Combining Logical
Assertions, Inheritance,
Relations and Entities

Introduction to the CLAIRE Programming Language Version 4.0

DRAFT

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

Table of Contents

0. Introduction

CLAIRE is a high-level functional and object-oriented language with 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,
- propagation rules based on events,
- 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,
- 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 first 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 C4.0 release has substituted Go to C++ to increase the robustness. The key set of features that distinguishes CLAIRE from other programming languages has been dictated by our experience in solving complex optimization problems. Of particular interest are two features that distinguish CLAIRE from procedural languages such as C++, Go or Java:

- 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. Unlike most logic programming languages, this type of backtracking covers any user-defined structure, not simply a set of logic variables.
- Production rules: CLAIRE supports rules that bind a CLAIRE expression (the conclusion) to the combination of an event and a logical condition. Whenever this event occurs, if the condition is verified, then the conclusion is evaluated. The emphasis on events is a natural evolution from rule-based inference engines and is well suited to the description of reactive algorithms such as constraint propagation.

CLAIRE provides automatic memory allocation/de-allocation. Also, set-oriented programming is much easier with a set-oriented language like CLAIRE than with libraries. CLAIRE is close to 30 years old, but this new 4.0 release reaches a new level of robustness and performance. Appendix C, CLAIRE's user guide, provides a release history that details the changes from CLAIRE 2.0 to 3.0 and 3.0 to 4.0, and gives some insights about earlier versions.

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 Go, since a CLAIRE program can be naturally translated into a Go program (We now use Go as our target language of choice, but CLAIRE's compiler could be extended to produce Java as it did in the past). 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. Sets and lists may be typed to provide a more robust and expressive framework. 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.

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 tables (relations) are defined. Classes belong themselves to a 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. Module decomposition can either be parallel to the class organization (mimicking C++ encapsulation) or orthogonal (e.g., encapsulating one service among multiple classes). CLAIRE module approach is a close match to the concept of "packages" in Go.

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.

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 well-formed 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 covers the most original aspects, namely rules and versions. It introduces the notion of generalized tables and event-based rules. The rules in v3.2 are a departure from the older production rules that were part of earlier CLAIRE versions. 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 on GitHub (http://github.com/ycaseau/CLAIRE4.0).

This last appendix also contains a few tips for migrating a program from earlier versions of CLAIRE.

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. Notice that you may obtain a warning if you load CLAIRE and no file « init.cl » is found in your current directory. You can ignore this message for a while, then you may use such a file to store some of your favorite settings. You are now 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 ~A\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 but is strongly advised.

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")
eval[1] true
claire> main()
eval[2] claire release 4.0.4
claire> q
%
```

Each CLAIRE program is organized into blocks, which are surrounded by parentheses, and definitions such as 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).

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 becomes a CLAIRE object

[1] In the following we assume that CLAIRE is invoked in a workstation/PC environment using a command shell. You must first find out how to invoke the CLAIRE system in your own environment.

[2] The release is a string « 4.X.Y » and the version is a float X.Y, where X is the version number and Y the revision number. The release number in this book (4) should be the same as the one obtained with your system. Changes among different version numbers should not affect the correctness of this documentation.

that is evaluated). It can also be compiled (through the intermediate step of C++ code generation). To compile a program, one must place it into a **module**, which plays a double role of a compilation unit and namespace. The use of modules will be explained later on.

Let us now write a second program that prints the first 11 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 n-th Fibonacci number.

```
fib(n:integer) : 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)))
```

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 mandatory if the function returns a useful result since the default range is *void*, which means that no result is expected. Conditionals 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); (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),
                      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 any character is 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. Using x+2 as a variable name is not a good idea, but being able to use names such as *% that include “arithmetic” characters is quite useful.

> **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 important 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 a table *fib[n]*, as in the following version of our program.

```
fib[n:integer] : integer :=  1

main()
  ->  (for i in (2 .. 10) fib[i] := fib[i - 1] + fib[i - 2],
       for i in (0 .. 10) printf("fib(~S) = ~S\n",i,fib[i]) )
```

An interesting feature of CLAIRE is that the domain of a table is not necessarily an interval of integers. It can actually be any type, which means that tables can be seen as "extended dictionaries" (Section 5.1). On the other hand, when the domain is a finite set, CLAIRE allows the user to define an "initial value" using the **:=** keyword, as for a global variable assignment. 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_print(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) )
```

Module organization is a key aspect of software development and should not be mixed with the code. Modules' definitions are placed in the init.cl file which is loaded automatically by the interpreter or the compiler. It is also possible to put module definitions in a project file, and to load this file explicitly.

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

The statement *part_of = y* inside the definition of a module x says that x is a new child of the module y. 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. Part 6).

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 place these definitions into the module called phone_application. We also use comments that are defined in CLAIRE as anything that in on the same line after the character ';' or after the characters '//' as in C++.

```
// definition of the module
begin(phone_application)

// value is a table that stores the phone #
private/value_string[s:string] : string

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

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

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

```
end(phone_application)
```

This example illustrates many important features of modules. Modules are first-class objects; the statement *begin*(x) tells CLAIRE to use the namespace associated with the module x. We may later return to the initial namespace with *end*(x). When begin(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 *phone_application* itself or its descendents. 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 currently active module, if it is declared with the qualified form. Secondly, 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 table (in the CLAIRE sense) anywhere but in the descendents of the module *phone_application*

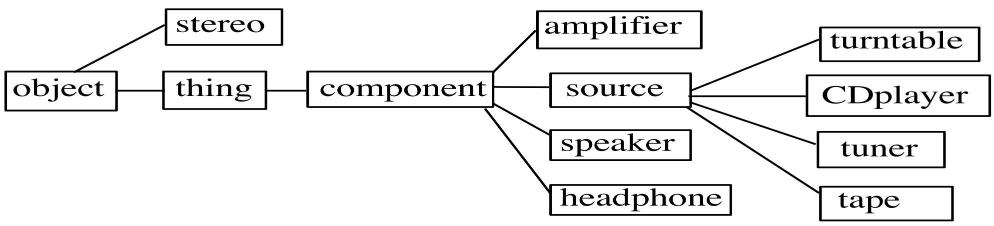
The previous example may be placed in any file and loaded at any time. However, the preferred way to write the code associated with a module is to place it in one of the files that have been identified in the *made_of* slot (here, "phone.cl"). These files may be loaded inside a module's namespace using the *load(m:module)* method, without any explicit use of begin/end. For instance, we could remove the first and last lines in the previous example and put the result in the file *phone.cl*.

Appendix C shows the command-line syntax for invoking CLAIRE. For the time being, it is useful to know that claire –f <file> invokes claire and loads the file <file>. Also, claire –m <module> is similar but loads the module <module> which is defined in the *init.cl* file.

1.2 Objects and Classes

Our next example is a small pricing program for hi-fi 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, ohm:{4,8})
headphone <: component(maxpower:integer,  ohm:{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)
```

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

Now that we have defined the taxonomy of all the objects in our hive 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")
s1 :: speaker(    ohm = 8, maxpower = 150,
                  price = 1000, brand = "Magisound")
s2 :: speaker(    ohm = 8, maxpower = 80,
                  price = 400, brand = "Magisound")
s3 :: speaker(    ohm = 4, maxpower = 40,
                  price = 150, brand = "Cheapy")
ph :: speaker(ohm = 4, maxpower = 40, price = 50, brand = "Okyonino")
    etc ...
```

Now that we have defined some 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)

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

InventoryTotal:integer :: 0
```

Note here the use of set image (we consider the list of all *x.price* for all x in the following set: the union of *s.sources, {s.amp}* and *s.out*). 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 a table 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).

```
[best_price(s:set[component]) : integer
  -> let p := 100000 in
     (if (size(s) = 0) p := 0
      else if (size(s) = 1) p := price(s[1])
      else 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 + best_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)
```

A rule is defined by a condition and a conclusion (using a pattern rule(*condition* => *conclusion*)). The condition is the combination of an event pattern and a Boolean expression. The event pattern tells when the Boolean expression should be checked, in case of success the conclusion is evaluated. Here are some simple rules that will raise exceptions when some technical requirements are not met.

```
compatibility1() :: rule(
     st.speaker :add sp & not(sp.ohms % st.amplifier.ohms) )
     => technical_problem(s = "conflict speakers-amp"))

compatibility2() :: rule(
     st.sources :add x & size(st.sources) > st.amp.inputs
     => technical_problem(s = "too many sources"))

compatibility3() :: rule(
    (st.out :add x) & x.maxpower < st.amp.power
     => 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, suppose that 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* trigger the evaluation of the concerned rules.

```
my_system :: stereo(amp = amp1)
(exists(sp in speaker |
   (try (my_system.out :add sp, true)
    catch technical_problem
       (//[0] rejects ~S because ~A // sp, exception!().s,
        my_system.out :delete sp,
        false))),
```

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. To do so, we can 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 *choice()* and *backtrack()* respectively create a new world (i.e., create a choice point) 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>() , syst:stereo := stereo() in
      (for a in amplifier
         (syst.amp := a,
          for sp in speaker  try
            (choice(),
             syst.out := set(sp),
             for h in headphone try
               (choice(),
                syst.out :add h,
                for s1 in musical_source try
                  (choice(),
                   syst.sources := set(s1),
                   for s2 in {s in musical_source |
                                owner(s) != owner(s1) & s.price < s1.price} try
                      (choice(),
                       syst.sources :add s2,
                       solutions :add copy(syst),
                       backtrack())

                     catch technical_problem backtrack(),
                 backtrack())
             catch technical_problem backtrack(),
             backtrack())
          catch technical_problem backtrack(),
          backtrack())
```

```
    catch technical_problem  backtrack()),
    solutions)
```

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

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 -> (total_price(s1) <= total_price(s2))

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

1.4 Worlds & Hypothetical Reasoning

We shall conclude this tutorial with a classical SUDOKU example, since it illustrates the benefits of built-in hypothetical reasoning in CLAIRE using the "world mechanism" (cf. Section 5.3).

The first part of our last program describes the Sudoku data structures: cells, grid and cell sets. Cells are straightforward, defined by x,y coordinates and value, which is the integer between 1 and 9 that we need to find. A grid is simply a 9 x 9 matrix of cells. The only subtlety of our data model is the explicit representation of lines, column and 3x3 squares as subsets of cells called CellSets (with a unique property: each digit must appear exactly once in each such set).

We notice that we declare *value* and *count* to be defeasible slots (cf. Section 5) which will enable hypothetical reasoning (to search for the solution). We also create an "event" property (*countUpdate*) to be used with a rule.

```
// data structure
Cell <: object
CellSet <: object

// cell from a 9 x 9 Sudoku grid
Cell[x,y] <: object(x:integer,
                    y:integer,
                    possible:list<boolean>,  // list of possible values for the cell
                    count:integer = 9,       // number of possible value
                    value:integer = 0,       // assigned value to the cell (0 = none)
                    line:CellSet,            // the line to which the cell belongs
                    column:CellSet,          // same for column …
                    square:CellSet)          // one of the 9 3x3 squares

// a set of cells: line, column, square that holds the AllDiff constraint
CellSet[cells] <: object(cells:list<Cell>,          // cells that belong to the set
                         counts:list<integer>)      // a possible value counter

// two defeasible slots for hypothetical reasoning, but possible uses direct store
store(value,count)

// event that signals an update on counts for value v
countUpdate :: property(domain = CellSet, range = (1 .. 9))

// creates a cell
makeCell(a:integer,b:integer) : Cell
  -> Cell(x = a, y = b, possible = list<boolean>{true | i in (1 .. 9)}, value = 0)

// A sudoku grid
Grid <: object(cells:list<Cell>,
               lines:list<CellSet>,
               columns:list<CellSet>,
               squares:list<CellSet>)

// useful for debug
nth(g:Grid,i:(1 .. 9),j:(1 .. 9)) : Cell -> some(c in g.cells | c.x = i & c.y = j)

// creates a grid
makeGrid() : Grid
  -> let g := Grid() in
      (for i in (1 .. 9)
         for j in (1 .. 9) g.cells :add makeCell(i,j),
       for i in (1 .. 9)
         let li := list<Cell>{c in g.cells | c.x = i},
             cs := CellSet(cells = li, counts = list<integer>{9 | i in (1 .. 9)}) in
           (g.lines :add cs,
            for c in li c.line := cs),
       for j in (1 .. 9)
        let co := list<Cell>{c in g.cells | c.y = j},
```

```
          cs := CellSet(cells = co, counts = list<integer>{9 | i in (1 .. 9)}) in
         (g.columns :add cs,
          for c in co c.column := cs),
      for k1 in (1 .. 3)
        for k2 in (1 .. 3)
          let sq := list<Cell>{c in g.cells | abs(3 * k1 - c.x - 1) <= 1 &
                                              abs(3 * k2 - c.y - 1) <= 1},
              cs := CellSet(cells = sq, counts = list<integer>{9 | i in (1 .. 9)}) in
             (g.squares :add cs,
              for c in sq c.square := cs),
        g)
```

We now define a few rules that capture the reasoning about possible values for each cells. The first rule is triggered when we select a value for a cell. We disable the counters associated with the three "cell sets" of the cell (its line, its column and its 3x3 neighbor square) and the value v that got picked : the assignment of 0 to the counter tells that it is no longer necessary since the value v was used for these "CellSets". We then propagate the information that, since v was picked, all other values that were still possible for this cell are no longer possible (using the "noLonger" method. Last, we propagate to the three CellSets the fact that v is not allowed any more, using the "forbid" method. This method main ambition is to remove v for the "c.possible" list of possible value (using the store defeasible update which is explained in chapter 5), and then to maintain the "counters" associated with the cellsets. However, there is a trick: there are other rules, and it is possible that some of the inferences yield contradictory conclusion. Therefore, when we remove a possible value, we must make sure that it has not been picked as the current value by some other action. If this is the case, we raise a "contradiction", a special kind of CLAIRE exception.
The second rule is much simpler: when there is only one possible value for a cell, we can deduce that the cell must contain this value ☺ The third rule is triggered by the "updateCount" event, which occurs when we change the counter associated with a CellSet (in the "oneLess" method). This rule (r3) says that when a counters reaches 1, we may assign this value to the only cell in the CellSet which is still a candidate.

```
// first propagation rule
r1() :: rule(
     c.value := v => (store(c.line.counts,v,0),        // disable counts[v] since v was found !
                      store(c.column.counts,v,0),
                      store(c.square.counts,v,0),
                      for v2 in (1 .. 9)                // for all values v2 that were still OK
                        (if (v != v2 & c.possible[v2]) noLonger(c,v2),
                      for c2 in (c.line.cells but c) forbid(c2,v),       // v is used for c.line
                      for c2 in (c.column.cells but c) forbid(c2,v),     // … and c.column
                      for c2 in (c.square.cells but c) forbid(c2,v)))) // … and c.square

// noLonger(c,v2) tells that v2 is no longer a possible value
noLonger(c:Cell,v2:(1 .. 9)) : void
  ->  (store(c.possible,v2,false),       // avoid double count
       oneLess(c.line,v2),               // v2 looses one support cell for c.line
       oneLess(c.column,v2),             // same for c.column
       oneLess(c.square,v2))             // and c.square

// forbid a value
// Attention: if we forbid a value that is assigned, we must raise a contradiction
forbid(c:Cell,v:(1 .. 9))
  -> (//[3] forbid ~S(~A) -> ~A // c,c.value,v,
      if (c.value = v) (//[5] contradiction while propagation //,
                        contradiction!())
      else if (c.value = 0 & c.possible[v])
         (store(c.possible,v,false),
          c.count :- 1,
          oneLess(c.line,v),
          oneLess(c.column,v),
          oneLess(c.square,v)))

// remove a value in a CellSet
oneLess(cs:CellSet,v:(1 .. 9)) : void
  -> let cpos := cs.counts[v] in
      (if (cpos > 0)                   // cpos = 0 ⇔ counter is inactive
         (store(cs.counts,v,cpos - 1),     // update the counter
          updateCount(cs,v)))              // creates an event

// second rule : if c.count = 1, the only possible value is certain
r2() :: rule(
     c.count := y & y = 1 =>  c.value := some(y in (1 .. 9) | c.possible[y]))


// third rule (uses the CellSetSupport event) :
// if a value v is possible only in one cell, it is certain
r3() :: rule(
     updateCount(cs,v) & cs.counts[v] <= 1
```

```
=> when c := some(c in cs.cells | c.value = 0 & c.possible[v]) in c.value := v
   else contradiction!())
```

The hard part of the program is the set of rules, because it captures the logic inferences. Solving the puzzle is easy because we may leverage CLAIRE's built-in hypothetical capabilities, that is, the ability to explore a search tree. To define the search tree, we create a method "*findPivot*" which select the cell with smallest "support" set of possible values. The exploration of the search tree (solve) is defined recursively: pick the pivot cell, for each value in the possible set, try to assign this value to the cell and recursively call the solve method. We use the *branch(X)* control structure (cf. Section 3.6), which creates a "branch" of the search tree and evaluate the CLAIRE expression X within this branch. If X returns true, the search is considered a success and the current state is returned. If X returns false, the search has failed and the branch is removed, that is, CLAIRE returns to its previous state before branch(X) was invoked. Notice that the method solve is only 5 lines long and that it is very easy to modify to accomplish other goals, such as counting the number of solutions to the Sudoku problem.

```
// finds a cell with a min count (naive heuristic)
[findPivot(g:Grid) : any
  -> let minv := 10, cmin := unknown in
       (for c in g.cells
          (if (c.value = 0 & c.count < minv)
              (minv := c.count, cmin := c)),
        cmin) ]

// solve a sudoku : branch on possible values using a recursive function
// branch(...) does all the work :)
[solve(g:Grid) : boolean
  -> when c := findPivot(g) in
       exists(v in (1 .. 9) |
              (if c.possible[v] branch((c.value := v, solve(g)))
               else false))
     else true]

// show the solution
[see(g:Grid)
  -> printf("\n\t------------------\n"),
     for i in (1 .. 9) printf("\t~I\n",(for j in (1 .. 9) printf("~A ",g[i,j].value))) ]
```

To play with this program, all we need is a small method that translates an existing Sudoku problem (taken from a magazine, expressed as a list of list of integers, where 0 represents the absence of value).

```
// create a grid from a problem
[grid(l1:list[list[integer]]) : Grid
   -> let g := makeGrid() in
        (assert(length(l1) = 9),
         for c in g.cells
           let i := c.x, j := c.y, val := l1[i][j] in
             (if (val != 0)  c.value := val),
          g) ]
// example from Yvette
S1 :: grid(list(list(0,3,0,0,9,0,0,1,0),
                list(0,0,7,0,0,0,0,0,6),
                list(0,0,0,0,3,4,0,0,7),
                list(0,0,0,0,0,0,0,0,3),
                list(8,2,1,0,5,0,4,7,9),
                list(9,0,0,0,0,0,0,0,0),
                list(4,0,0,5,2,0,0,0,0),
                list(3,0,0,0,0,0,2,0,0),
                list(0,6,0,0,4,0,0,5,0)))

// this could be entered from the CLAIRE top-level ☺
(solve(S1), see(S1))
```

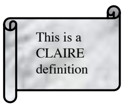
2. OBJECTS, CLASSES AND SLOTS

2.1 Objects and Entities

A program in CLAIRE is a collection of entities (everything in CLAIRE is an entity). Some entities are pre-defined, we call them primitive entities, and some others may be created when writing a program, we call them objects. The set (a class) of all entities is called **any** and the set (a class also) of all objects is called **object**.

Primitive entities consist of **integers**, **floats**, **symbols**, **strings**, **ports** (streams) and **functions** (cf. Section 2.7). More details about primitive entities may be found in Section 2.7 and in the "CLAIRE Description" (Appendix A).

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



Definition: *A **class** is a generator of objects, which are called its instances. Classes are organized into an inclusion hierarchy (a tree), so a class can also be seen as an extensible set of objects, which is the set of instances of the class itself and all its subclasses. A class has one unique father in the inclusion hierarchy (also called the inheritance hierarchy), called its superclass. It is a subclass of its superclass.*

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, which implies that for any object x, **`x.isa = owner(x).`**

Thus the focus on *entities* in CLAIRE can be summarized as follows: everything is an *entity*, but not everything is an *object*. An entity is described by its owner class, like an object, but objects are "instantiated" from their classes and new instances can be made, while entities are (virtually) already there and their associated (primitive) classes don't need to be instantiated. A corollary is that the list of instances for a primitive class is never available.

2.2 Classes

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 entities. 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**. A named object is an object that has a name, which is a symbol that is used to designate the object and to print it. A named object is usually created with the x :: C() syntax (cf. Section 3.5) but can also be created with new(C, name).

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 and placed 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. The proper management of default values, or their absence through unknown, is a key feature of CLAIRE.

From a set-oriented perspective, a class is the set union of all the instances of its *descendants* (itself, its subclasses, the subclasses of its subclasses, etc.). In some cases, it may be useful to "freeze" the data representation at some point: for this, two mechanisms are offered: *abstract* and *final*. First, a class *c* can be declared to have no instances with *abstract*(*c*) such as in the following:

```
abstract(person)
```

An abstract class is not an empty set, it contains the instances of its descendants. Second, a class can also be declared to have no more new descendants using *final* as follows:

```
final(colors)
```

It is a good practice to declare *final* classes that are leaves in the class hierarchy and that are not meant to receive subclasses in the future. This will enable further optimizations from the compiler. A class may keep or not the set of its instances. When CLAIRE keeps the extension of the class (the set of instances, accessible through the instances set), the class may be used (for instance, iterated) as a set. This is the default behavior for named objects (then the class inherits from **thing**), but not for other classes (that inherit from **object**).To force CLAIRE to maintain the class extension, the class must be declared with **instanced.**

```
action <: object(on:any, performed_by:object)
instanced(action)
```

Implementation Note: in previous versions of CLAIRE (2 & 3), it was the opposite : all classes would maintain their extensions and you had to declare a class as "ephemeral" to disable the extension bookkeeping.

A class definition can be executed only once, even if it is left unchanged. On the other hand, CLAIRE supports the notion of a class *forward* definition. A forward definition contains no slots and no parentheses. It simply tells the position of the class in the class hierarchy. A forward definition must be followed by a complete definition (with the same parent class !) before the class can be instantiated. Attempts to instantiate a class that has been defined only with a forward definition will produce an error. A forward definition is necessary in the case of recursive class definitions. Here is a simple example.

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

Although the *father* of a **child** is a **parent** (in the previous example), creating an instance of **child** does not create an implicit instance of **parent** that would be stored in the *father* slot. Once an instance of child is created, it is your responsibility to fill out the relevant slots of the objects. There exists a way to perform this task automatically, using the *close* method. This method is the CLAIRE equivalent to the notion of a constructor (in a C++ or Java sense). CLAIRE does not support class constructors since its instantiation control structure may be seen as a generic constructor for all classes (cf. Section 3.5). However, there are cases when additional operations must be performed on a newly created object. To take this into account, the *close* method is called automatically when an instantiation is done if a relevant definition is found. Remember that the *close* method must always return the newly create object, since the result of the instantiation is the result of the *close* method. Here is an example that shows how to create a *parent* for each new **child** object :

```
close(x:child) -> (x.father := parent(), x)
```

Slots can be mono- or multi-valued. A *multi-valued* slot contains multiple values that are represented by a set (without duplicates). CLAIRE assumes by default that a slot with range set is multi-valued. However, the multi-valuation is defined at the property level. This is logical, since the difference between a mono-valued and a multi-valued slot only occurs when inversion or rules are concerned, which are both defined at the property level (cf. Section 4.5). This means that CLAIRE cannot accept slots for two classes with the same name and different multi-valuation status. For instance, the following program will cause an error:

```
A <: thing(x:set[integer])        // forces CLAIRE to consider x as multi-valued
B <: thing(x:stack[integer])      // conflict: x cannot be multi-valued
```

On the other hand, it is possible to explicitly tell CLAIRE that a slot with range list or set is mono-valued, as in the following correct example:

```
A <: thing(x:set[integer])
x.multivalued? := false     // x is from A U B  -> (set[integer] U stack[integer])
B <: thing(x:stack[integer])
```

It is sometimes advisable to set up manually the multi-valuation status of the property before creating the slots, in order to make sure that this status cannot be forced by the creation of another class with a mono-valued slot with the same name (this could happen within a many-authors project who share a namespace). This is achieved simply by creating the property explicitly:

```
x :: property(multivalued? = true)  // creates the property
…                  // whatever happens will not change x’s multi-valuation
B <: thing(x:set[integer])          // safe definition of a multi-valued slot
```

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 default method for printing an object takes this parametric definition into account. Objects from a class C are printed as <C>, unless the method **self_print** is defined for C (see Section 6.1). Objects from a parametric class are printed C(..), where the value of the parameters are printed with the parentheses.

We shall see in Section 4 that CLAIRE includes a type system that contains parametric class selections. For instance, the set of real numbers can be defined as a subset of complex with the additional constraint that the imaginary part is 0.0. This is expressed in CLAIRE as follows:

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

In the previous example with stacks, parametric sub-types can be used to designate typed stacks. We can either specify the precise range of the stack (i.e., the value of the *of* parameter) or say that the range must be a sub-type of another type. For instance, the set of stacks with range integer and the set of stacks which contain integers are respectively:

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

2.4 Calls and Slot Access

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 arguments between parentheses. A call is used to invoke a method. Slot accesses follow the usual field access syntax « x.s » where s if the name of the slot. CLAIRE uses generic objects called properties to represent the name of a *method*, used as the selector f of a function call f(...), or a *slot*, used as the selector s in a slot access x.s. In the following example, *eval* is a function and price is a property. Properties and functions are two kinds of relation.

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

Implementation Note:

For upward-compatibility reasons, CLAIRE supports both x.s and s(x) as valid expressions to read the slot s from object x. In CLAIRE 4, x.s is equivalent to read(s,x) and will complain if the value is uknown (unless unknown belongs to the range of s). s(x) is equivalent to get(s,x), it will return unknown without generating an error if no value has been set.

If a slot is read before being defined (its value being unknown), an error is raised. This only occurs if the default value is unknown. To read a slot that may not be defined, one must use s(x) or the get(r:property,x:object) method.

```
John.father          // may provoke an error if John.father 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**). Following a philosophy borrowed from LISP, all values are assimilated to true, except for false, empty lists and empty sets. The special treatment for the empty lists and the empty sets (cf. Conditionals, Section 3.3) yields a simpler programming style when dealing with lists or sets. Notice that in CLAIRE 3.0, contrary to previous releases, there are many empty lists since empty lists can be typed (list<integer>(), list<string>(), … are all different).

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. This is the key for writing parametric methods using the inline capabilities of CLAIRE (cf. Section 4.1). This also means that using *call* is not a safe way to force dynamic binding, this should be done using the property *abstract*. An abstract property is a property that can be re-defined at any time and, therefore, relies on dynamic binding. 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:function, g:function, x:any) => f(g(x))
```

is equivalent to

```
compose(f:function, g:function, x:any) => call(f, call(g,x))
```

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.

```
x.age := 10,  John.father := mary
```

When the assignment 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 function with arity 2 that has been explicitly declared as an operation).

```
x.age :+ 1,  John.friends :add mary, x.price :min 100
```

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

Warning: The next section describes an advanced feature and may be skipped

2.6 Reified Slots

CLAIRE supports the reification of objects' slots. This means that the value of slot, such as x.age, can be an object (with a value) that is used to represent, for instance, modal knowledge about x.age (such as in sure(x.age) = true). This is achieved through the reify declaration:

```
reify(age)
```

A reified slot must have a range which is a class that contains objects which understand the *read* and *write* methods, since the reader will substitute x.age with read(x.age) and x.age := y by write(x.age,y). Such a class is usually called a container class. Reification is the representation of each value pair age(x,y) by a container object (that can contain additional information). Here is an example that is also quite useful. We define the Store container class, which is a defeasible reference to an object, which keeps the world in which the object was last updated. Worlds are explained in Section 5.4.

```
Store[of] <: ephemeral_object(of:type, value:any, world:integer = -1)

self_print(x:Store) -> printf("store(~S)",get(value,x))

write(x:Store<X>,y:X)
-> (if (world?() > x.world)
       (put_store(value,x,y,true), put_store(world,x,world?(),true))
    else x.value := y)]

read(x:Store<X>) : type[X] => x.value
```

We can now use our container class in the following example:

```
A <: thing(x:Store<integer>, y:Store<string>)
reify(x,y)

a :: A(x = Store(integer), y = Store(string))
a.x := 1
(if (a.x > 0) a.y := "positive")
```

Notice how we can use a.x and a.y as if x and y were normal slots, and use get(x,a) and get(y,a) to access the associated container objects. We leave it as an exercise to the reader, once familiar with Section 5, to see why it may be interesting to use these Store objects to reduce the growth of the trailing stack for worlds.

2.7 Primitive entities

Integers in CLAIRE 4 are 64-bits integer. Integers have a common syntax (A more formal CLAIRE syntax is presented in the first appendix):

```
0, 1, 123456, -2013
```

Floats are 64-bits from the host language (Go). Their syntax follows C/Go convention, with the addition of the % macro-caracter, to indicate percentage values.

```
0.123, 3.14159, 12.3e8, -2.12e-13, 20%,  21.2%
```

Characters in CLAIRE are also inherited from the host language (8 bits char). Constants are represented using “’’ as separators; the two special values \n (new line) and \t (tab) are also supported.

```
‘a’, ‘0’, ‘\n’, ‘\t’, char!(123), char!(#/a)
```

Symbols are inherited from LISP, they represent names that are associated to a module (namespace) and hashed for quick retrieval. Symbols may be thought of as names; they can be created dynamically with methods such as **symbol!** or statically with the following syntax:

```
Core/”a symbol”, claire/”class”, m1/”g3”, symbol!(m,”g” /+ “12“), symbol!(claire,string!(12))
```

Strings in CLAIRE are character chains that are not necessarily constants (contrary to Go or new versions of C++); for C programmer, they are close to “char*” type. Strings’s syntax is very classical using “”” as a separator. To find out all the string methods that are supported by CLAIRE, you may enter **methods(string,string)** at the top-level, or look at the 2nd appendix. Strings may be created dynamically using the **make_string(..)** method.

```
“a string”, “a string with \n”, make_string(100,’a’), make_string(list(‘a’,’b’,’c’))
```

Ports are CLAIRE entities that represent ports from the host language. Ports are only created using the **fopen** method that is similar to that of C or C++.

Last, “external functions” are pointers to host language functions that will be linked to the CLAIRE program at compile time (their use is explained in Appendix C). The syntax is the following:

```
function!(pow), function!(make_array)
```

2.8 Inspecting Objects

CLAIRE is a truly reflective language; everything is an entity that belongs to a class and that can be inspected. The method owner applies to any entity and return a class (owner(x) = x.isa if x is an object). The method show is useful to inspect any entity or object:

```
show(“a string”), show(class), show(quote(1 + 2)), show(list(‘a’,’b’,’c’))
```

CLAIRE has a built-in object inspector : `inspect(x:any)` with a simple short-cut : ? x.

```
? class                          // inspect a class
? isa                            // inspect a property
? (+ @ integer)                  // inspect a method
? quote(let x := 1 in (x + 1)    // inspect a CLAIRE expression
```

3. Lists, Sets and Instructions

3.1 Lists, Sets and Tuples

CLAIRE provides two easy means of manipulating collections of objects: sets and lists. Lists are ordered, possibly 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)
```

A major feature of CLAIRE is the fact that lists or sets *may* be typed. This means that each bag (set or list) may have a type slot named *of*, which contains a type to which all members of the list must belong. This type is optional, as is illustrated by the previous examples, where no typing was given for the lists or sets. To designate a type for a new list or a new set, we use a slightly different syntax:

```
list<thing>(a,b,c,d)   list<integer>(1,2 + 3)   list<float>()
set<thing>(a,b,c)   set<integer>(1, 2 + 3)
```

Typing a list or a set is a way to ensure that adding new values to them will not violate typing assumptions, which could happen in earlier versions of CLAIRE. Insertion is now always a destructive operation (add(l,x) returns the list l, that has been augmented with the value x at its end).

Since typing is mandatory in order to assume type-safe updates onto a list or a set, if no type is provided, CLAIRE will forbid any future update: the list or the set is then a "read-only" structure. *This is the major novelty in CLAIRE 3.2*: there is a difference between:

```
list(a,b,c,d)   set(1,2 + 3)   list{i | i in (1 ..2)}
```

which are *read-only* structures, and

```
list<thing>(a,b)   set<integer>(1,2 + 3)  list<integer>{i | i in (1 ..2)}
```

which are structures that can be updated (modified).

List or set types can be arbitrarily complex, to represent complex list types such as list of lists of integers (cf. Section4). However, it is recommended to use a global constant to represent a complex type that is used as a list type, as follows:

```
 MyList :: list<integer>
set<MyList>(list<integer>(1), list<integer>(2,3))
```

Constant sets are valid CLAIRE types and can be built using the following syntax:

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

The expressions inside a constant set expression are not evaluated and should be primitive entities, such as integers or strings, named objects or global constants. Constant sets are constant, which means that inserting a new value is forbidden and will provoke an error.

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. Similarly, one may decide to create a typed or an un-typed list or set, by adding the additional type information between angular brackets. For instance, here are two samples with and without typing:

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

When does one need to add typing information to a list or a set ? A type is needed when new insertions need to be made, for instance when the list or set is meant to be stored in an object's slot which is itself typed.

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<integer>{size(x.slots) | x in class}
```

For example, we have the traditional *average_salary* method:

```
average_salary(s:set[man]) : float -> (sum(list{m.sal | 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 *some* construct: `some(x in a | f(x))`. For instance, we can write:

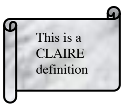
```
exists(x in (1 .. 10) | x > 2)         ;; returns true
some(x in (1 .. 10) | x > 2)           ;; returns 3 in most implementations
exists(x in class | length(x.ancestors) > 10)
```

The difference between *exists* and *some* is that the first always returns a **boolean**, whereas the second returns one of the objects that satisfy the condition (if there exists one) and unknown otherwise. It is very often used in conjunction with **when** (cf. next section), as in the following example:

```
when x := some(x in man | rich?(x)) in
    (borrow_from(x,1000), ...)
else printf("There is no one from whom to borrow! ")
```

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 two following examples returns false (because of 1):

```
forall(x in (1 .. 10) | x > 2)
forall(x in (1 .. n) | exists( y in (1 .. x) | y * y > x))
```



Definition: *A **list** is an ordered collection of objects that is organized into an extensible array, with an indexed access to its members. A list may contain duplicates, which are multiple occurrence of the same object. A **set** is a collection of objects without duplicates and without any user-defined order. The existence of a system-dependent order is language-dependent and should not be abused. The concept of **bag** in CLAIRE is the unifier between lists and sets : a collection of objects with possible duplicates and without order.*

A read-only (untyped) list can also be thought as tuples of values. For upward compatibility reasons, the expression tuple(a_1,…,a_n) is equivalent to list(a_1,…,a_n):

```
tuple(1,2,3),  tuple(1,2.0,"this is heterogeneous")
```

Since it is a read-only list, a tuple cannot be changed once it is created, neither through addition of a new member (using the method *add*) or through the exchange of a given member (using the *nth=* method). CLAIRE offers an associated data type, as explained in Section 4.2. For instance, the following expressions are true:

```
tuple(1,2,3) % tuple(integer,integer,integer)
tuple(1,2,3) % tuple(0 .. 1, 0 .. 10, 0 .. 100)
tuple(1,2.0,"this is heterogeneous") % tuple(any,any,any)
```

Typed tuples are used to return multiple values from a method (cf. Section 4.1). Because a tuple is a bag, it supports membership, iteration and indexed access operations. However, there is yet another data structure in CLAIRE for homogeneous arrays of fixed length, called *arrays*. **Arrays** are similar to lists but their size is fixed once they are created and they must be assigned a subtype (a type for the members of the array) that cannot change. Because of these strong constraints, CLAIRE can provide an implementation that is more efficient (memory usage and access time) than the implementation of bags. However, the use of arrays is considered an advanced feature of CLAIRE since everything that is done with an array may also be done with a list. Arrays are described in Section 3.7.

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, empty sets and empty lists.

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

Local variables can be introduced in a block with the **let** construct. These variables can be typed, but it is **not** mandatory (CLAIRE will use type inference to provide with a reasonable type). On the other hand, unlike languages such as C++, you always must provide an initialization value when you define a variable. 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)
```

Notice that CLAIRE uses **:=** to represent assignment and = to represent equality. The compiler will issue a warning if a statement (x = y) is used where an assignment was probably meant (this is the case when the value of the assignment is not needed, such as in x := 1, y = 3, z := 4).

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
```

The name of a local variable can be any identifier, including the name of an existing object or variable. In that case, the new variable overrides the older definition within the scope of the *let*. While this may prove useful in a few cases, it should be used sparingly since it yields to code that is hard to read. A rule of thumb is to avoid mixing the name of variables and the name of properties since it often produces errors that are hard to catch (the property cannot be accessed any more once a variable with the same name is defined). The control structure *when* is a special form of *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)
```

The default behavior when the value is unknown can be specified using the **else** keyword. The statement following the else keyword will be evaluated and its value will be returned when the value of the local variable is unknown.

```
when f := get(father,x) in printf("his father is ~S\n",f)
else printf("his father is not known at the present time \n")
```

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

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

The tuple of variable is simply introduced as a sequence of variables surrounded by two parentheses. 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 ...
```

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

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

Although this is mostly used for assigning the result of tuple-valued functions without any useless allocation, it is interesting to note that the previous example will be compiled into a nice value-exchange interaction 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 x.r := 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 := x.r in (x.r := y, let result := e in (x.r := old_v, result)
```

CLAIRE provides automatic type inference for variables that are defined in a let so that explicit typing is not necessary in most of the cases. Here are a few rules to help you decide if you need to add an explicit type to your variable or even cast a special type for the value that is assigned to the variable:

(a) Type inference will provide a type to a Let variable **only** if they do not have one already.

(b) when you provide a type in let x:t := y, the compiler will check that the value y belong to t and will issue a warning and/or insert a run-time type-check accordingly.

(c) if you want to force the type that is infered to something smaller than what CLAIRE thinks for y, you must use a cast:

```
let x := (y as t2) in ...
```

To summarize,

- in most cases CLAIRE range inference works, so you write `let x := y in` ...
- you use `let x:t := y` to weaken the type inference, mostly because you want to put something of a different type later,
- you use `let x := (y as t)` to narrow the type inferred by CLAIRE.

3.3 Conditionals

if statements have the usual syntax (if <test> x else y) with implicit nesting (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 false 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 an *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)). The expression on which the switching is performed is usually a variable, but can be any expression. However, it should not produce any side effect since it will be evaluated many times.

Starting with CLAIRE 3.3, only Boolean expressions should be used in the <test> expression of a conditional statement. The implicit coercion of any expression into a Boolean is still supported but should not be used any longer. The compiler will issue a warning if a non-Boolean expression is used in an If.

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 on any *collection*:

```
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 collection here is taken in a very general sense, i.e., 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 user classes that inherit from the collection root. For instance, set!(n:integer) returns the subset of (0 .. 29) that is represented by the integer n taken as a bit-vector. To tell CLAIRE that her new class is a collection, the user must define it as a subclass of **collection**. If x is a *collection*, then

- for z in x
- (z % x)

are supported. When defining a new subclass of collection, the methods *set!* and *%* must be defined for this new class, and it is also advisable to define *size* and *iterate* to get compiler speed-ups (if *size* is not defined, an implicit call to *set!* is made). Other collection handling methods, such as add, delete, etc may be defined freely if needed.

Notice that it is possible that the expression being evaluated inside the loop modifies the set itself, such as in

```
for x in {y in S | P(y)} P(x) := false
```

Because the CLAIRE compiler tries to optimize iteration using lazy evaluation, there is no guarantee about the result of the previous statement. In this case, it is necessary to use an explicit copy as follows:

```
for x in copy({y in S | P(y)}) P(x) := false
```

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 subclass (C) of **collection**. 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 exit 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 false. 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))
```

There is one restriction with the use of break: it cannot be used to escape from a try … catch block. This situation will provoke an error at compile-time.

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 “instanced” classes (a class is “instanced”, that is maintains its extension, if has been declared as such or if it inherits from the *thing* class). The creation of a new instance of a class yields 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)
```

Notice that the identifier used as the name of an object is a constant that cannot be changed. Thus, it is different from creating a global variable (cf. Section 6.4) that would contain an object as in :

```
aGoodGuy:person :: 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)
```

The previously mentioned instantiation form only applies to a parameterized class. It is possible to instantiate a class that is given as a parameter (say, the variable v) using the *new* method. *New*(v) creates an instance of the class v and *new*(v,s) creates a named instance of the class v (assumed to be a subclass of *thing*) with the name s.

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 an "ephemeral" class, so the list of instances is not kept. In fact, raising an exception *e* is achieved 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. Also, as noticed previously, the body of a handler cannot contain a break statement that refers to a loop defined outside the handler.

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,price(x))
```

Another very useful type of exception is *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 (*worlds* are explained in section 5.4) :

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

In fact, this is such a common pattern that CLAIRE provides a special instruction, *branch*(x), which evaluates an expression inside a temporary world and returns a boolean value, while detecting possible contradiction. The statement branch(x) is equivalent to

```
try (    choice(),
         if x true else (backtrack(), false)
catch contradiction (backtrack(),      false)
```

If we want to find a value for the slot x.r among a set x.possible that does not cause a contradiction (through rule propagation) we can simply write :

```
when y := some(y in x.possible | branch(x.r = y)) in x.r := y
else contradiction!()
```

3.7 Arrays

An array can be seen as a fixed-size list, with a *member type* (the slot name is *of*), which tells the type of all the members of the array. Because of the fixed size, the compiler is able to generate faster code than when using lists, so lists should be used when the collection shrinks and grows, and an array may be used otherwise. This is especially true for arrays of floats, which are handled in a special (and efficient) way by the compiler.

Arrays are simpler than lists, and only a few operations are supported. Therefore, more complex operations such as *append* often require a cast to list (list!). An array is created explicitly with the *make_array* property :

```
let l := make_array(10,float,0.0) in
  l[1] := l[3] + l[4]
```

Note that the *of* type must be given explicitly (it can be retrieved with *member_type*(l)), as well as a default value (0.0 in the previous example). An array is printed as [0.0,0.0, …, 0.0], similarly to a list but with surrounding brackets. Operations on arrays are described in the API and include copying, casting a bag into an array (*array!*), defeasible update on arrays using *store*, and returning the length of the array with *length*. An array can also be made from a list using **array!**, which is necessary to create arrays that contain complex objects (such as arrays of arrays). For instance,

```
Matrix :: array!(list<float[]>{ make_array(10,float,0.0) |
                                i in (1 .. 10)})
```

is correct, while the following will not work because the internal one-dimension array will be shared for all columns.

```
Matrix :: make_array(10,float[],make_array(10,float,0.0))
```

Since they are collections, arrays can be iterated, thus all iteration structures (image, selection, ...) can be used.

3.8 Map Sets

Dictionaries are now first-class citizens of CLAIRE. A **map_set** is a set of association pairs (x:y) such that the value of the dictionary for entry x is y. Map sets have been introduced in CLAIRE4 to leverage the Go underlying implementation. They are collections, so they can be iterated.

Map sets are typed, with a domain (to which x belongs) and a range (to which y belongs). The common usage is to create an empty directory using map! :

```
D :: map!(string,integer)
put(D,"toto",4)
get(D,"toto")
```

CLAIRE also supports the explicit creation of a map set from value pairs:

```
MD :: map<string,integer>("toto":4,"ludicrous":9)
set!(MD)   // returns {4,9}
```

4. METHODS AND TYPES

4.1 Methods

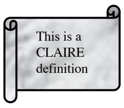
A method is the definition of a property for a given signature. A method is defined by the following pattern :

- 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 (an expression or a let statement introduced by -> or =>).

```
<selector>(<typed parameters>) : <range>opt ->|=>   <body>
```

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

Definition: *A **signature** is a Cartesian product of types that always contains the extension of the function. More precisely, a signature $A_1 \times A_2 \times \dots \times A_n$, also represented as list($A_1,\dots,A_n$) or $A_1 \times A_2 \times \dots \times A_{n-1} \rightarrow A_n$, is associated to a method definition* `f(...) : An -> ...` *for two purposes: it says that the definition of the **property** f is only valid for input arguments (x_1, x_2, ..., x_{n-1}) in $A_1 \times A_2 \times \dots \times A_{n-1}$ and it says that the result of f(x_1, x_2, ..., x_{n-1}) must belong to A_n. The property f is also called an overloaded function and a method m is called its **restriction** to $A_1 \times A_2 \times \dots \times A_{n-1}$.*



If two methods have intersecting signatures and the property is called with an argument list (list of objects) that belongs to 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 {date!@integer, string!@integer, make_string@integer}
```

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

```
loadMM() -> (begin(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). A method's range *must* be declared void if it does not return a value (for instance, if its last statement is, recursively, a call to another method with range void). It is important not to mix restrictions with void range with other regular methods that do return a value, since the compiler will generate an error when compiling a call unless it can guarantee that the void methods will not be used.

The default range was changed to *void* in the version 3.3 of CLAIRE, in an effort to encourage proper typing of methods: "no range" means that the method does not return a value. This is an important change when migrating code from earlier versions of CLAIRE.

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.

CLAIRE also supports functions that return multiple values using **tuples**. If you need a function that returns n values $v_1,v_2,\dots,v_n$ of respective types $t_1,t_2,\dots,t_n$, you simply declare its range as *tuple*($t_1,t_2,\dots,t_n$) and return *tuple*($v_1,v_2,\dots,v_n$) in the body of the function. For instance the following method returns the maximum value of a list and the "regret" which is the difference between the best and the second-best value.

```
[max2(l:list[integer]) : tuple(integer,integer)
   -> let x1 := 1000000000, x2 := 1000000000 in
         (for y in l
```

```
 (if (y < x1) (x2 := x1, x1 := y) else if (y < x2) x2 := y),
tuple(x1,x2)) ]
```

The tuple produced by a tuple-valued method can be used in any way, but the preferred way is to use a tuple-assignment in a let. For instance, here is how we would use the max2 method:

```
let (a,b) := max2(list{f(i) | i in (1 .. 10)}) in …
```

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:

```
lambda[(<typed parameters>), <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 Appendix C. It is important to notice that in CLAIRE, methods can have at most 12 parameters. Methods with 40 or more parameters that exist in some C++ libraries are very hard to maintain. It is advised to use parameter objects in this situation.

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[4]. 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 (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({x.age | 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] The condition for substitution is implementation-dependent. For instance, 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 occurrence of the parameter.

CAVEAT: A recursive macro will cause an endless loop that may be painful to detect and debug.

For upward compatibility reasons (from release 1.0), CLAIRE still supports the use of external brackets around method definitions. The brackets are there to represent boxes around methods (and are pretty-printed as such with advanced printing tools). For instance, one can write:

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

Brackets have been found useful by some users because one can search for the definition of the method m by looking for occurrences of « [m ». They also transform a method definition into a closed syntactical unit that may be easier to manipulate (e.g., cut-and-paste).

When a new property is created, it is most often implicitly with the definition of a new method or a new slot, although a direct instantiation is possible. Each property has an extensibility status that may be one of:

- open, which means that new restrictions may be added at any time. The compiler will generate the proper code so that extensibility is guaranteed.
- undefined, which is the default status under the interpreter, means that the status may evolve to open or to closed in the future.
- closed, which means that no new restriction may be added if it provokes an inheritance conflict with an existing restriction. An inheritance conflict in CLAIRE is properly defined by the non-empty intersection of the two domains (Cartesian products) of the methods.

The compiler will automatically change the status from undefined to closed, unless the status is forced with the abstract declaration:

```
abstract(p)
```

Conversely, the final declaration:

```
final(p)
```

may be used to force the status to closed, in the interpreted mode. Note that these two declarations have obviously an impact on performance: an open property will be compiled with the systematic used of dynamic calls, which ensures the extensibility of the compiled code, but at a price. On the contrary, a final property will enable the compiler to use as much static binding as possible, yielding faster call executions. Notice that the interface(p) declaration has been introduced (cf. Appendix) to support dynamic dispatch in a efficient manner, as long as the property is *uniform*.

4.2 Lambdas

Lambdas are name-less functions that can created dynamically. Lambdas are inherited from LISP and may be created with the "classical" CLAIRE syntax

```
lambda[(x:any), x]
lambda[(x:integer,y:integer), (x + y)]
lambda[(f:lambda,g:lambda), lambda[(x:any), funcall(f,funcall(g,x))]]
```

In CLAIRE4, we have introduced a simpler syntax : (v*){e*}. The variables v may be typed, otherwise the type is assumed to be any

```
(x){x}
(x:ingteger,y:integer){x + y}
(f:lambda,g:lambda){(x){funcall(f,funcall(g,x))}
```

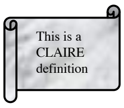
Lambdas may be used with a few classical operators. The main pattern of functional programming language is to pass lambdas as parameters but remember that you can pass a method or a property as a parameter as well in CLAIRE.

```
funcall((x){x * x}),12)   // 144
apply((x:ingteger,y:integer){x + y},list(1,2))   // 3
map((x){x * x},{1,2,4})   // {1,4,16}
```

4.3 Types

CLAIRE uses an extended type system that is built on top of the set of classes. Like 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 attaching methods to complex sets of objects.

Definition: *A (data)* ***type*** *is an expression that represents a set of objects. Types offer a finer-granularity partition of the object world than classes. They are used to describe objects (range of slots), variables and methods (through their signatures). An object that belongs to a type will always belong to the set represented by the type.*



Any class (even parameterized) is a type. A parameterized class type is obtained by filtering a subset of the class parameters with other types to which the parameters must belong. For instance, we saw previously that `complex[im:{0.0}]` is a parametrized type that represent the real number subset of the complex number class. This also applies to typed lists or sets which use the *of* parameter. For instance, `list[of:{integer}]` is the set of list whose of parameter is precisely integer. Since these are common patterns, CLAIRE offers two shortcuts for parameterized type expressions. First, it accepts the expression `C[p = v]` as a shortcut for `C[p:{v}]`. Second, it accepts the expression C<X> as a shortcut for `C[of = X]`. This applies to any class with a type-valued parameter named of; for instance, the stack class defined in Section 2.3. Thus, `stack<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.

Finite constant sets of objects can also be used as types. For example, `{john, jack, mary}` and {1,4,9} are types. Intervals can be used as types; the only kind of intervals supported by CLAIRE 3.0 is integer intervals. Types may also formed using the two intersection (^) and union(U) operations. 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).

Subtypes are also as type expressions. First, because types are also objects, CLAIRE introduces subtype[t] to represent the set of all type expressions that are included in t. This type can be intersected with any other type, but there are two cases which are more useful than other, namely subtypes of the list and set classes. Thus, CLAIRE uses set[t] as a shortcut for set ^ subtype[t] and list[t] as a shortcut for list ^ subtype[t]. Because of the semantics of lists, one may see that list[t] is the union of two kinds of lists:

(a) "read-only" lists (i.e., without type) that contains objects of type t,

(b) typed list from list<X>, where X is a subtype of t.

Therefore, there is a clear difference between

- list<t>, which only contains types lists, whose type parameter (of) must be exactly t.
- list[t], which contains both typed lists and un-typed lists.

Obviously, we have list<t> <= list[t].When should you use one or the other form of typed lists or sets ?

(1) use list[t] to type lists that will only be used by accessing their content. A method that uses l:list[t] in its signature will be polymorphic, but updates on l will rely on dynamic (run-time) typing.

(2) use list<t> to type lists that need to be updated. A method that uses l:list<t> in its signature will be monomorphic (i.e., will not work for l:list<t'> with t' <= t), but updates will be statically type-checked (at compile time).

Last, CLAIRE uses *tuple* and *array* types. The array type t[] represents arrays whose *member type* is t (i.e., all members of the array belong to t). Tuples are used to represent type of tuples in a very simple manner: *tuple*($t_1,t_2,\ldots,t_n$) represents the set of tuples tuple($v_1,v_2,\ldots,v_n$) such that $v_i \in t_i$ for all *i* in (1 .. n). For instance, tuple(integer, char) denotes the set of pair tuples with an integer as first element and a character as second. Also you will notice that tuple(class,any,type) belongs to itself, since class is a class and type is a type.

To summarize, here is the syntax for types expressions in CLAIRE v3.0 :

```
<type> ≡   <class> | <class>[<parameter>:<type>seq] | {<item>seq } |
           (<integer> .. <integer>) |(<type> U <type>) | (<type> ^ <type>) |
           set[<type>] | list[<type>]  | <type>[] | subtype[<type>] |
           tuple(<type>seq)
```

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.

Warning: The next section describes an advanced feature and may be skipped

4.4 Polymorphism

In addition to the traditional "objet-oriented" polymorphism, CLAIRE also offers two forms of parametric polymorphism, which can be 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<X>, y:X) -> ( s.content :add y, s.index :+ 1)
```

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

```
<class>[pi=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. 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+]
```

(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 $e(x_1,...,x_n)$, where $x_1, ..., x_n$ 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 $e(t_1,...,t_n)$ for any types $t_1,...,t_n$ 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 id 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<X>) : type[X] -> s.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 who c belongs (cf. definition of member). This translates into the following CLAIRE definition.

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

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

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

```
<function> (<vi>:<ti>, 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[<et>] | list[<et>] | <et>[] | tuple(<et>^seq) |
          <class>[〈 <var>:<et> | <var>=〈<var>|<const>〉 〉^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>]
```

4.5 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 quite explicit:

```
<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. There is, however, one convenient exception to this rule, which is the casting into a list parametric type. When an untyped list is casted into a typed list, the value of its *of* parameter is actually modified by the interpreter, once the correct typing of all members has been verified. For instance, the two following expressions are equivalent:

```
list<integer>(1,2,3,4)
list(1,2,3,4) as list<integer>
```

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 languages.

```
<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).

A language is type-safe if the compiler can use type inference to check all type constraints (ranges) at compile-time and ensure that there will be no type checking errors at run-time. CLAIRE is not type-safe because it admits expressions for which type inference is not possible such as read(p) + read(p). On the other hand, most expressions in CLAIRE may be statically type-checked and the CLAIRE compiler uses this property to generate code that is very similar to what would be produced with a C++ compiler. A major difference between CLAIRE 3.0 and earlier versions is the fact that lists may be explicitly typed, which removes the problems that could happen earlier with dynamic types. Lists and sets subtypes support inclusion polymorphism, which means that if A is a subtype of B, list[A] is a subtype of list[B]; for instancelist[(0 .. 1)] <= list[integer]. Thus only read operations can be statically type-checked w.r.t. such type information. On the other hand, array subtypes, as well as list or set parametric subtypes, are monomorphic, since A[] is not the set of arrays which contain members of A, but the set of arrays whose member type (the *of* slot) contains the value A. Thus if A is different from B, A[] is not comparable with B[], and list<A> is not comparable with list<B>. This enables the static type-checking of read and write operations on lists. The fact that CLAIRE supports all styles of type disciplines is granted by the combination of a rich dynamic type system coupled with a powerful type inference mechanism within the compiler, and is a key feature of CLAIRE.

4.6 Selectors, Properties and Operations

As we said previously, 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:.

```
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 tables (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[5] and the set associated with an object x is supposed to be the set of values associated with x through the relation. Notice that other aspects of multi-valuation were covered in Section 2.2.

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

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 of inversion 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

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

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`.

Warning: The next section describes an advanced feature and may be skipped

4.7 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 that CLAIRE will replace any occurrence 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 $2^n - 3$ where n is the *index*), 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<any> = list<any>())
set!(x:htable) -> {y in x.arg | known?(y)}
insert(x:htable,y:any)
     -> let l := x.arg in
           (if (x.count >= length(l) / 2)
                (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)))
```

Note that CLAIRE provides a few other functions for hashing that would allow an even simpler, though less self-contained, solution. To iterate over such hash 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 be taken into account in the compiled code unless one uses *oload*, which 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}
```

When defining a restriction of *iterate*, one must not forget the handling of values returned by a *break* statement. In most cases, the code produce by *iterate* is itself a loop (a for or a while), thus this handling is implicit. However, there may be multiples loops, or the final value may be distinct from the loop itself, in which case an explicit handling is necessary. Here is an example taken from class iteration:

```
iterate(x:class,v:Variable,e:any) : any
    => (for %v_1 in x.descendents
            let %v_2 := (for v in %v_1.instances e) in       // catch inner break
              (if %v_2 break(%v_2)))                  // transmit the value
```

Notice that it is always possible to introduce a loop to handle breaks if none are present; we may replace the expression e by:

```
while true (e, break(nil))
```

Last, we need to address the issue of parametric polymorphism, or how to define new kinds of type sets. The previous example of hash-sets is incomplete, because it only describes generic hash-sets that may contain any element. If we want to introduce typed hash-sets, we need to follow these three steps. First we add a type parameter to the **htable** class:

```
htable[of] <: object( of:type = any,  count:integer = 0, ...)
```

Second, we use a parametric signature to use the type parameter appropriately :

```
insert(x:htable<X>,y:X) -> ...
```

Last, we need to tell the compiler that an instance from **htable**[X] only contains objects from X. This is accomplished by extending the *member* function which is used by the compiler to find a valid type for all members of a given set. If x is a type, *member*(x) is a valid type for any y that will belong to a set s of type x. If T is a new type of sets, we may introduce a method *member*(x :T, t :type) that tells how to compute member(t) if t is included in T. For instance, here is a valid definition for our *htable* example:

```
member(x:htable,t:type) -> member(t @ of)
```

This last part may be difficult to grasp (do not worry, this is an advanced feature). First, recall that if t is a type and p a property, (t @ p) is a valid type for *x*.p when *x* is of type t. Suppose that we now have an expression *e*, with type *t1*, that represents a **htable** (thus *t1* <= *htable*). When the compiler calls *member*(t1), the previous method is invoked (x is bound to a system-dependent value that should not be used and t is bound to t1). The first step is to compute (*t1* @ of), which is a type that contains all possible values for y.of, where y is a possible result of evaluating e. Thus, *member*(*t1* @ of) is a type that contains all possible values of y, since they must belong to `y.of` by construction. This type is, therefore, used by the compiler as the type of the element variable v inside the loop generated by *iterate*.

Iteration is equally covered in the section 3.6 of the Appendix C, with the ability to optimize the iteration of specific language expressions. This kind of tuning is outside the scope of regular CLAIRE usage, but is provided to make CLAIRE a great tool to build DSL (Domain Specific Languages).

5. Tables, Rules and Hypothetical Reasoning

5.1 Tables

Named arrays, called tables, can be defined in CLAIRE with the following syntax:

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

The <type> is the range of the table and <expression> is an expression that is used to fill the table. This expression may either be a constant or a function of the variables of the table (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 table may thus be infinite. If the default expression is a function, then the table 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)
```

Notice that the compounded expression x * x is put inside parenthesis because grammar requires a « closed » expression, as for a method (cf. Appendix A). Tables can be accessed through square brackets and can be modified with assignment expressions like for local variables.

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

Tables have been extended in CLAIRE by allowing the use of 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 defining such a table (i.e., an associative array) is, therefore,

```
<table> ≡  <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 a table 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, tables can have an explicit inverse, which is either a property or a table. Notice that this implies that the inverse of a property can be set to a table. However, inverses should only be used for one-dimension array. Thus the inverse management is not carried if the special two-dimension update forms such as « cost[x,y] := 0 » are used.

5.2 Rules

A rule in CLAIRE is made by associating an event condition to an expression. The rule is attached to a set of free variables of given types: each time that an event that matches the condition becomes occurs for a given binding of the variables (i.e., association of one value to each variable), the expression will be evaluated with this binding. The interest of rules is to attach an expression not to a functional call (as with methods) but to an event, with a binding that is more flexible (many rules can be combined for one event) and more incremental.

Definition: *A **rule** is an object that binds a condition to an action, called its conclusion. Each time the condition becomes true for a set of objects because of a new event, the conclusion is executed. The condition is expressed as a logic formula on one or more free variables that represent objects to which the rule applies. The conclusion is a CLAIRE expression that uses the same free variables. An event is an update on these objects, either the change of a slot or a table value, or the instantiation of a class. A rule condition is checked if and only if an event has occurred.*

A novelty in CLAIRE 3.0 is the introduction of event logic. There are two events that can be matched precisely: the update of a slot or a table, and the instantiation of a class. CLAIRE 3.2 use expressions called event pattern to specify which kind of events the rule is associated with. For instance, the expression x.r := y is an event expression that says both that x.r = y and that the last event is actually the update of x.r from a previous value. More precisely, here are the events that are supported:

- x.r := y, where r is a slot of x.
- a[x] := y, where a is a table.
- x.r :add y, where r is a multi-valued slot of x (with range bag).
- a[x] :add y, where a is a multi-valued table.

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

In addition, a new event pattern was introduced in CLAIRE 3.0 to capture the transition from an old to a new value. This is achieved with the expression x.r := (z -> y) which says that the last event is the update of x.r from z to y. For instance, here is the event expression that states that x.salary crossed the 100000 limit:

```
x.salary := (y -> z) & y < 100000 & z >= 100000
```

In CLAIRE 3.2 we introduced the notion of a “pure” event. If a property p has no restrictions, then p(x,y) represents a virtual call to p with parameters x and y. This event may be used in a rule in a way similar to x.p := y, with the difference that it does not correspond to an update. We saw an example in the Sudoku example of our Section 1 tutorial. Virtual events are very generic since one of the parameter may be arbitrarily complex (a list, a set, a tuple …). The event filter associated to a virtual event is simply the expression “p(x,y)”. To create such an event, one simply calls p(x,y), once a rule using such an event has been defined. As a matter of fact, the definition of a rule using p(x,y) as an event pattern will provoke the creation of a generic method p that creates the event.

Virtual event may be used for many purposes. The creation of a virtual event requires neither time nor memory; thus, it is a convenient technique to capture state transition in your object system. For instance, we can create an event signaling the instantiation of a class as follows:

```
 instantiation :: property(domain = myClass, range = string)
[close(x:MyClass) : MyClass -> instantiation(x,date!(1)), x ]
 controlRule() :: rule( instantiation(x,s)
                      => printf(“--- create ~S at ~A \n”,x,s))
```

To define a rule, we must indeed define:

- a condition, which is the combination of an event pattern and a CLAIRE Boolean expression using the same variables
- a conclusion that is preceded by =>.

Here is a classical transitive closure example:

```
r1() ::  rule(
    x.friends :add y
    => for z in y.friend x.friends :add z )
```

Rules are named (for easier debugging) and can use any CLAIRE expression as a conclusion, using the event parameters as variables. Rule triggering can be traced using *trace(if_write)*, as shown in Appendix C. Notice that a rule definition in CLAIRE 3.2 has no parameters; rules with parameters require the presence of the ClaireRules library, which is no longer available.

For instance, let us define the following rule to fill the table *fib* with the Fibonacci sequence.

```
r3() :: rule(
     y := fib[x] & x % (0 .. 100)
   => when z := get(fib,x - 1) in fib[x + 1] := y + z)

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

Warning: CLAIRE 2's logical rules are no longer supported. If you define a rule with arguments "r1(x:<type>,y:<type>) :: rule(…), you will get an error message.

5.3 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 tables x on which hypothetical reasoning will be done need to be specified with the declaration store(x). For instance,

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

Each time we ask CLAIRE to create a new world, CLAIRE saves the status of tables and slots declared with the store command. Worlds are represented with numbers, and creating a new world is done with *choice*(). Returning to the previous world is done with *backtrack()*. Returning to a previous world n is done with *backtrack(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.

Definition: *A **world** is a virtual copy of the* defeasible *part of the object database. The object database (set of slots, tables and global variables) is divided into the defeasible part and the stable part using the* **store** *declaration.* ***Defeasible*** *means that updates performed to a defeasible relation or variable can be undone later; r is defeasible if* store(r) *has been declared. Creating a world (*choice*) means storing the current status of the defeasible database (a delta-storage using the previous world as a reference). Returning to the previous world (*backtrack*) is just restoring the defeasible database to its previously stored state.*

This is a CLAIRE definition

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 *commit* and *commit=* methods. *commit()* 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. *commit=(n)* repeats *commit()* until the current world is *n*. Notice that this "collapse" will simply make the updates that were made in the current world (n) look like they were made in the previous world (n – 1); thus, these updates are still defeasible. A stronger version, *commit0*, is available that consider the updates made in the current world as non-defeasible (as if they belonged to the world with index 0). Thus, unless *commit* is used to return to the initial world (with index 0) – in which case commit and commit0 are equivalent - commit grows the size of the current world since it does not free the stack memory that is used to trail updates.

Last, we have seen in the Sudoku example from the Tutorial and in Section 3.6 the existence of the **branch(X)** control structure which creates "a branch of a search tree" through the use of worlds.

To summarize:

- **choice()** creates a "branching point" (a copy of the stored slots and tables that can be backtracked to).
- **backtrack()** returns to the previously saved world, that is, the value of each slot and stable which has been declared as "defeasible" through the **store(…)** declaration is returned to what it was when **choice()** was invoked.
- **World?()** returns an integer, the number of branches that have been made using **choice()**.
- **commit()** makes all changes made in the current world (n) part of the previous world (n – 1), which becomes the current world.
- **branch(<exp>)** creates a new world, evaluate <exp>, if the result is true returns the **true** Boolean value in the new world, otherwise backtrack to the initial state and returns **false**. A

seen in section 3.6, branch creates a handler that catches the raise of a contradiction, which is interpreted as a failure (hence causes a backtrack and returns **false**).

The amount of memory that is assigned to the management of the world stack is a parameter to CLAIRE, as explained in Appendix C. Defeasible updates are fairly optimized in CLAIRE, with an emphasis on minimal book-keeping to ensure better performance. Roughly speaking, CLAIRE stores a pair of pointers for each defeasible update in the world stack. There are (rare) cases where it may be interesting to record more information to avoid overloading the trailing stack. For instance, trailing information is added to the stack for each update even if the current world has not changed. This strategy is actually faster than using a more sophisticated book-keeping but may yield a world stack overflow. The example of Store, given in Section 2.6, may be used as a template to remedy this problem.

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
store(column, possible)

r1() :: rule(
    column[x] := z => for y in ((1 .. 8) but x) possible[y,z] := false)
r2() :: rule(
  column[x] := z => let d := x + z in
                    for y in (max(1,d - 8) .. min(d - 1, 8))
                        possible[y,d - y] := false )
r3() :: rule(
  column[x] := z => let d := z - x in
                    for y in (max(1,1 - d) .. min(8,8 - d))
                        possible[y,y + d] := false)

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

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* table. 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 table 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 table *possible*, which tells us whether the n-th queen can be set on the p-th line, is filled by means of rules triggered by *column* (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]. If the boolean value is always true, the shorter form *store(l,x,y)* may be used. Here is a typical use of store:

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

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

```
A[i:(1 .. 10)] : tuple(integer,integer,integer) := list<integer>(0,0,0)
(let l := A[x] in
    (l[1] := 3, l[3] := 3))
```

even if *store(A)* is declared, the manipulation on l will not 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), store(l,3,3)))
```

There is another problem with the previous definition. The expression given as a default in a table definition is evaluated only once and the value is stored. Thus the same list<integer>(0,0,0) will be used for all A[x]. In this case, which is a default value that will support side-effects, it is better to introduce an explicit initialization of the table:

```
(for i in (1 .. 10) A[i] := list<integer>(0,0,0))
```

There are two operations that are supported in a defeasible manner: direct replacement of the i-th element of l with y (using *store*(l,i,y)) and adding a new element at the end of the list (using *store*(l,y)). All other operations, such as *nth+* or *nth-* are not defeasible. The addition of a new element is interesting because it either returns a new list or perform a defeasible side-effect. Therefore, one must also make sure that the assignment of the value of *store*(l,x) is also made in a defeasible manner (e.g., placing the value in a defeasible global variable). To perform an operation like nth+ or delete on a list in a defeasible manner, one usually needs to use an explicit copy (to protect the original list) and store the result using a defeasible update (cf. the second update in the next example)

It is also important to notice that the management of defeasible updates is done at the relation level and not the object level. Suppose that we have the following:

```
C1 < : object(a:list<any>, b:integer)
C2 < : thing(c:C1)
store(c,a)
P :: C1()
P.c := C2(a = list<any>(1,2,3) , b = 0)        // defeasible but the C2 object remains
P.c.a :=  delete(copy(P.c.a), 2)               // this is defeasible
P.c.b := 2                                     // not defeasible
```

The first two updates are defeasible but the third is not, because *store*(b) has not been declared. It is also possible to make a defeasible update on a regular property using *put_store*.

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>.Unless toto is a parameterized class, in which case x will be printed as toto(…),where the parenthesis contain the parameters' values.

In the case of bags (sets or lists), strings, symbols or characters, 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 patterns ~S, ~I, ~A and ~F<n><%>. The macro requires as many arguments as there are "tilde patterns" in the string, and pairs in order of appearance arguments together with tildes. These control patterns 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. The ~F pattern is new in CLAIRE 3.4 and takes two additional arguments which are appended to the ~F pattern: a one-digit integer to tell how many digits following the comma should be printed, and % to tell that the float should be printed as a percent. The ~Fn pattern uses the **printFDigit** method (see Appendix B).

A mnemonic is A for alphanumeric, S for standard, I for instruction and F for floats. 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) )
```

Here is an example about how to print a float:

```
Let pi := 3,141592653589 in printf("pi = ~A, ~S, ~F2, ~F% \")
  3.141592635, 3.14159, 3.14, 314,1%
```

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, a memory buffer 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) ) )
```

A CLAIRE port is a wrapper around a stream object from the underlying language (C++ or Java). Therefore, the ontology of ports can be extended easily. In most implementations, ports are available as files, interfaces to the GUI and strings (input and output). To create a string port, you must use *port!*() to create an empty string you may write to, or *port!(s:string)* to read from a string s (cf. Appendix B).

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.

CLAIRE also offers a simple method to redirect the output towards a string port. Two methods are needed 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. You can only use *print_in_string* with one output string at a time; more complex uses require the creation of multiple string ports.

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 sends to the user (which are all implemented with *trace*). When a program is compiled, only the trace statements which verbosity is less than the verbosity level of the compiler (default value is 2, but can be changed with –v) are kept. This means that verbosity levels 1 and 2 are meant to be used with compiled modules and levels 3 and 4 for additional information that only appears under the interpreter. How does one write debug trace statements that can be used in a compiled module ? The proper solution is to use a global variable to represent the verbosity:

```
TALK:integer :: 1
trace(TALK,"Enter the main loop with x = ~S\n",x)
```

By changing the value of TALK, one may turn on and off the printing of these trace statements.

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.

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 insure compatibility with earlier versions, CLAIRE also recognizes lines that begin with ; as comments. Conversely, CLAIRE also supports the C syntax for block comments: anything between /* and */ will be taken as a comment. Comments in CLAIRE may become active comments that behave like trace statements if they begin with [<level>] (see Appendix C, Section 2). The global variable NeedComment may be turned to true (it is false by default) to tell the reader to place any comment found before the definition of a class or a method in the *comment* slot of the associated CLAIRE object.

The second type of special instructions is immediate conditionals. An immediate conditional is defined with the same syntax as a regular conditional but with a #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)
```

Note that the expression can be a block (within parentheses) which is necessary to place a definition (like a rule definition) inside a #if. Last, there exists another pre-processing directive for loading a file within a file: #include(s) loads the file as if it was included in the file in which the #include is read.

There are a few differences between CLAIRE and C++ or Java parsing that need to be underlined:

- Spaces are important since they act as a delimiter. In particular, a space cannot be inserted between a selector and its arguments in a call. Here is a simple example:

  ```
  foo (1,2,3) // this is not correct, one must write foo(1,2,3)
  ```

- = is for equality and := for assignment. This is standard in pseudo-code notations because it is less ambiguous.
- characters such as +, *, -, etc. do not have a special status. This allows the user to use them in a variable name (such as x+y). However, this is not advisable since it is ambiguous for many readers. A consequence is that spaces are needed around operations within arithmetic examples such as:

  ```
  x + (y * z)  // instead of x+y*z which is taken as (one) variable name
  ```

- The character '/' plays a special role for namespace (module) membership.

6.3 Modules

Organizing software into modules is a key aspect of software engineering: modules separate different features as well as different levels of abstraction 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.

> **Definition:** *A **module** is a CLAIRE object that represents a namespace. A **namespace** is a set of identifiers : each identifier (a symbol representing the name of an object) belongs to one unique namespace, but is visible in many namespaces. Namespaces allow the use of the same name for two different objects in two different modules. Modules are organized into a visibility hierarchy so that each symbol defined in a module m is visible in modules that are children of m.*
>
> This is a CLAIRE definition

Identifiers always belong to the namespace in which they are created (*claire* by default). The instruction *module!()* returns the module currently opened. To change to a new module, one may use *begin(m : module)* and *end(m : module)*. The instruction *begin(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

```
begin(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:

```
begin(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 is 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 benefits 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). CLAIRE expects the set of file names to be different from module names, otherwise a confusion may occur at compile time. Here is an example of a module definition, as one would find it in the *init.cl* file:

```
// a simple module that is defined with two files, placed in a common directory
rt1 :: module(part_of = claire,
              source = *src* / "rtmsv0.1",
              uses = list(Reader),
              made_of = list("model","simul"))
```

A last important slot of a module is *uses*, a list of other modules that the new module is allowed to use. This list has two purposes that only exist at compile time. The first one is to restrict the importation of symbols from other modules. A module is considered a legal import if it included itself in this *uses* list, or, recursively, if its father module is legal or if the module is legal for one of the modules in this list. An attempt to read a symbol m/s from a module that is not a legal import will provoke a compiler error. Second, this list is used by the compiler to find out which binaries should be included in the link script that is produced by the compiler.

The usual value is list(Reader), which is the module that contains the CLAIRE run-time environment and that supports the interpreter. It is possible to use list(Core) if the module does not require the interpreter to run, which implies, among other things, that the module contains the main@list method (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
```

Notice that, contrary to languages such as C++, you always must provide an initialization value when you define a global variable (it may be the unknown value). 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
```

The value assigned to a global variable may be made part of a world definition and thus restored by backtracking using *choice*/*backtrack*. It is simply required to declare the variable as a defeasible (i.e., “backtrack”-able) variable using the *store* declaration as for a relation :

```
store(tata)
(tata := 1, choice(), tata := 2, backtrack(), assert(tata = 1))
```

6.5 Conclusion

This concludes the first part of this document. You should now be able to write your first CLAIRE programs and use most original features. There is a lot to be learned from experience, so you should take advantage of the numerous public libraries that are available on the WEB. The rest of the document contains three subparts that you will find useful once you start programming with CLAIRE:

- A precise description of CLAIRE’s syntax
- A description of the API methods, which are the public method in CLAIRE provided by the system.
- A user guide, which describes the features of the programming environment and give you more detailed explanation about the compiler and its options.

APPENDIX A: CLAIRE DESCRIPTION

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

<a>^{seq} denotes a (possibly empty) list of <a> separated by commas
<a>$^{seq+}$ denotes a non empty list of <a> separated by commas
<a>^{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 'l'

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. An identifier should not start with the character #. Each sequence of characters defines one unique identifier, inside a given namespace. Identifiers are used to name objects from **thing**, and unqualified identifiers are used for variables.

```
<ident> ≡ <unqualified identifier> | <qualified identifier>
<qualified identifier> ≡ <identifier>/<unqualified identifier>
<unqualified identifier> ≡ <'a' .. 'Z'><basic character>*
<var> ≡ <unqualified identifier>
```

Implementation note: in CLAIRE, 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 sub-module 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 that 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.

CLAIRE now supports a simple syntax for creating symbols directly, in addition to the various methods provided in the API. Symbols can be entered in the same way that they are printed, by indicating the module (namespace) to which the symbol belongs and the associated string, separated by a « / ».

```
<CLAIRE symbol> ≡ <module>/<string>
```

c. Characters and Strings

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

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

Implementation note: A character is an ASCII representation of an 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, an 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> >   〉
           〈%〉opt
```

Implementation note: in CLAIRE 4, integers and floats use a 64 bits representation.

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>
                        〈,NEW_ALLOC | UPDATE_SLOT | UPDATE_BAG  〉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.

f. Spaces, Lines and Comments

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

```
<separator> ≡     | | [  |  ] |  { |  } |  ( |  ) |  :  |  " |  ' | ; |
                     @ |  SPACE  | EOL |  /
```

Comments may be placed after a '//' on any line of text. Whatever is between a ‘//’ or a ';' and a EOL character is considered as a comment and ignored. Also, the C syntax for block (multiple lines) is supported:

```
<comment> ≡ //'<character - EOL>* EOL |
             ;'<character - EOL>* EOL |
            /*  〈<character> | *<character - />〉* 〈*〉*  */
```

Comments that use ';' are provided for upward compatibility reasons. However, CLAIRE comments defined with //, as in C++, have a special status since they are passed into the code generated by the compiler (for those comments that are defined between blocks). Thus, it may be useful to use « old-fashioned » comments when this behavior is not desirable.

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 <table>.

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> | <declaration 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> |
                  branch( <statement> ) |
                  <comp-exp> | <update>

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

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

<body> ≡ <exp>  |  let  〈 <var def> := <comp-exp>  〉^seq+ in  (<exp>)

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

<event pattern> ≡ 〈 <var>.<property> | <table>[<var>]〉 〈 := | :add 〉^opt  <var>  |
                  〈 <var>.<property> | <table>[<var>]〉 := ( <var> -> <var>)  |
                   <property>(<var>,<var>)
```

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

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

<exp> ≡    <ident> | <set exp> | <fragment> |
           <call> | <slot> | break(<exp>) |
           <table>[<exp>] | <class>(⟨ <property> = <exp> ⟩^seq ) |
           lambda[⟨ <var>:<type>  ⟩^seq, <exp>]
           (⟨ <var>⟨:<type>⟩⟩^opt  ⟩^seq ){⟨<exp>⟩^seq }

<set exp> ≡        set<memtype>^opt(<comp-exp>^seq+) |
                   list<memtype>^opt (<comp-exp>^seq+) |
                   { <const>^seq+ } | <type> |
                   list<memtype>^opt{ <var> in <exp> | <statement> } |
                   ⟨set<memtype>⟩^opt{ <var> in <exp> | <statement> } |
                   list<memtype>^opt{ <exp> | <var> in <exp> } |
                   ⟨ set<memtype> ⟩^opt{ <exp> | <var> in <exp> } |
                   forall(<var> in <exp> | <statement> )  |
                   some(<var> in <exp> | <statement> )  |
                   exists(<var> in <exp> | <statement> )

<memtype> ≡        '<'<type>'>'


<call> ≡ <function> ⟨@ <type> ⟩^opt(<comp-exp>^seq)

<slot> ≡ <exp> .<property>
```

In CLAIRE, function calls are limited to 12 parameters at most. The binary operators (as, :=, | (OR), &(AND), and the <operation> objects) are grouped according the their precedence value (stored as a slot and user-modifiable). Operators with lower precedence values are applied first. Here is the default preference values for CLAIRE binary operators :

as	**:**	**0**
^	**:**	**9**
add, delete, @, %, meet, inherit?, join, <<, >>, and, or, cons, /+, /, *, mod, but	**:**	**10**
+, - , min, max	**:**	**20**
..	**:**	**30**
**U, **	**:**	**50**
=, !=, <, >, <=, >=, less?	**:**	**60**
:=	**:**	**100**
&	**:**	**1000**
|	**:**	**1010**

The typing system is the following

```
<var def> ≡ <var>:<type>

<type> ≡ <class> | <class>[⟨ <var> : <type> ⟩^seq+] | <class><memtype> |
           set[<type>] | list[<type>] | subtype[<type>] | {<const>^seq+ } |
           tuple(<type>^seq+ ) | (<comp-type>)

<comp-type> ≡ <type> U <type> | <type> ^ <type> | <const> .. <const>

<const> ≡  <integer> | <float> | <named object> | <string> | <char> |
```

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

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

<range> ≡<type> | type[<exp>]
```

The condition language used for rules (to describe the event and the boolean condition to which a rule is attached) is defined as follows.

```
<rule condition> ≡ <event pattern> ⟨& <exp>⟩opt
<event patter> ≡ <rel-exp> := <var> |
                 <rel-exp> := (<var> -> <var>) |
                 <property>(<var>,<var>)

<rel-exp> ≡ <var>.<property>  | <table>[<var>]
```

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. Methods that have a unified semantic and multiple implementations (e.g., self_print) may be abstracted into a single method.

^ **Kernel** **method**

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

(x ^ 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 ^ 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 ^ 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 **Core** **method**

```
^2(x:integer) → integer
```

^2(x) returns 2^x

% **Kernel** **method**

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

(x % y) returns (x ∈ y) for any entity x and any abstract set y. An abstract set is an object that represents a set, which is a type or a list. Note that membership to a static type is actually “diet”.

***** **Kernel** **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!).

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

/ **Kernel** **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!).

+ **Kernel** **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 an integer, otherwise an error is raised (explicit conversion is supported with float!).

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

- **Kernel** **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) returns the opposite of x.

/+ — Kernel — method

```
x:list /+ y:list → list
x:string /+ y:string → string
x:symbol /+ y:(string ∪ 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. It is not represented with + because it is not commutative. 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.

.. , -- — Kernel — method

```
..  x:integer .. y:integer → Type
--  x:integer .. y:integer → Interval
```

(x .. y) returns the interval {z | x ≤ z ≤ y}. Intervals are only supported for integers, in CLAIRE v3.0. Notice that (3 .. 1) returns the empty set, which is a type. The new method (x – y) is an explicit interval constructor (it produces an error if the first argument is larger than the second).The result is an object from the class *Interval*, which is a type.

=, != — Kernel — 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? — Core — method

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

returns true if x and y denote the same type. For example `=type?(boolean, {true, false})` returns true because `final(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 set equality will hold ever after.

<=, >=, <, > — Kernel — 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.

<< , >> — Kernel — method

```
l:list << n:integer → list
x:integer << n:integer → integer
x:integer >> n:integer → integer
l:string << n:integer → string
```

(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. (x <<n) and (x >> n) are the result of shifting the integer x seen as a bitvector respectively to the left and to the right by n positions.

(s << n) removes the n first characters of a string s. This is an efficient but destructive operation (no allocation, but the initial string is lost).

@ Core method

```
p:property @ t:type → entity
p:property @ l:list[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>(...)`.

(p @ list(t1,..tn)) is similar and returns the restriction of p that applies to arguments in t1 ×… × tn.

(t @ p) returns the type that is inferred for `x.p` when `x` is an object of type t and p a parameter (read-only property).

abs Core method

```
abs(x:integer) → integer
abs (x:float) → float
```

abs(x) returns the absolute value (-(x) is x is negatibe, x otherwise).

abstract Core method

```
abstract(c:class) → void
abstract(p:property) → void
```

abstract(c) forbids the class c to have any instance. abstract(p) 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.

active? Compile slot

```
compiler.active? → boolean
```

This boolean is set to true when the compiler is active (i.e., compiling CLAIRE code into C++ code). This is useful to introduce variants between the compiled and interpreted code (such as different sizes). Note that there is another flag, loading?, to see if a file is loaded by the compiler.

add Kernel method

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

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. The pseudo-destructive behavior of *add* is similar to that of *add**, which is described below.

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* Kernel 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.

and Kernel method

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

and(x,y) returns the bitwise intersection of two integers (seen as bitvectors).

apply Core 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.

array! — Kernel — method

```
array!(x:list,t:type) → type[t[]]
```

creates a copy of the list x that is represented as an array. The member type must be given as a parameter t and an error will occur if a member of the list does not belong to t..

arg1 / arg2 — Kernel — method

```
arg1(x:Interval) → any
arg2(x:Interval) → any
```

These slots contain respectively the minimal and maximal element of a CLAIRE interval..

begin — Kernel — method

```
begin(m:module) → void
```

sets the current namespace to m (a module).

but — Core — method

```
but(s:any,x:any) → any
```

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

car, cdr — Kernel — 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 — Kernel — method

```
call(p:property, l:listargs) → any
call(x:lambda, l:listargs) → any
```

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

cast! — Kernel — method

```
cast!(s:bag,t:type) → bag
```

cast(s,t) sets the member type of the bag s to t. This is a system method, that should not be used lightly since it does not perform any check and may yield nasty errors. The proper way to cast a bag is to use “as”: (s as t).

char! — Kernel — method

```
char!(n:integer) → char
```

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

class! — Core — 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 — Core — method

```
close(m:module) → module
close(c:class) → class
close(e:exception) → any
close(v:global_variable) → global_variable
```

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 **Kernel** **method**

```
cons(x:any, l:list) → list
```

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

contradiction!() **Kernel** **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 **Kernel** **method**

```
copy(x:object) → object
copy(s:bag) → bag
copy(a:array) → array
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.

cos **Kernel** **method**

```
cos(x:float) → float
```

cos(x) returns the cosine of x (x is expressed in radians).

date! **Kernel** **method**

```
date!(i:integer) → string
```

date!(i) returns the date, using the integer parameter i to indicate whether the full date is needed or only the day or the time. For instance

```
date!(0) = "Thu Mar  9 08:04:22 2000"
date!(1) = "Thu Mar  9 2000"
date!(2) = "08:04:22"
```

delete **Kernel** **method**

```
delete(p:relation, x:object, y:any) → any
delete(s:bag, x:any) → bag
```

delete(s,x) returns s if x is not in s and the list (resp. set) s without the first (resp. only) occurrence of 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. The proper way to use delete, therefore, is either destructive (*l :delete x*) or non-destructive (*delete(copy(l),x)*).

descendents **Core** **slot**

```
descendents(x:class) → set[class]
```

For a class c, c.*descendents* is the set all classes that are under c in the hierarchy (transitive closure of the subclass relation).

difference **Kernel** **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.

domain **Core** **slot**

```
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, r.*domain* is the list formed by the types of the parameters required by the method. For a relation r, r.*domain* is the type on which r is defined.

end_of_string — Kernel — 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().

erase — Kernel — method

```
erase(a:table) → any
erase(r:property,x:any) → any
```

erase(a) removes all value pairs contained in the table. This means that, on one hand, the value a[x] becomes unknown for each object x, and also that any references to an object from the table's domain or an associated value is lost, which may be useful to allow for complete garbage collection.

erase(p,x) removes the value associated to x with the property p. The default value, or the unknown value, is placed in the slot x.p, and the inverse if updated (if any).

exception! — Kernel — method

```
exception!() → exception
```

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

exit — Kernel — method

```
exit(n:integer) → void
```

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

factor? — Kernel — method

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

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

fcall — Core — method

```
fcall(f:external_function, s1:class, x:any, s:class) →      any
fcall(f:external_function, s1:class, x:any, s2:class, y:any, s:class) →     any
fcall(f:external_function, s1:class, x:any, s2:class, y:any,
        s3:class,z :class,s :class) →       any
```

fcall provide an easy interface with external (C++) functions. *fcall*(f,s1,x,s) applies an external function to an argument of sort s1. The sort of the returned value must be passed as an argument (cf. Appendix C). . *fcall*(f,s1,x,s2,y,s) is the equivalent method in the two-arguments case.

final — Core — method

```
final (c:class) → void
final (p:property) → void
```

final(c) forbids the user to create any subclass of the class c. If c is a constant class, this is taken as a "diet" compiling directive.

final(p) change the extensibility status of the property p (represented with the slot open) so that the property p becomes closed, which means that a new restriction may no longer be added if it causes an inheritance conflict.

finite? — Core — 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! — Kernel — method

```
float!(x:integer) → float
```

```
float!(x:string) → float
```

transforms an integer or a string into a float.

flush Kernel 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 Kernel 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 Kernel 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 Core slot

```
formula(m:method) → lambda          Core
formula(d:demon) → lambda           Core
```

formula gives the formula associated with the method/demon.

funcall Core slot

```
funcall(m:method, x:any) → any
funcall(m:method, x:any, y:any) → any
funcall(m:method, x:any, y:any, z:any) → any
funcall(f:function, x:any, cx:class, crange:class) → any
funcall(f:function, x:any, cx:class, y:any, cy:class, crange:class) → any
funcall(f:function, x:any, y:any, cy:class, z:any, cz:class, crange:class) → any
```

funcall provide an easy interface with external (C++) functions. *funcall*(f,s1,x,s) applies an external function to an argument of sort s1. The sort of the returned value must be passed as an argument (cf. Appendix C). *funcall*(f,s1,x,s2,y,s) is the equivalent method in the two-arguments case, and *funcall*(f,s1,x,s2,y,s3,y, s) is the equivalent method in the three-arguments case. Notice that the LAST argument is the sort of the result, and that giving an erroneous sort argument will likely produce a fatal error.

funcall also applies a method or a lambda to one or two arguments.

Last, *funcall* may be applied directly to a **function**, that is a primitive entity that represents a C++ function. This method is provided for expert users, since it is a system method that requires the type of each arguments (cx,cy, …) and the type of the return value (crange), which must be provided as classes. Failure to provide the proper sort (i.e., this type information that is usually found in the *srange* slot of the method) will provoke a system failure.

gc Kernel method

```
gc() → any
```

gc() forces a garbage collection to take place

gensym Kernel method

```
gensym() → symbol
gensym(s:string) → symbol
```

gensym() generates randomly a new symbol. *gensym(s)* generates randomly a new symbol that begin with s.

get Kernel method

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

get(p,x) is equivalent to *p(x)*, but without any verification on *unknown*. So does *get(a,x)* for a table.*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))*

get_module — Core, Optimize — method

```
get_module(s:symbol) → module                   Core
get_module(x:thing) → module                    Optimize
```

get_module returns the module where the identifier s was created.

get_value — Kernel — method

```
get_value(s:string) → any
get_value(m:module, s:string) → any
```

returns the object whose name corresponds to the string; if a module argument is passed, the associated symbol is sought in the module's namespace, otherwise the module `claire` is used by default. To find the value associated to a string within the current module, simply use `get_value(module!(),s)`.

getc — Kernel — method

```
getc(p:port) → char
```

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

getenv — Kernel — 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 — Kernel — method

```
hash(l:list,x:any) → integer
hash(n:integer,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 used to be the basis of the table implementation before version 4.

Id — Kernel — method

```
Id(x:any) → type[x]
```

Id(x) returns x. Id has a special behavior when compiled which makes it useful. The argument is evaluated before being compiled. The intended use is with global variables: the compiler uses the actual value of the variable instead of a reference to the global variable. This is very convenient to introduce parameters that are defined outside the module that is being compiled.

This is also used to tell the compiler that an iteration should make explicit use of all iterations rules that may apply to some subclasses of the set expression that is being iterated.

inherit? — Core — method

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

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

instances — Kernel — 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).

instanced — Core — method

```
instanced(c:class) → void
```

instanced(c) tells CLAIRE to maintain the list of instances (e.g, c.instances). This is not necessary if c inherits from things, but otherwise, CLAIRE will assume by default that the extension is not kept. This choice has a strong impact:

- if the extension is kept, the class may be used as a set but objects will not be garbage collected (explicit kill is necessary)
- otherwise, the class cannot be enumerated (like in “for c in class show(c)”) but garbage collection is implicit (memory is reclaimed as soon as the object is no longer used).

integer! **Kernel** **method**

```
integer!(s:string) → integer
integer!(f:float) → integer
integer!(c:char) → integer
integer!(l:set[(0 .. 29)]) → integer
integer!(s:symbol) → 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. Last, *integer(s)* returns a unique index associated to a symbol s.

inverse **Kernel** **slot**

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

r.*inverse* 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)).

invert **Core** **method**

```
invert(r:relation,x:any) → any
```

invert(r,x) return $r^{-1}(x)$ assuming that r has an inverse.

isa **Core** **slot**

```
isa(x:object) → class
```

returns the class of which x is an instance.

kill, kill! **Kernel** **method**

```
kill(x:object) → any                Kernel
kill(x:class) → any                 Kernel
kill!(x:any) → any                          Kernel
```

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? **Kernel** **method**

```
known?(p:relation, 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 **Kernel** **method**

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

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

length **Kernel** **method**

```
length(l:bag) → integer
length(a:array) → integer
length(l:string) → integer
```

returns the length of an array, a bag or a string. The length of a list is not its *size* ! The following is true: length(set!(l)) = size(l) = size(set!(l)).

list! **Kernel** **method**

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

For any array or set x, *list!(s)* transforms x into a list. If x is a set, the order of the elements in the list can be anything.

load, sload, oload, eload — Reader — method

```
load(s:string) → any
sload(s:string) → any
oload(s:string) → any
eload(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. *eload(s)* is similar to *load(s)* but assumes that the file only contains expressions (such as f(1,2)). This is convenient for loading data input files using a functional format.

loading? — Compile — slot

```
compiler.loading? → boolean
```

This boolean is set to true when the compiler is loading a file before compiling it. This is useful to introduce variants between the compiled and interpreted code (see also the *active*? flag)

log — Kernel — method

```
log(x:float) → float
```

computes log(x) – base *e*.

made_of — Kernel — slot

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

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

make_array — Kernel — method

```
make_array(n:integer,t:type,x:any) → type[x[]]
```

returns an array of length n filled with x. The parameter t is the member_type of the array, thus x must belong to t, as well as any future value that will be put in the array. Note that x is shared for all members of the array, which cause a problem if updates can be performed (for instance if x is iyts

make_list — Kernel — 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<any>(0,0,0)). This is a typed list with member type any, thus it can be updated.

make_string — Kernel — method

```
make_string(i:integer, c:char) → string
make_string(s:symbol) → string
make_string(l:list) → 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").

make_string(l) creates a string from the list of its characters.

member — Core — method

```
member(x:type) → 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.

member_type **Kernel** **method**

```
member_type(x:array) → type
```

member_type(x) returns the type of all members of the array x. Therefore, member(a) = member_type(a) for an array a.

methods **Reader** **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 **Core** **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 **Kernel** **method**

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

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

module! **Core, Optimize** **method**

```
module!() → module
```

```
module!(r:restriction) → module
```

module!(r) returns the module where the method r was created.

module!() (= system.module!) returns the current module, that is the module into which the reader is currently reading..

new **Core** **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 **Kernel** **method**

```
not(x:any) → boolean
```

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

nth, nth=, nth+, nth- **Kernel** **method**

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

```
nth_put(l:string, i:integer, x:char) → string       Kernel
nth_get(l:string, i:integer) → string               Kernel
```

nth is used for accessing elements of structured data: nth(l,i) is the i^{th} element of the bag l, nth(s,i) is the i^{th} character of the string s. For tables, 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 i^{th} 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 i^{th} 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 i^{th} 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 i^{th} place has been erased.

Strings in CLAIRE can be used as buffers (arrays of characters) using the methods *nth_get* and *nth_put* that do not perform bound checking. The string does not need to be terminated by a null character and any position may be accessed. This use of strings may provoke severe errors since there are no bound checks, thus it should be used scarcely and with a lot of care.

occurrence — Language — method

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

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

open — Core — slot

```
open(c:class) → integer
open(r:relation) → integer
```

x.open is a slot that tells the extensibility level of the class or relation x.

For a class, there are 6 values: -1 (system.close) means that the class cannot be extended neither with instances nor subclasses; 0 (abstract) means that the class cannot have any instances; 1 (final) means that no new subclasses could be created; 2 (default) is the default status, 3 (system.open) means that the class is explicitly casted as extensible; 4 (ephemeral) says that the class is a subset of ephemeral_object (the list of instances is not maintained). Section 2.2 shows how to define the open status of a class using the proper declarations.

For a relation: open = 1 means that some of the restrictions have been compiled, hence no conflicting new restriction definition is allowed (cf. section 4.1 : extensibility status = closed); open = 2 means undefined; open = 3 means that the extensibility status is “open”, that new restriction may be defined or re-defined at any time.

or — Kernel — method

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

or(x,y) returns the bitwise union of two integers (seen as bitvectors).

owner — Kernel — 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*).

parts, part_of, — Kernel — slot

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

m.part_of 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).

port! — Kernel — method

```
port!() → port
port!(s:string) → port
```

creates a port that is bound to a string. The first method creates an empty string port that is used for writing. The value of the string associated with the port may be retrieved with the method `string!(p:port)`. The second method transforms an existing string into a port that can be read. This is useful to read an expression stored in a string, although the simpler method `read(s:string)` is most often enough for the task.

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 indented (the backquote here is to prevent the instruction from begin evaluated).

princ, print — Kernel — method

```
princ(x:integer) → void
princ(x:string)   → void
princ(x:char)     → void
princ(x:symbol)  → void
princ(x:bag) → 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 — Kernel — 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.

put — Kernel — method

```
put(p:property, x:object, y:any) → any
put(a:table, 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.

put_store — Kernel — method

```
put_store(r1: relation, x:any, v:any,b:boolean) → void
```

put_store(r,x,v,b) is equivalent to put(r,x,v) but also stores this update in the current world if b is true. The difference with the use of the statement store(p) is that put_store allows the user to control precisely which update should be backtracked. Put_store(r,x,y,b) does nothing if r(x) = y.

putc — Kernel — method

```
putc(c:char, p:port) → void
```

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

random, random! — Kernel — method

```
random(n:integer) → integer
random (n:integer,m:integer) → integer
random (b:boolean) → boolean
random (l:bag) → any
random!(n:integer) → void
```

random(n) returns an integer in (0 .. n-1), supposedly with uniform probability. *random(n,m)* returns an integer between *n* and *m*. *random(b:Boolean)* returns a random boolean (true or false) is b is true, and false otherwise. *random(l:bag)*returns a random member of the bag *l*. *random!(n)* resets the seed for the random number generation process.

range — Kernel, Language — method

```
range(r:restriction) → any              Kernel
range(r:relation) → any                 Kernel
range(v:global_variable) → any          Kernel
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 Kernel, Reader method

```
read(p:property, x:object) → any                    Kernel
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*.

release Core method

```
release() → string
```

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

restrictions Kernel 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).

safe Optimize method

```
safe(x:any) → any
```

safe(x) is semantically equivalent to x and is ignored by the interpreter (x = safe(x)). On the other hand, this tells the compiler that the expression x must be compiled with the safe setting of the optimizing options. This is useful when a complete program requires high optimization settings for performance reasons but you still want to ensure that (say) overflow errors will be detected. A typical use would be

```
try safe( x * y) catch error MAXINT
```

to implement a bounded multiplication that can be placed in an optimized module.

self_print Kernel 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! Core method

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

set!(s) returns an enumeration of the collection 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 $(x / 2^i) \bmod 2 = 1$. This method considers x as the bitvector representation of a subset of (0 .. 29). The inverse is integer!.

shell Kernel 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 Kernel 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.

sin Kernel method

```
sin(x:float) → float
```

sin(x) returns the sine of x (x is expressed in radians).

size **Core** **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 **Kernel** **method**

```
slots(c:class) → any
```

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

sort **Core** **method**

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

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)).

In CLAIRE 3, the compiler is able to "macroexpand" the definition of sort (using a *quicksort* algorithm) when the method is a constant and when the call to sort is used to define a single-instruction method that sorts a given list (with a void range). If we define:

```
SortByf(l:list<myObject>) : void -> sort(myOrder @ myObject, l)
```

The compiler will produce a very efficient implementation for this method through code generation, which is not a trivial feature since quicksort is doubly recursive. Notice that this optimization will only take place if:

- the sort(…) message is the unique instruction of the method, which must return nothing
- the sorting method is an expression of the kind (p @ class)
- the list argument is the unique argument of the method

sqr **Kernel** **method**

```
sqr(x:integer) → integer
sqr(x:float) → float
```

returns the square of x, that is x * x.

sqrt **Kernel** **method**

```
sqrt(x:float) → float
```

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

stat **Kernel** **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). If the verbosity is more than 5, stat() produces a more detailed report about the way memory is used in CLAIRE.

store **Kernel** **method**

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

store(r1,r2,...) declares the relations (properties or tables) as defeasible (using the world mechanism). If x is an array or a list, *store(x,n,v,b)* is equivalent to x[n] := v but also stores this update in the current world if b is true. As a syntactical convenience, the argument b may be omitted if it is true. Note that there is a similar method for properties called *put_store*. *store(v)* can be used to declare a global_variable v as defeasible (notice that only one argument is allowed).

string! **Kernel** **method**

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

string! converts a symbol, an integer or a float 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 **Kernel** **method**

```
substring(s:string, i:integer, j:integer) → string
substring(s1:string, s2:string, b:boolean) → integer
```

substring(s,i,j) returns the substring of s starting at the i^{th} character and ending at the j^{th}. 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). *substring*(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). When s2 cannot be identified with any subsequence of s1, the returned value is 0.

symbol! **Kernel** **method**

```
symbol!(s:string) → symbol
symbol!(s:string, m:module) → symbol
```

symbol!(s) returns the symbol associated to s in the *claire* module. For example, *symbol*!("toto") returns claire/«toto». *symbol*!(s,m) returns the symbol associated to s in the module m.

time_get, time_set, time_show, time_read **Kernel** **method**

```
time_get() → integer
time_read() → 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*(). *time_read*() can be used to read the value of the time counter without stopping it.

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 **Core** **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.

uniform? **Core** **method**

```
uniform?(p:property) → boolean
```

Tells if a property is uniform, that is contains only methods as restrictions, with the same types for arguments and ranges. Note that interface properties should be uniform, as well as all properties that are used dynamically in a "diet" program.

use_as_output **Kernel** **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.

vars **Kernel** **slot**

```
system.vars → list[string]
```

system.*vars* contains the list of arguments passed on the shell command line (list of strings).

verbose **Kernel** **slot**

```
system.verbose → integer
```

verbose(system) (also *verbose*()) is the verbosity level that can be changed. Note that trace(i:integer) sets this slot to i.

version **Kernel** **slot**

```
system.version → float
compiler.version → float
```

the version if a float number (<version>.<revision>) that is part of the release number.

world?, commit,choice, backtrack **Kernel** **method**

```
world?() → integer
choice() → void
backtrack() → void
backtrack(n:integer) → void
commit() → void
backtrack0() → void
commit(n:integer) → 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; *choice*() creates a new world and steps into it; *backtrack*() pops the current world and returns to the previous one; *backtrack*(n) returns to the world numbered with n, and pops all the intermediary worlds. The last three methods have a different behavior since they are used to return to a previous world *without* forgetting what was asserted in the current world. The method *commit*() returns to the previous world but carries the updates that were made within the current world; these updates now belong to the previous world and another call to *backtrack*() would undo them. On the other hand, *backtrack0*() also return to the previous world, but the updates from the current world are permanently confirmed, as if they would belong to the world with index 0, which cannot be undone. Last, *commit*(n) returns to the world numbered with n through successive applications of *commit*().

write **Core** **method**

```
write(p:property, x:object, y:any) → 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. An expression is entered, followed by <enter> keystroke. 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(0).

```
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>⟩opt
          ⟨-n | -v  <integer> | -f  <file> | -m <module> | -mx <module> |⟩*
          ⟨ -S <flag>⟩* ⟨ -D | -O | -O2 ⟩opt ⟨ -p ⟩opt ⟨ -safe ⟩opt
          ⟨ -od <dir> ⟩opt
          ⟨ ⟨ ⟨⟨-cm | -cc | -cx  | -sf ⟩ <module>⟩ | -cx <file>⟩ ⟨-o <file>⟩opt ⟩opt
```

Note that `claire ?` or `claire –help` will produce a summary of all the options and their meaning, as follows:

```
options    -s <int> <int> : set memory allocation size
           -f <filename>  : load <filename>
           -n : do not load the init file
           -m <module> : load <module>
           -mx <module> : load <module> and call main()
           -v <int> : upgrade the verbosity level
           -S <flag> : sets the global variable <flag> to true
           -o <name> : sets the name of the executable
           -od <name> : sets the output directory
           -p : profiling mode
           -D : debug mode
           -safe : safe mode
           -O : optimizing mode
           -O2 : optimizing mode
           -cm <module>: compiles a module -> executable
           -cx <module>: compiles a module when main() is launched
           -cc <module>: compiles a module -> target files
           -sf <module> : generates a system file associated to a module
           -sx <module> : generates a system file that includes main()
```

The -s option allows changing the size of the stack memory zone allocated for CLAIRE. There are three stacks that are allocated : the evaluation stack, the “world” stack (management of defeasible updates) and the string buffer zone. The method *stat()* is useful to find out if you need larger stacks for your application.

Whenever CLAIRE starts, it looks for the *init.cl* file in the current directory. This file is loaded before any other action is started.

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 `load(test)`) . The option –mx is used to tell CLAIRE that the main() method should be launched (it implies that the method must have been defined). This option is useful to create test scripts.

The option -S 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. The options –od is used to designate the output directory (i.e., where the Go code generated by CLAIRE will be produced).

There are five options that invoke the CLAIRE compiler: -cm, -cx, -cc, -sf and -sx. They are used to generate respectively a (configuration) system file (-sf and -sx) or a module (3modes). The presence of “x” means that the method main() will be added to the executable files. The -o option may be used to give a new name to the executable that is generated (if any). The options -O, -O2 and -D are used respectively to increase the optimization or the debugging level (cf. Section 3). The option –safe resets the optimizing level to 1, which is safe for most applications.

The -cc option is the lightest compiling strategy for a module: `claire -cc m` will produce a Go file for each CLAIRE file in `m.made_of`. It does not produce system file and assumes that the user want to keep a complete control over the generation of the executable.

The easier way to use the compiler is the -cm option which produces an executable <module>.exe from a module. In addition to producing the go files associated to m, it produces a system file (that links all necessary modules) and calls the Go compiler to build an executable that includes the interpreter. For most users, `claire –cm` is the only option that they need to know. The -mx option is similar, with the addition of a call to the `main()` method. Notice that args that are added in the shell command (as in `m arg1 arg2`) may be found in `args()`.

Last, The option –sf is used to generate the system file separately and is aimed at serious CLAIRE developers (usually, this is called in Makefile, where separating -cc and -sf gives more flexibility). As you may have guessed, -sx does exactly the same but includes a call to `main()`.

The option –p tells the compiler to generate code that is instrumented for the Go profiler. The execution of the compiled module m will generate a profiler file : m.prof, that can be examined with any profiler tool, such as `go tools pprof m.prof`.

Changes from CLAIRE 3.4 to CLAIRE 4.0

CLAIRE 4.0 is a brand-new version of CLAIRE based on Go (the golang programming language) with new levels of robustness and performance. There are also a few changes:

- integer and float-ranged slot must have a default value.
- By default, all classes but subclasses of “thing” are “ephemeral” in CLAIRE 3 vocabulary: they do not keep their extensions (`c.instances` is not maintained).
- New syntax (syntactic sugar) for lambdas.
- Dictionaries are introduced as `map_set`.
- C++ extensibility through “interfaces” and “import” has been discontinued.

Changes from CLAIRE 3.3 to CLAIRE 3.4

CLAIRE 3.4 is the 20-th anniversary version of CLAIRE ☺. It adds a few useful features to CLAIRE but its main goal is the migration towards newer development and code sharing tools, as well as the port onto distributed environments such as cloud computing. Here is a short list of what’s new:

- The square (x^2) method is introduced for integers and floats; same for *abs* (absolute value).
- Trigonometry functions *sin* and *cos* for floats.
- Random has new methods on integer × integer, Boolean and bag.
- Printf has a new ~F pattern with either #digit or % as an option
- Percent (%) macro-character is allowed
- The class `measure` has been introduced. A measure is a small object that may record a series of float value. Each measure object is uniquely identified with an index (integer). It is created with a regular instantiation `measure().` A measure object is used through the following methods:
  a. `add(`m:measure,v:float) records the value and returns the object m
  b. `mean(`m:measure) returns the mean value of the serie
  c. `stdev(`m:measure) returns the standard deviation of the serie
  d. `logMeasure(`s:string) creates a file with name s, that contains all the measure objects from the current CLAIRE program. This file can be reloaded later, using `load(s)` command, as long as the measure objects exist. Loading the log file will add the stored series to the existing ones.

Changes from CLAIRE 3.2 to CLAIRE 3.3

The list of other changes from 3.2 is as follows.

- sort(<method>,<list>) is macro-expanded by the compiler using a quicksort algorithmic pattern, when sort(…) is used to define a method as in the following example:
  sortByValue(l:list<Task>) : list<Task> -> sort(byValue @ Task, l)
- The compiler enforces the Claire 3.3 syntax and issues a warning when an If statement is found which test expression does not return a Boolean, and when an equality expression is found which value is not used (probably meant as an assignment)
- The default range for a method without range declaration is void. This small change may cause a lot of trouble when the user does not usually provide a correct range for her methods. The CLAIRE compiler is now more strict when checking that void values are not wrongly used in expressions (compiler error # 205).
- The compiler is able to perform type inference and type checking on *for* and *while* statements that use a *break(x)* expression to return a value. Adding a value to a list is also better type-checked.

Changes from CLAIRE 3.0 to CLAIRE 3.2

The key change is the fact that types list[t] may apply to untyped list.

* Lists now exist in two flavors: read-only untyped lists and typed lists, which support (safe) updates.

* Propagation rules have been simplified dramatically. They are now reduced to simple event-propagation rules, but they are a standard feature of the CLAIRE language, as opposed to an external library, which was the case for version 3.0.

* The debugger now checks the range of the method for each call, a long awaited feature !

Changes from CLAIRE 2.0 to CLAIRE 3.0

- Lists and sets are strongly typed – this is THE major change. Because we do no longer rely on dynamic typing, the following is no longer true (in 3.0, but actually true in 3.2):

```
list(1,2,4) % list[integer]
```

- Tuples are no longer lists, they are an independent subtype of bag. This should not cause any problems, unless you were using list methods on tuple – a really poor idea.

Avoiding common mistakes:

Here are a few unwise programming practices that occur naturally:

* Using a global variable to store a complex set expression that will only be used in an iteration. Compare:

```
let s := {x in S | P(x)} in
  for y in s f(y)
```

With

```
for y in {x in S | P(x)} f(y)
```

the second approach is better because the compiler will not build the intermediate selection set if it is just built to be iterated.

* Declare the range of a slot as C U {unknown}, as in

```
Person <: thing(father: (Person U {unknown}))
```

This is perfectly correct, but declaring the range as integer will be more efficient, because the compiler has been tuned to deal with the unknown value. Notice that one can reset the value to unknown with the method *erase*(p,x).

* Using the `{f(x) | x in S}` where a `list{f(x) | x in S}` is sufficient. The first form implies a duplicate elimination after the collection process.

* Using the same name for a module and a file, which causes a problem at compile time.

* Using a complex expression with make_list or make_array, where the expected behavior is to get multiple evaluations of the expression whereas CLAIRE shares the same result. For instance,

```
make_list(10,myObject())                  // does NOT create a list with 10 different objects

A :: make_array(10,list[],list{i | i in (1 .. 10)}) // A[1] and A[2] are the same list !
```

2. The Environment

CLAIRE provides a few simple but powerful tools for software development: interactive debugger, tracer and inspector. As told earlier, CLAIRE4.0 inherits its profiling capabilities from Go.

2.1 Tracing

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 statements are very useful while debugging. 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 character 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) x := f(x,x) else x :- 1,
         //[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.

Note : implicit tracing requires the debugger to be activated, using the *debug*() command. The statement *debug*() is designed to be entered at the top-level and not placed within methods.

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.

Spy may become a burden from a execution time point of view, so the statement `spy(p1, p2, ...)` tells CLAIRE that the `spy()` call should only be executed during the evaluation of a function call f(...) where f is one of the properties p1, p2, ... Note that this instance of the spy method takes a variable number of arguments, that all must be properties. A typical use is when you want to find a bug in a part of your program that is executed long after its start, say in the result printing stage. By declaring `spy(printResult)`, the spy() method will only be called once the program has entered the printResult method.

2.2 Debugging

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] > 1 / 0
debug[2] > f(0)
debug[3] > f(1)
debug[4] > f(2)
debug> block(10)
debug[1] > 1 / 0
debug[2] > f@integer(x = 0, y = -1)
```

```
debug[3] > f@integer(x = 1, y = 0)
debug[4] > f@integer(x = 2, y = 1)
```

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. However, note that this is a eval(read) loop with no implicit printing (to keep the dialog short) thus you need to input queries that cause explicit printing such as show(class) or print(1 + 2). To exit the breakpoint top-level loop, one must enter ‘q’.

2.3 Inspecting

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.

Tracing and using the spy methods are powerful tools for debugging and understanding a program. However, most often they yield too much information because we are only interested in a short part of the program, which is executed after quite a while. CLAIRE provides a function call counter and the ability to activate tracing or spying only after a given number of calls have been processed. The call counter is reset to 0 each time a new expression is evaluated at the top level.

* `trace(where)` activates the call counter. Each implicit trace contains the counter value (between brackets)
* `trace(x,y)` sets the verbosity level to x after processing y calls.
* `trace(spy,y)` activates the spy method only after processing y calls.
* `step(y)` activates the stepper only after processing y calls.

Finally, CLAIRE provides the ability to stop when the interpreter enters a given property with a given list of arguments. This is useful to find out why a given method was invoked (using the debugger). Remember that to stop for any set of arguments you may use step(p).

`stop(p:property,a1:any, …, an:any)`	tells the interpreter to stop when the call p(a1,..,an) is evaluated.
`stop(p)`	cancels all stopping statements about p.

3. The Compiler

3.1 Compiler Architecture

The CLAIRE compiler is based on a reflective architecture, where everything is represented by objects, with associated methods that may be redefined or extended. It is organized into three separate components:

- The **Optimizer**, which is represented by the Optimize module. The optimizer transforms a CLAIRE instruction into an equivalent but faster, optimized, instruction.
- The generic code producer, which is part of the **Generate** module, and which contains a set of generic methods that produce target code by pretty-printing optimized instructions;
- The target-specific code producer, which is represented by an object PRODUCER from a specific meta-class that is dependent on the target language. Producer objects are intended to be inter-changeable, so that generating Java code, for instance is achieved by defining a java_producer object. The Generate module contains the definition of the default Go producer.

Thus, one can summarize CLAIRE compiling as follows. The first step is a source-to-source transformation that is done at the object level, using CLAIRE's reflective nature. This is the most complex part of the compiler (the Optimize module). The second step is the generation of Go-like code, which is simply obtained by an adequate "pretty-printing" of the objects that represent the optimized instructions. The last step is only applied when this pretty printing is different from one target language to another. This "ideal architecture" is actually implemented for the major part of the CLAIRE language, in the sense that it is indeed enough to re-define a small part of the compiler code to generate code for a new target language. As of CLAIRE4, Go is the default, and only-supported, target language.

3.2 Go Code Generation

The CLAIRE compiler generates Go files from a CLAIRE file (or a set of files associated with a module). For a given file `f.cl`, it will produce a code file `f.go`. In the case of a module m, it will produce one specific "meta" file (named <m>-meta.go) that contains the reflective descfription of classes and methods together with the multiple code files associated to each file `f.cl` in `m.made_of`. Each such code file contains a list of Go functions for each method in the file, plus one large method that contains the Go 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 Go generated file. In addition, the compiler can also be used to generate a "system file", which name is f.go, where f is the output parameter. This system file is either produced implicitly by using the -cm/–cx options of the compiler, or it is produced explicitly with the -sf/sx option.

CLAIRE provides a way to include Go 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. The compiler assumes that no value is returned (type void). If the value of the Go expression must be returned, then its type (i.e., its sort that is a CLAIRE class) must be given as the second argument. For instance, to define a bitwise and operation we may use

```
bit&(x:integer,y:integer) : integer -> externC("(x & y)",integer)
```

3.3 What the Compiler Produces

Reading or using the Go 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). There are, however, two things that make the Go code more complex than the input CLAIRE code

- Go does not support exception handling, so the compiler will insert all the necessary additional tests
- Statically-typed fragments yield straightforward equivalent Go code, but code that requires dynamic typing uses "EID" struct (a pair made of a class and a value).

The first output of the compiler is a set of class definition that is placed in the module meta file. Each CLAIRE class that is an object sort (i.e., that is included in object) produces a Go structure with inheritance as embedding, where each slot of the class becomes a data member in the Go structure. This class will be used to access CLAIRE objects within a Go program as if it was a standard Go object. For instance, a definition like

```
C <: object(x:string,y:int,z:float)
```

will produce

```
type C struct {
      ClaireObject
      x *ClaireString
      y int y
      z float64}
```

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 characters that are acceptable for Go. 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 Go keyword for a class name).

The second output of the CLAIRE compiler is a set of CLAIRE classes and Go packages that represent each module. Each module <m> is represented by a package with the same name. Hence, assuming that compiler.output = "src", the Go files generated for <m> are put in `src/m`.

For each named object x in the module m (i.e. that belongs to thing), CLAIRE generates a Go variable m.x. As previously, special characters are translated, to avoid conflict with Go reserved keywords. Moreover, a "Claire" prefix is added to the identifier generated for each class, thus, for example, class is represented as `Kernel.ClaireClass`.. To find out which identifier is generated, one may use the **`g_test`** method. This method is an on-line compiler that is intended to show what to expect. **`g_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

```
g_test(`(for x in class show(x)))
```

Let us consider a small example that will show how to create a claire object from Go or how to invoke a method. Suppose that we define :

```
point <: class(x:integer, y:integer, tag:string)
f(p:point,s:string) -> (p.tag := s)
```

The code shown by

```
g_test(`f(point(x = 1), "test"))
```

will be (modulo the GC statements that depend on the settings and that will be discussed later) :

```
{ var arg_1 *ClaireAny
     { var _CL_obj *Point   = ToPoint(new(Point).Is(C_point))
      _CL_obj.X = 1
      arg_1 = Language.F_close_class(ToClass(_CL_obj.Id())).Id()
     }
result = ToPoint(arg_1).F(MakeString("test"))
}
```

The third output of the CLAIRE compiler is a set of functions (or methods). CLAIRE generates a Go 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 input variables (as for any local variables) are a straightforward translation from CLAIRE (same name, equivalent Go type). The body of the function is the Go code that is equivalent to the original CLAIRE body of the method. The Go code generated by CLAIRE is an almost straightforward translation of the source code, modulo the two caveat previously made (handling of exceptions and dynamically-typed code). In addition, CLAIRE also produces one load function for each module m (with name "**`m.MetaLoad()`**") that contains code that builds all the objects, including the classes and methods, contained in the file.

To improve the readability of the generated code – this is precisely what happens in this example with the generate go method "F" – , the compiler will produce a Go method (versus a function) when:

1. the method is defined in the same module as the domain class (a Go requirement)
2. the method is not in polymorphic conflict with another method with the same name (property).

Because Go is much stricter than C or C++ about function pointers and dynamic calls, the CLAIRE compiler actually generates two functions/method for each CLAIRE method

- the native function whose arguments are the Go equivalents types
- the EID function that receives all values in EID form and translates them into the native form. This is how the interpreter is implemented.

The naming convention (how the compiler generates names for classes and methods) is more complicated with Go than with previous versions because Go requires a capital letter to make an identifier public in a package. Hence the CLAIRE compiler uses the following strategy:

- Classes names start with the "Claire" prefix and are capitalized (point -> ClairePoint)
- When two functions are generated for a method m@c, the names are F_m_c for the native version and E_m_c for the EID version
- When m is transformed into a Go method, the name is simply capitalized.

3.4 System Integration

Methods are usually defined within CLAIRE. However, it is also possible to define a method through a Go function, since most entities in CLAIRE can be shared with Go. The Go function must accept the method's parameters with the Go 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.

Objects are represented as pointers to Go stuctures: to each class we associate a Go structure with the same name where each slot of the object becomes a field (instance variable) in the structure. Integers and floats share the same representation with Go and characters are represented with Runes. A CLAIRE string is a pointer to a ClaireString that holds a Go string. Let us illustrate how 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 Go file the two C function *cos_for_claire* as follows.

```
func F_cos_for_claire(y FLOAT) float64 { return math.Cos(x)}

func E_cos_for_claire(x EID) EID {
  return EID{C__FLOAT, FVAL(cos_for_claire(FLOAT(x))}}
```

When the two files are compiled and linked together, the method cos is defined on floats and can be used freely.

The only catch is the naming convention due to polymorphism and extensibility. The default strategy is to generate the function *F_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 multiple definitions 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 Go code otherwise to check the exact name.

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

```
func F_fib_integer(int x) int { …
```

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 Go objects. When a CLAIRE file is compiled, the structure definitions associated with the classes are placed in a module meta file.

WARNING: the use of Go keywords as names for CLAIRE named objects is not supported and may cause errors when the Go compiler is called.

3.5 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 Go 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 assume that there will be no selector errors or range errors at run-time we trust explicit types & super. Run-time checks are removed. The type information contained in local variable definition (inside a let) and in a super (f@c(...)) has priority over type inference. This allows the compiler to perform further static binding.

3 → the compiler does not check for list or array index bounds. Also, it does not perform overflow checking (integer & arrays), in addition to level 2. This correspond to the "-O2" option and should be used carefully since errors will be caught by the underlying Go execution.

The slot *overflow?*(compiler) is used to control separately the overflow checking for integer arithmetic. When it is turned to true, the compiler will produce safe code with respect to overflows. This is useful since un-detected overflow errors can yield run-time crashes that are hard to debug (cf. troubleshooting).

The slot *inline?*(compiler) tells the compiler that inline methods should include their original CLAIRE body in the compiled code so that further programs that use these inline methods can be compiled with macroexpansion. The default is false, since this option (turning to true) requires the reader module to be linked with the generated module. This is only necessary is you are developing a module that will be used as a library for some other programs.

The two slots *active?*(compiler) and *loading?*(compiler) are used to represent the status of the compiler. The first one simply tells if the compiler is in use or not. The second one distinguishes between the first step of the compiler (loading the program to be compiled) and the second step (actually compiling code).

The slot *libraries*(compiler) contains a list of strings, each of which is the name of a library (packages) that needs to be imported by the generated Go file.

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 20-200% range.

The CLAIRE compiler also generates code to check that object 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 tables) that are given as parameters in a *known!* statement.

3.6 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(c.slots) > 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 expressions 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 uses: 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 occurrence 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 exercise 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(c.slots) > 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) ...
```

A word of warning about the iteration of complex expression : this type of optimization is based on code substitution and will not work if the construction of the set is encapsulated in a method. Consider the following example:

```
f1() => list{f(x) | x in {i in (1 .. n) | Q(i) > 0}}
for x in f1() print(x)
f2() -> list{f(x) | x in {i in (1 .. n) | Q(i) > 0}}
for x in f2() print(x)
```

The first iteration will be thoroughly optimized and will not yield any set allocation, whereas the second example will yield the construction and the allocation of the set that is being iterated.

Patterns are also useful to add new code substitution rules. This is achieved 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(y)[1]) & (x != eval(args(y)[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])
```

The use of *patterns* and *Iterate* is geared towards expressions of the language (meta-programming), whereas iterate is intended to describe data structures. Notice that if you define *iterate* on a new data structure, say a FloatInterval, it will only be used by the compiler to macroexpand the iteration `for x in s e` when the compiler can determine precisely that s is of type FloatInterval. There is a way to tell the compiler that all existing iteration strategies that apply to s should be applied. We use Id as a syntactical marker (as for explicit evaluation during compiling) and write `for x in Id(s) e`. For instance, if there are two possible types for s that have a restriction of iterate (FloatInterval et otherType), the following code should be produced:

```
if (s % type1)   <iteration with iterate@s1>
else if (s % type2) <iteration with iterate@s1>
else <usual iteration>
```

One can see that this technique should be used carefully, especially when the type inferred for s is too general. This is why we rely on an explicit syntactical marking from the programmer. This is, on the other hand, very convenient to write fast and generic code when sub-classing is used to provide with different implementations (with different iteration strategies) of one single generic data structure.

4. Troubleshooting

4.1 Debugging CLAIRE Errors

The easiest way to debug a CLAIRE error (i.e., an error that is reported by CLAIRE) is to use the debugger. If the error occurs in a compiled program, you must use the –D option when you compile your code. There are three tools that run under the debugger and that are most useful: trace, spy and stop (cf. Section 2). The inspector (?) is also very convenient to observe your own data structure and find out what went wrong. Also, notice that **`stat()`** will produce a detailed report about memory usage.

Here is a list of the CLAIRE-generated errors. They are all represented by an integer code (0-100 for "system" error and 100-200 for high-level error; the codes over 200 are used by the compiler as we shall later see). Most error message are self-explanatory but some may tolerate a few additional explanations …

```
[5]     nth[~S] outside of scope for ~S
[7]     Skip applied on ~S with a negative argument ~S
[8]     List operation: cdr(()) is undefined
[9]     String buffer is full: ~S
[10]    Cannot create an imported entity from NULL reference
[11]    nth_string[~S]: string too short~S
[12]    Symbol Table table full
[13]    Cannot create a subclass for ~S [~A]
[16]    Temporary output string buffer too small"
[17]    Bag Type Error: ~S not in ~S"
[18]    definition of ~S is in conflict with an object from ~S
[19]    Integer overflow
[20]    Integer arithmetic: division/modulo of ~A by 0
[21]    Integer to character: ~S is a wrong value
```

[22] `Cannot create a string with negative length ~S`

[23] `Not enough memory to instal claire`

[24] `execution stack is full [~A]`

[26] `Wrong usage of time counter [~A]"`

[27] `internal garbage protection stack overflow`

[29] `There is no module ~S`

[30] `Attempt to read a private symbol ~S`

[31] `External function not compiled yet`

[32] `Too many arguments (~S) for function ~S`

[33] `Exception handling: stack overflow`

[34] `User interrupt: EXECUTION ABORTED`

[35] `reading char '~S': wrong char: ~S`

[36] `cannot open file ~A`

[37] `world stack is full`

[38] `Undefined access to ~S`

[39] `cannot convert ~S to an integer"`

[40] `integer multiplication overflow with ~S and ~S`

[41] `wrong NTH access on ~S and ~S`
A list/set access l[i] failed because the index i was not in (1 .. length(l))

[42] `Wrong array[~S] init value: ~S`

[101] `~S is not a variable:`
An assignment (x := …) is executed where x is not a variable

[102] `the first argument in ~S must be a string`

[103] `not enough arguments in ~S`

[104] `Syntax error with ~S (one arg. expected)`

[105] `cannot instantiate ~S`
The class cannot be instantiated because it was declared as abstract.

[106] `the object ~S does not understand ~S`

[107] `class re-definition is not valid: ~S`

[108] `default(~S) = ~S does not belong to ~S`

[109] `the parent class ~S of ~S is closed`
Cannot create a subclass of X, which was declared final.

[110] `wrong signature definition ~S"`

[111] `wrong typed argument ~S"`
While reading the signature of a method (list of typed arguments)

[112] `wrong type expression ~S"`

[113] `Wrong lambda definition lambda[~S]`

[114] `Wrong parametrization ~S`

[115] `the (resulting) range ~S is not a type`

[116] `~S not allowed in function!`

[117] `loose delimiter ~S in program [line ~A ?]`

[118] `read wrong char ~S after ~S`

[119] `read X instead of a Y in a Z`
This is produced when the parser finds a grammar error. Please check the syntax for instructions of type Z

[120] `the file ~A cannot be opened   Produced by a load_file`

[121] `unprintable error has occurred.`
This happens if the printing of an error produced another error. The most common reason is because the self_print method of one of the arguments it itself bugged.

[122] `~A is not a float`

[123] `YOU ARE USING PRINT_in_string_void RECURSIVELY.`
In CLAIRE 3.0, print_in_string() cannot be used recursively

[124] `the value ~S does not belong to the range ~S.`
This error is produced by type safety checks produced by the compiler. You may look at the generated code to understand which range is violated if it is not self-evident.

[125] `ephemeral classes cannot be abstract`

[126] `ephemeral classes cannot be set as final`

[127] `~S can no longer become abstract.`
The property was ‘closed’ by the compiler and cannot be set as an ‘open’ property

[128] `~S should be an integer.`
within the inspector loop, the proper syntax to store a value in a variable is
put(<**integer**>, <name of variable>).

[129] `trace not implemented on ~S`

[130] `untrace not implemented on ~S`

[131] `Cannot profile a reified property ~S`

[132] `Cannot change ~S(~S)`
The property was declared as a read-only

[133] `Inversion of ~S(~S,~S) impossible`

[134] `Cannot apply add to ~S.`
The property is not multi-valued

[135] `~S does not belong to the domain of ~S`

[136] `~S is not a collection`
In CLAIRE 3.0, only members of the **collection** class may be iterated.

[137] `~S and ~S cannot be inverses for one another`

[138] `The value of ~S(~S) is unknown`
The value of a slot or an array is unknown

[139] `~S: range error, ~S does not belong? to ~S.`

[140] `The property ~S is not defined (was applied to ~S).`
There are no restrictions for the property, probably a typo …

[141] `~S is a wrong arg list for ~S.`
No method was found corresponding to the types of the parameters

[142] `return called outside of a loop (for or while).`

[143] `~I not allowed in format`
Format is a method, not a control structure like printf. Thus, it does not support the ~I option.

[144] `evaluate(~S) is not defined`

[145] `the symbol ~A is unbound`

[146] `The variable ~S is not defined`

[147] `a name cannot be made from ~S`

[148] `Wrong selector: ~S, cannot make a property`

[149] `wrong keyword (~S) after ~S.`
expecting a -> or => in a method definition

[150] `Illegal use of :~S after ~S.`

[151] `~S not allowed after ~S`

[152] `Separation missing between ~S and ~S [~S?`

[153] `eof inside an expression`

[154] `~S<~S not allowed .`
The form C<X> is reserved for parameterized classes

[155] `missing | in exists / forall`

[156] `wrong use of exists(~S ~S ...`

[157] `~S cannot follow list{`

[158] `wrong type in call ~S@~S`

[159] `missing ( after ~S@~S`

[160] `wrong use of special char #`

[161] `Missing keyword ~S after ~S`

[162] `Missing separator ~S after ~S`

[163] `wrong separator ~S after ~S`

[164] `~S cannot be assigned with :=`

[165] `~S is illegal after a let/when`

[166] `Missing ( after case ~S`

[167] `missing ) or , after ~S`

[168] `missing | in selection`

[169] `missing separation between ~S and ~S`

[170] `cannot use ~S in a set constant"`

[171] `Read the character ~S inside a sequence`

[172] `the sequence ...~S must end with ~A`

[173] `Expression starting with else`

[174] `Wrong instantiation list ~S(~S...`

[175] `Wrong form ~S in ~S(~S)`

[176] `Missing ] after ~S`

[177] `subtyping of ~S not allowed`

[178] `cannot enumerate ~S`
Iteration is only supported for sets expressions (i.e., members of the collection root class, arrays and integers)

[179] `(~S % ~S): not a set!`
Membership is only supported for sets expressions (i.e., members of the collection root class, arrays and integers)

[180] `nth[~S] out of scope for ~S`
There is not enough memory to allocate an objet – the parameter is the size (in cells) that is required for this object.

[181] `cannot override a slot for a closed property ~S`
You are redefining an existing (inherited) slot for a new class while the property is closed.

[182] `the interval (~A -- ~A) is empty`
You cannot define an empty interval with -- . This is precisely what - - is there for: guarantee that the returned value is a true, non-empty, interval.

[183] `min/max of an empty set is not defined`
The min or max method was applied to the empty set.

[184] `the close method … has the wrong range`
The close method is called automatically after a new instantiation and must return the new object onto which it is applied.

[185] `cannot define … as a uniform property`
The declaration interface(p) was applied onto a property that is not uniform. Uniformity is the fact that all restrictions have the same signature with the exception of the first member (the domain).

[186] `Definition conflict between … and … on a closed property … is not allowed`
Once a property is closed, a new restriction may be added only if it does not cause an inheritance conflict with another restriction of this property.

[187] `The class ~S cannot be declared as ephemeral because of its subclass ~S`
All subclasses of an ephemeral class must be ephemeral .

[188] `The property ~S is already defined`
One cannot make an explicit property definition if an implicit (or another explicit) definition has already been made .

4.2 Debugging System Errors

A system error here is an error reported the go run-time (once the program is compiled with the go compiler). A system error that occurs during the execution of an interpreted program is due to a bug in CLAIRE. You should:

- use the tracing methods to detect where the problem occurs and try to find an alternate programming paradigm
- see if an endless loop occurs by using the –s * 0 option that will make a small execution stack. An endless loop often produces a system error that is not properly handled by the operating system.
- send a bug report to clairebugs@yahoo.fr

If a system error occurs with a compiled program, it may be due to a bug in the compiler or to the use of an optimization level that is not appropriate (cf. the discussion about compiler safety). You must first make sure that you reduce the optimization level to 2 or lower. This can be done easily by using the –safe option of the compiler. If the bug persists, it should be treated as previously with the additional option of using a Go debugger to find out where the bug is occurring. A bug that occurs only with low levels of safety is probably due to a compiler warning that got ignored.

4.3 Debugging Compiler Errors

During the compilation, the compiler may detect three kinds of anomalies, and will issue respectively a note, a warning or an error. A note is meant to inform the user and does not necessarily reflect a problem. Notes can be ignored safely, although it is better to look into them. A warning is usually associated to a real problem and they must be looked into. A warning may simply point to a non-usual but nevertheless correct situation, yielding a correct executable program. However, most often it produces a non-correct executable and yields a system error at run-time.

Therefore, it is necessary to observe all warnings and treat them accordingly. Last, the compiler may detect a situation which makes code generation impossible and stop with an error message.

The next release of the documentation will include a list of the compiler warnings. Here is the current list of compiler errors:

[201] `Loose delimiter in program,`
most often an unmatched) or] that was not caught by the CLAIRE reader.

[202] `A do should have been used for …,`
a list or a set construction is not necessary if the result is not used.

[203] `You should have used a FOR here: …,`
an image or selection is built and not used (a for was enough)

[204] `break not inside a For or While: …,`
a break statement must be embedded into a for or a while loop, and must not be embedded into an inner try/catch statement

[205] `message … sent to void object`
the receiver (first argument) of the function call is of type void.

[206] `use of void … in … :`
use of a void argument, that is a an expression that has received a void type (for instance the "return value" of a method with range void).

[207] `inline …: range … is incompatible with … (inferred) :` the type inferred for an inline method (=>) is not compatible with the one that was declared

[208] `wrong type declaration for case … in …:` A case must use type expressions as tags for branches

[209] `the first argument in … must be a string:` the first argument of a printf is the format string

[210] `not enough arguments in … :`
a printf must have exactly as many arguments as there are ~X in the format string

[211] `… cannot be both the name of a file and of a module`
In claire3, the files that make a module cannot be given the same name as the module. For instance, a simple module *foo* cannot be defined with a simple file names *foo.cl*.

[212] `the value … of type … cannot be placed in the variable …:` the type inferred for the argument of an assignment is not compatible with the type of the variable. This is often the case if the type of the variable is itself inferred (wrongly) from the initial value. It is, therefore, necessary to give an explicit type to this variable.

[213] `… is not a variable` : assignment require variables

[214] `cannot assign …:` a global constant was assigned

[215] `the symbol … is unbound :`
an identifier that was never defined is used (most often a typo).

[216] `… has more than 10 parameters :`
The compiler only supports dynamic calls with fewer than 12 parameters.

[217] `… and … cannot be defined in the same module :`
There is a conflict name between two properties.

[218] `sort error: cannot compile … :`
the given range for a method and the one inferred by CLAIRE are very different and correspond to different sorts, making code generation impossible.

Here is the list of the warnings that may be generated by the compiler. If the verbosity is set to 3, notes may also be printed. Notes may be ignored, whereas warnings are importants.

[251] `the bag addition … is poorly typed:` an add message has been found that applies to a typed bag (lit or set) and the type that is inferred for the value is not compatible with this type.

[252] `unsafe update on bag: type … into … :` this is similar to the previous situation, but the operation on the typed bag is a write.

[253] `sort error in … : … is a … :` an update (x.s := e) has a poor typing which is likely to provoke a sort error at compile time. The compiler prints the value and its inferred type, which is not compatible with the receiver (a slot, an array or a table)

[254] `Non diet call …` : when the compiler uses the diet mode (for instance with a Java code producer) and when it finds a non diet message, a warning is generated.

[255] `The property … is undefined` : a call to a property that does not exists has been found by the compiler. Although this is not an error in CLAIRE. It is strongly un-advised, and a better practice is to declare in advance properties that will receive a definition (i.e., methods) later.

[256] `wrongly typed message … [X]` : a call has been found by the compiler that cannot be statically bound. The argument X is the type that was inferred for the receiver of the message.

[257] `… of type … is put in the variable …` : a typed variable is assigned a value that is incompatible with its range.

[258] `range of variable in … is wrong :` the compiler has inferred a range for this variable that is incompatible with the one that was given by the user.

[260] `CLAIRE 3.3 SYNTAX - Test in … should be a boolean :` the compiler generates a warning when the condition in an If statement is not a boolean.

[261] `unsafe typed collect (…) : … is not in …:` a typed list or set construction using an image/collection such as list<X>{….} is poorly typed since the inferred type of what is stored in the newly constructed bag is incompatible with the declared type X.

[262] `unsafe typed list/set: … not in …:` a typed list/set construction list<X>(….) is poorly typed since one of the element has an inferred type which is not compatible with X.

[263] `… = … will fail :` the compiler has found an equality test between two members that have static different types.

[264] `equality meant as an assignment : … :` the compiler has found an equality test whose value is not used. This suggests a classical confusion with C/C++ where = is meant for assignment.

[265] `… is unknown`: the compiler has found an unknown named object definition.

[266] `wrong status … -> …i:` A method with an explicit native definition (using the function!(…) pattern) has received a status that is not understood by Claire (cf. Appendix C, section 3.4).

The CLAIRE logo represents:

- three foundation paradigms: Objects, Relations, Functions
- three high-level features: Types, Rules, Versions
- the triangle stands for the tight integration
- the circle is a symbol of unity and ease of use

INDEX

NOTES