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

For 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 the Mozart programming language, and also an inspiring Professor and author,

Gustavo Gutiérrez and Sébastien Doeraene for their kind help,

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 from the lack of functionalities of 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 available 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 Oz more expressive.

Among all the languages we have tried, we consider Oz as 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 Oz, this implies that this new concept must be usable with all the possibilities Mozart already offers.

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

Contents

Introduction **1**

1 Oz and comprehensions **2**

1.1 Oz, a functional language and more . . . 2

Syntactic sugars . . . 2

Expressions and statements . . . 2

Declarations . . . 3

Variables . . . 5

Functions and procedures . . . 5

Records and tuples . . . 6

Lists . . . 8

If statements . . . 8

Pattern matching . . . 9

Recursion . . . 10

Threads . . . 11

Streams . . . 13

Laziness . . . 14

Functors . . . 15

Cells . . . 15

Dictionaries . . . 16

Classes and objects . . . 16

1.2 List comprehensions . . . 16

Principle . . . 17

Terminology . . . 18

1.3 Record comprehensions . . . 19

Principle . . . 19

Terminology . . . 19

1.4 Complete syntax . . . 20

List comprehensions . . . 20

Record comprehensions . . . 23

1.5 Complete grammar . . . 25

1.6 Unofficial functionalities . . . 26

2 The compiler of Mozart 28
2.1 Generic design of a compiler using a virtual machine . . . 29
Lexical analysis, the lexer . . . 29
Syntax analysis, the parser . . . 30
Semantic analysis . . . 31
Optimization . . . 31
Code generation . . . 32
Mozart . . . 32
2.2 The abstract syntax tree of Mozart . . . 33
Positions . . . 33
Basic types . . . 33
Equalities . . . 34
Skip statements . . . 34
Local declarations . . . 34
Operations . . . 34
Procedure and function calls . . . 35
Procedure and function declarations . . . 35
If statements . . . 35
Threads . . . 36
Successive statements . . . 36
Records, tuples and lists . . . 36
Layers and range generators . . . 37
For loops . . . 38
Step Points . . . 39
2.3 New nodes for comprehensions . . . 40
Bounded buffers . . . 40
Ranges generated by records . . . 40
Expressions . . . 40
List comprehension levels . . . 41
List comprehensions . . . 41
Collectors . . . 42
Record comprehensions . . . 43
2.4 Syntactic sugars compilation in Mozart . . . 43
The lexer, lexical analysis . . . 44
Macros . . . 44
The parser, syntax analysis . . . 45
Checking the syntax . . . 49
Coordinates in case of failure . . . 50
The unnester, transforming syntactic sugars . . . 50

3 Functional transformations 52

3.1 Functional transformations 52

An inspiration, for loops examples 52

Returning a list 54

C-style generators 55

Integer generators 56

List generators 57

Function generators 57

Record generators 57

Level conditions 60

Multi layer 61

Multi level 63

Multi output 65

Output conditions 67

Laziness 67

Bodies 69

Bounded buffers 69

Collectors 71

Record comprehension 74

3.2 Implementation 77

Architecture 77

List comprehensions 77

Record comprehensions 79

4 Tests 80

4.1 General tests 80

One level – multi layer 80

Function generators 81

Multi level – one layer 81

Multi level – multi layer 81

Record generators 81

Multi output 82

Labeled output 82

Output conditions 82

Laziness 82

Bodies 82

Collectors 83

Miscellaneous 83

Record comprehensions 83

4.2 Equivalences with other languages 83

Erlang . . . 83
Python . . . 84
Haskell . . . 85
Comparison . . . 86
4.3 Performance tests . . . 86
Space analysis . . . 88
Time analysis . . . 88
4.4 Application examples . . . 89
Sort . . . 89
Map . . . 89
Factorial . . . 90
Flatten . . . 91
Permutations . . . 92

5 Concurrency using list comprehensions 93
5.1 Laziness . . . 93
5.2 Producer – consumer . . . 94
5.3 Multi output and conditions . . . 96
5.4 Logic gates . . . 96
5.5 Bounded buffer . . . 98

Conclusion 101
Possible evolutions . . . 101
Conclusion . . . 101

References 102
Offline . . . 102
Online . . . 103

Appendices 104

A Generic transformations 105
A.1 List comprehensions . . . 105
A.2 Record comprehensions . . . 110

B Transformed AST example 113
B.1 General AST . . . 113
B.2 Pre-level . . . 114
B.3 Level 1 . . . 115
B.4 Level 2 . . . 116

C Tests **117**

C.1 One level – one layer . . . 117

C.2 One level – multi layer . . . 117

C.3 Function generators . . . 118

C.4 Multi level – one layer . . . 119

C.5 Multi level – multi layer . . . 120

C.6 Record generators . . . 120

C.7 Multi output . . . 121

C.8 Labeled outputs . . . 122

C.9 Output conditions . . . 123

C.10 Laziness . . . 123

C.11 Bodies . . . 124

C.12 Collectors . . . 125

C.13 Miscellaneous . . . 127

C.14 Record comprehensions . . . 128

D Tutorial on comprehensions **131**

D.1 List comprehensions . . . 131

Generator, ranger and layer . . . 131

Multi layer . . . 134

Multi level . . . 134

Laziness . . . 135

Bounded buffers . . . 135

Multi output . . . 136

Bodies . . . 137

Collectors . . . 137

D.2 Record comprehensions . . . 138

Introduction

TODO: introduction
Mention my github and the official one

Chapter 1

Oz and comprehensions

1.1. Oz, a functional language and more

This section describes what kind of language Oz 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 Oz, we strongly recommend [14].

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

In addition to this paradigm, Oz offers 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. Comprehensions will be available in any paradigm. The principal programming paradigms are shown in the figure 1.1.

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

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 a variable can not change, then it is impossible to make a loop that eventually stops.

Oz is full of such syntactic sugars.

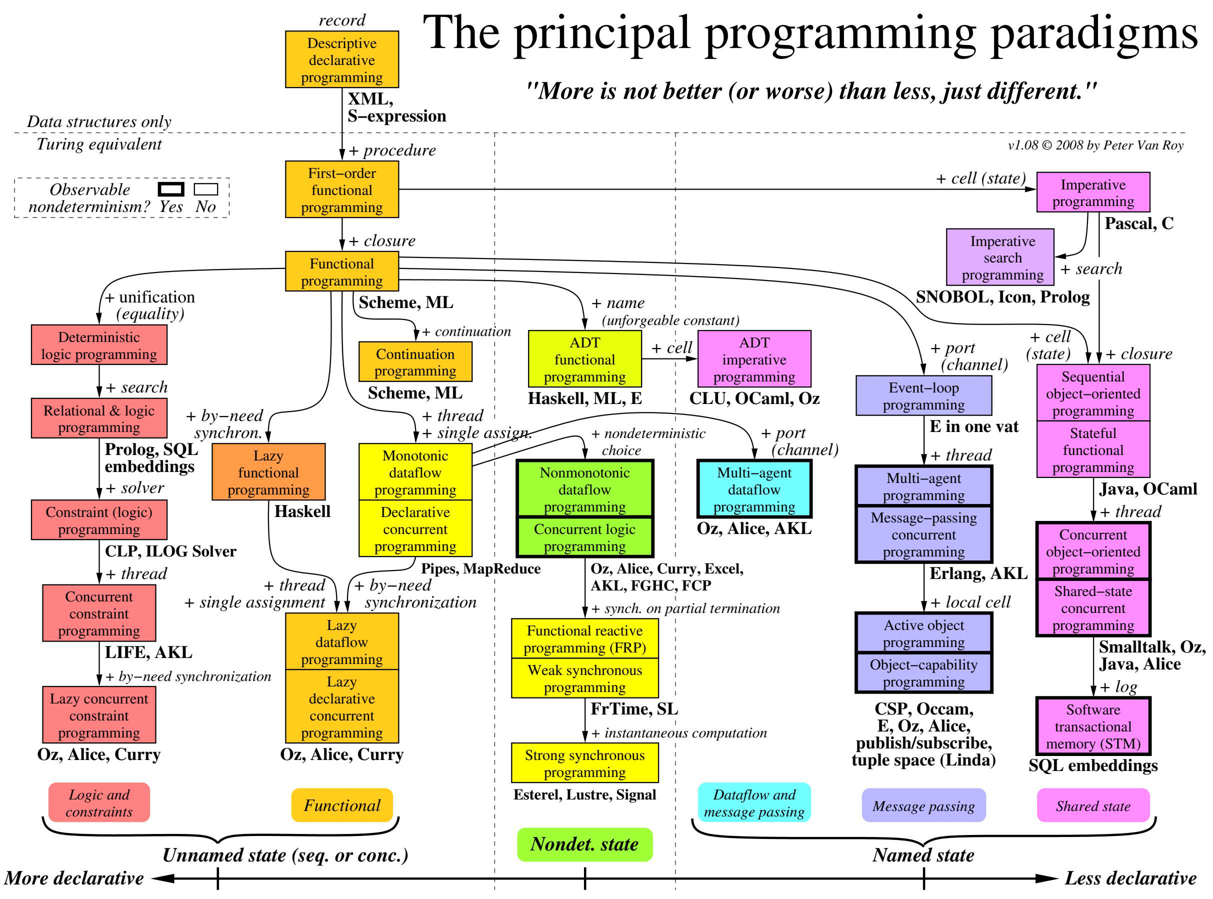

The principal programming paradigms
"More is not better (or worse) than less, just different."
v1.08 © 2008 by Peter Van Roy
record
Descriptive declarative programming
XML, S−expression
Data structures only
Turing equivalent
+ procedure
Observable nondeterminism? Yes No
First−order functional programming
+ cell (state)
Imperative programming
Pascal, C
+ closure
Functional programming
Imperative search programming
+ search
SNOBOL, Icon, Prolog
+ unification (equality)
Scheme, ML
+ continuation
Continuation programming
Scheme, ML
+ name (unforgeable constant)
ADT functional programming
Haskell, ML, E
+ cell
ADT imperative programming
CLU, OCaml, Oz
+ port (channel)
Event−loop programming
E in one vat
+ cell (state)
+ closure
Sequential object−oriented programming
Stateful functional programming
Java, OCaml
Deterministic logic programming
+ search
Relational & logic programming
Prolog, SQL embeddings
+ solver
Constraint (logic) programming
CLP, ILOG Solver
+ thread
Concurrent constraint programming
LIFE, AKL
+ by−need synchronization
Lazy concurrent constraint programming
Oz, Alice, Curry
+ by−need synchron.
Lazy functional programming
Haskell
+ thread
+ single assign.
Monotonic dataflow programming
Declarative concurrent programming
Pipes, MapReduce
+ thread
+ single assignment
+ by−need synchronization
Lazy dataflow programming
Lazy declarative concurrent programming
Oz, Alice, Curry
+ nondeterministic choice
Nonmonotonic dataflow programming
Concurrent logic programming
Oz, Alice, Curry, Excel, AKL, FGHC, FCP
+ synch. on partial termination
Functional reactive programming (FRP)
Weak synchronous programming
FrTime, SL
+ instantaneous computation
Strong synchronous programming
Esterel, Lustre, Signal
+ port (channel)
Multi−agent dataflow programming
Oz, Alice, AKL
+ thread
Multi−agent programming
Message−passing concurrent programming
Erlang, AKL
+ local cell
Active object programming
Object−capability programming
CSP, Occam, E, Oz, Alice, publish/subscribe, tuple space (Linda)
+ thread
Concurrent object−oriented programming
Shared−state concurrent programming
Smalltalk, Oz, Java, Alice
+ log
Software transactional memory (STM)
SQL embeddings
Logic and constraints
Functional
Dataflow and message passing
Message passing
Shared state
Unnamed state (seq. or conc.)
Nondet. state
Named state
More declarative
Less declarative

Figure 1.1: *Principal programming paradigms, Oz is an example of several paradigms. Source: [25].*

Expressions and statements

In Oz, 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 many 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 a 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 Oz 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 Oz. 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 Mozart 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 as a stack. Every time a new scope is declared, it is pushed onto the stack. When the end of a scope 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 = ... % Oz "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

Oz 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 = 73       % 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 Oz 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 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 = 73
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 the paragraph about cells.

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 in 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 all its arguments are declared.

Records and tuples

We just saw how variables can be assigned to procedures. Apart from procedures, variables can also be assigned to records. A record is a structure with a label and an 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 Oz 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. Four examples are given below.

```
declare R1 R2 R3 R4 Be 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 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 hand 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 the succession of $N + 1$ nested lists, one for each element and one for termination - the empty list, `nil`.

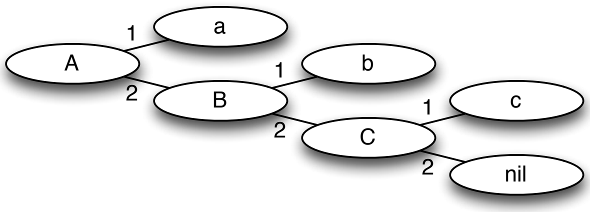


Figure 1.2: *Representation of the list layout.*

```
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 is 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

Oz 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.

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

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 elements 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. Note that recursion does not have to use lists, we use them because they are nice examples and because we will use them a lot.

```
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. Most of the time we will use the principle used in the first procedure because it keeps the order.

Threads

Oz 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:

```
1 declare Consume2 in
2 fun {Consume2 Xs}
3      case Xs
4      of nil then nil
5      [] H|Ts then Next in
6           Next = {Consume2 Ts}
7           2*H|Next
8      end
9 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. This is particularly true for list comprehensions.

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. The creation stops only when it is finished. 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 each of the elements of its result to 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.

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 Mozart in your path. You can then run the module - if it makes sense - by using

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

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
```

Dictionaries

A dictionary is comparable to an indexed table. A dictionary contains any number of items. Each item is indexed by its key and contains a value. Here is an example:

```
declare Dico in
{Dictionary.new Dico}
{Dictionary.put Dico key value}
{Browse {Dictionary.get Dico key}} % browse 'value'
```

Dictionaries implement a kind of explicit state - similarly to cells.

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
```

1.2. List comprehensions

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

Principle

This paragraph aims at explaining the concept of list comprehensions. The complete and formal syntax will be explained later.

As the name indicates, list comprehensions are structure able to comprehend, to *understand* a list - or a set - with its mathematical description. This means that they add the functionality to define - declare - 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 Oz. 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 used in the same expression along with an optional condition for each `for`.

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 is 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`.

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 list 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 parenthesis instead of square brackets to delimit the comprehension.

The condition specified with `if` allows skipping some fields in the resulting record-s. The same kind of condition as in list 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:

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

Note that record comprehensions are an experimental functionality for the moment.

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 if 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 file *Syntax.pdf* contains a tutorial on comprehensions. It can be found in appendix D.

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 Oz.

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.

```
forListDecl:alt( % a layer
    lvl0 from lvl0 % 1st lvl0 is ranger, 2nd is range, a function here
    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     lvl0 % range is a list/stream
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 D.1, 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.

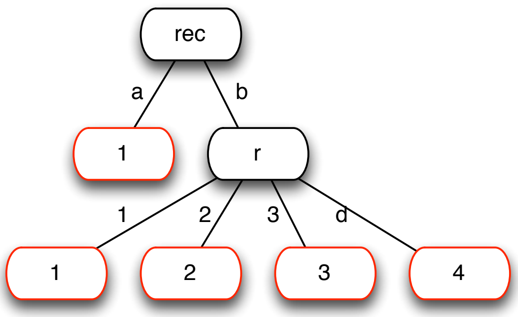


Figure 1.3: *Representation of the tree formed by rec(a:1 b:r(1 2 3 d:4)). The red indicate the leaves.*

Line 10 corresponds to a range coming from a list or a stream. The reason to differentiate list and record generators is that we will implement a specific and more efficient traversal for lists and streams thanks to their structure.

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

```
declare
fun {Fun} 2 end
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]
L8 = thread [A for lazy A in 1..10] end % L8 = _
{Value.makeNeeded L8.2}                    % L8 = 1|2|_
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 functionality we are aiming for is bodies. It 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(do phrase)]
```

The optional keyword `do` 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 do {Browse 1}]}
% browses:
% 1
% 1
% [1 2]
```

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. Collectors are specific to lists so they are not implemented in record comprehensions. On the other hand, bodies can be used 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 in lvl0 alt(
                                                    (opt(if lvl0) opt(of lvl0))
                                                    (of lvl0 if lvl0)
                                                   ) opt(seq2('do' phrase) unit
   ))
```

The alternative at the end allows users to choose the order of the filter and the decider.

Here are some examples:

```
declare
R1 = (A for A in r(a:2 b:3 1))           % R1 = r(1:1 a:2 b:3)
R2 = (A+1 for A in r(a:2 b:3 1))         % R2 = r(1:2 a:3 b:4)
R3 = (F for F:A in r(a b c))
% R3 = r(1 2 3)
R4 = (F+1 for F:A in r(a b c))
% R4 = r(2 3 4)
R5 = (F#A for F:A in r(a:1 b:r(1 2))]    % R5 = r(a:a#1 b:r(1#1 2#2))
R6 = (F#A for F:A in r(a:1 b:r(1 2)) if {IsRecord A} orelse A > 1)
% R6 = r(b:r(2:2#2))
R7 = (F#A for F:A in r(a:1 b:r(1 2)) of F == a)
% R7 = r(a:a#1 b:b#r(1 2))
R8 = (F#A for F:A in r(a:1 b:r(1 2)) if {IsRecord A} orelse A > 1
                                          of F == a)
% R8 = r(b:b#r(1 2))
C = {NewCell _}
R9 = (@C for A in r(a:2 b:3 1) do C := A) % R9 = r(1:2 a:3 b:4)
```

A more complex example where we handle a binary tree follows:

```
declare Rec = tree(key:1 left:leaf right:tree(key:2 left:leaf right:leaf))
Res = (if F==key then N+1 else N end for F:N in Rec of F==left orelse F==
   right)
% Res = tree(key:2 left:leaf right:tree(key:3 left:leaf right:leaf))
```

1.5. Complete grammar

The complete grammar can be found in the *Grammar.html* in an HTML format. Black is used for non-terminals. They are bold when their definition follows, otherwise one can click on the rule to go its definition. Purple is used for keywords and symbols. Terminals are blue. Green indicates a multiplicity such as `plus`, `star` or `opt`. Finally, each line of a definition constitutes an alternative definition.

To formalize even more the grammar, we have to write the EBNF grammar. As we only add comprehensions, we will only state what needs to be changed or added. The rest can be found in appendix C of [14]. As comprehensions are always expressions declaring records - recall that lists are records - we only need to change the rule called `<term>` because this is the rule defining the record declarations. Here are the modifications:

```
<term>        ::= ... // other rules
                | ‘[‘ { condOutput }+
                      { for <listCompDec> [ if <expression> ] }+
                      [ do <statement> ] ‘]‘
                | ‘(‘ { condOutput }+
                      for [ <variable> ‘:‘ ] <expression> in <expression>
                      (([ of <expression> ] [ if <expression> ])
                       | (of <expression> if <expression>))
                      [ do <statement> ] ‘)‘

<condOutput>  ::= [ <feature> ‘:‘ ] <expression> [ if <expression> ]

<listCompDec> ::= <expression> in <expression>
                | <expression> in <expression> ‘..‘ <expression>
                                                  [ ‘;‘ <expression> ]
                | <expression> in <expression> ‘;‘  <expression>
                                                  [ ‘;‘ <expression> ]
                | <expression> in ‘(‘ <expression> ‘;‘ <expression>
                                                  [ ‘;‘ <expression> ] ‘)‘
                | <expression> from <expression>
                | <variable> ‘:‘ <expression> in <expression>
                | lazy
```

1.6. Unofficial functionalities

The official version implements all of these functionalities. In addition to the latter, we have decided to implement more functionalities for list comprehensions that are not in the official repository. However they are implemented in our unofficial version. Let us see two additional functionalities. We will see their transformation and implementation along with the official ones as they are implemented in the unofficial version. This is why our GitHut repository contains duplicates, the official version and the unofficial one.

The first functionality, `collect`, uses the bodies. 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 do {C A}{C A+1}]
% L = [1 2 2 3]
```

The second functionality is bounded buffers. 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.

The adaptation to parse bounded buffers is the following:

```
forListGen:alt( % a range
    lvl0 .. lvl0 opt(; lvl0) % range from lvl0 to lvl0 by step 1 or lvl0
    lvl0 (: integer) % range is a list/stream, optional integer is the
   buffer size
    alt((forGenC) forGenC)) % C-style range
```

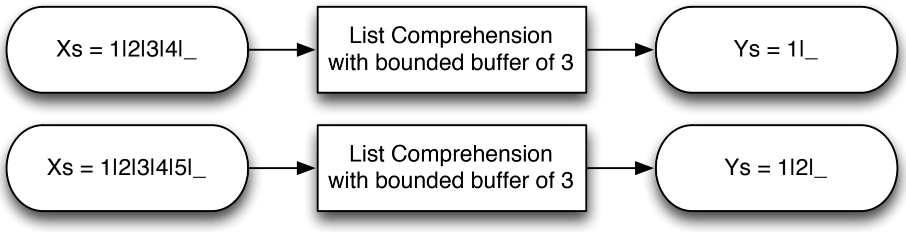

Xs = 1|2|3|4|_
List Comprehension with bounded buffer of 3
Ys = 1|_
Xs = 1|2|3|4|5|_
List Comprehension with bounded buffer of 3
Ys = 1|2|_

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

And here is an example:

```
L = [A for lazy A in thread [A for lazy A in 1..10 do {Delay 1000}]end:3] %
    buffer of 3
```

The unofficial grammar can be seen in the file *Grammar_unofficial.html*. Here are the modifications to make to the previously seen EBNF grammar:

```
<term>         ::= ... // other rules
                 | ‘[‘ { condOutput
                         | ( <feature> ‘:‘ collect ‘:‘ <expression> ) }+
                       { for <listCompDec> [ if <expression> ] }+
                       [ do <statement> ] ‘]‘

<listCompDec> ::= <expression> in <expression> [ ‘:‘ <integer> ]
                 | ...
```

Chapter 2

The compiler of Mozart

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 Mozart. Only the relevant parts for the implementation of comprehensions are detailed. All the sources are available at [23] and also in the directory *Code/mozart2*. In the sources of Mozart, the compiler is in *lib/compiler*.

Oz is compiled, not interpreted like Shell or Python. Interpreters take one instruction at a time and translate 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 Mozart. 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 Oz 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 Mozart compiler for comprehensions. We also state how these parts must be adapted to accept the transformations of the next chapter.

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

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

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 *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 of input, we take tokens as 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 Oz 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 Oz:

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

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 // this is a comment
    } 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('}')}
```

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

Syntax analysis, the parser

From the flat set of tokens created by the lexer, the parser creates an *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 one 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.

A parser defines a grammar in a comparable way that a lexer defines a lexical field. The grammar of Oz can be found in the file *Grammar.html*. A grammar is defined by all its rules.

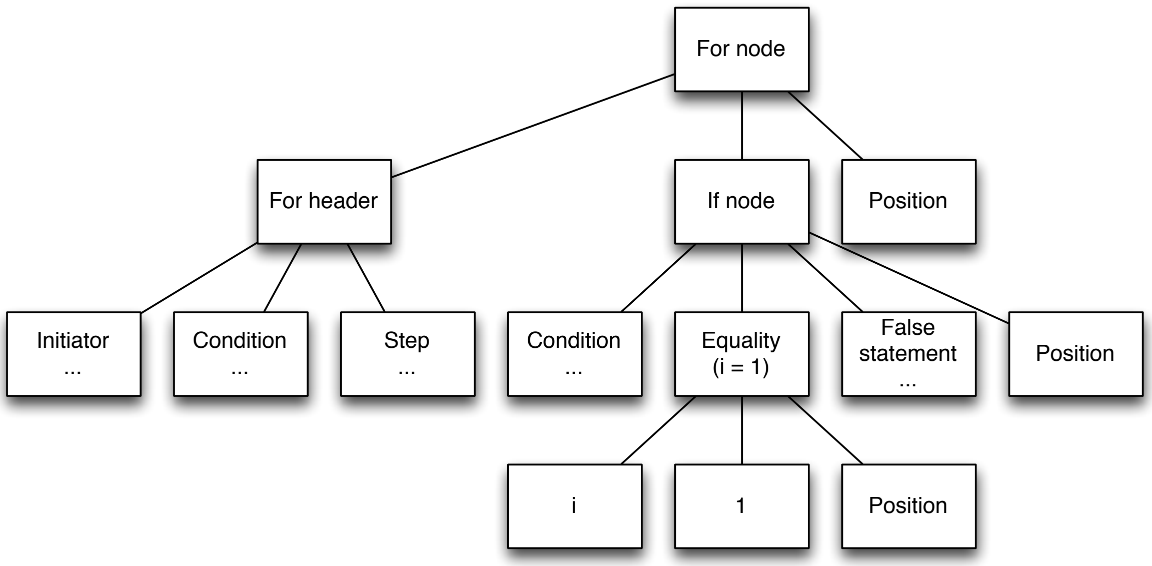


Figure 2.1: *An AST with an if statement encapsulated in a for loop. Three dots indicate a node that has been simplified, normally it should be a subtree.*

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 AST is created by the parser, 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 which 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 goes into the code to ensure its correctness.

Optimization

This module is optional. Its goal is to make the AST more efficient using some 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.

This step of the compilation can also contain some general AST transformations such as the ones we will apply to implement comprehensions.

Sometimes such optimizations are not always made 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 strongly 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 at all. 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.
Recall that our goal is to make a programming language more powerful.

Mozart

Oz is based on a relatively small syntax called the core language. Many possibilities that Oz offers are actually syntactic sugars. It means that these expressions or statements are transformed by the compiler into structures respecting the core language. The latter is detailed in [14]. Some syntactic sugars were explained in the previous chapter. Like for loops or functions, comprehensions are syntactic sugars.

The compiler of Mozart 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 that modify it. One of this pass is the transformations of syntactic sugars into their equivalent unsugared structures. This is done by the *unnester* which will be developed in the last section of this chapter.

2.2. The abstract syntax tree of Mozart

The main structure of the compilation is the one that we are interested in. It is the AST. In Mozart, nodes are tuples. The label indicates of what type is the node while the fields are all the parts of the node along with a position for some nodes. As tuples are used, the order of the children is important - except if the features are explicitly specified - and it often follows 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.

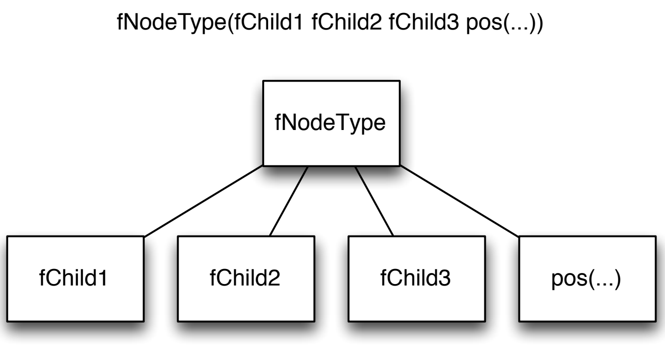


Figure 2.2: *Small Mozart AST example: node fNodeType with three children and a position.*

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 exist because it was created by the 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
```

Equalities

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

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

Skip statements

The skip statement is a statement that does nothing so its node is very simple:

```
% 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, Oz uses a recursive definition that allows to concatenate two statements. No position is 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 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 most general one - the one containing a whole list comprehension - depends on the specific ones. Most of the nodes we use have already been 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 along 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. The first one contains the current feature of the field, the second is for the value of the field, the third one is for the record itself and the last one is for the optional condition function on nested records - the decider. Here is the syntax:

```
% syntax
forRecord(Feature Value Record Condition)
% example: F:A in Rec
forRecord(fVar('F' unit) fVar('A' unit) fVar('Rec' unit) unit)
% example: _:A in Rec of Fun
forRecord(fWildcard(unit) fVar('A' unit) fVar('Rec' unit) fVar('Fun' 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. Recall that the node `fColon` is used to link an output expression to its feature. 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)) ...)
```

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. It is the root of a list comprehension, the one we will transform in the next chapter. 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 a 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.

Collectors

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)))
```

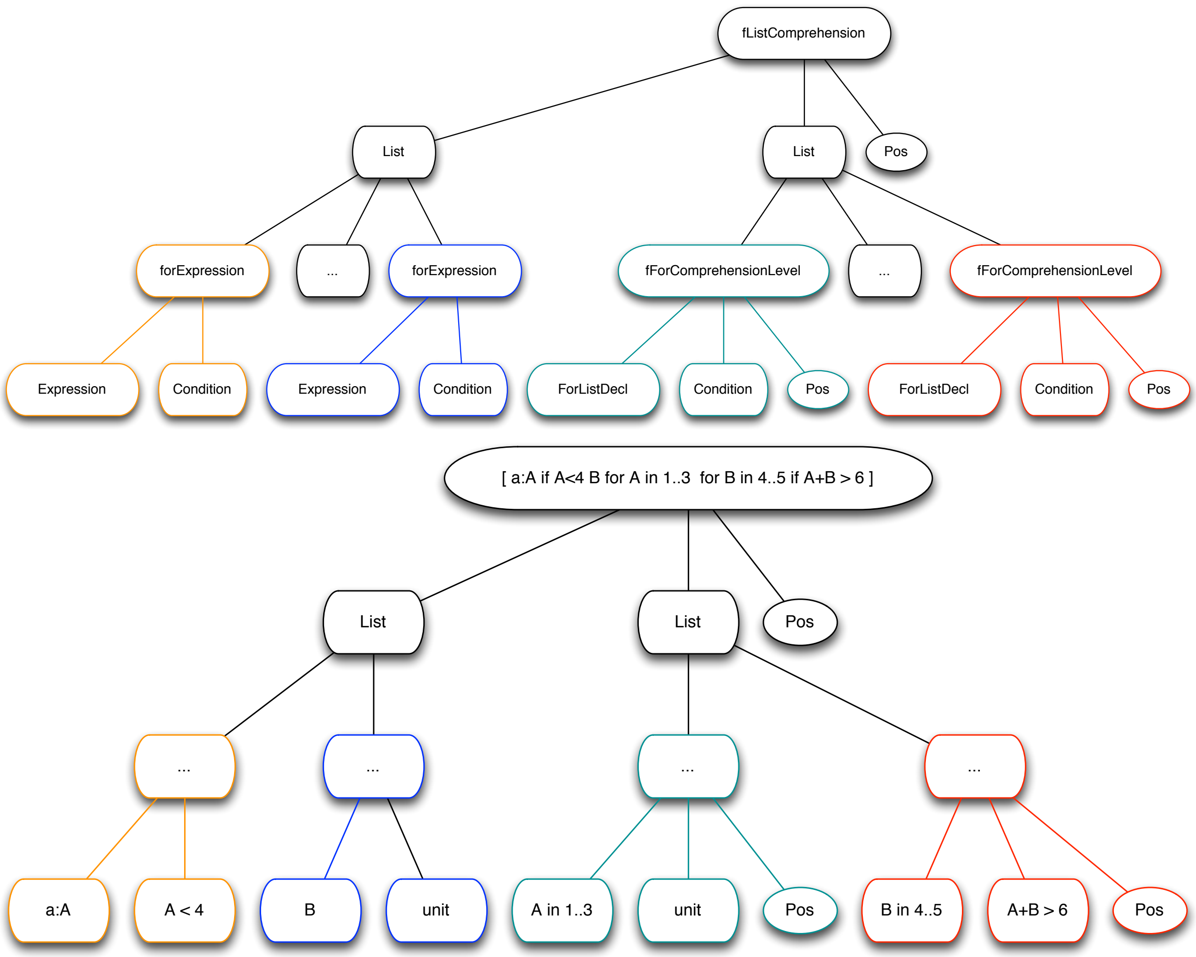


Figure 2.3: *Representation of an example of AST for a list comprehension.*

Record comprehensions

As record comprehensions imply only one level, one layer and no output condition, they are simpler to express. The new node needs seven children: the outputs - there can several so a list like in list comprehensions - the ranger, the record, the optional filter, the optional decider, the optional body and the position. The node `fColon` is used when the feature is specified.

```
% syntax
fRecordComprehension([Outputs] Ranger Record Filter Decider Body Position)
% example: A for F:A in Rec if ... with no position
fRecordComprehension([fVar('A' unit)] fColon(fVar('F' unit) fVar('A' unit))
                     fVar('Rec' unit) ... unit unit unit)
% example: a:A for A in Rec of ... do ... 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 Mozart

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 transformation of the syntactic sugar into recursive procedures and other things. The idea is to have a powerful procedure that transforms the whole transformation - one for record and the other for list comprehensions. These procedure will both 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.

The main adaptation is the parser. Indeed, the grammar changes so we have to specify the new rules coming from the new nodes we have described in the previous section and from the syntax of chapter 1.

The lexer, lexical analysis

The syntax previously seen at the end of chapter 1 does not use any extra keyword or symbol so we do not need to add anything. Indeed, we decided to reuse existing keywords. It is rarely a good idea to add keywords because simple atoms would then become keywords. This can lead to compatibility issues.

Macros

Macros are special instructions that one can use to change the state of the compiler or the state of the running Mozart. 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 parser, syntax analysis

The parser of Mozart is based on a Parsing Expression Grammar - PEG. This kind of grammars is similar to the classical Context-Free Grammars - CFGs. The main difference is that alternatives are ordered with PEGs but not with CFGs. This means PEGs do not try a rule if one of its predecessors is a match. PEGs can be parsed in linear time if memoization - see [4] - is used. The new version of Mozart uses Packrat parsing, a parser implementation in linear time - see [5]. For more information on PEGs, we recommend [6].

The part of the parser that interests us is the definition of the syntactic rules. These rules define the grammar of Oz. Every symbol has a rule, typically composed of several ordered alternatives. Each alternative is followed by a function called when the parser chooses the corresponding rule. This function must return the AST elements of the rule.

First let us introduce some useful general rules used by the new one:

```
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
pB               % rule for beginning position
pE               % rule for ending position
modifier(rule)   % recursive definition
[rule ... rule]  % an ordered sequence of rules
otherRule        % another rule
token            % a token
```

Along with the definition of a rule is a function called when the rule is chosen by the parser. It returns the corresponding node. An example of the format of a rule is the following:

```
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('do' 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
                    'in' lvl0 opt(seq2('if' lvl0) unit) opt(seq2('of' lvl0)
                    unit) opt(seq2('do' phrase) unit) ')' pE]
              #fun{$ [P1 _ S _ F L1 _ L2 IF OF DO _ P2]}
                    if F == unit then
                         fRecordComprehension(S L1 L2 IF OF DO {MkPos P1 P2})
                    else
                         fRecordComprehension(S fColon(F L1) L2 IF OF DO
                                                   {MkPos P1 P2})
                    end
               end
              [pB '(' plus(subtree) 'for' opt(seq1(lvl0 ':') unit) lvl0 'in'
                   lvl0 'of' lvl0 'if' lvl0 opt(seq2('do' phrase) unit) ')' pE]
              #fun{$ [P1 _ S _ F L1 _ L2 _ OF _ IF DO _ P2]}
                    if F == unit then
                         fRecordComprehension(S L1 L2 IF OF DO {MkPos P1 P2})
                    else
                         fRecordComprehension(S fColon(F L1) L2 IF OF DO
                                                   {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 - then it means that the expression part of the list comprehension is made of one expression with no feature - or a tuple instead of a record. 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 - recall that lists and tuples are specializations 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 the definition of record fields 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 features as for loops like `break:Break`.

Note that the order of the alternatives are important since pattern are tested in order until any match is found.

```
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

Mozart 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 Do C) then
        {Phrase S}         % check all expressions
        {Phrase A}         % check ranger
        {Phrase R}         % check record
        {Phrase F}         % check filter
        {Phrase Cd}        % check condition
        {Phrase Do}        % check body
        {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 for the user. The file *TupleSyntax.oz* implements a function returning the coordinates of a given node. The function abstracts the fact that some nodes do not have a position child. In that case, we must call recursively the function with the child having the position. 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
```

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 comprehensions.

The complete pseudo-code for the transformation of list comprehensions is in appendix A.1. The one for record comprehensions is in appendix A.2.

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 different 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:

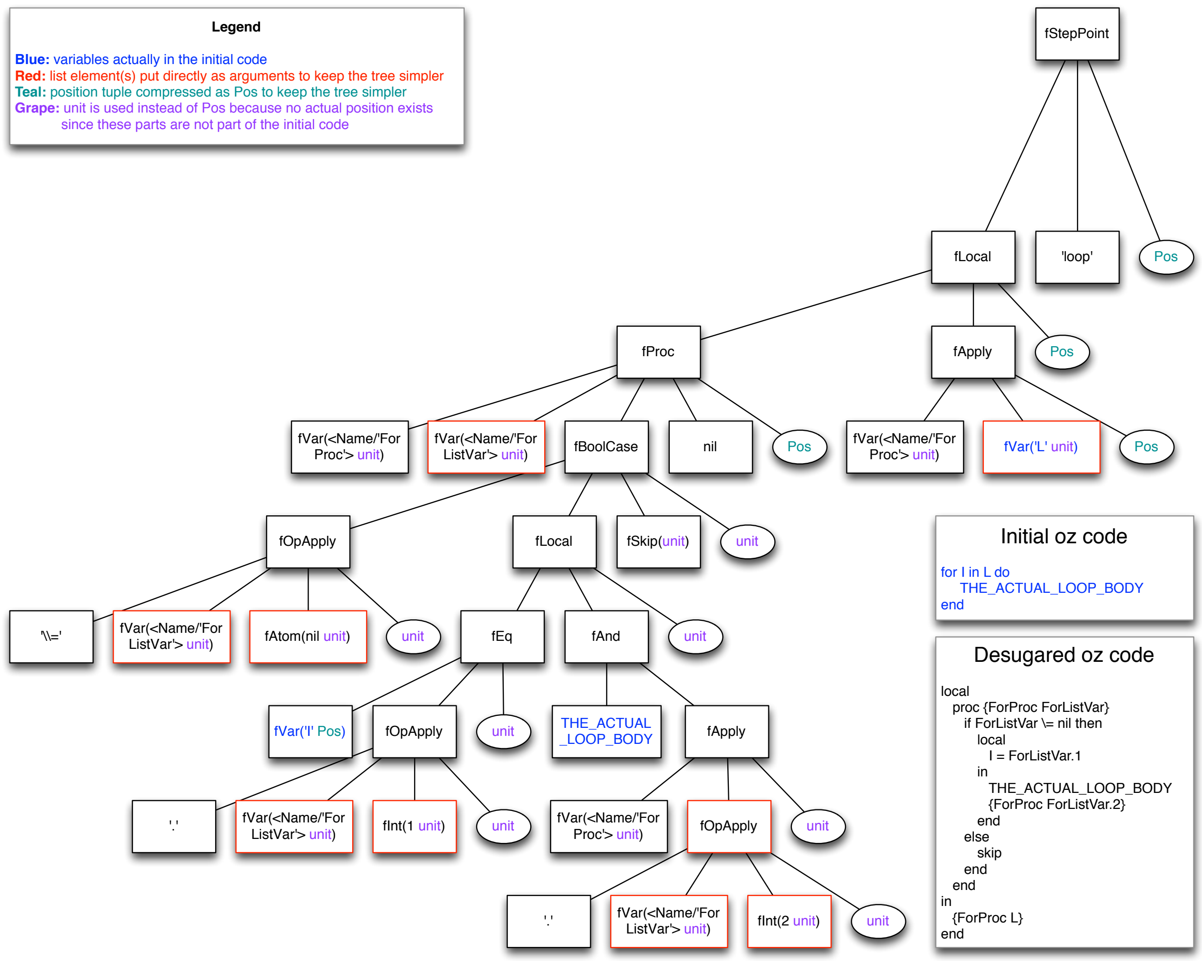
```
for A in [1 2] do ... end
```

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 Mozart transforms such a for loop into the AST represented in figure 3.1.



Legend

Blue: variables actually in the initial code
Red: list element(s) put directly as arguments to keep the tree simpler
Teal: position tuple compressed as Pos to keep the tree simpler
Grape: unit is used instead of Pos because no actual position exists since these parts are not part of the initial code

Initial oz code

```
for I in L do
   THE_ACTUAL_LOOP_BODY
end
```

Desugared oz code

```
local
   proc {ForProc ForListVar}
      if ForListVar \= nil then
         local
            I = ForListVar.1
         in
            THE_ACTUAL_LOOP_BODY
            {ForProc ForListVar.2}
         end
      else
         skip
      end
   end
in
   {ForProc L}
end
```

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

Nodes are represented by rectangles except position nodes which are inside circles. Some variable names are created by the compiler itself. In that case, their 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 red rectangles.

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 function generators. The main principle is exactly the same: one recursive procedure. There are four modifications to make for the other generators which 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 created 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 and 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 is going to be used after the call. This is better for both time and memory. If we ensure terminal recursion then laziness and handling streams will be simple and straightforward.

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. The latter must have been declared beforehand. This technique ensures keeping the order - recall that accumulators reverse the output.

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*A 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. This the basis for the remaining of this chapter.

C-style generators

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.

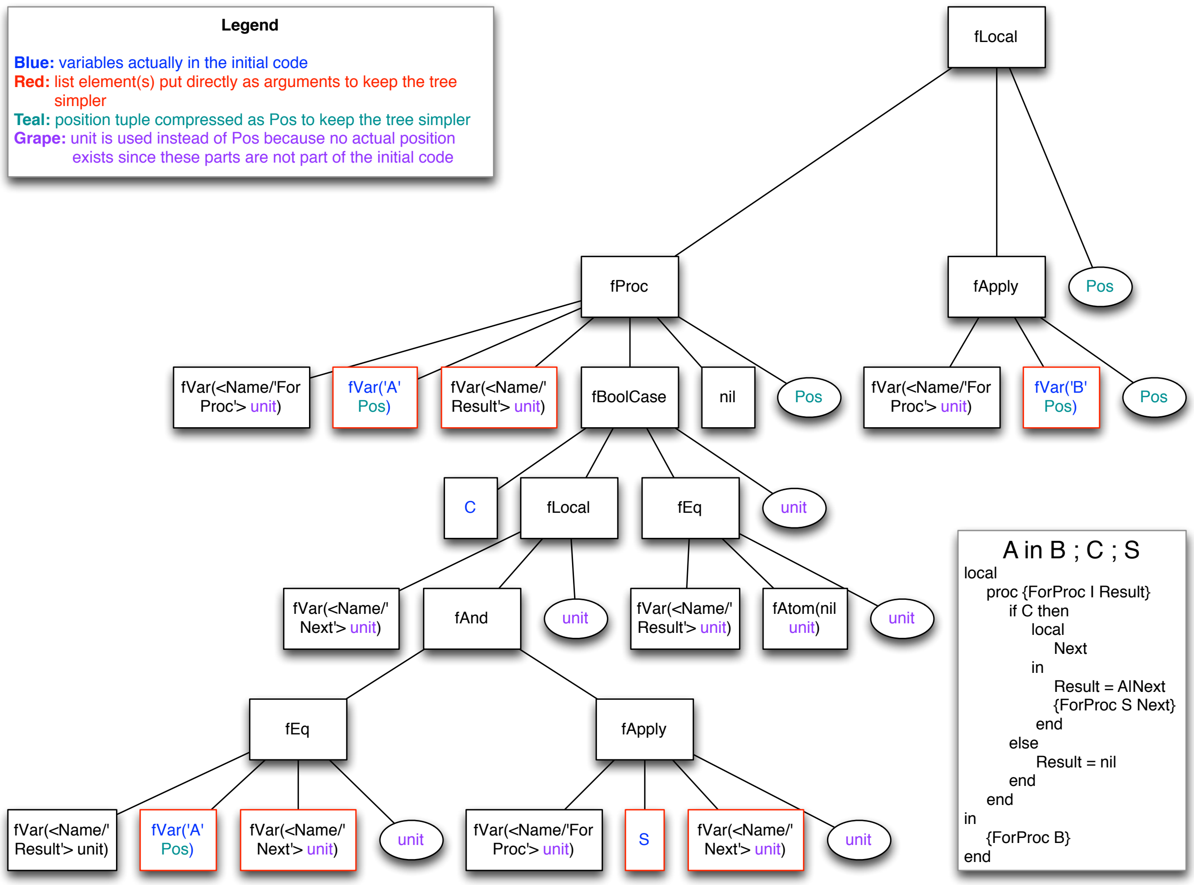
Here is a complete example. The procedure becomes:

```
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
```

A representation is depicted in figure 3.2.


Legend

Blue: variables actually in the initial code
Red: list element(s) put directly as arguments to keep the tree simpler
Teal: position tuple compressed as Pos to keep the tree simpler
Grape: unit is used instead of Pos because no actual position exists since these parts are not part of the initial code

A in B ; C ; S

```
local
   proc {ForProc I Result}
      if C then
         local
            Next
         in
            Result = A|Next
            {ForProc S Next}
         end
      else
         Result = nil
      end
   end
in
   {ForProc B}
end
```

fLocal
fProc
fApply
Pos
fVar(<Name/'For Proc'> unit)
fVar('A' Pos)
fVar(<Name/' Result'> unit)
fBoolCase
nil
Pos
fVar(<Name/'For Proc'> unit)
fVar('B' Pos)
Pos
C
fLocal
fEq
unit
fVar(<Name/' Next'> unit)
fAnd
unit
fVar(<Name/' Result'> unit)
fAtom(nil unit)
unit
fEq
fApply
fVar(<Name/' Result'> unit)
fVar('A' Pos)
fVar(<Name/' Next'> unit)
unit
fVar(<Name/'For Proc'> unit)
S
fVar(<Name/' Next'> unit)
unit

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 generators

Integer generators can be 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 generic integer generator:

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

It can always 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. Integer generators can then be considered as a nice expressive way to create sequential lists as in Matlab.

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

List generators

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 require 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 a coded example.

Function generators

The last kind of generator are function 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 generators

We previously stated that there are 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 Oz. 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 nesting of records onto the stack. We 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 - onto 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 - as invariant - 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:

```
proc {FindNext stacks(FeatStack ValueStack) ?Result}
   local
      Feat = FeatStack.1 % top feature
      Val = ValueStack.1 % top value
      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
```

Our invariant is always verified because the modifications 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.

Each generator implies a time complexity in $O(n)$ where n is the number of elements of the generators - the number of nested record in the case of record generators. For function generators n is infinite.

Level conditions

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. The implementation of such conditions is the same with one or several layers and/or levels.

What a condition says is that we can 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. One could decide to write only one recursive call by always creating `Next` and assigning it to `Result` if the condition is false. This would work but it is more efficient not to do it because it brings extra variable assignments and declarations.

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 the 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 to 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 - level 1 - calls child - level 2.
{Level2 3 A}
%% Child - level 2 - calls back parent - level 1.
{Level1 A+1}
```

Note that we must pass all rangers to levels that follow them because they can be used 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 a generator is a list. 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 = Arg1.1 in
            {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 = Arg1.1 in
            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 then raise an error.

Once we have this 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 if no error occurs.

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.

To make the implementation easier and cleaner, we have decided that the level procedures always act as if the output was a record of list-s. Even if there is only one non-featured output, so even if we must output a list, the level procedure handle a record of list-s. When the output is a list, we need to adapt the pre-level a bit so that the output list is the only of the record. Here is the modification:

```
%% Pre-level
proc {PreLevel ?Result}
    {Level1 {{ Initiators }} '#'(1:Result)}
end
```

Thanks to this trick, the output is a list when it needs to be one while all the level procedures do not care about this. Consequently they are simpler because they do not have to handle returning a list.

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

Output conditions

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 even if it is transparent for the user.

With level conditions, we filter iterations that do not fulfill the level condition. 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 and this action must be output specific.

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 the current list to assign to the next list to assign if the condition is not verified. Here is the transformation:

```
%% 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 while all the previous functionalities still work.

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

Laziness is implemented using the procedure `WaitNeeded` that waits - sleeps - until the value passed as its only argument is required for a computation or anything else that makes it needed. So the only thing to do is to put some `WaitNeeded` at the right places inside the level procedure. Recall that there can be one lazy flag per level.

The right place to put the `WaitNeeded` call is at the very beginning of the procedure. One could say that it should be closer to the addition of elements or to call to the next level. If we do that then the level condition is tested. If the generator is a lazy stream then we make one element needed before using it. This is not what we want so we wait before doing anything in level procedures.

When there is only one output, we can simply call `WaitNeeded` as follows:

```
%% Only one element, its feature is F
{WaitNeeded Result.F}
```

When there is more than one output, the procedure must be woken up when any of the output lists has been made needed. This requires to wait for all output in parallel. We need to create one thread for every output. Each thread waits for its output to be needed. In the main thread - the one of the procedure - we declared a variable reachable inside the waiting threads. This variable is unbound. Any thread that does not have to wait anymore, because its output was made needed, assigns this variable to `unit`. This will not lead to an error since all threads assign the variable to the same value, `unit`. After the declaration of all threads, the procedure waits for this variable to be bound.

Once it is bound, it means that at least one output was needed so the procedure must go on executing. Since the other outputs will be bound as well as the one needed, all the threads will terminate - `WaitNeeded` stop waiting when its argument is bound. Here is the adaptation:

```
%% Features are E F G
local
   LazyVar
in
   thread {WaitNeeded Result.E} LazyVar = unit end
   thread {WaitNeeded Result.F} LazyVar = unit end
   thread {WaitNeeded Result.G} LazyVar = unit end
   {Wait LazyVar} % wait for LazyVar to be bound
end
```

This is enough to make any level procedure lazy.

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

Bodies

The body of a list comprehension is just a statement - that can be composed of many statements - executed just before adding elements to the output. This is the place to put the body because it is like the body of some nested loops. We decided to put this body before appending elements to the output because this makes the use of the `Delay` procedure more convenient. Furthermore we think it is more intuitive.

To sum up, all the instructions of the body are put just before assigning the output the next output to assign in the last level procedure. An example will be given with collectors.

The file *Transformations/List_comprehensions/Tr_Ex_Body.oz* contains a coded example of a list comprehension with a body along with other functionalities.

Bounded buffers

Before transforming bounded buffers, let us see the principle used to implement a bounded buffer. Recall that this functionality is unofficial.

Rephrasing what a bounded buffer does leads to this definition: to handle a buffer of size n on the list L, one can keep track of the list n elements ahead of L. So this means that we are going to keep two variables, one with our current list - the same as for list generator - and one n elements ahead. The latter ensures that there are at least n elements - following the current one - that are marked as needed. The figure 3.3 gives a graphical example.

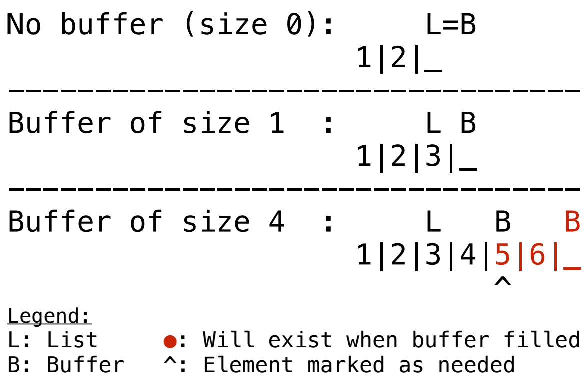


Figure 3.3: *Representation of bounded buffers.*

So what we need to do is to declare a variable holding the buffer. The latter must be n elements ahead of the corresponding list. Then when update the list to its second element then we do the same for the buffer. After the buffer is initialized, it is like a mirror of the list. A buffer is qualified of full when it is exactly n elements ahead of its corresponding list - the list might catch up with the buffer.

To implement this, we first need to keep track of the buffer. So we will put the buffer in a tuple together with its corresponding list instead of just putting the list as argument of the level procedure. Every time we take the second element of the list, we do the same for the buffer. This only happens in the next call.

There are still two important things to think about for the procedure. First the buffer will always reach the end of the input list before the list argument. So we must take this case into account. When the buffer has reached the end, it does not have any effect anymore. Second, we have to put computations on the buffer inside other threads. This is because we want the list to be able to go through the elements already created independently of the creation of the elements. Without these threads we would force the buffer to *always be full* but this is not how it should work so we must use threads. Here are the adaptations to the level procedure:

```
%% Signature: arguments (before: Arg1)
Arg1#Buffer1
%% Next call (before: Arg1.2)
Arg1.2#thread % inside a thread to make Arg1 independent of Buffer1
            if Buffer1 == nil then
                Buffer1 % nothing if buffer has reached the end
            else Buffer1.2 % mirror action
            end
       end
```

The declaration does not change, nor does the condition. We still have the list and that is all we need for those.

The initiator must change. We must set the second element of the tuple to the list n elements ahead of the input. To do this, we use the function `List.drop` that does exactly what we need. Here is the new initiator:

```
%% Buffer size is N
%% New initiator (before: List1)
List1#thread {List.drop List1 N} end
```

The buffer is created inside a thread because we do not to wait until the buffer is full to begin iterating.

The file *Transformations/List_comprehensions/Tr_Ex_Bounded_Buffer.oz* contains a coded example of a list comprehension with a bounded buffer along with other functionalities.

The reason why this functionality is unofficial is that one can use bounded buffers without the functionality implemented. Usually a bounded buffer is implemented using a lazy function that takes a list/stream as only argument and returns it lazily. Transparently, there is the buffer handled. Here is this function:

```
%% Ss : list or stream
%% N  : size of the buffer
fun {BoundedBuffer Ss N}
   fun lazy {Aux Ss End}
      case Ss of nil then nil
      [] H|T then
         H|{Aux T thread if End==nil then End else End.2 end end}
      end
   end
   %% Initialize End to be N elements in advance
   End = thread {List.drop Ss N} end
in
   {Aux Ss End}
end
```

So using this function, we can do exactly the same as what we added as unofficial functionality.

Collectors

The body being transformed, we can now move on to collectors. To implement collectors, we need to use explicit state *in extenso* cells. Indeed, the call to a collector can happen at any time and we have no way of handling the change of state in a purely declarative environment.

Using cells does not change the fact that list comprehensions are declarative. From the outside one can not see the difference so it is acceptable to use cells. One constraint must be respected though. At some point we will access the cell and then write a value. This must be done atomically - the first can not happen without the other happening right after. This is done using the procedure `Exchange`. It ensures the read and write operations to be atomic. This procedure takes three arguments. The first is the cell. The second is an expression that will be equal to the content of the cell. The third is the new value to put in the cell.

The principle for a collector is that we keep track of a cell storing the initial list that can be filled by calling the collector. Every time we call this collector, we have to assign the list to the given element followed by a new unbound list that will replace the value of the cell. So for now, we have a cell and a procedure for each collector:

```
%% Cell
CellC = {NewCell _} % cell contains an unbound list
%% Procedure
proc {CollectorC X} % X is the element to add
    local N in % new unbound list
        {Exchange CellC X|N N} % old unbound list is assigned to X|N
        % cell now contains N, the next list to assign
    end
end
```

What the `Exchange` does is first to ensure the equality between the content of the cell and `X|N`. Recall that, thanks to declarativity, we can write `A=42` or `42=A` to express exactly the same instruction *in extenso* that the two values must be equal to each other. Here `A` is the only one that can be assigned, so Oz forces it to be equal to 42, leading to an error if `A` was already bound to another value. Calling `Exchange` is similar to executing the following instructions atomically:

```
%% Exchange call
{Exchange CellC X|N N}
%% Non atomic equivalent
X|N = @CellC % or @CellC = X|N
CellC := N
```

Ending the list can be done by calling:

```
{Exchange CellC nil _}
```

We do not care about the new value of the cell, once the list is finished, we can forget the cell.

Now that we have the infrastructure to handle collectors, we have to integrate them into our transformation. First we must link the initial content of the cell to our output. This can be done inside the pre-level. We just need to state that the field of the output record - recall that we always handle a record and never a list with collectors because the feature is mandatory - is the initial content of the cell. So when we assign the result inside the pre-level, collectors are handled as follows

```
%% [c:collect:C A for A ...]
Result = '#'(1:Next1 c:@Cellc)
```

But as we do not need this field because we never assign it and the user never uses anything than the collector procedure, we can omit it inside the result record passed by the pre-level to the first level. So the call to the first level becomes:

```
%% [c:collect:C A for A ...]
{Level1 {{ Initiators }} '#'(1:Next1)}
```

Indeed, the collected list in not concerned by adding new elements by the list comprehension itself. However, this implementation requires that the cell is accessible anywhere inside the list comprehension because the body might use them, or any other part of the list comprehension. The `WaitNeeded` needs it as well. So the cells and the collector procedures are declared in the same scope as all the level procedures.

Let us now end the list at the end of the list comprehension. We already stated how to end the list. This instruction must be executed at the same place where we assign the outputs to `nil` in the normal case.

The last thing to do is to handle laziness. We do as for the normal output except that we wait for the current content of the cell to be needed. All in all this gives us the following transformation:

```
%% List comprehension
[c:collect:C for lazy A in GENERATOR do BODY]
%% Transformation
local
     Cellc = {NewCell _}
     proc {C X} N in {Exchange Cell1 X|N N} end
     proc {PreLevel ?Result}
          Result = '#'(c:@Cellc)
          {Level1 1 '#'()}
     end
     proc {Level1 A ?Result}
          {WaitNeeded @Cellc}
          if {{ Condition of GENERATOR }} then
               {{ BODY }}
               {Level1 {{ Next call of GENERATOR }} '#'()}
          else
               {Exchange Cellc nil _}
          end
     end
end
```

The file *Transformations/List_comprehensions/Tr_Ex_Collect.oz* contains a coded example of a list comprehension with collectors along with other functionalities.

The reason why this functionality is unofficial is that one can use collectors without the functionality implemented. Furthermore, it does not add any expressivity. It is just sometimes a more convenient way to use list comprehensions. Here is how one can use collectors without the implementation we have just detailed:

```
%% With a collector
{Browse [1:collect:C for A in 1..5 do {C A}].1}
%% Equivalent without a collector
local L in
   local C in
      C = {NewCell _}
      L = @C
      _ = [_ for A in 1..5 do local N in {Exchange C A|N N} end]
      @C = nil
   end
   {Browse L}
end
```

Record comprehension

The transformation concerning record comprehensions will use some parts of the one for list comprehensions but some parts will differ since we have to output a record - or a record of records.

Throughout this transformation, we will give the transformation for the following record comprehension:

```
(A+1 for A in r(r1(1 2 3) r2(4 5 6))
      if {Not {IsRecord A}} orelse {Label A} == r2 of true do {Browse A})
```

The first thing about record comprehensions is that we only allow one level and one layer. So we only have one recursive level procedure in addition to the pre-level.

The pre-level is the same as before, the only thing that changes is the initiator but this concerns the only generator of record comprehensions, record generators. Such generators are not the same as in list comprehensions because we do not traverse them in the same way. The initiator is simply the record itself. Similarly to list comprehensions, we can output several records. So the pre-level is as follows:

```
proc {PreLevel ?Result}
    {Level r(r1(1 2 3) r2(4 5 6)) '#'(1:Result)}
end
```

On the other hand, the level procedure is very different. When the output is a list, filtering elements is easy because we do not care about the exact position of the element inside the list. However when we output a record, we have to know in advance if a field will be there or not. This requires a pre-process step. More precisely, each record that is not considered as a leaf has to be preprocessed to compute its children that have to be considered. The pre-processing output two lists. The first is a list of the features that will be in the result. The second is a list a boolean values of the same size as the arity of the input record. Each boolean i of the list corresponds to the element i of the arity. If the boolean is true, then it means the feature must be outputted and so the first list contains this feature. If the boolean is false, then the feature is not outputted and it is not in the first list. The procedure doing this is looping on all fields of its input record and tests the level condition, the filter. Here is this procedure:

```
%% Procedure
proc {For1 Ari Rec ?AriFull ?AriBool}
   if Ari \= nil then % for all elements of the arity
      local
         Feat = Ari.1 % not feature in ranger, not create it
         A = Rec.Feat % same A as in ranger
         NextFull  NextBool
      in
         % evaluate the condition for the current field
         AriBool = {Not {IsRecord A}} orelse {Label A} == r2|NextBool
         % include feature iff condition is true
         AriFull = if AriBool.1 then Feat|NextFull else NextFull end
         {For1 Ari.2 Rec NextFull NextBool} % recursive call
      end
   else
      AriFull = nil  AriBool = nil
   end
end
```

Now by calling this procedure, we get the results with which we can create the output-s. With all this data, we can call another procedure. The latter assigns output leaves to their value and calls the level procedure if a field is record. So indirectly the level procedure recursively calls itself. Here is the level procedure:

```
proc {Level Rec ?Result}
   local
      Ari = {Arity Rec}
      Lbl = {Label Rec}
      AriFull  AriBool
   in
```

```
        {For1 Ari Rec AriFull AriBool}
        Result.1 = {Record.make Lbl AriFull} % create output
        {For2 Rec AriFull AriBool Result}
    end
end
```

The function `Record.make` returns a record with the specified label and features and with all values unbound.

The last thing to implement is the procedure `For2`. This procedure goes through all fields that have passed the filtering. If the field is a record and the decider is true, then we call the level procedure, this is the indirect recursive call. When the field is - considered as - a leaf, then we output the expression given in the record comprehension after executing the body if any. Here is the code:

```
proc {For2 Rec AriFull AriBool ?Result}
    if AriFull \= nil then % for all fields that passed the filtering
        if AriBool.1 then % if filter is true
            local
                Feat = AriFull.1 % not feature in ranger, not create it
                A = Rec.Feat     % same A as in ranger
            in
                if {IsRecord A} andthen {Arity A} \= nil
                   andthen true then % decider is true
                    {Level A '#'(1:Result.1.Feat)}
                else
                    {Browse A} % body
                    Result.1.Feat = A + 1
                end
            end
            {For2 Rec AriFull.2 AriBool.2 Result}
        else % filter is false, go to next field
            {For2 Rec AriFull   AriBool.2 Result}
        end
    end
end
```

Understanding this procedure shows that we traverse the tree formed by the input record in depth-first mode as for list comprehensions. We can see that decider has the same place as in list comprehensions. A difference is that here it is not a function. The decider has access to `A` because it is declared in the ranger. Here we declare the ranger in the procedures `For1` and `For2`, so that respectively the filter and the decider have access to the variables declared in the ranger. As the feature is not declared in the ranger in our example, we create our own variable.

The file *Transformations/Record_comprehensions/Tr_Ex.oz* contains a coded example of a record comprehension.

3.2. Implementation

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

Architecture

Our goal is to replace the nodes `fListComprehension` and `fRecordComprehension` into their respective transformation in the AST. We already saw how to transform them functionally so now we will detail the architecture to implement these transformations. We will not explain all the details of the implementation. For this we recommend reading the commented code in the file *Modifications/ListComprehension.oz* and *Modifications/RecordComprehension.oz* which are respectively the same as *mozart2/lib/compiler/ListComprehension.oz* and *mozart2/lib/compiler/RecordComprehension.oz*. These two couples of files are respectively almost identical to *Transformations/List_comprehensions/Test_ListComprehension.oz* and *Transformations/Record_comprehensions/Test_RecordComprehension.oz* which respectively implement the same function but that can be tested directly with the examples provided inside the file. The two latter are the files we used to test our implementation before compiling Mozart.

The main function is called `Compile`. It takes the comprehension node as argument and returns the complete transformation tree rooted at `fStepPoint`. As seen before, the latter node has three children, the first - the body - is the transformation itself, the second is a label set to `'listComprehension'` or `'recordComprehension'` depending on which comprehension we are dealing with. The last child is the position that we set to the position of the comprehension. The body is a `fLocal` because we need to declare the pre-level and level procedures. The body of this main `fLocal` is just a call to the pre-level. For now on, every position is set to `unit` except when we use the data given in the comprehension or when we create threads - because they require having a position.

Each level is created by a function that puts the level procedure inside a dictionary and then returns the variable containing the function.

We now separate the explanations for list and record comprehensions.

List comprehensions

The pre-level is created by a function that in turn calls a big function that creates levels. The pre-level calls this big function to create the first level which then calls it again to create the second level and so on. This way levels can pass all the information needed to their child. Each level takes all the arguments of its parent plus its own ones, so a level receives a list of its parent arguments, add its own ones and gives this list to its child.

Before calling the creation of the first level, the pre-level calls a function that parses all the output specifications - the list of `forExpression` - so that the first, last and lazy levels - the only ones that need to know what the output is - have all the data they need. This function handles - missing - features and collectors. It returns the collectors, the fields, the values and the output conditions in a convenient way for the rest of the compilation of the comprehension.

Each level receives the list - a list of `fForComprehensionLevel` - of all the levels after the current one as well as the index of the current level. The first level receives the complete list from the pre-level.

Each level calls a function that parses all the ranges of a given level. This function returns the new arguments for this level, the declarations to make, the iteration conditions, the arguments for the next level and whether the level is lazy.

With this data from the function and its arguments, the level function has enough to create the AST of its level procedure.

Other functions exists to help and to make things more simple and/or clear but the principle stays the same. One example is the function that creates all the range initiators when a `fForComprehensionLevel` node is given.

Another example is the function that declares the cells for collectors. The latter puts the cell declaration inside the same dictionary where the level procedures are and returns the variable containing the cell.

A last example is the function that declare the function `FindNext` used for record generators. This latter can be declared in two flavors: one with a discriminate function given, the other one without. The function declaring these function also puts them in the dictionary and returns the variables containing the functions.

An example of the complete AST created by the following list comprehension can be found in appendix B.

```
[A B for A in LA for B in BB ; CB ; SB if A+B > 4]
```

Record comprehensions

Record comprehensions are simpler to implement because their structure is more constant. There always are the pre-level, one level, the `For1` and `For2 procedures`. We also use a dictionary to declare these four procedures.

The function parsing the output specifications is very similar but is more simple because there are no collectors allowed. Initiators are also simpler because it is always a record.

In general the implementation of record comprehensions has been greatly inspired from the implementation of list comprehensions for two reasons. First, there are less flexible and basically a subset of the functionalities of list comprehensions. Secondly because we implemented list comprehensions first.

Chapter 4

Tests

This chapter goes through all the tests that our solution had to pass. All these tests can be found in the directory called *Tests* or in the directory *tests/comprehensions* of the Mozart2 official repository at [23]. A shell script named *run.sh* allows testing them in a systematic way.

As we use different files, 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 *Tester.oz*. This file also contains other more specific functions explained when needed.

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 implementation 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. All general tests can be found in appendix C.

We have decided to write the tests incrementally. First because it followed our incremental implementation of comprehensions. Second because, this way we can easily see until which set of tests everything is fine. We can also detect the kind of bug easily with different files. This allows us to run only a subset of all the tests too.

The first part has one goal: test whether the simplest use cases of list comprehensions work. For these basic tests, only list comprehensions with one layer, one level and one non-featured output without condition are tested. Further, there are no body nor collectors. We test all the possible ranges except the ones with `from` or generated with records. They are tested in two other specific files explained later. We also test level condition and expressions coming from function calls. These 12 tests are in section C.1.

One level – multi layer

The next incremental step is to allow several layers. We limit ourselves to the same generators as before and level conditions are also tested. We also test empty rangers set to `_`, the wildcard. It means that we do not use it but the execution is the same. It is important to test it because our implementation can not make the assumption that the user will always provide a non-empty ranger. These 12 tests are in section C.2.

Function generators

We only have tested ranges that use the `in` keyword - and not records yet. 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. We do the same kind of tests as in the previous paragraph but we also use function generators now. To be complete, we must test function generators where a variable is given and where the function is directly given inside the list comprehension. These 9 tests are in section C.3.

Multi level – one layer

Now we can test several levels. To differentiate errors that come only from several levels, we first test them with one layer. Again, we limit ourselves to the same generators as before and level conditions are also tested and we also test ranger set to the wildcard. These 7 tests are in section C.4.

Multi level – multi layer

Now that we have tested multi level and multi layer separately, we can test them together. We also have tested function generator so we mix all these together. We also test level conditions for several levels. At this point, we have tested the basis of lists comprehensions. Most other languages implementing list comprehensions would pass all the tests up to now. The tests to come check more *exotic* functionalities. A comparison will be done in the next section. These 4 tests are in section C.5.

Record generators

Now comes the time to test iterating over a record. The two arguments boolean function allowing the discrimination between nodes must also be tested as well as level conditions. We also need to test empty features and/or values *in extenso* when the feature and/or the value are set to the wildcard. As we already tested multi layer and multi levels, we must test them with record generators too. Finally, the tests help ensuring that only the leaves of the record are considered in a depth-first mode. These 20 tests are in section C.6.

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 Oz list comprehensions. For now, we restrict ourselves to non-featured outputs. This is to follow the implementation process we followed. This way, we can more easily detect errors due to user specified features. As this functionality is independent of the generators, we do not test all of them again.

On the other hand, we still try level conditions, multi level and multi layer. These 7 tests are in section C.7.

Labeled output

The previous paragraph ensures that the multi output functionality works so we now can test giving a feature to some outputs. Note the fact that features in one list comprehension must always be all different otherwise the transformation can not access all fields. The tests are similar to the ones of the previous paragraph. These 8 tests are in section C.8.

Output conditions

Output specifications have been tested but we still need to test the output condition that can go with every individual output. Again, as output specifications are independent from the generators, we do not need to test all generators.

From now on, we can have outputs of different length as results of the same list comprehension. These 5 tests are in section C.9.

Laziness

Laziness will be tested in details in the next chapter. For now, we just test this functionality a bit.

For this category of tests the general test function does not work properly 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. The old function would work but it does not check the laziness which is the thing we want to test.

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. As the implementation is completely independent of the rest, we can use just a subset of all the functionalities already tested. These 7 tests are in section C.10.

Bodies

Bodies can now be tested. We test them before collectors because we will use bodies for collectors. For this set of test, we also tested laziness so we needed two lists of tests. The first is for the normal tests and the second one is for the lazy tests - similar to the one used when we tested laziness. We want to test laziness again because bodies offer a new functionality so it is a potential source of new bugs for laziness. These 13 tests - 8 normal tests and 5 lazy tests - are in section C.11.

Collectors

Collectors can now be tested because bodies work. We also test that the collector can be used anywhere inside the list comprehension and outside if given as argument of a function which is one of the main goal collectors.

For this set of test, we also tested laziness so we needed two lists of tests. The first is for the normal tests and the second one is for the lazy tests - similar to the one used when we tested laziness. We want to test laziness again because collectors offer a new way to output lists so it is a potential source of new bugs. These 20 tests - 10 normal tests and 10 lazy tests - are in section C.12.

Miscellaneous

As list comprehensions must still accept all pre-existing structures, some more tests are required such as testing pattern matching in the ranger. Nested list comprehensions must also be tested as well as cells. The list comprehension returning a list must also be tested with elements directly appending to them like in `1|[A for A in 2..3]`. We also must test this without the syntactic sugar.

Tests in this paragraph are typically tests that do not go into any other category. These 18 tests are in section C.13.

Record comprehensions

As record comprehensions offer less functionalities than lists comprehensions, we have decided to test all record comprehensions in one batch of tests.

We test, multi - labeled - output, bodies, deciders and filters in this category. We begin with simple tests and increase their complexity step by step. These 22 tests are in section C.14.

4.2. Equivalences with other languages

We now have tested enough possibilities to test that what be done in other languages can also be done in Oz. 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. An Erlang list comprehension uses the same delimiters, the squared brackets. The separation between the output and the list specification is `||`. The keyword `in` of Oz becomes `<-`. Conditions and levels are delimited by comas.

Here are some examples of equivalences between Erlang and Oz as well as the expected outputs. The examples come from [1] and [15].

```
%% Erlang list comprehension
%% [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]

%% [Y || {X1, Y} <- L, X == X1]
[Y for [X Y] in LL if X == 1]#[2]
```

Python

Python is a good example of language for list comprehensions. So it is a good idea to ensure that what can be done in Python can also be done is Oz.

Nested lists comprehensions is a nice functionality of Python that has an Oz equivalent. Furthermore, Python list comprehensions can go through tuples and output a list. Using a record generator, we can do exactly the same.

The Python examples come from [8] and [21]. Here are the equivalences:

```
%% Python list comprehension
%% [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]]

%% [a for a in (1,2,3)]
[A for _:A in tuple(1 2 3)]#[1 2 3]
```

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 Oz. Their syntax is similar to the one of Erlang.

Again nested lists are used as well as the `take` operator that only takes the five first elements of the output. We do not have the exact same functionality in Oz but by using an extra layer with the right number of elements, we get the same effect. It is like a trick. The Haskell examples come from [13] and [16].

```
%% Haskell list comprehension
%% [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] % trick: _ in 1..5
#[[[1 1] [2 1]] [[1 2] [2 2]] [[1 3] [2 3]] [[1 4] [2 4]] [[1 5] [2 5]]]
```

Comparison

From what we just tested in the previous paragraph, we can see that Oz does not need to use its full syntax to copy the behavior of list comprehensions in other languages. Oz can do things with comprehensions that other languages can not do.

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]
% 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 in Rec if F > 0)
% 9
[@C for A in 1..Lim do C:=A] % C was previously declared as a cell
% 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 significant. 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 *Space_performance_X.oz*, respectively *Time_performance_X.oz*, where *X* is from 01 to 10.

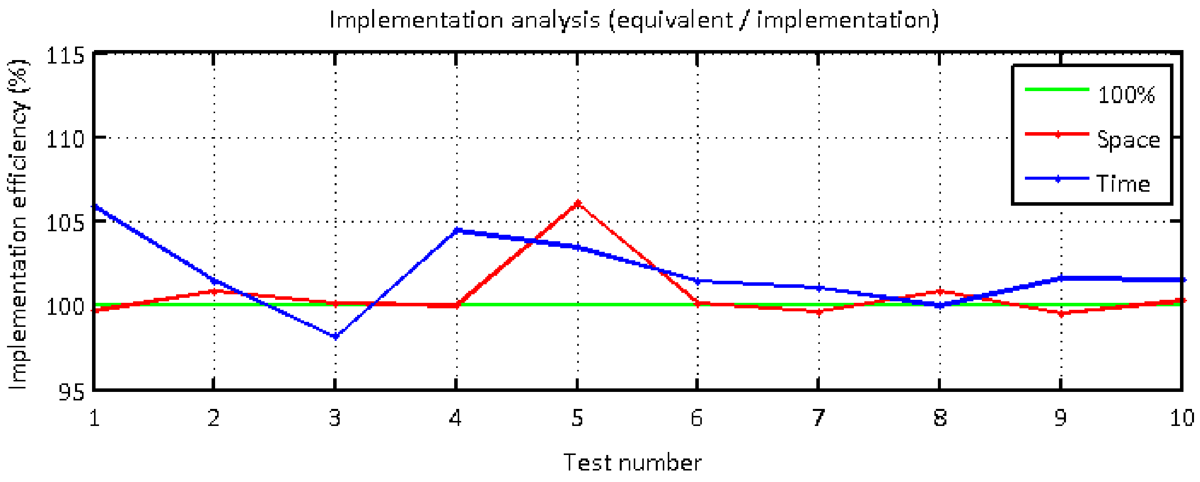


Figure 4.1: *Comparison of the performance between the implementation and its equivalent.*

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 Mozart 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.

Sort

Sorting lists using the Quicksort algorithm - see [26] - 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.

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 allows 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 in 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.

Factorial

Computing factorials can also be done using list comprehensions. Actually list comprehensions allow us to easily compute all the factorials from 0 to N, the idea is to go through all the values from 0 to N and to collect the current result in a list. At the end, each element is the factorial of its index in the list. Here is an example:

```
local N Fs Vs in
   N = 10 % compute factorials from 0 to 10
   '#'(factOf:Fs value:Vs) = [value:A factOf:I for I in 0..N
                                                  A in 1 ; A*(I+1)]
   for F in Fs V in Vs do
      {Browse {VirtualString.toAtom "Fact("#F#") = "#V}}
   end
end
```

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.

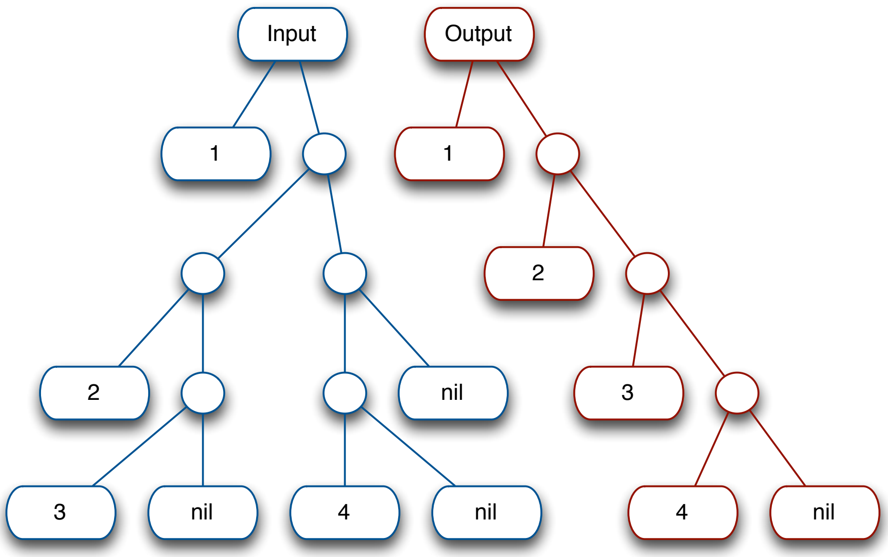


Figure 4.2: *Example of input and output for the flatten function.*

Treating the input list as a record generator, a list comprehension can 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.

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]]
```

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 one sees everything that is going on. For this to happen, it requires a dynamic way of displaying results *in extenso* a system that can handle unbound variable becoming assigned. As the Mozart browser shows such a property, we have decided to run the tests about concurrency in Mozart 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/comprehensions/Concurrency* inside the sources of the new version of Mozart.

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 [22] - an extension of Scala.

5.1. Laziness

Laziness is a concept that can appear at many places in Oz. 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 a generation without an end. One list comprehension is enough to create both streams:

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

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 *Laziness.oz* contains the complete test.

5.2. Producer – consumer

A classical example of using streams is the producer-consumer system. 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. One could compare streams to Unix pipes. One thread writes while another reads the same data. Each thread is an agent.

The producer-consumer consists in two agent. 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 agent 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 depicted in figure 5.1.

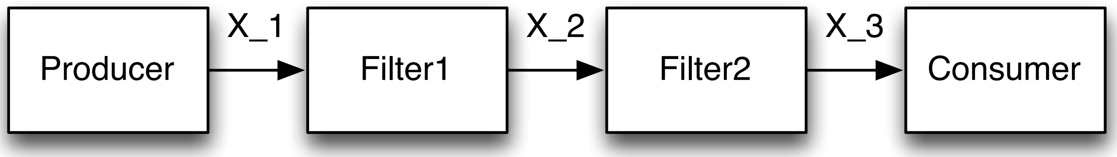


Figure 5.1: *A graphical representation of a chain of streams for a producer-consumer with 2 filters.*

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

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

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 *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. To do this we add the following body to the previous list comprehension:

```
Xs = thread [... do {Delay 1000}] end % 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 = thread [2*A for A in Xs if A mod 2 == 1] end
```

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 filter for each output.

Let us extend the previous example. 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 = thread [2*A if A mod 2 == 1 3*A if A mod 2 == 0 for A in Xs] end
```

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

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

This application can be tested using the file *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 a 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. 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 to the multi output possibility, we can do this inside one list comprehension like this:

```
Xs1#Xs2 = thread [{OS.rand} mod 2 {OS.rand} mod 2 for lazy A in 0; A+1] end
```

Again, this list comprehension never stops but is lazy. In the implementation in the file *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.

Logical functions are the functions that takes two elements as input and apply their 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.

Input	Output		
	AND	OR	XOR
0 0	0	0	0
0 1 *or* 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 functions 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 takes the operator as argument. Here is this function implemented with a list comprehension:

```
fun {GateMaker F} % returns a gate function with the given operator
   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 outputs 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 *Logic_Gates.oz*

5.5. Bounded buffer

When an agent goes lazily through a lazy stream, Oz allows bounded buffers as explained 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 the 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:

```
Xs1 = thread [A for lazy A in 0..10 do {Delay 1000}] end
Xs2 = thread [A for lazy A in 0..5  do {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 here. 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:

```
Ys1#Ys2#Ys3 = thread [A-B A*B A+B for A in Xs1:1 for lazy B in Xs2:3] end
```

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

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

Another thing we can make is to implement the bounded buffer function with a list comprehension in the official version. Recall that the function is:

```
fun {BoundedBuffer Ss N}
    fun lazy {Aux Ss End}
        case Ss of nil then nil
        [] H|T then
            H|{Aux T thread if End==nil then End else End.2 end end}
        end
    end
in
    {Aux Ss thread {List.drop Ss N} end}
end
```

One could be tempted to do it the following way:

```
fun {BoundedBuffer Ss N}
    thread [A for lazy A in Ss _ in thread {List.drop Ss N} end] end
end
```

But this is not what we want because one can only use one element when a new one is created. In other words, the buffer always has the same size. However, what we want is that the buffer has at most the given size. So we must change the previous transformation to the following:

```
fun {BoundedBuffer Ss N}
    End = thread {List.drop Ss N} end
    C = {NewCell End}
    L = {NewLock}
in
    thread [A for lazy A in Ss do thread lock L then if @C \= nil then N in
        {Exchange C _|N N} end end end] end
end
```

First, the `Exchange` aims at creating a new element. It must be inside a thread because the creation can not block the list comprehension. As the list might have an end, we can only ask the creation of a new element if there is one. That is why we need the condition. However, a new problem arises because we want the condition and the `Exchange` to be done atomically, so we have to use a lock.

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] Crockford, D., *JavaScript: The Good Parts*, O'Reilly, 2008, pp 44-45.

[5] Ford, B., *Packrat Parsing: Simple, Powerful, Lazy, Linear Time, Bryan Ford*, International Conference on Functional Programming, October 4-6, 2002, Pittsburgh.

[6] Ford, B., *Parsing Expression Grammars: A Recognition-Based Syntactic Foundation*, 31st ACM Symposium on Principles of Programming Languages, January 14-16, 2004, Venice.

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

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

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

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

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

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

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

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

Online

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

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

[17] 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`

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

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

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

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

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

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

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

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

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

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

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

Appendices

Chapter A
Generic transformations

A.1. List comprehensions

When we use three %, it means that the entire line is only used for the unofficial version.

```
%% Big local
local
    %% Collectors and cells
    {{ ForAll I in Collectors }}
        Cell_I = {NewCell _}
        proc {I X} N in {Exchange Cell_I X|N N} end
    {{ end ForAll }}
    %% pre level
    proc {PreLevel ?Result}
        %% return list when one output with no feature
        if {{ ReturnList }} then
%%%         if {{ Bounded_Buffer }} then
%%%             %% create buffers
%%%             local
%%%                 Range1At1 = List_1
%%%                 End1At1 = thread {List.drop Range1At1 Buffer} end
%%%             in
%%%                 %% call next level with buffers
%%%                 {Level1 {{ Initiators_For_Next_Level }} '#'(1:Result)}
%%%             end
%%%         else %% no buffers
                {Level1 {{ Initiators_For_Next_Level }} '#'(1:Result)}
%%%         end
        else %% return record
            %% create the tuple of outputs
            local
                {{ Next_1 ... Next_N }}
            in
                Result = {{ '#'(field1:Next1
                            ... fieldCollect1:@Cell1 ... fieldN:NextN) }}
%%%             if {{ Bounded_Buffer }} then
%%%                    %% create buffers
%%%                    local
%%%                        {{ ForAll I with Bounded Buffer }}
%%%                            RangeIAt1 = List_I
%%%                            EndIAt1 = thread
%%%                                        {List.drop RangeIAt1 Buffer}
%%%                                      end
%%%                        {{ End ForAll }}
%%%                    in
%%%                        %% call next level with buffers
%%%                        {Level1 {{ Initiators_For_Next_Level }}
%%%                         '#'(field1:Next1 ... fieldN:NextN)}
%%%                    end
%%%                else %% no buffers
                       %% call level 1 with its initiators and with the tuple
                       {Level1 {{ Initiators_For_Next_Level }}
                        '#'(field1:Next1 ... fieldN:NextN)}
%%%                end
            end
        end
    end
    %% previous level number: X (if exists)
    %% current  level number: Y
    %% next     level number: Z (if exists)
    proc {LevelY {{ This_Level_Arguments }} {{ Previous_Levels_Arguments }}
          {{ Previous_List_Ranges_And_Stacks }} ?Result}
        %% handle lazy if needed
        if {{ Is_Lazy }} then
            if {{ Multi_Output }}
                %% wait for need of any output
                local LazyVar in
                    {{ Forall I in Fields_Name }}
                        thread
%%%                         if {{ I is Collector }} then
%%%                             {WaitNeeded @Cell_I}
%%%                         else
                                {WaitNeeded Result.I}
%%%                         end
                            LazyVar = unit
                        end
                    {{ end Forall }}
                    %% LazyVar is assigned (to unit)
                    %% as soon as any output is needed
                    {Wait LazyVar}
                end
            else
                % one output, so just wait for it to be needed
%%%             if {{ I is Collector }} then
%%%                 {WaitNeeded @Cell_1}
%%%             else
                    {WaitNeeded Result.{{ Fields_Name.1 }}}
%%%             end
            end
        end % end of handle laziness
        % test if no more iterations
        if {{ Ranges_Conditions_For_This_Level }} then
            % still at least one iteration
            local
                % local needed iff generatorList
                % or forRecord in ranges of this level
                {{ Forall generatorList GL }}
                    {{ Range_For_GL }} = {{ GL_Argument }}.1
                {{ end Forall }}
                {{ Forall forRecord FR }}
                    {{ FR_Feature }} = {{ FR_Features }}.1
                    {{ FR_Range }} = {{ FR_Record }}.{{ FR_Feature }}
                {{ end Forall }}
            in
                % test condition given by user
                if {{ This_Level_Condition }} then
                    if {{ Last_Level }} then
                        % last level, add elements
                        % add elements for each output
                        local
                            {{ Next_1 ... Next_N }}
                        in
                            {{ Forall I in Fields_Name }}
                                % append to result iff optional condition
                                % given by user for output I is fulfilled
                                % no condition given is equivalent to true
                                Result.I = if {{ Condition.I }} then
                                               Expression.I|Next_I
                                           else Next_I
                                           end
                            {{ end Forall }}
                            % call next iteration of this level
                            {LevelY
%%%                            if {{ Bounded_Buffer }} then
%%%                               {{ Next_Iter_LevelY_Buff }}
%%%                           else
                                  {{ Next_Iter_LevelY }}
%%%                           end
                              {{ Previous_Levels_Arguments }}
                              {{ Previous_List_Ranges }}
                              {{ '#'(field1:Next1 ... fieldN:NextN) }}
                             }
                        end
                    else
                        %% not last level, call next level
                        {LevelZ
                         %% handle buffers for next level
%%%                      if {{ Bounded_Buffer }} then
%%%                          local
%%%                              {{ ForAll I with Bounded Buffer }}
%%%                                  RangeIAtZ = List_I
%%%                                  EndIAtZ = thread {List.drop
%%%                                                 RangeIAtZ Buffer}
%%%                                            end
%%%                              {{ End ForAll }}
%%%                          in
%%%                              {{ Initiators_For_Next_Level }}
%%%                          end
%%%                      else
                             {{ Initiators_For_Next_Level }}
%%%                      end
                         {{ This_Level_Ranges }} % may contain buffers
                         {{ Previous_Levels_Arguments }}
                         {{ List_Arguments }}
                         {{ Previous_List_Ranges }}
                         Result
                        }
                    end
                else
                    %% this level condition not fulfilled
                    %% call next iteration of this level
                    {LevelY
%%%                  if {{ Bounded_Buffer }} then
%%%                      {{ Next_Iter_LevelY_Buff }}
%%%                  else
                         {{ Next_Iter_LevelY }}
%%%                  end
                     {{ Previous_Levels_Arguments }}
```

```
164                      {{ Previous_List_Ranges }}
165                      Result
166                     }
167                 end
168             end
169         else
170             %% no more iterations for this level, call previous one
171             if {{ First_Level }} then
172                 %% no previous level, end output-s by appending nil
173                 if {{ Multi_Output }} then
174                     {{ Forall I in Fields_Name }}
175 %%%                     if {{ Output I is Collector }} then
176 %%%                         {Exchange Cell_I nil _}
177 %%%                     else
178                             Result.I = nil
179 %%%                     end
180                     {{ end Forall }}
181                 else
182 %%%                 if {{ Output 1 is Collector }} then
183 %%%                     {Exchange Cell_1 nil _}
184 %%%                 else
185                         Result.1 = nil
186 %%%                 end
187                 end
188             else
189                 %% call previous level X
190                 %% same as lines 157 to 166 of previous level X
191                 {{ Call_Previous_Level_With_Next_Iteration }}
192             end
193         end
194     end
195 in
196     %% actually call the cascade of procedures
197     {PreLevel}
198 end
199
200 %%% Unofficial, for bounded buffers only
201 fun {{ Next_Iter_LevelY_Buff }}
202     case Ranges
203     of nil then nil
204     [] H|T then
205         case H
206         of RangeIAtX#EndIAtX then %% buffer is here
207             (RangeIAtX.2#thread if EndIAtX == nil then EndIAtX
                                        else EndIAtX.2
                                        end)|{... T}
          else H|{... T}
          end
     end
end
```

A.2. Record comprehensions

```
%% Big local
local
     %% pre level
     proc {PreLevel ?Result}
          %% return one record when one output with no feature
          if {{ ReturnOneRecord }} then
               {Level {{ Initiator_Record }} '#'(1:Result)}
          else %% return record of records
               %% create the tuple of outputs
               local
                    {{ Next_1 ... Next_N }}
               in
                    Result = {{ '#'(field1:Next1 ... fieldN:NextN) }}
                    %% call level with initiators and with the tuple
                    {Level {{ Initiator_Record }}
                     '#'(field1:Next1 ... fieldN:NextN)}
               end
          end
     end
     %% Level
     proc {Level Rec ?Result}
          local
               Ari = {Arity Rec}
               Lbl = {Label Rec}
               AriFull
               AriBool
          in
               {For1 Ari Rec AriFull AriBool}
               {{ ForAll Features F }}
                    Result.F = {Record.make Lbl AriFull}
               {{ end ForAll }}
```

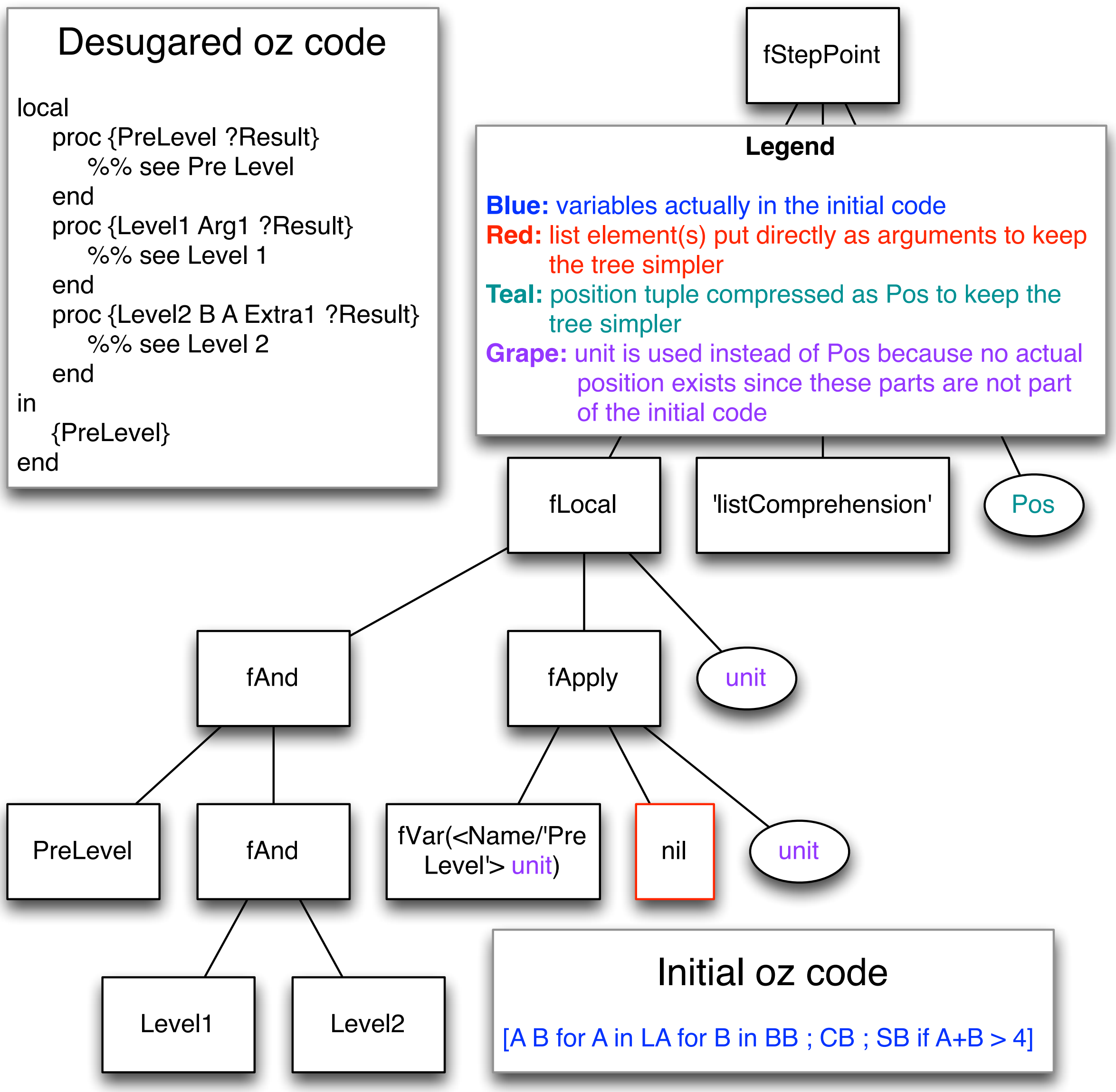
```
            {For2 Rec AriFull AriBool Result}
        end
    end
    %% For 1
    proc {For1 Ari Rec ?AriFull ?AriBool}
        if Ari \= nil then
            local
                {{ Feature Given By User Or New One }} = Ari.1
                {{ Value Given By User Or New One }} =
                    Rec.{{ Feature Given By User Or New One }}
                NextFull
                NextBool
            in
                AriBool = {{ Filter }}|NextBool
                AriFull = if AriBool.1 then
                              {{ Feature Given By User Or New One }}
                               |NextFull
                          else NextFull
                          end
                {For1 Ari.2 Rec NextFull NextBool}
            end
        else
            AriFull = nil
            AriBool = nil
        end
    end
    %% For 2
    proc {For2 Rec AriFull AriBool ?Result}
        if AriFull \= nil then
            if AriBool.1 then
                local
                    {{ Feature Given By User Or New One }} = AriFull.1
                    {{ Value Given By User Or New One }} =
                        Rec.{{ Feature Given By User Or New One }}
                in
                    if {IsRecord {{ Value Given By User Or New One }}}
andthen
                       {Arity {{ Value Given By User Or New One }}} \= nil
andthen
                       {{ Decider }} then
                        {Level {{ Value Given By User Or New One }}
                         '#'( {{ ForAll Feature F }}
                                    F:Result.F.{{ Feature Given By User
                                                              Or New One }}
```

```
                                            {{ end ForAll }}
                                         )}
                              else
                                 {{ Body }}
                                 {{ ForAll Feature F }}
                                     Result.F.{{ Feature Given By User Or New One }}
                                        = {{ Output Expression F }}
                                 {{ end ForAll }}
                              end
                          end
                          {For2 Rec AriFull.2 AriBool.2 Result}
                      else
                          {For2 Rec AriFull   AriBool.2 Result}
                      end
                  end
              end
          end
```

Chapter B

Transformed AST example

B.1. General AST


fStepPoint
Legend
Blue: variables actually in the initial code
Red: list element(s) put directly as arguments to keep the tree simpler
Teal: position tuple compressed as Pos to keep the tree simpler
Grape: unit is used instead of Pos because no actual position exists since these parts are not part of the initial code
fLocal
'listComprehension'
Pos
fAnd
fApply
unit
PreLevel
fAnd
fVar(<Name/'PreLevel'> unit)
nil
unit
Level1
Level2
Initial oz code

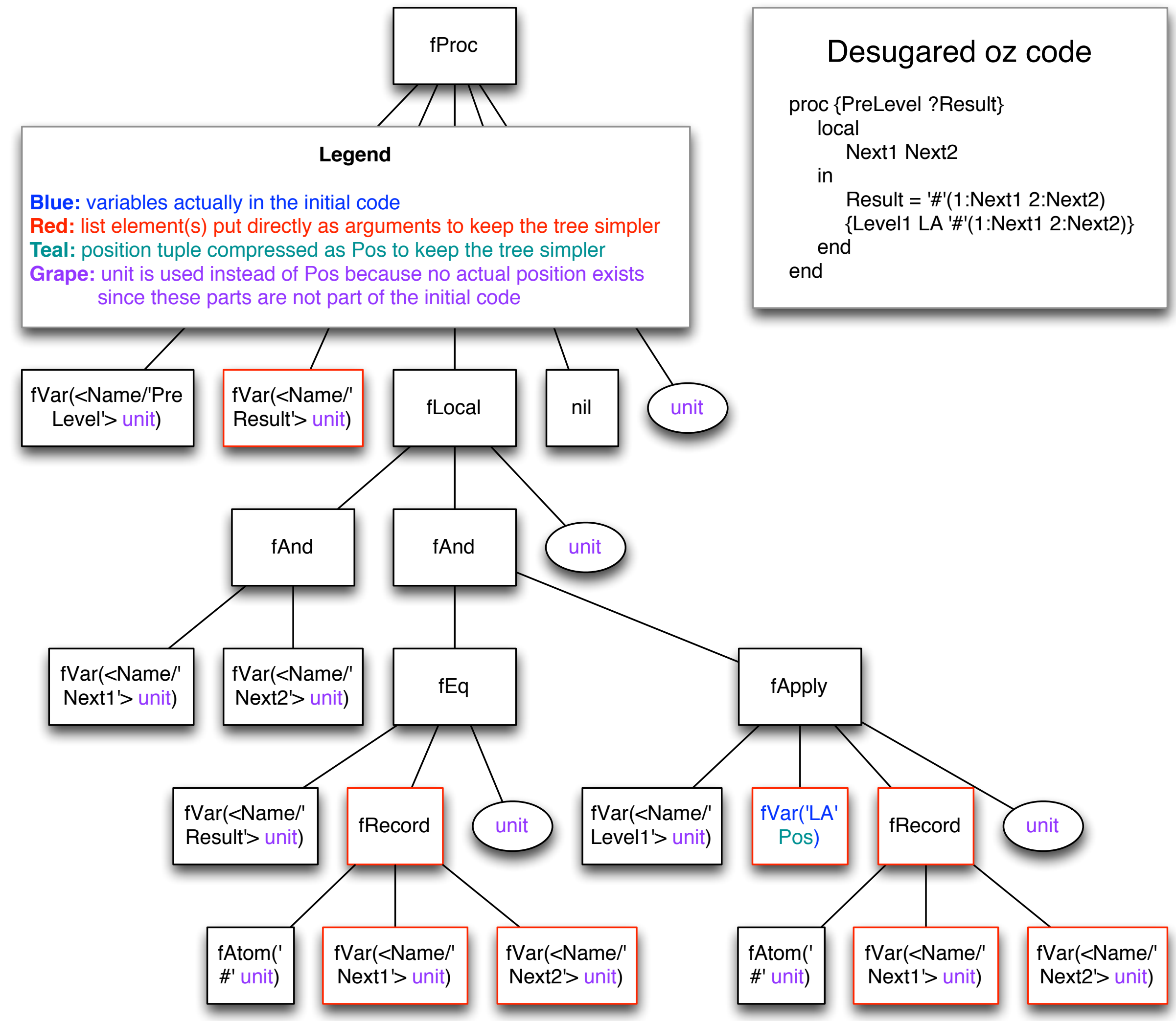
Desugared oz code

```
local
    proc {PreLevel ?Result}
        %% see Pre Level
    end
    proc {Level1 Arg1 ?Result}
        %% see Level 1
    end
    proc {Level2 B A Extra1 ?Result}
        %% see Level 2
    end
in
    {PreLevel}
end
```

```
[A B for A in LA for B in BB ; CB ; SB if A+B > 4]
```

Figure B.1: *General transformation.*

B.2. Pre-level


Legend

Blue: variables actually in the initial code
Red: list element(s) put directly as arguments to keep the tree simpler
Teal: position tuple compressed as Pos to keep the tree simpler
Grape: unit is used instead of Pos because no actual position exists since these parts are not part of the initial code

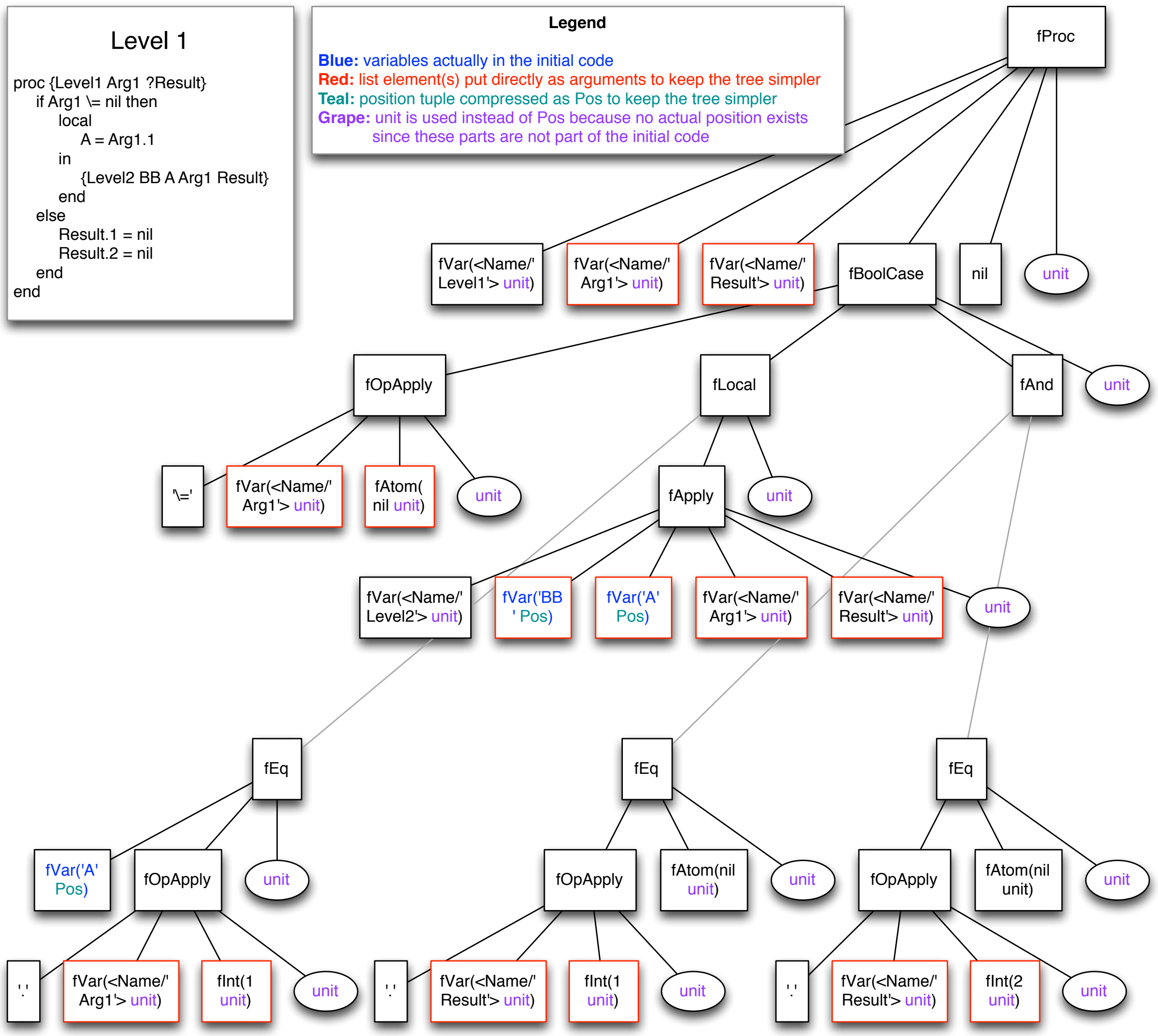
Desugared oz code

```
proc {PreLevel ?Result}
   local
      Next1 Next2
   in
      Result = '#'(1:Next1 2:Next2)
      {Level1 LA '#'(1:Next1 2:Next2)}
   end
end
```

fProc
fVar(<Name/'PreLevel'> unit)
fVar(<Name/'Result'> unit)
fLocal
nil
unit
fAnd
fAnd
unit
fVar(<Name/'Next1'> unit)
fVar(<Name/'Next2'> unit)
fEq
fApply
fVar(<Name/'Result'> unit)
fRecord
unit
fVar(<Name/'Level1'> unit)
fVar('LA' Pos)
fRecord
unit
fAtom('#' unit)
fVar(<Name/'Next1'> unit)
fVar(<Name/'Next2'> unit)
fAtom('#' unit)
fVar(<Name/'Next1'> unit)
fVar(<Name/'Next2'> unit)

Figure B.2: *Pre-level.*

B.3. Level 1



Level 1

```
proc {Level1 Arg1 ?Result}
   if Arg1 \= nil then
      local
         A = Arg1.1
      in
         {Level2 BB A Arg1 Result}
      end
   else
      Result.1 = nil
      Result.2 = nil
   end
end
```

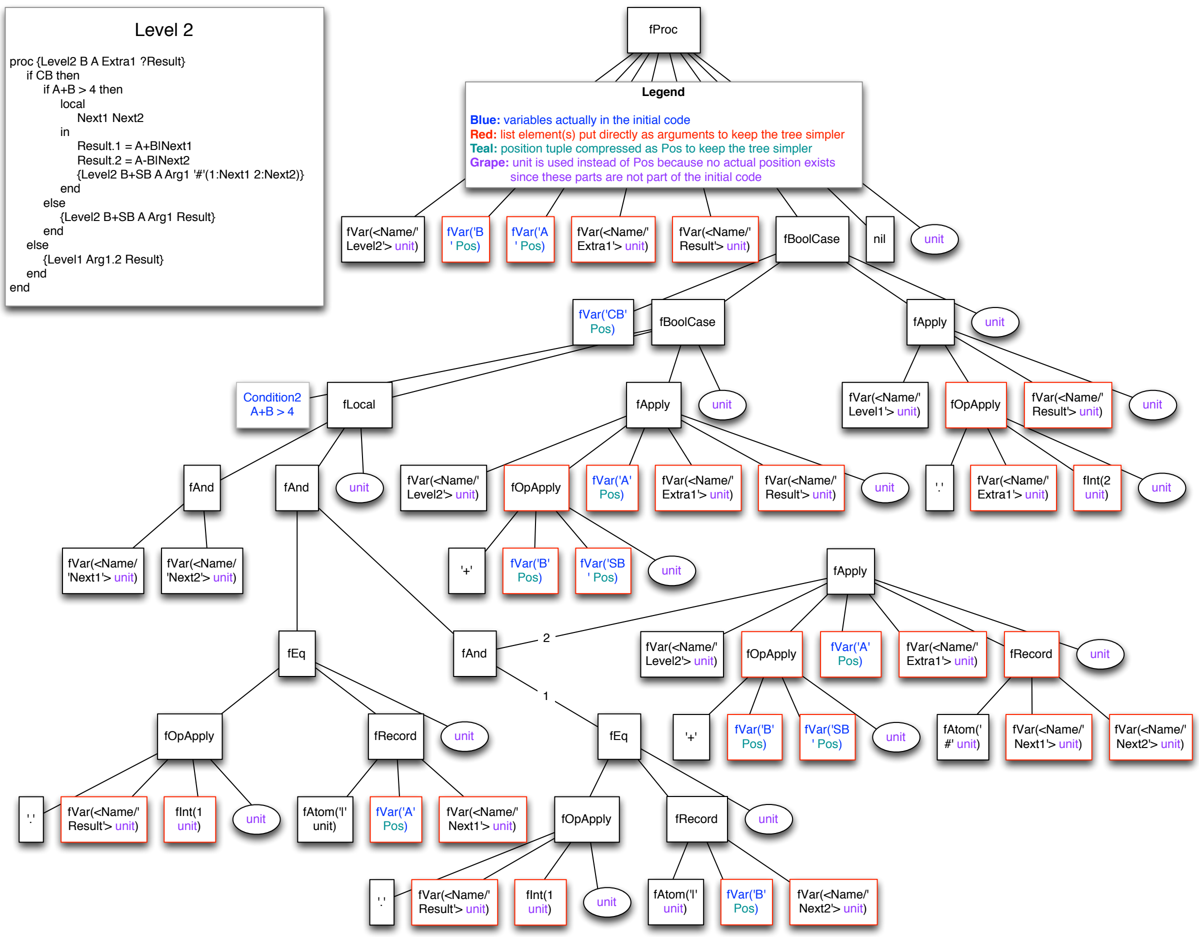
Legend

Blue: variables actually in the initial code
Red: list element(s) put directly as arguments to keep the tree simpler
Teal: position tuple compressed as Pos to keep the tree simpler
Grape: unit is used instead of Pos because no actual position exists since these parts are not part of the initial code

fProc
fVar(<Name/' Level1'> unit)
fVar(<Name/' Arg1'> unit)
fVar(<Name/' Result'> unit)
fBoolCase
nil
unit
fOpApply
fLocal
fAnd
unit
'\='
fVar(<Name/' Arg1'> unit)
fAtom(nil unit)
unit
fApply
unit
fVar(<Name/' Level2'> unit)
fVar('BB ' Pos)
fVar('A' Pos)
fVar(<Name/' Arg1'> unit)
fVar(<Name/' Result'> unit)
unit
fEq
fEq
fEq
fVar('A' Pos)
fOpApply
unit
fOpApply
fAtom(nil unit)
unit
fOpApply
fAtom(nil unit)
unit
'.'
fVar(<Name/' Arg1'> unit)
fInt(1 unit)
unit
'.'
fVar(<Name/' Result'> unit)
fInt(1 unit)
unit
'.'
fVar(<Name/' Result'> unit)
fInt(2 unit)
unit

Figure B.3: *Level 1.*

B.4. Level 2



Level 2

```
proc {Level2 B A Extra1 ?Result}
   if CB then
      if A+B > 4 then
         local
            Next1 Next2
         in
            Result.1 = A+BINext1
            Result.2 = A-BINext2
            {Level2 B+SB A Arg1 '#'(1:Next1 2:Next2)}
         end
      else
         {Level2 B+SB A Arg1 Result}
      end
   else
      {Level1 Arg1.2 Result}
   end
end
```

Legend

Blue: variables actually in the initial code
Red: list element(s) put directly as arguments to keep the tree simpler
Teal: position tuple compressed as Pos to keep the tree simpler
Grape: unit is used instead of Pos because no actual position exists since these parts are not part of the initial code

Figure B.4: *Level 2.*

Chapter C

Tests

C.1. One level – one layer

```
% [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]
```

C.2. One level – multi layer

```
% [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]
```

C.3. Function generators

```
% [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]
```

C.4. Multi level – one layer

```
% [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]
```

C.5. Multi level – multi layer

```
% [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]]
```

C.6. Record generators

```
% [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]
```

C.7. Multi output

```
% [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])
```

C.8. Labeled outputs

```
% [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])
```

C.9. Output conditions

```
% [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]
```

C.10. Laziness

```
% [listComprehension]#[expectedList]#batchSize
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
```

C.11. Bodies

```
% [listComprehension]#[expectedList]
C = {NewCell _}
Tests = [
    [@C for A in 1..2 do C:=A]
    #[1 2]

    [@C for A in 1..2 do C:=A C:=@C*A]
    #[1 4]

    [@C for A in 1..2 if A > 1 do C:=A]
    #[2]

    [@C if @C > 1 for A in 1..2 do C:=A]
    #[2]
    [@C-1 for A in 1..2 do C:=A+1]
    #[1 2]

    [@C for _ in 1..1 A from fun{$} 1 end do C:=A]
    #[1]

    [@C for _:A in r(r(r(1) r(2))) do C:=A]
    #[1 2]

    [@C for A in 1..2 for B in 1..2 do C:=A+B]
    #[2 3 3 4]
]
L1 = thread [@C for lazy A in 1..2 do C:=A] end
L2 = thread [@C for lazy A in 1..2 do C:=A C:=@C*A] end
L3 = thread [@C for lazy A in 1..2 if A > 1 do C:=A] end
L4 = thread [@C if @C > 1 for lazy A in 1..2 do C:=A] end
L5 = thread [@C for lazy A in 1..2 for B in 1..2 do C:=A+B] end
TestsLazy = [
    L1#[1 2]#1
    L2#[1 4]#1
    L3#[2]#1
    L4#[2]#1
    L5#[2 3 3 4]#2
]
```

C.12. Collectors

```
% [listComprehension]#[expectedList]
Test = [
    [c:collect:C for A in 1..2 do {C A}]
    #'#'(c:[1 2])

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

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

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

    [c:collect:C for A in 1..2 for B in 3..4 do {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 do {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 do {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} do {C A}]
    #'#'(c:[1 1 1])

    [c:collect:C for A in 1..3 if {Ext C true} do {C A}]
    #'#'(c:[1 1 1 2 1 3])
]
L1 = thread [c:collect:C for lazy A in 1..2 do {C A}] end
L2 = thread [c:collect:C for lazy A in 1..2 do {C A}{C A+1}] end
L3a = thread [1:collect:C1 2:collect:C2 for lazy A in 1..2 do {C1 A}{C2 A
   +1}] end
L3b = thread [1:collect:C1 2:collect:C2 for lazy A in 1..2 do {C1 A}{C2 A
   +1}] end
L4 = thread [c:collect:C for lazy A in 1..2 for B in 3..4 do {C A+B}] end
L5 = thread [c:collect:C for A in 1..2 for lazy B in 3..4 do {C A+B}] end
L6a = thread [1:collect:C1 2:collect:C2 for lazy A in 1..2 for B in 3..4 do
    {C1 A}{C2 B}] end
L6b = thread [1:collect:C1 2:collect:C2 for lazy A in 1..2 for B in 3..4 do
    {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 do {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 do {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
]
```

C.13. Miscellaneous

```
% [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]
```

C.14. Record comprehensions

```
% [recordComprehension]#[expectedRecord]
(A for A in Rec)
#Rec

(A for _:A in Rec)
#Rec

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

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

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

(1 for _ in r(1 2 rr(3 rrr(4)) 5))
#r(1 1 rr(1 rrr(1)) 1)

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

(1 a:2 for A in 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 in r(1 2 rr(3 rrr(4)) 5))
#r(2 3 rr(4 rrr(5)) 6)

(A+1 A-1 for A in 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 in r(1 2 rr(3 rrr(4)) 5) of {Bool A})
#r(1 2 rr(3 rrr(4)) 5)

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

(A for F:A in 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 in 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 in [1 2 3])
#[1#1 1#2 1#3]

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

```
(A for A in Tree)
#Tree

(A+1 for A in Tree)
#tree(tree(leaf(2) leaf(3)) leaf(4))

(F#A for F:A in Tree)
#tree(tree(leaf(1#1) leaf(1#2)) leaf(1#3))

(F#A for F:A in Tree of F == 1)
#tree(tree(leaf(1#1) 2#leaf(2)) 2#leaf(3))

(A for F:A in Tree if F == 1)
#tree(tree(leaf(1)))

(A for A in Tree do Body = unit) % we check if Body is bound later
#Tree

(if F==key then N+1 else N end for F:N in OBTree of F == left orelse F ==
    right)
#obtree(key:2 left:leaf right:obtree(key:3 left:leaf right:leaf))
```

Chapter D

Tutorial on comprehensions

This paper aims at explaining the complete syntax of comprehensions in Mozart2. There are two kinds of comprehensions implemented for the moment: list and record comprehensions. There are two main differences between the two later. First, list comprehensions return list-s while record comprehensions return record-s. Second, there are more functionalities available with list comprehensions.

D.1. List comprehensions

A list comprehension allows the creation of lists in an easy and flexible way. In this section, we explain their complete syntax incrementally.

A example of a very simple list comprehension is:

```
[A+1 for A in 1..5] %= [2 3 4 5 6]
```

Let us analyze this comprehension. The first thing that interests us is the `for A in 1..5`. This means that the list comprehensions creates a new variable named `A` and that this variable will take values from 1 to 5. The output is specified by the first part *in extenso* `A+1`. It means that we just add the current value of `A` incremented by 1 to the output list. This expression can be anything.

We will start from this simple syntax and, step by step, add functionalities. All functionalities are compatible with each other, even if examples do not cover all combinations of functionalities.

Generator, ranger and layer

A *generator* is the specification of a structure on which the comprehension iterates. In our example, `1..5` is a generator, it iterates from 1 to 5.

The *ranger* is the structure that is assigned to the current value of its corresponding generator at each iteration. In our example, `A` is the ranger of the generator we just saw.

A *layer* is the aggregation of a ranger and its generator.

There are five kinds of generators. Here are the descriptions of the first four - examples follow:

Integer generator: `in Low..High ; Step`
: Goes from integers `Low` to `High` by integer step of `Step`. The step is optional, default is 1. `Low` must be smaller or equal to `High`.

C-style generator: `in ( Init ; Condition ; Next )`
: Initiate its ranger to `Init`, iterates while `Condition` evaluates to true and updates the ranger to `Next` at the end of each iteration. The parenthesis are optional. The condition is optional, default is `true` - this implies that the generator never stops iterating by itself.

List generator: `in List`
: `List` can be a variable containing a list or the definition of new list. The generator iterates over all the elements of the list in the same order as they appear in the list. The list can be a stream.

Function generator: `from Function`
: `Function` can be a variable containing a function or the definition of a unnamed function. In both cases, the function does not take any arguments. The generator never stops. At each iteration, the current value of the generator is the result of calling `Function`.

```
% Integer generator
[A for A in 1..5 ; 2] % [1 3 5]
[A for A in 1..5 ; 1] % [1 2 3 4 5]
[A for A in 1..5]     % [1 2 3 4 5]
% C-style generator
[A for A in 1 ; A<6 ; A+1]      % [1 2 3 4 5]
[A for A in ( 1 ; A<6 ; A+1 )] % [1 2 3 4 5]
[A for A in 1 ; A+1]            % [1 2 3 4 5 ...] (never stops)
% List generator
L = [1 2 3]
[A for A in L]          % [1 2 3]
[A for A in [1 2 3]]    % [1 2 3]
[A for A in [1 [2] 3]] % [1 [2] 3]
% Function generator
Fun = fun{$} 1 end
[A for A from Fun]            % [1 1 1 1 ... ] (never stops)
[A for A from fun{$} 1 end] % [1 1 1 1 ... ] (never stops)
```

The fifth kind of generator, record generators, is a bit more complicated. It is comparable to list generators but record generators are more general. A record generator has the following form:

```
F:V in Record of Function
```

As for lists, `Record` can be a variable containing a record or the definition of a new record. If it is a variable, then we have no way to know in advance what kind of generator it is. That is why we need a way to differentiate them. The ranger contains a feature `F` and a value `V`. This is how we differentiate record from list generators. Note that this is the reason why the function generator uses the `from` keyword and not `in`.

Iterating over a record with a feature and a value means that at every iteration, the feature is assigned to the current feature and the value is assigned to the current value. Such rangers allow having all the field information available.

The iteration goes through all fields of the record in the same order as its arity. When the record contains records, it forms a tree as in figure D.1. In that case, only the leaves of the tree are considered in the iterations. The tree is traversed using the depth-first mode.

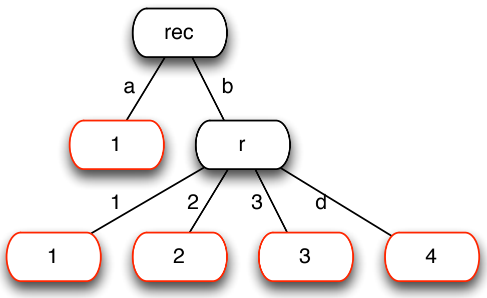


Figure D.1: *Representation of the tree formed by the record rec(a:1 b:r(1 2 3 d:4)).*

We did not explained what is `Function` yet. It is a function taking two arguments that returns a boolean. It is called by the list comprehension every time it encounters a nested record inside the given record. The arguments are respectively the feature and the value of the field containing the nested record. When the result is true, the nested record is considered as such and we traverse it in depth-first mode. On the other hand, when the result is false then the nested record is treated as a leaf.

Here are some examples of list comprehensions with record generators:

```
R = r(a:1 b:r(1 2))
Fun = fun{$ F V} F \= b end
[A for _:A in 1#2#3]                                        % [1 2 3]
[A for _:A in R]                                            % [1 1 2]
[F for F:_ in R]                                            % [a 1 2]
[F#A for F:A in R]                                          % [a#1 1#1 2#2]
[F#A for F:A in r(a:1 b:r(1 2)) of fun{$ F V} F \= b end]  % [a#1 b#r(1 2)]
```

```
[F#A for F:A in R of Fun]                                    % [a#1 b#r(1 2)]
```

Multi layer

Now that all kinds of generators have been specified, we can move on to specifying several layers. One specifies several layers as follows:

```
[... for Layer_1 ... Layer_N]
% Example
[[A B] for A in 1..3 B in [a b]] % [[1 a] [2 b]]
```

Layers are traversed simultaneously. It means that a layer stops iterating when its iteration has reached its end or once at least one of its neighbor layers stops iterating. So the *smallest* generator - with the smallest number of iterations - decides when to stop iterating. In our example above, `B` is the range of the smallest generator and will take the values `a` then `b`. `A` will follow B and take the values 1 then 2, never 3.

Thanks to this functionality, function generators can be stopped as well as C-style generators without a condition.

Multi level

A *level* is the aggregation of one or more layers traversed simultaneously - inside the same `for` - together with an optional condition. A list comprehension contains at least one level.

Levels are like nested loops. It means that for each element of the main loop, the nested loop will run a complete iteration. Here is an example with two levels:

```
L = [4 7]
[[A 2*B] for A in L for B in 3..5] % [[4 6] [4 8] [7 6] [7 8]]
```

In addition to all its layers, a level can have a condition. The latter is always at the end of the level and is delimited by the keyword `if`. The condition is a boolean expression. When the condition evaluates to true, the iteration is accepted and the next level is called if it exists, otherwise an element is added to the output list. When the condition evaluates to false, the current iteration of the current level is skipped. Here are some examples:

```
L = [1 2 4 7]
[[A 2*B] for A in L if A>6 for B in 3..5] % [[7 6] [7 8] [7 10]]
[[A 2*B] for A in L for B in 3..5 if A>B] % [[4 6] [7 6] [7 8] [7 10]]
[[A 2*B] for A in L for B in 3..5 if A>B andthen B<5] % [[4 6] [7 6] [7 8]]
```

Laziness

A level can be specified as lazy. It is done by specified a lazy flag as any other layer - the flag is considered as a layer:

```
declare L in
thread L = [A for lazy A in 1..5] end
% L = _
{List.drop L 1 _}
% L = 1|_
{List.drop L 3 _}
% L = 1|2|3|_
```

As levels are specified as lazy, one must choose wisely which level-s must be lazy as in the following examples:

```
declare L in
thread L = [A#B for lazy A in 1..5 for B in [a b]] end
% L = _
{List.drop L 1 _}
% L = 1#a|1#b|_
{List.drop L 2 _}
% L = 1#a|1#b|_
{List.drop L 3 _ }
% L = 1#a|1#b|2#a|2#b|_
```

As the first level is declared as lazy, this level waits for an element to be needed to launch the iteration over its next element. When a value is needed, the first level allows one iteration to execute. This means that the second level can traverse all of its iterations because it is not lazy. That is why in the above example, elements are created by two - the number of iterations of the second level.

Bounded buffers

When one uses a list generator, a new functionality appears. It is called the bounded buffer[1]. It specifies the length of the bounded buffer to keep for the corresponding list. The syntax is the list followed by the size of the buffer separated by a colon. Here is a complete example:

[1]For more information on why using bounded buffers, please consult Van Roy, P., & Haridi, S., *Concepts, Techniques, and Models of Computer Programming*, MIT Press, 2004. We can also consult https://github.com/francoisfonteyn/thesis_public/blob/master/Thesis.pdf

```
declare Xs Ys Gen in
fun lazy {Gen I N}
   if I =< N then I|{Gen I+1 N}
   else nil
   end
end
thread Xs = {Gen 1 10} end
thread Ys = [A for lazy A in Xs:3] end
% Xs = 1|2|3|_        Ys = _
{List.drop Ys 1 _}
% Xs = 1|2|3|4|_      Ys = 1|_
{List.drop Ys 2 _}
% Xs = 1|2|3|4|5|_    Ys = 1|2|_
```

A representation of the bounded buffer in the example above is in figure D.2.

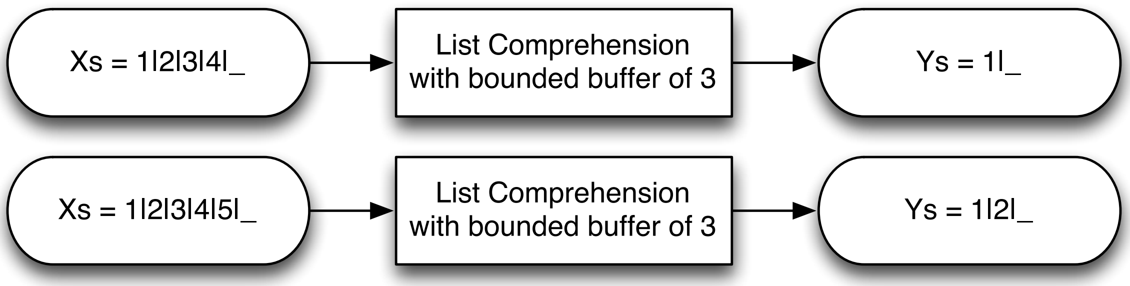

Xs = 1|2|3|4|_
List Comprehension with bounded buffer of 3
Ys = 1|_
Xs = 1|2|3|4|5|_
List Comprehension with bounded buffer of 3
Ys = 1|2|_

Figure D.2: *Representation of a bounded buffer.*

The existence of bounded buffers is one of the reasons to differentiate list and record generators - recall that list are special records.

Multi output

Up to now, we limited ourselves to one output list. List comprehensions allow the result to have more than one output lists. All these output lists are put inside a record. One just specifies one expression by output list as follows:

```
[a b for _ in 1..2] % [a a]#[b b]
```

When features are not specified, they are implicitly created to be the smallest unused integers from one. The label is always '#' so that the record can also be equivalent to a tuple declared using its syntactic sugar.

To specify the feature, one just needs to use a colon to separate the feature from the expression. Here are two examples:

```
[a:a b:b for _ in 1..2] % '#'(a:[a a] b:[b b])
[a 1:b for _ in 1..2]   % [b b]#[a a]
```

In addition, each output can have a condition called the *output condition*. This condition allows filtering the elements of the corresponding list. Thanks to this functionality, output lists can have different sizes. Here is an example:

```
[smallerEquals:A if A =< 3 bigger:A if A > 3 for A in [3 4 2 8 9 5 7 1 6]]
% '#'(smallerEquals:[3 2 1] bigger:[4 8 9 5 7 6])
```

The output is a list only when there is one output expression without a feature, otherwise, the result is a record of list-s. So the two following examples are different:

```
[1:A if A < 3 for A in 1..5] % '#'(1:[1 2])
[A   if A < 3 for A in 1..5] % [1 2]
```

Bodies

As a list comprehensions has an arbitrary number of levels, it is comparable to nested loops. This comparison has driven us to allow list comprehensions to have a body. The *body* of a list comprehensions is a statement that will be executed - just - before each time the comprehensions adds an element to each output - or when the comprehension tries appending, even if output conditions are false. In the comparison with nested loops, the body is the statement inside the *deepest* loop of the nesting. Here is an example:

```
[A for A in 1..5 do {Delay 1000}] % [1 2 3 4 5] after 5 seconds
```

Collectors

The last functionality of list comprehensions is collectors. As in for loops, a *collector* is a procedure that appends its only argument to the end of its corresponding list. So when specified, a collector is assigned to a procedure that appends an element that a list. When one wants to use a collector, he must specify it as an output with a mandatory feature, the flag `collect` and the variable that will be assigned to the procedure, all separated by colons. Here are some examples:

```
[c:collect:C for A in 1..5 do {C A}] % '#'(c:[1 2 3 4 5])
[c:collect:C for A in 1..5 do {C A}{C A+1}] % '#'(c:[1 2 2 3 3 4 4 5 5 6])
[1:collect:C1 2:collect:C2 for A in 1..5 do {C1 A-1}{C2 A+1}]
% [0 1 2 3 4]#[2 3 4 5 6]
```

Collectors can also be passed as arguments of external procedures. Note that the lists corresponding to collectors are closed at the end of the list comprehension.

D.2. Record comprehensions

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 are restricted 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. 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 not to use a function.

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. Collectors are specific to lists so they are not implemented in record comprehensions.

As record comprehensions use a syntax very similar to the one of list comprehensions, we need to differentiate them. This is done by using parenthesis instead of square brackets to delimit the comprehension.

Collectors are specific to list comprehensions but bodies are not.

Here are some examples:

```
(A   for A   in r(a:2 b:3 1))                   % r(1:1 a:2 b:3)
(A+1 for A   in r(a:2 b:3 1))                   % r(1:2 a:3 b:4)
(@C  for A   in r(a:2 b:3 1) do C := A)         % r(1:2 a:3 b:4)
(F   for F:A in r(a b c))                       % r(1 2 3)
(F+1 for F:A in r(a b c))                       % r(2 3 4)
```

```
(F#A for F:A in r(a:1 b:r(1 2)))                  % r(a:a#1 b:r(1#1 2#2))
(F#A for F:A in r(a:1 b:r(1 2)) of F == a)      % r(a:a#1 b:b#r(1 2))
(F#A for F:A in r(a:1 b:r(1 2)) if {IsRecord A} orelse A>1)
    % r(b:r(2:2#2))
(F#A for F:A in r(a:1 b:r(1 2)) if {IsRecord A} orelse A>1 of F == a)
    % r(b:b#r(1 2))
```

A more complex example where we handle a binary tree follows:

```
declare Rec = obtree(key:1 left:leaf right:obtree(key:2 left:leaf right:
    leaf))
{Browse (if F==key then N+1 else N end for F:N in Rec of F==left orelse F==
    right)}
% browses
obtree(key:2 left:leaf right:obtree(key:3 left:leaf right:leaf))
```