Università degli Studi di Torino

Department of Computer Science

Bachelor's degree in Computer Science

Accademic Year: 2019/2020

Session: November 2020

Stage report

The import mechanism of the Yarel reversible language

Supervisor: Prof. Luca Roversi

Candidate: Matteo Palazzo

Student ID: 859134

Curriculum: Languages and Systems

"Dichiaro di essere responsabile del contenuto dell'elaborato che presento al fine del conseguimento del titolo, di non avere plagiato in tutto o in parte il lavoro prodotto da altri e di aver citato le fonti originali in modo congruente alle normative vigenti in materia di plagio e diritto d'autore. Sono inoltre consapevole che nel caso la mia dichiarazione risultasse mendace, potrei incorrere nelle sanzioni previste dalla legge e la mia ammissione alla prova finale potrebbe essere negata."

"I declare that I am responsible of the content of this document, which I submit in order to achieve the title. I declare that i have not plagiarized, in whole or in part, the work produced by other. I also declare that I have cited the original sources in a way that is congruent with the current regulations on plagiarism and copyright. I am aware that if my declaration is found to be false, then I could incur the penalties provided for by the law and that my admission to the final exam may be denied"

Contents

Introduction

Yarel is a reversible language developed by Luca Roversi, Claudio Grandi and Dariush Moshiri. The project can be found on Github on [1].

Even though in Yarel it was already possible to write code in different modules, and integrate these modules between each other through an import mechanism, the mechanism lacked of many features and had several problems. The goal of this stage is to solve these problems and allow a better use of the modules and the import mechanism.

The solutions to these problems are discussed in Chapter 5.

1.1 The Xtext framework

In order to improve the import mechanism, it was necessary to make several changes and additions to the Yarel compiler (parser, validator and generator). This compiler is realized through the *Xtext* framework, which is an Eclipse plug-in that allows to easily create DSL (*Domain-Specific-Language*) and their compiler (or interpreters), starting from the DSL grammar.

Chapter 4 presents an overview of the Xtext framework, in order to better understand the change and addition realized in my work.

1.2 Text structure

This text is structured in the following chapters:

- *Chapter 2*: A quick overview of reversible computation.
- *Chapter 3*: Description of the Yarel language.

- *Chapter 4*: Description of the functioning of the Yarel compiler and a overview of the Xtext framework.
- *Chapter 5*: The focus of this work, problems of the import system and their solutions.
- *Chapter 6*: An example of how the changes made impact on the Yarel code.
- *Chapter 7*: A conclusion on the state of Yarel after my work.

Therefore Chapter 2 and 3 give a theoretical background on this work; Chapter 4 is useful to understand the changes made in this work; the remaining chapters expose the results of this work, and how they were achieved.

Reversible Computing

Since always computation has been intended in a *forward* way. For this reason each program is built so that given an input it returns an output. Once the output is obtained there is no way to proceed *backward*, i.e., given the output it is not possible to freely obtain the original input.

With the reversible computation this is possible. Through it the program can proceed forward or backward (*bi-directional execution*) so that it can move from the input to the output or viceversa, passing through possible intermediate states.

An algorithm (therefore a program) can be considered as a function, whose domain is composed by each possible input configurations, while the co-domain is composed by each output configurations. In order to be reversible, a program must be equivalent to an *invertible-function* and therefore a *bijective* function, i.e, a function in which for each output exists one and only input. It is therefore possible to obtain an inverse function, and so to reverse the program, since given any output the input can be univocally generated.

2.1 Reversible languages

A reversible program can be written in a language that is not strictly reversible. In fact many traditional languages contain a sub-set of reversible instructions. It is therefore possible to write a reversible program by using only these instructions.

If even a single irreversible construct is used, then the whole program become irreversible. However this program may be converted into a reversible one by using particular memory techniques that add a temporal and spatial overhead to the execution.

Thus when a reversible program is built in this way efficiency is not granted. For this reason languages that are reversible by nature have been developed. These languages beside granting an efficient backward execution, offer a set of reversible constructs that give a greater expressiveness to the developer.

In order to highlight this distinction Perumalla [8] divides programs in three categories:

- IP (Irreversible program): programs written in traditional languages that are irreversible because they use at least an irreversible construct.
- IRP: programs written in traditional languages but are reversible because they only use reversible constructs.
- RP (Reversible program): reversible programs written in reversible languages which use peculiar reversible constructs or memory operations. This programs cannot be expressed with irreversible languages.

2.2 Reversible language issues

In order to be completely reversible, a language must redefine the concept of flow control and primitive operations. Perumalla's [8, Chapter 8] expose a wide variety of problems related to the reversibility of basic statements.

E.g. it is not straightforward to reverse the *if* statement. When it is executed forward the flow control is splitted in two o or more branches that are then merged into a single flow. At this point the guard value is lost, so it becomes impossible to reverse the execution. In fact when executing backward, information is needed in order to chose which of the *if* branches must be traversed in reverse. This can be solved by introducing auxiliary bits that memorize which branch was executed.

A similar problem concerns loop statements. They can execute an arbitrary number of times. When executing backward the exact count of iteration is lost so the statement cannot be reversed. In order to reverse the execution a counter variable can be introduced.

Another issue relates to the assignment. A reversible language must prohibit each irreversible assignment, i.e., assignments in which there is no relation between the assigned variable, its old value and its new one.

2.3 Applications

Reversible computing has several applications in many different fields. Some of them are:

- Parallel computing: it is possible to decrease the time spent in the synchronization and increase concurrency if we assume that processors can execute code in a bi-directional way and if we let them compute asynchronously. In this way the synchronization can proceed in background and its only job is to detect at run-time whether there are some violations of data dependency order. If a violation is detected, a roll-back is made by going backward to the point where the violation was made, once it is resolved the computation restarts forward.
- Processor architecture:
  - Instruction execution: given a list of instructions it is possible to execute the i-th instruction before the ones that precede it. This can be done only if the instructions do not depend by each other. However speculative execution may be used in order to execute instructions in a different order even though there is a potential conflict. I.e., after a later instruction is executed before the earlier ones, conflict detection is performed. If a conflict is detected then the results of the speculative execution are quashed and a roll-back occurs. If the program is reversible the roll-back can be done by reverse executing the speculated instruction.
  - Recovering register values: before a value inside a register is going to be overwritten, the value is stored in the main memory. Since the access to the memory is slower than the one to the register it is preferable to keep operations confined to register accesses. Thus when the lost register value is again needed, it is sometimes possible to recover the value by reverse computing earlier instructions, instead of retrieving the value from the main memory.
- Debugging: When running a program, if an unexpected condition occurs, the program execution can be traced back to the point where the error occurred. In order to do that the states between the operations needs to be stored. Because there can be many of them, it is not always possible to save all of them. As a consequence only a small execution window can be stored, hence debugging the program becomes hard. To solve this problem reversible computing comes in help; instead of saving the variable values and recovering them from memory, it is possible to

proceed backward until the point in which the error occurred, because the code that led to the problematic code admits reversible execution.

- Fault detection: it is possible to use reversible programming to check whether a fragment of a program P has been correctly executed. In fact if P is executed in a state S resulting in a state S', then the reverse execution of P starting from S' must result in S.
- Roll-back: reversible computing can be used to efficiently roll-back failed transition, without the use of auxiliary data-structures.

Perumalla's [8, Chapter 2] deepens these applications and exposes a wide variety of further applications.

Yarel

Yarel (Yet-Another-REversible-Language) is an experimental reversible functional language, where given a function f it is possible to "freely" obtain its inverse *inv[f]*.

Yarel implements the class of Reversible Primitive Permutation (RPP). Therefore Yarel inherits RPP's main properties:

- It is complete with respect to PRF (Primitive Recursive Function).
- It allows to implement functions with type $\mathbb{Z}^k \to \mathbb{Z}^k$ only.

3.1 Primitive Recursive Function

PRF (*Primitive Recursive Function*) is the class $\mathcal{C}$ of functions such that:

- $\mathcal{C}$ contains:
  - Each constant function:

    $$c(x_1, \ldots, x_n) := m \quad (m \in \mathbb{N})$$

  - Each projection function:

    $$p(x_1, \ldots, x_n) := x_i \quad (1 \leq i \leq n)$$

  - The successor function:

    $$s(x) = x + 1$$

- $\mathcal{C}$ is closed with respect to:
  - Composition:

    Given a function $h \in \mathcal{C}$ and n functions $g_i \in \mathcal{C}$, then *f*, which is defined as follows, is in $\mathcal{C}$:

$$f(\vec{x}) = h(g_1(\vec{x}), \ldots, g_n(\vec{x}))$$

– Recursion:

Given two functions g and h in $\mathcal{C}$, then f, defined as follows, is in $\mathcal{C}$:

$$f(0, \vec{x}) = g(\vec{x})$$
$$f(s(y), \vec{x}) = h(y, f(y, \vec{x}), \vec{x})$$

Hence, PRF is inductively defined: the base case are the primitive function (costant, projection and successor), while the induction is made through composition and recursion.

Even though PRF is highly expressive: "programs which terminate but do not belong to PRF are rarely of practical interest" ([7]), PRF is not closed under inversion, therefore a primitive recursion function is not necessarily reversible. E.g., it is not always possible to reverse the projection function: given the tuple $t = \langle 3, 5, 2, 6 \rangle$ we know that $p_2(t) = 5$, however it is not possible to define the inverse function, thus it is impossible to obtain t starting from the function output 2.

3.2 RPP

RPP (Reversible Primitive Permutation) is a class of permutation first introduced in [7]. This "language" is able to express each function defined in PRF, it is PRF complete and sound. However RPP actual benefit is that it is reversible by definition. It is in fact closed under inversion, i.e., for each function f defined in RPP, the inverse function f^{-1} belongs to RPP.

RPP is strictly included in the set of all the permutations and is composed by functions with type $\mathbb{Z}^k \to \mathbb{Z}^k$ only.

In analogy with PRF, RPP is inductively defined, it is the closure of composition schemes on basic functions:

- RPP basic functions are the following ones:

$$\begin{aligned} I\langle x\rangle &= \langle x\rangle && \text{(Identity)}\\ S\langle x\rangle &= \langle x+1\rangle && \text{(Successor)}\\ P\langle x\rangle &= \langle x-1\rangle && \text{(Predecessor)}\\ N\langle x\rangle &= \langle -x\rangle && \text{(Sign-change)}\\ \left\lfloor \begin{matrix}1 & 2\\ 2 & 1\end{matrix}\right\rfloor^{2}\langle x,y\rangle &= \langle y,x\rangle && \text{(Transposition)} \end{aligned}$$

- RPP compositive schemes are the following ones:
  - Series composition $(f;g)$:

$$(f;g)\langle x_1,\ldots,x_n\rangle = (g\circ f)\langle x_1,\ldots,x_n\rangle$$

  - Parallel composition $(f||g)$:

$$(f||g)\langle x_1,\ldots,x_n\rangle\langle y_1,\ldots,y_n\rangle = (f\langle x_1,\ldots,x_n\rangle)\cdot(g\langle y_1,\ldots,y_n\rangle)$$

  - Finite iteration it$[f]$:

$$\text{it}[f](\langle x_1,\ldots,x_n\rangle\cdot x) = ((\underbrace{f;\ldots;f}_{|x|\text{ times}}||I))(\langle x_1,\ldots,x_n\rangle\cdot x)$$

  - Selection if$[f,g,h]$:

$$\text{if[f, g, h]} = \begin{cases}(f||I)(\langle x_1,\ldots,x_k\rangle\cdot x) & \text{if x} > 0\\ (g||I)(\langle x_1,\ldots,x_k\rangle\cdot x) & \text{if x} = 0\\ (h||I)(\langle x_1,\ldots,x_k\rangle\cdot x) & \text{if x} < 0\end{cases}$$

Inverse functions

For each of the functions defined above the inverse exists:

$$I^{-1} := I, N^{-1} := N, S^{-1} := P, P^{-1} := S$$

$$\left(\left\lfloor \begin{matrix}1 & 2\\ 2 & 1\end{matrix}\right\rfloor^{2}\right)^{-1} := \left\lfloor \begin{matrix}1 & 2\\ 2 & 1\end{matrix}\right\rfloor^{2}$$

$$(g;f)^{-1} := f^{-1};g^{-1}$$

$$(f||g)^{-1} := f^{-1}||g^{-1}$$

$$(it[f])^{-1} := it[f^{-1}]$$

$$(if[f,g,h])^{-1} := if[f^{-1},g^{-1},h^{-1}]$$

Because each RPP function is defined by the composition of some primitive functions and since each primitive function and composition scheme is invertible, every RPP function is invertible, thus reversible.

3.3 The Yarel language

As already said, Yarel is a *domain-specific-language* (DSL) which implements RPP.

The outcome of this work is to allow to write Yarel programs in modules, which contain declarations and definitions of functions. It is possible to declare functions through the statement **dcl** that has the following syntax[1]:

```
'dcl' FunctionName ':' n int
```

where *FunctionName* is a valid function name, while n is an integer bigger than 0.

The definition of declared functions can be made through the **def** directive:

```
'def' FunctionName ':=' BodyFun
```

where *BodyFun* contains the function body which is defined by composition of primitive functions, which are Yarel's constructs.

```
1 module addition{
2     dcl addition : 2 int
3     /* Input:   x   y */
4     /* Output:  x+y y */
5     def addition := it[inc]
6 }
```

Listing 3.1: Yarel module example

Listing 3.1 contains an example of a Yarel module, which contains the definition of the sum.

[1]The actual syntax of the declaration is more complex than the reported one. It provides the possibility of declaring functions with multiple different types which can differ from the integers. However when this work was realized the only implemented type was the one representing the integers (int).

3.3.1 Main Constructs

Since Yarel is an implementation of RPP, its main constructs represent the RPP primitive functions and composition schemes.

Therefore for each primitive function there is a built-in function in Yarel. Each of these functions has type $\mathbb{Z}^n \to \mathbb{Z}^n$.

- The *identity* function **`id`** stands for the RPP identity **`I`**. Its ariety is 1. When applied to an integer x the function returns the same integer x.
- The *increment* function **`inc`** stands for the RPP successor function **`S`**. Its ariety is 1 and when applied to an integer x returns its successor $x + 1$.
- The *decrement* function **`dec`** stands for the RPP predecessor function **`P`**. Its ariety is 1 and when applied to an integer x it returns the predecessor of the integer $x - 1$.
- The *negation* function **`neg`** that stands for the RPP sign-change function **`N`**. Its ariety is 1 and when applied to an integer x returns an integer that has the same value but opposite sign $-x$.
- The *permutation* function **`/`**$i_1, i_2, \ldots i_n$**`/`**, is a generalization of the RPP transposition. Its ariety is n (with $n \geq 1$). If applied to a tuple $t = \langle x_1, x_2, \ldots, x_n \rangle$ returns a permutation $t' = \langle x_{i_1}, x_{i_2}, \ldots, x_{i_n} \rangle$ where the first element of t' is the i_1-th of t, the second element is the i_2-th and so on.

Instead for each RPP composition scheme, Yarel implements a construct that takes one or more functions (primitive or composed) as a parameter and returns a new one.
The constructs are the following ones:

- The *series composition* **`";"`**: given two functions f and g with the same ariety n, the series composition **`f ; g`** returns a function that has the same ariety n and whose semantics is the same as RRP series composition. If applied to a tuple t the function returns the tuple $t' = g(f(t))$.
- The *parallel composition* **`"|"`**: it represents RPP parallel composition. Given two functions f and g with ariety n and m, **`f | g`** returns a function with ariety $m+n$. As the name says, the parallel composition allows to apply f and g in parallel to a tuple of $m+n$ elements. I.e., f is applied to the first n elements, while g to the remaining m. Hence if

applied to a tuple $\langle x_1, \ldots, x_n, \ldots, x_{n+m} \rangle$ the function returns the tuple $t_f \cdot t_g$ where $t_f = f(\langle x_1, \ldots, x_n, \rangle)$, while $t_g = g(\langle x_{n+1}, \ldots, x_{n+m} \rangle)\rangle$.

- The *iteration* **`it`**: this construct represents RPP finite iteration. It takes a function with ariety n as a parameter and returns a function with ariety $n+1$. If applied to a function f, **`it[f]`** returns a function that allows to apply f to a tuple $t = \langle x_1, \ldots, x_n \rangle$ a finite number of times. In fact if applied to $t \cdot \langle x_{n+1} \rangle$ the function applies f to t x_{n+1} times and appends x_{n+1} to the final result in order to let the function reversible.

- The *selection* **`if`**: it represents RPP selection. The function takes the functions f, g, h that have the same ariety n as parameters. **`if[f, g, h]`** returns a function with ariety $n+1$ that if applied to a tuple $\langle x_1, \ldots, x_{n+1} \rangle$ applies f, g or h to the first n elements, depending on whether x_{n+1} is less, equal or greater than 0. The result of the function will then be $t \cdot \langle x_{n+1} \rangle$, where:

$$\begin{cases} t = f\big(\langle x_1, \ldots, x_n \rangle\big) & (x > 0) \\ t = g\big(\langle x_1, \ldots, x_n \rangle\big) & (x = 0) \\ t = h\big(\langle x_1, \ldots, x_n \rangle\big) & (x < 0) \end{cases}$$

Beside from the constructs introduced above, Yarel presents other two main constructs: the *function call* and the *inverse* (**`inv`**).

The first one allows to apply a declared function f inside the definition of another function g. This allows to factorize the code and to build even more complex functions.

The latter is maybe one of the most important constructs in Yarel. In fact as we already said, Yarel is an intrinsically reversible language. Therefore given any function f (whether it is built-in or defined by the user), **`inv[f]`** can be used to "freely" obtain the inverse of f, hence to reverse it.

Yarel Compiler

Yarel compiler has been written in Xtext. Therefore before making an overview, the Xtext framework will be introduced.

4.1 Xtext

Xtext is an Eclipse framework for the development of DSL. It allows to quickly implement all the aspects of a programming language starting from the parser, the validator and the code generator, up to a full Eclipse IDE Integration.

This chapter will briefly analyze some of the main aspects of a programming language which can be defined with Xtext. A wider description can be found on the Xtext documentation ([2]) or on Bettini [6].

4.1.1 Parsing

Xtext allows to easily create a parser. The user only needs to define a grammar. Once the grammar is defined, a parser (written in Java) is automatically generated.

The generated parser beside from checking the syntactic correctness of the programs, builds an AST (*Abstract Syntax Tree*). Each tree's node is an *EObject*, a peculiar kind of object introduced by the Xtext framework. An EObject represents a construct of the parsed program and contains additional information about it. Hence the AST represents the syntactic structure of the program and each of its nodes represents a construct of the program. Therefore the AST can be used to perform additional semantic checks: If they all succeed, the AST can be used to translate the program.

Defining a grammar

The syntax used to define an Xtext grammar is similar to the one used to define a CFG (*Context Free Grammar*). An Xtext grammar is in fact composed by rules of type: "$S : \alpha$;". In which S is a non-terminal symbol, while α is a string of terminal and non terminal symbols. α is annoted by actions which are used by the parser in order to construct the AST. These actions specify how to generate, and with which field, the EObjects that represent the constructs that are being parsed.

E.g, consider the following production rules, taken from the Yarel grammar:

```
1 Declaration: 'dcl' name=ID ':' signature=Signature;
2
3 Signature: types += Type (',' types+=Type)*;
4
5 Type: {Type} value=(INT)? 'int';
```

These rules describe the syntax of a declaration.

The first rule specifies that a **Declaration** is composed by the keyword **'dcl'** followed by an identifier **ID**, the colon **':'** and finally a **Signature**.
The assignments "name = ID" and "signature = Signature" specify that the EObjects that represent the declarations will be composed by two fields: name and a signature. The first will contain the ID (hence the name) of the declared function, while the latter will contain its signature.

The second rule states that a **Signature** is composed by one more **Type** separated by a comma **','**.
The assignment "types += Type" specifies that the EObjects of type Signature must have a field that represents a list containing a reference to each Type that composes the signature.

The last rule specifies that a **Type** is made of an optional integer value followed by the keyword **'int'**. Since the integer is optional, if it is not parsed then the value field would not exist, hence the EObject would not be created. In order to always create the Eobject the annotation "{Type}" is used. In this way the EObject is created as soon as a Type is parsed.

4.1.2 Validation

Validation rules are a built-in mechanism of Xtext. As explained in [6, Chapter 4], these rules allow to introduce additional constraints to the language. These constraints do not depend on the syntax, therefore they cannot be

easily expressed in the grammar. In fact a best practice is to do as little as possible in the grammar and as much as possible in the validation.

The validation rules can be defined in the Validator in a declarative way. A validation rule is nothing but a method expressed in a class that represents the Validator. This method is annotated with **@Check** and take as a parameter an EObject, which represent a language's construct. The type of the parameter is important because during the validation this method is going to be called on each AST's EObject, whose type is compatible with the type of the parameter. I.e, the type of the parameter specifies which language construct must respect the validation rule.

Inside the validation rule method it is possible to validate the passed construct by using the parameter fields (an example of why it is useful to annotate the AST with EObjects). Once all the checks are done, it is possible to throw a warning or an error, Which can contain a message, a type and the feature of the construct against which it is thrown.

Chapter 5.5 expose a variety of validation rules examples.

4.1.3 Code generation

Xtext allows to implement a code generator that can translate our DSL in any other language (Java, XML and so on).

The code generator must define a **doGenerate(Resource res, IFileSystemAccess fsa)** method, in which the compilation begins.

The parameter **res** contains the AST of the parsed program. It is possible to use the AST in order to generate the translated code in an efficient and intuitive way. In fact using the AST, it is possible to retrieve the EObjects of all the constructs contained in the program, which can then be used to generate the code.

The parameter **fsa** is instead a object that allows to create the file that will contain the translated program.

Chapter 4.2.2 contains an overview of the Yarel code generator, which can be used as an example of how to create a compiler in Xtext.

4.1.4 Xtend

Xtext is a Java-like language, introduced with the Xtext framework. All the aspects of a Xtext DSL can be implemented using Xtend. Xtend is widely

described in [6, Chapter 3], its documentation can be found at [3].

"The Xtend goal is to have a less 'noisy' version of Java"([6, Chapter 3]). In fact its syntax is very similar to the Java one, but it is less redundant and verbose.

Furthermore, it introduces a series of features that allows to write code in a functional way. Some of them are:

- **Type Inference**: If the type of a variable or a method can be inferred, it is not required to specify it.
- **Everything is an expression**: There are no statement, everything can be considered as an expression, therefore the last expression of a block is a return statement, hence the return keyword can be avoided.
- **Extensions method**: With this mechanism instead of passing the first argument inside the parentheses of a method invocation, the method can be called with the first argument as its receiver, e.g., "method(object)" becomes "object.method".
- **Lambda Expression**: Even though lambda expression has been introduced in Java 8, Xtend introduced lambda extension before Java. Lambda expressions in Xtend have their own type called function type, unlike Java where Lambda expressions are simple interfaces.
- **Multi-line template expressions**: These allow to write well-formatted strings without getting lost in a verbose syntax, so that the final result is perfectly readable. These expressions can be used to write strings that represent the output of the code generator. E.g., suppose that it is required to generate an empty class. In Java in order to generate an empty class the following method can be used:

```
1      public void generateClass(String className){
2        return "public class" + className + "{"
3          + "\n \t //Empty Class \n"
4          + "}";
5      }
6
```

  This method is hard to read and its not straightforward to understand what the output string represents.
  In Xtend the empty class can be instead generated by the following method:

```
    def generateClass(Sting name){
        '''
        public void <<name>>{
            //Empty Class
        }
        '''
    }

```

where inside the guillemets («») a variable is used.

The resulting string will be formatted as specified in the expression. This example shows how simple is to write well formatted strings in Xtend.

Beside from these and many other mechanisms described in [6], Xtend provides a wide variety of methods that can be used to traverse and navigate the AST of a program in a straightforward way, that is natural to read and maintain. This way, the implementation of all the aspects of the compiler becomes easier.

4.2 Overview of the Yarel compiler

Chapter 5 will show how my work changed the Yarel compiler in order to improve the Yarel import system. Hence, this chapter will expose an overview of the Yarel compiler in order to better understand the changes that has been made.

4.2.1 The grammar

Before my work, Yarel grammar was the following one:

```
Model: 'module' name=ID '{' elements+=Element* '}';

Element: Import | Declaration | Definition;

Import:
  'import' importedNamespace=QualifiedNameWithWildcard;

QualifiedName: ID ('.' ID)*;

QualifiedNameWithWildcard: QualifiedName '.*'?;

Declaration: 'dcl' name=ID ':' signature=Signature;

Signature: types += Type (',' types+=Type)*;

Type:  {Type} value=(INT)? 'int';

```

```
18 Definition:
19     'def' declarationName=[Declaration] ':=' body=Body;
20
21 Body: SerComp;
22
23 SerComp returns Body:
24   ParComp ({SerComp.left=current} ';' right=ParComp)*;
25
26 ParComp returns Body:
27   BodyBase ({ParComp.left=current} '|' right=BodyBase)* ;
28
29 BodyBase returns Body:
30    '(' Body ')' | Atomic;
31
32 Atomic returns Body:
33     {BodyId}  funName='id'
34   | {BodyInc} funName='inc'
35   | {BodyDec} funName='dec'
36   | {BodyNeg} funName='neg'
37   | {BodyFor} function='for' '[' body=Body ']'
38   | {BodyInv} function='inv' '[' body=Body ']'
39   | {BodyIt}  function='it' '[' body=Body ']'
40   | {BodyIf}  function='if' '[' pos=Body ',' zero=Body ','
      neg=Body ']'
41   | {BodyFun} funName=[Declaration]
42   | {BodyPerm} permutation=Permutation;
43
44 Permutation:
45   '/' indexes+=Digit (indexes+=Digit)* '/';
46
47 Digit:
48   value=INT;
```

Listing 4.1: Yarel Grammar

The grammar establishes that a Yarel file is basically a module composed by an optional series of import, declaration and definition (line 1).

The import structure is obvious, while the structure of declarations has been exposed in chapter 4.1.1. We will then analyze the definitions grammar.

A definition is composed by the keyword **'def'** followed by the name of the function that we want to define, the symbols ':=' and a **Body**. Through the statement "declarationName = [Declaration]" (line 19) the grammar requires that the name of the function that we are going to define, is the name of a function that has been declared[1].

The most interesting part is the Body grammar. A Yarel function is realized through the composition of primitive functions, where the composition

[1]This is an example of an Xtext mechanism called *cross-reference*, that will be analyzed in chapter 5.2

scheme with the lowest priority is the series composition. Therefore, we would like to define the body of a function as something like:

$$\begin{aligned} Body &\rightarrow SerComp \\ SerComp &\rightarrow SerComp\ ';'\ SerComp \\ SerComp &\rightarrow ParComp \\ &\dots \end{aligned}$$

However we cannot define the Body of a function like that because it is a left recursive grammar[2], which cannot be parsed by the deterministic top-down parser generated by Xtext. Therefore, in order to remove left recursion, the actual Yarel grammar has been obtained using a technique called *left-refactoring*.

The final grammar defines the Body of a function as the series composition of the parallel composition of atomic operators, which can be Yarel primitive functions (id, inc, dec, neg, for, inv, permutation), or other compositional schemes (if, it for), or the function call. The grammar also defines that the operator with the highest priority are the atomic ones, followed by the parallel composition and the series composition. It also defines that both parallel and series composition are left associative. The reason behind this is in the depth of a node in the AST: the greater the depth of the node the greater the priority of the construct it represents (series composition is usually the first construct to be parsed, hence it is the first node of the AST). This is because the parent node must be solved in terms of child nodes.

The annotations "{SerComp.left = current}" and "right = ParComp" (line 24) specify that, if a serial composition is parsed, that is, the part $(\dots)^*$ is parsed at least once, then the node of the AST representing the serial composition is annotated with an EObject of type SerComp. The EObject will have two fields: left and right; the first contains the left part of the serial composition, the latter the right part.
Same for the parallel composition.

We will now analyze the production rules for the atomic symbols; some of them were changed in my work. In Yarel grammar an atomic symbol can be:

- a primitive function (**id**, **inc**, **dec**, **neg**);

[2]A grammar is left recursive if it has at least one left recursive rule, that is a rule where the first symbol is non-terminal and refers to the rule itsel ($SerComp \rightarrow SerComp\ ';'\ SerComp$)

- a **it**, **for** or **if** composition scheme;
- a **function call**;
- an **inv** construct;

Each production rule is annotated with something like "{BodyId}", "{BodyInc}" etc. Through this annotation the parser creates an EObject of a different type for each atomic symbol. This way, in the validator or in the code generator we can distinguish the various atomic constructs. Finally, with "**Atomic returns Body**" we state that the EOBjects that represents an atomic construct will extend the class Body.

Figure 4.1 shows the UML Diagram of the EObjects that represent the Body of the definition of the following function:

```
1 //input: a, b
2 //output: a + b - 1, b
3 f := it[inc] ; (dec | id).
```

This diagram is basically the annotated AST, that is generated when the body is parsed.

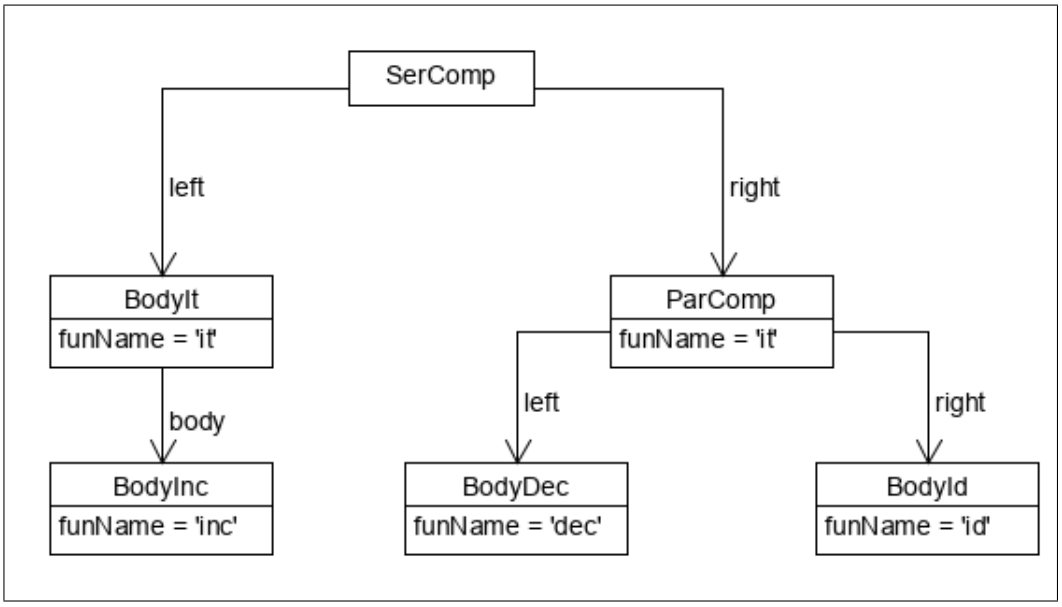

Figure 4.1: AST of the body of a definition

4.2.2 The code generator

Yarel is compiled in Java. To create a reversible output code, for each Yarel function inside a module, two classes are created, one for the function, the other one for its inverse. All these classes are then grouped into a single package that has the same name as the Yarel module. In order to create two classes for each function, every module is compiled in two "direction": forward and backward. In the forward direction the compiler generates classes for the base functions, in the backward direction it generates classes for their inverse.

Each of these classes implements the same interface RPP, so that they can be used interchangeably. In order to implement the RPP interface, a class must have a method "**int getA()**" which returns the ariety of the function, and a method **int[] b(int[] x)** that allows to apply the function to an array of integers (hence a tuple) x. The implementation of this method will therefore change according to the function that is being compiled.

It is useful to have a common RPP interface because since most of Yarel functions are defined as the composition of two or more functions, the class F that compiles a function f will be built using a reference to the classes that compile the functions that compose f. F can indiscriminately reference and use these classes because they all implements the common RPP interface.

Each class that translates a Yarel function is usually composed by one or more RPP fields. These fields contain a reference to the implementation of the functions that compose the compiled function. E.g, the class which represent a series or parallel composition, is composed by two field: the first references the function at the left of the composition, the latter references the function at the right. For example series composition is compiled in the following way:

```
1  RPP l = new RPP() {
2      <<compile(body.left, fwd)>>
3  };
4  RPP r = new RPP() {
5      <<compile(body.right, fwd)>>
6  };
7  private final int a = l.getA();
8  public int[] b(int[] x) { // Implements a serial composition
       .
9      return
10         <<IF fwd>>this.r.b(this.l.b(x))<<ENDIF>>
11         <<IF !fwd>>this.l.b(this.r.b(x))<<ENDIF>>;
12 }
13 public int getA() { return this.a; }
```

Listing 4.2: Compilation of the series composition

Note the specification «IF fwd» and «IF !fwd» (line 10 and 11). They are used to create a different method according to the direction of the compilation (forward or backward). **Fwd** is a flag that is used in the compiler in order to distinguish the direction of the compilation.

As already said the fields l and r, compile and reference the functions to the left and right of the composition. In fact they contain a reference to two RPP classes that are the output of the compilation of the functions to the left and the right of the composition (line 1-6).

The b method applies the serial composition to a tuple x. In fact, if it is compiled in the fwd direction, it applies l.b() to x ("this.l.b(x)"), the result of this application is then passed to r.b() ("this.r.b(this.l.b(x))") (line 10). I.e, let be f and g the functions to the left and right of the composition; let be x an input tuple; then the b method first applies f to x and then applies g to the result of f *(x)*. The final result is therefore $g(f(x))$ which is the definition of the series composition. A similar reasoning holds true for the backward direction.

Java Yarel Generator

The Yarel compiler is defined inside the *JavaYarelGenerator* class. The main method of this class is **compile(Body b, boolean fwd)**, which generates the bodies of the Yarel functions definitions. The first parameter of the method is an EObject which contains the body of the definition, while the second parameter is the forward flag (which determines the direction of the compilation).

The method is recursively defined and has the following structure:

```
1  private def String compile(Body b, boolean fwd){
2  /*This switch works by checking the variable type of b,
3  similar to a java instanceof*/
4      switch (b) {
5          //For each type of function,
6          //different java code is generated
7          SerComp: /*Code for serial composition*/
8          ParComp: /*Code for parallel composition*/
9          BodyId: /*Code for id application*/
10         BodyInc: /*Code for inc application*/
11         BodyDec: /*Code for dec application*/
12         BodyNeg: /*Code for neg application*/
13         BodyFun: /*Code for function call*/
14         BodyPerm: /*Code for permutation application*/
15         BodyInv: /*Code for inv application*/
16         BodyFor: /*Code for 'for' construct*/
```

```
17          BodyIf: /*Code for if construct*/
18          BodyIt: /*Code for it construct*/
19      }.toString
20  }
```

Listing 4.3: The compile method

As defined in the Yarel grammar, Body is a super class while the actual body of a parsed function can be of many different types: a SerComp (series composition), a ParComp (parallel composition) a BodyId (the id function) and so on. Hence the **compile** function makes a switch on the type of the passed body, and generates a different code according to the type of the body (checking if it is a series composition, a parallel composition, etc).

The base cases are BodyId, BodyInc, BodyDec, BodyNeg, BodyFun and BodyPerm. In fact, they represent the application of function already defined by the user or in Yarel. Hence the compiler only generates a Java code that makes a call to these functions. Instead the remaining cases are the inductive ones because they represent the compositional scheme. Therefore before compiling them, we must compile the functions that are being composed. Thus, the compile method is recursively called on the Body of the functions that we are composing. Listing 4.2 shows the code that compiles an inductive case, where the series composition is compiled by recursively calling the compile method on the left (body.left) and right (body.right) functions of the composition.

Beside from these methods, JavaYarelGenerator contains many other methods which compile the classes that represent the Yarel primitive functions. These classes will then be grouped into a package named "yarelcore".

Import System

5.1 Yarel before my work

Besides from the base functionalities Yarel also supported a basic import system. However the system had just one purpose: it made visible the functions declared in other modules within the module they were imported.

The system however had some problems.

1. Functions and modules names ambiguity:
   (a) If two different functions with the same name were imported inside a module, then the compiler would not be able to distinguish them and solve the call to one of the them.
   (b) The Java code resulting from the compilation of a module might not work, this was due to the fact that it did not respect any Java convention about class and package names.
2. Unhandy syntax: imports had to be done inside Yarel modules. For this reason the syntax was different from the one of all the other programming languages, resulting in a not very intuitive syntax.
3. Lack of some validation rules, this caused an instability in the compiler output Java code.
4. Wrong definitions scope: inside a given module it was possible to define functions declared in a different module.

This chapter will analyze this problems, show their solutions and give a detailed explanation of the solutions implementation.

5.2 Introduction of Qualified Names

To solve the ambiguity that point 1.a refers to, the possibility to use function by their qualified names has been introduced. The qualified names have the following syntax: *moduleName.functionName*.

Qualified names allow to univocally identify functions, because within each module two functions cannot have the same name; same for two modules within the same classpath. In this way if two functions with the same name f are imported in the module mod1 from two different modules mod2 and mod3, then it is possible to distinguish them by using their qualified names: mod2.f and mod3.f.

The grammar

In order to allow to apply functions by their qualified names, rule at line 32 in the Yarel grammar (Listing 4.1) has been changed as follows:

```
1 Atomic returns Body:
2     ...
3     | {BodyFun} funName = [Declaration | QualifiedName]
4     ...
```

The updated rule specifies that when a function call is made, the function name must respect the qualified name syntax, i.e.:

```
QualifiedName: ID ('.' ID)*
```

Note that since this rule accept single IDs, the use of the qualified name is optional.

Inside the square brackets a *cross-reference* is specified. This is a peculiar Xtext mechanism explained in [6], chapter 10. This mechanism with rule like: "[<Type> | <Syntax>]" allows to specify that a parsed string *s* must respect the <Syntax> and refers to a given element that exists in the program and has the specified <Type>.
This check is made by requesting the scope of the context where the string *s* is found, and verifying that this scope contains an element that can be referenced by the *s*.

In this case, the grammar requires that the string used to call the function, is not just a qualified name, but is the qualified name of a function that is declared (Declaration) inside the program (within the same module or in another one) and whose name is contained in the scope where the call is

made. This scope by default contains both the simple and qualified name of the function.

The following example explain how *cross-reference* works:

```
1  import mod2.h
2  import mod1{
3      dcl g : int
4      def g := id
5
6      dcl k : int
7      def k := id
8
9      dcl f : int
10     def f := function_name
11 }
```

Listing 5.1: Cross-reference example

The f definition scope is composed by the following declared function names: "h", "mod2.h", "g", "mod1.g", "k", "mod1.k", "f", "mod1.f". The cross-reference therefore checks that ***function_name*** is a qualified name (or a simple name) and that it refer to a declared function, which name is contained in the scope. **Function_name** must then be one of these functions: "h", "mod2.h", "g", "mod1.g", "k", "mod1.k", "f", "mod1.f"[1].

Code generation

As explained in Chapter 4.2.2, the Yarel compiler works in the following way: for each module a package is created, while for each function *f* two Java classes are generated, one is the compilation of *f*, the other is the compilation of the inverse of *f*. This two classes implement the RPP interface, that represents the reversible functions. Each class contains:

- One or two RPP fields.
- An integer field named "a": it contains the function ariety.
- The ariety getter.
- A method called "b".

To solve the ambiguity related to the existence of two different functions with the same name in two different modules we are interested in how the compilation of the function call works. The output of the compilation of the call to the function *f*, is a Java class, whose only purpose is to delegate the

[1]Since in the scope of the **f** definition the same symbol f appear, the function f could be recursively defined. This should be avoided by introducing a new validation rule.

function call to another RPP object whose type represents the function f, and that will actually apply f to the given input. To solve the function's name ambiguity in the generated Java code, when the object that represents the function f is created, it is always created using the qualified name of its class. The function call is compiled in the BodyFun case of the **`compile( Body b, boolean fwd)`** method (Listing 4.3):

```
BodyFun: {
    val qualifiedName = qnp
                        .getFullyQualifiedName(b.funName);
    val moduleName = qualifiedName.firstSegment;
    var functionName = (fwd ? "" : "Inv") +
            qualifiedName.lastSegment.toFirstUpper;
    if(moduleName != b.getContainerOfType(typeof(Model)).
    name)
        functionName = moduleName.toFirstLower + "."
                + functionName;
    '''
    RPP function = new <<functionName>>();
    private final int a = function.getA();
    public int[] b(int[] x) {
        return this.function.b(x);
    }
    public int getA() { return this.a; }
    '''
}
```

Listing 5.2: Function call compilation

The object **`qnp`** has type *IQualifiedNameProvider*. It computes the qualified name[2] of each program element. With its method **`getFullyQualifiedName`** we can get the qualified name of the called function in the Yarel program.

In Yarel each function is contained inside a module, which is not contained in anything else. For this reason each function qualified name will only have two segments: the function name and the module name. We can then get these names with the methods **`firstSegment`** and **`lastSegment`**[3]. Once we get the function name, we should also consider if we are generating the class for the inverse function, if so we add the "Inv" prefix to the class name.
In order to generate a more readable Java code, the qualified name of the class will be used only if the called function is not declared inside the module where it's used.

[2]In the Xtext frame work the qualified name of an element is created from its name, that's appended at the qualified name of its container that is recursively generated in the same way. Names are therefore composed by various segment (each of them is the name of a container) separated by a dot.

[3]firstSegment and lastSegment methods are called through the peculiar Xtend mechanism called *Extension methods* (see Chapter 4.1.4)

We can now generate the code, which is represented by the string inside the triple quotes[4].
First of all we create an RPP field called **function**, this will contain an object whose type represents the called function. This object will be created using the qualified name of the class (contained in the variable ***functionName***); in this way, as already explained, there's no possibility to have an ambiguity related to the function / class name.
We then add a field **a** that represents the function ariety.
Finally we add the method **b**. This method will delegate the call to the **b** method of **function**, which will actually solve the function application.

Note that there is no return statement since Xtend interprets the last expression inside a block as a return.

5.3 Class and package names ambiguity

Yarel compiler could sometimes generate a not working Java code. This was due the ambiguity of the generated packages and classes name, that occasionally entered in conflict with names already contained in the Java's library.
This was caused by the fact that class and package names were generated without any logic. They only had the name same of the module and function they represented; for this reason it could happen that the compiler generated package whose name started in uppercase, or classes with a name starting in lowercase. Obviously this went against Java names convention.

Files names

First of all, folders that represents packages must have a name that start in lower case, while files that contains classes must start in uppercase. This is now granted in the **doGenerate** method:

```
override doGenerate(Resource resource, IFileSystemAccess2 fsa,
    IGeneratorContext context) {
//Looks for the Model in the .rl file
    val model = resource
                .allContents
                .toIterable
                .filter(Model).get(0)
    val packageName = "yarelcore"//model.name
```

[4]The string inside the triple quotes is a *multi-line template expression* (see Chapter 4.1.4)

```
8       fsa.generateFile(packageName+"/WrongArityException.java",
9           exceptionsGenerator(packageName))
10      fsa.generateFile(packageName+"/RPP.java",
11          RPPGenerator(packageName))
12      fsa.generateFile(packageName+"/Id.java",
13          IdGenerator(packageName))
14      fsa.generateFile(packageName+"/InvId.java",
15          InvIdGenerator(packageName))
16      fsa.generateFile(packageName+"/Inc.java",
17          IncGenerator(packageName))
18      fsa.generateFile(packageName+"/InvInc.java",
19          InvIncGenerator(packageName))
20      fsa.generateFile(packageName+"/Dec.java",
21          DecGenerator(packageName))
22      fsa.generateFile(packageName+"/InvDec.java",
23          InvDecGenerator(packageName))
24      fsa.generateFile(packageName+"/Neg.java",
25          NegGenerator(packageName))
26      fsa.generateFile(packageName+"/InvNeg.java",
27          InvNegGenerator(packageName))
28      collectArities(model)
29      //Generates java code starting from the Model
30      compile(fsa, model)
31      //Tests
32      fsa.generateFile(
33          model.name.toFirstLower + "/" +
34          model.name.toFirstUpper + "Test.java",
35          testFileGenerator(model)
36      )
37      //Play source
38      fsa.generateFile(
39          model.name.toFirstLower + "/" +
40          model.name.toFirstUpper + "PlayWith.java",
41          playGenerator(model.name.toFirstLower, model)
42      )
43  }
```

Listing 5.3: Yarel file compiler

The **doGenerate** method is where the compilation starts, while the method **generateFile(String fileName, CharSequence contents)** is used to generate a file named *fileName* whose content is *contents*.

We can see that the package "yarelcore", which contains the implemenation of all the base Yarel functions (Id, Inc, Inv, etc), respects the conventions: the folder that represents the package starts in lowercase: "yarelcore", while the file of each classes starts in uppercase, e.g. "Id.Java". Same for the class Test and PlayWith (testing classes that are automatically generated when

a Yarel file is compiled), where with model.name.toFirstLower we require that the folder (package) name, that contains them, starts in lower case; while with model.name.toFirstUpper + "Test.java" / "PlayWith" we make the files names, where the classes are written, start in uppercase.

Such a thing happens also in the **`compile(fsa, model)`** call, where the packages and files, that represent the Yarel modules and functions, are generated.

```
1  private def compile(IFileSystemAccess2 fsa, Model model) {
2      var definitions = model.elements.filter(Definition)
3      val folder = model.name.toFirstLower + "/"
4      for(definition: definitions) {
5          var compilation = compile(model, definition, true)
6          fsa.generateFile(
7              folder +
8              definition.declarationName
9                        .name
10                       .toFirstUpper + ".java",
11             compilation
12         )
13         compilation = compile(model, definition, false)
14         fsa.generateFile(
15             folder +
16             "Inv"+definition.declarationName
17                             .name
18                             .toFirstUpper+ ".java",
19             compilation
20         )
21     }
22 }
```

Listing 5.4: Module compiler

Remember that for each module a package with the name of the module is generated; while for each function, two classes are generated inside this package: one that compiles function, the other one that compiles its inverse.

In this compile method we assure that the conventions, for these packages and classes names, are respected.
The **`folder`** variable contains the package name, this will start in lowercase due the toFirstLower call (line 3).
While the **`definitions`** variable contains all the declared functions of the module. For each of them two files are generated through the two **`fsa.generateFile`** calls at lines 6 and 14. Inside these calls, we get the function name using definition.declarationName.name, and we make it starts in uppercase through the toFirstUpper call. If the file will contain the inverse function we then put the "Inv" prefix before the function name. It is easily visible that the names of the folder and file generated by this call, will respect the naming convention.

Actual code

Once we create the file for the class we must also assure that the actual code respects the naming conventions. This is done inside the **`compile(Model model, Definition definition, boolean fwd)`** method, that create a string that represents the output Java code of the compiled function.

```
1  private def compile(Model model, Definition definition,
       boolean fwd) {
2      '''
3      package <<model.name.toFirstLower>>;
4      import java.util.Arrays;
5      import java.lang.Math;
6      import yarelcore.*;
7      public class
8          <<IF !fwd>>Inv<<ENDIF>><<definition.declarationName
9                                          .name
10                                         .toFirstUpper>>
11         implements RPP {
12             <<compile(definition.body, fwd)>>
13     }
14     '''
15 }
```

Listing 5.5: Function compiler

In the code returned by this function, the "package" statement is completed with the name of Yarel module in which the compiled function is declared. Due to the toFirstLower call, this name will start in lowercase. Line 8 assures that the class will be named as the compiled function, and that this will start in uppercase. If we are compiling the inverse function then the class name will have the prefix "Inv". The Java naming conventions are then respected.

Finally we must be sure that when, for any reason, class or package names are used (e.g. when an object is being created), they are correctly named. We must therefore pay attention to the **`compile(Body b, boolean fwd)`** (Listing 4.3) that compiles the code of the body of each function. An example is the compilation of a function call, reported in the listing 5.2. There, the variable ***`functionName`*** contains the qualified name of the class that represents the called function. It assures that this name respects the Java naming convention because imposes that the package name starts in lowercase, while at line 4 we make the function name start in uppercase.

5.4 Moving import outside modules

As already said, in Yarel imports had to be done inside modules. This caused the syntax to be unhandy and unintuitive. For this reason imports have been

moved outside modules.
So, grammar has been updated as follows:

```
Model:
    imports += Import*
    'module' name = ID '{' elements += Element* '}';
Element:
    Declaration | Definition;
Import:
    'import' importedNamespace =
        QualifiedNameWithWildcard;
QualifiedNameWithWildcard:
    ID ('.' ID)* '.' (ID | '*');
```

In this new grammar, the production rule for the model, that is the program, requires the user to do a series of imports (that can be empty) before implementing the module. Beside from the grammar, no other change to the compiler was needed.

5.5 Validation rules

Validation rules are a built-in mechanism of Xtext. As explained in [6] chapter 4, they allow to introduce new constraints on the language. These contstraints do not depend on the syntax, and cannot be checked by the parser.

Yarel lacked from some validation rules, without which the Java code resulting from the compilation could not work.

Below the introduced one will be shown and explained.

5.5.1 Existence of the imported module

Yarel did not check whether the imported module existed or not. It only checked if its name respected the syntax, that is the name of the module had to follow the following regex:

```
QualifiedNameWithWildcard: ID ('.' ID)* '.' (ID | '*');
```

Obviously this check was not enough, because it did not give any feedback to the programmer in case he mistook the name of the module or the function to import.

For this reason the following validation rule has been introduced:

```
@Check(CheckType::NORMAL)
def checkImport(Import impt){
    val importedModule =
```

```
4           impt.visibleModules
5               .findFirst[mod |
6                   mod.name.equals(impt.importedModule)
7               ]
8               //should be only one
9       if(importedModule === null){
10          //check if the module exist
11          error(
12              ''' '<<impt.importedModule>>' cannot be
13                  resolved as a module''',
14              YarelPackage::eINSTANCE.import_ImportedNamespace
    ,
15              ERROR_IMPORT
16       )
17      }
18      else{
19         //Check that the imported function
20         //is declared in the imported module
21         val importedFunction = impt.importedFunction
22       if(importedFunction != '*'){
23          //the wildcard is not used
24          if(!importedModule.declarations
25                              .map[name]
26                              .contains(importedFunction)){
27              //check if the imported module contain the
    function
28             error(
29             "'" + importedModule.name + "'" +
30                 " does not declare function: " +
31                 "'" + importedFunction + "'",
32             YarelPackage::eINSTANCE.import_ImportedNamespace,
33             ERROR_IMPORT)
34         }
35       }
36     }
37  }
```

Listing 5.6: checkImport rule

We use the statement *@Check(CheckType::NORMAL)]* to specify that the check must be done when the file is saved, this because it may be costly. At line 4 we check if the classpath contains a module with the same name as the one from which we are trying to import a function.
Through the call to **`impt.visibleModules`** we get all the modules that we can see from the import statement.
While with *findFirst[mod | mod.name.equals(impt.importedModule)]*[5] we get the first module that has the same name of the one from which we are importing one or more function. If the result of this operation is null, then the class path does not contain the module and the compiler throws an error. Otherwise the compiler goes on and checks the existence of the function inside the module.

[5]This function takes a lambda expression as a parameter

If we are using a wildcard no check is needed. If not, at line 24 we verify whether the module from which we are importing, contains a function that has the same name as the one we are trying to import. If not present the compiler throws another error.

5.5.2 Existence of duplicate modules

Yarel did not check if in a classpath two modules with the same name coexisted. This is essential. Without this check, there will not be the univocity of module names and therefore of the qualified names. Hence it would be impossible to always distinguish two functions with the same names, declared in two different modules. To solve this problem the following validation rule has been introduced:

```
1  @Check(CheckType::NORMAL)
2  def checkModuleDuplicate(Model currentModule){
3      if(currentModule.visibleModules
4                          .exists[mod |
5                              mod != currentModule
6                              && mod.name == currentModule.name
7                          ]
8      ){
9          error(
10             '''The module '<<currentModule.name>>'
11                 is already defined''',
12             YarelPackage::eINSTANCE.model_Name,
13             ERROR_DUPLICATE_MODULE
14         )
15     }
16 }
```

Listing 5.7: checkModuleDuplicate rule

This check is also costly and so must be done when the file is saved.
The actual check is made at line 3. This statement returns true if and only if, between the visible modules there is a module that is different from the one that's being validated, but has the same name. If true is returned, then the compiler throws an error.

5.5.3 Definitions count

The last validation rule is not strictly related to the import mechanism, still it was needed.
In fact, there was no check whether a function had multiple definitions or did not have any. So this rule has been introduced:

```
1  @Check
2  def checkOneDefinition(Declaration decl){
```

```
3     val currentModule = decl.getContainerOfType(typeof(Model
      ))
4     var count = 0
5     var i = 0
6     while(count < 2 && i < currentModule.definitions.size){
7         //count the number of definition
8     if(currentModule.definitions.get(i).declarationName ==
      decl) count++
9         i++
10    }
11    if(count >= 2){
12        //check if the declaration has multiple definition
13        error(
14            '''The declared function '<<decl.name>>'
15            has multiple definitions''',
16            YarelPackage::eINSTANCE.declaration_Name,
17            ERROR_INVALID_DEFINITION_COUNT
18        )
19    }
20    else if(count == 0){
21        //check if the declaration has no definition
22        error(
23            '''The declared function '<<decl.name>>'
24            has no definition''',
25            YarelPackage::eINSTANCE.declaration_Name,
26            ERROR_INVALID_DEFINITION_COUNT
27        )
28    }
29    //else noError
30 }
```

Listing 5.8: checkOneDefinition rule

At line 3 we assign the module where the declaration that is being validated is made, to the variable **`currentModule`**. Inside the while cycle we count how many definitions there are for the declared function.
If there are two or more definitions or there is no one, the compiler throws an error, otherwise it goes on.

5.6 Definitions scope

The scoping mechanism of Xtext, described in [6] chapter 10, allows to define for each context, the language objects a symbol can refers to and with which name.

In Yarel, when a function f was imported from a module mod1 in a module mod2, beside from using f inside other functions it was possible to redefine it in mod2, even if no the declaration of f existed in mod2.
This was due the fact that when an import was made, the imported function names were added to the scope of the context in which the function to define is being chosen.

To avoid this problem, the default scope implementation is overwritten by the following one:

```
1  class YarelScopeProvider extends AbstractYarelScopeProvider
       {
2      @Inject extension YarelUtils
3      /*
4       * Define the scope of the Definition.
5       * In this way the user can define only
6       * the function that are declared in the
7       * same module where the definition is made
8       */
9      override getScope(EObject context, EReference reference)
       {
10         if(reference ==
11             YarelPackage::eINSTANCE
12                         .definition_DeclarationName){
13             if(context instanceof Definition){
14                 val dclFuns = context.getContainerOfType(
15                                                 typeof(Model))
16                                             .declarations
17                 return Scopes::scopeFor(dclFuns)
18             }
19         }
20         return super.getScope(context, reference)
21     }
22 }
```

Listing 5.9: Scope redefinition

We change the default scope by realizing a more specific implementation of *AbstractYarelScopeProvider* and by overriding the **`getScope`** method.

This must be done according to Xtext where, to change the scope, we must specify the feature (**`reference`**) for which we want to declare a particular scope and the **`context`** in which that feature will have that scope.

In our case, we want to modify the scope for the symbol that will refer to a declaration in the context in which a definition is being made. We specify this at lines 10 and 12.

Finally through the *return Scopes::scopeFor(dclFUns)* statement, we say that, in the specified context, the scope is made up of all and only names of function declared in the same module in which the definition is made.

In this way it is no longer possible to define functions that do not belong to the module, i.e., when a definition is being made, it is not possible to refer to functions declared outside the module.

5.7 The index class

Several times, it was necessary to get all the visible modules from a certain point in the Yarel code.
In order to realize this request, the following utility function was realized:

```
def getVisibleModules(EObject o){
    val index = rdp.getResourceDescriptions(o.eResource)
    val rd = index.getResourceDescription(o.eResource.URI)
    cm.getVisibleContainers(rd, index)
      .map[container |
        container
        .getExportedObjectsByType(YarelPackage::eINSTANCE
                                                  .model)
      ]
      .flatten
      .map[mod |
        var modObj = mod.EObjectOrProxy
        //the result could be a proxy
        if(modObj.eIsProxy){
            modObj = o.eResource
                      .resourceSet
                      .getEObject(mod.EObjectURI, true)
            //resolve the proxy
        }
        modObj as Model
    ]
}
```

Listing 5.10: getVisibleModules utility function

The function gets all the modules that are visible from the object **o**[6]. It works in the following way: it access the index, which stores a reference to all the objects in all the resources. In the Yarel case it has a reference to all the modules, declared functions, function definition, etc. Once it get the index, it recovers from this all the container [7] that are visible from **o**. Afterward it filter all the containers that are modules.

Due to efficiency reasons, the index does not store a reference to the real object but to their *IEObjectDescription*, which are a proxy for the objects and that have a URI that can be used to find the object and load it into memory as soon as needed. For this reason, after all the objects that represent the modules have been obtained, the function verifies whether they are proxy or not, to eventually resolve them. Finally the function can return the list of all the modules that are visible from the object **o**.

[6] In this chapter when we talk about object, we do not talk about any Java object, but about *EObject*. Every EObject represents an instance of language elements and contains additional info about them. (see Chapter 4.1)

[7] A container is a object that contains other objects. In Yarel modules are an example of containers

Building an example

In this chapter we will build an example step by step. This example will not only show how the new features works, but will also give an overview of Yarel as a DSL (*Domain-specific-language*). The full example can be found at [4].

6.1 Create a new Yarel project

Since Xtext is an Eclipse plug-in, in order to use the Yarel compiler we must have Eclipse and Xtext installed (which can be found at [5]). We must then download the Yarel compiler (that can be found at [1]).

We can now launch the Yarel-IDE by running the **`org.di.unito.yarel`** project as an Eclipse Application.

Once the new Eclipse istance is launched, we must create a new Java Project (remember that the outputs of the Yarel compiler are Java classes). We can then create our first module in the file "mod1.rl" (rl is the extension of the Yarel files). As soon as this file is created we will be asked to add the Xtext nature to the language. We accept that in order to make Yarel-IDE correctly work.

6.2 Yarel modules examples

Beside from creating the file for the module mod1, we also create two files for module mod2 and mod3. This section will show the content of these modules and through them will take a look at the features introduced in the previous chapter, like qualified names, validation rules and scoping.

```
1 module mod1{
2     dcl to_import : int
3     def to_import := id
```

```
4
5       dcl ambiguous_name : int
6       def ambiguous_name := id
7   }
```

Listing 6.1: mod1.rl

There is nothing to say about mod1. It only declares some functions that will be used in the other modules.

```
1   import mod1.*
2   module mod2{
3       dcl f : int
4       dcl g : int
5       //The function to import declared in mod1 can
6       //be called both with
7       //her fullied qualified name and without
8       def f := to_import
9       def g := mod1.to_import
10
11      //The function ambiguous_name declared in mod1
12      //can be overrided
13      dcl ambiguous_name : int
14      def ambiguous_name := neg
15  }
```

Listing 6.2: mod2.rl

In the mod2.rl file, according to the new syntax, before the module definition we import all the mod1 functions. For this reason inside the mod2 functions definitions we can call all the mod1 functions by using their simple names. This is shown at line 8, where there is a call to the function **`to_import`** declared in mod1 made with the plain name of the function. However this is not mandatory as the call could also be made by using the qualified name, as shown at line 9.

We then declare a function that has the same name as a function declared in mod1, that is **`ambiguous_name`**. This way we show that a module can redeclare an imported function.

```
1   import mod1.ambiguous_name
2   import mod2.ambiguous_name
3   module mod3{
4       dcl f : int
5       //the function to_import declared in mod1
6       //can be used with her qualified name
7       //even though the function is not imported
8       def f := mod1.to_import
9
10      dcl g : int
11      dcl h : int
12
```

```
13      //the ambiguity of the function ambiguous_name
14      //is resolved using the qualified name
15      def g := mod1.ambiguous_name
16      def h := mod2.ambiguous_name
17  }
```

Listing 6.3: mod3.rl

In mod3.rl before defining the module we import two functions with the same name: **ambiguous_name**. At lines 15 and 16 we can see that we can distinguish them by using their qualified name. If this name is not used, the compiler throws the following error: "Couldn't resolve reference to Declaration 'ambiguous_name'". As shown in the problem view in figure 6.1.

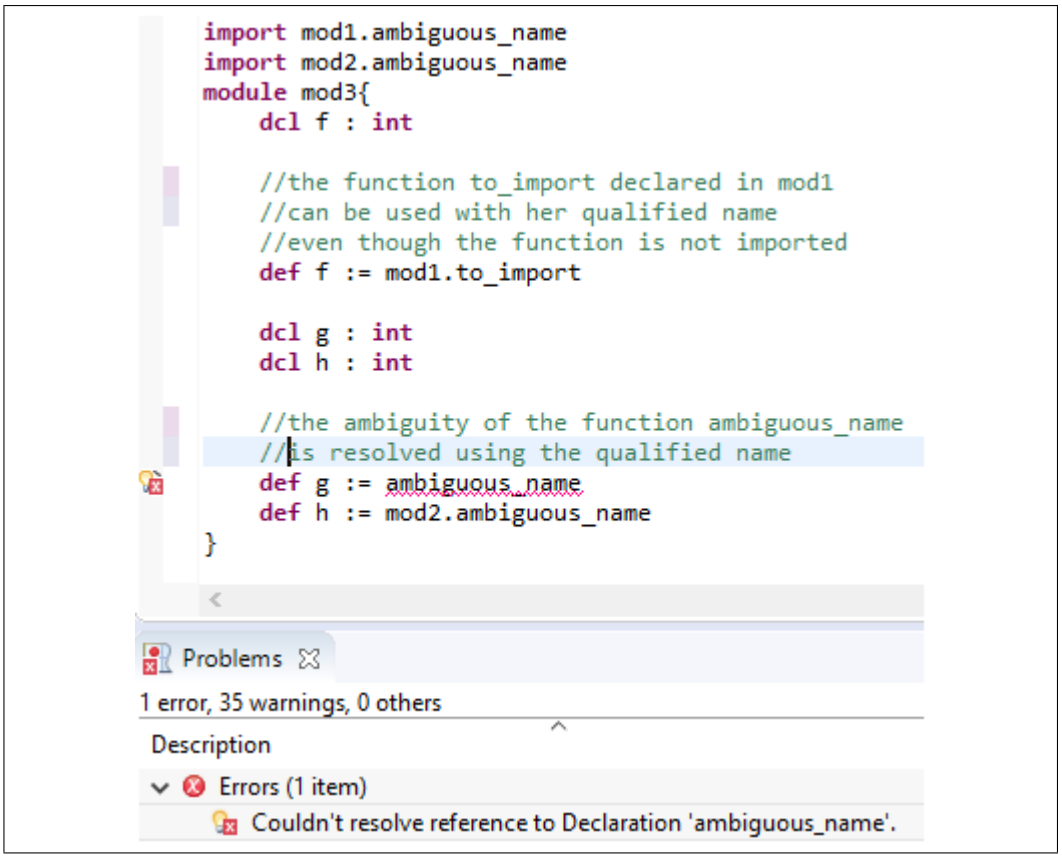

Figure 6.1: mod3.rl

Thanks to the qualified name, in this module definition we can also see that a call to a function declared in an outer module can be made even though the function is not imported. This is done in the definition at line 8 (Listing 6.3), where the call to the function **to_import** is made through the qualified name of the function.

A broken module

Since all the previous modules correctly compiled, we did not have a glimpse at how the new validation rules and scope mechanism work. The next example is therefore a module that does not compile. This module called **`brokenMod`** is in figure 6.2

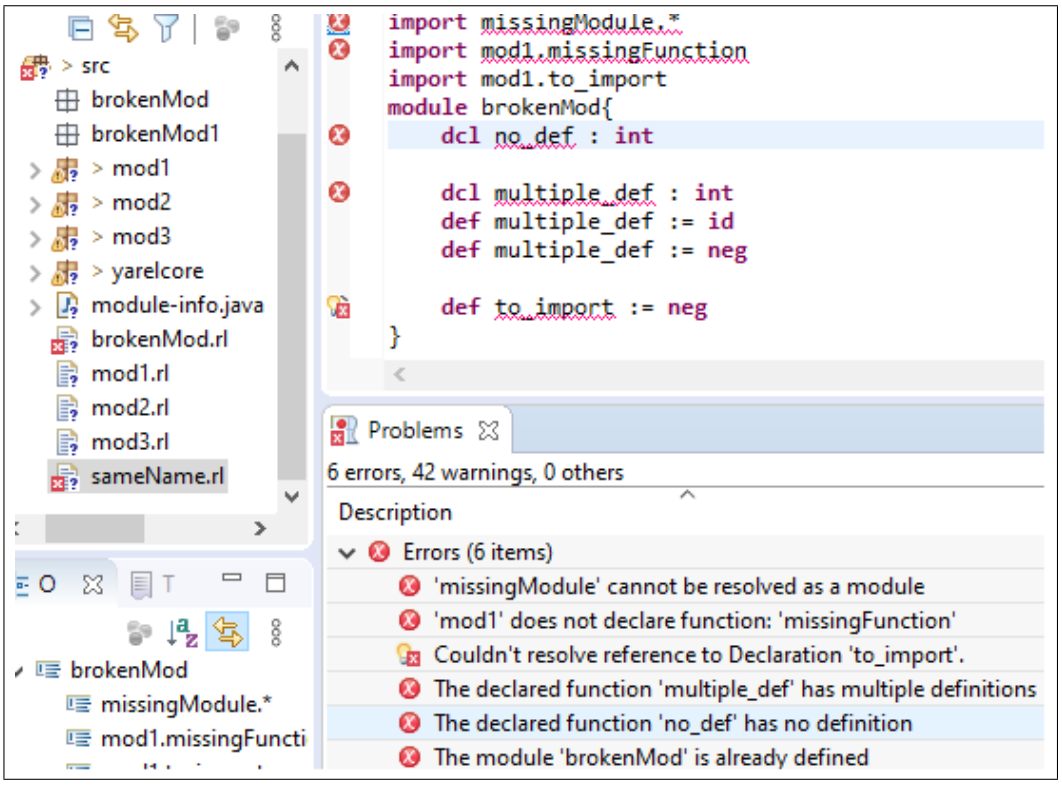

Figure 6.2: brokenMod.rl

As the many highlights and the problem view shows, this Yarel module does not compile due to different errors. These errors are reported by the new validation rules.

Starting from the imports, we can see from the error messages that they are trying to import from a non existing module or they are trying to import a missing function. As already explained, in this way the programmer is immediately warned in case he misspelled a function or module name.

Then we are notified that a module with the same name already exists (it is defined in the "sameName.rl" file), so the module does not compile because otherwise two modules with the same name would coexist in the same classpath.

Moreover the compiler notifies that it cannot correctly compile the two functions **`no_def`** and **`multiple_def`**, because the first one has no definition, while the latter has two of them. This shows that the **`checkOneDefinition`** rule actually works.

Finally we can see that the new scope for the declaration is correct, because when we try to define the imported function **`to_import`** (that is not redeclared in the module) the compiler warns us that it cannot resolve the reference.

6.3 Output Java Code

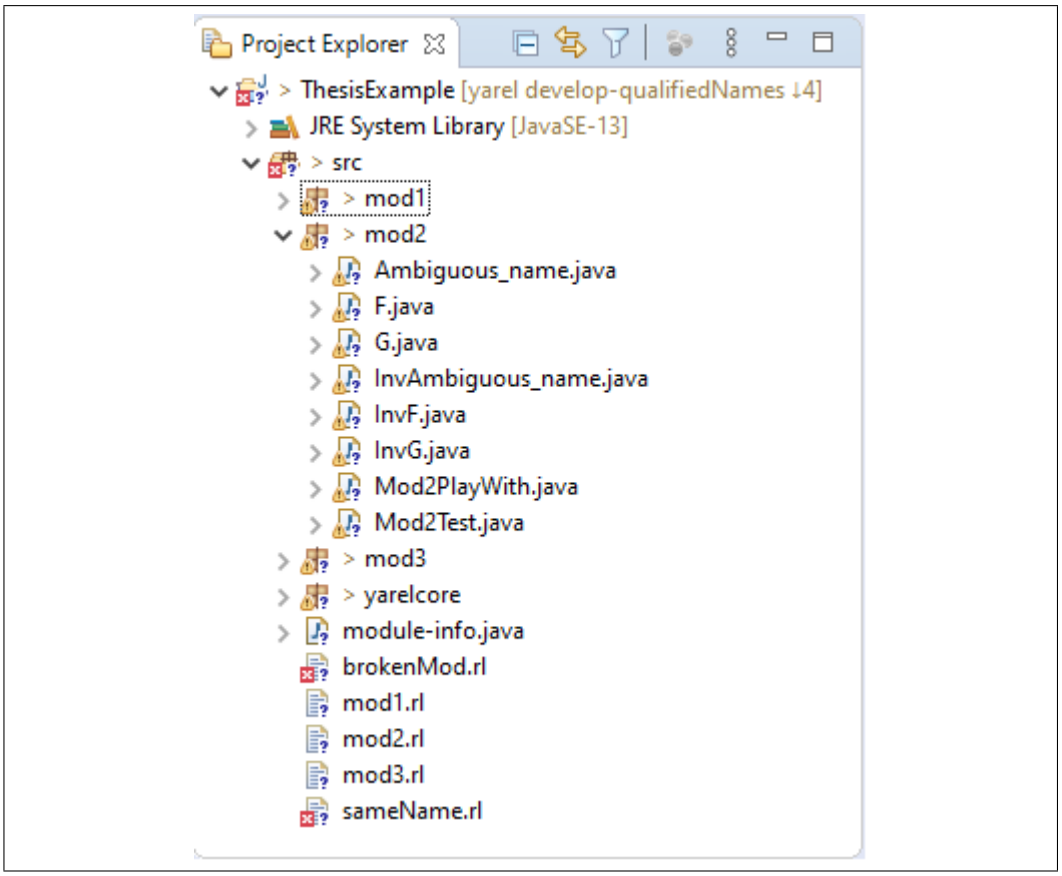

Figure 6.3: Project Explorer

After having shown how the qualified names can be used and how the new validation rules and scoping mechanism work, we will analyze the output Java code resulting from the compilation in order to understand how the

changes to the compiler, exposed in Chapters 5.2 and 5.3, affect the generated code.

First of all by taking a look to Figure 6.3, that represents the Project Explorer of this example, we can see that all the generated package folders, that correspond to a module, start in lowercase. Furthermore we can see that all the class files in mod2, which represent the module functions, start in uppercase. We can therefore say that, at least for filenames, naming conventions are respected.

Listing 6.4 shows the compiled code that results from the compilation of mod2.g (reported in listing 6.2). This function will be taken as an example.

```
1  package mod2;
2  import java.util.Arrays;
3  import java.lang.Math;
4  import yarelcore.*;
5  public class G implements RPP {
6      public G() { }
7      RPP function = new mod1.To_import();
8      private final int a = function.getA();
9      public int[] b(int[] x) {
10         return this.function.b(x);
11     }
12     public int getA() { return this.a; }
13 }
```

Listing 6.4: mod2/G.java

In the first place we notice that the package statement is completed by a name starting in lowercase *"mod2"*. Same for the *"yarelcore.*"* import statement. The package naming convention are therefore respected.

With a closer look we can see that nothing is imported from the mod1 package, even though the Yarel module mod2 imports all the functions from mod1. This is because since each object that represents a Yarel function is created by using the qualified name of its class, there is no need to actually import the class.

The class is generated with a name starting in uppercase *"G"* and like all the other classes that represent a function, implements the RPP interface.

We now analyze the body of the class. Recall that in Yarel **g** is defined in the following way:

```
1      dcl g : int
2      def g := mod1.to_import
```

The function is then compiled as a simple function call. Hence it only has an RPP **function** field which contains the called function **mod1.to_import**. In order to avoid any name ambiguity this object is created by using the qualified name of the class.

By definition the ariety of **g** will be the same as the one of **to_import**, that is 1. So the compiled **a** field will contain the same value as the **function .a** field. Recall that **function** is a reference to an object of type **mod1. to_import**.

Finally there is the **b** method, that will be used when we want to apply the function **g** to an integer x. Since this function is composed by the only call to **to_import**, the **b** method will have the only responsibility to delegate the call to **function.b()** which will actually apply the **to_import** function to x.

Conclusion

7.1 Testing

Chapter 6 showed how our updates work. However this example does not demonstrate the validity of the changes. In order to have a better warranty that these changes work (not a formal proof), a *Test Driven Development* (TDD) has been followed: after every change to the compiler a tests suite is executed in order to confirm that the change satisfies the goal, and that it does not negatively impact the functioning of the compiler.

These tests has been written using the *Junit 4* framework. They were grouped in 5 different classes: **YarelGenerationTest** (which tests the code generator), **YarelIndextTest** (which tests the index), **YarelParsingTest** (which tests the parser), **YarelScopeTest** (which tests the scope mechanism) and **YarelValidationTest** (which tests the Validator).

7.2 Yarel current state and future works

As a result of this work Yarel becomes much more module oriented. In fact, thanks to the new grammar and validation rules, the import system can be used in a easier and more intuitive way. Furthermore, since it is possible to remove any function name ambiguity by using the qualified names, it is safer to simultaneously import multiple functions from different modules. Qualified names can also be used to easily integrate modules between each other. In fact they allow to use functions declared in outer modules without having to explicitly import them.

However Yarel still have some problems regards its usability.

Concerning its completeness, when this work was realized the only type in Yarel were the Integer. Although this fact creates continuity with RPP, it

reduces Yarel's expressivity. Furthermore recursion has not yet been implemented; however an issue on Github has already been opened.

Even though thanks to this work Yarel its handier to use, it still has some flaws. First of all, since Xtext is an Eclipse plug-in and since Yarel has been implemented in Xtext, the Yarel compiler must be launched from Eclipse. This is unhandy and bonds the use of Yarel to Eclipse. In order to avoid this problem, a stand alone command-line compiler can be created, so that a Yarel file can be compiled without having to use Eclipse (see [6, Chapter 5] for more details).

Another problem regards the low number of basic functions. In fact beside from the primitive functions (id, inc, dec, etc) Yarel does not have any other default functions, hence another way to make Yarel handier, is to introduce a library of functions that are imported by default in each Yarel module. [6, Chapter 10] "Providing a library" explains how to do this.

Finally, we already said that Xtext allows to implement various IDE integrations such as: syntax highlighting, error markers, quickfixes, etc. However none of them has been implemented in Yarel. [6, Chapter 6] shows how to customize some of these aspects.

Even though Yarel lacks of all these functionality it still succeeds in its original purpose, that is be an implementation of the RPP class, i.e., be a reversible language complete with respect of PRF. All these functionality could be instead the subjects of future works.

Bibliography

[1] Yarel project link: https://github.com/yarel-di/yarel.

[2] Xtext documentation link: https://www.eclipse.org/Xtext/documentation/index.html.

[3] Xtend documentation link: https://www.eclipse.org/xtend/documentation/index.html.

[4] Yarel example link: https://github.com/yarel-di/yarel/tree/master/yarel-import-examples/YarelImportExample.

[5] Xtext framework link: https://www.eclipse.org/Xtext/.

[6] Lorenzo Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. PACKT, second edition.

[7] Luca Paolini, Mauro Piccolo, and Luca Roversi. A class of Recursive Permutations which is Primitive Recursive complete, June 2016. Sotto considerazione per pubblicazione. Disponibile su http://www.di.unito.it/~paolini/papers/2017rpp.pdf.

[8] Kalyan S. Perumalla. *Introduction to Reversible Computing*. Chapman & Hall/CRC Computational Science. Taylor & Francis, 2013.