities, since they can be designated by their names. The following convention is used in this syntax description for any class C:

```
<C>   ≡   <x:identifier, where x is the name of a member of class C>
```

This convention will be used for `<class>`, `<property>` and `<array>`.

The set of designated entities is, therefore, defined by:

```
<designated entity> ≡  <thing> | <integer>  | <float> |
                       <boolean> | <external function> |
                       <CLAIRE character> |  <string>
```

A2. Grammar

Here is a summary of the grammar. A program (or the transcript of an interpreted session) is a list of blocks. Blocks are made of definitions delimited by square brackets and of expressions, either called exp, when they need not to be surrounded by parentheses or fragment when they do.

```
<program> ≡  <block>^seq+

<block> ≡     (<fragment>) | <definition> | <call>

<fragment> ≡ 〈 <statement | <conditional> 〉^seq

<statement> ≡ for <var def> in <exp> <statement> |
              while <exp> <statement> |
              until <exp> <statement> |
              let 〈 <var def> := <comp-exp> 〉^seq+ in <statement> |
              when <var def> := <comp-exp>  in <statement> |
              case <exp> ( 〈 <type> <statement> 〉^seq+ ) |
              try <statement> catch <type> <statement> |
              <comp-exp> | <update>

<definition> ≡    <ident> :: <exp> |
                  <var def> :: <exp> |
                  [ <ident>(〈 <var>:<type with var> 〉^seq) : <range>
                      〈-> | => 〉 <fragment> ] |
                  [ <ident>[<var def>] : <type> -> <fragment> ] |
                  [ <ident>〈[<var>^seq+]〉^opt <: <class>
                    〈 (〈 <property> : <type> 〈 = <exp> 〉^opt 〉^seq+)  〉^opt ] |
                  [ <ident>(<var def> 〈, <var def>〉^opt) ::
                      <assertion> 〈=> <fragment>〉^opt ]

<conditional> ≡ if <exp> <statement> 〈 else <conditional> | <statement> 〉^opt

<update> ≡ 〈<var> | <property>(<exp>) | <array>[<exp>^seq+] 〉
              :<operation> <comp-exp>
```

The basic building block for a CLAIRE instruction is an expression. The grammar considers different kinds of expressions :

```
<comp-exp> ≡  <exp> | <comp-exp> <operation> <comp-exp>
```

$$<exp> \equiv <exp>\ \textbf{as}\ <class>\ |\ <ident>\ |\ <set\ exp>\ |\ <fragment>\ |\ <call>\ |$$
$$<array>\textbf{[}<exp>\textbf{]}\ |\ <class>\textbf{(<}\ <property>\ \textbf{=}\ <exp>\ \textbf{>}^{seq}\ \textbf{)}\ |\ break\textbf{(}<exp>\textbf{)}$$
$$lambda\textbf{[}\langle\ <var>\textbf{:}<type\ with\ var>\ \rangle^{seq}\textbf{,}\ <exp>\textbf{]}$$

$$<set\ exp> \equiv\ \ \textbf{set(}<const>^{seq^+}\textbf{)}\ \ \textbf{list(}<const>^{seq^+}\textbf{)}\ |$$
$$\textbf{\{}\ <statement>^{seq^+}\ \textbf{\}}\ |\ <type>\ |$$
$$\langle\textbf{list}\rangle^{opt}\textbf{\{}\ <var>\ \textbf{in}\ <exp>\ \textbf{|}\ <statement>\ \textbf{\}}\ |$$
$$\langle\textbf{list}\rangle^{opt}\textbf{\{}\ <statement>\ \textbf{|}\ <var>\ \textbf{in}\ <exp>\ \textbf{\}}\ |$$
$$\textbf{forall(}<var>\ \textbf{in}\ <exp>\ \textbf{|}\ <statement>\ \textbf{)}\ \ |$$
$$\textbf{exists(}<var>\ \textbf{in}\ <exp>\ \textbf{|}\ <statement>\ \textbf{)}$$

$$<call> \equiv <property>\ \langle\textbf{@}\ <type>\ \rangle^{opt}\textbf{(}<exp>^{seq}\textbf{)}\ \textbf{|}\ <exp>\textbf{.}<property>$$

The typing system is the following

$$<var\ def> \equiv <var>\textbf{:}<type>$$
$$<var> \equiv <ident>$$

$$<type> \equiv <class>\ |\ <class>\textbf{[<}\ <var>\ \textbf{:}\ <type>\ \textbf{>}^{seq^+}\textbf{]}\ |$$
$$\textbf{set[}<type>\textbf{]}\ |\ \textbf{list[}<type>\textbf{]}\ |\ \textbf{subtype[}<type>\textbf{]}\ |\ \textbf{\{}<const>^{seq^+}\ \textbf{\}}\ |$$
$$\textbf{tuple(}<type>^{seq^+}\ \textbf{)}\ |\ \textbf{(}<comp\text{-}type>\textbf{)}$$

$$<comp\text{-}type> \equiv <type>\ \textbf{U}\ <type>\ |\ <type>\ \textbf{\^{}}\ <type>\ |\ \textbf{|}<const>\ \textbf{..}\ <const>$$

$$<const> \equiv <integer>\ |\ <float>\ |\ <named\ object>\ |\ <string>\ |\ <char>\ |\ <symbol>$$

Typing also includes second-order typing which has special syntactical conventions :

$$<type\ with\ var> \equiv <var>\ |\ <type>\ |\ \textbf{\{}<var>\textbf{\}}\ |$$
$$\textbf{tuple(}<type\ with\ var>^{seq^+}\ \textbf{)}\ |$$
$$<class>\textbf{[}\langle\ <var>\textbf{:}<type\ with\ var>\ \textbf{|}\ <var>\textbf{=}\langle<var>\textbf{|}<const>\rangle\ \rangle^{seq^+}\textbf{]}$$

$$<range> \equiv <type>\ |\ \textbf{type[}<exp>\textbf{]}$$

The logic language used for rules (to describe the logic condition to which a rule is attached) describes the condition as a logical assertion, itself made of logical expressions.

$$<assertion> \equiv <expression>\ <comp>\ <expression>\ |$$
$$\textbf{exists(}<var>\ \textbf{|}\ <var\ def>\textbf{,}\ <assertion>\textbf{)}\ |$$
$$\textbf{if (}<expression><comp><expression>\textbf{)}\ <assertion>$$
$$\textbf{else}\ <assertion>\ |$$
$$<assertion>\ \textbf{\&}\ <assertion>\ |$$
$$<assertion>\ \textbf{|}\ <assertion>$$

$$<expression> \equiv <var>\ |\ <designated\ entity>\ |\ <property>\textbf{(}<expression>\textbf{)}\ |$$
$$<array>\textbf{[}<expression>\textbf{]}\ |$$
$$\langle\textbf{list}\rangle^{opt}\textbf{\{}<var>\ \textbf{in}\ <expression>\ \textbf{|}\ assertion\textbf{\}}\ |$$
$$<expression>\ <operation>\ <expression>$$

APPENDIX B: CLAIRE'S API

This section contains the list of all methods and (visible) slots in CLAIRE. For each method we give the signature of the restrictions, the modules where they are defined and a brief description of their use.

^	System	method

```
x:integer ^ y:integer → integer
x:float ^ y:float → float
x:list ^ y:integer → list
x:set ^ y:set → set
```

$(x ^\wedge y)$ returns x^y when x and y are numbers. If x is an integer, then y must be a positive integer, otherwise an error is raised.

$(l ^\wedge y)$ skips the y first members of the list l. If the integer y is bigger than the length of the list l, the result is the empty list, otherwise it is the sublist starting at the y+1 position in l (up to the end).

$(s_1 ^\wedge s_2)$ returns the *intersection* of the two sets s_1 and s_2 that is the set of entities that belong to both s_1 and s_2.

Other internal restrictions of the property ^ exist, where ^ denotes the intersection (it is used for the type lattice)

^2	System	method

```
^2(x:integer) → integer
```

$^\wedge 2(x)$ returns 2^x

%	Class	method

```
x:any % y:class → boolean
x:any % y:any → boolean
```

$(x \% y)$ returns $(x \in y)$ for any entity x and any abstact set y. An abstract set is an object that represents a set, which is a type or a list.

*	System	method

```
x:integer * y:integer → integer
x:float * y:float → float
```

$(x * y)$ returns $x \times y$ when x and y are numbers. If x is an integer, then y must also be an integer, otherwise an error is raised (explicit conversion is supported with float!).

The operation * defines a commutative monoid, with associated divisibility operator divide? and associated division /.

/	System	method

```
x:integer / y:integer → integer
x:float / y:float → float
```

(x / y) returns x / y when x and y are numbers. If x is an integer, then y must also be an integer, otherwise an error is raised (explicit conversion is supported with float!).

+	System	method

```
x:integer + y:integer → integer
x:float + y:float → float
```

$(x + y)$ returns $x + y$ when x and y are numbers. If x is an integer, then y must be a an integer, otherwise an error is raised (explicit conversion is supported with float!).

The operation + defines a group structure, with associated inverse -.

-	System	method

```
x:integer - y:integer → integer
x:float - y:float → float
-(x:integer) → integer
-(x:float) → float
```

(x - y) returns x + y when x and y are numbers. -(x) return the opposite of x.

.   **System**   **method**

```
x:list . y:list → list
x:string . y:string → string
x:symbol . y:(string U symbol) → symbol
```

(x . y) returns the *concatenation* of x and y (. represents the append operation). Concatenation is an associative operation that applies to strings, lists and symbols. When two symbols are concatenated, the resulting symbol belongs to the namespace (module) of the first symbol, thus the second symbol is simply used as a string. By extension, a symbol can be concatenated directly with a string.

..   **System**   **method**

```
x:integer .. y:integer → interval
x:float .. y:float → interval
x:char .. y:char → interval
x:string .. y:string → interval
x:type .. y:type → interval
```

(x .. y) returns the interval $\{z \mid x \leq z \leq y\}$. Intervals are supported for integers, floats, characters, strings and types, using the system-defined order <= (see method <= below). The result is an object from the class *interval*, which is a type.

=, !=   **System**   **method**

```
x:any = y:any → boolean
x:any != y:any → boolean
```

(x = y) returns true if x is equal to y and nil otherwise. Equality is defined in Section 2: equality is defined as identity for all entities except strings, lists and sets. For lists, sets and strings, equality is defined recursively as follows: x and y are equal if they are of same size *n* and if x[i] is equal to y[i] for all *i* in (1 .. *n*).

(x != y) is simply the negation of (x = y).

=type?   **System**   **method**

```
=type?(x:all, y:all) → boolean
```

returns true if x and y denote the same type. For example =type?(boolean, {true, false}) returns true because defined(boolean) was declared after the two instances true and false were created, so the system knows that no other instances of boolean may ever be created in the future. This equality is stronger than set equality in the sense that the system answers true if it knows that the answer will hold everafter.

<=, >=, <, >   **Class**   **method**

```
x:integer <= y:integer → boolean
x:float <= y:float → boolean
x:char <= y:char → boolean
x:string <= y:string → boolean
x:type <= y:type → boolean
x:X < y:X → boolean  for X = integer, float, char and string
x:X > y:X → boolean  for X = integer, float, char and string
x:X >= y:X → boolean for X = integer, float, char and string
```

The basic order property is <=. It is defined on integers and floats with the obvious meaning. On characters, it is the ASCII order, and on strings it is the lexicographic order induced by the ASCII order on characters. The order on types is the inclusion: ((x <= y) if all members of type x are necessarily members of type y).

(x < y), (x > y) and (x >= y) are only defined for numbers, char and strings with the usual meaning.

<<	**System**	**method**

```
l:list << n:integer → list
```

(l << n) left-shifts the list l by n units, which means that the n first members of the list are removed. This is a method with a side-effect since the returned value is the original list, which has been modified.

@	**Class**	**method**

```
p:property @ t:type → entity
t:type @ p:parameter → type
```

(p @ t) returns the restriction of p that applies to arguments of type t. When no restrictions applies, the value nil is returned. If more than one restriction applies, the value unknown is returned. Notice that the form p@t (without blank spaces) is used to print the restriction and also in the control structure `<property>@<class>(...)`.

abstract	**System**	**method**

```
abstract(x:class) → void
```

```
abstract(p:property) → void
```

abstract(c) forbids the class c to have any instance. abstract(p) tells CLAIRE that the property p should be kept open to future re-definition. This makes p an abstract parameter of the program. (cf. § 4.5).

add	**System, Class**	**method**

```
add(s:set,x:any)  → set              System
add(l:list,x:any) → list             System
add(p:property,x:object,y:any) → any    Class
```

add(s,x) adds x to the set s. The returned value is the set s © {x}. This method may modify the set s but not necessarily. When x is a list, add(l,x) inserts x at the end of l. The returned value is also the list obtained by appending (x) to l, and l may be modified as a result but not necessarily.

add(p,x,y) is equivalent to *p(x) :add y* (This form is interesting when one wants to write such an expression for a variable p)

add*	**System**	**method**

```
add*(l1:list, l2:list) → list
```

add*(l1,l2) returns the concatenated list l1 . l2, but it is destructive: it uses l1 as the data structure on which to perform the concatenation. Hence, the original l1 is no longer available after the method add* has been called

apply	**System**	**method**

```
apply(p:property, l:list) → any
```

```
apply(f:external_function, ls:list[class], lx:list) → any
```

```
apply(la:lambda, lx:list) → any
```

```
apply(m:method, lx:list) → any
```

apply(p,l) is equivalent to a function call where the selector is p and the argument list is l. For instance, apply(+,list(1,2)) = (1 + 2) = call(+,1,2).

apply(f,ls,l) applies the function f to the argument list l, where ls is the list of sort of the arguments and the result (i.e. length(ls) = length(l) + 1). For instance, if f is the external function that defines + @ integer, apply(f,list(integer,integer,integer),list(1,2)) = 1 + 2.

apply(la,lx) applies the lambda expression to the argument list. *apply(m,lx)* applies the method to the argument list.

but	**System**	**method**

but(s:any,x:any) → **any**

Returns the set of members of s that are different from x.

car, cdr	**System**	**method**

car(l:list) → **type[member(l)]**
cdr(l:list) → **type[l]**

These two classical LISP methods return the head of the list , e.g. l[1] (for car) and its tail, e.g. the list l starting at its second element (for cdr).

call	**Class**	**method**

call(p:property, l:listargs) → **any**

call(p,x_1,x_2 ,...,x_n) is equivalent to apply(p,list(x_1,x_2 ,...,x_n)).

char!	**System**	**method**

char!(n:integer) → **char**

char!(n) returns the character which ASCII code is n.

class!	**System**	**method**

class!(x:any) → **class**

class!(x) returns the intersection of all classes y such that x <= y (Such an intersection always exists since classes are organized in a lattice). Hence, if c is a class class!(c)=c.

close	**CLAIRE**	**method**

close(m:module) → **module**   **Class**
close(c:class) → **class**   **Language**
close(e:exception) → **any**   **System**
close(v:global_variable) → **global_variable**   **System**

The method close is called each time an object is created. It is executed and returns the created object. It can sometimes be very helpful to define new restrictions, they will be automatically called when an instance is created. Exceptions are a special case: raising an exception is done internally by creating an instance of exception. The method close is responsible for looking for the innermost handler, etc.

cons	**System**	**method**

cons(x:any, l:list) → **list**

This traditional method appends x at the beginning of l and returns the constructed list.

contradiction!()	**System**	**method**

contradiction!() → **void**

This method creates a contradiction, which is an instance of the class *contradiction*. It is equivalent to *contradiction*() but is more efficient and should be preferred.

copy	**System**	**method**

copy(x:object) → **object**
copy(s:bag) → **bag**
copy(s:string) → **string**

copy(x) returns a duplicate of the object x. It is not recursive : the slots of the copied object are shared with that of the original one. Similarly, the copy of a bag (a set or a list) returns a fresh set or list with the same elements and the copy of a string is ... a copy of the string.

current_module	**Class**	**method**

current_module() → **module**

current_module() returns the module which is currently open.

defined	Class	method

```
defined(c:class) → void
```

defined(c) forbids the user to create any new instance of the class c.

delete	Class, System	method

```
delete(p:property, x:object, y:any) → any          Class
delete(s:bag, x:any) → bag                          System
```

delete(s,x) returns s if x is not in s and the list (resp. set) s without x otherwise. *delete(p,x,y)* is equivalent to *p(x) :delete y*. This is a destructive method in the sense that it modifies its input argument unless the result is nil (there is only one empty list). The proper way to use delete, therefore, is either destructive (*l :delete x*) or non-destructive (*delete(copy(l),x)*).

description	Logic	array

```
description[p:property] → Logic/relational_description
```

The description associated with a property is an object that tells CLAIRE how the property should be understood inside a logical assertion. `relational_description` is a class with the following instances: `bijection` (invertible one-to-one function), `binary_operation` (operation with two arguments), `comparison` (binary_operation that returns a boolean), `monoid` (associative & commutative operation), `group_operation` (monoid with an inverse function) or `mapping` (operation with two inverse functions). The most common case is using *description* to define a property as a comparison (in which case no further work is required). For more advanced relational descriptions, it may also be necessary to update the following relations :

```
Logic/function_inverse[p:property] → property       ; e.g.   +  →  -
Logic/ternary_inverse[p:property] → property        ; e.g.   *  →  /
Logic/comparison_inverse[p:property] → property     ; e.g.   *  →  factor?
Logic/projection1[p:property] → property            ; e.g.   cons  →  car
Logic/projection2[p:property] → property            ; e.g.   cons  →  cdr
```

If description[p] = `bijection`, you are expected to define inverse(p). If description[p] = `monoid`, you are expected to define ternary_inverse[p] and comparison_inverse[p]. If description[p] = `group_operation`, you are expected to define ternary_inverse[p] and function_inverse[p]. If description[p] = `mapping`, you are expected to define projection1[p] and projection2[p].

The semantic of these relations is defined by the following re-writing rules

$[\,x + y = z \Leftrightarrow x = z - y\,]$ if `function_inverse[+] = -`

$[\,x * y = z \Leftrightarrow f?(z,y) \wedge x = z / y\,]$ if `ternary_inverse[*] = /` and `comparison_inverse[*] = f?`

$[\,x \otimes y = z \Leftrightarrow x = p1(z) \wedge y = p2(z)\,]$ if `projection1[⊗] = p1` and `projection2[⊗] = p2`

In addition, CLAIRE also uses the following array:

```
Logic/compatible[p:property] -> set[property].   ; e.g.  + → {=,!=,<,>}
```

compatible[op] is the set of comparison that are compatible with the operation op. For instance, < is compatible with + since $x < y \Rightarrow x + z < y + z$.

difference	System	method

```
difference(s:set, t:set) → set
```

difference(s,t) returns the difference set s - t, that is the set of all elements of s which are not elements of t.

divide?	System	method

```
divide?(x:integer, y:integer) →       boolean
```

divide?(x,y) returns true if y is a multiple of x.

domain	Class	method

```
domain(r:restriction) → list
domain(r:relation) → any
```

A restriction is either a slot or a method. If r is a slot, domain(r) is the class on which r is defined. If r is a method, domain(r) is the list formed by the types of the parameters required by the method. For a relation r, domain(r) is the type on which r is defined.

end_of_string	**System**	**method**

```
end_of_string() → string
```

end_of_string() returns the string containing everything that has been printed since the last call to print_in_string().

exception!	**System**	**method**

```
exception!() → exception
```

exception!() returns the last exception that was raised.

exit	**System**	**method**

```
exit(n:integer) → void
quit() → void
```

exit(n) stops CLAIRE running and returns to the hosting system the value n. What can happen next is platform-dependent.

extensible	**System**	**method**

```
extensible(p:property) → void
```

defines p as an extensible property. This is used by the compiler to preserve the ability to add new restrictions to p in the future that would change its semantics on existing classes. By default, a property is extensible until it is compiled. A corollary is that function calls that use extensible properties are compiled using late binding.

factor?	**System**	**method**

```
factor?(x:integer, y:integer) →     boolean
```

factor?(x,y) returns true if x is a multiple of y.

finite?	**System**	**method**

```
finite?(t:type) → boolean
```

finite?(t) returns true if the type t represents a finite set. Set iteration (with the for loop) can only be done over finite sets

float!	**System**	**method**

```
float!(x:integer) → float
```

transforms an integer into a float.

flush	**System**	**method**

```
flush(p:port) → void
```

Communications with ports are buffered, so it can happen that some messages wait in a queue for others to come, before being actually sent to their destination port. *flush(p)* for input and output ports and empties the buffer associated with p, by physically sending the print messages to their destination.

fopen, fclose	**System**	**method**

```
fopen(s1:string,s2:string) → port
fclose(p:port) → any
```

fopen returns a port that is handle on the file or external device associated with it. The first string argument is the name of the file, the second is a combination of several control characters, among which 'r' allows reading the file, 'w' (over)writing the file and 'a' appending what will be write at the end of the file. Other possibilities may be offered, depending on the underlying possibilities. Such other possibilities are platform-dependent.

format	**System**	**method**

```
format(string,list) → any
```

This method does the same thing as printf, except that there are always two arguments, thus the arguments must be replaced by an explicit list.

formula	**Class, System**	**slot**

```
formula(m:method) → lambda              Class
formula(d:demon) → lambda               System
```

formula gives the formula associated with the method/demon.

funcall	**System**	**slot**

```
funcall(m:method, x:any) → any
```

```
funcall(m:method, x:any, y:any) → any
```

Applies a method or a lambda to one or two arguments.

gc	**System**	**method**

```
gc() → any
```

gc() forces a garbage collection to take place

gensym	**System**	**method**

```
gensym() → symbol
```

gensym() generates randomly a new symbol.

get	**Class, System**	**method**

```
get(p:property + slot, x:object) → any      Class
get(a:array, x:any) → integer               Class
get(s:string, c:char) → integer             System
get(l:list, x:any) → integer                System
get(m:module) → integer                     System
```

get(p,x) is equivalent to *p(x)*, but without any verification on *unknown*.   So does *get(a,x)* for an array.*get(s,x)* returns i such that s[i]=x (if no such i exists, 0 is returned). So does *get(l,x)* for a list. *get(m)* is equivalent for a module m to *(load(m), open(m))*

getc	**System**	**method**

```
getc(p:port) → char
```

getc(p) returns the next character read on port p.

getenv	**System**	**method**

```
getenv(s:string) → string
```

getenv(s) returns the value of the environment variable s if it exists (an error occurs otherwise since an attempt is made to create a string from the NULL value that is returned by the environment).

hash	**System**	**method**

```
hash(l:list,x:any) → integer
```

hash(l,x) returns an integer between 1 and length(l) that is obtained through generic hashing. To obtain the best dispersion, one may use a list of size 2^i-3. This function can be used to implement hash tables in CLAIRE; it is the basis of the array implementation.

Id	**System**	**method**

```
id(x:any) → type[x]
```

Id(x) returns x. Id would be useless (by definition) but it has a special behavior when compiled. The argument is evaluated before being compiled. Thus Id(v) may be used to capture the value of the variable v in the compiling environment.

index	**Class**	**method**

```
index(s1:string, s2:string, b:boolean) → integer    System
```

index(s1,s2,b) returns i if s2 is a subsequence of s1, starting at s1's i[th] character. The boolean b is there to allow case-sensitiveness or not (identify 'a' and 'A' or not).In case s2 cannot be identified with any subsequence of s1, the returned value is 0.

inherit?	**Class**	**method**

```
inherit?(c1:class, c2:class) → boolean
```

inherit?(c1,c2) returns (c2 % ancestors(c1))

instances	**Class**	**slot**

```
instances(c:class) → type[set[c]]
```

returns the set of all instances of c, created up to now (if c has not been declared ephemeral).

integer!	**System**	**method**

```
integer!(s:string) → integer
integer!(f:float) → integer
integer!(c:char) → integer
integer!(l:set[(0 .. 29)]) → integer
```

integer!(s) returns the integer denoted by the string s if s is a string formed by a sign and a succession of digits, *integer!(f)* returns the lower integer approximation of f, *integer!(c)* returns the ASCII code of c and *integer!(l)* returns the integer represented by the bitvector l, i.e. the sum of all 2^i for i in l.

inverse	**Class**	**slot**

```
inverse(r:relation) → relation
```

inverse(r) contains the inverse relation of r. If the range of r inherits from bag then r is considered multi-valued by default (cf. Section 4.5). If r and its inverse are mono-valued then if r(x) = y then inverse(r)(y) = x. If they are multi-valued, then inverse(r)(y) returns the set (resp. list) of all x such that (y % r(x)).

isa	**Class**	**slot**

```
isa(x:object) → class
```

returns the class of which x is an instance.

kill, kill!	**Reader, System**	**method**

```
kill(x:thing) → any              Reader
kill(x:class) → any              Reader
kill!(x:any) → any               System
```

kill is used to remove an object from the database of the language. *kill* does it properly, removing the object from all the relation network but without deallocating. *kill!* is more brutal and deallocates without any checking.

known?	**Class**	**method**

```
known?(p:property, x:object) → boolean
known?(x:any) → boolean
```

known?(p,x) is equivalent to *get(p,x) != unknown* . The general method known? simply returns true whenever the object exists in the database.

last	**System**	**method**

```
last(l:list) → type[member(l)]
```

last(l) returns *l[length(l)]*

length	**System**	**method**

```
length(l:list) → integer
length(l:string) → integer
```

   returns the length of a list or a string. The length of a list is not its *size* !

list!	System	method

```
list!(s:set) → type[list[member(s)]]
```

list!(s) transforms s into a list. (The order of the elements in the list can be anything)

load, sload, oload	Reader	method

```
load(s:string) → any
sload(s:string) → any
oload(s:string) → any
load(m:module) → any
sload(m:module) → any
oload(m:module) → any
```

   These methods load a file (or the files associated to a module). The difference between them is that *load(s)* reads and evaluates all the instructions found in the file named s, whereas *sload(s)* reads, prints, evaluates and prints the results of the evaluation of all the instructions found in the file named s. *oload(s)* is similar to *load(s)* but also optimizes the methods that are newly defined by substituting an optimized version of the lambda abstraction.

made_of	Class	slot

```
made_of(m:module) → list[string]
```

made_of(m) contains the list of files that contain the code of the module.

make_list	System	method

```
make_list(n:integer,x:any) → type[list[x]]
```

returns a list of length n filled with x (e.g., make_list(3,0) = list(0,0,0)).

make_string	System	method

```
make_string(i:integer, c:char) → string
make_string(s:symbol) → string
```

make_string(i,c) returns a string of length i filled with the character c.

   make_string(s) returns a string denoting the same identifier. If s is given in the qualified form (module/identifer), than the result will contain the name of the module ("module/identifier")

mem	System	method

```
mem() → list[integer]
```

mem returns a list of 4 integers (a,b,c,d) where

   a is the number of cells used by chunks (objects and lists of size > 5)
   b is the number of cells used by small objects and lists
   c is the number of cells used by imported objects
   d is the number of cells used by symbols.

The method stat() pretty prints this information.

member	System	method

```
member(x:any) → type
```

member(x) returns the type of all instances of type x, assuming that x is a CLAIRE type which contains objects y such that other objects z can belong to. If this is the case, member(x) is a valid type for all such z, otherwise the returned value is the empty set. For instance, if x is list[integer], all instances of x are lists that contain integers, and all members of these lists are integers. Therefore, member(list[integer]) is integer.

methods	System	method

```
methods(d:class,r:class) → set[method]
```

methods(d,r) returns the set of methods with a range included in r and a domain which is a tuple which first component is included in d.

min / max	Class	method

```
min(m:method[domain:tuple(X,X), range:boolean],
    l:set[X] U list[X]) → type[X]
min(x:integer,y:integer) →  integer
max(x:integer,y:integer) →  integer
```

given an order function (m(x,y) returns true if x <= y) and a bag, this function returns the minimum of the bag, according to this order. min/max on integer returns the smallest/largest of two integers.

mod	System	method

```
mod(x:integer, y:integer) → integer
```

mod(x,y) is the rest of the Euclidean division of x by y.

module!	Class, System, Optimize	method

```
module!(r:restriction) → module         Class
module!(n:integer) → module             System
module!(s:symbol) → module              System
module!(x:thing) → module               Optimize
```

module! returns the module where the identifier s was created.

new	Class	method

```
new(c:class) → any
new(c:class, s:symbol) → thing
```

new is the generic instantiation method. *new(c)* creates an object of class c (It is equivalent to c()). *new(c,s)* creates an object of class c with name s.

not	System	method

```
not(x:any) → boolean
```

not(x) returns false for all x except false, nil, the empty set and the empty list.

nth, nth=, nth+, nth-	Class, System	method

```
nth(a:array, x:any) → any               Class
nth(x:integer, i:integer) → boolean     System
nth(l:bag, i:integer) → any             System
nth(s:string, i:integer) → char         System
nth=(a:array, x:any, y:any) → any       Class
nth=(l:list, i:integer, x:any) → any    System
nth=(s:string, i:integer, x:char) → char System
nth+(l:list, i:integer, x:any) → bag    System
nth-(l:list, i:integer) → bag           System
```

nth is used for accessing elements of structured data: nth(l,i) is the ith element of the bag l, nth(s,i) is the ith character of the string s. For arrays, nth(a,x) is equivalent to a[x], even when x is not an integer. Finally, nth also deals with the bitvector representation of integers: nth(x,i) returns true if the ith digit of x in base 2 is 1.

nth= is used for changing an element at a certain place to a certain value. In all the restrictions *nth=(s,i,x)* means: change the ith value of s to x.

There exists two other ways of modifying the values in such data structures: nth+ and nth-. nth+ uses the same syntax as nth= : *nth+(l,i,x)* returns a list (that may be l) where x has been inserted in the ith position. By extension, *i* may be *length(l) + 1*, in which case *x* is inserted at the end of *l*.

nth- is used for removing an element. *nth-(s,i)* returns a value that differs from s only in that the ith place has been erased.

occurrence	Language	method

```
occurrence(exp:any, x:variable) → integer
```

returns the number of times when the variable x appears in exp

open	Class	method

```
open(m:module) → any
```

open(m) enters the namespace m. Each identifier that will be created between the call open(m) and end(m) will be created by default in the namespace m.

owner	Class	method

```
owner(x:any) → class
```

owner(x) returns the class from which the object is an instance. It x is an object, then *owner(x) = isa(x)* = the unique class *c* such that *x % instances(c)*.

pair, pair_1, pair_2	Class	method

```
pair(x:any,y:any) → tuple(any,any)
pair_1(tuple(any,any)) → any
pair_2(tuple(any,any)) → any
```

A simpler way to manipulate pairs inside logic rules:

pair(x,y) = list(x,y), pair_1(l) = l[1], pair_2[l] = l[2].

parts, part_of,	Class	slot

```
parts(m:module) → list
part_of(m:module) → module
```

part_of(m) contains the module to which m belongs. parts is the inverse of part_of : parts(m) is the set of submodules of m (in the module hierarchy).

pretty_print	Language	method

```
pretty_print(x:any) → void
```

performs the pretty_printing of x. For example, you can pretty print CLAIRE code: if <inst> is a CLAIRE instruction pretty_print(`<inst>) will print it nicely idented (the backquote here is to prevent the instruction from begin evaluated).

princ, print	System	method

```
princ(x:integer) → void
princ(x:string)  → void
princ(x:char)    → void
princ(x:symbol)  → void
princ(x:list)    → void
print(x:any)     → void
```

print(x) prints the entity x (x can be anything). princ(x:integer) is equivalent to print(x). If x is a string / char / symbol/ bag, print(x) prints x without the " / ' / '/ separator.

print_in_string	System	method

```
print_in_string() → void
```

print-in-string() opens a new output port that will be stored as a string. The user is given access to the string at the end of the transcription, when the call to *end_of_string()* returns this string.

propagate	System	method

```
propagate(p:relation, x:object, y:any) → void
```

propagate(r,x,y) triggers the rules that would be triggered by the update p(x) := y (resp. p[x] := y). The propagation only occurs if p(x) = y.

put	Class	method

```
put(p:property, x:object, y:any) → any
put(a:array, x:object, y:any) → any
put(s:slot, x:object, y:any) → any
put(s:symbol,x:any) → any
```

put(p,x,y) is equivalent to *p(x) := y* but does not trigger the rules associated to r or the inverse of r. Besides, this operation is performed without any type-checking. The method *put* is often used in conjunction with *propagate*. *put(s,x)* binds the symbol s to the object x.

putc	System	method

```
putc(c:char, p:port) → void
```

putc(c,p) sends c to the output port p.

random	System	method

```
random(n:integer) → integer
```

random(n) returns an integer in (0 .. n-1), supposedly with uniform probability.

range	Class, System, Language	method

```
range(r:restriction) → any                          Class
range(r:relation) → any                             Class
range(v:global_variable) → any                      System
range(v:Variable) → any                             Language
```

For a relation or a restriction r, *range(r)* returns the allowed type for the values taken by r over its domain. For a variable v, *range(v)* is the allowed type for the value of v.

read	Class, Reader	method

```
read(p:property, x:object) → any                    Class
read(p:port) → any                                  Reader
read(s:string) → any                                Reader
```

read(p,x) is strictly equivalent to p(x): it reads the value and raises an exception if it is unknown. *read(p)* and *read(s)* both read an expression from the input port *p* or the string *s*.

read_in_string	Reader	method

```
read_in_string(r:meta_reader, s:string) → void
```

considers s as the new stream of input. meta_reader is the class of the object *reader*.

release	System	method

```
release() → any
```

returns a release number of your CLAIRE system (<version>.<release>).

restrictions	Class	method

```
restrictions(p:property) → list[restriction]
```

returns the list of all restrictions of the property. A property is something a priori defined for all entities. A restriction is an actual definition of this property for a given class (or type).

self_print	Class	method

```
self_print(x:any) → any
```

this is the standard method for printing unnamed objects (objects that are not in thing). It is called by default by printf on objects.

set!	System	method

```
set!(s:abstract_set) → set
set!(x:integer) → set[(0 .. 29)]
```

set!(s) returns an enumeration of the abstract set s. The result is, by definition, a set that contains exactly the members of s. An error occur if s is not finite, which can be tested with finite?(x).

set!(x) returns a set that contains all integers i such that $(x / 2^i)$ mod 2 = 1. This method considers x as the bitvector representation of a subset of (0 .. 29). The inverse is integer!.

shell	**System**	**method**

```
shell(s:string) → any
```

Passes the command s to the operating system (the shell).

show	**Reader**	**method**

```
show(x:any) → any
```

The method show prints all the information it can possibly find about the object it has been called on: the value of all its slots are displayed. This method is called by the debugger.

shrink	**Reader**	**method**

```
shrink(x:list,n:integer) → list
shrink(x:string,n:integer) → string
```

The method shrink truncates the list or the string so that its length becomes n. This is a true side-effect and the value returned is always the same as the input. As a consequence, shrink(l,0) returns an empty list that is different from the canonical empty list nil.

size	**System**	**method**

```
size(l:bag) → integer
size(x:any) → integer
```

size(l) gives the number of elements in l. If x is an abstract set (a type, a class, ...) then size(x) denotes the number of elements of type x. If the set is infinite, an exception will be raised. Note that the size of a list is not its length because of possible duplicates.

slots	**Class**	**method**

```
slots(c:class) → any
```

slots(c) returns the list of all slots that c may have

sort	**Class**	**method**

```
sort(m:method, l:list) → type[l]        Class
```

The method sort has two arguments: an order method m such that m(x,y) = true if x <= y and a list of objects to be sorted in ascending order (according to m). The method returns the sorted list. The method is usually designated using @ as in sort(< @ integer, list(1,2,8,3,4,3)).

sqrt	**System**	**method**

```
sqrt(x:float) → float
```

returns the square root of x. Returns an irrelevant value when x is strictly negative.

stat	**Class**	**method**

```
stat() → void
```

stat() pretty prints the result given by mem(): it prints the memory situation of the CLAIRE system: the number of used cells and the number of remaining cells for each type of cell (chunk, small object, imported object, symbol)

store	**System**	**method**

```
store(r1: relation, r2:relation ...) → void
store(l:list,n:integer,v:any,b:boolean) → void
```

store(r1,r2,...) declares the relations (properties or arrays) as defeasible (using the world mechanism). store(l,n,v,b) is equivalent to l[n] := v but also stores this update in the current world if b is true.

string!	System	method

```
string!(s:symbol) → string
string!(n:integer) → string
```

string! converts a symbol or an integer into a string, for example string!(toto) returns "toto" and string!(12) returns "12". Unlike make_string, it returns the unqualified form (string!(Francois/agenda) = "agenda", whereas make_string(Francois/agenda) = "Francois/agenda").

substitution	Language	method

```
substitution(exp:any, v:Variable, f:any) → any
```

substitution(exp,v,f) returns exp where any occurrence of the free variable v is substituted by f. Hence, if occurrences(exp,v) = 0 then substitution(exp,v,f) returns exp for any f.

substring	System	method

```
substring(s:string, i:integer, j:integer) → string
```

returns the substring of s starting at the ith character and ending at the jth. For example, substring("CLAIRE",3,4) returns "AI". If i is negative, the empty string is returned and if j is out of bounds (j > length(s)), then the system takes j=length(s).

symbol!	System	method

```
symbol!(s:string) → symbol
```

symbol!(s) returns the symbol associated to s. For example, symbol!("toto") returns 'toto'

time_get, time_set, time_show	System	method

```
time_get() → integer
time_set() → void
time_show() → void
```

time_set() starts a clock, time_get() stops it and returns an integer proportional to the elapsed time. Several such counters can be embedded since they are stored in a stack. time_show() pretty prints the result from time_get().

type!	Language	method

```
type!(x:any) → any
```

returns the smallest type greater than x (with respect to the inclusion order on the type lattice), that is the intersection of all types greater or equal to x.

U	System	method

```
U(s1:set, s2:set) → set
U(s:set, x:any) → any
U(x:any, y:any) → any
```

U(s1,s2) returns the union of the two sets. Otherwise, U returns a type which is the union of its two arguments. This constructor helps building types from elementary types.

use_as_output	System	method

```
use_as_output(p:port) → port
```

uses_as_output(p) changes the value of the current output (the port where all print instructions will be sent) to p. It returns the previous port that was used as output which can thus be saved and possibly restored later.

value	Reader	method

```
value(s:string) → any
```

returns the object whose name corresponds to the string.

verbose	Class	slot

   `verbose(s:meta_system)` $\rightarrow$ `integer`

verbose(system) (also verbose()) is the verbosity level that can be changed.

world?, world=, world+, world-	Class	method

   `world?()` $\rightarrow$ `integer`
   `world+()` $\rightarrow$ `void`
   `world-()` $\rightarrow$ `void`
   `world=(n:integer)` $\rightarrow$ `void`
   `world!-()` $\rightarrow$ `void`
   `world!=(n:integer)` $\rightarrow$ `void`

These methods concern the version mechanism and should be used for hypothetical reasoning: each world corresponds to a state of the database. The slots s that are kept in the database are those for which store(s) has been declared. These worlds are organized into a stack, each world indexed by an integer (starting form 0). world?() returns the index of the current world; world+() creates a new world and steps into it; world-() pops the current world and returns to the previous one; world=(n) returns to the world numbered with n, and pops all the intermediary worlds. world!-() returns to the previous world but carries the updates that were made within the current world; world!=(n) returns similarly to the world numbered with n.

write	Class	method

   `write(p:property, x:object, y:any)` $\rightarrow$ `any`

This method is used to store a value in a slot of an object. write(p,x,y) is equivalent to p(x) := y.

APPENDIX C: USER GUIDE

1. CLAIRE

When you run CLAIRE, you enter a *toplevel* loop. A prompt `claire>` allows you to give commands one at a time The expression is entered, followed by <enter> on the Macintosh version or <return> on the UNIX or OS/2 version[8]. The expression is evaluated and the result of the evaluation is printed out after an `eval[n]>` prompt where n starts from 0 and gets incremented by one on each evaluation. This counter is there to help you keep track of your session. To quit, you can type ^D, q (for quit) or exit(1).

```
claire> 2 + 2
eval[0]> 4
```

The value returned at the level n can also be retrieved later using the array EVAL. EVAL[n] contains the value returned by `eval[n]>`, modulo the size of this array. To prevent the evaluation of an instruction, one may use the backquote character (`) in a way similar to LISP's quote.

```
claire> `(2 + 2)
eval[1]> 2 + 2
```

Formally, the expression entered at the toplevel can be any <fragment>, to avoid painful parenthesis. To prevent ambiguities, the newline character is taken as a separator inside compounded expressions (cf. Appendix A, <comp-exp>). This restriction is only true at the top-level and not inside a file. It prevents from writing

```
claire>  1 + 2
         + 3
```

but not

```
claire>  1 + 2 +
         3
```

The CLAIRE system takes care of its memory space and triggers a garbage collection whenever needed. If CLAIRE is invoked from a shell, it can accept parameters according to the following syntax:

claire ⟨-s <int> <int>⟩^{opt}
⟨-n▐ -v <integer> ▐ -f <file> ▐ -m <module>⟩*
⟨-F <flag>⟩* ⟨-D ▐ -O ⟩^{opt}
⟨ ⟨ ⟨⟨-cm ▐ -cc⟩ <module>⟩ ▐ ⟨-cf ▐ -c⟩ <file>⟩ ⟨-o <file>⟩^{opt} ⟩^{opt}

The -s option allows to change the size of the memory zone allocated for CLAIRE. The first number is a logarithmic increment[9] for the static zone (bags, objects, symbols), the second number is a logarithmic increment for the dynamic zone (the stacks). For instance, -s 0 0 provides the smallest possible memory configuration and -s 1 1 multiplies the size of each memory zone by 2. The method *stat()* is useful to find out if you need more memory for your application. A good sign is the presence of numerous garbage collection messages.

Whenever CLAIRE starts, it looks for the *init.cl* file in the current directory. This file is loaded before any other action is started. When the environment does not include a shell (e.g. Macintosh), the first line of the init.cl file can be a comment of the form

```
;claire ⟨-s <int> <int>⟩ᵒᵖᵗ
```

---

[8] The prompt may also be `¿claire?` depending on your operating system.

[9] A logarithmic increment n means that the size is multiplied by 2^n.

The parameters after CLAIRE will be used as if they were entered from a shell. The loading of the init.cl file can be prevented with the -n option. The -v (for verbose) option will set the value of verbose() to the integer parameter and thus produce more or fewer messages.

The options -f and -m are used to load files and modules into CLAIRE. The argument <file> is a name of a file (e.g. `-f test` is equivalent to `load("test")`). The argument <module> is the name of a module that is either part of the CLAIRE system or defined in the *init.cl* file (`-m test` is equivalent to `get(test)`).

The option -F is used to set the value of a global_variable <flag> to false. This option can be used in conjunction with #if if to implement different versions of a same program in a unique file.

There are four options that invoke the CLAIRE compiler: -c, -cf, -cm and -cc. They are used to compile respectively a file (without and with linking), a module and a multi-module project. The -o option may be used to give a new name to the executable that is generated (if any). The options -O and -D are used respectively to increase the optimization or the debugging level (cf. Section 3).

When `claire -c test` is invoked, the file compiler takes a CLAIRE file, produces an equivalent C or C++ file and another C file called the system file. The first file is named <file>.c (here test.c) and the second file is named <out>-s.c (here test-s.c). They are both placed in the directory `source(compiler)` (cf. Section 3). The name <out> is <file> by default and is changed with the -o option. The option -cf is similar except that the generated files are compiled and linked directly by CLAIRE. This is done by producing a makefile <out>.mk that links the generated binaries with the necessary CLAIRE modules. The option -cf is the option that people use most, while the -c option corresponds to using CLAIRE as a code generator.

The -cc option is the equivalent of -c for a module: `claire -cc m` will produce a C/C++ file for each CLAIRE file in `made_of(m)`. It does not produce a system file, because it is designed to be used in a project with multiple modules. A unique system file for a set of modules is obtained with the *compile* method, as explained in section 3. On the other hand, the -cm option is the module equivalent of the -cf option. It is similar to -cc, but in addition it produces a system file for the module that is being compiled and a makefile which is executed by CLAIRE, producing the C/C++ compilation and link of the generated code.

If the environment does not provide a shell, compiling becomes a more complex task. One can use the *compile* method that is presented in section 3 to generate C or C++ files from CLAIRE files or modules. In addition the method *compile* must be used to generate the system file that contains the start up procedures. These files need to be compiled and linked explicitly using the users' choice of programming environment.

2. The Environment

CLAIRE provides a few simple but powerful tools for software development: interactive debugger, stepper and inspector. All three of them are contained in the System library and have the same structure of top level loops.

CLAIRE provides a powerful tool to trace programs. Trace statements are either explicit or implicit. To create an explicit trace statement, one uses the instruction

```
trace(level:integer,pattern:string,l:listargs)
```

which is equivalent to a format(pattern,l) onto the port trace_output() if verbose() is more than *level*. . Explicit trace statement are very useful while debuging. They may often be seen as "active comments" that describe the structure of an algorithm. For instance, we may use

```
trace(DEBUG, "start cycle exploration from node ~S\n",x)
```

Such a statement behaves like a "printf" if the verbosity level is less than the value of the global variable DEBUG, and is inactive otherwise. The goal is to be able to selectively turn on and off pieces of the debugging printing statements. By changing the value of the DEBUG variable, we can control the status of all trace statements that use this variable as their verbosity level.

It would be nice if we could separate visually these tracing statements from the rest of the code, especially since too many trace statements can quickly reduce the readability of the original

algorithm. To achieve this goal, CLAIRE provides the notion of extended comments. An extended comment is a comment that starts with //[..], and which is treated like an explicit trace statement. For instance, the previous trace statement would be written as

```
//[DEBUG] start cycle exploration from node ~S // x
```

More precisely, an extended comment can only be used inside a block (i.e., within parentheses). The verbosity level is the string contained between the two brackets after //, the rest of the line is the concatenation of the pattern string and the argument list, separated with another "//", unless the list is empty. The last chararacter should be a comma if a comma would be required after a trace statement in a similar position (i.e., if the trace statement is not the last statement of the block). Here is a simple example :

```
let x := 1 in
  ( //[1] start the loop with ~S // x,
    while (x < 10)
      (if g(x) f(x,x),
        //[2] examine ~S // x
      ))
```

Implicit trace statements are produced by tracing methods or rules. The instruction `trace(m:property)` will produce two trace statements at the beginning and the end of each restriction of m (method). For instance, here what we could get by tracing the function *fib*.

```
1:=> fib(3)
2:=>> fib(2)
3:=>>> fib(1)
[3]>>> 1
3:=>>> fib(0)
[3]>>> 1
[2]>> 1
2:=>> fib(1)
[2]>> 1
[1]> 3
```

The level associated with the method's trace statement is the current level of verbose(). At any time, the trace statements can be deactivated with *untrace(m:method)*. The other way to generate trace statements is to activate the trace generation of the rule compiler with *trace(if_write)*. Whenever *trace(if_write)* is active, the code generated by the rule compiler will be instrumented with trace statements. Therefore, a statement will be printed as soon as the rule is triggered. One can play with *trace(if_write)* and *untrace(if_write)* to selectively instrument some rules and not the others, and later to activate/deactivate the trace statements that have been generated.

The output_port can be set with trace(p:port) or trace(s:string) which creates an implicit port fopen(s,"w"). In addition, trace(...) can be used for two special functions. trace(m:module) activates a compiled module, which means that its compiled methods can be traced exactly like interpreted method. This will only happen if the module was compiled with the -D option (cf. Section 3). trace(spy) activates the *spy* property if the method spy @ void has been defined previously. This means that spy() is invoked after each method call. This will slow down execution quite a lot but is extremely useful to detect which method has caused an undesired situation. Suppose for instance that the value r(X) must always be lower than 10. After the execution of your program, you find that r(X) = 12. If you try

```
[spy() -> assert(r(X) <= 10)]
trace(spy)
```

and run your program, it will stop exactly after the "wrong" method call that violated your assumption. *assert*(X) is a convenient macro which is equivalent to (if not(X) error(...)). The error message indicates in which file/line the error occurred. assert() statements are not compiled unless the debug mode of the compiler is active, or unless safety(compiler) = 0. Thus, assert statements should be used freely in a CLAIRE program since they are known to have a very positive effect on code safety and reliability.

The *debugger* is a toplevel loop that allows the user to inspect the stack of function calls. The debugger is invoked each time an error occurs, or by an explicit call through a *breakpoint()* statement. First, the debugger must be activated with the *debug()* method which works as a toggle (activate/deactivate). Then, whenever an error occurs, the debug toplevel presents the debug> prompt. In addition to being a standard read-eval-print toplevel (thus any CLAIRE expression can be evaluated), the following additional methods are supported:

where(n:integer)	shows the n last function calls in the stack. For each call, only the selector (the property) and the value of the arguments are shown
block(n:integer)	shown the n last function calls with the explicit method that was called, all the local variables (including the input) parameters and their current values.
dn(n:integer)	moves the current top of the stack down by n levels
up(n:integer)	moves the top of the stack up by n levels

For instance, here what we could get

```
[f(n:integer) -> let y := n - 1 in 1 / n + f(y)]
debug()
f(2)
--------------------- Debug ---------------------------
Integer arithmetic: division/modulo of 1 by 0
debug> where(10)
debug[1] > f(2)
debug[2] > f(1)
debug[3] > f(0)
debug[4] > 1 / 0
debug> block(10)
debug[1] > f@integer(x = 2, y = 1)
debug[2] > f@integer(x = 1, y = 0)
debug[3] > f@integer(x = 0, y = -1)
debug[4] > 1 / 0
```

The debugger only shows method calls that occur in interpreted code or in compiled code from an active module. As for trace statements, an active module needs to be compiled with the -D option first and activated with the *trace(m)* statement. For a compiled method, the *block(n)* instruction will only show the module where the method is defined.

The debugger can be invoked explicitly with the *breakpoint*() statement, which allows the user to inspect the stack of calls and the values of the local variables at the time the breakpoint is set. Once the inspection is completed, the execution resumes normally (as opposed to the usual error handling case). The debugger prompt allows the user to evaluate any expression, thus to inspect the current state of any objects.

This library also provides a stepper. The method *step()* invokes the stepper, which will be active for the next message evaluation. The stepper can also be turned on after a given method p is evaluated, with the command step(p). Once it is triggered, the stepper stops at each function call, shows the name of the method and the value of the arguments and offers the following menu:

[s,i,o,q,t,b,x]

s:step	the stepper will evaluate the function
i:in	the stepper enters the function and stops at the first function call
o:out	the stepper exits the function

q:quit   the stepper stops (but may restart if there are other calls to active methods)

t:trace the stepper starts tracing the function

b:breakpoint you enter a breakpoint toplevel, to inspect the current state of objects.

x:exit   you exit the stepper by raising an exception. You come back to the interpreter prompt.

Finally, an inspector is also available for browsing CLAIRE objects. It is turned on by the method inspect. Calling *inspect*(x) will give information about x (the same information that the method show would give, that is the list of all slots and their values) and make you enter another toplevel loop with an `inspect>` prompt.

Each information about the inspected object is numbered. Typing in a number will make the inspector focus on the corresponding slot of the object.

If the inspected object is x, typing the property p will drive the inspector to the object p(x)

Typing in the name of an object will focus the inspector on that object

Typing up will return to the previously inspected entity.

Typing q will have you quit the inspector.

3. The compiler

3.1 C/C++ code generation

The CLAIRE compiler generates C or C++ files from a CLAIRE file (or a set of files associated with a module). The use of C or C++ is determined by a global variable as we shall later see. For a given file f.cl, it will produce a code file f.cp (or f.c) and a header file f.h. In the case of a module m, it will produce a unique header file m.h and multiple code files. Each code file contains a list of C functions for each method in the file, plus one large method that contains the C code for generating the CLAIRE objects that are defined in f.cl. In addition to a few compiler-generated comments lines, the comments lines that begin with // in the CLAIRE source file are also included in the C generated file.

The interface file contains the C structures or C++ classes generated for each CLAIRE class, the function prototypes for each compiled method, and the extern definition of all the generated identifiers (see later). In addition, the compiler can also be used to generate a "system file", which name is f-s.c, where f is the output parameter. This system file is either produced directly by a call to `compile(out:string,l:list[module])` or implicitly by using the -cm or -cf options (out is the name of executable and l is the list of necessary modules).

The system file contains code for building all the modules and loading them in the right order. Its key generated function is run_claire() which must be invoked by the main() function of your program (this is done automatically with the main.c file that is used with the -cm or -cf option). If you decide to write your own main() function, you must remember to call run_claire before using any CLAIRE objects. The *mainm.c* file that the CLAIRE compiler uses for the -cm and -cf option is straightforward. It contains a very simple toplevel and provides only two features: it loads the start.cl file before entering the toplevel (if it exists) and it accepts the "-s a b" option as shell parameters to change dynamically the size of the memory allocated to your program.

CLAIRE provides a way to include C++ directly into the generated code. The method *externC* has one argument (a string) and no effect when interpreted. On the other hand, a call to externC is compiled into its string argument. For instance, to define a bitwise and operation we may use

```
[bit&(x:integer,y:integer) : integer -> externC("(x & y)") ]
```

3.2 What the compiler produces

Reading or using the C++ generated code is very easy as soon as you have a vague idea of what is produced by the compiler (here we assume that you have already read Section 6.5). The first output of the compiler is a set of structure or class definition that is placed in the header file. Each CLAIRE class that is an object sort (i.e., that is included in object) produces such a structure, where each slot of

the class becomes a data member in the structure. This structure will be used to access CLAIRE objects within a C program as if it was a standard C/C++ object. For instance, a definition like

```
[C <: object(x:string,y:int,z:float)]
```

will produce

```
class C : public object
        {    char *x;
             int y;
             double *z;}
```

The name used for the structure is exactly the same as the CLAIRE name, with the exception of special characters in {'.', '&', '-', '"','+', '%', '*', '?', '!', '<', '>', '=', '^', '@'} that are translated into a short sequence of charcacters that are acceptable for C/C++. Using the CLAIRE name for the structure has the advantage of simplicity but the user must keep this in mind to avoid name conflicts (such as using a C++ keyword for a class name).

The second output of the CLAIRE compiler is a set of identifiers that correspond to the set of CLAIRE named objects. For each named object x (i.e. that belongs to thing), CLAIRE generates a C identifier with name L_x (the prefix "L_" is added). If the name is a qualified symbol m/x from module m, the generated identifier is L_m_x. As previously, special characters are translated. To find out which identifier is generated, one may use the c_test method. This method is an on-line compiler that is intended to show what to expect. `c_test(x:any)` takes an instruction x and shows what type will be inferred and what code will be produced. For instance, c_test(x:thing) will show which identifier will be generated. To use *c_test* with a complex instruction, one may use the ` (backquote) special character that prevents evaluation. For instance, one may try

```
c_test(`(for x in class show(x)))
```

Let us consider a small example that will show how to create a claire object from C++ or how to invoke a method. Suppose that we define :

```
[point <: class(x:integer, y:integer, tag:string)]
[f(p:point,s:string) -> tag(p) := s ]
```

The code shown by

```
c_test(`f(point(x = 1), "test")
```

will be (modulo the GC statements that depend on the settings and that will be discussed later) :

```
{    point * v_arg1;
     char * v_arg2;
     { point * _CL_obj = (point *) make_object_class(L_point);
       _CL_obj->x = 1;
       add_I_property(L_instances, L_point, 11, _object_(CL_obj));
       v_arg1 = _CL_obj;}
     v_arg2 =  "test";
     f_point_claire(v_arg1,v_arg2);}
```

In addition, CLAIRE also generates a special variable for each module that contains their index value. This variable is obtained with the prefix "N_".

The third output of the CLAIRE compiler is a set of functions. CLAIRE generates a C function for each method in the CLAIRE file. The function uses a name that is unique to the method as explained in Section 6.5. The function name associated to a method can be printed with the `c_interface(m:method)` method. The input variables (as for any local variables) are a straightforward translation from CLAIRE (same name, equivalent C/C++ type). The body of the function is the C code that is equivalent to the original CLAIRE body of the method. The C code generated by CLAIRE is an almost straightforward translation of the source code. The only exceptions are the additional GC protection instructions that are added by the compiler. These macros (GC...) can be ignored when reading the code (they are semantically transparent) but they should not be removed ! In addition, CLAIRE also produces one load function for each file f (with name "load_f") that contains code which builds all the objects, including the classes and methods, contained in the file.

3.3 Customizing the compiler

There are a few parameters that the user can control the CLAIRE compiler. They are all represented by slots of the compiler object. The string *source*(compiler) is the directory where all generated C code will be placed. You must replace the default value of this slot by the directory that will contain the generated code.

The second slot *safety*(compiler) contains an integer that tells which level of safety and optimization is required, according to the following table:

> 0 -> super-safe: the type of each value returned by a method is checked against its
>   range, and the size of the GC protection stack is minimized. All assertions
>   are checked
>
> 1 -> safe (default)
>
> 2 -> we trust explicit types & super. The type information contained in local variable
>   definition (inside a let) and in a super (f@c(...)) has priority over type inference
>   and run-time checks are removed.
>
> 3 -> no overflow checking (integer & arrays), in addition to level 2
>
> 4 -> we assume that there will be no selector errors or range errors at run-time.
>   This allows the compiler to perform further static binding.
>
> 5 -> we assume that there will be no type errors of any kind at run-time.
>
> 6 -> unsafe (level 5 + no GC protection). Assumes that garbage collection will
>   never be used at run-time

The slot *external*(compiler) contains the name of the C/C++ compiler that should be used by the -cm and -cf options. For instance, its default UNIX value is "gcc". It could be changed to "gcc -p" to use the profiler (for instance).

The slot *headers*(compiler) contains a list of strings, each of which is a header file that needs to be used the generated C file. This is useful when you define methods by external functions, whose prototypes are in a given header (such as a GUI library header).

The last slot, *debug?*(compiler), contains a list of the modules for which debuggable code must be generated. This slot is usually set up directly using the -D option. By default, generated code is not instrumented which means that the tracer, the debugger or the stepper cannot be used for compiled methods. On the other hand, when debuggable code is generated, they can be used just as for interpreted code. One just needs to activate the compiled module with a trace(m) statement. The overhead of the instrumentation is marginal when the module is not active. Once it is active, the overhead can vary in the 10-100% range.

The last way to customize the compiler is to introduce new imported sorts, as defined in Section 6.5. This is done by defining a new class *c t*hat inherits from the root `imported` and telling the compiler what the equivalent C type is with the *c_interface* method. c_interface(c:class,s:string) instructs the compiler to use s as the C type for the external representation of entities of type c. For instance, here is a short CLAIRE program that defines a new type: long integer (32bits integers).

```
long <: imported()
(#if loaded(Compile) c_interface(long,"long"))

[+(x:long,y:long) : long -> function(plus_long) ]
[self_print(x:long) : void -> function(print_long) ]
```

Notice that we guard the c_interface declaration with an #if to make sure that the compiler is loaded. We may now define the C implementation of the previous method as follows.

```
long plus_long(long x, long y) { return x + y;}
void print_long(long x) {fprintf(LO.port,"%dL",x);}
```

Last, we must make sure that the header file corresponding to the previous functions is included by the CLAIRE compiler using the headers(compiler) slot. The choice between C and C++ cannot be reversed lightly since the C and the C++ code generated by CLAIRE are incompatible. However, it is easy to force the compiler by using the two following variables (in the Compile module). *C++* is a boolean that tells that C++ should be used, *fe* is a string that contains the extension for the generated files.

The CLAIRE compiler also generates code to check that objects slots do not contain the special "unknown" value. This can be avoided by declaring one or many properties as "known", through the following declaration :

```
known! (<relation>*)
```

The compiler will not generate any safety check for the relations (properties or arrays) that are given as parameters in a *known!* statement.

3.4 Iteration and Patterns

We have seen how CLAIRE supports the optimization of iteration and membership for sets that are represented with new data structure. This is done through the addition of inline restrictions to respectively the *iterate* and the % property. However, there are cases where sets are better represented with expressions than with data structures. Let us consider two examples, *but* and *xor*, with the following samples

```
for c in ({c in class | length(slots(c)) > 5} but class) ....

(for x in (s1 & s2) ...          ;; iterate the intersection
 for x in (s1 xor s2) ...        ;; iterate the rest of (s1 U s2)
```

The definition of the sets are as follows; (s but x) is the set of members of s that are different from x; (s1 xor s2) is the set of members of s1 or s2 but not both. It would be perfectly possible to implement these sets with either simple methods (set computation) or new data structures, with the appropriate optimization code. However, there are two strong drawbacks to such an approach

- it implies an additional object instantiation which is not necessary,
- it implies evaluating the component sets to create the instance, which could have been prevented as shown by our first example (the selection set can be iterated without being built explicitly).

A better approach is to manipulate expression that represent sets directly and to express the optimization rules directly. Although this is supported by CLAIRE through the use of reflexion and thus out of scope for this manual, we have identified a subset of expressions for which a better (simpler) support for such operations is provided.

The key concept is the *pattern* concept, which is a set of function calls with a given selector and a list of types of the arguments (that is a list of types to which the results of the expressions that are the arguments to the call must belong). A pattern in CLAIRE is written p[tuple(A,B,...)] and contains calls p(a,b,...) such that a is an expression of type A ... and so on. Patterns have two usage: the iteration of sets represented by expressions and the optimization of function composition (including membership on the same expressions). To better understand what will follow, it is useful to know that each function call is represented in CLAIRE by an object with two slots: *selector* (a property) and *args* (the list of arguments).

First, the CLAIRE compiler can be customized by telling explicitly how to iterate a certain set represented by a function call. This is done by defining a new inline restriction of the property *Iterate*, with signature (x:p[tuple(A,B,...)],v:Variable,e:any). The principle is that the compiler will replace any occurence of (for v in p(a,b,...) e) by the body of the inline method as soon as the type of the expressions a,b,... matches with A,B,.... This is very similar to the use of *iterate*, but we leave as an exercice for the reader to find out why two different properties are needed.

For instance, we can define two new restrictions of *Iterate* as follows.

```
[Iterate(x:but[tuple(any,any)],v:Variable,e:any)
   => for v in eval(args(x)[1]) (if (v != eval(args(x)[2])) e) ]

[Iterate(x:xor[tuple(any,any)],v:Variable,e:any)
   => (for v in eval(args(x)[1]) (if not(v % eval(args(x)[2])) e),
       for v in eval(args(x)[2]) (if not(v % eval(args(x)[1])) e) ]
```

If we need to have access to a component of the call that matches the pattern, we use a special *eval* call: instead of performing the substitution, the compiler will evaluate what is inside the *eval* call. Here is what will be obtained for our two initial examples :

```
for c in get_instances(class)
    (if (length(slots(c)) > 5)
        (if (c != class) ....

(for x in (s1 & s2) ...              ;; iterate the intersection
  (for x in s1 (if not(x % s2) ...
   for x in s2 (if not(x % s1) ...
```

Patterns are also useful to add new code substitution rules. This is achived with a restriction (an inline method) whose signature contains one or more patterns and the class *any*. The compiler tries to use it based on the matching of the expressions (pattern-matching as opposed to type-matching). For instance, here is how we optimize the membership to sets represented by a "but" expression.

```
[%(x:any,y:but[tuple(any,any)])
   => x % eval(args(x)[1]) & (x != eval(args(x)[2]))]
```

The use of patterns is an advanced feature of claire, which is not usually available in programming languages. It corresponds to what could be called composition polymorphism, where the implementation of a call *f(...,y, ...)* may change if *y* is itself the result of applying another function *g*. It allows to implement simplification rules such as

$(A + B)[i,j] = A[i,j] + B[i,j]$

by declaring

```
[nth(x:+[tuple(matrix,matrix)],i:any,j:any)
   => eval(args(x)[1])[i,j] + eval(args(x)[2])[i,j] ]
```

INDEX