UNIVERSITE CATHOLIQUE DE LOUVAIN

ECOLE POLYTECHNIQUE DE LOUVAIN

Comprehensions in Mozart

Supervisor: Peter Van Roy
Readers: Rea Der
Rea Der

Thesis submitted for the master degree
in computer science and engineering
options networking and security
gestion
by François Fonteyn

Louvain-la-Neuve
Academic year 2013 - 2014

Thanks

To conclude this Master's thesis, I would like to thank

Peter Van Roy, my promotor who has given me both the liberty and the ideas I needed in addition to its passion for the beauty of Mozart, and also an inspiring Professor and author,

All the Professors of the computer science department at U.C.L.,

My parents and family.

Foreword

After spending almost five years studying computer science engineering, we have learnt many things about programming languages. We got stuck on problems, we found solutions, we have enjoyed the beauty and simplicity of some languages, or at least enjoyed some principles and ideas about all languages tried. On the other hand, we also have suffered of the lack of functionalities in some languages. Every language is a different flavor, a different point of view on how to reach the ultimate goal: efficiently developing programs.

One could ask why so many languages exist out there. For us, one reason is obvious. Every computer scientist has its own vision of the ideal programming language, no matter what are the problems to solve. Does this perfect language exist ? Most probably it does not, but our goal is to contribute to this quest.

Despite all these different views, we think simplicity and expressivity are two essential features that should be present as much as possible. Python is famous for its simplicity and the high-level abstractions and structures it provides. One of these is the concept of list comprehension. This concept mixes both simplicity and expressivity thanks to its mathematical formulation. Every computer scientist has been in a situation where he felt the need to declare efficiently and simply any kind of list. List comprehension - or at least a similar concept - is also part of many languages such as Erlang, Haskell, Scala and Ruby.

Going further than lists, the concept of comprehension can be applied to others structures such as tuples. Again, such a functionality aims at making Mozart more expressive.

Among all the languages we have tried, Mozart is the one presenting the widest range of paradigms and possibilities while keeping things simple. This is why we have chosen to add the possibility to use comprehensions in Mozart. In order to keep the beauty of Mozart, this implies that this new concept must be usable with all the possibilities Mozart already offers.

As a new version of Mozart has recently been released, known as Mozart2, we will focus our work on this version only.

Contents

Thanks **3**

Introduction **1**

1 Mozart and comprehensions **2**

1.1 Mozart2, a functional language . . . 2

Syntactic sugars . . . 2

Expressions and statements . . . 2

Declarations . . . 3

Variables . . . 5

Functions and procedures . . . 5

Records and tuples . . . 6

Lists . . . 7

If statements . . . 8

Pattern matching . . . 9

Recursion . . . 10

Threads . . . 11

Streams . . . 12

Laziness . . . 14

Cells . . . 15

Classes and objects . . . 15

Functors . . . 15

1.2 List comprehensions . . . 16

Principle . . . 16

Terminology . . . 18

1.3 Record comprehensions . . . 18

Principle . . . 18

Terminology . . . 19

1.4 Complete syntax . . . 20

List comprehensions . . . 20

Record comprehensions . . . 24

2 The compiler of Mozart2 **26**
2.1 Generic design of a compiler using a virtual machine . . . 27
Lexical analysis, the lexer . . . 27
Syntax analysis, the parser . . . 28
Semantic analysis . . . 29
Optimization . . . 30
Code generation . . . 30
Mozart2 . . . 30
2.2 The abstract syntax tree of Mozart2 . . . 31
Positions . . . 31
Basic types . . . 32
Equality . . . 32
Skip statement . . . 32
Local declarations . . . 32
Operations . . . 32
Procedure and function calls . . . 33
Procedure and function declarations . . . 33
If statements . . . 33
Threads . . . 34
Successive statements . . . 34
Records, tuples and lists . . . 35
Layers and range generators . . . 36
For loops . . . 36
Step Points . . . 37
2.3 New nodes of AST for comprehensions . . . 37
Bounded buffers . . . 38
Ranges generated by records . . . 38
Expressions . . . 38
Collects . . . 39
List comprehension levels . . . 39
List comprehensions . . . 40
Record comprehensions . . . 40
2.4 Syntactic sugars compilation in Mozart2 . . . 42
Macros . . . 42
The lexer, lexical analysis . . . 43
The parser, syntax analysis . . . 43
Checking the syntax . . . 46
Coordinates in case of failure . . . 47
The unnester, transforming syntactic sugars . . . 48

Keywords in Emacs . . . 49

3 Functional transformations 50

3.1 Functional transformations . . . 50

An inspiration, for loops examples . . . 50

Returning a list . . . 52

C-style generator . . . 53

Integer generator . . . 54

List generator . . . 55

From generator . . . 55

Record generator . . . 55

Level condition . . . 58

Multi layer . . . 59

Multi level . . . 61

Multi output . . . 63

Output condition . . . 65

Laziness . . . 65

Body . . . 65

Collector . . . 65

Record comprehension . . . 65

3.2 Implementation . . . 66

Architecture . . . 66

Applying the transformations . . . 66

4 The tests 67

4.1 General tests . . . 67

One level − one layer . . . 67

One level − multi layer . . . 68

Multi level − one layer . . . 69

Multi level − multi layer . . . 69

From . . . 70

Records as generators . . . 71

Multi output . . . 72

Labeled output . . . 73

Output conditions . . . 74

Laziness . . . 74

Miscellaneous . . . 75

Collectors and body . . . 76

Record comprehensions . . . 78

4.2 Other languages . . . 80

Erlang . . . 80
Python . . . 81
Haskell . . . 82
4.3 Performance tests . . . 82
Space analysis . . . 84
Time analysis . . . 84
4.4 Application examples . . . 85
Permutations . . . 85
Map . . . 85
Flatten . . . 86
Sort . . . 86
Factorial . . . 87

5 Concurrency using list comprehensions 89
5.1 Laziness . . . 89
5.2 Producer − consumer . . . 90
5.3 Multi output and conditions . . . 92
5.4 Logic gates . . . 92
5.5 Bounded buffer . . . 94

Conclusion 96
Possible evolutions . . . 96
Conclusion . . . 96

References 97
Offline . . . 97
Online . . . 98

Introduction

TODO: introduction

Chapter 1

Mozart and comprehensions

1.1. Mozart2, a functional language

This section describes what kind of language Mozart2 is as well as a subset of its possibilities. Going through all its functionalities is not the goal of this thesis. We only explain features that interest us for later parts. For a complete documentation on the features of Mozart2, we strongly recommend [11].

Mozart2 is a functional language. It means that a program is seen in a mathematical way. This implies that variables are constants - nobody can change their value - and that functions are constant - a function is typically constant is any language. In such languages, functions play a very important role. Programming in a functional environment often means passing functions as arguments.

In addition to this paradigm, Mozart2 implements many other paradigms but to keep things simple and to ensure that comprehensions can be used in any situation, we will not use these other paradigms for the implementation except if it is explicitly stated. The principal programming paradigms are shown in the figure 1.1.

People who feel comfortable enough with Mozart can skip all of or parts of the first section. Its division into small and specific paragraphs is helpful to quickly find a given information if you feel lost in the later chapters.

Syntactic sugars

A syntactic sugar is a more convenient way to express something that is formally more difficult to express. It is called a sugar because the compiler transforms it into its complicate - and true - structure. It can be considered as a functionality developed outside of the core of the language but that is always usable and that has been integrated in the syntax. For instance, we will see that for loops are a syntactic sugar because if variable can not change, then it is impossible to make a loop that stops.

Mozart2 is full of such syntactic sugars.

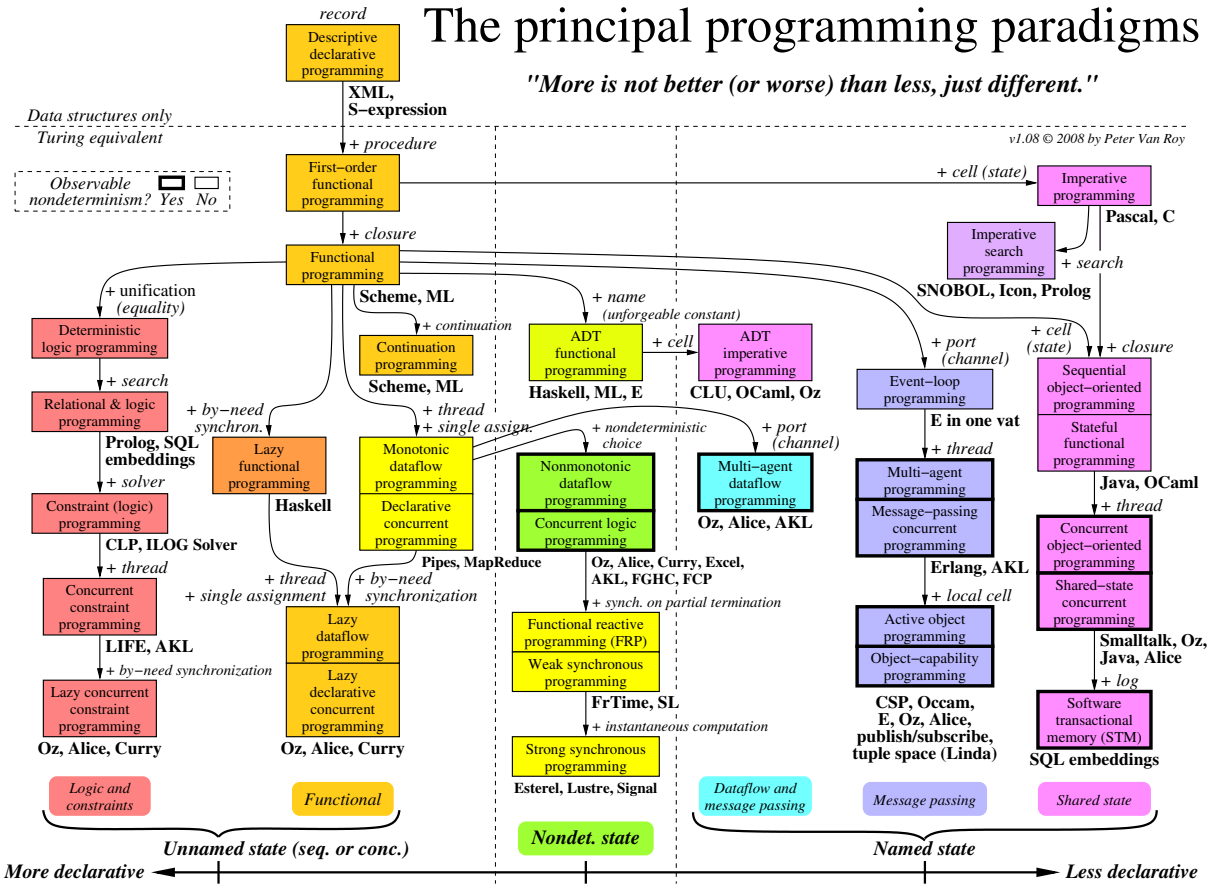

Figure 1.1: *Principal programming paradigms, Oz is an example of several paradigms. Source: [22].*

Expressions and statements

In Mozart2, an important distinction is made between instructions that act on data and instructions that *are* data. A statement is an instruction that acts on the state of the program by performing any kind of actions. It goes from assigning a variable to a given value to calling a function.

An expression has a value. It can be an addition, a list, a value, and much more things. For instance, the fact of assigning a variable to a value in a statement, it does not have a value but the variable and the value are both expressions, they have a value. When we will say an function returns a value, it means it returns an expression. So the body of a function is optionally made of statements and must end with an expression *in extenso* the value to return.

Declarations

In Mozart as in probably all programming languages, variables are used. The first thing to do in order to use them is to declare them. There are two ways to do that in Mozart2. Before explaining these two possibilities, we need to understand what scope is.

The scope of a variable is the whole part of code that has access to this variable. There exists one general scope, where all global variables are. Such variables are accessible from anywhere. The functions and procedures available in Mozart2 are part of this global scope. One can add variables, functions and procedures to the latter.

Scopes are nested inside each other. Nested scopes can be thought of like a stack. Every time a new scope is declared, it is pushed onto the stack. When a scope end is reached, the first element of the stack is popped. If the popped scope is not the scope ending then the nesting is wrong. Two variables with the same name can exist if and only if they come from different scopes. Only the one with the closest scope from the top of the stack can be accessed by its name. Another one can still be accessed if another accessible variable was assigned to it.

To declare global variables, one uses the structure `declare ... in ...`. To delimit a new scope, one has to use the `local ... in ... end` structure. A syntactic sugar allows omitting the `in` keyword when declaring global variables. Another syntactic sugar allows omitting the `local` and the `end` when declaring a scope inside another structure. When the scope is the main structure *in extenso* when it is not nested inside another structure, the latter sugar can not be used. A third syntactic sugar allows declaring more than one variable at a time, it transforms the declaration of N variables into N single declarations. Here are some examples with syntactic sugars used at the end.

```
%% No sugars
declare X in % X is now accessible everywhere
...

local X in % this instance of X momentary "overrides" the previous one-s
    ...    % the new X is accessible only here
end        % the new X is no longer accessible, the old X is accessible

%% Sugars
declare % syntactic sugar: no in
X = ... % Mozart "guesses" all the variables to declare

% beginning of the nesting structure
...     % syntactic sugar: no local
    X Y % locally declare X and Y
in
    ... % X and Y are accessible only here
...     % X and Y are no longer accessible
% rest of the nesting structure
```

Variables

Mozart is declarative so variables are constant[1] but they can be unbound or equivalently just declared but not assigned. An unbound variable is assimilated to be assigned to _. Once a variable is assigned, its value can not change but it still can be assigned to the same value. Variable names always start with an uppercase letter. Here is a small example.

```
declare Var in % Var is declared but still unbound, Var = _
Var = 42       % Var is definitely assigned to 42
Var = 42       % No error, Var is still assigned to 42
Var = 71       % Error because Var already assigned to another value
```

Types are implicit. The different types that interest us - there are others - are integer, floating point number, boolean, string, atom, character, procedure, record, cell and dictionary. Integers and floating points number do not overflow thanks to Mozart2 which uses a theoretically unlimited number of bits to store them. Atoms are groups of characters delimited by simple quotes that are optional if the atom contains no spaces and starts with a lower case letter. Strings are a representations of characters delimited by double quotes. In fact a string is a list of integers representing ASCII numbers. Lists will be explained in a few paragraphs. Briefly a list is an ordered set of values. A variable can also be assigned to `unit`. In a way, this can be compared to `null` in the sense that `unit` is not typed. On there other hand, an unbound variable is different from `unit`. An example is given below.

```
Int = 71
NegInt = ~42      % the minus sign is ~ when not used as a subtraction
Float = 3.14
Boolean = true    % or false
Unit = unit
String = "hello" % String is actually the list [104 101 108 108 111]
Atom = 'world'    % or equivalently Atom = world
Char = &c         % the ascii of 'c'
```

Functions and procedures

Functions are in fact a specific case of what is called procedures. A procedure is a routine one can call with or without arguments and that does not return anything. A function is actually a procedure with one extra argument which is unbound at calling time and that the procedure - the function - assigns. The value assigned to this variable is the actual result returned by the function. In a strict notation, functions do not exist, they are a syntactic sugar. This principle can be extended to several variables, meaning that the procedure assigns these variables. One

[1] There exist cells which contain modifiable variables, see in a later paragraph.

can use the symbol ? before an argument to indicate that the argument is unbound at calling time. A example of function, its equivalent procedure and a procedure with three unbound variables is the following:

```
declare Function EquivalentProcedure Procedure3 in
fun {Function A} % function returning 2 times its only argument
   2*A
end
proc {EquivalentProcedure A ?R} % equivalent procedure, the interrogation
   R = 2*A                      % point is optional and indicates unbound
end                             % variable at calling time
proc {Procedure3 A ?R1 ?R2 ?R3} % one bound argument, three unbound ones
   ... % procedure body
end
```

Calling a function or its equivalent procedure can be done is two equivalent ways. As a function returns, we can use a notation with an equal sign. As procedures do not return anything, we can use a notation with all arguments inside the call, result included. Here are examples:

```
declare Fun Proc R in
fun {Fun} 1 end
proc {Proc ?R} R=1 end
R = {Fun}  % R = 1
{Fun R}    % no error, R = 1
R = {Proc} % no error, R = 1
{Proc R}   % no error, R = 1
```

Variables can be assigned to the basic types we already saw earlier. When a variable is assigned to a procedure, the variable contains the procedure. Strictly speaking, a procedure never has a name, a variable is assigned to the procedure and we call the name of the procedure the name of the variable. This can be explicitly stated using the dollar character. The following code shows the equivalence.

```
declare Procedure Equivalent in
Procedure = proc {$ ...} ... end % no sugar
proc {Equivalent ...} ... end    % sugar
```

The body of a procedure defines a scope where its arguments are declared.

Records and tuples

We just saw how variables can be assigned procedures. Apart from procedures, variables can also be assigned to records. A record is a structure with a label and a arbitrary

number of fields. The label is an atom. A field is identified by its feature which is an atom, an integer or a character. The value of a field can be anything. The structure is `label(feature1:value1 ... featureN:valueN)`. Note that features can be skipped because Mozart will then use the value of an integer counter starting at 1 for the missing features. The order of the fields is not important if they have a feature. Some examples are given below.

```
declare R1 R2 R3 R4 in
R1 = iAmRecord1(a:1 b:2)
R1 = iAmRecord1(b:2 a:1)
R2 = iAmRecord2(1:a b:2 c:3 3:c any:thing can:Be inside:R1)
R3 = iAmRecord3(1:a 2:b 3:c)
R3 = iAmRecord3(a b c) % equivalent to previous definition of R3: no error
R4 = iAmRecord4(1:a b:2)
R4 = iAmRecord4(a b:2) % equivalent to previous definition of R4: no error
```

Tuples are a specialization of records. The only additional constraint is that features are only integers from 1 to the number of fields in the tuple. As for records, the order of the fields is not important if they have a feature. Tuples do not restrict the rules of records for the label. A syntactic sugar allows users to declare tuple more easily. It consists of just separating values by a hashtag. The label is then a hashtag. Here are some examples.

```
declare T1 T2 in
T1 = easyIsntIt(1:a 3:c 2:b)
T1 = easyIsntIt(a b 3:c)
T1 = easyIsntIt(a b c)
T2 = '#'(3:c 2:b 1:a)
T2 = '#'(3:c 2:b a)
T2 = '#'(a b c)
T2 = a#b#c
```

The arity of a record is the list of all its features. So for a tuple it is simply a list from 1 to the length of the tuple. For a record the list can be composed of any kind of features. The arity is always in ascending order with integers before atoms.

Accessing an element of a tuple or a record is simple. For this, the feature must be used and that is why we need unique features in the same record. To get the value of a given field, one just needs to do the following:

```
T.feat % accessing value with feature 'feat' of tuple/record called T
```

Lists

We explained the specialization of records into tuples, so now let us specialize tuples into lists. The label of a list is always `'|'`. A list contains one or two elements, no more, no less. A list contains one element if and only if it is empty, the element is then always the atom `nil`.

When a list has two elements, the first one can be anything and the second has to be a list. The definition of a list is consequently recursive. The last constraint is that at the last element of a list is followed by `nil` - or by an unbound variable. A syntactic sugar exists, the `'|'` allows appending the left element to the start of the list on the right side. The graph representation in figure 1.2 is equivalent to the following statements. We also give some syntactic sugars to express lists more easily. Note that a - flat - list containing N elements is in fact succession of $N+1$ lists, one for each element and one for termination - the empty list, `nil`.

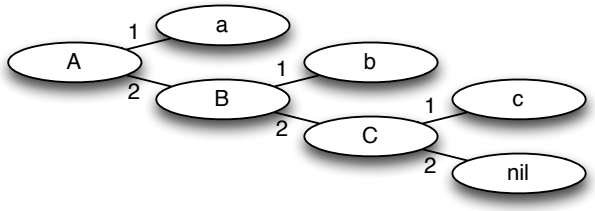

Figure 1.2: *Representation of the list layout*

```
declare A B C End in
A = '|'(1:a 2:'|'(1:b 2:'|'(1:c 2:nil))) % no sugar
A = '|'(a '|'(b '|'(c nil)))             % features skipped
A = a|(b|(c|nil))                        % label skipped
A = a|b|c|nil                            % levels skipped
A = [a b c]                              % most powerful syntactic sugar
A = a|B                                  % first level
B = b|C                                  % second level
C = c|End                                % third level
End = nil                                % last level
```

As lists are specializations of records, accessing an element is done by using '.1' or '.2' since lists contains only two elements - except empty lists. In the above example, the expressions `A.2.2.1` and `A.2.2.2` respectively evaluate to `c` and `nil`.

Using such a representation for lists in very powerful knowing that we can not change the value of variables. Indeed this definition allows very efficient recursive procedures as we will see shortly in the paragraph about recursion.

If statements

If statements are straightforward and well delimited into three parts: the condition, the true statement and the false statement. The latter is optional. The structure is the following:

```
if Condition then % if ... then
    ...           % instructions if true
else              % optional
    ...           % instructions if false, optional
end               % end delimiter
```

The condition can be any expression that evaluates to true or false. One can compose conditions using conjunctions with the keyword `andthen` and disjunctions with the keyword`orelse`. Here are some examples:

```
% Examples of what Condition can be:
true
false
A == C                        % true iff A is C
A.2 == nil andthen B+2 > 0 % true iff A.2 is nil and if B+2 > 0
A == unit orelse IsLazy     % true iff A is unit or if IsLazy is true
```

Pattern matching

As we will see in the next paragraph, it is very useful to check whether a list has one or two elements and to react accordingly. This can be done using if statements. Sometimes however, the pattern matching can be more than useful to do that. As its name says, pattern matching consists in matching an expression to a given pattern, a given shape, a given layout. Here is an example with lists:

```
case List          % apply pattern matching on Expression called List
of nil then        % if List is matched to nil
    ...
[] first|Tail      % if List is a list with its first element being 'first'
    ...            % and Tail being assigned to the second element
[] '|'(1:Head 2:Tail) then % if List has two elements: new scope
    ...                    % Head is the first element, Tail the second
else               % if none of the above
    ...
end                % end of matching
```

The advantage of this in comparison to if statements is that we directly have the elements of the list in a new scope. This structure is more flexible. We can match to basically anything: records, tuples, lists, atoms, *etc.* Here is an example with a record:

```
case rec(a:1 b:2)
of rec(a:A b:B c:C) then % no match because no feature 'c'
   ...
[] r(a:A b:B) then       % no match because wrong label
   ...
[] rec(a:A b:B _) then   % no match because wrong number of fields
   ...
[] rec(a:A b:B) then     % match, A = 1 and B = 2
   ...
end
```

Note that patterns are tested in order. So the first pattern to match is the only one chosen even if others match as well - in the above example with lists, if `List` matches `first|Tail`, it also matches `Head|Tail` but only the first will be considered. Syntactic sugars can be used in patterns as well, like in `first|Tail`.

Recursion

Mozart allows using for loops but they are a syntactic sugar. As variable are constants, we can not create true loops. What we can do is use recursion to iterate. There is nothing that loops can do that recursion can not do.

The compiler translates a for loop into a recursive procedure which is the best way to go through all the elements of a list or to iterate in general. The very constrained shape of lists allows us to get only two patterns: the list has two element so we go on iterating with the second element - recall that the second element is always a list - or the list is `nil` and we have reached the last iteration.

The principle is to create a function that calls itself - hence recursive. Of course, at some point this recursion should stop. As variable are declarative, we can use accumulators to accumulate the future result to return. As a list with N elements is in fact the nesting of $N+1$ lists, each iteration creates a list. The last iteration creates the empty list. Here is an example procedure that assigns `Next` to `List` times 2.

```
declare ProcTimes2 FunTimes2 AccTimes2 in
proc {ProcTimes2 List ?Next} % Next is bound to List times 2
   case List
   of nil then
      Next = nil % end of List, so end Next by assigning it to nil
   [] H|T then Nt in % declare new variable Nt
      Next = 2*H|Nt % Next is assigned to 2*List.1 followed by Nt
      {ProcTimes2 T Nt} % Recursive call to assign Nt
   end
end
```

```
% equivalent function
fun {FunTimes2 List}
    case List
    of nil then nil
    [] H|T then 2*H|{FunTimes2 T}
    end
end
% equivalent with an accumulator
fun {AccTimes2 List}
    fun {Aux L Acc}
        case L
        of nil then Acc
        [] H|T then {Aux T 2*H|Acc}
        end
    end
in
    {Reverse {Aux List nil}} % Aux result is reversed so unreverse it
end
```

The first two versions work in the same manner. The procedure is more explicit but the function is translated to the procedure by the compiler. The first iteration assigns the result to return to the first element followed by an unbound variable. The latter is then passed as the new result to the second iteration and so on.

The last function uses an accumulator. The principle is different. Each iteration creates a list with the first element being the current element of the iteration and the second being the accumulator. The result is similar but has a drawback - sometimes it can be an advantage - it reverses the order of the output. If the result is reversed then the drawback becomes that the program has to go through two lists instead of one - the function `Reverse` is similar to `AccTimes2` but without the multiplication.

In chapter 3 - about the functional transformations - we will how for loops are transformed into recursive procedures and get inspiration from that for comprehensions.

Threads

Mozart2 has been designed for concurrent applications. Threads are then easy to create. The following code shows how to create a thread.

```
% thread creation
thread
    ... % the body of the thread, what it executes
end
```

The `Wait` procedure makes execution go to sleep until its argument is bound. We can use this procedure to wait for a thread to be done as follows:

```
% wait for thread to finish
declare Done in % Done is assigned only when thread is done
thread
    ...
    Done = unit % thread is over, assign Done to unit
end
{Wait Done} % wait for thread to be done
```

As a variable can be assigned as many times as we want if it is always to the same value, we can extend the previous solution to wait for several threads to be done. Here is how:

```
% wait for one of the two threads to finish
local Done in
    thread
        ...
        Done = unit
    end
    thread
        ...
        Done = unit
    end
    {Wait Done}
end
```

In the previous examples, the body of a thread was a statement - a statement can be made of other ones. The body of a thread can also be an expression, it can return a value. Even if it might seem useless for now, we mention it because this will be used later. For now, here is a small example:

```
declare L in
L = [1 thread 1+1 end 3] % L = [1 _ 3] when thread is not done
% L = [1 2 3] when thread is done
```

Streams

A stream is an unbound list. For instance `1|2|3|_` is a stream. Why do we make this distinction ? A stream is very useful for concurrency because we can act on the part of the

stream already assigned and wait for the rest in parallel with other executions. Without streams, we need to wait for the end of the stream to begin acting on it. A concrete example is helpful to understand:

```
declare Produce Consume Xs Ys in
fun {Produce I N} % creates a stream from I to N
   if I =< N then I|{Produce I+1 N}
   else nil
   end
end
fun {Consume Xs} % returns a stream which is 2 times the input
   case Xs
   of nil then nil
   [] H|Ts then 2*H|{Consume Ts}
   end
end
thread Xs = {Produce 1 20} end
thread Ys = {Consume Xs} end
```

The nice thing with this code is that `Ys` is created before the generation of `Xs` is over. In the above example, it practically does not change anything but streams are usually much more longer or infinite so this concept is essential. Additionally, the producer might take a long time to create each element.

Consider now the following similar consumer:

```
declare Consume2 in
fun {Consume2 Xs}
   case Xs
   of nil then nil
   [] H|Ts then Next in
      Next = {Consume2 Ts}
      2*H|Next
   end
end
```

The only difference is that we explicitly state that we first compute the next elements of the output then append the double of the head to the result of the previous operation. The result is indeed the same as before but the difference makes this version very different from the other one. How ? For two reasons. The first is that line 6 will block until all the input stream is known. Indeed, the recursion will operate on line 6 until `nil` is reached, not before. The second reason is that Mozart has to keep in memory all the instances of the function because there is still an instruction - line 7 - after getting the next elements. This results in slower execution

and memory explosion if the input is too big.

The solution to this is to respect the property called *terminal recursion*. A recursive procedure - or function - is recursive terminal if and only if its only recursive call is the last instruction so that the calling procedure can be forgotten without any arm. The previous version of the consumer is recursive terminal because the compiler always transforms line 13 of `Consume` into line 7 *then* line 6 of `Consume2`

For our implementation of comprehensions, we need to keep in mind that respecting terminal recursion is mandatory to deal with streams and to avoid time and/or memory explosions.

Laziness

Now that we have seen streams, we can go further into concurrency. Laziness is a property of recursive functions. Until now every time a function is called, it creates its output eagerly or in other words, it never waits for some events to continue the recursion. A lazy function waits for its result to be needed before creating its output. Here is a detailed example:

```
declare Produce in
fun lazy {Produce I} % Produce is declared as lazy
   % basically Produce sleeps here until its result is needed
   I|{Produce I+1} % append an element then recursive terminal call
end
```

`Produce` will never finish. If it was not lazy then it would never stop creating without waiting and this will create an overhead as the output stream keeps growing. Fortunately `Produce` is lazy so it waits for its result the be needed by another thread to actually generate it. When the next element of the list is needed, `Produce` creates one element and then waits again after calling itself.

What makes the result needed ? Any computation directly using it makes it needed. We can also use the procedure `Value.makeNeeded` to explicitly make it needed.

Laziness is implemented using one procedure only, `WaitNeeded`, which does the all the job that consists in blocking until another thread calls `Value.makeNeeded` on the result or use it for calculations. Here is an example with `WaitNeeded` and `Value.makeNeeded`:

```
declare Produce Xs in
proc {Produce I ?Result} % Produce is not explicitly declared as lazy.
   {WaitNeeded Result} % Produce sleeps until Result is needed.
   Result = I|{Produce I+1} % Append an element then recursive terminal
                            % call. Here we treat Produce as a function.
end
```

```
thread Xs = {Produce 1} end % or {Produce 1 Xs}
% For now Produce waits and Xs = _
{Value.makeNeeded Xs} % Xs = 1|_
{Value.makeNeeded Xs.2} % Xs = 1|2|_
{Value.makeNeeded Xs.2.2} % Xs = 1|2|3|_
```

Laziness will show to be an essential possible property for list comprehensions.

Cells

Cells are not used in the implementation of comprehensions except for collectors - see chapter 3 - but as they are used in the compiler, we describe what they are. A cell is a variable that can be updated. They implement explicit state. They work as follows:

```
declare C in
C = {NewCell 0} % C is a Cell initiated with value 0
C := @C + 1     % assign using := and access via @, now C contains 1
```

Classes and objects

Classes and objects are not used in the implementation of comprehensions but as they are used in the compiler, we describe what they are. Classes are syntactic sugars. A class has a name, some attributes which are cells and some methods. Here is how it works along with an object declaration and use:

```
declare MyClass MyObject R1 in
class MyClass       % create class MyClass
    attr var1 var2 % as many attributes as wanted, atoms
    meth init()    % method called init with no arguments
        var1 := 1 var2 := 2
    end
    meth get1(R1) % method get1 with one argument
        R1 = @var1
    end
    meth get2($)  % method get2 is a function because of the $
        @var2
    end
end
MyObject = {New MyClass init()} % instance of MyClass and call init
{MyObject get1(R1)}             % R1 = 1
{Browse {MyObject get2($)}}     % Browsing 2
```

Functors

A functor is a structure that allows declaring a module. A module is an external resource that any code can import to use the module. Here is how to declare a functor:

```
functor MyModule % the name is optional
import
    System % import module System
    % import OtherModule from OtherModule.ozf in same directory
    OtherModule at 'OtherModule.ozf'
export
    proc1 : Proc1 % MyModule.proc1 points to Proc1
    proc2 : Proc2 % MyModule.proc2 points to Proc2
define
    proc {Proc1} ... end
    proc {Proc2} ... end
end
```

To compile a module in terminal, use

```
ozc -c Module.oz -o Module.ozf
```

after adding the binaries directory of Mozart2 in your path. You can then run the module - if it makes sense - by using

```
ozengine Module.ozf ozc-x Module.oz
```

1.2. List comprehensions

The previous section was about what Mozart2 already proposes. This short one is about what list comprehensions are and the terminology we will use.

Principle

As the name indicates, list comprehensions are able to comprehend, to *understand* a list - or a set - with its mathematical description. This means that they add the functionality to create lists easily similarly to their mathematical definition. There is a wide variety of ways to express them mathematically. Typically, a set is expressed as follows:

$$L = \{\text{pattern element} \mid \text{where the pattern comes from} : \text{conditions on pattern}\}$$

For instance,

$$L = \{(a, 2 * b) \mid \forall a \in A : \forall b \in [3, 5] : a > b\}$$

is the set of all the couples $(a, 2 * b)$ for all a coming from set A, for all b from 3 to 5 and only if $a > b$. If $A = \{1, 2, 4, 7\}$ then $L = \{(4, 6), (7, 6), (7, 8), (7, 10)\}$.

The basic idea is to allow this very flexible way of declaring any kind of lists in Mozart2. To express this, we will use a syntax inspired from Python:

```
% syntax
L = [Expression for ... if ... for ... if ...] % as many for ... if ...
% example
LA = [1 2 4 7]
L = [[A 2*B] for A in LA for B in 3..5 if A>B] % [[4 6] [7 6] [7 8] [7 10]]
```

The last example above uses several `for` because we nest the iteration on B inside the iteration on A. We also want to allow going through lists simultaneously like this:

```
% syntax
L = [Expression for ... ... if ...]
% example
LA = [1 2 4 7]
L = [[A 2*B] for A in LA   B in 3..5] % [[1 6] [2 8] [4 10]]
```

Of course these two possibilities - nested or simultaneous - can both be use in the same expression.

In addition we also want to be able to create as many lists as we want in one comprehension. To easily access all the resulting lists, we also want these output lists to be fields of an output record like this:

```
L = [1:Expression1 a:Expression2 for ...]
LA = [1 2 4 7]
L = [A a:B for A in LA B in 3..5] % '#'(1:[1 2 4] a:[3 4 5])
```

Again, all the previous functionalities must work together with the latter and also with the ability to filter the output lists separately like this:

```
L = [1:Expression1 if ... a:Expression2 if ... for ...]
LA = [1 2 4 7]
L = [A if A > 3 a:B for A in LA B in 3..5] % '#'(1:[4] a:[3 4 5])
```

So far, we showed examples with lists generated from other lists or from a start point to an end point. We also want to add the possibility to generate lists with functions, records and with a C-like generator. Here are examples:

```
% C-style
L = [A for A in 3 ; A<6 ; A+1] % [3 4 5]
```

```
% function
L = [A+B for A in 3 ; A<6 ; A+1 B from fun{$} 1 end] % [4 5 6]
% record
L = [F#A for F:A in rec(a:1 2:b)] % [a#1 2#b]
```

When a record in specified as generator, one must use the `_:_` notation. The first field is the feature and the second is the value of the field. The record is traversed in depth-first mode in case there are nested records - the record is a tree. Only the *leaves* of the tree are considered. To allow more flexibility we also want to give the user the possibility to choose which records should be treated as leaves instead of forcing leaves to be only the real leaves of the tree formed by the nested records. This is done by specifying a function taking two arguments, the record to test and its feature and returning a boolean, true if the record must be treated as a record, false otherwise. Here is an example:

```
L = [F#A for F:A in rec(1:1 a:r(a b) 2:2 of fun{$ F _} {IsInt F} end]
% [1#1 a#r(a b) 2#2]
```

All of these functionalities must work together.

Terminology

Here we define some terms that will be used in the chapters to come:

Range : A list, a stream, a function, a record or a generator that we go through using a `for` such as `LA` or `3..5` in the example.

Ranger : Every variable created into a `for` and that goes through a range such as `A` or `B` in the example.

Layer : A combination of a ranger and its range such as `A in LA` or `B in 3..5` in the example.

Level : A combination of all layers inside one given `for` and its condition if exists such as `for A in LA B in 3..5 if true`.

Creator : An expression before all `for` in a list comprehension - together with its feature is specified - such as `[A 2*B]` in the example.

Output specification : A creator together with its condition - if exists.

1.3. Record comprehensions

Similarly to lists comprehensions, this section describes what a record comprehension is.

Principle

The principle is basically the same as for list comprehensions except that instead of returning a list - or several ones - record comprehensions return a record - or several ones. The idea is to keep the same shape, the same arity as the input record. For this reason, record comprehensions only take one record as input. In other words, we could say that record comprehensions restrict list comprehensions to one layer and one level. On the other hand we still keep the multi output but without individual conditions, only the level condition can be specified.

As record comprehensions will use a very similar syntax as list comprehensions, we do not to change much. On the other hand, we need to differentiate them. This is done by using the keyword `through`.

The condition specified with `if` allows skipping some fields in the resulting record-s. The same kind of condition as in lists comprehensions on records to treat them as leaves or not can be specified but they are directly specified, not using a function. Here are some examples:

```
declare
Tree = tree(tree(leaf(1) leaf(2)) leaf(3))
R1 = [A for A through Tree] % R1 = tree(tree(leaf(1) leaf(2)) leaf(3))
R2 = [A+1 for A through Tree] % R2 = tree(tree(leaf(2) leaf(3)) leaf(4))
R3 = [F#A for F:A through Tree]
% R3 = tree(tree(leaf(1#1) leaf(1#2)) leaf(1#3))
R4 = [F#A for F:A through Tree of F == 1]
% R4 = tree(tree(leaf(1#1) 2#leaf(2)) 2#leaf(3))
R5 = [A for F:A through Tree if F == 1] % R5 = tree(tree(leaf(1)))
```

Terminology

Here we define some terms that will be used in the chapters to come, many definitions are the same as in list comprehensions.

Input : The record used as generator, the input of the record comprehension.

Ranger : The combination of the variables taking the current feature and value - the feature is optional. A ranger goes through the input.

Creator : An expression before the `for` in a record comprehension, together with its feature is specified.

Filter : The condition specified with `if` that decides which fields to drop in the traversal.

Decider : The condition specified with `of` that decides which fields to treat as leaves in the traversal.

1.4. Complete syntax

This section details the complete additional syntax brought by comprehensions. The complete syntax of Mozart2 can be found in the file *Grammar.html*. The HTML format has been used to allow users to follow definitions using links.

First here are some definitions to understand the rest of the syntax:

```
%% Definitions
plus(...)      % means that ... occurs at least once
star(...)      % means that ... occurs from 0 to infinity times
opt(...)       % means that ... occurs 0 or 1 time
alt(... ...)   % means 1 time any of the ... (as many ... as wanted in alt)
atom           % an atom
integer        % an integer
character      % a char
variable       % a variable, so an ID
feature        % alt(variable atom integer character)
lvl0           % any kind of expressions
subtree        % equivalent to "opt(feature :) lvl0"
phrase         % several successive lvl0
```

List comprehensions

We now detail what a list comprehension will have for syntax in Mozart2.

As we already saw shortly, list comprehensions return either a list or a record - which is not a list and which can be a tuple. Let us see incrementally the definition of the syntax we want to achieve for list comprehensions. While explaining the syntax we will also explain what is the purpose of each possible syntax. The list comprehension syntax is the following:

```
[plus(forExpression) plus(forComprehension)] % List comprehension
```

A list comprehension is delimited by square brackets. We can have one or more `forExpression`. A `forExpression` is made of a subtree and a condition, a subtree being a field of a record so either a value either a feature and its associated value. The condition is optional and allows to filter output. After these -this- `forExpression`, is one or more `forComprehension` which correspond to all the levels. The reason behind the `plus(forExpression)` is that our goal is to allow list comprehensions to return more than one output. We want them to be able to return a record if more than one output is given. Here are some examples:

```
%% L is a list
L = [A for A in 1..5] % L = [1 2 3 4 5]
L = [A for A in 1..5 if A<3] % L = [1 2]
L = [A if A<3 for A in 1..5] % L = [1 2]
%% L is a record of list-s
L = [1:A for A in 1..5] % L = '#'(1:[1 2 3 4 5])
L = [1:A for A in 1..5 if A<3] % L = '#'(1:[1 2])
L = [1:A if A<3 for A in 1..5] % L = '#'(1:[1 2])
L = [A 2*A for A in 1..5] % L = [1 2 3 4 5]#[2 4 6 8 10]
L = [a:A 2"A for A in 1..5] % L = '#'(1:[2 4 6 8 10] a:[1 2 3 4 5])
L = [1:1 2:2 3:3 for _ in 1..5] % L = [1 1 1 1 1]#[2 2 2 2 2]#[3 3 3 3 3]
```

A `forExpression` consists in an output specification, so an optional feature followed by one an expression and optionally by a condition `if`. The formal definition is below.

```
forExpression: % an output specification
   opt(feature:) lvl0 opt(if lvl0)
```

A `forComprehension` consists in a level, so a `for` followed by one or more `forListDecl` and optionally by a condition `if`. The formal definition is below.

```
forComprehension: % a level
   for plus(forListDecl) opt(if lvl0)
```

A level is composed of one or more layers. Each layer can have different range definitions and must have a different ranger. The following code defines what a layer can be.

```
1  forListDecl:alt( % a layer
2     lvl0 from lvl0 % 1st lvl0 is ranger, 2nd is range, a function here
3     lvl0 in forListGen % lvl0 is the ranger
4     lvl0 : lvl0 in lvl0 opt(of lvl0) % 1st lvl0 is feature, 2nd is ranger
5                                      % 3rd is range, a record here
6                                      % 4th is discriminate function
7     atom) % lazy
8  forListGen:alt( % a range
9     lvl0 .. lvl0 opt(; lvl0) % range from lvl0 to lvl0 by step 1 or lvl0
10    opt(lvl0 :) lvl0 % range is a list/stream first lvl0 for bounded buffer
11    alt((forGenC) forGenC))  % C-style range
12 forGenC: % a C-style range
13    lvl0 ; lvl0          % no condition
14    lvl0 ; lvl0 ; lvl0  % classic C-style
```

Line 7 corresponds to a `lazy` layer - the atom must be `lazy`. This makes the whole level lazy. Line 2 corresponds to a range generator being a function with no arguments, note that this range never stops by itself. Line 9 corresponds to a range generated by an start, an end and optional a step. Line 14 corresponds to a C-style range generator such that the ranger is initiated to the first `lvl0` and is assigned to the third one while the second `lvl0` is true - so it must be a condition. Line 13 is the same as line 14 except that the condition is set always true.

Line 4 corresponds to the range being a record. In this case, the list comprehension will go only through the leaves of tree formed by the nesting of records. An example of can be seen in figure 1.3, the red indicates leaves. The first `lvl0` can be used to keep track of the current feature of the visited field. The optional `lvl0` can be used to specify the function with two arguments - respectively the feature and the value of the field - returning true if the node has to be treated as a tree, false if it must be considered as a leaf.

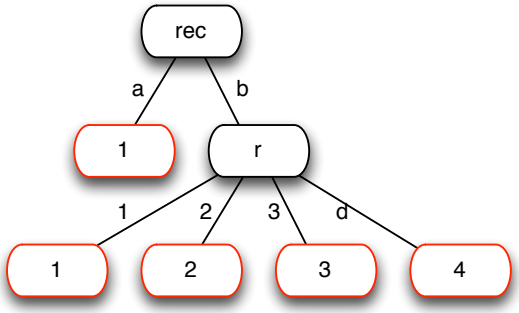

Figure 1.3: *Representation of the tree formed by rec(a:1 b:r(1 2 3 d:4). The red indicate the leaves.*

Line 10 corresponds to a range coming from a list or a stream. The first `lvl0` of this line - the optional one if exists or the only if not - is the list or stream. The second `lvl0`, if it exists, must be evaluated to a positive integer. This value is the requested size of the buffer for this range.

This functionality has one goal. In the case where the range is a stream lazily produced, we ask the producer of this stream to generate the next element only when we need it *in extenso* when it is the element of the current iteration. This can be a bottleneck if, for instance, the generation takes a while. To fight this, we can specify "how much" must the producer be in advance comparing to the output of the list comprehension. In other words, we can specify the number of elements that are needed before actually being used by the list comprehension. A representation is given in figure 1.4.

Here are some examples of how to use the different ranges:

```
declare
fun {Fun} 2 end
```

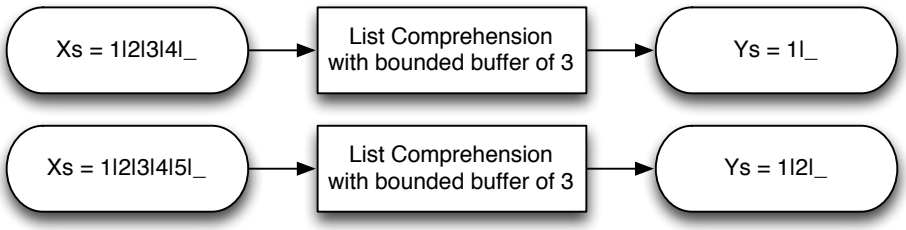

Figure 1.4: *Representation of a list comprehension with a bounded buffer.*

```
L1 = [B if B<3 1:A if A>1 C for A#B#C in [1#2#3 4#5#6]]
% L1 = [4]#[2]#[3 6]
L2 = [A#B for A from Fun B in 1..2] % L2 = [2#1 2#2]
L3 = [A for A in 1..10 ; 2]              % L3 = [1 3 5 7 9]
L4 = [A for A in 1 ; A<10 ; A+2]         % L4 = [1 3 5 7 9]
L5 = [A for A in [1 2 3 4 5]]            % L5 = [1 2 3 4 5]
L6 = [A for _:A in r(a:2 b:3 1)]         % L6 = [1 2 3]
L7 = [F for F:_ in r(1:1 a:3 2:2)]       % L7 = [1 2 a]
Xs = thread [A for lazy A in 1..10] end % Xs = _
L8 = thread [A for lazy A in Xs:3] end  % L8 = _ and Xs = 1|2|3|_
{Value.makeNeeded L8.2}                  % L8 = 1|2|_ and Xs = 1|2|3|4|5|_
L9 = [F#A for F:A in r(a:1 b:r(1 2))]   % L9 = [a#1 1#1 2#2]
L10 = [F#A for F:A in r(a:1 b:r(1 2)) of fun{$ F V} F \= b end]
% L10 = [a#1 b#r(1 2)]
```

The last two functionalities we are aiming for are the `collect` and the `body`. The latter consists in allowing some actions to be executed every time the list comprehension tries to add elements to the output. In other words, it can be seen as the body of the last nested for loop. For this we expand the syntax of list comprehensions as follows:

```
[plus(forExpression) plus(forComprehension) opt(body phrase)]
```

The optional keyword `body` delimitates any kind of actions that are executed every time the list comprehension tries to add elements. Here is an example:

```
{Browse [A for A in 1..2 body {Browse 1}]}
% browses:
% 1
% 1
% [1 2]
```

The last functionality, `collect`, uses the `body`. Collecting consists in assigning a procedure to a unbound variable given by the user. This procedure takes one argument. When executed,

typically in the body, it appends its argument to the list specified as output or this collector procedure. The collection is ended when the list comprehension is done. Here is the adaption needed for the collector:

```
forExpression:alt(
    feature : atom : lvl0
    opt(feature:) lvl0 opt(if lvl0)
    )
```

And here is an example:

```
L = [c:collect:C for A in 1..2 body {C A}{C A+1}]
% L = [1 2 2 3]
```

Record comprehensions

Because record comprehensions output a record - or records - similar to the input, we decided not to allow several levels. If one wants to use several layers then it would imply that all input records have similar - nested - arities. For these reasons, we decided to restrict record comprehensions to one level and one layer. As a direct consequence, the output makes no sense because it the same as the - unique - level condition. The collect operation is specific to lists so it is not implemented in record comprehensions.

Featured multi output is kept as well the possibility to use a range with a feature. In list comprehensions, one can specify a boolean function with two arguments to discriminate leaves when traversing a record. With record comprehensions, the same thing is possible but we decided to put the condition directly instead of putting in a function. This is because we think it is more unified with the - unique - level condition.

The exact syntax of record comprehensions is the following:

```
[plus(subtree) for opt(lvl0 :) lvl0 through lvl0 alt(
                                        (opt(if lvl0) opt(of lvl0))
                                        (of lvl0 if lvl0)
                                            )]
```

The alternative at the end allows users to choose the order in which they put the filter and the decider.

Here are some examples:

```
declare
L1 = [A for A through r(a:2 b:3 1)]          % L1 = r(1:1 a:2 b:3)
L2 = [A+1 for A through r(a:2 b:3 1)]        % L2 = r(1:2 a:3 b:4)
L3 = [F for F:A through r(a b c) of {Arity A} \= nil]
% L3 = r(1 2 3)
L4 = [F+1 for F:A through r(a b c) of {Arity A} \= nil]
% L4 = r(2 3 4)
L5 = [F#A for F:A through r(a:1 b:r(1 2))]   % L5 = r(a:a#1 b:r(1#1 2#2))
L6 = [F#A for F:A through r(a:1 b:r(1 2)) if {IsRecord A} orelse A > 1]
% L6 = r(b:r(2:2#2))
L7 = [F#A for F:A through r(a:1 b:r(1 2)) of F \= b]
% L7 = r(a:a#1 b:b#r(1 2))
L8 = [F#A for F:A through r(a:1 b:r(1 2)) if {IsRecord A} orelse A > 1
                                          of F \= b]
% L8 = r(b:b#r(1 2))
```

Chapter 2

The compiler of Mozart2

Compiling consists in transforming source code into code directly executable by a processor.

This chapter aims at understanding parts of the design of the compiler of Mozart2. Only the relevant parts for the implementation of comprehensions are detailed. All the sources are available at [20] and also in the directory *Code/mozart2*. In the sources of Mozart2, the compiler is in *lib/compiler*.

Mozart2 is compiled, not interpreted like Shell or Python. Interpreters take one instruction at a time and translates it into executable code. It is kind of an *online compilation*, in real time. Compilers allow more checking and more intelligence hidden in the analysis, optimizations and transformations. Another advantage of compilers is that it generally results in programs executed faster.

There are two big kinds of compilers. First are the compilers without virtual machines such as `gcc`[1] or `clang`[2] which is used to compile the compiler of Mozart2. Such compilers directly create an executable file that can be executed only on the architecture they have been compiled for. This property makes them very efficient in term of speed of execution. On the other hand, such a solution is not portable between different computer architectures.

The second kind is compilers that rely on a virtual machine. A virtual machine is a program that allows running a special kind of code no matter what the platform is. The idea is to compile the input code into this *virtual machine language*. Such an output can be executed on any platform using the virtual machine. The latter is responsible to offer a determined set of functionalities no matter what the platform is. The complexity of dealing with platform-specific instructions is abstracted by the virtual machine.

The last kind of compilers is what Mozart uses. It means there exists a virtual machine that runs Mozart2 executables. This machine is not our concern here. In the rest of this chapter we first explain the general architecture of a compiler that uses a virtual machine then we go more in details through relevant parts of the Mozart2 compiler for comprehensions. We also state how these parts must be adapted to accept the result of the next chapter about functional

[1] For more information, visit http://gcc.gnu.org/

[2] For more information, visit http://clang.llvm.org/

transformations.

2.1. Generic design of a compiler using a virtual machine

A generic compiler is divided into five modules or steps. The input of the first one, called a compilation unit - is the code to compile. The output of the fourth first steps is given as input to the next step. The output of the last level is the special *virtual machine code*.

Lexical analysis, the lexer

The first step in to read the input - typically a file or a block - and to tokenize it. This is done by the lexer. Tokenizing means that instead of taking characters as the smallest unit, tokens are the smallest unit. A token can be a keyword such as `fun, proc, if` or `thread`, a symbol such as `+, {, [` or `]`, an identifier - a variable - such as `Value, Browse` or `MyVariable`, an integer, a floating point number, a string a character or an atom - that is not a keyword. Space and new line characters are used to distinguish tokens but they are *forgotten* by the lexer afterwards. Comments are simply skipped by the lexer.

The keywords are atoms that are reserved for a purpose. Here is the complete list of all the keywords of Mozart2 in alphabetical order:

andthen	at	attr	case	catch	choice	class	cond
declare	define	dis	do	div	else	elsecase	elseif
elseof	end	export	fail	feat	finally	from	for
fun	functor	if	import	in	local	lock	meth
mod	not	of	or	orelse	prepare	proc	prop
raise	require	self	skip	then	thread	try	

The symbols are special characters. Here is the complete list of all the symbols of Mozart2:

(	)	[	]	{	}	|	#
:	...	=	.	:=	^	[]	$
!	_	~	+	-	*	/	@
< -	,	!!	<=	==	\=	<	=<
>	>=	=:	\=:	<:	=<:	>:	>=:
::	:::	..	;				

The output of this first module is an ordered set of tokens. The advantage to use a tokenizer abstraction is that we can now deal only with entities instead of characters, words or lines. If

an error occurs during this step, then the input contains unknown characters. No coherence between tokens is checked here.

An example of input can be:

```
for (initiator; condition; step) {
    if (condition) {
        i=1
    } else {
      false statement
    }
}
```

Which leads to the following set of tokens:

```
{keyword('for'), symbol('('), initiator, symbol(';'), condition, symbol(';'),
 step, symbol(')'), symbol('{'), keyword(if), symbol('('), condition,
 symbol(')'), symbol('{'), var('i'), symbol('='), int(1), symbol('}'),
 keyword('else'), symbol('{'), false statement, symbol('}'), symbol('}')}
```

Notice that all the spaces have disappeared as well as the new lines. The output does not remove any information and helps formalizing for the next step *exempli gratia* the keywords are labeled as such and not as atoms or strings.

Syntax analysis, the parser

From the flat set of tokens created by the lexer, the parser creates a abstract syntax tree which we will abbreviate by AST. The true structure of a program is generally reflected by its indentation. Indeed, an `if ... then ... else ... end` statement has three parts: the condition, the true statement and the false statement. So it would be a good idea to have a node corresponding to an if statement that has three children, one for each of its parts. Applying this reasoning to all the structures and declaring that a compilation unit is rooted at a root node, we get a single AST by compilation unit. This tree structure is much more expressive than a flat set structure.

Usually, in addition to its required parts, a structure also stores a position in the compilation unit. The reason for this is that in case of error, the compiler can locate easily the source of the error for the user. Without this position, no error could know where it comes from. An example of AST for the set of tokens in the lexer example can be found in figure 2.1. Keep in mind that it is just a generic example, the structure of nodes differs from on language to another.

If an error occurs during the syntax analysis, then the input does not respect at least one of the requirements imposed by the structures - recall the if node example.

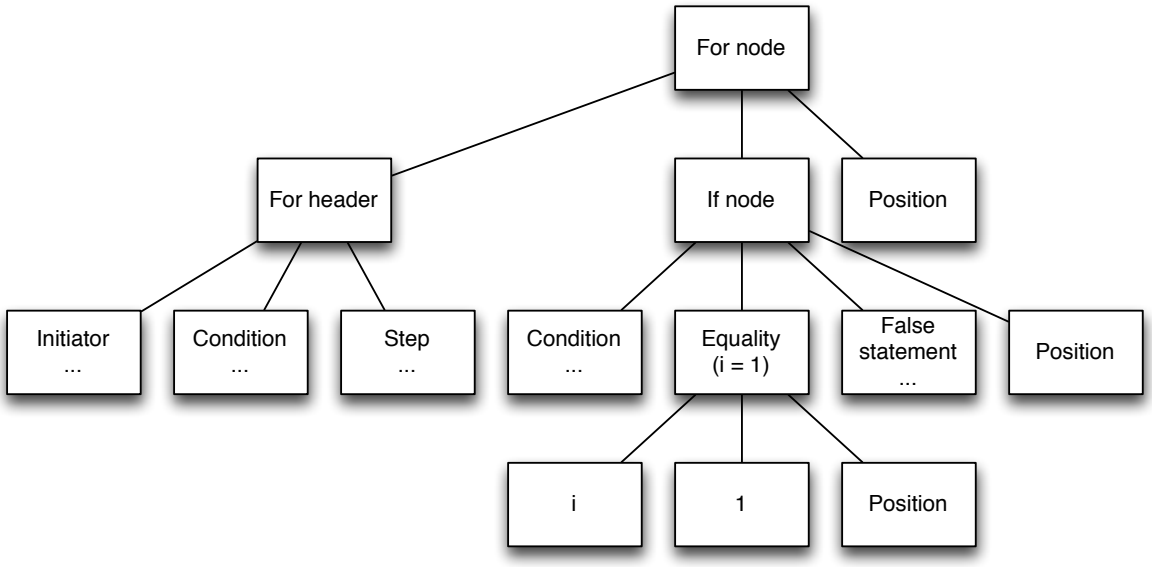

Figure 2.1: *An AST with an if statement encapsulated in a for loop. Three dots indicate a node that has been simplified, normally it should be a subtree.*

A parser defines a grammar in a comparable way that a lexer defines a lexical field. The grammar of Mozart2 can be found in the file *Grammar.html*. A grammar is defined by all its rules. Rules refer to each other. To choose the next node, the parser uses the rules to find the node corresponding to the rule. An example rule for an if statement could be:

```
keyword('if') symbol('(') condition symbol(')') symbol('{') statement
   symbol('}') keyword('else') symbol('{') statement symbol('}')
```

where `condition` and `statement` are other rules. This rule corresponds to the if node. The creation of the if node will recursively create the nodes for the other rules.

For more information on grammars, rules and parsing, we recommend [3].

Semantic analysis

Once the first AST is created, we have a basis for the steps to come. The semantic analysis consists in creating contexts or scopes and enforcing some language rules. The context or scope aims at knowing what variable are declared where and what is their type. Based on this analysis, the compiler can check the correctness of a program with respect to the use of variables. Subsequently, an error at this step is generated because of a variable misuse such as a wrong type, a redeclaration, a non-existence, *etc.*

So far, an intuitive understanding could be that the lexer is a smart reader which feeds the parser token by token - or batch of tokens by batch of tokens. The parser would then be a tool giving substance to the tokens, it gives *relief* to the flat sequence of tokens. The structure

arises from the parser. In this context, the semantic analysis can be considered as the first true intelligence of the compiler as it really into the code to ensure the correctness of the use of its content.

Optimization

This module is optional. Its goal is to make the AST more efficient using so generic or language specific strategies. For instance, a for loop iterating 10 times without any doubt - so all executions do exactly 10 iterations of this for loop - can be optimized by replacing the loop by 10 times the body of the loop. This is more efficient because it avoids the condition checking and the jumps in the code. Of course there exist many different optimizations there no general rules to ensure here.

Sometimes such optimizations are not always at this moment of the compilation. Recall that our goal here is to have a general idea of the whole process of compiling when using a virtual machine.

Code generation

The code generation is the final step of the compiler. It consists in the transformation of the AST accompanied by its contexts - or scopes - into the language specific to the target virtual machine. This step hardly depends on the virtual machine used.

One could decide to directly write code using this specific language but this shows to be really hard and generally not straightforward. Such a language does not aim at being written by humans, it aims at being systematically generated by a compiler and then at being executed by the right virtual machine.

Mozart2

Mozart2 is based on a relatively small syntax called the core language. Many possibilities that Mozart2 offers are actually syntactic sugars. It means that these expressions or statements are transforms by the compiler into structures respecting the core language. The latter is detailed in [11]. Some syntactic sugars were explained in the previous chapter. Like for loops or functions, lists comprehensions are syntactic sugars.

The compiler of Mozart2 is more complex than this generic compiler because of all the possibilities it offers. For a more detailed analysis, we recommend [2]. Briefly, the AST created by the parser is passed through several passes than modify it. One of this pass is the transformations of syntactic sugars into their equivalent unsugared structures. This is done is the unnester which will be developed in the coming section.

2.2. The abstract syntax tree of Mozart2

The main structure of the compilation is the one that we are interested in. It is the AST. In Mozart2, nodes are tuples. The label indicates of what type is the node while the fields are all nodes parts along with a position for some nodes. As tuples are used, the order of the children is important and it often following an intuitive logic. The label always starts with an 'f' expect the position nodes. Here is a small example in figure 2.2. The remaining of this section describes the relevant existing nodes of the AST of Mozart2. The next section describes the new nodes we will use for comprehensions.

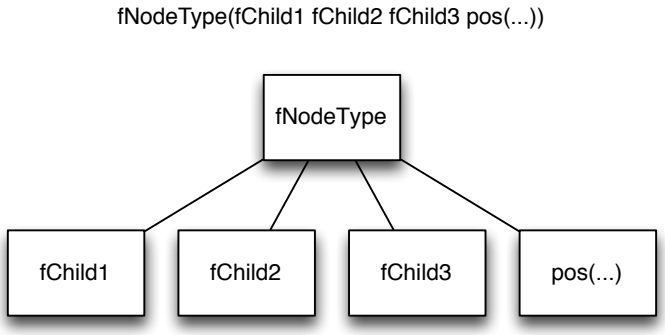

Figure 2.2: *Small Mozart2 AST example: node fNodeType with three children and a position.*

Positions

Also referred as coordinates, positions exist in three shapes: delimitation, unique or empty. Here is the syntax:

```
% delimitation
% from file F1 at line L1 column C1 to file F2 line L2 column C2
pos(F1 L1 C1 F2 L2 C2)
% example
pos("file.oz" 10 14 "file.oz" 10 17)

% unique
% at file F at line L column C
pos(F L C)
% example
pos("file.oz" 10 14)

% empty: position does not because created by compiler
unit
```

For simplicity, the explanations for the other structures will always an empty position.

Basic types

The basic types we will use are atoms, integers, and variables - strings are in fact lists so declared as such. Here is the syntax:

```
% atom, integer and variable have the same structure: f...(... Pos)
% Examples
fAtom('iAmAnAtom' unit) % atom 'iAmAnAtom' with empty position
fInt(1 unit)            % integer 1 with empty position
fVar('MyVar' unit)      % variable MyVar with empty position
```

Equality

Equalities consist in value assignments. Note that, thanks to declarativity, left and right members have the same status. Here is the syntax:

```
% equality between nodes A and B with empty position
fEq(A B unit)
```

Skip statement

The skip statement is a statement that does nothing so its node is very simple. Here is the syntax:

```
% skip statement with empty position
fSkip(unit)
```

Local declarations

Local declarations are made of two parts. The declarations part states the new variable to declare and the body part defines the scope of the freshly declared variables. Here is the syntax:

```
% syntax
fLocal(Declarations Body Position)
% example: declare A with empty position with undefined Body
fLocal(fVar('A' unit) ... unit)
```

Operations

Operations regroup all the transformations that can be applied to variables such as addition, field selection, *etc.* The operation is the first field - stated as an atom - and the second is a list of operands - the order of the elements matters. Here is the syntax:

```
% syntax
fOpApply(Operation [Operands] Position)
% example: A+2 with empty position
fOpApply('+' [fVar('A' unit) fInt(2 unit)] unit)
% example: A.1-2 with empty position
fOpApply('-'
         [fOpApply('.' [fVar('A' unit) fInt(1 unit)] unit)
          fInt(2 unit)]
         unit)
```

Procedure and function calls

The first thing to state, the first field, is the name of the procedure - the variable containing it. As the number of arguments can be anything, the arguments are put inside a list where the order matters. This list is the second part. Here is the syntax:

```
% syntax
fApply(Procedure [Arguments] Position)
% example: {Produce 1 10} with empty position
fApply(fVar('Produce' unit) [fInt(1 unit) fInt(10 unit)] unit)
```

Procedure and function declarations

The declaration of a procedure requires four parts - plus the position. The first is the name, the variable containing the procedure. The second is the list of arguments for which the order matters. The third is the body of the procedure. The fourth is a list of flags. Here is the syntax:

```
% syntax
fProc(Procedure [Arguments] Body [Flags] Position)
fFun(Function [Arguments] Body [Flags] Position)
% example: proc {Produce A B} ... end with empty position
fProc(fVar('Produce' unit) [fVar('A' unit) fVar('B' unit)] ... nil unit)
% example: fun lazy {Produce} ... end with empty position
fFun(fVar('Produce' unit) nil ... [fAtom('lazy' unit)] unit)
```

If statements

If statements have three parts: the condition, the true statement and the false statement. An useful operator is the one that takes the conjunction of two conditions. Here are the syntaxes of if statements and conjunctions:

```
% syntaxes
fBoolCase(Condition True False Position)
fAndThen(Condition1 Condition2 Position)
% example: if true then 1 else 0 end with empty position
fBoolCase(fAtom(true unit) fInt(1 unit) fInt(0 unit) unit)
% example: true andthen false with empty position
fAndThen(fAtom(true unit) fAtom(false unit) unit)
```

Threads

Threads only have a body and a position. Note that position can not be empty for threads. Here is the syntax:

```
% syntax
fThread(Body Position)
% example: thread 1 end at position "file.oz" line 1 column 10 to 22
fThread(fInt(1 unit) pos("file.oz" 1 10 "file.oz" 22))
```

Successive statements

A important thing is when there are successive statements. So far, we have not seen how to handle them. As usual, Mozart2 uses a recursive definition that allows to concatenate two statements. No position are used since this concatenation two by two does not appear in the input code. Here is the syntax:

```
% syntax
fAnd(Statement1 Statement2)
% example: A=1 B=2
fAnd(fEq(fVar('A' unit) fInt(1 unit) unit) fEq(fVar('B' unit) fInt(2 unit)
    unit))
% example: A=1 B=2 C=3
fAnd(fEq(fVar('A' unit) fInt(1 unit) unit)
     fAnd(fEq(fVar('B' unit) fInt(2 unit) unit)
          fEq(fVar('C' unit) fInt(3 unit) unit))
    )
```

Records, tuples and lists

As lists are tuples and tuples are records, these three structures use the same node. The first field is the label. The second is an ordered list of the fields. As a record has fields with features, there is another node to link the feature to its value. When features are integers from 1 to the length of the record then we can omit the feature. Here are the syntaxes:

```
% syntaxes
fRecord(Label [Fields] Position)
fColon(Feature Value)
% example: record(a:1 2:b) with empty position
fRecord(fAtom(record unit)
        [fColon(fAtom(a unit) fInt(1 unit))
         fColon(fInt(2 unit) fAtom(b unit))]
        unit)
% example: tuple(1:a) with empty position
fRecord(fAtom(tuple unit)
        [fColon(fInt(1 unit) fAtom(a unit))] unit)
% example: a#b#c with empty position
fRecord(fAtom('#' unit)
        [fColon(fInt(1 unit) fAtom(a unit))
         fColon(fInt(2 unit) fAtom(b unit))
         fColon(fInt(3 unit) fAtom(c unit))]
        unit)
% example: [1] with empty position
fRecord(fAtom('|' unit)
        [fInt(1 unit) fAtom(nil unit)]
        unit)
% example: [1 2] with empty position
fRecord(fAtom('|' unit)
        [fInt(1 unit)
         fRecord(fAtom('|' unit) [fInt(2 unit) fAtom(nil unit)] unit)]
        unit)
% example: [1 2 3] with empty position
fRecord(fAtom('|' unit)
        [fInt(1 unit)
         fRecord(fAtom('|' unit)
                 [fInt(2 unit)
                  fRecord(fAtom('|' unit)
                          [fInt(3 unit) fAtom(nil unit)]
                          unit)]
                 unit)]
        unit)
```

Layers and range generators

A layer is the combination of a ranger and its range - see chapter 1. A range is generated using a range generator. Depending on the kind of range generator we have, we use different nodes. Here are the syntaxes:

```
% syntaxes
forPattern(Range Generator) % layer generated with the keyword 'in'
forFrom(Range Generator)    % layer generated with the keyword 'from'
forGeneratorC(Init Condition Step) % C-style generator
forGeneratorInt(From To Step)      % Ints generator
forGeneratorList(List)             % List/stream generator
% example: A in 1..2 (no step so unit is used as Step)
forPattern(fVar('A' unit)
           forGeneratorInt(fInt(1 unit) fInt(2 unit) unit))
% example: A in 1..2 ; 3
forPattern(fVar('A' unit)
           forGeneratorInt(fInt(1 unit) fInt(2 unit) fInt(3 unit)))
% example: A in 1;true;A+1
forPattern(fVar('A' unit)
           forGeneratorC(fInt(1 unit)
                         fAtom(true unit)
                         fOpApply('+' [fVar('A' unit) fInt(1 unit)] unit)
                         ))
% example: A in 1;A+1 (when no condition, Step is unit)
forPattern(fVar('A' unit)
           forGeneratorC(fInt(1 unit)
                         fOpApply('+' [fVar('A' unit) fInt(1 unit)] unit)
                         unit
                         ))
% example: A in List
forPattern(fVar('A' unit) forGeneratorList(fVar('List' unit)))
% example: A in [1]
forPattern(fVar('A' unit)
           forGeneratorList(fRecord(fAtom('|' unit)
                                    [fInt(1 unit) fAtom(nil unit)]
                                    unit)))
% example: A in {Produce}
forPattern(fVar('A' unit)
           forGeneratorList(fApply(fVar('Produce' unit) nil unit)))
% example: A from Function
forFrom(fVar('A' unit) fVar('Function' unit))
```

For loops

For loops have two parts in addition to their position. The first is a list of layers, the second is the body. Note that layers also be special features of for loops like break or continue. A layer can also be a flag like lazy. Here are the syntaxes:

```
% syntaxes
fFOR([Layers] Body Position)
forFeature(Feature Variable)
forFlag(Flag)
% example: for [...] do skip end with no position
fFOR([...] fSkip(unit) unit)
% example: for A from F do skip end with no position
fFOR([forFrom(fVar('A' unit) fVar('F' unit))] fSkip(unit) unit)
% example: for break:B A from F do skip end with no position
fFOR([forFeature(fAtom(break unit) fVar('B' unit))
      forFrom(fVar('A' unit) fVar('F' unit))]
     fSkip(unit)
     unit)
% example: for lazy A in [1] do skip end with no position
fFOR([forFlag(fAtom(lazy unit))
      forPattern(fVar('A' unit)
                 forGeneratorList(fRecord(fAtom('|' unit)
                                          [fInt(1 unit) fAtom(nil unit)]
                                          unit)))]
     fSkip(unit)
     unit)
```

Step Points

This node is used by the code generator and also as the root of for loop transformations to encapsulate the coordinates. This node has three parts: the body, the tag and a position. Here is the syntax:

```
% syntax
fStepPoint(Body Tag Position)
% example: desugared for loop with no position
fStepPoint(... % the transformation
           'loop'
           unit)
```

2.3. New nodes of AST for comprehensions

For this section, we describe the six new nodes to add in order for the AST to contain list comprehensions and the new node for record comprehensions. We start with list comprehensions from the most specific one since the more general one - the one containing a whole list comprehension - depends on these. Most of the nodes we use are already declared in the previous section. We reuse many nodes coming from ranges and for loops. Indeed layers already have nodes to describe them: `forPattern`, `forFrom` and `forFlag`.

The last node described in this section is the new one needed for record comprehensions.

Bounded buffers

When a bounded buffer is specified we need to store the size of the buffer with the stream concerned. Here is the syntax:

```
% syntax
fBuffer(Stream Size)
% example: Xs:10
fBuffer(fVar('Xs' unit) fInt(10 unit))
```

Ranges generated by records

When using a record as range, we need a special node. Since this functionality does not exist in for loops, we created it. This node requires four children. One for the current feature of the field, one for the value of the field, one for the record and one for the optional condition on nested records. Here is the syntax:

```
% syntax
forRecord(Feature Value Record Condition)
% example: F:A through Rec
forRecord(fVar('F' unit) fVar('A' unit) fVar('Rec' unit) unit)
% example: _:A through Rec of ...
forRecord(fWildcard(unit) fVar('A' unit) fVar('Rec' unit) ...)
```

Expressions

As we can specify a condition for each output, we need a structure to contain both the expression and its optional condition - set to `unit` if not given. Here is the syntax:

```
% syntax
forExpression(Expression Condition)
% example: A
forExpression(fVar('A' unit) unit)
% example: a:A
forExpression(fColon(fAtom(a unit) fVar('A' unit)) unit)
% example: a:A if ...
forExpression(fColon(fAtom(a unit) fVar('A' unit)) ...)
```

Collects

We have just seen how the expressions - to output - are handled in the AST. However we did not describe how to deal with collect operations. This is because it requires an specific node. For this we reuse the node `forFeature` with two children but we change a bit their meaning. This first element is the collect atom and the second is the combination of the feature and the collector. Here is the syntax:

```
% syntax
forFeature(Collect FeatureCollector)
% example: c:collect:C
forFeature(fAtom(collect unit) fColon(fAtom(c unit) fVar('C' unit)))
```

List comprehension levels

Levels need to be describe as nodes. They must contain a list of layers - the layers of this level - and the condition of the level. When no condition is given, we put `unit`. They also have a position. Here is the syntax:

```
% syntax
fForComprehensionLevel([Layers] Condition Position)
% example: for A in L with no position
fForComprehensionLevel([forPattern(fVar('A' unit)
                                   forGeneratorList(fVar('L' unit)))]
                       unit unit)
% example: for A from F with no position
fForComprehensionLevel([forFrom(fVar('A' unit) fVar('F' unit))] unit unit)
% example: for lazy B in LB:3 if true with no position
fForComprehensionLevel([forFlag(fAtom(lazy unit))
                        forPattern(fVar('B' unit)
                                   forGeneratorList(
                                        fBuffer(fVar('LA' unit)
                                                fInt(3 unit))
```

```
                        ))]
                       fAtom(true unit)
                       unit)
```

List comprehensions

Finally we need to add the node containing a whole list comprehension. This node must contain all the expressions to output, so for this we use a list of expressions - each expression might have a feature. The other thing to have is list of all the levels in the list comprehension. Finally there is a position. Here is the syntax:

```
% syntax
fListComprehension([Expressions] [Levels] Position)
% example: [A for ...] with no position
fListComprehension([forExpression(fVar('A' unit) unit)] [
    fForComprehensionLevel(...)] unit)
% example: [A B for ... for ...] with no position
fListComprehension([forExpression(fVar('A' unit) unit)
                    forExpression(fVar('B' unit) unit)]
                   [fForComprehensionLevel(...)
                    fForComprehensionLevel(...)]
                   unit)
% example: [A#B for ...] with no position
fListComprehension([forExpression(
                        fRecord(fAtom('#' unit)
                                [fVar('A' unit) fVar('B' unit)]
                                unit)
                        unit)]
                   [fForComprehensionLevel(...)]
                   unit)
% example: [fun:{F A} if ... for ...] with no position
fListComprehension([forExpression(
                        fColon(fAtom(fun unit)
                               fApply(fVar('F' unit) [fVar('A' unit)] unit)
                              )
                        ...)]
                   [fForComprehensionLevel(...)]
                   unit)
```

A representation of a list comprehension AST is in figure 2.3.

Record comprehensions

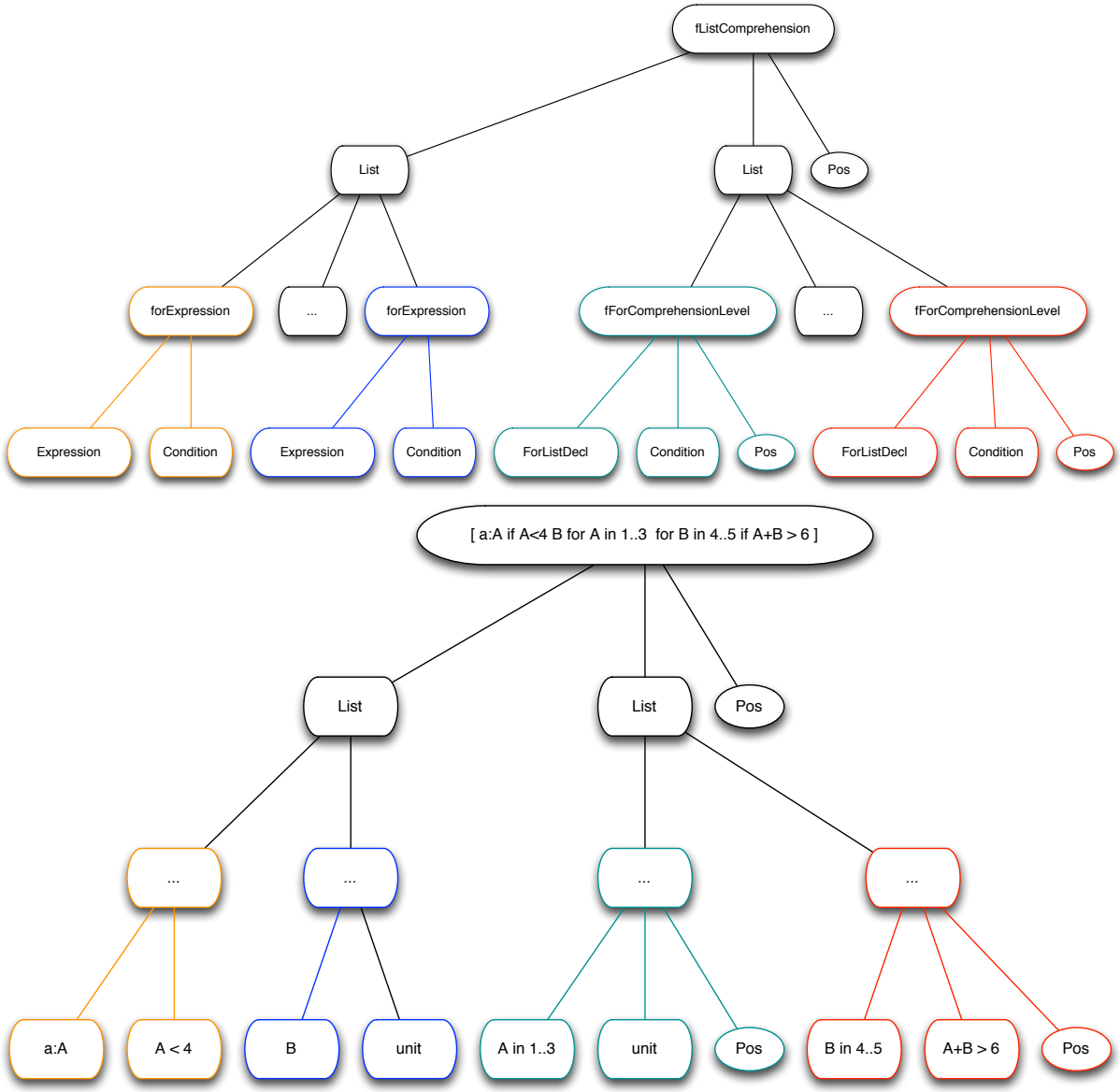

Figure 2.3: *Representation of an example of AST for a list comprehension.*

As record comprehensions imply only one level, one layer and no output condition, they are much simple to express. The new node needs four children in addition to the position: the outputs - there can several so a list, the ranger, the record and an optional condition which is a function with one argument.

```
% syntax
fRecordComprehension([Outputs] Ranger Record Filter Condition Position)
% example: A for F:A through Rec if ... with no position
fRecordComprehension([fVar('A' unit)] fColon(fVar('F' unit) fVar('A' unit))
    fVar('Rec' unit) ... unit unit)
% example: a:A for A through Rec of ... with no position
fRecordComprehension([fColon(fAtom(a unit) fVar('A' unit))]
```

```
                    fVar('A' unit) fVar('Rec' unit) unit ... unit)
```

2.4. Syntactic sugars compilation in Mozart2

This section details the parts of the compiler that have to change in order to implement comprehensions. The real intelligence in implementing comprehensions is the their transformation - the syntactic sugar - into recursive procedures. The idea is to have a powerful procedure that does the transformation. This procedure will be called by the unnester.

Comprehension transformations can be considered as two modules so we will implement them inside two new files called *ListComprehension.oz* and *RecordComprehension.oz*. To include these files in the compilation process, we need to add their name into the structure `COMPILER_FUNCTORS_0` of the file *mozart2/lib/CMakeLists.txt*.

In order to reach this call and to make all the compilation process work, some adaptations are required in addition to the transformations. These adaptations are detailed in this section along with the reasons for these changes.

Macros

Macros are special instructions that one can use to change the state of the compiler or the state of the running Mozart2. We do not go into details since macros are not relevant to explain for lists comprehensions. The only thing that matters here is that every node of an AST can be asked if it contains any macro. This is done in *Macros.oz*.

The adaptation required for macros is to add the rules of checking whether the new node types contain a macro. This is done by recursively checking the children of the nodes if the node can contain a macro. Some nodes which are destined to be small can not contain macros. For instance, position nodes can not contain macros. Here are the adaptations:

```
fun {ContainsMacro E} % true iff node E contains at least a macro
   case E
   ...
   [] fListComprehension(_ _ _ _) then      % fListComprehension can not
      false                                 % contain macros
   [] fForComprehensionLevel(_ _ _) then    % fForComprehensionLevel
      false                                 % can not contain macros
   [] forExpression(_ _) then               % forExpression can not
      false                                 % contain macros
   [] fBuffer(_ _) then                     % fBuffer can not
```

```
        false                                    % contain macros
    [] forRecord(_ _ _ _) then                   % forRecord can not
        false                                    % contain macros
    [] fRecordComprehension(_ _ _ _ _ _) then % fRecordComprehension
        false                                    % can not contain macros
    ...
    end
end
```

The lexer, lexical analysis

The syntax previously seen at the end of chapter 1 does not use any extra keyword or symbol except `through` and `body` so we only need to add them as keywords and not a simple atoms. In order to add the latter keywords, we just need to specify them in the list of keywords called `OzKeywords` in the file *Lexer.oz* where the lexer is located. The lexer does the rest of the lexical analysis for us.

The parser, syntax analysis

The parser defines all syntactic rules. The format of a rule is the following:

```
ruleName:modifier(rule)
% where modifier can be one of (non-exhaustive)
alt(rule ...) % one of the rules, returns the node of the chosen rule
plus(rule)    % at least 1 time the rule, returns a list of nodes
star(rule)    % rule as many times as wanted, returns a list of nodes
opt(rule no)  % 0 or 1 time the rule, returns node if rule, no otherwise
seq2(wd rule) % atom wd followed by rule, returns node of rule
seq1(rule wd) % rule followed by atom wd, returns node of rule
% where rule is
modifier(rule)  % recursive definition
[rule ... rule] % an ordered set of rules
otherRule       % another rule
token           % a token
% along with the definition of a rule, we give a function
% called when rule is chosen by parser that returns
% the corresponding node

% EXAMPLE
% pB: rule for beginning position
% pE: rule for
lvl6: % rule named lvl6
    alt(% rule is one of the followings
```

```
        [lvl7 pB '|' pE lvl6] % lvl7|lvl6 --> list
       % rule tupled with the function returning the node
       #fun{$ [S1 P1 _ P2 S2]} % the argument is matched to the rule
           % create a node fRecord
           fRecord(fAtom('|' {MkPos P1 P2}) [S1 S2])
        end
       lvl7 % next level, no function because function in lvl7
      )
```

A list comprehension returns a record. So we need to put a new rule at the right place to create a record. A record comprehension also returns a record so the new rule goes at the same place. Here are these new rules:

```
functor
...
define
    ...
    Rules =
     g(
        ...
       % record rule
       atPhrase:alt(
            ... % others atPhrase rules
            [pB '[' plus(forExpression) forComprehension
                                  opt(seq2('body' phrase) unit) ']' pE]
           #fun{$ [P1 _ S1 FC BD _ P2]}
                % create fListComprehension node
                % plus returns a list so S1 directly
                fListComprehension(S1 FC BD {MkPos P1 P2})
            end
            [pB '[' plus(subtree) 'for' opt(seq1(lvl0 ':') unit) lvl0
                'through' lvl0 opt(seq2('if' lvl0) unit)
                               opt(seq2('of' lvl0) unit) ']' pE]
           #fun{$ [P1 _ S _ F L1 _ L2 IF OF _ P2]}
                if F == unit then
                    fRecordComprehension(S L1 L2 IF OF {MkPos P1 P2})
                else
                    fRecordComprehension(S fColon(F L1) L2 IF OF
                                         {MkPos P1 P2})
                end
            end
            [pB '[' plus(subtree) 'for' opt(seq1(lvl0 ':') unit) lvl0
                'through' lvl0 'of' lvl0 'if' lvl0 ']' pE]
           #fun{$ [P1 _ S _ F L1 _ L2 _ OF _ IF _ P2]}
```

```
                if F == unit then
                    fRecordComprehension(S L1 L2 IF OF {MkPos P1 P2})
                else
                    fRecordComprehension(S fColon(F L1) L2 IF OF
                                         {MkPos P1 P2})
                end
            end
            ...
          )
        ...
      )
    ...
end
```

Record comprehensions do not need any extra definitions since all they need are inside the rule defined just above so the rest of the rules are all for lists comprehensions.

Sometimes the output of a list comprehension is a list - or a tuple - it means that the expression part of the list comprehension is made of one expression with no feature. This does not change anything else. One could be tempted to change list or tuple specific rules but since we do not use their syntactic sugars, we just have to change the rule of records as previously done.

Now we still need to define the two rules used in the previous definition. The first new rule is `forExpression`. It is greatly inspired from fields of records mixed with an optional condition.

The second new rule used, `forComprehension`, is inspired from the ones used by for loops but they differ a bit because list comprehensions can have bounded buffers, go through records and because they can not have the same layers as for loops like `break:Break`. Note that the order of the alternatives are important since pattern are tested in order until a match.

```
forExpression:alt( % an output specification
    [feature ':' atom ':' lvl0]
    #fun{$ [F _ A _ L]}forFeature(A fColon(F L))end
    [subtree opt(seq2('if' lvl0) unit)]
    #fun{$ [S1 S2]}forExpression(S1 S2)end
    )
forComprehension: % a level (multi layer and optional condition)
    plus([pB 'for' plus(forListDecl) opt(seq2('if' lvl0) unit) pE])
    #fun{$ Ss} % Ss is the list of nodes returned by plus(...)
        % each element becomes a fForComprehensionLevel node
        {FoldR Ss fun{$ Xn Y}
                      case Xn of [P1 _ FD CD P2] then
```

```
                        fForComprehensionLevel(FD CD {MkPos P1 P2})|Y
                        [] nil then Y
                        end
                    end
        nil}
    end
forListDecl:alt( % a layer
    [lvl0 'in' forListGen]                          % generated by in
    #fun{$ [A _ S]}forPattern(A S)end
    [lvl0 'from' lvl0]                              % generated by from
    #fun{$ [A _ S]}forFrom(A S)end
    [lvl0 ':' lvl0 'in' lvl0
        opt(seq2('of' lvl0) unit)]                  % generated by record
    #fun{$ [F _ A _ R OF]}forRecord(F A R OF)end
    atom                                            % lazy flag
    #fun{$ A}forFlag(A)end
    )
forListGen:alt( % range generator (in)
    [lvl0 '..' lvl0 opt(seq2(';' lvl0) unit)] % Ints
    #fun{$ [S1 _ S2 S3]}forGeneratorInt(S1 S2 S3)end
    ['(' forGenC ')'] % C-style
    #fun{$ [_ S _]}S end
    forGenC             % C-style
    [lvl0 ':' lvl0]   % bounded buffer
    #fun{$ [S1 _ S2]}forGeneratorList(fBuffer(S1 S2))end
    lvl0                % stream/list
    #fun{$ S}forGeneratorList(S)end
    )
```

Checking the syntax

Mozart2 checks for the syntax of its AST. This is essential to spot structural errors and also avoid pattern matchings from not finding any match. Indeed, if a node - a tuple - does not have the right syntax then it might cause the compiler to crash since the pattern does not have the right number of fields. This syntax checking is done in *CheckTupleSyntax.oz*. To ensure that the compiler accepts the new nodes, we need to add them in the syntax checker. Here are the adaptations:

```
proc {Phrase X} % check syntax
    case X
    ...
    [] fListComprehension(E Fs Bd C) then
        {ForAll E Phrase}  % check all expressions
```

```
         {ForAll Fs Phrase} % check all levels
         {Phrase Bd}        % check body
         {Coord C}          % check coordinates
      [] fForComprehensionLevel(RG CD C) then
         {ForDecl RG}       % ForDecl checks range generator
         {Phrase CD}        % check condition
         {Coord C}          % check coordinates
      [] forExpression(E C) then
         {Phrase E}         % check expression
         {Phrase C}         % check condition
      [] fBuffer(Xs N) then
         {Phrase Xs}        % check list/stream
         {Phrase N}         % check buffer size
      [] forRecord(F A R Fc) then
         {Phrase F}         % check feature
         {Phrase A}         % check ranger
         {Phrase R}         % check record
         {Phrase Fc}        % check record
      [] fRecordComprehension(S A R F Cd C) then
         {Phrase S}         % check all expressions
         {Phrase A}         % check ranger
         {Phrase R}         % check record
         {Phrase F}         % check filter
         {Phrase Cd}        % check condition
         {Coord C}          % check coordinates
      ...
      end
end
```

Coordinates in case of failure

In case of error, we must ensure that nodes can be located by the compiler in order for it to provide an useful error report. Mo The file *TupleSyntax.oz* implements a function returning the coordinates of a given node. Of course, we must handle the new nodes in this function. Here is the adaption:

```
% returns the coordinates of the outermost leftmost construct
fun {CoordinatesOf P}
   case P
   ...
   [] fListComprehension(_ _ _ C) then
      C                  % coordinates are inside the node
   [] fForComprehensionLevel(_ _ C) then
```

```
        C                   % coordinates are inside the node
    [] forExpression(E _) then
        {CoordinatesOf E} % coordinates are the ones of the expression
    [] fBuffer(E _) then
        {CoordinatesOf E} % coordinates are the ones of the list/stream
    [] forRecord(F _ _ _) then
        {CoordinatesOf F} % coordinates of feature
    [] fRecordComprehension(_ _ _ _ _ C) then
        C                   % coordinates are inside the node
    ...
    end
end
```

The unnester, transforming syntactic sugars

As stated above, the unnester calls the procedure transforming comprehension syntactic sugars into an desugared expression. The following adaptations are done in the file *Unnester.oz*.

```
functor
import
    ...
    ListComprehension(compile)   % import the procedure in
   ListComprehension
    RecordComprehension(compile) % import the procedure in
   RecordComprehension
    ...
...
define
    ...
    % call _Comprehension.compile when f_Comprehension encountered
    class Unnester
        ...
        meth UnnestExpression(FE ToGV $)
            case FE
            ...
            [] fListComprehension(_ _ _ _) then
                Unnester, UnnestExpression(
                        {ListComprehension.compile FE} ToGV $)
            [] fRecordComprehension(_ _ _ _ _ _) then
                Unnester, UnnestExpression(
                        {RecordComprehension.compile FE} ToGV $)
            ...
            end
```

```
            ...
      end
      ...
end
```

Keywords in Emacs

Another change to make in order to get a syntactic coloration in Emacs is to specify the new keywords `through` and `body`. This is done by adding them as keywords in the file *opi/emacs/oz.el*.

Chapter 3

Functional transformations

This chapter is the main goal of this Master's thesis. It contains all the functional transformations of comprehensions into procedures and how we implemented them all.

3.1. Functional transformations

In this section, first we will explain how for loops are transformed and get inspiration from that because for loops are similar to list comprehensions with one level. Then we will incrementally see how to transform all the functionalities of list comprehensions. At the end, we will see transformations needed for record comprehension.

An inspiration, for loops examples

As stated in the previous chapter, for loops transformations are encapsulated inside an AST node called `fStepPoint`. What interests us is the its first child, the actual transformation. As we can not use explicit state *in extenso* cells, we have to create a recursive procedure to fake the iterations of a for loop. The idea for this procedure is to have as argument all the information needed to handle the iterations.

There are four different range generators for loops accept. The first one is when a list is given, either directly or inside a variable. Here is an example:

```
for A in [1 2] do ... end
```

The nice thing about lists is their constant shape that allows us to go through them using a simple recursive procedure like this:

```
proc {For L}
    if L \= nil then A in
        A = L.1
        ... % body that can use A
        {For L.2} % recursive call
    end
end
```

This procedure does exactly what is needed by a for loop. Indeed, inside the body, one has access to a variable `A` which takes the value of the current element of the iteration.

The compiler of Mozart2 transforms such a for loop into the AST represented in figure 3.1.

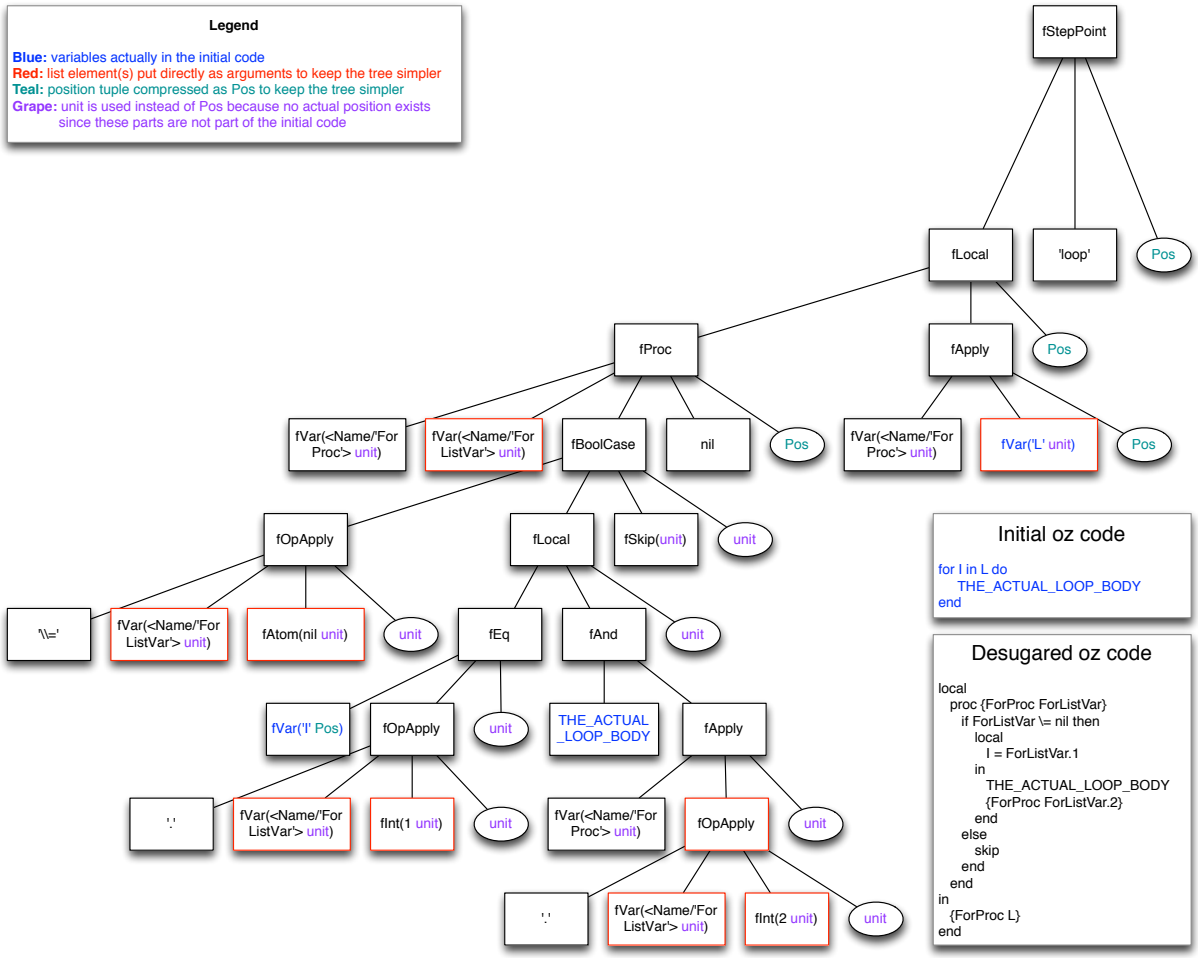

Figure 3.1: *Representation of the transformed AST for a loop with one list as generator.*

The node is represented by a rectangle except the position nodes which are inside circles. Some variable names are created by the compiler itself. In that case, its name is a bit different. When the compiler gives the name `MyVar` to a variable then the AST contains a variable named `<Name/'MyVar'>`. This is to avoid any collision between the user variables and new ones.

When a child of a node is a list, we sometimes put all its elements successively as direct children for simplicity. These elements are then in a red rectangle.

We can see that the actual body of the step point is the declaration of the recursive procedure followed by a call to this procedure.

From this we can get the general layout of such a kind of transformation. Here is some terminology we will use:

Initiator: The argument for the call to the recursive procedure, the complete list in our example.

Condition: The condition - on the argument of the recursive procedure - to fulfill in order to keep on iterating, whether the list in argument is empty in our example.

Declaration: The declaration to make in order to get a variable called as the ranger containing the element of the current iteration, the declaration of `A` in our example.

Next call: The expression to use as argument for the recursive call, the expression `L.2` in our example.

The other generators that a for loop can have are C-style generators, integer generators and from generators. The main principle is exactly the same: one recursive procedure. There are four modifications which are exactly correspond to the four definitions above.

Returning a list

From the general layout seen above, we first need to return something. Indeed, for loops do not return anything - except when some features are used like `collect` - but list comprehensions return a list. So we must adapt the recursive procedure used by for loops. We need to add the result which is a list creating in parallel of the traversing. To keep the same order as the input, we will use a procedure with a new argument, the next list to assign - recall that a list is in fact composed of many nested lists.

Terminal recursion is a property that we want to keep for efficiency reason and for laziness ans streams that we will see later. For now, we just ensure that this property is fulfilled to make our procedure efficient because at each recursive call, the calling procedure can be forgotten since it does not hold any information that are going to be used. This is better for both time and memory.

Assume we want to output the square of the input list. At each iteration we assign the current result to the square of the current element appended with the next list to assign. When there are no more iterations to do, we must set the current list to the empty list otherwise the result of our procedure will not be completely bound. In other words, we would return a stream. The resulting recursive terminal procedure is the following:

```
proc {For L ?Result}
    if L \= nil then A in
        A = L.1
```

```
        ... % body that can use A
        local Next in % the next list to assign
            % the current list is A squared appended with the next list
            Result = A*A|Next
            {For L.2 Next} % recursive call with Next
        end
    else
        Result = nil % no more iteration, end list
    end
end
```

Calling this procedure binds the last argument to the resulting list. This procedure will be used and transformed to add functionalities.

C-style generator

We begin with the C-style generator because it is the most explicit. Starting from the recursive procedure returning a list, we just need to adapt four things. Let us see illustrate these modifications with an example. Consider a C-style generator like this:

```
A*A in 0 ; A < 10 ; A+1
```

The initiator is the first part after the `in`, so 0. The condition is the second part while the next call is the last part. As for the declaration, there is none as we directly get A as argument.

Here is a complete example. The procedure becomes:

```
proc {For A ?Result} % A is declared as argument
    if A < 10 then % condition of C-style
        local
            Next
        in
            Result = A*A|Next
            {For A+1 Next} % recursive call
        end
    else
        Result = nil
    end
end
```

The call to this procedure is the following:

```
Result = {For 0} % argument is initiator of C-style generator
```

Here is a representation in figure 3.2.

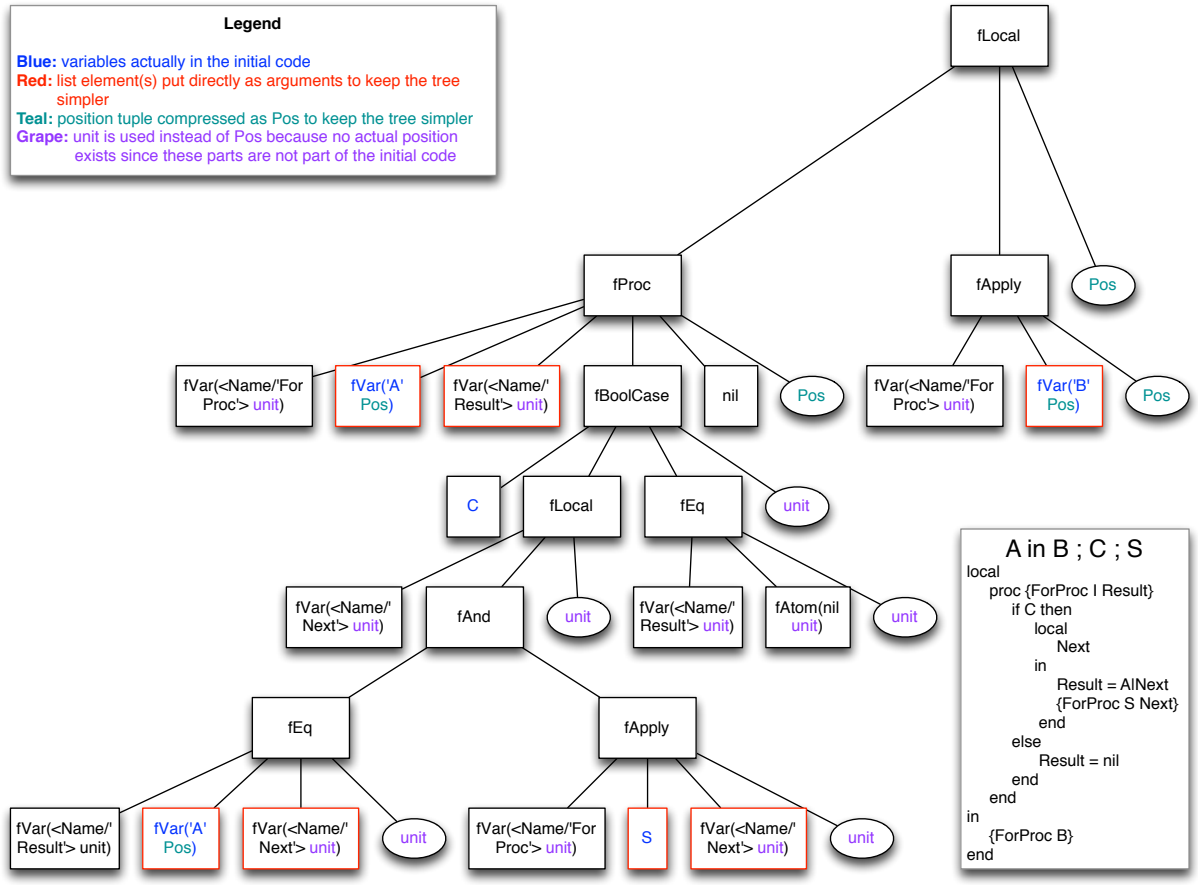

Figure 3.2: *Representation of the AST for C-style generator and its call.*

The file *Transformations/List_comprehensions/Tr_Ex_C.oz* contains a coded example.

Integer generator

Integer generators can thought of as specializations of C-style generators. This is true because one can always express an integer generator as a C-style generator. Consider the following general integer generator:

```
A in Low..High ; Step
```

It can be transformed into:

```
A in Low ; A =< High ; A+Step
```

If the step in not specified as it is optional then we just replace it by 1.

Thanks to this transformation we can indeed handle integer generator as special cases of C-style generators.

The file *Transformations/List_comprehensions/Tr_Ex_Int.oz* contains a coded example.

List generator

List generators have already been approached in the paragraph about for loops. However, we formalize them in this paragraph.

The main difference between C-style generators - and integer generators since they are a specialization of the latter - and list generators is that the ranger is not passed as argument. This is because list generators a whole list to be passed as argument. This is not a problem at all but this just implies declaring the ranger inside the recursive procedure.

Formally a list generator of the generic shape:

```
A in L % L may be a list directly declared e.g. [4 2]
```

leads to an initiator being `L`, the list itself. The condition is the fact that the list as argument is empty - `nil` - or not. The declaration is the assignation of `A` to `L.1`. Finally the next call is the tail of the list, `L.2`.

The file *Transformations/List_comprehensions/Tr_Ex_List.oz* contains an coded example.

From generator

The last kind of generator are from generators. The canonical shape is the following:

```
A in fun{$} 1 end % the name of a function could be specified instead
```

As the elements directly come from a function call, they do not have any condition - recall that they never stop by themselves. As the function does not have any argument, the initiator and the next call are both just a call to that function. As we directly use the result of the function call as argument, we do not need any declaration, the argument is the ranger.

The file *Transformations/List_comprehensions/Tr_Ex_From.oz* contains a coded example.

Record generator

We previously stated that there four generators. This is the case when we consider for loops.

We have decided to add one generator. It consists in a record. As a record can itself contains records inside its fields, we have decided to go only through the leaves of the tree formed by this nesting. The leaves are traversed in the popular depth-first mode. To provide more flexibility we also provide a way to specify whether a non-leaf node has to be considered as a leaf. Let us see how to transform this step by step.

Consider the following generic layout of a record generator where the function is optional - recall that the ranger must have a feature to distinguish record generators from list generators:

```
F:A in r(a:1 b:r(aa:2 bb:3 cc:r(aaa:4) dd:5) c:6) of fun{$ F V} ... end
% a variable containing a record could have been used for the record
% a variable containing a function could have been used for the of part
```

A question arises: how to traverse systematically a tree in depth-first mode ? The solution is to use a stack. A stack is a data structure where one can push -add - and pop - remove - elements. The policy for removing elements is last-in first-out - LIFO. It means that the next element removed - the first-out - is the last pushed element - the last-in.

Such a structure can be implemented using a simple list in Mozart2. Indeed, one can easily add and remove elements at the beginning of the list.

A stack makes the tree traversal very easy. We begin by pushing the root node of the tree in the stack. We can then iterate while the stack is not empty. The idea is to pop a node from the stack and visit it. Visiting means doing whatever we want with this node - we will see that later - and push all its children - while keeping their order - of on the stack. This procedure ensures visiting all nodes in a depth-first mode.

Since we have to keep track of both the values and the features of the fields, we will use two stacks: one for the features and one for the values. These two stacks will then always have the same size and more precisely, element i of the features stack is the feature of the value at element i of the values stack.

Our goal is to be able to declare the ranger - `F:A` in our example - with these two stacks and the root record itself. For this we state that at all time, the top of the stacks contains the next node to visit. For now, let us assume that we have a function that returns the next node and updates the stacks.

With this function, we have enough to specify the initiator, the condition, the declaration and the next call for the recursive procedure iterating over a record.

The initiator is a tuple labeled `stacks` with two elements. The first is the arity of the record, the second is all the values in the same order. So in our example, the initiator is:

```
stacks({Arity Record1At1} {Record.toList Record1At1})
% where, Record1At1 was declared beforehand
Record1At1 = r(a:1 b:r(aa:2 bb:3 cc:r(aaa:4) dd:5) c:6)
```

Our invariant is verified, the two first elements of the two stacks correspond to `a:1`.

The condition is similar to the one of list generators, we keep iterating if one of the two stacks is non-empty, it does not matter which one because they always have the same size. So the condition of our example is:

```
% Stacks1At1 is the argument of the recursive procedure
Stacks1At1.1 \= nil
```

The declaration consists in calling the function returning the next feature, value and stacks. In our example, the declaration is:

```
% FindNext is the name of the function not implemented yet
% Stacks1At1 is the argument of the recursive procedure
F#A#NewStacks1At1 = {FindNext Stacks1At1}
```

With this declaration, our ranger is now declared and can be used. The third declared variable contains the updated stacks.

The next call uses the freshly declared variable `NewStacks1At1`. Indeed, it is enough to call the next iteration.

Now we can see how to implement the `FindNext` function. Basically, it must check whether the top of the values stack is a record or not. If it is a record, then it must push the children of this record onto the two stacks. It is not a record then the function can return the feature, the value and the updated stacks. Note that an atom is in fact a record with no fields. So we do not consider atoms - or empty records - as records because it would no sense to visit their - non-existing - children. We check this by ensuring that the arity of the record is not empty. Because functions do not really exist, we use its equivalent procedure. This procedure is the following:

```
1 proc {FindNext stacks(FeatStack ValueStack) ?Result}
2     local
3         Feat = FeatStack.1 % top feature
4         Val = ValueStack.1 % top value
5         PoppedFeatStack = FeatStack.2   % one feature has been popped
6         PoppedValueStack = ValueStack.2 % one value has been popped
7     in
8         if {IsRecord Val} andthen {Arity Val} \= nil then
```

```
9               {FindNext stacks({Append {Arity Val} PoppedFeatStack}
10                               {Append {Record.toList Val} PoppedValueStack})
11                      Result}
12          else
13              Result = Feat#Val#stacks(PoppedFeatStack PoppedValueStack)
14          end
15      end
16  end
```

We directly see that our invariant is always verified because the modification on both stacks are symmetric.

The last thing to handle for record generators is their optional function condition. We just need to call it and use its result at the right place. This place is the condition of line 8 of the function `FindNext`. To make this procedure general, we add an argument to this procedure. For the implementation, we will actually create two procedures `FindNext`, one with an extra argument for the function to call, one without. So for our example, the procedure becomes:

```
proc {FindNext stacks(FeatStack ValueStack) Fun ?Result}
    local
        Feat = FeatStack.1 % top feature
        Val = ValueStack.1 % top value
        PoppedFeatStack = FeatStack.2   % one feature has been popped
        PoppedValueStack = ValueStack.2 % one value has been popped
    in
        if {IsRecord Val} andthen {Arity Val} \= nil
           andthen {Fun Feat Val} then % call user function
            {FindNext stacks({Append {Arity Val} PoppedFeatStack}
                             {Append {Record.toList Val} PoppedValueStack})
                      Result}
        else
            Result = Feat#Val#stacks(PoppedFeatStack PoppedValueStack)
        end
    end
end
```

The file *Transformations/List_comprehensions/Tr_Ex_Record.oz* contains a coded example.

Level condition

Even if we did not see how to transform multi layer nor multi level list comprehensions, we can already see how to handle a level condition. This implementation of such conditions are the same with one or several layers and/or levels.

What a condition says is that we should not add elements when it evaluates to false. But we must still make the recursive call. So we just have to encapsulate the addition of an element to the list inside a condition. Here is the modified recursive procedure:

```
proc {For {{ Arguments }} ?Result}
    if {{ Range conditions }} then
        if {{ Level condition }} then % as before
            local Next in
                Result = A*A|Next
                {For {{ Next calls }} Next}
            end
        else % condition not fulfilled, call next iteration directly
            {For {{ Next calls }} Result}
        end
    else
        Result = nil
    end
end
```

Note that we also have to encapsulate the call to the next iteration because the last argument differs. This is because when we add an element, we must use the new next list to assign. When no element is added, then the old next list to assign is also the new one.

Multi layer

All the transformations of the generators used in our new syntax have been explained so let us now focus on how to handle several layers in one list comprehension.

Having N layers means going through N generators simultaneously. So we have to use the same recursive procedure because otherwise we would not traverse them in parallel but sequentially. Additionally, using the same procedure is more efficient because we only use one iteration *infrastructure* to traverse N generators.

What we have to do in order to be able to handle any number of layers is to use N arguments instead of 1 of the recursive procedure. This implies also using N initiators, N conditions N declarations, and N next calls instead of 1. A generic example follows.

The one layer version of the recursive procedure follows. The notation `{{ ... }}` is used as an abstraction for range dependent expressions *in extenso* the initiator, the condition, the declaration and the next call.

```
% call
{For {{ Initiator Generator1 }} Result}
% procedure
proc {For Arg1 ?Result}
    if {{ Condition Arg1 }} then
        local
            {{ Declaration Arg1 }}
        in
            local Next in
                Result = {{ Ranger }}|Next
                {For {{ NextCall Arg1 }} Next}
            end
        end
    else
        Result = nil
    end
end
```

Adapting this generic procedure for N layers leads to:

```
% call
{For {{ Initiator Generator1 }} ... {{ Initiator GeneratorN }} Result}
% procedure
proc {For Arg1 ... ArgN ?Result}
    if {{ Condition Arg1 }} andthen ... andthen {{ Condition ArgN }} then
        local
            {{ Declaration Arg1 }}
            ...
            {{ Declaration ArgN }}
        in
            local Next in
                Result = {{ Ranger }}|Next
                {For  {{ NextCall Arg1 }} ... {{ NextCall ArgN }} Next}
            end
        end
    else
        Result = nil
    end
end
```

That is it. We do not to change anything more. Note two things. First note that, as expected we stop iterating as soon as at least one of the N conditions becomes false. Finally note that some generators do not have any condition and/or declaration but this does not change anything, we

just omit them if they do not exist.

The file *Transformations/List_comprehensions/Tr_Ex_From.oz* contains a coded example of a multi layer list comprehension along with other functionalities.

Multi level

Multi layer was quite straightforward to transform. On the other hand, multi level brings more complexity. Unlike layers, each level requires its own procedure. We still can put all the layers of a level in one procedure but we now need to handle as many procedures as there are levels.

The order of layers do not matter as long as they are in the same level. This is not true for levels. The first level must call the second and not the other way around. To be more specific, a nested level is called by its parent level. When the child level is done iterating, it must then call back its parent. This structure can be seen as a chain in both directions. We begin by the first element - the first level - and we iteratively go to the last element. At the end of the iteration, we go back from the last to the first element. This shows us two important facts. A level must be aware of its parent and child - they are both unique. Secondly, we start and end at the first level. The latter implies that the first level is the one in which we have to end the output list. It also implies that only the last level appends elements to the output.

All in all, the first level must be the only one the finish the output. Instead of finishing the output, each level - except the first - must call back its parent - they all have one since we do not do this for the first level.

The last level must be the only one the add elements to the output. So instead of adding elements to the output, each level - except the last - must call its child - they all have one since we do no do this for the last level. The last level must add elements then it must call itself back otherwise it would break the recursion.

A level that is not the last one has to call its child. This call must also contain the initiators of the child level. On the other hand, the child must call its parent with all the next calls of its parent. Here are these two generic calls.

```
% parent calls child
{LevelChild {{ Child initiators }} {{ Rangers of this level }}}
% child calls back parent
{LevelParent {{ Next calls for parent level}}}
```

To illustrate this, consider the following list comprehension:

```
% list comprehension with 2 simple levels
```

```
L = [A#B for A in 1..2 for B in 3..4]
```

The calls become - parent is level 1 and child is level 2:

```
% parent calls child
{Level2 3 A}
% child calls back parent
{Level1 A+1}
```

Note that we must pass all rangers to levels that follows them because they can be use anywhere after their declaration. Here `A` must be passed to level 2 because it might use it. Furthermore, it is required to call back level 1.

Another issue arises when the generator is a list. Indeed, the ranger of a list generator does not have the information necessary to make the next call. This implies that we must also pass the current list as argument in order for the next level to be able to call back. Here is a complete example:

```
% list comprehension with 2 simple levels generated with a list
L = [A#B for A in LA for B in LB]
% level 1
proc {Level1 Arg1 ?Result}
    if Arg1 \= nil then
        local A in
            A = Arg1.1
            {Level2 LB A Arg1} % call child
        end
    else
        Result = nil % finish output
    end
end
% level 2
proc {Level2 Arg1 A ExtraArg1 ?Result}
    if Arg1 \= nil then
        local B in
            B = Arg1.1
            local Next in
                Result = A#B|Next % add element
                {Level2 LB.2 A ExtraArg1} % call itself back
            end
        end
    else
        {Level1 ExtraArg1.2 Result} % call parent
```

```
    end
end
```

Calling the list comprehension is done by this instruction:

```
Result = {Level1 LA}
```

As this call depends on the initiators of the first level, we have decided to add a *fake level* to make the call to the list comprehension constant. We called this level the pre-level. Here is an example:

```
proc {PreLevel ?Result}
    {Level1 LA Result}
end
```

Now a simple call to the pre-level works for any list comprehension. Note that this pre-level will get more complex as we add functionalities.

The file *Transformations/List_comprehensions/Tr_Ex_From.oz* contains a coded example of a multi level list comprehension along with other functionalities.

Multi output

Up to now, we only saw how to output one list. Let us see how to get rid of this limitation. Our decision is that when there is more than one output, list comprehensions output a record of lists. The features are given inside the specifications of the list comprehension or we have to keep track of the features ourselves. In the latter case, features are integers from 1. Of course, we can mix specified and unspecified features.

When one specifies the following list comprehension:

```
[A 1:A+1 for A ...]
```

the result will be a tuple with two fields. The second field being generated with `A`. This might seem weird but it mandatory because we need to have access to every field of the output record so we have to know that the user already 1 as feature when we parse the first expression.

The only way of knowing this is to go trough all the features once before going through every output specification. We have to pre-analyze. This pre-analysis traverses all the specified features that are integers and creates an ordered list with these. As we have to traverse all the features, we create this incrementally, inserting every new element at its right position to keep the list sorted in ascending order. If the new element is already in the list then it means that the user specified two identical features, we them raise an error.

Once we have that sorted list of features, we go through all fields and assign the smallest integer - starting from 1 - not in our sorted list as feature for fields that do not have one. This way, we know that there can not be any collision.

With one output, we just pass the last argument, `Result`, as the next list to be assigned. With several outputs, we use a similar idea. Every output list is put inside the resulting record and instead of passing the next list to assign, we pass a record with the same arity and with all the next lists to assign.

For this step, we then use the pre-level to assign the result to a record. Then every time we add elements we must update the record of results. We always use `'#'` as label for the output in order to allow the syntactic sugar of tuples to work. Here is an example:

```
% list comprehension
[A 1:A+1 A-1 for A in 1..2]
% pre-level
proc {PreLevel ?Result}
   local Next1 Next2 Next3 in
      Result = '#'(1:Next2 2:Next1 3:Next3)
      {Level1 1 '#'(1:Next2 2:Next1 3:Next3)}
   end
end
% level 1
proc {Level1 A ?Result}
   if A =< 2 then
      local Next1 Next2 Next3 in
         Result.1 = A+1|Next2
         Result.2 = A|Next1
         Result.3 = A-1|Next3
         {Level1 A+1 '#'(1:Next2 2:Next1 3:Next3)}
      end
   else
      Result.1 = nil
      Result.2 = nil
      Result.3 = nil
   end
end
```

Only the pre-level, the first level and the last level are concerned by this change. Indeed, the pre-level must assign the result to the corresponding record. The first level must finish all the results at the end. The last level must add elements to all results.

The file *Transformations/List_comprehensions/Tr_Ex_Int.oz* contains a coded example of a

multi output list comprehension along with other functionalities.

Output condition

Output conditions could have been explained together with level conditions but they do not really make sense without the multi output functionality. More specifically, they do not change the result but because of their different implementation, the execution is a bit different.

With level conditions, we filtered iterations that do not fulfill the level conditions. Here we do not want to filter iterations because conditions might differ from on output to another. So we must act just before adding the element to the list.

We can do this easily when we assign the list to assign to the specified expression appended with the next list to assign. We just need to assign to the next list to assign if the condition is not verified. Here is an example:

```
% condition passed
Result.X = {{ ExprX }}|NextX
% condition not passed
Result.X = NextX
% express both in one expression
Result.X = if {{ OutputConditionX }} then {{ ExprX }}|NextX else NextX end
```

The main advantage of this technique is that nothing more has to change.

The file *Transformations/List_comprehensions/Tr_Ex_Int.oz* contains a coded example of a list comprehension with an output condition along with other functionalities.

Laziness

Body

Collector

Record comprehension

3.2. Implementation

For this part, we need to implement the previously mentioned files *ListComprehension.oz* ans *RecordComprehension.oz* which respectively transform list and record comprehension sugars into procedures and a procedure call. This section describes how the transformations explained in the previous section are implemented together.

Architecture

Applying the transformations

Chapter 4

The tests

This chapter goes through all tests that our solution had to pass. All these tests can be found in the directory called *Tests*. A shell script named *run.sh* allows testing them in a systematic way.

As we use categories, we have decided to create a functor providing some general functions to test comprehensions. The main idea is to make the test file as simple as possible. We have decided to create a function taking a list as only argument to automatize the testing. Every element of this list must be a tuple with its first element being the comprehension to test and the second being the expected result. The function goes through this list and displays messages according to the result of the testing. This functor is in the file *Tests/Tester.oz*. This file also contains other more specific functions explained in the corresponding paragraphs.

4.1. General tests

This section explains all the categories of tests that were created in order to test the correctness of our implementation. Our final one passes all these tests.

All categories are about lists comprehensions except the last one which is about record comprehensions.

Most tests about concurrency will be detailed in the next chapter.

One level – one layer

This first category of tests has one goal: test whether the simplest uses of list comprehensions work. For these basic tests, only list comprehensions with one layer, one level and one non-featured output are tested. We test all the possible ranges except the ones with `from` or generated with records. The latter is tested in a another specific file explained later. Here are the tests and their expected output:

```
% [listComprehension]#[expectedList]
[A for A in 0..10                                ]#[0 1 2 3 4 5 6 7 8 9 10]
[A for A in 0..10 if A mod 2 == 0                ]#[0 2 4 6 8 10]
[A for A in 0..10 ; {Get 2}                      ]#[0 2 4 6 8 10]
```

```
[A for A in 0..10 ; 2 if A > 3                          ]#[4 6 8 10]
[A for A in 0 ; A<11 ; A+1                              ]#[0 1 2 3 4 5 6 7 8 9 10]
[A for A in 0 ; A<11 ; A+1 if A mod 2 == 0              ]#[0 2 4 6 8 10]
[A for A in {Get0} ; {Cond1 A} ; {Plus2 A}              ]#[0 2 4 6 8 10]
[A for A in {Get0} ; {Cond1 A} ; A+1 if {Cond2 A}]#[0 2 4 6 8 10]
[A for A in L                                           ]#[0 1 2 3 4 5 6 7 8 9 10]
[A for A in L if A mod 2 == 0                           ]#[0 2 4 6 8 10]
[A for A in [0 2 4 6 8 10]                              ]#[0 2 4 6 8 10]
[A for A in [0 1 2 3 4 5 6 7 8 9 10] if {Cond2 A}]#[0 2 4 6 8 10]
```

One level – multi layer

Allowing several layers gives more possibilities so it is the object of these tests. Here they are:

```
% [listComprehension]#[expectedList]
[[A B] for A in 0..9   B in 10..19]
#[[0 10] [1 11] [2 12] [3 13] [4 14] [5 15] [6 16] [7 17] [8 18] [9 19]]

[[A B] for A in 0..9   B in 10..19   if A > 2 andthen B < 15]
#[[3 13] [4 14]]

[[A B C] for A in 0..9   B in 10..19   C in 20..29]
#[[0 10 20] [1 11 21] [2 12 22] [3 13 23] [4 14 24] [5 15 25] [6 16 26] [7
   17 27] [8 18 28] [9 19 29]]

[[A B C] for A in 0..9   B in 10..19   C in 20..29   if A+B+C > 33]
#[[2 12 22] [3 13 23] [4 14 24] [5 15 25] [6 16 26] [7 17 27] [8 18 28] [9
   19 29]]

[[A B C] for A in L   B in {Get 10}..19 ; 4   C in 20 ; C<3*B ; C+A]
#[[0 10 20] [1 14 20] [2 18 21]]

[[A B C] for A in 1 ; A<3 ; A+1   B from Fun   C in [4 5 6]]
#[[1 1 4] [2 1 5]]

[[A B C] for A in [1 2]   B in [3 4 5]   C in [6 7 8 9]]
#[[1 3 6] [2 4 7]]

[[A B C D] for A in [1 2]   B in [3 4 5]   C in [6 7 8 9]   D in L]
#[[1 3 6 0] [2 4 7 1]]

[1 for _ in [1 2]   _ in [3 4 5]   _ in [6 7 8 9]   _ in L]
```

```
#[1 1]

[A+B for A in 1 ; A<10 ; 1 B in 1..4 ; 2]
#[2 4]

[A for A in 1 ; A+1 _ in [1 1 1 1]]
#[1 2 3 4]

[A for A in 1 ; A+1 _ in [1 1 1 1] if A < 4]
#[1 2 3]
```

Multi level – one layer

Rolling back to the one layer tests and allowing several layers gives the following tests:

```
% [listComprehension]#[expectedList]
[[A B] for A in 1..2 for B in 3..4]
#[[1 3] [1 4] [2 3] [2 4]]

[A#B#C for A in [0 2] for B in 4 ; B<10 ; B+2 for C in 8..10 ; 2 if B<7]
#[0#4#8 0#4#10 0#6#8 0#6#10 2#4#8 2#4#10 2#6#8 2#6#10]

[A#B#C for A in [0 2] if A<10 for B in [4 6 7] for C in [8 10] if B<7]
#[0#4#8 0#4#10 0#6#8 0#6#10 2#4#8 2#4#10 2#6#8 2#6#10]

[A#B#C for A in 0..2 ; 2 if A<10 for B in 4..7 ; 2 for C in 8..10 ; 2 if B
    <7]
#[0#4#8 0#4#10 0#6#8 0#6#10 2#4#8 2#4#10 2#6#8 2#6#10]

[A#B#C for A in 0 ; A=<2 ; A+2 if A<10 for B in 4 ; B<7 ; B+2 for C in 8 ;
    C=<10 ; C+2 if B<7]
#[0#4#8 0#4#10 0#6#8 0#6#10 2#4#8 2#4#10 2#6#8 2#6#10]

[1 for _ in 1..2 for _ in 1..2 for _ in 1..2]
#[1 1 1 1 1 1 1 1]

[A+B#C for A in 1..10 if A < 3 for B in [3 4] for C in A ; C<10 ; C+1 if C
    < 4]
#[3 2 1 4 3 2 3 2 4 3]
```

Multi level – multi layer

Now we allow several levels and several layers:

```
% [listComprehension]#[expectedList]
[A#B#C#D#E for A in 1..4 B in 11..13 if A+B<16 for C in 1 ; C<10 ; C+2 D in
    [1 2] E in 30..100 if A+B+C+D+E<100]
#[1#11#1#1#30 1#11#3#2#31 2#12#1#1#30 2#12#3#2#31]

[A#B#C#D#E#F for A in 1..4 B in 11..13 if A+B<16 for C in 1 ; C<10 ; C+2 D
   in [1 2] E in 30..100 if A+B+C+D+E<100 for F in 1..1]
#[1#11#1#1#30#1 1#11#3#2#31#1 2#12#1#1#30#1 2#12#3#2#31#1]

[A#B#C#D#E#F for A in 1..4 B in 11..13 if A+B<16
    for C in 1 ; C<10 ; C+2 D in [1 2] E in 30..100 if A+B+C+D+E<100
        for F from Fun _ in 1..1]
#[1#11#1#1#30#1 1#11#3#2#31#1 2#12#1#1#30#1 2#12#3#2#31#1]

[[A AA B] for A in 1..100 AA in [1 0 3] if A == AA for B in [f o l o] if B
   \= l]
#[[1 1 f] [1 1 o] [1 1 o] [3 3 f] [3 3 o] [3 3 o]]
```

From

We only have tested ranges that use the `in` keyword. The reason for this is that ranges generated with `from` never stop. Indeed, due to their simplicity they do not provide a stopping condition. So the only way to stop them is to use another layer which will stop - recall that when several layers are used, the level stops iterating when at least one layer is done iterating. Here are the tests:

```
% [listComprehension]#[expectedList]
[I*A for A in [1 2 3] I from Fun1]#[2 4 6] % Cell = 0
[I*A for A in [1 2 3] I from Fun2]#[1 4 9] % Cell = 4

[I*A#J*B for A in [1 2 3] I from Fun2 for B in 1..2 J from Fun2]
#[5#6 5#14 18#10 18#22 39#14 39#30]

[A for _ in 1..1 A from Fun1]#[2]

[Z+Y for Z from Fun0 _ in 1..2 for _ in [1 2] Y from Fun1]
#[2 2 2 2]

[Z*Y for Z from Fun0 A in 1..2 if A < 2 for _ in [1 2] Y from Fun1]
#[0 0]

[Z+Y for Z from Fun0 A in 1..2 if A < 2 for _ in [1 2] Y from Fun1 if Y ==
```

```
    1]
#nil

[Z+Y for Z from Fun0 A in 1..2 if A < 2 for _ in [1 2] Y from Fun1 if Y ==
    {Fun1}]
#[2 2]

[A for A from fun{$} 1 end _ in 1..2]
#[1 1]
```

Records as generators

Now comes the time to test iterating over a record. We need to test with and without features. Here are the tests:

```
% [listComprehension]#[expectedList]
[A for _:A in 1#2#3                ]#[1 2 3]
[A for _:A in 1#2#3 if A > 1       ]#[2 3]
[A for _:A in Rec                  ]#[a b c d]
[A for _:A in Rec if A \= c        ]#[a b d]

[B#A for _:A in Rec _:B in 1#2#3]
#[1#a 2#b 3#c]

[B#A for _:A in Rec _:B in 1#2#3 if B > 1]
#[2#b 3#c]

[A#B for _:A in Rec if A == a for _:B in 1#2#3]
#[a#1 a#2 a#3]

[A#B#C for _:A in Rec if A == a for _:B in 1#2#3 _:C in 4#5]
#[a#1#4 a#2#5]

[A+B for A in 1..2 _:B in 3#4]
#[4 6]

[A+B for A in 1..2 for _:B in 3#4]
#[4 5 5 6]

[A#F for F:A in rec(a:1 b:2)]
#[1#a 2#b]

[F for F:_ in 6#7#8]
```

```
#[1 2 3]

[A for _:A in 1#2#(3#4#(5#6)#7)#8]
#[1 2 3 4 5 6 7 8]

[A for _:A in [1 [2] [3 [4] 5] 6 [[7 [8]]]] if A \= nil]
#[1 2 3 4 5 6 7 8]

[F#A for F:A in r(a:1 b:2 cc:r(c:3 d:4 ee:r(e:5)))]
#[a#1 b#2 c#3 d#4 e#5]

[F#A for F:A in r(a:1 b:2 cc:r(c:3 d:4 ee:r(e:5))) if A \= 1]
#[b#2 c#3 d#4 e#5]

[F#A if F\= b for F:A in r(a:1 b:2 cc:r(c:3 d:4 ee:r(e:5))) if A \= 1]
#[c#3 d#4 e#5]

[A#B for _:A in 1#2#(3#4) for _:B in 10#r(20)]
#[1#10 1#20 2#10 2#20 3#10 3#20 4#10 4#20]

[1#A for _:A in r(1 2 a(3 4)) of fun{$ _ V} {Label V}\=a end]
#[1#1 1#2 1#a(3 4)]

[A for _:_ in r(1 r(2 3)) of fun{$ F _} F == 1 end A in 1..10]
#[1 2]
```

Multi output

Now that all ranges and all size of layers and levels have been tested, we can go on and test the multi output possibility of Mozart2 list comprehensions.

```
% [listComprehension]#[expectedList]
[A A+3 for A in 1..3]
#([1 2 3]#[4 5 6])

[A B for A in 1..3 B in 4..6]
#([1 2 3]#[4 5 6])

[A B for A in 1..3 B in 4..6 if A+B<9]
#([1 2]#[4 5])

[A B C for A in 1..2 for B in 3..4 for C in 5..6]
#([1 1 1 1 2 2 2 2]#[3 3 4 4 3 3 4 4]#[5 6 5 6 5 6 5 6])
```

```
[A B C for A in 1..2 B in 3..4 C in 5..6]
#([1 2]#[3 4]#[5 6])

[A B C D for A in 1..2 B in 3..4 C in 5..6 D in 7..8]
#([1 2]#[3 4]#[5 6]#[7 8])

[A B C for A in 1..2 for B in 3..4 C in 5..6]
#([1 1 2 2]#[3 4 3 4]#[5 6 5 6])
```

Labeled output

The previous paragraph ensures that the multi output functionality works so we now can test giving a feature to some output. Note the fact that features in one list comprehension are always all different.

```
% [listComprehension]#[expectedList]
[A for A in 0..10]
#[0 1 2 3 4 5 6 7 8 9 10]

[a:A for A in 0..10]
#'#'(a:[0 1 2 3 4 5 6 7 8 9 10])

[whatever:A for A in 0..10]
#'#'(whatever:[0 1 2 3 4 5 6 7 8 9 10])

[a:A 2*A for A in 0 ; A<11 ; A+1]
#'#'(1:[0 2 4 6 8 10 12 14 16 18 20] a:[0 1 2 3 4 5 6 7 8 9 10])

[1:A 2*A for A in 0 ; A<11 ; A+1]
#'#'(1:[0 1 2 3 4 5 6 7 8 9 10] 2:[0 2 4 6 8 10 12 14 16 18 20])

[A 2*A for A in 0 ; A<11 ; A+1]
#([0 1 2 3 4 5 6 7 8 9 10]#[0 2 4 6 8 10 12 14 16 18 20])

[a:A 2*A b:A for A in 0 ; A<11 ; A+1]
#'#'(1:[0 2 4 6 8 10 12 14 16 18 20] a:[0 1 2 3 4 5 6 7 8 9 10] b:[0 1 2 3
    4 5 6 7 8 9 10])

[3:A+1 1:2*A A for A in [A for A in 0..10]]
#'#'(1:[0 2 4 6 8 10 12 14 16 18 20] 2:[0 1 2 3 4 5 6 7 8 9 10] 3:[1 2 3 4
    5 6 7 8 9 10 11])
```

Output conditions

Output specifications have been tested but we still need to test the output condition that can go with every individual output.

```
% [listComprehension]#[expectedList]
[X if X<3 for X in [1 2 3 4]]
#[1 2]

[a:X if X<3 Y if Y>4 for X in [1 2 3 4] Y in [5 6 7 8]]
#('#'(a:[1 2] 1:[5 6 7 8]))

[X if X>3 Y if Y>7 for X in [1 2] for Y in [5 6 7 8]]
#('#'(1:nil 2:[8 8]))

[smallerEqual:A if A=<4 bigger:A if A>4 for A in [2 5 4 3 6 1 7]]
#'#'(smallerEqual:[2 4 3 1] bigger:[5 6 7])

[A if B>2 for A in [so hello world] B in [2 3 4]]
#[hello world]
```

Laziness

Laziness will be tested in details in the next chapter. For now, we just test this functionality a bit. For this the general test function does not work because it does not correctly check whether the output is really lazy. So a new test function specific to laziness must be added to the tester functor. This function is similar to the general one but must additionally check that the output is only created at the right time. Since the number of elements generated at each need of the next value depends on the levels "deeper" than the lazy one, a extra argument is needed. It specifies how many new values are created at every need of an extra value.

```
% [listComprehension]#[expectedList]
thread [A for lazy A in 1..3] end
#[1 2 3]#1

thread [A+B for lazy A in 1..2 for B in [1 2 3]] end
#[2 3 4 3 4 5]#3

thread [A+B for A in 1..2 for lazy B in [B for A in 1..1 for B in [A for A
    in 1..3]]] end
#[2 3 4 3 4 5]#1

thread [A+B for lazy A in 1..2 for lazy B in 1..3] end
```

```
#[2 3 4 3 4 5]#1

thread [A+B for A in 1..2 for lazy B in 1..3 if A > 1] end
#[3 4 5]#1

thread [A+B#C+D for lazy A in 1..2 B in 3..4 for C in [1 2 3 4] D in 3 ; D
    <6 ; D+1 if D<5] end
#[4#4 4#6 6#4 6#6]#2

thread [A for lazy A from Fun _ in 1..3] end
#[1 1 1]#1
```

Miscellaneous

As list comprehensions must still accept all pre-existing structures, some more tests are required such as testing pattern matching. Nested list comprehensions must also be tested as well as cells and using and appending elements to a list comprehension.

```
% [listComprehension]#[expectedList]
[A+B for A#B in [1#2 3#4 5#6]]
#[3 7 11]

[A+B#C for A#B in [1#2 3#4 5#6] for C in 0 ; B+C<5 ; C+2]
#[3#0 3#2 7#0]

[A+B#C for A#B in [1#2 3#4 5#6] for C in 0 ; B+C<5 ; {Fun2}]
#[3#0 3#2 7#0]

[A+B#C for A#B in [1#2 3#4 5#6] for C in (Cell:=0 @Cell) ; B+@Cell<5 ; (
    Cell := @Cell + {Fun1} @Cell) ]
#[3#0 3#2 7#0]

[A#B for A in 1..3 for B in {Fun3 A}]
#[1#1 1#2 1#3 2#2 2#3 2#4 3#3 3#4 3#5]

[{Fun1} for _ in [1 2 3 4 5]]
#[2 2 2 2 2]

[A for A in [1 2 3] ; A\=nil ; A.2]
#[[1 2 3] [2 3] [3]]

[1 for _ in (Cell:=0 @Cell) ; @Cell<5 ; (Cell:=@Cell+1 @Cell)]
#[1 1 1 1 1]
```

```
[{Fun1} for _ in 1..5]
#[2 2 2 2 2]

[A for A|_ in [[1 foo] [2 foo] [3 foo]]]
#[1 2 3]

[[A for A in B ; A<10 ; A+1] for B in 1..5]
#[[1 2 3 4 5 6 7 8 9] [2 3 4 5 6 7 8 9] [3 4 5 6 7 8 9] [4 5 6 7 8 9] [5 6
   7 8 9]]

[a:[A for A in B ; A<10 ; A+1] for B in 1..5]
#'#'(a:[[1 2 3 4 5 6 7 8 9] [2 3 4 5 6 7 8 9] [3 4 5 6 7 8 9] [4 5 6 7 8 9]
    [5 6 7 8 9]])

[[A for A in B ; A<10 ; A+1] [[C+A for C in 1..2] for A in B..B+3 ; 2] for
   B in 1..5]
#([[1 2 3 4 5 6 7 8 9] [2 3 4 5 6 7 8 9] [3 4 5 6 7 8 9] [4 5 6 7 8 9] [5 6
    7 8 9]]#[[[2 3] [4 5]] [[3 4] [5 6]] [[4 5] [6 7]] [[5 6] [7 8]] [[6 7]
    [8 9]]])

([A for A in 1..2]#[A for A in 3..4])
#([1 2]#[3 4])

[A B for A in 1..2 B in 3..4]
#([1 2]#[3 4])

(1|[A for A in 2..6])
#[1 2 3 4 5 6]

(1|2|[A for A in 3..6])
#[1 2 3 4 5 6]

'|'(1:1 2:[A for A in 2..6])
#[1 2 3 4 5 6]
```

Collectors and body

Collectors and body are tested simultaneously because if the use of the collectors inside the body of a list comprehension works then it means that the body works as well. We also test that the collector can be used anywhere inside the list comprehension and outside if given as argument of a function. For this set of test, we also tested laziness so we needed two lists of input. Here are the tests:

```
% [listComprehension]#[expectedList]
Tests = [ %% each element is [listComprehension]#[expectedList]
    %% add tests form here...
    [c:collect:C for A in 1..2 body {C A}]
    #'#'(c:[1 2])

    [c:collect:C for A in 1..2 if A == 1 body {C A}{C A+1}]
    #'#'(c:[1 2])

    [c:collect:C for _ in 1..1 A from fun{$}1 end body {C A}{C A+1}]
    #'#'(c:[1 2])

    [c:collect:C for _:A in r(r(r(1) r(2))) body {C A}]
    #'#'(c:[1 2])

    [c:collect:C for A in 1..2 for B in 3..4 body {C A+B}{C A*B}]
    #'#'(c:[4 3 5 4 5 6 6 8])

    [1:collect:C1 2:collect:C2 for A in 1..2 body {C1 A}{C1 A*A}{C2 A+1}]
    #([1 1 2 4]#[2 3])

    [1:collect:C1 2:A+1 for A in 1..2 body {C1 A}{C1 A*A}]
    #([1 1 2 4]#[2 3])

    [c:collect:C for A in 1..3 if local skip in {C A} 0 == 1 end]
    #'#'(c:[1 2 3])

    [c:collect:C for A in 1..3 if {Ext C false} body {C A}]
    #'#'(c:[1 1 1])

    [c:collect:C for A in 1..3 if {Ext C true} body {C A}]
    #'#'(c:[1 1 1 2 1 3])
    %% ...to here
]
L1 = thread [c:collect:C for lazy A in 1..2 body {C A}] end
L2 = thread [c:collect:C for lazy A in 1..2 body {C A}{C A+1}] end
L3a = thread [1:collect:C1 2:collect:C2 for lazy A in 1..2 body {C1 A}{C2 A
    +1}] end
L3b = thread [1:collect:C1 2:collect:C2 for lazy A in 1..2 body {C1 A}{C2 A
    +1}] end
L4 = thread [c:collect:C for lazy A in 1..2 for B in 3..4 body {C A+B}] end
L5 = thread [c:collect:C for A in 1..2 for lazy B in 3..4 body {C A+B}] end
L6a = thread [1:collect:C1 2:collect:C2 for lazy A in 1..2 for B in 3..4
    body {C1 A}{C2 B}] end
```

```
L6b = thread [1:collect:C1 2:collect:C2 for lazy A in 1..2 for B in 3..4
    body {C1 A}{C2 B}] end
L7a = thread [1:collect:C1 2:collect:C2 for lazy A in 1..2 for lazy B in
    3..4 body {C1 A}{C2 B}] end
L7b = thread [1:collect:C1 2:collect:C2 for lazy A in 1..2 for lazy B in
    3..4 body {C1 A}{C2 B}] end
TestsLazy = [ %%
    L1.c#[1 2]#1
    L2.c#[1 2 2 3]#2
    L3a.1#[1 2]#1
    L3b.2#[2 3]#1
    L4.c#[4 5 5 6]#2
    L5.c#[4 5 5 6]#1
    L6a.1#[1 1 2 2]#2
    L6b.2#[3 4 3 4]#2
    L7a.1#[1 1 2 2]#1
    L7b.2#[3 4 3 4]#1
]
```

Record comprehensions

For record comprehensions, we created one set of tests to check all its functionalities. Here are the tests:

```
% [recordComprehension]#[expectedRecord]
[A for A through Rec]
#Rec

[A for _:A through Rec]
#Rec

[F for F:A through Rec of {Arity A} \= nil]
#rec(1:1 b:b c:c d:d)

[[F A] for F:A through Rec of {Arity A} \= nil]
#rec(1:[1 a] b:[b b] c:[c c] d:[d d])

[1 for A through Rec of {Arity A} \= nil]
#rec(1:1 b:1 c:1 d:1)

[1 for _ through r(1 2 rr(3 rrr(4)) 5)]
#r(1 1 rr(1 rrr(1)) 1)
```

```
[1 for A through r(1 2 rr(3 rrr(4)) 5) of {Bool A}]
#r(1 1 rr(1 1) 1)

[1 a:2 for A through r(1 2 rr(3 rrr(4)) 5) of {Bool A}]
#'#'(1:r(1 1 rr(1 1) 1) a:r(2 2 rr(2 2) 2))

[A+1 for A through r(1 2 rr(3 rrr(4)) 5)]
#r(2 3 rr(4 rrr(5)) 6)

[A+1 A-1 for A through r(1 2 rr(3 rrr(4)) 5)]
#(r(2 3 rr(4 rrr(5)) 6)#r(0 1 rr(2 rrr(3)) 4))

[A for A through r(1 2 rr(3 rrr(4)) 5) of {Bool A}]
#r(1 2 rr(3 rrr(4)) 5)

[{Treat A} for A through r(1 2 rr(3 rrr(4)) 5) of {Bool A}]
#r(2 4 rr(6 8) 10)

[A for F:A through r(1:bb(w(2)) 2:y(1) 3:w(2) 4:n(0) 5:rr(n(0) 6:w(2) 7:y
    (1) 8:y(1)) 9:rr(rrr(y(1) n(0))))
       if {Not {IsRecord A}} orelse {Label A} \= w
       of {Label A} \= bb orelse F > 3]
#r(1:bb(w(2)) 2:y(1) 4:n(0) 5:rr(1:n(0) 7:y(1) 8:y(1)) 9:rr(rrr(y(1) n(0)))
    )

[A for A through rec(a:yes b:no c:yes rec:rec(a:no b:yes)) of {Arity A} \=
    nil if {Label A} == yes orelse {Label A} == rec]
#rec(a:yes c:yes rec:rec(b:yes))

[F#A for F:A through [1 2 3]]
#[1#1 1#2 1#3]

[F#A for F:A through [1 2 3] of 1 == 0]
#'|'(1#1 2#[2 3])

[A for A through Tree]
#Tree

[A+1 for A through Tree]
#tree(tree(leaf(2) leaf(3)) leaf(4))

[F#A for F:A through Tree]
#tree(tree(leaf(1#1) leaf(1#2)) leaf(1#3))
```

```
[F#A for F:A through Tree of F == 1]
#tree(tree(leaf(1#1) 2#leaf(2)) 2#leaf(3))

[A for F:A through Tree if F == 1]
#tree(tree(leaf(1)))
```

4.2. Other languages

We now have tested enough possibilities to test that what be done in other languages can also be done in Mozart2. Actually only the four first paragraphs of testing were mandatory to test other languages possibilities.

Erlang

Erlang proposes list comprehensions. Their syntax differs a bit from the one we chose. Here are some examples of equivalences between Erlang and Mozart2. The examples come from [1] and [12].

```
% [listComprehension]#[expectedList]
%% [X || X <- [1,2,a,3,4], X > 3]
[X for X in [1 2 &a 3 4] if X > 3]
#[&a 4]

%% [X || X <- [1,2,a,3,4], integer(X), X > 3]
[X for X in [1 2 a 3 4] if {IsInt X} andthen X > 3]
#[4]

%% [X || X <- [1,5,2,7,3,6,4], X >= 4]
[X for X in [1 5 2 7 3 6 4] if X >= 4]
#[5 7 6 4]

%% [{A,B,C} || A <- lists:seq(1,12), B <- lists:seq(1,12), C <- lists:seq
    (1,12), A+B+C =< 12, A*A+B*B == C*C]
[A#B#C for A in 1..12 for B in 1..12 for C in 1..12 if A+B+C =< 12 andthen
    A*A+B*B == C*C]
#[3#4#5 4#3#5]

%% [{A,B,C} || A <- lists:seq(1,N-2), B <- lists:seq(A+1,N-1), C <- lists:
    seq(B+1,N), A+B+C =< N, A*A+B*B == C*C]
[A#B#C for A in 1..N-2 for B in A+1..N-1 for C in B+1..N if A+B+C =< N
    andthen A*A+B*B == C*C]
#[3#4#5]
```

```
%% [Fun(X) || X <- L]
[{Fun X} for X in L]
#[2 4 6 8]

%% [X || L1 <- LL, X <- L1]
[X for L1 in LL for X in L1]
#[1 2 3 4]
```

Python

Python is good example for list comprehensions. So it is a good idea to ensure that what can be done in Python can also be done is Mozart2. Among others, nested lists comprehensions and going through the elements of a tuple are nice functionality of Python that have a Mozart2 equivalent. Here are the equivalences:

```
% [listComprehension]#[expectedList]
%% [x**2 for x in range(10)]
[{Pow X 2} for X in 1..9]
#[1 4 9 16 25 36 49 64 81]

%% [x for x in Li if x >= 0]
[X for X in Li if X >= 0]
#[0 4 8]

%% [abs(x) for x in Li]
[{Abs X} for X in Li]
#[8 4 0 4 8]

%% [(x, x**2) for x in range(6)]
[[X {Pow X 2}] for X in 0..5]
#[[0 0] [1 1] [2 4] [3 9] [4 16] [5 25]]

%% [x for x in (1,2,3)]
[X for _:X in '#'(1 2 3)]
#[1 2 3]

%% [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
[[X Y] for X in [1 2 3] for Y in [3 1 4] if X \= Y]
#[[1 3] [1 4] [2 3] [2 1] [2 4] [3 1] [3 4]]

%% [num for elem in Vec for num in elem]
[Num for Elem in Vec for Num in Elem]
```

```
#[1 2 3 4 5 6 7 8 9]

%% [[row[i] for row in matrix] for i in range(4)]
[[{Nth Row I} for Row in Matrix] for I in 1..4]
#[[1 5 9] [2 6 10] [3 7 11] [4 8 12]]
```

Haskell

Haskell is a functional language that can be considered as a reference in the domain. So it is good to have equivalence between its list comprehensions and the ones of Mozart2. Again nested lists are used as well as the `take` operator.

```
% [listComprehension]#[expectedList]
%% [2*a | a <- L]
[2*A for A in L]#[4 8 14]

%% [isEven a | a <- L]
[{IsEven A} for A in L]#[true true false]

%% [2*a | a <- L, isEven a, a>3]
[2*A for A in L if {IsEven A} andthen A>3]#[8]

%% [a+b | (a,b) <- Pairs]
[A+B for A#B in Pairs]#[5 3 15]

%% [a+b | (a,b) <- Pairs, a<b]
[A+B for A#B in Pairs if A<B]#[5 15]

%% [(i,j) | i <- [1,2], j <- [1..4]]
[[I J] for I in [1 2] for J in 1..4]#[[1 1] [1 2] [1 3] [1 4] [2 1] [2 2]
   [2 3] [2 4]]

%% [[ (i,j) | i <- [1,2]] | j <- [3,4]]
[[[I J] for I in 1..2] for J in 3..4]#[[[1 3] [2 3]] [[1 4] [2 4]]]

%% take 5 [[ (i,j) | i <- [1,2]] | j <- [1...]]
[[[I J] for I in 1..2] for J in 1 ; J+1 _ in 1..5] % '_ in 1..5' replaces '
   take 5'
#[[[1 1] [2 1]] [[1 2] [2 2]] [[1 3] [2 3]] [[1 4] [2 4]] [[1 5] [2 5]]]
```

4.3. Performance tests

Knowing that the implementation works thanks to the tests of the previous section is not enough. Tests about the time and space performance must also be executed. First we compare the memory taken by the implementation and its equivalent procedures. Secondly, we apply the same testing - often with parameters modified - for the time taken. What is expected is similar results. Indeed this would mean that the implementation is the equivalent of the procedures - which is our goal of course.

We created ten test cases and ran them for both space testing and time spacing separately. Each test was ran ten times for each technique - so the implementation and its equivalent. The reason behind running ten time each test is that we try to average out noise as much as possible. The order in which tests were run was random.

The measure we report in the graph in figure 4.1 is the ratio of the average space, respectively time, taken by the equivalent over the same measure for the implementation. So being above a hundred percents indicates a better performance for the implementation.

The ten tests are the following:

```
% 1
[A for A in LL]
% 2
[a:A b:B for A in 1..HA B in 1..HB]
% 3
[a:A b:B for A in 1..HA for B in 1..HB]
% 4
[A for A from Fun B in 1..H if B > 0]
% 5
[A if A mod 2 == 0 for A in thread [A for lazy A in 1..Lim] end:3]
% 6
[FA#A if A>10 1:B A#B for FA:A in 10#20 for _:B in Rec]
% 7
[A+B a:A if A>0 for A in 1 ; A<CA ; A+1 if A>4 for B in 2*A..CB ; 2 if A+B
    >5 for _:C in 1#2#3#4#5#6#7#8#9#10 if C == 3]
% 8
[2:A+2 1:A+1 3:A+3 for F:A through Rec if F > 0]
% 9
[1:collect:C1 2:collect:C2 for A in 1..Lim body {C1 A}{C1 A+1}{C2 y}{C2 n}]
% 10
[FA#A if A>10 1:B A#B for FA:A in 10#20 of Fct for _:B in Rec of Fct]
```

Note that the parameters change depending on whether we are testing space or time. This is to make results more readable. With these tests every functionality is tested so we can check

that every part of the implementation seems to be right with respect to its equivalent.

The tests about the memory, respectively time, are all the files *Tests/Space_performance_X.oz*, respectively *Tests/Time_performance_X.oz*, where X is from 1 to 10.

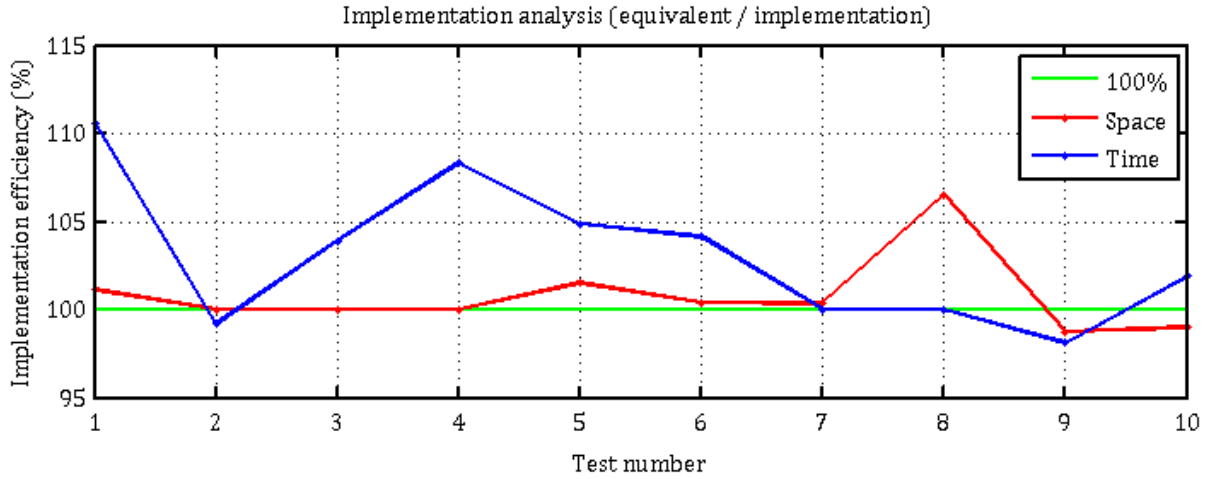

Figure 4.1: *Comparison of the performance between the implementation and its equivalent.*

Space analysis

The results in the figure 4.1 indicate that the space needs of the implementation are basically the same as the equivalent. The ratio oscillates around the green line representing a hundred percents.

The emulator of Mozart2 has a default total memory usage of 1500 megabytes. So in order to avoid any trouble in the results, it is required not to use over this threshold at the end of the ten tests. This implies not using tests taking over 75 megabytes of memory. This is the reason why we had to change the parameters. Indeed, with such parameters, the time taken is too small to be relevant.

The small observed variations are probably due to external actions taken by the emulator. They can also be due to rounding approximations but they are not significant.

In conclusion for space performance, we can say that the implementation and its equivalent are both really similar in memory usage. From this we deduced that the implementation actually corresponds to the previously seem transformations.

Time analysis

The results in the figure 4.1 indicate that the time needs of the implementation are slightly smaller than its equivalent. These variations might be due to external actions taken by the emulator or other processes running on the testing machine during the test phase. They can

also be due to rounding approximations or a slightly different transformation of the equivalent into code due to the fact that we feed this directly instead of replacing a node in the AST at some point of the compilation.

To conclude the time analysis, we can state that our implementation is the equivalent to the transformations of chapter 3.

4.4. Application examples

This last section aims at giving some concrete and small applications of comprehensions. There are many more than the followings.

Permutations

The ability to put several levels gives rise to the possibility to generate every permutation of a given size within a list, as in the following example where we want all permutations of two coin tosses:

```
declare
Coin = [head tail]
{Browse [[X Y] for X in Coin for Y in Coin]}
% [[head head] [head tail] [tail head] [tail tail]]
```

Map

The function Map takes a list as input as well as a mapping function with one argument. The result is a list whose elements are the result of applying the mapping function to each element of the input lists. Here is an example and its equivalent using a list comprehension:

```
declare
L = [1 2 3]
% mapping function
fun {Fun X} 2*X end
% normal Map
{Browse {Map L Fun}}    % [2 4 6]
% comprehension Map
fun {MapLC L Fct}
   [{Fct X} for X in L]
   % we could also do [2*X for X in L]
end
{Browse {MapLC L Fun}} % [2 4 6]
```

Actually simple list comprehensions with one level without condition nor buffer, one layer and one unfettered output without output condition is the same as a mapping operation.

Introducing record comprehensions allow the mapping to also work on records. Here is how:

```
declare
L = rec(1 a:2 3)
% mapping function
fun {Fun X} 2*X end
% normal Map
{Browse {Record.map L Fun}} % rec(2 6 a:4)
% comprehension Map
fun {MapRC R Fct}
   [{Fct X} for X through R]
   % we could also do [2*X for X through R]
end
{Browse {MapRC L Fun}}       % rec(2 6 a:4)
```

Thanks to the possibilities of comprehensions, we can also add conditions so that the result does not contain all the mapping of all the elements of the input.

Flatten

Flattening a list consists in replacing all nested lists with their elements to obtain a *flat* list *in extenso* where all nested lists do not have a list has first element. The figure 4.2 shows a graphical example.

Using a list as input of a list comprehension, we can easily flatten it like this:

```
declare
L = [[[1] 2] 3 4 [[5 [6 7]] 8] 9 [[[10]]]]
fun {FlattenLC L}
   [A for _:A in L if A \= nil]
end
{Browse {Flatten L}}   % [1 2 3 4 5 6 7 8 9 10]
{Browse {FlattenLC L}} % [1 2 3 4 5 6 7 8 9 10]
```

The condition is necessary because otherwise, the result might contain several nil depending on the input.

Sort

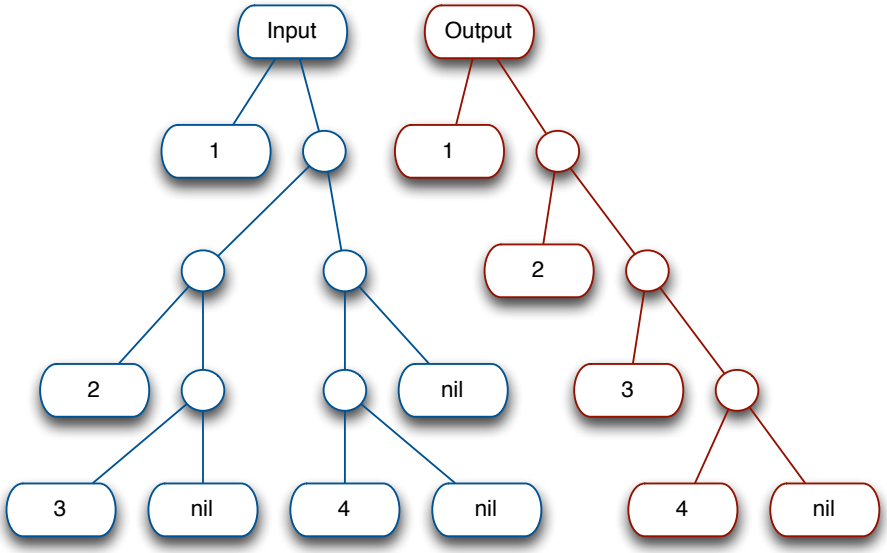

Figure 4.2: *Example of input and output for the flatten function.*

Sorting lists using the Quicksort algorithm - see [23] - can be facilitate with lists comprehensions. Thanks to their multi output and output condition functionalities, they efficiently split a list into a list with all elements smaller than a given element called the pivot, and a second list with all its elements greater or equal to the pivot. Here is the Quicksort function with lists comprehensions:

```
declare
fun {Sort L}
   case L
   of nil then nil
   [] H|T then Split in
      Split = [smaller:X if X<H greaterOrEqual:X if X>=H for X in T]
      {Append {Sort Split.smaller} H|{Sort Split.greaterOrEqual}}
   end
end
{Browse {Sort [3 9 0 1 5 1 4 3 10 ~1]}}
% [~1 0 1 1 3 3 4 5 9 10]
```

Note that we could do the same while also removing all duplicate values. This requires to change the greater or equal into a strict greater. We could also return a sorted list with all its elements fulfilling a criteria. Lists comprehensions allow many modifications to this algorithm, and others.

Factorial

Computing a factorial can also be done using list comprehensions. To compute the factorial of N, the idea is to go through all the values from N to 0 and to collect the current result in a list. At the end, the factorial is the last element of the result. Here is the complete function:

```
declare
fun  {FactLC N}
   R L
in
   L = [c:collect:C for A in N ; A >= 0 ; A-1 body
                        if A == N then {C 1} end
                        if A == 0 then R = {Nth L.c N-A+1}
                        else {C {Nth L.c N-A+1}*A}
                        end]
   R
end
```

Chapter 5
Concurrency using list comprehensions

Some tests can not easily be executed in a systematic way, especially the ones concerning concurrency, laziness and bounded buffers. Additionally we consider that tests about concurrency are better when the user sees everything that is going on. For this to happen, it requires a dynamic way to display results *in extenso* it can handle unbound variable becoming assigned. As the Mozart2 browser shows such a property, we have decided to run the tests about concurrency in Mozart2 and not in shell as in the previous chapter.

The tests of this chapter can also be used as examples of applications. That is why we decided to dedicate a whole chapter to concurrent applications of list comprehensions only. All the applications can be found in the directory *Tests/Concurrency*.

Another reason to separate this chapter from the previous one is that, in our opinion, list comprehensions are very helpful but their most important advantage is their efficiency for concurrent applications as we will see in the sections to come.

Concurrency in list comprehensions is a very powerful functionality that only a few languages support. One example is Ozma - see [19] - an extension of Scala.

5.1. Laziness

Laziness is a concept that can appear at many places in Mozart2. List comprehensions are one of this places. Laziness was already tested a bit in the previous chapter but these tests were limited. Laziness is easier to test when one can see its effects directly.

For this example, we will consider the following scenario. We want to create two streams. One with all the integers from 0 to infinity and a second from 0 to infinity but containing only even numbers. As our streams never stop, it is mandatory for them to be created lazily to avoid an generation without an end.

One list comprehension is enough to create both streams. Here is its declaration:

```
Xs1#Xs2 = [1:A 2:A if A mod 2 == 0 for lazy A in 0 ; A+1]
```

Note that we use the C-style generator without any condition on the generation so this list comprehension will never stop. Fortunately, the laziness ensures that the comprehension does not create the output without stopping. It is desirable because otherwise, we would quickly be out of memory. Thanks to the laziness the list comprehension waits for a least one of its output to be needed.

Without doing anything more than declaring the two streams `Xs1` and `Xs2`, they are both unbound. If one makes the first element of `Xs1` needed - for instance by using in a computation or by using the `Value.makeNeeded` function - then the streams become:

```
Xs1 = 0|_
Xs2 = 0|_
```

This makes sense, as the first element of `Xs1` was needed, it was created. As we create the two streams together the first element of the second stream `Xs2` also becomes assigned - note that 0 is even.

If one makes the second element of `Xs1` needed then the streams become:

```
Xs1 = 0|1|_
Xs2 = 0|_
```

It might seem weird not to get the second element of `Xs2` but it is in fact logical. Indeed, the output condition of `Xs2` filtered its second element before appending it so the second element is actually not an element at all.

Conversely, if one makes the second element of `Xs2` needed the streams would become:

```
Xs1 = 0|1|2|_
Xs2 = 0|2|_
```

The file *Concurrency/Laziness.oz* contains a complete test framework.

5.2. Producer − consumer

A classical example of using streams is the producer-consumer. Recall that a stream is an unbound list *in extenso* a list with its current last element not assigned yet. Streams are handy to allow threads to communicate because one thread can add elements to the list while another can read them and perhaps wait for new elements. A comparison of streams is Unix pipes. One writes while another reads the same data. Each thread is an agent.

The producer-consumer consists in two threads. One producing elements - the writer - and another consuming elements and acting accordingly - the reader. It is a general case of many concurrent applications. Additionally one can decide to add one or more filters. A filter is another intermediate thread that acts between the producer and the consumer. So the filter reads from the producer and writes to the consumer. There can be several filters. Filters can remove and/or add elements to their input. A graphical representation of the chain of streams is in figure 5.1.

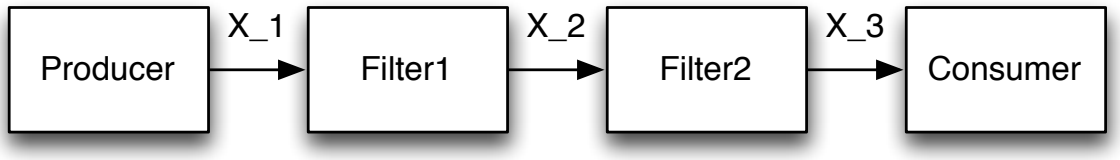

Figure 5.1: *A graphical representation of a chain of streams for a producer-consumer with 2 filters.*

Let us see a generic example. We want to get the double of only the odd elements inside the input list. The input lists is made of integers from 10 to 25. So the producer must create a list with the right integers, the filter must remove the even elements and the consumer must multiply each element by two.

The producer is typically recursive function creating the right input. This producer can be written using the following list comprehension:

```
Xs = [A for A in 10..25]
% or
Xs = [A for lazy A in 10..25]
```

The input stream can be declared as lazy in order for the it to be created only when needed by the consumer. The implementation in the file *Concurrency/Producer_Consumer.oz* uses a slightly different list comprehension to create the input because we want to add a delay in order to really see the input list while it is a stream, we want to see the execution. So to so this we add the following body to the previous list comprehension:

```
Xs = [... body {Delay 1000}] % wait for 1000 milliseconds in body
```

Now that we have created the input, the producer, we can create the filter and the consumer at the same time thanks to the functionalities of list comprehensions. Indeed we can easily take the input and filter its elements. We can then multiply each remaining element by two. Here is the list comprehension:

```
Ys = [2*A for A in Xs]
```

Note that the two list comprehensions seen above must be inside different threads in order for them to execute concurrently.

5.3. Multi output and conditions

A very useful functionality available with list comprehensions it to allow multiple outputs with only one list comprehension. Additionally, we can specify an output-specific for each output.

Let us extend the previous example. Instead of output the double of the odd elements only, we still want to output the double of the odd elements but we now also want to output the triple of the even elements in another list.

The producer is exactly the same as before. Again the filter and the consumer can be written in one list comprehension. Here is how:

```
Ys1#Ys2 = [2*A if A mod 2 == 1 3*A if A mod 2 == 0 for A in Xs]
```

Another way of doing this is to use collectors. Here is the adaptation:

```
Ys1#Ys2 = [1:collect:C1 2:collect:C2 for A in Xs
                                        body
                                        if A mod 2 == 1 then {C1 2*A}
                                        else {C2 3*A}
                                        end]
```

This application can be tested using the file *Concurrency/Multi_output.oz*.

5.4. Logic gates

A specific case of producer-consumer is logic gates. The principle is to simulate the behavior of logic electronic gates. As always in electronic, the information is transmitted using binary values *in extenso* 0 and 1. So the flux of information is succession of zeros and ones. This succession is in fact a stream. So we will model the fluxes of data using streams. Binary values 0 and 1 are often referred to as respectively false and true.

A gate is a device that typically takes two fluxes as input and outputs one stream of data where the element i is the result of applying a given logical operation on the two elements i of the two inputs. In general a logic gate can take an arbitrary number of inputs but we will focus

on gates with two inputs only knowing that we can easily extend our solution to the general case.

The producers of this problem must generate binary data. To generate a binary value, we simply take a random integer and take the remaining of its division by 2 *in extenso* we take the modulo 2 of a random integer. Thanks the multi output possibility, we can do this inside one list comprehension like this:

```
Input1#Input2 = [{OS.rand} mod 2 {OS.rand} mod 2 for lazy A in 0 ; A+1]
```

Again, this list comprehension never stops and is lazy. In the implementation in the file *Concurrency/Logic_Gates.oz*, we also add a delay to see the execution in real time and we put a limit to the output for practical reasons. As always, every agent is executed in its own thread.

Logical functions are the functions that takes two elements as input and apply the binary operation. For this example, we will use three different operations. The first one is the `AND` operation which outputs true if and only if its two inputs are true. The second operation is `OR` which outputs true if and only if at least one input is true. The last operation is the `XOR` operation. Its name stands for exclusive or. It outputs true if and only if both inputs are different. The truth tables of these three logical operators are in table 5.1. A truth table gives the output for all the different combination of inputs. As there are two binary inputs, there are $2^2 = 4$ combinations. In consequence, a truth table completely defines an operator.

	AND	OR	XOR
0 0	0	0	0
0 1	0	1	1
1 0	0	1	1
1 1	1	1	0

Table 5.1: *The truth tables of the AND, OR and XOR logical operators.*

These operators can be easily implemented as functions:

```
fun {And X Y} X*Y end
fun {Or X Y} X+Y-X*Y end
fun {Xor X Y} (X+Y) mod 2 end
```

Now that we have the operators, we can create the consumers. They are the function taking two input streams of data of outputting one data stream. Let us first make one consumer by operations, so three logical gates. For this we create a function that returns a function. The latter is our gate. Such a function maker just take the operator as argument. Here is this function implemented with a list comprehension:

```
%% returns a gate function with the given operator
fun {GateMaker F}
   fun {$ Xs Ys}
      thread [{F X Y} for X in Xs Y in Ys] end
   end
end
```

Using this gate maker, we can now create our three gates as follows:

```
%% returns a gate function with the given operator
AndG = {GateMaker And}
OrG  = {GateMaker Or}
XorG = {GateMaker Xor}
```

Every one of these three functions takes two streams as arguments and outputs a stream which is the result of applying one the three previously seen operators on every element.

Just to show a bit more about list comprehensions, let us make a big gate, that output three streams, one for each operator. The gate function is written using only the following list comprehension:

```
thread [and:{And X Y} 'or':{Or X Y} xor:{Xor X Y} for X in Xs Y in Ys] end
```

Note that the `or` atom has to be put inside quotes because it is a keyword otherwise.

The complete implementation can be tested in the file *Concurrency/Logic_Gates.oz*

5.5. Bounded buffer

When an agent goes lazily through a lazy stream, Mozart2 allows bounded buffers as explaining in the complete syntax in chapter 1. This example is a special case of producer-consumer where the producer generates lazy outputs and where the consumer wants the producer to be in advance compared to it.

The example implements the following situation. There are two input streams, both lazily generated. The first goes from 0 to 10 and takes 1000 milliseconds to generate every element. The second stream goes from 0 to 5 and needs 1500 milliseconds to generate each element.

With these two streams we want to create three ones all generated with following idea: for each element A of the first input, we go through each element B of the second input. The first output must be the made of $A - B$, the second is $A * B$ and the last is made of all $A + B$.

We do not want to create the output right away. We will wait for a dozen of seconds, but when we start, we want the first three elements to be created - almost - instantly. We do not want to wait 4500 milliseconds *in extenso* the time for the second input to create three elements.

To implement this, we first start with the producers. Unlike before, as the inputs are lazy streams created independently with different delays, we can not use the same list comprehension. We must use two different ones. Here are these two producers:

```
thread In1 = [A for lazy A in 0..10 body {Delay 1000}] end
thread In2 = [A for lazy A in 0..5  body {Delay 1500}] end
```

To implement the consumers on the other hand, we can - and we will - use only one list comprehension. The latter has three outputs and two levels. The first level has the first input as generator. Additionally, we specify a bounded buffer of size 1 for this level. Indeed, to create the three first elements of each output, we need one value for the first level, so a size of 1 is enough - we could use a bigger size.

The second level goes through the second input and is declared lazy. We could have declared the first level as lazy but the effect would be the same. The buffer we specify for this second level has a size of 3. This is the minimum required in order for the three first elements of each output to be created - almost - instantly if enough time is left between the call to this list comprehension and the need for the output.

Here is the code for the consumer:

```
Out1#Out2#Out3 = [A-B A*B A+B for A in In1:1 for lazy B in In2:3]
```

The complete implementation is in the file *Concurrency/Bounded_Buffer.oz*.

This example is quite basic but can declined in many shapes depending on the situations.

Conclusion

Possible evolutions

enhance iterating over records, BFS, ...
new features like collect
make from Function better
add possibilities to record comprehensions

Conclusion

TODO

References

Offline

[1] Armstong, J., Virding, R., Wikstrom, C., *Concurrent programming in Erlang*, Prentice Hall, 1996.

[2] Bauduin, R., Master's thesis: *A modular Oz compiler for the new 64-bit Mozart virtual machine*, UCL, 2013.

[3] Campbell, B., Iyer, S., Akbal-Delibas, B., *Introduction to Compiler Construction in a Java World*, CRC Press, 2013.

[4] Grune, D., van Reeuwijk, K., Bal, H., Jacobs, C., Langendoen, K., *Modern Compiler Design*, Springer, 2012.

[5] Lutz, M., Ascher, D., *Learning Python*, O'Reilly, 2004.

[6] Peyton Jones, S., *The Implementation of Functional Programming Languages*, Prentice Hall, 1987.

[7] Reade, C., *Elements of Functional Programming*, Addison-Wesley, 1989.

[8] Schaus, P., *Slides of LINGI2132 - Languages and translators*, UCL, 2013.

[9] Smith, J., Nair, R., *Virtual Machines: Versatile Platforms for Systems and Processes*, Elsevier, 2005.

[10] Thompson, S., *Haskell: The Craft of Functional Programming*, Addison-Wesley, 1996.

[11] Van Roy, P., & Haridi, S., *Concepts, Techniques, and Models of Computer Programming*, MIT Press, 2004.

Online

[12] *Erlang list comprehension*, consulted in 2014,
http://www.erlang.org/doc/programming_examples/list_comprehensions.html

[13] *Haskell list comprehension*, consulted in 2014,
http://www.haskell.org/haskellwiki/List_comprehension

[14] Van Roy, P., *How to say a lot with few words*, IRCAM, 2006, consulted in 2014,
http://www.info.ucl.ac.be/courses/INGI1131/2007/Scripts/ircamTalk2006.pdf

[15] *Mozart documentation*, consulted in 2014,
http://mozart.github.io/mozart-v1/doc-1.4.0

[16] *Mozart Hackers mailing list*, consulted in 2014,
https://groups.google.com/forum/#!forum/mozart-hackers

[17] Van Roy, P., *Programming Paradigms for Dummies: What Every Programmer Should Know*, consulted in 2014,
http://www.info.ucl.ac.be/~pvr/VanRoyChapter.pdf

[18] *Python list comprehension*, consulted in 2014,
http://docs.python.org/2/tutorial/datastructures.html#list-comprehensions

[19] *Scale concurrent list comprehensions*, consulted in 2014,
https://github.com/sjrd/ozma

[20] *Sources of Mozart2*, consulted in 2014,
https://github.com/mozart/mozart2

[21] *The Mozart Programming System*, consulted in 2014,
http://mozart.github.io

[22] *The programming paradigms*, consulted in 2014,
http://www.info.ucl.ac.be/~pvr/paradigmsDIAGRAMeng108.pdf

[23] *The Quicksort algorithm*, consulted in 2014,
http://algs4.cs.princeton.edu/23quicksort/

[24] *Wikipedia about list comprehension*, consulted in 2014,
http://en.wikipedia.org/wiki/List_comprehension

[25] *Wikipedia about Mozart*, consulted in 2014,
http://en.wikipedia.org/wiki/Oz_(programming_language)