Christian-Albrechts-University Kiel
Department of Computer Science
Programming Languages and Compiler Construction

Diploma Thesis

An Eclipse-Based Integrated Development Environment for Curry

Marian Palkus (map@informatik.uni-kiel.de)

December 4, 2012

Advised by:
Prof. Dr. Michael Hanus

Abstract

We present the *Curry IDE*, an integrated development environment (IDE) for the declarative multi-paradigm language Curry. The Curry IDE provides various features known from modern IDEs including syntax highlighting, code completion, code navigation, and error reporting. Any Curry runtime system can be used to execute Curry programs in an integrated console view. The Curry IDE is restricted to support the *haskell* case-mode and does not provide type checking. Curry analyses are loaded dynamically from an external tool and can be executed out of the Curry IDE. Additionally, we present a generic interface that allows developing visualizations for analyses which can be integrated into the Curry IDE at runtime as well. The Curry IDE is based on *Eclipse* and implemented using the *Xtext* framework.

Contents

List of Figures ix

List of Tables xi

Listings xiii

1 Introduction 1

2 Curry 3
- 2.1 Main Features of Curry . . . 3
  - 2.1.1 Overview . . . 3
  - 2.1.2 Predefined Types and Operations . . . 4
  - 2.1.3 Functions . . . 4
  - 2.1.4 Pattern Matching . . . 5
  - 2.1.5 Conditions . . . 5
  - 2.1.6 As-Patterns . . . 6
  - 2.1.7 Local Declarations . . . 6
  - 2.1.8 Constraints and Equality . . . 7
  - 2.1.9 Higher-order Features . . . 7
  - 2.1.10 Layout . . . 8

3 Requirements and Specification 11
- 3.1 Existing Tools . . . 11
- 3.2 The Curry IDE . . . 15
  - 3.2.1 General Requirements . . . 15
  - 3.2.2 Curry-Specific Requirements . . . 15
- 3.3 Integration of Curry Analysis Tools . . . 18
  - 3.3.1 Curry Analyses . . . 19
  - 3.3.2 Protocol . . . 19
  - 3.3.3 Integration . . . 20

4 Foundations 21
- 4.1 Theoretical Foundations . . . 21
  - 4.1.1 Language Recognition Workflow . . . 21

4.1.2 Lexical Analysis . . . 21
4.1.3 Syntactic Analysis . . . 22
4.1.4 EBNF . . . 23
4.1.5 Recursive-Descent Parsing . . . 24
4.1.6 Parser Generators . . . 28
4.1.7 Syntactic and Semantic Predicates . . . 28
4.2 ANTLR - A Parser Generator . . . 28
4.2.1 ANTLR-Works . . . 28
4.3 Eclipse Rich Client Platform . . . 29
4.3.1 Eclipse . . . 29
4.3.2 Eclipse RCP . . . 29
4.4 Xtext . . . 30
4.4.1 Generation Workflow . . . 32
4.4.2 Customization . . . 32
4.4.3 The Grammar Language . . . 33
4.4.4 Data Model . . . 34
4.4.5 The generated IDE . . . 38

5 The Curry IDE **39**
5.1 Why use Xtext? . . . 40
5.2 The Grammar . . . 40
5.2.1 Lexicon . . . 41
5.2.2 Basic Xtext Grammar . . . 42
5.2.3 Resolving Grammar Issues . . . 48
5.3 The Layout . . . 58
5.4 Linking . . . 62
5.4.1 Scoping . . . 64
5.4.2 Implementation . . . 67
5.4.3 Import- and Export Mechanism . . . 69
5.4.4 External Paths and Libraries . . . 70
5.5 Finishing . . . 72
5.5.1 Outline . . . 72
5.5.2 Labeling and Documentation . . . 72
5.5.3 Additional validation . . . 73
5.5.4 Curry Perspective . . . 74
5.6 Testing . . . 76

6 Integration of Curry Analysis Tools **79**
6.1 Concept . . . 79
6.2 Implementation . . . 80
6.2.1 The Curry Analysis SDK . . . 80

6.2.2 Curry IDE Integration . . . 81
6.3 A Sample Curry Analysis Visualization . . . 84
6.3.1 Prerequisite . . . 84
6.3.2 Development . . . 84
6.3.3 Deployment . . . 86

7 Conclusions 87
7.1 Summary . . . 87
7.2 Discussion . . . 87
7.3 Future Work . . . 88

Appendices 88

A Complete Xtext Grammar 91

B Installation Guide 99
B.1 Prerequisite . . . 99
B.2 Installation . . . 100
B.2.1 Step 1 . . . 100
B.2.2 Step 2 . . . 100
B.2.3 Step 3 . . . 102
B.3 Update . . . 104

C Project structure 105
C.1 Overview . . . 105
C.1.1 de.kiel.uni.informatik.ps.curry.CurryIDE . . . 105
C.1.2 de.kiel.uni.informatik.ps.curry.CurryIDE.ui . . . 106
C.1.3 de.kiel.uni.informatik.ps.curry.CurryIDE.analysis . . . 107
C.1.4 de.kiel.uni.informatik.ps.curry.CurryIDE.sdk . . . 107
C.1.5 CurryUpdateSite . . . 107

Declaration 111

List of Figures

3.1 Curry development process 14

4.1 Language recognition workflow 22
4.2 Xtext generation process 31
4.3 SimpleCurry data model 37

5.1 Grammar rule diagram: identifier issue 49
5.2 Syntax trees of ambiguous grammar 54
5.3 Layout example 59
5.4 TokenSource interface 60
5.5 Layout algorithm outline 63
5.6 Module scope example 64
5.7 Local definition scope example 66
5.8 Do-expression scope example 66
5.9 List comprehension scope example 67

6.1 Curry analysis visualizations interface 81
6.2 Overview: Curry Analysis Tool integration 82
6.3 Context menu for Curry analyses 84
6.4 Example of a Curry visualization extension 85

B.1 Installation guide: Step 1 100
B.2 Installation guide: Step 2 101
B.3 Installation guide: Step 3 102
B.4 Installation guide: Security warning 103
B.5 Installation guide: New Curry project 103

List of Tables

5.1 Scoping: search strategies 68

Listings

4.1 Sample SimpleCurry-program. 33

5.1 The remaining rule definitions of the Xtext-grammar. 45
5.2 Modified rules *Expr* and *FunctExpr* 48
5.3 The different meanings of a dot in Curry. 49
5.4 Compact definition of the rules *SimpleTypeExpr*, *SimplePat*, and *BasicExpr*. 52
5.5 Modified CurryParser. 59

1 Introduction

In the last decades, the computer has become an essential part of modern societies. More and more tasks are delegated to computer systems for reason of speed, efficiency, security, costs, and other advantages over different solutions. To satisfy all these expectations, the development process of such systems has to be efficient and affordable. Generally, they consist of hardware which is controlled by software. Hence, the software plays a key role for the success of computer systems.

At the beginning of every software project, a programming languages has to be chosen that should be used for the development. Programming languages can be used to specify instructions which are executed by a machine, particularly a computer. They usually abstract from machine instructions to ease the work of developers. This level of abstraction differs from one language to another. High level programming languages promise a lot of advantages over programming languages of lower levels, including the following: Shortened development time, higher readability and maintainability of code, and less error-proneness. Hence, a basic step for every software project is to choose the right programming language(s) to use. Generally, this choice is significantly influenced and restricted by particular project conditions, especially the target platform(s). Another factor is the existing tooling for a programming language. There are often various tools for widely used programming languages which simplify the use of the corresponding language and raise productivity of software developers, often they are combined into an *integrated development environment* (IDE). Additionally, the need for tool support rises with the number of developers working together on large software projects. Modern IDEs simplify the work of developers in various ways including fast navigation through code, code completion, and instant error notification.

This thesis deals with tool support for *Curry*. Curry is a declarative programming language aiming to combine functional and logic programming paradigms. It is developed by an international initiative intended to provide common platform for research, teaching, and application of integrated functional logic languages. Curry is a high level language and, hence, may be a good choice for software projects. Chapter 3 lists the existing tools for Curry and identifies their shortcomings compared to modern IDEs. The result shows that the tooling for Curry can and, having the importance of tool support in mind, should be improved. The objective of this thesis is to provide a modern IDE for Curry oriented towards the Eclipse Java IDE[1]. It will be called *Curry IDE* in the

[1] The Eclipse Java IDE is described in Chapter 3

following. An additional objective is the integration of existing Curry analysis tools.

The development of a modern IDE is quite complex so that this thesis is meant to build a usable IDE that can be enhanced in the future. It is necessary to restrict the recognition strength of the IDE slightly and to exclude type checking to keep the programming effort within reasonable limits.

This thesis is structured as follows: In Chapter 2, the programming language Curry is introduced. Chapter 3 provides an overview of existing tooling for Curry, their shortcomings compared to the Eclipse Java IDE, and defines the specification for the Curry IDE. In Section 3.3 the integration of Curry analysis tools is specified. In Chapter 4 basic knowledge for this thesis is introduced, including theoretical foundations in Section 4.1 and the technologies applied in Sections 4.2, 4.3, and 4.4. Chapter 5 describes the concept and implementation of the Curry IDE as well as encountered problems and corresponding solutions. In Chapter 6, the realization of the Curry analysis tool integration is documented. Finally, Chapter 7 evaluates the resulting Curry IDE and lists its limitations regarding the specifications.

2 Curry

Curry is a general-purpose declarative programming language that integrates functional with logic programming. Therefore, Curry seamlessly combines the key features of functional, logical, and concurrent programming.

- *Functional Programming*: nested expressions, lazy evaluation, higher-order functions
- *Logical Programming*: logical variables, partial data structures, built-in search
- *Concurrent Programming*: concurrent evaluation of constraints with synchronization on logical variables

Moreover, Curry provides additional features in comparison to the pure languages (compared to functional programming: search, computing with partial information; compared to logic programming: more efficient evaluation due to the deterministic evaluation of functions).

The development of Curry is an international initiative intended to provide a common platform for research, teaching and application of integrated functional logic languages. The language specification including its syntax and operational semantics can be found in the *Curry Report* [Han12].

There are different implementations of Curry available, some of these will be mentioned in Chapter 3. In this thesis, we will use *PAKCS* (Portland Aachen Kiel Curry System) [HAB+12] as the default implementation.

There is a tutorial [Ant07] that gives a good introduction to programming in Curry, further, the Curry Report [Han12] describes every feature of the language in detail. We will summarize Curry's key features and give a short introduction by using examples from these sources in the following.

2.1 Main Features of Curry

2.1.1 Overview

The major elements of a Curry program are *expressions*, *functions* and *data structures*:

- An *expression* is either a symbol or literal value or the application of an expression to another expression. Symbols and literal values constitute the most elementary expressions, for example, numbers and the Boolean symbols "True" and "False".

- A *function* defines a computation similar to an expression, with the difference that it is named and, hence, can be executed over and over again in the same program with possibly different parameters. Moreover, a function provides a *procedural abstraction*. Rather than coding a computation by means of a possibly complicated expression, you can factor out portions of this computation and abstract them by their names.

- A *data structure* is a way to organize data. The way in which information is organized may ease some computations, such as retrieving portions of information, and is intimately related, through pattern matching, to the way in which functions are coded.

2.1.2 Predefined Types and Operations

A *type* is a set of values. Ubiquitous types, such as integers or characters, are predefined in Curry. We call these types *built-in*. The availability of built-in types as well as their characteristics may depend on a specific implementation of Curry. As mentioned above, we use PAKCS to execute Curry programs which provides the module "Prelude" containing all predefinitions. In PAKCS, the *buit-in* types *Integer*, *Floating Point Numbers*, *Boolean*, *Character*, *String*, *List*, *Tuples*, *Success*, and *Unit* are available. The details of these types can be found in the PAKCS User Manual [HAB+12].

Additionally, there are frequently-used functions and infix operators predefined in Curry including *Boolean equality*, *Constrained equality*, *Boolean conjunction*, *Boolean disjunction*, *Parallel conjunction*, and *Constrained expression*. Curry predefines many more functions and operations, a complete list can be found both in the Curry Report [Han12] and the "Prelude".

2.1.3 Functions

In Curry, functions abstract functions in the mathematical sense by being a device that takes arguments and returns a result. The result is obtained by evaluating an expression which generally involves the function's arguments. For instance, the following function computes the *square* of a number:

```
1   square x = x * x
```

The symbol "square" is the name or *identifier* of the function and the symbol "x" is the function's *argument*. The above declarations referred to as an equation defining

a function. Functions can also be *anonymous*, i.e., without a name. An anonymous function definition has the following structure:

```
1  \args -> expression
```

2.1.4 Pattern Matching

The definition of a function can be broken into several equations. A single equation would suffice many cases. However, several equations allow a definition style, called *pattern matching*, which is easier to code and understand. This feature allows a function to dispatch the expression to be returned depending on the values of its arguments.

For instance, the following definition:

```
1  not x = if x == True then False else True
```

can be rewritten using pattern matching as follows:

```
1  not True    = False
2  not False   = True
```

which seems to be a lot more readable.

2.1.5 Conditions

Each equation defining a function can include one or more *conditions*. For Boolean conditions, an equation has the following general structure:

```
1  functionId arg1, ..., argm | cond1 = expr1
2                             | ... = ...
3                             | condn = exprn
```

A condition is tested after binding the arguments of a call to the corresponding arguments in the left-hand side of the equation. The function is applied to the arguments only if the condition holds. Each condition $expr_i$ is an expression of type *Boolean*. The conditions are tested in their textual order so that the first right-hand side with a condition evaluable to True is taken. Furthermore, the last condition can be "otherwise" which is equivalent to True, i.e., it is always taken, if none of the preceding conditions is True. The following example shows a definition of the maximum of two numbers:

```
1  max x y | x < y     = y
2          | otherwise = x
```

2.1.6 As-Patterns

The patterns in a defining equation, i.e., the arguments of the left-hand sides, are required to be data terms without multiple occurrences of variables. A pattern denotes some part of a structure of the actual argument. The as-pattern allows to reuse this structure in the right-hand side of the defining equation by identifying this structure by a variable. An as-pattern has the form *v@pat* where the variable *v* identifies the structure matched by the pattern *pat*. For instance,

```
1  dropFalse (False:ys)  = ys
2  dropFalse xs@(True:_) = xs
```

is equivalent to

```
1  dropFalse xs = dropFalse´ xs
2   where
3    dropFalse´ (False:ys) = ys
4    dropFalse´ (True:_)   = xs
```

Local declarations introduced by the keyword *where* are discussed in the following.

2.1.7 Local Declarations

Since not all auxiliary functions should be globally visible, it is possible to restrict the scope of declared entities. Note that the scope of parameters in function definitions is already restricted since the variables occurring in parameters of the left-hand side are only visible in the corresponding conditions and right-hand sides. The visibility of other entities can be restricted using *let* in expressions or *where* in defining equations. Both keywords introduce a set of local names. The list of local declarations can contain function definitions as well as definitions of constants by pattern matching. There is a small difference regarding visibility and scoping of local definitions in *let*- and *where*-clauses, however, this is discussed later in this thesis in detail. For now, we consider two code snippets to get a first impression of local definitions. For instance, the expression:

```
1  let a = 3 * b
2      b = 6
3  in 4 * a
```

reduces to the value *72*. The function equation:

```
1  sumSquares a b = (square a) + (square b)
2    where square x = x * x
```

can be used to evaluate the expression “sumSquare 3 5” to *34*.

Free Variables

Since Curry is intended to cover functional as well as logic programming paradigms, expressions (or constraints, see Section 2.1.8) might contain free (unbound, uninstantiated) variables. The idea is to compute values for these variables such that the expression is reducible to a data term or that the constraint is solvable. For instance, consider the following definitions:

```
1  mother John     = Christine
2  mother Alice    = Christine
3  mother Andrew   = Alice
```

We can use free variables and constraints for special computations. For instance, a child of *Alice* can be computed by solving the equation "mother x =:= Alice". Similarly, we can compute a grandchild of *Christine* by solving the equation "mother (mother x) =:= Christine" which yields the value "Andrew" for x.

2.1.8 Constraints and Equality

An equation can also have a *constraint* as a condition, which is an expression of the type *Success*. In this case, the constraint is checked for satisfiability in order to apply the equation. Constraints have the form $e_1 =:= e_2$ where e_1 and e_2 are expressions. $e_1 =:= e_2$ is satisfied if both sides are reducible to a same ground data term (cf. [Han12]). This notion of equality is called *strict equality*. As a consequence, if one side is undefined (non-terminating), then the strict equality does not hold. For instance, the equational constraint $[x] =:= [0]$ is satisfiable if the variable x is bound to 0.

Constraints can be combined into a *conjunction* written as $c_1 \& c_2$. The conjunction is interpreted *concurrently*: if the combined constraint $c_1 \& c_2$ should be solved, c_1 and c_2 are solved concurrently. The single constraints c_1 and c_2 can communicate using common variables.

2.1.9 Higher-order Features

Curry is a higher-order language supporting common functional programming techniques by partial function applications and lambda abstractions. A *function application* is denoted by juxtaposition the function and its argument. For instance, the well-known *map* function is defined in Curry by:

```
1  map :: (a->b)-> [a] -> [b]
2  map f []      = []
3  map f (x:xs)  = f x : map f xs
```

2.1.10 Layout

Curry allows to use layout information to define the structure of blocks, similarly to Haskell. Therefore, we define the *indentation* of a symbol as the column number indicating the start of this symbol. The indentation of a line is the indentation of its first symbol [Han12].

The layout rule applies to lists of syntactic entities after the keywords *let*, *where*, *do*, or *of*. The structure (beginning and ending of the list, separation of single entities) of these lists is specified by indentation: the indentation of a list of syntactic entities after *let*, *where*, *do*, or *of* is the indentation of the next symbol following the *let*, *where*, *do*, *of*. Any item of this list starts with the same indentation as the list. Lines with only whitespaces or an indentation greater than the indentation of the list continue the item in its previous line. Lines with an indentation less than the indentation of the list terminates the entire list. Moreover, a list started by *let* is terminated by the keyword *in*.

For the purpose of illustration, in the following example [Han12] we use curly braces ('{ }') to specify the beginning and ending of a block and a colon (';') to separate single entities:

```
1 f x = h x where  {g y = y+1 ; h z = (g z) * 2 }
```

This is written with the layout rules as

```
1 f x = h x where
2   g y = y+1
3   h z = (g z) * 2
```

or also as

```
1 f x = h x where
2   g y = y+1
3   h z = (g z)
4             * 2
```

To avoid an indentation of top-level declarations, the keyword *module* and the end-of-file token are assumed to start in column 0.

Infix Operators

Curry allows the definition and usage of infix operators. For instance, "$1 + 2 * 3 + 4 == x\&\&b$" is a valid expression using the infix operators $+$, *, $==$, and &&. However, there are various ways to interpret this expression depending on the associativity of the single operators. One way to define this expression unambiguously is by using brackets. For

instance, the expressions "$(((1 + 2) * (3 + 4)) == x)\&\&b$" and "$((1 + ((2 * 3) + 4)) == (x\&\&b))$" are unambiguously.

Indeed, we want the * to have a greater associativity than the + for instance to minimize the use of brackets. Therefore, Curry allows to assign an associativity and precedence to each operator by a *fixity declaration*. There are three kinds of associativities, non-, left-, and right-associativity (*infix*, *infixl*, *infixr*) and ten levels of precedence, 0 to 9, where level 0 binds least tightly and level 9 binds most tightly. All fixity declarations must appear at the beginning of a module. Any operator without an explicit fixity declaration is assumed to have the declaration *infixl 9*.

The associativities and precedences for the operators +, *, ==, and && are defined in the prelude, according to those fixity declarations, the expression "$1+2*3+4 == x\&\&b$" is equivalent to "$((((1 + (2 * 3)) + 4) == x)\&\&b)$".

List Comprehension

List comprehensions constitute another special case. They provide a compact notation for lists and have the general form (cf. [Han12]:

```
1  [ e | q1, ..., qk ]
```

where $k \geq 1$ and each q_i is a qualifier that is either

- a *generator* of the form p <- l, where p is a local pattern (i.e., an expression without defined function symbols and without multiple occurrences of the same variable) of type t and l is an expression of type [t], or
- a *guard*, i.e., an expression of type *Bool*.

The variables introduced in a local pattern can be used in subsequent qualifiers and the element description e. Such a list comprehension denotes the list of elements which are the result of evaluating e in the environment produced by the nested and depth-first evaluation of the generators satisfying all guards. For instance the list comprehension

```
1  [ (x,y) | x <- [1,2,3], y <- [4,5] ]
```

denotes the list $[(1, 4), (1, 5), (2, 4), (2, 5), (3, 4), (3, 5)]$.

3 Requirements and Specification

The intention of this paper is to develop a tool to simplify the use of Curry. To accomplish this goal, the current development process and tools of Curry developers are examined in Section 3.1. By comparing the current state with a modern and widely used IDE, like the Eclipse Jave IDE, a set of additional features, which are necessary to enhance the user experience of Curry developers, is identified. These features are used to write an informal specification of the Curry IDE in Section 3.2. The last part, Section 3.3, of the specification describes how existing Curry analyses are integrated into the Curry IDE.

3.1 Existing Tools

This section gives a short overview of the existing tooling for Curry. There are various types of tools available which focus on different tasks within the development process. The set of tasks includes writing, reading, debugging, refactoring, testing, compiling, executing, analyzing, and generating code. The following list shows some tools available[1] at the time of writing:

- *PAKCS*: The Portland Aachen Kiel Curry System is an implementation of Curry jointly developed by the Portland State University, the Aachen University of Technology, and the University of Kiel. PAKCS is an interactive system to develop Curry programs. It contains a lot of additional tools and libraries. PAKCS compiles Curry programs to Prolog programs. [2]
- *MCC*: The Münster Curry Compiler is a native code compiler for Curry which conforms to the Curry report except for committed choice which is not supported. [3]
- KiCS and KiCS2: The Kiel Curry System is a compiler that translates Curry to Haskell. The KiCS2[4] is a new implementation of KiCS.

[1] As listed on `http://www.curry-language.org`, last visited December 3, 2012
[2] `http://www.informatik.uni-kiel.de/~pakcs/`, last visited December 3, 2012
[3] `http://danae.uni-muenster.de/~lux/curry/`, last visited December 3, 2012
[4] `http://www-ps.informatik.uni-kiel.de/kics2/`, last visited December 3, 2012

- *FLVM*: The FLVM[5] is a virtual machine for functional logic computations. Curry programs are compiled to FLVM instructions that are executed by the FLVM interpreter that is implemented in Java. [AHLT05]
- *Sloth*: The Sloth system[6] is a compiler which translates Curry programs to Prolog programs. It is under development at the Technical University of Madrid. The Sloth system has an interactive interface to load programs and evaluate expressions.
- *COOSy*: The Curry Object Observation System is an implementation of a light-weight approach for debugging functional logic programs in Curry by observations. It consists of a library plus a viewing tool. Programmers can annotate their program with observation functions for data structures and functions which may both contain or be applied to logical variables. The parts of observed objects that are evaluated during the execution are recorded in a trace file and can be viewed by means of a pretty printer integrated in a graphical user interface. [BCHH04]
- *CurryBrowser*: CurryBrowser is a generic analysis environment for Curry. It supports browsing through the program code of an application written in Curry, i.e., the main module and all directly or indirectly imported modules. Each module can be shown in different formats (e.g., source code, interface, intermediate code) and, inside each module, various properties of functions defined in this module can be analyzed. In order to support the integration of various program analyses, CurryBrowser has a generic interface to connect local and global analyses implemented in Curry. CurryBrowser is completely implemented in Curry using libraries for GUI programming and meta-programming. [Han06]
- *CurryDoc*: CurryDoc is a documentation tool for declarative programs. It can be used for automatic generation of documentation manuals in HTML format from programs written in Curry. The documentation is generated by combining comments in the source program with information extracted from the program. It extends other tools with a similar goal (e.g., javadoc, lpdoc) by the inclusion of information in the generated documents which has been computed by analyzing the structure and approximating the run-time behavior of the program. [Han02]
- *CurryTest*: CurryTest[7] is a simple tool in the PAKCS distribution to write and run repeatable unit tests. CurryTest simplifies the task of writing test cases for a module and executing them.

[5] `http://web.cecs.pdx.edu/~antoy/homepage/download.html`, last visited December 3, 2012
[6] `http://babel.ls.fi.upm.es/research/Sloth/`, last visited December 3, 2012
[7] `http://www-ps.informatik.uni-kiel.de/currywiki/tools/currytest`, last visited December 3, 2012

- *EasyCheck*: EasyCheck is a library for automated, specification-based testing of Curry programs. It it distributed with the Curry implementation KiCS. The ideas behind EasyCheck are described in [CF08].
- *iCODE*: iCODE (Interactive Curry Observation DEbugger) is a tool to support programmers stepping on the lazy evaluation order of expressions at the source code level. Every executed expression is covered in a layout of the source code and its runtime value can be represented to the user.
- *Emacs plug-in*: Emacs[8] is an extensible, customizable text editor. There is an Emacs plug-in for Curry, which provides syntax coloring.

The following list contains older implementations that are no longer actively maintained.

- *Curry2Java*: Curry2Java is another back end for PAKCS which translates Curry programs to Java programs. Is uses Java threads to implement the concurrent non-deterministic features of Curry.
- *CIDER*: CIDER[9] is a graphical programming and development environment for Curry. CIDER is intended as a platform to integrate various tools for analyzing and debugging Curry programs. CIDER is completely implemented in Curry. Although the graphical debugger contains an interpreter for executing Curry programs, it is mainly intended for visualizing the execution of smaller programs but not for executing large programs. [HK01]

This overview shows that there is good tool support for Curry. However, most tools are standalone applications which leads to a poor user experience. This is also reflected by considering the development process for Curry programs.

Figure 3.1 illustrates the current workflow of Curry developers, which has been determined by consultation with experts. This process includes the use of various tools with different UIs and separate windows. The code is written and modified in Emacs. It is compiled using any of the existing runtime systems, such as PAKCS, KiCS, or MCC. This can be done within the UI of Emacs using a built-in console. If there are errors during compilation, these errors are printed to the console providing the line and position of the first error that occurred. The developer has to navigate manually to that code position to fix it. Usually, these steps are done multiple times. When the code compilation succeeds without errors, its runtime behavior has to be tested. Therefore, it is executed, again, using any available runtime system. If the program does not work

[8] `http://www.gnu.org/software/emacs/`, last visited December 3, 2012
[9] `http://www.informatik.uni-kiel.de/~pakcs/cider/`, last visited December 3, 2012

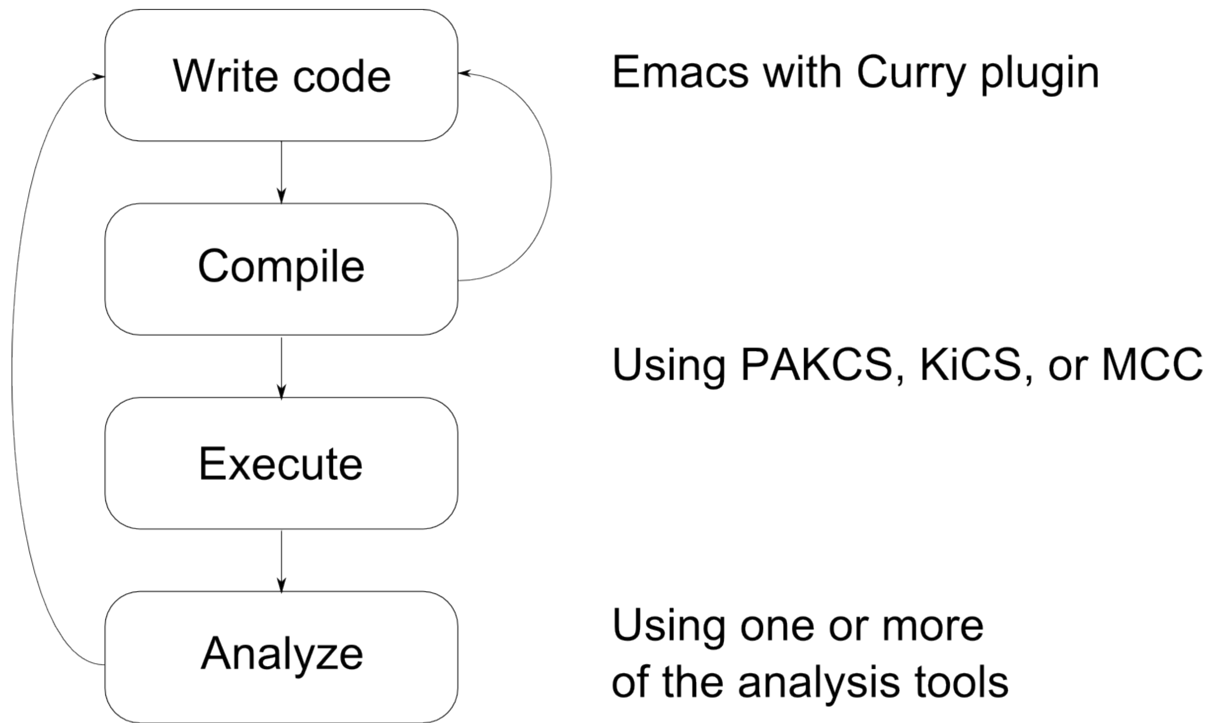


Figure 3.1: The current development process for Curry programs

as expected, which is a common case, the program has to be debugged or analyzed using any of the analysis tools described above. Some of these tools are implemented as console applications, which generally does not produce good user experience. The other analysis tools providing a GUI do not integrate in Emacs nor provide homogeneous user experience. Approaches like CurryBrowser deal with this problem and offer one GUI for various analyses and code navigation, however, it is not meant to modify code.

Besides this non-uniform landscape of tools, Emacs with Curry plug-in (or any text editor) does not offer the functionality of modern IDEs, like Eclipse Java IDE.

Summarized, the development of Curry programs usually comprises various tools with different usability and varying UIs. Although there are many tools, some features of modern IDEs are not available for Curry. As a result the user experience of Curry developers is poor compared to developers using other general purpose languages, like Java, C, C#, etc. The objective of this thesis is to provide a modern IDE for Curry unifying most of the functionality which is split among different tools so far. Curry developers should be able to accomplish all tasks of the development process using one tool without having to leave the Curry IDE. The result will be an improved user experience for Curry developers.

3.2 The Curry IDE

This section introduces the informal specification of the Curry IDE. At first, general functionality of a *modern* GUI is described in 3.2.1. The language-specific features of modern IDEs are listed in Section 3.2.2. The integration of existing Curry analyses is described in Section 3.3.

3.2.1 General Requirements

The basis for a modern IDE is a modern graphical user interface. But what turns a GUI into a modern GUI? This question cannot be answered in general, because the definition of a modern design for instance might be very subjective. However, we focus on the functionality, which is more objective. Hence, we define a feature set we expect from a modern application. As the Curry IDE has to contain an editor, the general requirements for a text editor are also discussed.

The following feature set defines the general, i.e., language independent, requirements for a modern GUI:

- *Layout*: The layout should be structured and flexible to support different window sizes.

- *File support*: The IDE should be able to open, save, rename, move and print files. Additionally, common auxiliary functions like opening recent files and displaying file properties should be provided.

- *Text editor*: The editor should provide actions like select, copy, paste, and cut text, search within text, undo, and redo last modifications.

- *Customizable appearance*: The user should be able to freely change the look of the application. That is, for instance, change the size, color, and type of font. Moreover, the Curry IDE might be able to display a lot of information to the user, it is desirable that the user can choose which information to show. Such modifications should be easy and intuitive.

3.2.2 Curry-Specific Requirements

This section specifies the main features that turn the application into a powerful IDE. Therefore, it describes the desirable functionality and discusses the extent to which it is provided by existing tools.

Project Explorer

The Curry IDE should provide a custom project explorer that is oriented towards the Java project explorer. It should support language specific properties of Curry projects including the definitions of libraries and external paths. To do so, a virtual layer should be added to every Curry project to display these properties. Additionally, this layer should contain an element for the source files, i.e., the Curry modules that are part of the project.

Code Presentation

In general, the developer will spend most of the time writing or reading code. The main part of an IDE is basically a standard text editor as described above. However, there are several language-specific features that make the work of developers simpler and more efficient. To improve the readability, the code can be colored in a special way, this is called *syntax highlighting*. Therefore, different parts of the code, for instance keywords, comments, variables, and strings, have different colors. Syntax highlighting is already available for Curry using the Emacs Curry plug-in. Another handy feature is *code folding*, it enables the developer to hide or show code blocks which might be temporarily (ir)relevant. This features is not provided by the Emacs Curry plug-in nor any other existing tool.

Error Visualization

One of the most important features of the Curry IDE is the visualization of errors and warnings. Modern IDEs display errors and warnings directly in the editor by marking the corresponding code. Additionally, the IDE shows the error and warning message, respectively. Up to now, there is no tool available that gives the developer immediate feedback when writing code. Currently, the runtime system outputs errors and corresponding parts, which have to be found manually by the developer. This feature will make a huge difference and accelerate the development process a lot.

Additional Functionality

Modern IDEs offer matured language support including *code completion* and so called *quick fixes*. The code completion is often very intelligent, since it is not only a textual completion of words which already have been written in the current document, but suggestions for completions that make sense in the current context. Such suggestions usually result in valid code and might contain language elements from other files or imported libraries. The quality of this feature is hard to define, however, the Curry IDE should provide code completion that suggests:

- Data types in signatures
- Valid expressions in right hand side of functions, i.e., accessible variables, functions, or constructors
- Existing functions or data types in export declarations.
- Existing functions or data types in import declarations.

Quick fixes are proposals by the IDE to fix existing errors. Again, this feature is hard to specify, but typically correct suggestions for typing errors should be made. The Curry IDE should also be able to detect and correct errors such as module names that do not correspond to file names. None of these features are provided by any existing tools.

Additional Information

The IDE should neatly provide additional information to the developer. For programs that consist of more than one file, it is important to have an overview of the project's file structure. This is provided by the *project explorer*. Aside from that, the IDE should offer an overview of the current module. It should contain all defined data types, signatures, functions, imports, and exports. This IDE part is called *outline*. This kind of information is provided by the CurryBrowser, which does not allow modifications of the code. At the same time, Emacs has a project explorer, but does not support a module outline. Modern IDEs also provide additional language-specific information to the developer. This might be the type of a variable or the needed parameters for a function. Moreover, the Eclipse Java IDE has *javadoc*[10] integration. Javadoc is a tool for generating API[11] documentation in HTML format from doc comments in source code. For every element that has a javadoc annotation, Eclipse shows the javadoc wherever this element is used. The corresponding tool for Curry is called CurryDoc. The Curry IDE should provide additional information to the developer including the signature of functions and CurryDoc annotations. This feature becomes very important for huge projects where developers constantly use code written by other developers. It displays the needed information to the developer so that in most cases it is not necessary to browse to the corresponding definition. This feature is not provided up to now and can be very time-saving.

[10] `http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html`, last visited December 3, 2012

[11] application programming interface

Execution

The code should be executable without leaving the IDE. However, the Curry IDE is not meant to implement another compiler or interpreter for Curry. The existing runtime systems should be integrated instead. Therefore, an interactive console for any of the existing runtime systems should be available within the IDE. It should be possible to directly start a runtime system and load the active module without manual interaction with the runtime system.

Code Navigation

It is really uncommon to read code sequentially, instead, it is usual to navigate from one definition to another which might be in another module or even project. That is the reason why reading code using an standard text editor (or even Emacs with the Curry plug-in) can be time-consuming right up to really frustrating. A solution for this lack of usability, is the integration of hyperlinks within the editor that reference the definition of any element. The Curry IDE should provide this feature, since it is not provided up to now. CurryBrowser supports the navigation using the module outline: by selecting an element in the outline the corresponding code is displayed and marked. This feature should be also included in the Curry IDE. Another handy type of code navigation is the advanced search including finding references to a selected element or search for elements of a specific type. The advanced search is desirable to be available in the Curry IDE.

By implementing all these features, the resulting IDE will provide the main features of modern IDEs like Eclipse, Netbeans, or VisualStudio. The consultation with experts certifies that such an IDE will enhance their workflow and simplifies the development process with Curry.

3.3 Integration of Curry Analysis Tools

To accomplish the goal of supporting the whole development process of Curry programs, the analysis of the code has to be considered as well. There are a lot of existing analyses for Curry that should be integrated into the Curry IDE instead of implementing them from scratch again. Attempts are being made to develop the *Curry Analysis Tool* which unifies Curry analyses and makes them accessible via socket communication. At the time of writing, this tool does not exist yet. However, a protocol has already been specified that will be supported by the *Curry Analysis Tool*. Basically, the tool is a server that allows socket connections on a specific port. After the connection has been established, text based commands can be sent to the tool which are terminated by a *linebreak*. The tool processes the request and sends an appropriate response to the client.

3.3.1 Curry Analyses

To delegate the execution of analyses to the *Curry Analysis Tool*, basic configuration is necessary. This includes the path of the module to analyze as well as all paths containing the Curry modules it refers to directly or indirectly. The Curry analyses are identified by unique names and can be applied to modules, functions, data types, and data constructors. Moreover, they might support different output types which specify how the result text should be interpreted.

3.3.2 Protocol

The protocol specifies valid requests as well as the format of corresponding responses. The tool sends two different types of responses according to whether the request is valid or not. If an error occurred, the server sends "error <msg>" where *<msg>* will be the error message that is terminated by a *linebreak*. In case of success, the response starts with the line "ok <n>" where *<n>* is the number of lines that the subsequent result text will consist of. Afterwards, the result text is sent line by line.

One key benefit of the *Curry Analysis Tool* is the generic approach which also implies that the provided analyses are not known until runtime. Hence, clients have to fetch the list of available analyses first, to make valid subsequent request. The corresponding command is "GetAnalysis", the server replies to this request with the list of analyses in the format "<analysis> <output>". The complete response to the "GetAnalysis" request could look as follows:

```
1 ok 5
2 Deterministic CurryTerm
3 Deterministic Text
4 Deterministic XML
5 HigherOrder   CurryTerm
6 DependsOn     CurryTerm
```

Before starting an analysis, the tool has to be configured by setting the necessary paths for the analysis. This is supported by the command "SetCurryPath <dir1>:<dir2>:..." where *<dir1>*, *<dir2>*... are the single paths separated by a colon.

There are four commands to start an analysis, they differ by the type of the target element:

```
1 AnalyzeModule            <analysis> <output> <module>
2 AnalyzeFunction          <analysis> <output> <module> <function>
3 AnalyzeDataConstructor   <analysis> <output> <module> <constr>
4 AnalyzeTypeConstructor   <analysis> <output> <module> <type>
```

where <*analysis*> is the kind of analysis and <*output*> is the output type, the values should correspond to an available analysis that is part of the response of the *GetAnalysis*-command. The remaining arguments are defined as follows: *module* specifies the module name, *function* specifies the function name, *constr* specifies the constructor name, and *type* specifies the data type name.

There is also a command to stop the server: "StopSever".

3.3.3 Integration

The Curry IDE should provide an easy way to use any of the supported analyses and present the result in an appropriate way. The protocol implies that the Curry IDE does not know the particular analyses available at runtime. Therefore, it has to integrate and call them generically. It is desirable that the available analyses can be started directly from the context menus of either the editor or the module outline. Moreover, the Curry IDE has to offer a flexible mechanism to visualize the results. It should be possible to dynamically load visualization for specific output types into the Curry IDE. This implies that multiple visualizations for one output type might exist, hence, the user has to be able to choose the desired visualization, if possible.

4 Foundations

This section introduces the basic knowledge for this thesis. At first, the theoretical foundations are summarized in Section 4.1, various following sections depend on this knowledge and reference relevant parts when necessary.

4.1 Theoretical Foundations

In this section the theoretical foundations for this thesis are introduced.

4.1.1 Language Recognition Workflow

The ability to "recognize" a language is the basis for most features of any IDE. So, what does "language recognition" mean? At first the IDE has to be able to decide whether an input source code is part of a particular language or not. Figure 4.1 depicts the essential steps of this process. Furthermore, the resulting data structure, the annotated abstract syntax tree (AST), serves as a basis for more complex analyses of the input program.

4.1.2 Lexical Analysis

The following definitions are taken from [ASU86]. The lexical analysis aims to divide a character string (the source code) into a sequence of *tokens*, which constitute atomic lexical entities. Tokens which are syntactically of the same kind are grouped into *symbol classes*. A *scanner* is a program for the lexical analysis and is formally defined as follows:

scanner: $\Sigma^* \rightarrow T^*$ where Σ is the alphabet and T is the set of symbol classes.

The scanner is able to ignore tokens of particular symbolic classes, so that they are not part of the output sequence of tokens. For instance, this is the case for comments and blanks in many programming languages. There is another special group of tokens, the *keywords*. A keyword is an unnamed token, i.e., it does not belong to a specific symbol class. It is possible that a particular character string can be a keyword and a member of one or more symbol classes, however, every token is either the member of exactly one symbol class or it is a keyword. There are widely used principles to resolve such issues. Often, keywords have higher priority than symbol classes. For instance, the token "class" is a keyword in Java, consequently, it is not possible to use this token for the name of a variable, function, or class.

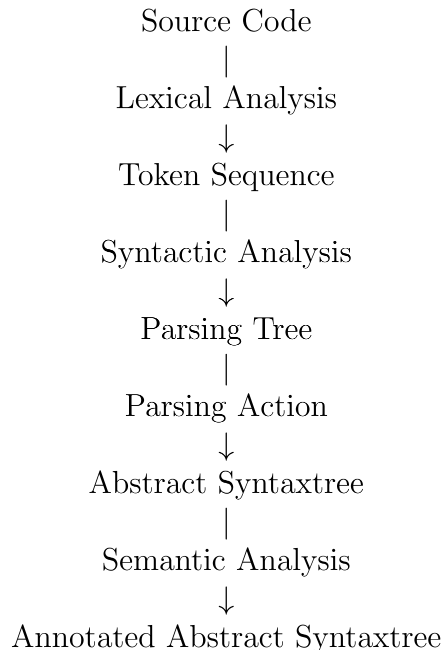


Figure 4.1: Workflow of the language recognition process

The *principle of longest match* is also widely used. It implies that the next token of a character string is the longest prefix belonging to any symbolic class. Hence, the string "className" is one token in Java (the identifier "className") instead of the keyword "class" and the token "Name".

If a character string could be a member of multiple symbol classes, often, the first appropriate symbol class is selected. Hence, the order of the declarations of symbol classes does matter.

The structure of symbol classes is regular. This makes it possible to describe scanners by finite automata. Consequently, the implementation of scanners can be automated completely. There are scanner generators, which compile regular expressions and corresponding symbol classes to a scanner program.

4.1.3 Syntactic Analysis

We now focus on the syntactic analysis, the aim of this phase is to identify syntactic entities of the target language from a token sequence. Generally, a scanner is not sufficient for this kind of task, since many constructs of programming languages are not regular. A program which does the syntactic analysis is called *parser*. A parser processes token sequences (which are the output of scanners) and creates an appropriate model repre-

senting their grammatical structure. Typically, this model is called *syntax tree* because of its tree-like structure [ASU86].

The syntax of most programming languages can be described by *context free grammars* and implemented by *stack machines*. Formally, a context free grammar defined as follows:

Definition 4.1.1. *A* context free grammar $G = (N, T, P, S)$ *consists of:*

- *A set N of non-terminal symbols*
- *A set T of terminal symbols where $N \cap T = \emptyset$ (which are basically the symbol classes of tokens)*
- *A start symbol $S \in N$*
- *A set P containing productions (or rules) of the following form:*
  $A \rightarrow \alpha_1\alpha_2 \dots \alpha_n$
  where $n \geq 0, A \in N, \alpha_i \in N \cup T$. We call $\alpha_1\alpha_2 \dots \alpha_n$ the body *of the production.*

To create a syntax tree for a specific token sequence, the parser tries to derive it from a grammar. Starting from the start symbol, the parser subsequently replaces a non-terminal with the body of one of its productions. Depending on the strategy which non-terminal to replace first, we talk about *right-most* and *left-most* derivation. Left-most derivation builds a syntax tree from the root to the leaves, hence it is called *top-down parsing*. Whereas *right-most* derivation builds a syntax tree in the opposite direction, from the leaves to the root, and is called *bottom-up parsing*.

In both cases, the root of the syntax tree consists of the start symbol, the leaves consist of terminal symbols only, inner nodes, and their children correspond to the production which has been applied. It is possible, that there are more than one syntax tree for one and the same input. A grammar that allows to build multiple syntax trees for one input is called *ambiguous*. Note that this characteristic is generally undesired for programming languages.

4.1.4 EBNF

The *Extended Backus-Naur Form (EBNF)* is a metasyntax notation that can be used to express context free grammar. It consists of terminal symbol and non-terminal production rules. The following symbols are:

Symbol	Meaning		Symbol	Meaning
=	Definition		[...]	Optional
,	Concatenation		(...)	Grouping
;	Termination		{...}	Repetition
|	Alternative		".." and '...'	Terminal string

4.1.5 Recursive-Descent Parsing

Any context free grammar can be analysed using the popular CYK-algorithm (Cocke, Younger, Kasami), however, its worst case running time is $O(n^3)$ [You67]. In general, this is not sufficient for use in field, where linear methods are wanted. For the sake of performance, specialized methods and corresponding context free grammars are considered.

In the following, we introduce the concept of *recursive-descent parsing*, a top-down parsing strategy. A recursive-descent parser reads a sequence of tokens from left to right and tries to consume it token by token. It consists of a set of functions where each function usually implements one of the production rules of the grammar. Functions related to terminal symbols consume the corresponding token from the input sequence. A recursive-descent parser accepts a token sequence of an input program, if the token sequence is completely consumed. Otherwise, the input program is not correct in sense of syntax defined by the grammar.

Many grammars contain multiple productions which have the same non-terminal on the left-hand side. In this case the function representing such a non-terminal has to be able to decide which function to apply. It is desirable that this decision can be made looking at the next token of the input sequence. This is not always possible, hence, there are more powerful parsing techniques, like *LR-parsing* which will not be discussed in this thesis.

There is a special class of grammars that can be parsed using the recursive-descent method. Grammars belonging to this class are called *LL(k) grammars*.

LL(k) Grammar

An LL(k) grammar is a context free grammar having additional characteristics. LL(k) stands for *left*-to-right parsing with *left-most* derivation and k-symbol-lookahead.

During the parsing process, there are often situations in which multiple production rules can be applied. This is due to the fact, that an arbitrary number of production rules can have it on the left-hand side. For LL(k) grammars it is possible to decide which production to apply by looking at the subsequent k token of the input sequence.

Consequently, ambiguous grammars and all grammars that contain *left recursion* (which is described in the following section) are no LL(k) grammars. However, most structures of programming languages can be expressed by an LL(1) grammar [ASU86].

The good thing is that recursive-descent parsers run in linear time [ASU86].
Formally, LL(k) grammars are defined as follows:

Definition 4.1.2. *Let $w \in T^*$ be a word. Then $start_k$ is defined as follows:*
$start_k(w) = \begin{cases} w & \text{if } |w| < k \\ u & \text{if } w = uv \text{ where } |u| = k \end{cases}$
Where $|w|$ is the length of w.

Definition 4.1.3. *A context free grammar $G = (N, T, P, S)$ is a $LL(k)$grammar, if the following is true:*
If
$S \overset{L}{\rightarrow}^* uA\alpha \overset{L}{\rightarrow}^* u\beta_1\alpha \rightarrow^* uv$
$S \overset{L}{\rightarrow}^* uA\alpha \overset{L}{\rightarrow}^* u\beta_2\alpha \rightarrow^* uw$
are two left derivations where $start_k(v) = start_k(w)$, then $\beta_1 = \beta_2$.

Further, we restrict the class of LL(k) grammars for the ability to make the decision independent of the context and, finally, achieve a constructional approach for decision-making.

Definition 4.1.4. *A context free grammar $G = (N, T, P, S)$ is called* strong LL(k) grammar*, if the following is true:*
If
$S \overset{L}{\rightarrow}^* u_1A\alpha_1 \overset{L}{\rightarrow}^* u_1\beta_1\alpha_1 \rightarrow^* uv$
$S \overset{L}{\rightarrow}^* u_2A\alpha_2 \overset{L}{\rightarrow}^* u_2\beta_2\alpha_2 \rightarrow^* uw$
are two left derivations where $start_k(v) = start_k(w)$, then $\beta_1 = \beta_2$.

Parsing Process

In the following, we describe the parsing process formally to get an impression how parser make their decisions. To do so, we assume to have a LL(k) grammar G and an input sequence of tokens, which should be parsed. As explained above, the parser consumes this sequence applying grammar rules and may need to make decisions regarding the selection of the right production to apply. To have a basis of decision-making, the set of all token sequences (of length k) is derived for every right-hand side of a production. By matching the subsequent k tokens of the input sequence with these sets, the right production can be chosen.

Formally, this set of tokens is defined as follows:

Definition 4.1.5. *Let $G = (N, T, P, S)$ be a context-free grammar, $\alpha \in (N \cup T)*$ a right-hand side of a production and $k > 0$. Then the set of all start token sequences which can be derived from α is defined as follows:*
$FIRST_k(\alpha) = \{start_k(\omega) | \alpha \rightarrow^* \omega \in T^*\}$

The $FIRST_k$-sets of the right-hand sides of non-terminal productions serve as a basis for making LL(k)-decisions. The special case where the right-hand side of a non-terminal production has the length $< k$ can be solved by looking at the following context of the non-terminal. Therefore, we define the "following context" as follows:

Definition 4.1.6. *Let $G = (N, T, P, S)$ as context-free grammar, $A \in N$ and $k > 0$. Define the following set:*
$FOLLOW_k(A) = \{\omega \in T^* | S \to^* uAv \text{ and } \omega \in FIRST_k(v)\}$

The set $FOLLOW_k(A)$ consists of all tokens which can follow A throughout derivations. The concatenation of $FIRST_k$ of a right-hand side of a production and $FOLLOW_k$ of the corresponding non-terminal symbol can be used to make necessary decision. We express this by defining an appropriate *director set*:

Definition 4.1.7. *Let $G = (N, T, P, S)$ be a context-free grammar and $(A \to \alpha) \in P$. Then*
$D_k(A \to \alpha) = start_k(FIRST_k(\alpha) \cdot FOLLOW_k(A))$
is the director set for this production.

The director sets of all productions are pairwise disjoint for any strong LL(k) grammar and can be used as a basis of decision-making for recursive-descent parsers.

Left-recursion

As mentioned above, left-recursive grammars are not contained in the set of LL(k) grammars. As an example, the following grammar is left-recursive:

```
1  Expr  →  Expr '+' Term
2  Expr  →  Term
```

The rule *Expr* contains a left-recursion. The problem for a recursive-descent parser is that it is not possible to determine how often the rule *Expr* has to be applied. Technically, a recursive-descent parser for a grammar containing the rule *Expr* would fall into infinite recursion when trying to parse any input sequence. However, any context free grammar can be transformed to an equivalent grammar that has no left-recursion by eliminating the left-recursion [ASU86]. Note that the transformed grammar may produce a different parse tree.

Generally, left-recursion can be eliminated as follows: Consider a grammar containing the production $X \to \alpha$ and the left-recursive production $X \to X\beta$ (where $\alpha \neq X$). Consequently, X can be derived to the following token sequences: $X \to {}^*\alpha\beta^*$

We can rewrite the rules as follows to make them right-recursive:

1 $X \to \alpha X'$
2 $X' \to \beta X'$
3 $X' \to$

Left-factoring

A common problem for LL-parsing is that productions have same prefixes. If the length of a common prefix of two productions is greater or equal to the lookahead constant k, the parser is not able to make a decision which of those productions to take (in corresponding situations). Usually, it is possible to rewrite the production rules to postpone the decision. This transformation is called *left-factoring*.

In general left-factoring can be applied as follows: Let $A \rightarrow \alpha\beta_1|\alpha\beta_2$ be two productions. We rewrite them as follows:

```
1    A → αA′
2    A′ → β1|β2
```

This transformation allows the parser to make the decision when necessary information is available.

Backtracking

Backtracking is an approach to extend the decision-making ability of top-down parsers. The idea of this approach is to test possible alternatives subsequently for their applicability. If an alternative has been tested successfully, the parser make the appropriate decision. The problem with backtracking is that, in the worst case, it may take time that is exponential in the input length.

There is also a method to realize backtracking that guarantees linear parsing time, in return, it has very huge memory consumption [FK02].

LL(*) Parsing

LL(*) is a top-down parsing strategy which focuses on expressiveness rather than on pure performance. [PF11] introduces the concept as follows:
LL(*) stands for LL(k) parsing with dynamic lookahead $k \geq 1$. The key idea is to use regular-expressions rather than fixed constant or backtracking with a full parser to do lookahead. This can be done by trying to construct a deterministic finite automaton (DFA) for each non-terminal in the grammar to distinguish between between alternative productions. If no suitable DFA for a non-terminal can be found, a backtracking strategy is used. As a consequence, LL(*) parsers can use arbitrary lookahead and, finally, fail over to backtracking depending on the complexity of the parsing decision. Although, the parsers decide on a strategy dynamically according to the input sequence. This means that just because a decision *might* have to scan arbitrarily far ahead or backtrack does not mean that it *will* at parse-time. In practice, LL(*) parsers only look ahead one or two tokens ahead on average despite needing to backtrack occasionally.

4.1.6 Parser Generators

The construction of parsing tables is very easy - but quite time-consuming. Hence, it is a good idea to automate this process. *Parser generators* use a grammar as input and yield a parser program as output. Some parser generators allow grammars which are enriched with *syntactic* and *semantic* predicates and embedded actions. At the beginning the parser generator supported LL(1) or LR(1)-languages only due to efficiency. Further research and more efficient algorithms enabled support for LL(k)- and even LL(*)-languages ($k \geq 1$) in the last few years.

4.1.7 Syntactic and Semantic Predicates

It is possible to improve the recognition strength of an LL-parser using *syntactic* and *semantic predicates* [PQ94]. Basically, predicates are actions that are embedded in the grammar. Syntactic predicates can be used help the parser to make a particular decision between productions. They tell the parser to try to parse the predicated production first, i.e., syntactic predicates introduce local backtracking procedures.

Semantic predicates are more expressive, they allow to execute arbitrary actions to help the parser to make decisions. They can be used to resolve finite lookahead conflicts and syntactic ambiguities with semantic information.

4.2 ANTLR - A Parser Generator

ANTLR, *Another Tool For Language Recognition*, is a parser generator, which can be used to construct parsers, interpreters and compilers. The underlying parsing strategy of ANTLR is LL(*) (cf. Section 4.1). The input to ANTLR is a context-free grammar augmented with *syntactic* and *semantic* predicates. Further, ANTLR accepts all but left-recursive context-free grammars, because it is based on LL(*) parsing [PF11]. ANTLR is widely used which is underpinned by the analysis of download statistics presented in [PF11], further, it is mentioned that ANTLR is used by a large number of projects including Google App Engine, IBM Tivoli Identity Manager, Yahoo! Query Language, Apple Keynote, Sun/Oracle JavaFX language, and NetBeans IDE.

4.2.1 ANTLR-Works

ANTLR-Works is a graphical development environment for ANTLR grammars. It contains an editor to create and modify ANTLR grammars. As well, it provides an interpreter and a language-diagnostic debugger. Moreover, it detects and visualizes errors in the grammar, like ambiguities and left-recursive rules.

4.3 Eclipse Rich Client Platform

The majority of modern software products that are meant to be used by some human being are developed with the focus on *user experience*. That is, the end-user shall be able to do his/her work quickly. Often, the *user interface* (UI) is *enriched* with various features, like drag and drop, system clipboard, navigation, and customization. That is why we talk about *rich user interfaces*. Additionally, such applications have to be very fast and, therefore, directly run on the client's machine. As many features are part of various rich client applications, it is a good idea to implement them once and just customize them where needed. This way, a lot of development time can be saved. *Rich client platforms* (RCP) provide such building blocks to easily create user interfaces.

4.3.1 Eclipse

Eclipse is an open-source based community that builds Java-based tools. The most popular output of this community is the Eclipse Java Integrated Development Environment. The Eclipse IDE is built on top of a rich client platform, called Eclipse RCP.

4.3.2 Eclipse RCP

[MLA10] gives a good overview of the main characteristics of Eclipse RCP:

Components - Eclipse applications are built by combining individual software components. There is a specification called *OSGi* which describes a modular approach for Java applications. It enables a development model where applications are dynamically composed of many different reusable components. The OSGi specification enables components to hide their implementation from other components while communicating through *services*, which are objects that are shared between components. The Eclipse platform contains *Equinox* which is one implementation of the OSGi specification and loads, integrates, and executes Eclipse components, which are called *plug-ins*.

Native user experience - A native user experience includes a responsive UI that integrates into the desktop. The Eclipse platform supports different user interface toolkits, such as:

- The Eclipse Standard Widget Toolkit (SWT) provides a graphical user interface toolkit for java that allows efficient and portable access to the native UI facilities of the operating system it is implemented on. This technology makes it possible to create Java-based applications that are indistinguishable from platform's native applications.

- *JavaFX* is designed to provide a lightweight, hardware-accelerated Java UI platform for enterprise business applications. With JavaFX, developers can access native system capabilities.

Portability - The Eclipse platform can be executed on various desktop operating systems, such as Windows, Linux, and Max OSX. Eclipse applications can even run on tablets as well as mobile and embedded devices.

Intelligent install and update mechanism - Eclipse plug-ins are versioned, this allows running multiple versions of the same plug-in side-by-side. Applications can be configured to run with the exact version they need. Moreover, the Eclipse platform enables plug-ins to be deployed and updated using various mechanisms: HTTP, Java Web Start, Update sites, simple file copying or sophisticated enterprise management systems.

Tool support - The Eclipse IDE serves as a first-class Java IDE with integrated tooling for developing, testing, and packaging rich client applications.

Component libraries - The Eclipse community has produced plug-ins for building pluggable UIs, managing help content, install and update support, text editing, consoles, product introductions, graphical editing frameworks, modeling frameworks, reporting, data manipulation, and much more.

This paper will refer to Eclipse 4.2, which as of this writing is the latest release. Note that the simultaneous release of Eclipse named *Juno* (in June 2012) also was based on Eclipse 4.2.

4.4 Xtext

The Eclipse project *Xtext* is a framework for developing programming languages or domain-specific languages (DSLs). Xtext provides a set of domain-specific languages and APIs to describe the different aspects of a programming language. That information is used to generate an Eclipse-based development environment providing features known from the Eclipse Java IDE. Therefore, Xtext generates a parser, a language model used for abstract syntax trees, a serializer, a code formatter, and a code generator or an interpreter. These runtime components are combined into a feature-rich Eclipse-based development environment for the specified language. Depending on the complexity of the target language, the following default functionality is provided among others:

- Syntax highlighting: the editor supports syntax coloring based on the lexical structure.
- Content Assist: The editor proposes valid code completions at any place in the document, helping developers with the syntactical details of the language.

- Validation and Quick Fixes: Xtext has support for static analysis and validation of language models. Errors and warnings are displayed in the editor which can be corrected by custom quick fixes.

For languages where the default functionality is not sufficient, Xtext provides DSLs and APIs that allow to configure or change the most common things very easily.

As mentioned above, languages can be described by grammars. That is why a grammar is the main input for Xtext. It can be written in an EBNF like notation, which is enriched with additional information.

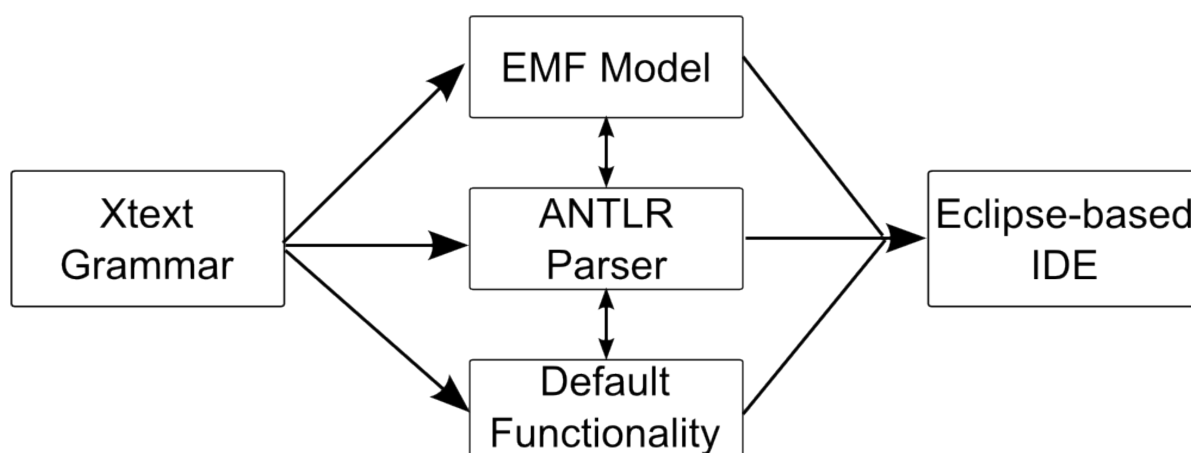


Figure 4.2: Xtext generation process

Figure 4.2 shows how Xtext generates an Eclipse-based IDE from an input grammar. Xtext creates a custom EMF (*Eclipse Modeling Framework*) model for the language. The Xtext-grammar is translated into an ANTLR-grammar, which is augmented by semantic actions to build an abstract syntax tree using the generated EMF model. The language specific default functionality for the resulting IDE is constructed to work with the EMF model as well. Finally, an Eclipse RCP application is created containing an editor, which is connected to the generated parser, and integrating all components to provide the default functionality. This application is (in most cases) a full-functional IDE for the defined language. As mentioned above, this works for “simple” DSLs, whereas more complex languages require additional configuration.

It is worth mentioning that Xtext uses ANTLR, but does not support semantic predicates nor allow including code blocks in the grammar. This might be due to the fact that Xtext makes heavy use of code blocks in the ANTLR grammar to make the parser build an abstract syntax tree that corresponds to the EMF model.

4.4.1 Generation Workflow

Now we take a look at the generation process of Xtext. The generator uses a special DSL called MWE2 (the modeling workflow engine[1] to configure the generation process. Basically, the generator executes a set of fragments that are implementations of the interface *IGeneratorFragment*[2]. Among other methods, the interface specifies the methods *generate* and *getGuiceBindings*, which can be used to generate a component based on the grammar and integrate it into the resulting IDE. The integration based on bindings and dependency-injection is described in the following paragraph.

Every Xtext project contains a MWE2 file with a default configuration of the generator. This configuration can be modified by removing or adding fragments. By default various fragments are integrated into the generation process including the following:

- EcoreGenerator: Generates the EMF model and the corresponding Java API
- XtextAntlrGenerator: Generates the ANTLR parser
- JavaValidator: Creates an empty implementation of a validator and integrates it into the resulting IDE.

We do not introduce MWE2 in this thesis, but it is worth mentioning that it comes with an engine to execute workflows. For the workflow of Xtext-projects, this means running the generation process of the IDE for the specified grammar.

The *XtextAntlrGenerator* is obviously a very important fragment. It generates the ANTLR grammar and runs ANTLR to generate the lexer and parser. The errors produced by ANTLR during the generation will be directly outputted on the console. The reasons for such errors are usually not immediately obvious, especially within complex grammars. As mentioned in Section 4.2.1, ANTLR Works simplifies the debugging of grammars. Xtext provides a special fragment that generates a debug version of the ANTLR grammar. This fragment can simply be added to the generation workflow. The resulting debug ANTLR grammar does not contain any code blocks, is more readable than the normal grammar, and, hence, good for debugging purposes.

4.4.2 Customization

One reason for Xtext's flexibility is that it uses dependency-injection for all components that are integrated into the resulting IDE. Dependency-injection is a software design pattern that allows a choice of component to be made at run-time rather than compile time. Xtext employs the lightweight dependency-injection framework *Guice*[3]

[1] `http://www.eclipse.org/Xtext/documentation.html#MWE2`, last visited December 3, 2012
[2] package: org.eclipse.xtext.generator
[3] `http://code.google.com/p/google-guice/`, last visited December 3, 2012

(pronounced 'juice') that is developed by Google. Guice provides the interface *Module* that contributes configuration information, typically interface bindings, which will be used to create an injector. An injector builds the graphs of objects that make up an application. It tracks the dependencies for each type and uses bindings to inject them.

Every Xtext project contains an implementation of Guice's *Module* interface with the default configurations. It enables users to inject custom implementations for every feature of the resulting IDE.

4.4.3 The Grammar Language

To get a better understanding how Xtext works, we consider a simple language that has a Curry-like syntax, named *SimpleCurry*. It allows the definition of a module header and its body containing any number of data declarations and function signatures. A valid input program could look as follows:

```
1 module MyModule where {
2   data MyDataType = MyDataConstructor;
3   myFunction :: MyDataType -> AnotherDataType;
4 }
```

Listing 4.1: Sample SimpleCurry-program.

The first step is the definition of a grammar for *SimpleCurry*. Being the heart of any Xtext-language specification, the grammar definition is the starting point and the corner stone of every Xtext project. It is defined using a special grammar language, which itself is a DSL designed for the description of textual languages. It is based on EBNF augmented with some extensions to direct the parser and to influence how the AST is constructed. The basic Xtext-grammar definition for *SimpleCurry* looks as follows:

```
1  Module hidden(ANY_OTHER):
2                    'module' ModuleID 'where' Body;
3  Body:             '{'
4                    (BlockDeclaration ';')*
5                    '}';
6  BlockDeclaration: DataDeclaration
7                    | Signature;
8  DataDeclaration:  'data' TypeConstrID '=' ConstrDecl;
9  ConstrDecl:       DataConstrID;
10 Signature:        FunctionID '::' TypeExpr;
11 TypeExpr:         TypeConstrID ('->' TypeExpr)?;
12 // Identifiers consist of any sequence of letters.
13 ModuleID:         ID;
```

```
14 TypeConstrID:     ID;
15 DataConstrID:     ID;
16 FunctionID:       ID;
17 terminal ID:            ('a'..'z' | 'A'..'Z')+;
18 terminal ANY_OTHER:     .;
```

In large parts, the grammar definition is compliant to the EBNF notation. It only differs by the keywords 'hidden' and 'terminal'. The 'terminal' keyword is used to define token classes for the lexer. In line 18 of Listing 4.1 the class of 'identifiers' is defined as all words consisting of one or more letters. In line 19 the '.' instructs the lexer to recognize everything else (non-*ID*s) as a token of the class named *ANY_OTHER*, so that, for instance, whitespaces and newlines are members of this token class. Tokens of the class *ANY_OTHER* are only used by the lexer to separate the identifiers, the parser is not interested in them. To handle this common case, Xtext provides the concept of hidden tokens. The 'hidden' keyword in line 1 of Listing 4.1 defines that all tokens of the class *ANY_OTHER* are hidden from the parser rules. Consequently, the parser works effectively on a token sequence from which all hidden tokens have been removed.

4.4.4 Data Model

In this section, we look at the data model that is created for input programs. Xtext provides a simple way to define and build the data model for custom abstract syntax trees. All necessary information is embedded in the grammar definition. The following expressions are supported: *assignments*, *actions*, and *cross-references*.

Assignments

Assignments are used to assign the consumed information to a property of the currently produced object. The type of the current object is specified by the return type of the parser rule. If it is not explicitly stated, it is implied that the type's name equals the rule's name. The type of the assigned property is inferred from the right hand side of the assignment:

```
1 Module:        'module' name=ModuleID 'where' Body;
```

The syntactic declaration of a module header starts with the keyword 'module' followed by an assignment:

```
1 name=ModuleID
```

The left-hand side refers to the property 'name' of the current object (which is a *Module*). The right-hand side can be a rule call, a keyword, a cross-reference, or an alternative comprised by the former. The type of the property needs to be compatible with the type

of the expression on the right. As *ModuleID* is a sequence of letters (i.e., a *String*), the property 'name' needs to be of type *String* as well. Besides the =-assignment operator, there are the *+=*-sign and the *?=*-sign that can be used as assignment operators. The *+=*-sign indicates that the property on the left is a list and the value on the right will be added to that list. The *?=*-sign expects a property of type boolean and sets it to true if the right-hand side was consumed independently from the concrete value of the right hand side.

Actions

The type of the object to be returned by a parser rule is determined from the specified return type of the rule which may have been inferred from the rule's name if no explicit return type is specified. However, *Actions* allow explicit creation of the return object. Xtext supports two kinds of *actions*: *Simple actions* and *assigned actions*. A *simple action* can be used to explicitly instantiate a particular object. For instance, we could redefine the rule *BlockDeclaration* as follows:

```
1 BlockDeclaration returns Declaration:
2           {DataDeclaration} DataDeclaration
3           | {Signature} Signature;
```

This will instruct the parser to create a *DataDeclaration*-object or a *Signature*-object depending on the rule alternative that is taken. Both data types have a common super type called *Declarations*.

Cross-References

In most programming languages, identifiers are used to refer to particular defined entities. The same goes for Curry as well as SimpleCurry. Consider the code from Listing 4.1, for instance, the identifier 'MyDataType' in line 3 is a reference to the *DataType* 'MyDataType' defined in line 2. In traditional compiler construction such cross-links are not established during parsing but in a later linking phase. The same goes for Xtext, however, the specification of the cross-link information is embedded in the grammar. A cross-link is defined by square brackets embracing the type of the object to reference and optionally the class of terminals that is expected as the identifier. To insert the cross-link within *Signatures* to *DataTypes* in SimpleCurry the rule *TypeExpr* has to be modified:

```
1 TypeExpr:        type=[DataDeclaration|TypeConstrID] ..
```

Additionally, the rule *DataDeclaration* has to be updated so that the 'name'-property of *DataDeclarations* is set appropriately.

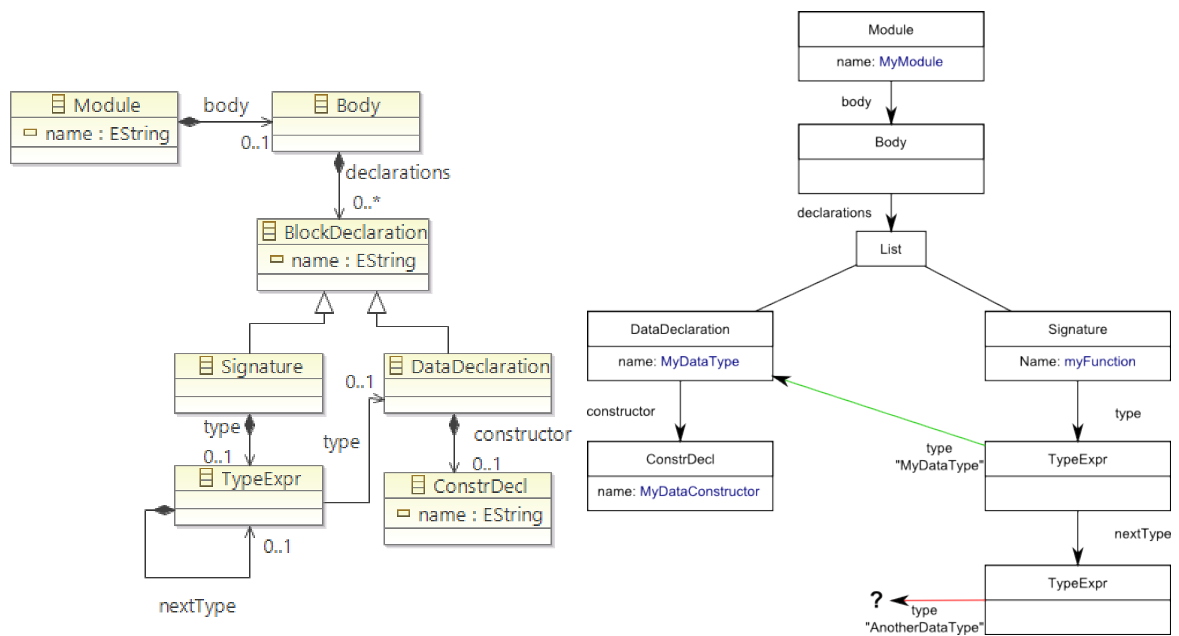
```
1  DataDeclaration:  'data' name=TypeConstrID '=' ConstrDecl;
```

Now, Xtext automatically establishes the cross-references and produces an error, if for a particular *TypeConstrID* within a *Signature* no target element can be found. For the sample program from Listing 4.1, this is the case for the identifier 'AnotherDataType'. So, the definition of cross-links implies semantic validation for references. The complete Xtext-grammar for SimpleCurry including data model information and the cross-link looks as follows:

```
1  Module hidden(ANY_OTHER):
2                     'module' name=ModuleID 'where' body=Body;
3  Body:              {Block} '{'
4                     (declarations+=BlockDeclaration ';')*
5                     '}';
6  BlockDeclaration: DataDeclaration
7                     | Signature;
8  DataDeclaration:  'data' name=TypeConstrID
9                         '=' constructor=ConstrDecl;
10 ConstrDecl:        name=DataConstrID;
11 Signature:         name=FunctionID '::' type=TypeExpr;
12 TypeExpr:          type=[DataDeclaration|TypeConstrID]
13                        ('->' nextType=TypeExpr)?;
14 // Identifiers consist of any sequence of letters.
15 ModuleID:          ID;
16 TypeConstrID:      ID;
17 DataConstrID:      ID;
18 FunctionID:        ID;
19 terminal ID:            ('a'..'z' | 'A'..'Z')+;
20 terminal ANY_OTHER:     .;
```

The generated data model for SimpleCurry is presented in Figure 4.3. Figure 4.3a shows the generated data types for SimpleCurry and their relations. As described, Xtext takes care of implicit inheritance, hence, *Signature* and *DataDeclaration* have the same super type *BlockDeclaration*.

Figure 4.3b depicts the AST that is constructed for the input program from Listing 4.1. The green line represents the cross-reference from the *TypeExpr* within the *Signature* of 'myFunction' to the *DataDeclaration* 'MyDataType'. On the contrary, the red line is meant to demonstrate the broken cross-reference for the *DataDeclaration* 'AnotherDataType' that does not exist. Indeed, this "AST" is no tree anymore, because the cross-references create cycles in the undirected graph of objects. However, in the following we will still refer to this kind of data representation as an AST to indicate that we



(a) The data model of SimpleCurry generated by Xtext.

(b) The “AST” for the sample program from Listing 4.1.

Figure 4.3: The data model for SimpleCurry.

mean the output of the parser.

4.4.5 The generated IDE

In this subsection, we give an overview of the parts Xtext generates for the IDE. Basically, an Xtext project consists of four separate projects. Assuming the project's name is "myIDE", the following individual projects are created:

- *myIDE*: An Eclipse plug-in that contains grammar definition and all runtime components (lexer, parser, linker, validation, etc.).
- *myIDE.sdk*: An Eclipse feature project, a *feature* is kind of a logical unit that contains a list of plug-ins. Features are used by the Eclipse update manager.
- *myIDE.test*: An individual project for Unit tests.
- *myIDE.ui*: Contains the editor and all the other workbench related functionality.

Besides the *src* folder, the projects *myIDE* and *myIDE.ui*, contain a sub folder called *src-gen*. The language specific generated code is saved in this folder. This includes an ANTLR grammar, the resulting Java parser, and the data model.

5 The Curry IDE

The goal of this thesis is to develop an IDE for Curry which satisfies the specification defined in Chapter 3. For the Curry IDE, it is sufficient to *understand* Curry to reach our goal, it does not have to *compile* nor *execute* Curry modules. For those tasks, existing runtime systems can be integrated. If we look at the workflow of compilers described in Section 4.1.1, we realize that *understanding* Curry means building a compiler-frontend. Whereas the compiler-backend makes the language *executable*, we leave this functionality to existing runtime systems.

This section describes the development process of the Curry IDE. At first, the reasons for the usage of the Xtext-framework are listed in Section 5.1. Afterwards, the conceptual considerations and the implementation is documented step by step. The main part of the implementation will be the definition of the Xtext-grammar, which is used to generate the basis for the Curry IDE (see Section 4.4). Therefore, we analyze its suitability for the Xtext-framework, solve the detected problems, and present a basic Xtext-grammar in Section 5.2.

The generated IDE can parse Curry modules correctly, i.e., it has the ability to detect and display syntactical errors as well as create a model representation of the input program. The model representation is used to validate the input program semantically, for instance, a typical validation is to check cross-references. Cross-references are links between elements of the language, for example, a function call is linked to its declaration using the function name as an identifier. This is quite complex with Curry due to the fact that the identifiers for different types of elements, like data types and data constructors, may have the same identifier. Moreover, Curry has local scopes as well as a complex im- and export mechanism. Section 5.4 describes how the linking is realized. After we have made these two important steps, it is possible to modify the generated IDE to satisfy all requirements defined in Chapter 3. Xtext generates a basic implementation for the majority of the target features which are well integrated into the IDE. Therefore, these implementations only have to be modified to match the expected behavior. This finishing of the generated features is documented in Section 5.5. At the end of this chapter, we have created a fully-functional IDE for Curry according to the specification of Chapter 3 which is shown on the basis of testing an existing Curry project in Section 5.6.

5.1 Why use Xtext?

One main approach of software engineering is to reuse pieces of software when possible. For this reason, it is common practice to use frameworks and generators when it comes to functionality which has to be implemented again and again. This makes the whole development process faster, more maintainable, and less error-prone, i.e., more efficient. A prime example is a parser generator, which provides a fully-functional parser by defining the grammar of the target language instead of programming it from scratch.

The same goes for the standard functionality of graphical user interfaces as well as the common features of IDEs. As described in Section 4.4, Xtext combines a parser generator and the Eclipse RCP technology to provide a working IDE for a specific language. Furthermore, Xtext is an official Eclipse project which has been awarded as the most innovative Eclipse project 2010[1]. Ever since, it has become more and more popular and matured under active development. The framework promises very high productivity, since it generates an IDE with all the features working out-of-the-box by only declaring an Xtext-grammar. However, Xtext is intended to be used for domain specific languages, which are generally not as complex as a general purpose language like Curry. It is not sufficient to translate the Curry grammar from the Curry report [Han12] to an Xtext-grammar and expect the framework magic to generate the IDE we aim at. As we will see in Section 5.2.2, *semantic predicates* would simplify the parser construction for Curry, but Xtext supports only syntactic predicates. However, Xtext is very flexible and allows a lot of modifications to satisfy our requirements. Overall, the usage of the Xtext-framework promises to have a lot of benefits including very high productivity.

5.2 The Grammar

The grammar definition is the heart of the whole development process. It serves as the basis for Xtext to generate the stub of the IDE. It is used to generate a parser and a corresponding data model for Curry. At runtime, the parser reads the input program and creates a model representation for it, which is basically a kind of annotated syntax tree[2]. All features we aim at can be realized by working with this model representation of the input program. At first, the Xtext grammar for Curry is defined. As Curry is a powerful general purpose language containing a lot of *syntactic sugar*[3], this task is quite complex.

[1] `http://www.eclipse.org/org/press-release/20100322_awardswinners.php`, last visited August 29, 2012

[2] We will see that Xtext generates a syntax graph instead of a syntax tree, because it contains cycles due to cross-references.

[3] syntax that is designed to make things easier to read or to express

At the end of this section, we present an Xtext-grammar which can be used to generate an IDE that parses Curry modules correctly, this is, it detects syntactical errors. However, semantic errors are not detected. This kind of validation is the topic of Section 5.4.

5.2.1 Lexicon

A lexicon is a set of words that serves as the basis for any syntax. The syntax defines valid combinations of words from the lexicon by grammar rules. Before we start to develop the grammar for Curry, we have to define the lexicon:

Curry's lexicon is defined as follows [Han12]:

- Every word beginning with a letter followed by any number of letters, digits, underscores, and single quotes, these words are called *identifiers* (ID)
- Any string of characters from the string "˜!@#$%ˆ&*+-=<>?./|\:", we call these words *infix operators*[4]
- Any word from (ID) enclosed in '...' like 'mod' is also called *infix operator*
- The strings "(", ")", "[", "]", "@", "{", "}", ";", "{-" and "-}", newlines, and whitespaces (blanks and tabs)
- Strings, characters, numbers, and floats like *"abc"*, *'a'*, *36*, and *3.14159*.

Moreover, the case of words matters, so that *abc* and *Abc* are two unequal identifiers. Curry supports four *case modes* which define different constraints on the case of particular identifiers. The modes are *prolog mode*, *gödel mode*, *haskell mode*, and *free mode*.

We implement the *haskell mode*, later it may be possible to weaken the constraints towards *free* mode. This seems to be a good approach, because the constraints prohibit ambiguities and, hence, facilitate the parsing.

As the name suggests, the *haskell mode* corresponds to the definitions for the lanuguage *Haskell*. The Haskell Report 2010 [Mar] defines the identifiers as follows:

There are six kinds of names in Haskell: those for variables and constructors denote values; those for type variables, type constructors, and type classes refer to entities related to the type system; and module names refer to modules. There are two constraints on naming:

Names for variables and type variables are identifiers beginning with lowercase letters or underscore; the other four kinds of names are identifiers beginning with uppercase letters.

[4] There are some exceptions which are mentioned later.

5.2.2 Basic Xtext Grammar

We start to develop the Curry IDE by defining a basic grammar that serves as a valid input for Xtext and sticks close to the grammar presented in [Han12, pp.67-71]. Afterwards, it is enhanced gradually with further validation and features. In the following, we will refer to the grammar defined in [Han12, pp.67-71] by the name *CRG* (Curry Report Grammar). The notation of Xtext grammars is really similar to EBNF, hence, the translation of *CRG* to a corresponding Xtext grammar is not that difficult. The *CRG* does not include the following non-terminal symbols defining classes of identifiers: *ModuleID*, *TypeConstrID*, *DataConstrID*, *TypeVarID*, *InfixOpID*, *FunctionID*, and *VariableID*.

All of these classes except *InfixOpID* consist of an initial letter, followed by any number of letters, digits, underscores, and single quotes. To satisfy the *haskell* case mode, we introduce one terminal class for each identifier type:

```
1 terminal ID_UPPER:
2   'A'..'Z' ('a'..'z'|'A'..'Z'|'_'|'0'..'9'|'\'')*;
3 terminal ID_LOWER:
4   ('a'..'z') ('a'..'z'|'A'..'Z'|'_'|'0'..'9'|'\'')*;
```

Consequently, the rules for the identifier classes look as follows:

```
1 ModuleID: ((ID_UPPER | ID_LOWER) '.')* ID_UPPER
2 TypeConstrID:   ID_UPPER
3 DataConstrID:   ID_UPPER
4 TypeVarID:  ID_LOWER
5 FunctionID: ID_LOWER
6 VariableID: ID_LOWER
```

In [Han12], the infix operators are described as any string of characters from the string “˜!@#$%ˆ&*+-=<>?./|\:” or another identifier enclosed in ‘..‘ like ‘mod‘:

```
1 terminal INFIX_OP_ID:  ('~'|'!'|'@'|'#'|'$'|'%'|'^'|'&'|'*'
2     |'+'|'-'|'='|'<'|'>'|'?'|'.'|'/'|'|'|'\' | ':')+
3   | | '`' (ID_LOWER | ID_UPPER) '`'
```

The following tokens are defined to recognize whitespaces and newlines:

```
1 terminal WS: ' ' | '\t'
2 terminal NL: ( '\r' | '\n' )+
```

Curry supports single-line as well as multiple-line comments. Single-line comments start with '−−' and end by the end of the line. Multiple-line comments are embraced by '{-' and '-}'. The corresponding rule definition looks as follows:

```
terminal COMMENTS:  '--' !('\r' | '\n')*
    | '{-'->'-}';
```

Values like *'a'* or *'0'* belong to the terminal class *CHAR*. Special characters can be denoted with a leading backslash, e.g., '\n'. Moreover, characters can be written as decimal or hexadecimal value, e.g., '\010' (decimal) or '\xA4' (hexadecimal). For strings, we define the terminal class *STRING* which consists of inputs like *"Hello"*, note that, strings may contain escaped characters as well.

For numbers, like *5* and *555* the terminal class *NATURAL* is defined. Floating point numbers, like *3.14159* or *5.0e-4*, belong to the terminal class *FLOAT*.

We group these terminal classes by the rule *Literals*:

```
terminal CHAR:
  "''"
  | "'" !("'"|'\\') "'"
  | "'" '\\' ('b'|'t'|'n'|'f'|'r'|'u'|"\\"|'"'|"'") "'"
  | "'" '\\' ('0'..'9')('0'..'9')('0'..'9') "'"
  | "'" '\\' 'x' ('a'..'f'|'A'..'F'|'0'..'9')
          ('a'..'f'|'A'..'F'|'0'..'9')
    "'";
terminal STRING:
'"' ('\\' ('b'|'t'|'n'|'f'|'r'|'u'|'"'|"'"|'\\')
    | '\\' ('0'..'9') ('0'..'9') ('0'..'9')
    | '\\' 'x' ('a'..'f'|'A'..'F'|'0'..'9')
                ('a'..'f'|'A'..'F'|'0'..'9')
    | !('\\' | '"'))*
  '"';
terminal NATURAL:
  ('0'..'9')+;
terminal FLOAT:
  NATURAL '.' NATURAL ('e' '-'? NATURAL)?;

Literal:  STRING | CHAR | NATURAL | FLOAT;
```

The entry rule of the grammar is called *Module*. We specify the list of terminal symbols we are not interested in by declaring them as *hidden*. This means that they can occur everywhere in the input. If any grammar rule that is applied directly or indirectly by the entry rule needs to see one of them, it can rewrite the list of hidden terminals.

```
Module hidden(WS, NL, COMMENTS, ANY_OTHER):
  'module' ModuleID (Exports)? 'where' Block
```

Like the *CRG*, this basic grammar does not support the full layout capabilities that Curry provides. The brackets '{' and '}' are used to structure code blocks and the semicolon ';' is used to mark the end of a line within a code block. The remaining grammar rules are defined as follows:

```
1  Exports : '(' Export? (',' Export)* ')';
2  Export: QFunctionName
3        | QTypeConstrID
4        | QTypeConstrID '(..)'
5        | 'module' ModuleID;
6
7  Block:  '{'
8        ( ImportDecl ';')*
9        ( FixityDeclaration ';')*
10       (BlockDeclaration ';')*
11       '}';
12
13 ImportDecl:   'import' ('qualified')? ModuleID
14                   ('as' ModuleID)? (ImportRestr)?;
15 ImportRestr:  '(' Import ( ',' Import)* ')'
16       | 'hiding' '(' Import ( ',' Import )* ')';
17 Import:   FunctionName
18       | TypeConstrID
19       | TypeConstrID '(..)';
20
21 BlockDeclaration:   TypeSynonymDecl
22       | DataDeclaration
23       | FunctionDeclaration;
24 TypeSynonymDecl:  'type' SimpleType '=' TypeExpr;
25 SimpleType:     TypeConstrID TypeVarID*;
26 DataDeclaration:  'data' SimpleType '=' ConstrDecl
27                                     ('|' ConstrDecl)*;
28 ConstrDecl:     DataConstrID SimpleTypeExpr*;
29 TypeExpr:     SimpleTypeExpr ('->' TypeExpr)?;
30 SimpleTypeExpr: QTypeConstrID SimpleTypeExpr*
31       | TypeVarID
32       | '_'
33       | '('')'
34       | '(' TypeExpr (',' TypeExpr)* ')'
35       | '[' TypeExpr ']'
36       | '(' TypeExpr ')';
37 FixityDeclaration:
38         FixityKeyword Natural INFIX_OP_ID ( ',' INFIX_OP_ID)*;
39 FixityKeyword:    'infixl' | 'infixr' | 'infix';
40 Natural:      DIGIT | DIGIT Natural;
```

```
41 FunctionDeclaration: Signature | Equat;
42 Signature:      FunctionNames '::' TypeExpr;
43 FunctionNames:    FunctionName (',' FunctionName)*;
44 FunctionName:   '(' INFIX_OP_ID ')' | FunctionID;
45 Equat:    FunLHS '=' Expr ( 'where' LocalDefs)?
46       | FunLHS CondExprs ('where' LocalDefs)?;
47 FunLHS:   FunctionName SimplePat*
48       | SimplePat INFIX_OP_ID SimplePat;
49 Pattern:  QDataConstrID SimplePat SimplePat* (':' Pattern)?
50       | SimplePat (':' Pattern)?;
51 SimplePat:  VariableID
52       | '_'
53       | QDataConstrID
54       | LITERAL
55       | '(' ')'
56       | '(' Pattern ',' Pattern (',' Pattern)* ')'
57       | '(' Pattern ')'
58       | '[' (Pattern (',' Pattern)*)? ']'
59       | VariableID '@' SimplePat;
60 LocalDefs:      '{'
61            ValueDeclaration ( ';' ValueDeclaration )*
62          '}';
63 ValueDeclaration: FunctionDeclaration
64       | PatternDeclaration
65       | VariableID (',' VariableID)* 'free';
66 PatternDeclaration:
67       Pattern '=' Expr CondExprs?;
68 CondExprs:      '|' Expr '=' Expr CondExprs?;
69 Expr: '\\' SimplePat SimplePat* '->' Expr
70       | 'let' LocalDefs 'in' Expr
71       | 'if' Expr 'then' Expr 'else' Expr
72       | 'case' Expr 'of' '{' (Alt (';' Alt)*)? '}'
73       | 'fcase' Expr 'of' '{' (Alt (';' Alt)*)? '}'
74       | 'do' '{' (Stmt ';')* Expr '}'
75       | Expr QInfixOpID Expr
76       | '-' Expr
77       | FunctExpr;
78 FunctExpr:      BasicExpr (BasicExpr)*;
79 BasicExpr:  QVariableID
80       | '_'
```

```
81         | QDataConstrID
82         | QFunctionID
83         | '(' QInfixOpID ')'
84         | (LITERAL | Natural)
85         | '(' ')'
86         | '(' Expr ')'
87         | '(' Expr (',' Expr)* ')'
88         | '[' (Expr (',' Expr)*)? ']'
89         | '[' Expr (',' Expr)? '..' Expr? ']'
90         | '[' Expr '|' Qual (',' Qual)* ']'
91         | '(' Expr QInfixOpID Expr')'
92         | '(' QInfixOpID Expr')';
93 Alt:
94           Pattern '->' Expr ('where' LocalDefs)?
95           | Pattern GdAlts ('where' LocalDefs)?;
96 GdAlts:       '|' Expr '->' Expr GdAlts?;
97 Qual:       Expr
98           | 'let' LocalDefs
99           | Pattern '<-' Expr;
100 Stmt:       Expr
101           | 'let' LocalDefs
102           | Pattern '<-' Expr;
103
104 QVariableID:    ( ModuleID '.')? VariableID;
105 QFunctionName:    '(' INFIX_OP_ID ')' | QFunctionID;
106 QFunctionID:    ( ModuleID '.')? FunctionID;
107 QTypeConstrID:    ( ModuleID '.')? TypeConstrID;
108 QDataConstrID:    ( ModuleID '.')? DataConstrID;
109 QInfixOpID:     (ModuleID '.')* INFIX_OP_ID;
```

Listing 5.1: The remaining rule definitions of the Xtext-grammar.

This grammar is a mindless translation from the *CRG* to an Xtext grammar. As mentioned in Chapter 4, Xtext and ANTLR, support LL(*)-languages only, that is why left-recursive grammars are not supported. However, the grammar rules *Expr* and *FunctExpr* are left-recursive and have to be modified as described in Section 4.1. This operation can also be performed automatically with ANTLR Works (note that this modification has to be copied to the Xtext grammar manually). The resulting modified grammar rules look as follows:

```
1  Expr: ('\\' SimplePat SimplePat* '->' Expr
2        | 'let' LocalDefs 'in' Expr
3        | 'if' Expr 'then' Expr 'else' Expr
4        | 'case' Expr 'of' '{' (Alt (';' Alt)*)? '}'
5        | 'fcase' Expr 'of' '{' (Alt (';' Alt)*)? '}'
6        | 'do' '{' (Stmt ';')* Expr '}'
7        | '-' Expr
8        | FunctExpr)
9            (QInfixOpID Expr)?;
10 FunctExpr:      BasicExpr (BasicExpr)*;
```

Listing 5.2: Modified rules *Expr* and *FunctExpr*

After this modification, the Xtext editor does not display any errors for this grammar. However, there are some issues that are reported when running the Xtext generation process. As mentioned in Section 4.4, some grammar errors are outputted to the console. Those errors are the redirected output of the ANTLR generation process and can be debugged using ANTLR Works (cf. Section 4.2.1).

5.2.3 Resolving Grammar Issues

Many grammar errors can be solved by using backtracking. However, this will only suppress various errors and the grammar might still be ambiguous. As a result, the parse tree is not created as expected. Additionally, backtracking is really expensive (see 4.1). Instead of using it, we try to resolve the errors by grammar modifications.

Identifiers

Consideration of ANTLR Works for the debug version of the grammar shows an issue according to the rule *ModuleID*. We use ANTLR Works to visualize the problem. Figure 5.1 shows the syntax diagram for the related rules.

The green and red arrows represent two different alternatives which can be used to parse one and the same input, e.g., the token sequence [ID_UPPER, '.']. It can be either

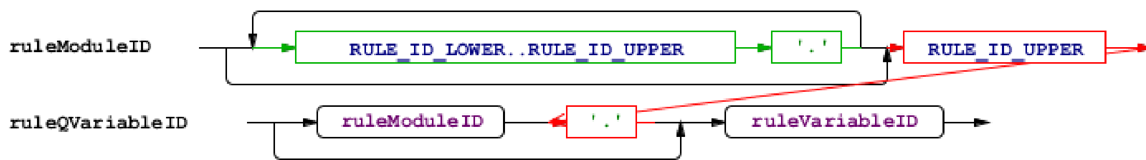


Figure 5.1: Inputs such as the token sequence [ID_UPPER, '.'] can be matched using multiple alternatives. Diagram generated by ANTLR Works.

the qualifying prefix of a *ModuleID* or the qualifying prefix of a *QVariableID* consisting of an unqualified *ModuleID* followed by a dot. Moreover, the dot can also be an infix operator. Consider the following example:

```
1 data Bool = True | False; -- Defined in module 'Prelude'
2 (.)    :: Bool -> Bool -> Bool
3 {- ... -}
4 f = True . False           -- This dot is an infix operator
5 f = Prelude.True           -- This dot is part of a qualifier
6 f = Prelude.True . Prelude.False  -- Mixed meaning
```

Listing 5.3: The different meanings of a dot in Curry.

The Curry code snippet in Listing 5.3 illustrates the different meanings of the dot. Line 4 shows an infix expression, whereas the expression in line 5 is a qualified data constructor. The expression in line 6 mixes the different meanings consisting of an infix expression with two qualified arguments. To distinguish these cases, no characters (e.g., spaces) are allowed between the dot and the names of modules and entities in qualified identifiers. On the other hand, if the infix operator of an infix expression starts with a dot, there must be at least one space or similar character behind the left expression [Han12]. Therefore, the lexer can make the decision which meaning a dot has. This is implemented by introducing a new class of terminal symbols called *QUALIFIER*. The rules for qualified identifiers are modified appropriately:

```
1 terminal DOT:   '.';
2 terminal QUALIFIER: (ID_UPPER | ID_LOWER) '.';
3
4 ModuleID:         QUALIFIER* ID_UPPER;
5 QVariableID:      QUALIFIER* VariableID;
6 QFunctionName:    '(' INFIX_OP_ID ')' | QFunctionID;
7 QFunctionID:      QUALIFIER* FunctionID;
```

```
8 QTypeConstrID:    QUALIFIER* TypeConstrID;
9 QDataConstrID:    QUALIFIER* DataConstrID;
```

Note that the rules *Import* and *Export* are affected as well, because they contain the anonymous terminal *'(..)'*. It is not enough to simply update this rule to *'(' DOT DOT ')'*. The token sequence '*DOT DOT*' will be recognized as an infix operator. To resolve this issue, we have to introduce a separate terminal *DOT_DOT*, which is of higher priority than *DOT*. Consequently, inputs like '..' are recognized as *DOT_DOT*. Now we can update the rules *Import* and *Export* to use the *DOT_DOT*:

```
1 terminal DOT_DOT: '..';
2 terminal DOT: '.';
3
4 Import: ...
5     | TypeConstrID '('DOT_DOT')';
6 Export: ...
7     | QTypeConstrID '(' DOT_DOT ')';
```

Though this fixes the dot-problem, there are still some remaining issues with identifiers of infix operators. Some symbols like '-', ':', '::', '@', '->', and '−−' are either unnamed terminals (since they are used in rule definitions) or terminals of the class of infix operator identifiers. The same applies to the token 'as', which is used as an unnamed terminal in the rule *ImportDecl* even though it is no keyword. The lexer will recognize those symbols as unnamed terminals due to priority setting and, consequently, the according identifiers will never appear in any token sequence.

This is desirable for the following tokens: '−−' (introduces comments), '\'(used in lambda expressions), '::' (used in signatures), '->' (used in lambda expressions and type expressions), '=' (used in equations), '@' (used in patterns), '|' (used in conditional expressions), and ':' (used as list constructor).

On the other hand, we want to allow the tokens 'as' and '-' to be used as identifiers as well. To resolve this issue, we have to introduce separate terminals for the dash and the 'as' and refer to them whenever necessary. The modifications look as follows:

```
1 terminal DASH:        '-';
2 terminal AS_KEYWORD:  'as';
```

The rules *Expr* and *ImportDecl* have to be updated replacing unnamed tokens ('-' or 'as') with the terminal class *AS_KEYWORD*.

Single line comments are another special case, for instance, we want inputs like '− − −\n' or '− − −−\n' to be recognized as comments. However, the lexer recognizes the '− − −' as an infix operator identifier, because it follows the principle of longest match (cf. Section 4.1.2).

Consequently, we prohibit specifically infix operators with the prefix '−−'. To do so, we introduce a helper class called *INFIX_OP_HELPER* and modify the definition of the terminal class *INFIX_OP_ID* as follows:

```
1 terminal INFIX_OP_HELPER:
2   '~'|'!'|'@'|'#'|'$'|'%'|'^'|'&'|'*'
3     |'+'|'='|'<'|'>'|'?'|':'|'/'|'|'|'\\';
4 terminal INFIX_OP_ID:
5   DASH? (INFIX_HELPER|DOT) (INFIX_HELPER | DASH | DOT )*
6   | '`' (ID_LOWER | ID_UPPER) '`';
```

Moreover, inheritance is not available for terminal symbols, so that the dash, the dot, and members of the class *INFIX_OP_HELPER* cannot be an infix operator identifier anymore. We need to introduce another rule to take these special cases into account, named *InfixOpIDWithSpecialCases*.

```
1 InfixOpIDWithSpecialCases:
2   INFIX_OP_ID
3   | INFIX_OP_HELPER
4   | DOT
5   | DASH;
```

All rules that directly refer to *INFIX_OP_ID* have to be updated to refer to *InfixOpID-WithSpecialCases* instead.

To make the token 'as' available as an identifier, all rules that refer to *LOWER_ID* need to allow the *AS_KEYWORD* as well.

Identifiers cause another issue in the rule *BasixExpr* (Basic Expression). The rule definition uses identifiers to distinguish between variables, data constructors, and functions. The problem here is that the rules *QFuntionName* and *QVariableID* are defined identically. The only way to distinguish between these alternatives was by using semantic or syntactic predicates. We only consider using syntactic predicates, as Xtext does not support semantic predicates. However, attributing these alternatives with syntactic predicates would cause one rule to be prioritized, which means one alternative would always be chosen and the other always omitted. To avoid this, the two rules can be merged. The correct semantic distinction will be made afterwards during the linking and validation phase. We implement this by introducing a new rule called *QFunctionOr-VariableID*:

```
1 QFunctionOrVariableID: QUALIFIER* (ID_LOWER | AS_KEYWORD);
2
3 BasicExpr:  QFunctionOrVariableID
4         | '_'
5         | QDataConstrID
```

```
6        | '(' QInfixOpID ')'
7        | LITERAL
8        ...
```

Now the case of the identifier (i.e., the corresponding terminal classes) helps the parser to choose the correct alternative, since identifiers of data constructors start with an upper case letter.

Rule Simplifications

Some of the existing errors occur because some rule definitions consist of alternatives that are partially identical and can be improved to be more compact. This is due to the fact that the grammar uses particular rule alternatives to describe their meaning. For instance, consider the rule *Equat* (Equation), see line 45 in Listing 5.1. The rule consists of two alternatives which start with the same rule, namely *FunLHS* (Function Left Hand Side). The rule can easily be rewritten so that the parser has to make a choice after the *FunLHS* rule has been applied. This avoids the need to look ahead and does not affect the set of recognized inputs. Additionally, the last optional part (local definitions) is identical in both alternatives and, hence, can be merged. The simplified rule can be defined as follows:

```
1 Equat:    FunLHS ('=' Expr | CondExprs) ('where' LocalDefs)?;
```

A similar case is the rule *Alt* (Alternative) as defined in Listing 5.1, line 93. Similar to the first definition of *FunLHS*, it is defined by two rule alternatives that are identical by a large part. It can be written more compact by merging the alternatives wherever possible. They only differ by an optional part, hence, the rule definition can be simplified as follows:

```
1 Alt:    Pattern GdAlts? '->' Expr ('where' LocalDefs)?;
```

We now look at the rules *SimpleTypeExpr* (Simple Type Expression, Listing 5.1, line 30, *SimplePat* (Simple Pattern, Listing 5.1, line 51), and *BasicExpr* (Basic Expression, Listing 5.1, line 79). These rule definitions include many alternatives for particular parenthesized terms. However, all of those alternatives start with a left parenthesis and end with a right parenthesis (or bracket). We can reduce the number of rule alternatives significantly, in return, the definitions become more complex and may be confusing:

```
1 SimpleTypeExpr: QTypeConstrID SimpleTypeExpr*
2        | TypeVarID
3        | '_'
4        | '(' (TypeExpr (',' TypeExpr)*)? ')'
5        | '[' TypeExpr ']';
```

```
6
7  SimplePat:  VariableID
8        | '_'
9        | QDataConstrID
10       | LITERAL
11       | '('(Pattern (',' Pattern)*)? ')'
12       | '[' (Pattern (',' Pattern)*)? ']'
13       | VariableID '@' SimplePat;
14
15 BasicExpr:  QFunctionOrVariableID
16       | '_'
17       | QDataConstrID
18       | '(' QInfixOpID ')'
19       | LITERAL
20       | '(' ((Expr ((',' Expr)+ | QInfixOpID Expr?)?)
21                      | QInfixOpID Expr)? ')'
22       | '[' (Expr ('|' Qual (',' Qual)*
23                  | (=>',' Expr)? ('..' Expr? | (',' Expr)*)))?
24            ']';
```

Listing 5.4: Compact definition of the rules *SimpleTypeExpr*, *SimplePat*, and *BasicExpr*.

If we take a close look at the last alternative of *BasicExpr* in Listing 5.4, line 15, we notice a syntactic predicate ('=>') in line 23. Consider the following situation: The parser has recognized a left bracket and an *Expr* so far and the following token is a comma. The parser cannot decide which path to take. The syntactic predicate leads the parser to apply the optional part "(',' Expr)?", whenever possible. Afterwards, the remaining decision can be made by the next token: '..', ',', or any other token.

Infix Expressions

The next crucial issue is caused by infix expressions. The rule definition *Expr* (Expression) contains some rule alternatives that end with recursive calls to itself. Additionally, every *Expr* may be an infix expression. This causes the definition to be ambiguous. Consider the following sample Curry code:

```
1  specialAdd a b = if a>0 then a else a + b
```

The current grammar definition allows multiple ways to parse this input. The relevant parts of the resulting syntax trees are depicted in Figure 5.2. The red lines mark the particular decision leading to the corresponding syntax tree.

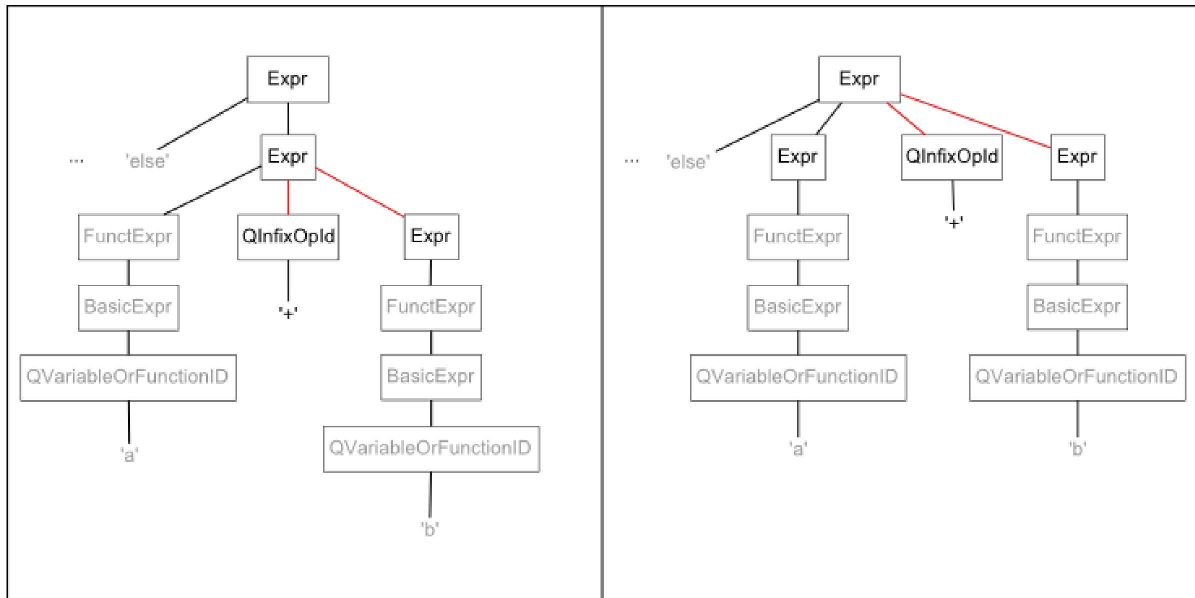


Figure 5.2: The ambiguous grammar definition allows creating multiple syntax trees for the same input.

The difficulty here is to decide which *Expr* is the left expression of the infix expression. It can be either the variable 'a' within the else-expression (left syntax tree in Figure 5.2) or a completed if-expression (right syntax tree in Figure 5.2). In Curry, the innermost expression has the highest associativity. In this case, the infix expression constitutes the else-expression, so the left syntax tree of Figure 5.2 represents the desired parsing result. Other results can be reached by the explicit use of brackets. Inserting an appropriate syntactic predicate is the method of choice to resolve this issue. This way the parser is lead to associate the infix operator with the innermost expression:

```
1 Expr: ...
2       (=> QInfixOpID Expr)?;
```

However, the definition of infix expressions is still ambiguous, because the associativities assigned to each infix operator as described in Chapter 2 are not taken into account. These associativities are not available until runtime (of the parser). To take them into account, semantic predicates are necessary – but, as pointed out before, they are not available in Xtext. In fact, it is not necessary for the Curry IDE to parse such infix expressions correctly. It is sufficient to build a flat list of infix expressions and correct the structure of the syntax tree later, if necessary.

Moreover, the rule *Expr* leads to an exponential runtime since it uses backtracking for deep nested infix operation expressions. This yields in a very bad user experience.

Parsing time for one expression, like the concatenation of 15 strings, can take up to more than one minute on a modern computer. Note that this is not meant to have any general validity, but it underlines that the exponential runtime is not acceptable for a good user experience. The reason for this problem is the need for backtracking when a sequence of infix operation expressions is parsed: every time an expression is followed by an infix operator, a new backtracking procedure is started. This is, the remaining part of the whole expression is parsed two times: at first, the parser *tries* to parse the optional infix operation part in backtracking-mode, and if the backtracking has been successful, it *applies* the optional infix operation part, so that the remaining part of the expression is parsed again in non-backtracking mode. This is done for every infix operator so that the second infix operator starts this procedure two times: the first time in backtracking-mode and again in non-backtracking mode of the backtracking procedure of the first infix operator. Consequently, the remainder of the whole expression is parsed 4 times.

Considering the example of a concatenation of 15 strings, the right-hand side of the last infix operation is parsed $2^{14} = 16384$ times. On the one hand, we need backtracking to parse the input, on the other hand we have to prohibit that such an exponential backtracking procedure is started. Therefore, we distinguish between expressions that may be the left-hand side of an infix operation and expressions which may be not. The idea is to let the first expression collect all following infix operations and prohibit the other expressions involved to be left-hand sides of an infix operation. Therefore, we split up the rule *Expr* and introduce the rule *ExprHelper* as follows:

```
1  Expr:
2    ExprHelper (=> QInfixOpID ExprHelper)*;
3
4  ExprHelper:
5    '\\' SimplePat+ '->' Expr
6    | 'let' LocalDefs 'in' Expr
7    | 'if' Expr 'then' Expr 'else' Expr
8    | 'case' Expr 'of' '{' (Alt (';' Alt)*)? '}'
9    | 'fcase' Expr 'of' '{' (Alt (';' Alt)*)? '}'
10   | 'do' '{' (=> Stmt ';')* Expr '}'
11   | DASH Expr
12   | FunctExpr;
```

As before, this will start one backtracking procedure per infix operator, however, those are not nested anymore and, hence, will not yield in exponential runtime.

Infix Constructors

In opposite to Haskell, on which Curry's syntax is based on, there are no infix constructors in Curry (cf. [Han12]). However, they will be available in the future. Currently, the

list constructor ':' constitutes the only exception, because it is already integrated into the language. Therefore, we will simply add it to the rules *Expr* and *BasicExpr* and do not care about the different meaning of infix operators and the constructor, since they are used in the same way:

```
1 Expr: ...
2       ExprHelper ((=> QInfixOpID | ':') ExprHelper)*;
3 BasicExpr:  ...
4     | '(' (QInfixOpID | ':') ')'
5     ...
6     | '(' ((Expr ((',' Expr)+ | (QInfixOpID| ':') Expr?)?)
7                   | (QInfixOpID| ':') Expr)? ')'
8     ...
```

Built-in Types

Curry provides some built-in types that are defined in the standard prelude. The Curry Report [Han12, Ch. 9] describes important operations for those types. Again, we are only interested in the effects on the syntax. Some of them are already taken into account: integer values, like "31", are considered as constructors (constants) of type *Int*. The same goes for values, like "3.14159", which are considered as constructors of type *Float*. To integrate these types, we have to allow the definition of data types without constructors by rewriting the rule *DataDeclaration*:

```
1 DataDeclaration:
2   'data' SimpleType ('=' ConstrDecl ('|' ConstrDecl)*)?;
```

External Functions

Curry also has a small number of built-in functions that are programmed in other programming languages. Therefore, Curry provides a simple interface (described in [Han12]) to connect those functions. Without going into detail here, we can integrate external functions into the rule *Equat* by using the keyword 'external':

```
1 Equat:  ...
2     | FunctionID 'external'
3     | '(' INFIX_OP_ID ')' 'external';
```

Resolve Remaining Issues

The remaining errors are mostly non-LL(*) decision problems caused by recursive rule invocations. These kind of problems can be resolved by insertion of appropriate syntactic

predicates. Again, we take a close look at the rule *Expr*. The definition of do-expressions in Listing 5.4 in line 6 is such a case. The rule *Stmt* (Statement), defined in Listing 5.1 in line 100, can be deduced to *Expr*, which is the last part in curly braces within the do-expression as well. When the parser has processed a token sequence such as ['do', '{'] and any number of statements followed by a semicolon, and the next incoming token initiates an *Expr*, the parser can not decide if it is a statement or the actual expression. By insertion of a syntactic predicate, the parser can be instructed to try to parse the *Expr* as a *Stmt*. If this does not work, it can be seen as the trailing expression:

```
1 Expr: ...
2       | 'do' '{' (=>Stmt ';')* Expr '}'
```

A very similar issue can be found in the rule definition of *SimpleTypeExpr*. The first rule alternative, defined in Listing 5.4 in line 1, contains a self-recursion. Again, we have to instruct the parser which path to take using a syntactic predicate:

```
1 SimpleTypeExpr: QTypeConstrID =>SimpleTypeExpr*
2         ...
```

Looking at the rules *Qual* (Qualifier within list comprehensions, defined in Listing 5.1 in line 97) and *Stmt* (Statement, defined in Listing 5.1 in line 100), we notice that their definitions are identical. The rules could be merged. However, since this duplication does not cause any problems, we will keep both definitions for the sake of clarity. Though, the definitions have to be modified, because there are problems associated with the decision which rule alternative to take. The first alternative consists of a rule call of *Expr*, which contains the definition of let-expressions. The second alternative defines a let-statement, that is in a sense an abbreviated version of the let-expression. Thus, when processing a 'let'-token it is not possible to distinguish between a let-expression and a let-statement. As in the previous cases, we instruct the parser to try to parse the let-statement first using a syntactic predicate. Additionally, the rules *Qual* and *Stmt* have an alternative that defines an assignment which starts with a *Pattern*. The problem here is that both, a *Pattern* and an *Expr*, can start with the same terminals (e.g. *ID_LOWER*, *ID_UPPER*, *LITERAL*, '(', '['). In such case, the parser should try to parse the assignment before trying to parse an *Expr*. Therefore, we insert the appropriate syntactic predicates in the rule definitions:

```
1 Qual:        Expr
2            | =>'let' LocalDefs
3            | =>Pattern '<-' Expr;
4 Stmt:        Expr
5            | =>'let' LocalDefs
6            | =>Pattern '<-' Expr;
```

Now we turn to the rule *ValueDeclaration*, defined in Listing 5.1 in line 63. In Curry, *ValueDeclarations* are part of local definitions in *let*- or *where*-blocks. They can be used to declare mutually recursive functions and definitions of constants by pattern matching. To distinguish between locally introduced functions and variables, local patterns are defined as a (variable) identifier or an application where the top symbol is a data constructor [Han12]. If the left-hand side of a local declaration is a *Pattern* (defined in Listing 5.1 in line 49), then all identifiers in this pattern that are not data constructors are considered as variables. This definition excludes the definition of 0-ary local functions since such a definition is considered as the definition of a local variable [Han12]. To satisfy this guideline, we instruct the parser to prefer *PatternDeclarations* to *FunctionDeclarations* by insertion of appropriate syntactic predicates:

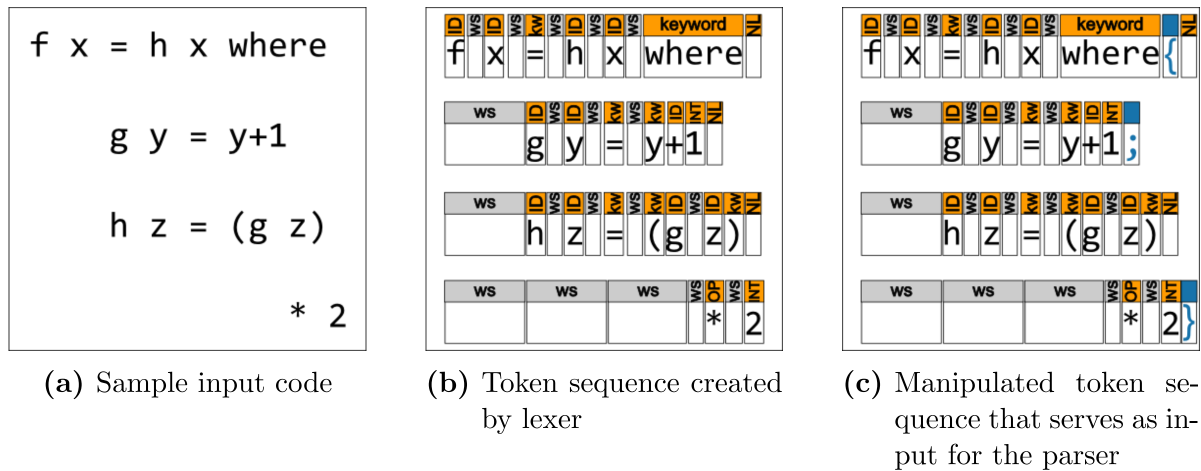
```
1
2 ValueDeclaration: =>PatternDeclaration
3                 | =>VariableID (',' VariableID)* 'free'
4                 | FunctionDeclaration;
```

The resulting modified grammar is free of errors and the Xtext generation process can be run successfully.

5.3 The Layout

The grammar introduced in Section 5.2.2 uses brackets to define the structure of blocks, because brackets serve as unambiguous markers for parsers even if they are nested. In Curry, the layout of code can be used to minimize the need for brackets. As described in Chapter 2, in many cases indentations and line breaks can embody the same information provided by brackets. To implement this layout, the parser has to determine and remember the indentation of symbols, which is defined by the column number indicating the start of that symbol [Han12]. To implement this layout, semantic predicates would have been of great value. Nevertheless, we will kind of simulate them. The idea is to leave the latest version of the grammar as it is. We rather manipulate the token stream that serves as the output of the lexer and the input of the parser (see 4.1.1). Therefore, curly braces and semicolons are inserted appropriately into the original token stream. Figure 5.3 illustrates the single steps of processing a sample input to an appropriate token sequence.

Figure 5.3a shows a sample Curry program. The lexer will process this character string and output a token sequence as depicted in Figure 5.3b. We will manipulate this token sequence before it is processed by the parser. This manipulated token sequence contains appropriate curly braces and semicolons corresponding to the indentations of the original input (illustrated in Figure 5.3c).



(a) Sample input code **(b)** Token sequence created by lexer **(c)** Manipulated token sequence that serves as input for the parser

Figure 5.3: Layout example

Thanks to Xtext's great adaptability, we can modify the default parser of the generated IDE to use the manipulated token source. To do so, we implement a new Java class *CurryParserUsingManipulatedTokenSource*[5] that inherits from the generated *CurryParser*-class and overwrites the following method:

```
1  protected TokenSource createLexer(CharStream stream) {}
```

This method initiates the generated Curry lexer and starts it with the specified input character stream. The lexer implements the interface *TokenSource*, which is used for a lazy implementation of the lexer (see Figure 5.4). It simply specifies a method called *NextToken*, that takes no arguments and returns a *Token*. The parser will use this method to fetch the tokens successively during the parsing process. Consequently, we introduce an implementation for *TokenSource* and call it *ManipulatedCurryTokenSource*[6]. The idea is that this token source uses a delegate token source, which will be the lexer that is generated by Xtext (cf. Listing 5.5, line 4), to process the input character stream and manipulate it as described. Hence, the implementation of *CurryParserUsingManipulatedTokenSource* looks as follows:

```
1  protected TokenSource createLexer(CharStream stream) {
2    ManipulatedCurryTokenSource tokenSource =
3          new ManipulatedCurryTokenSource();
4    tokenSource.setDelegate(super.createLexer(stream));
5    return tokenSource;
6  }
```

[5]package de.kiel.uni.informatik.ps.curry.parser
[6]package de.kiel.uni.informatik.ps.curry.parser

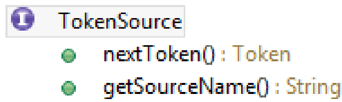


Figure 5.4: The interface *TokenSource*. TODO: Insert complete UML class diagram for relevant classes/interfaces

Listing 5.5: Modified CurryParser.

The modified parser has to be integrated into the generated IDE. Therefore, the following code is inserted into the *CurryRuntimeModule* (cf. Section 4.4):

```
1 public Class<? extends IParser> bindIParser() {
2     return CurryParserUsingManipulatedTokenSource.class;
3 }
```

Now we turn to the particular implementation of the *ManipulatedTokenSource*. As described in Section 2, we are interested in the keywords *let*, *where*, *do*, and *of*, because they introduce a new list of syntactic entities. Any item of such a list starts with the same indentation as the list. Lines with only whitespaces or an indentation greater than the indentation of the list continue the item in its previous line. Lines with an indentation less than the indentation of the list terminates the entire list. Moreover, a list started by *let* is terminated by the keyword *in* (cf. [Han12]). In the following we will call a particular list indentation *indentation level*.

A minor grammar modification is necessary to be able to refer to the curly braces, tabs, and semicolons as tokens. However, we only name these tokens, the language defined by the grammar stays the same:

```
1 terminal INDENT:
2   '{';
3 terminal DEDENT:
4   '}';
5 terminal END_OF_LINE:
6   ';';
7 terminal INLINE_TAB:
8   '\t';
```

Additionally, every occurrence of these tokens is replaced with the corresponding terminal and in the entry rule *Module* the list of hidden terminals is updated:

```
1 Module hidden(WS, INLINE_TAB, COMMENTS, NL, ANY_OTHER) ...
```

To implement the layout guidelines, the indentation of a particular line has to be compared to the indentation level(s) that have not been closed. Therefore, we keep track of all active indentation levels using a stack. Additionally, the tokens are requested one by one from the delegate which is the lexer generated by Xtext. However, it will be necessary to look ahead some tokens and it is very important that the impact on the performance is as little as possible. That is why we use a FIFO[7]-buffer to save tokens from the delegate temporarily while looking ahead. The diagram in Figure 5.5 outlines the algorithm that is used to manipulate the token sequence appropriately.

The *ManipulatedTokenSource* is stateful, the state consists of the FIFO-buffer which is called *buffer*, the original lexer which is called *delegate*, and a stack of numbers which is used to save indentation levels. The parser will make subsequent calls to the method *nextToken* which is the entry point of the algorithm outlined in Figure 5.5.

At first, we check whether the buffer is empty or not, if it is not empty we simply return the first token from the buffer. This situation occurs after lookahead actions which are described in the following. If the buffer is empty, we fetch the next token from the delegate. This token is classified as one of the following three types:

- *Special keyword*: Tokens that introduce a new indentation level: *let*, *wehere*, *do*, and *of*.
- *Newline*: Line breaks mark the starting point of indentations.
- *No special keyword*: All other tokens are not relevant in the current situation.

We handle the case where the token is *no special keyword* by simply returning this token. The other two cases are more interesting:

Special keyword: If we have fetched a *special keyword*, we have to determine the indentation level of the corresponding list of syntactic entities. This is done by looking ahead, for this, we save the current token to the buffer and start to read and save the next tokens from the delegate to the buffer until we find a *relevant token*. A token is a *relevant token* if it is none of the following: *comment*, *curry doc*, *whitespace*, *inline tab*, or *newline*. The column of this relevant token defines the new indentation level which is added to the corresponding stack. Moreover, we insert a '{' into the buffer, add the last token to the buffer, and return the first token at the front of the buffer.

Newline: On the other hand, if the token is a *newline* we look ahead to find the next *relevant token*, determine its indentation, and compare it to the current indentation level (the number on top of the stack). We distinguish the following three cases:

- Indentation > current indentation level: This means that this line continues the previous line. Consequently, we do not have to insert any additional token.

[7] First In - First Out

- Indentation = current indentation level: If the indentation of this line equals the one from the previous line, we have to insert a separator (';') to the front of the buffer.
- Indentation < current indentation level: In this case, the current indentation level has to be closed. In fact, all indentation levels with greater indentations than the determined one have to be closed. That is, we insert an appropriate amount of '}' to the front of the buffer and remove all closed indentation levels from the top of the stack.

Finally, the last token is saved to the buffer and the first token from the buffer is returned.

However, the algorithm outlined in Figure 5.5 does not produce the correct result. Although the use of curly braces is replaced by the layout, brackets and parentheses are still allowed in expressions or to define lists for instance. The algorithm corresponding to Figure 5.5 does not take into account that new indentation levels can be initiated within parenthesized expressions. This is a special case, because all of those indentation levels have to be closed *before* the surrounding parentheses or brackets are closed. Basically, this can be handled by replacing the stack which is used to save the active indentation levels with a stack of stacks of indentation levels. Consequently, for every opening parenthesis a new stack of indentation levels is instantiated and just in front of the corresponding closing parenthesis every indentation level of the current stack is closed and the stack is removed from the stack of stacks. This seems to be more a problem of implementation than a conceptual one, consequently, we will not describe the modified algorithm due to its complexity.

Overall, this solution to the layout problem seems to be a good approach. The good thing is that either the generated lexer as well as the generated parser can be used without modification. The manipulation is encapsulated and can easily be replaced or modified. However, the downside is that the error messages that appear due to an incorrect layout are not very meaningful.

5.4 Linking

In Chapter 4 we introduced the data model and linking mechanism provided by Xtext which constitutes a large part of semantic validation and serves as the basis for most IDE features. The insertion of additional information to make the parser build an appropriate data model is quite monotonous. To establish linking, we have to enhance the grammar by appropriate definition of cross-links. We want to make the following elements of Curry referable: data types, data constructors, functions, infix operators, and variables. Data types, functions, and infix operators can be used in import and export restrictions. Additionally, data types can be referenced in signatures or other data type declarations. Data constructors, functions, and variables can be used on the right hand side of function

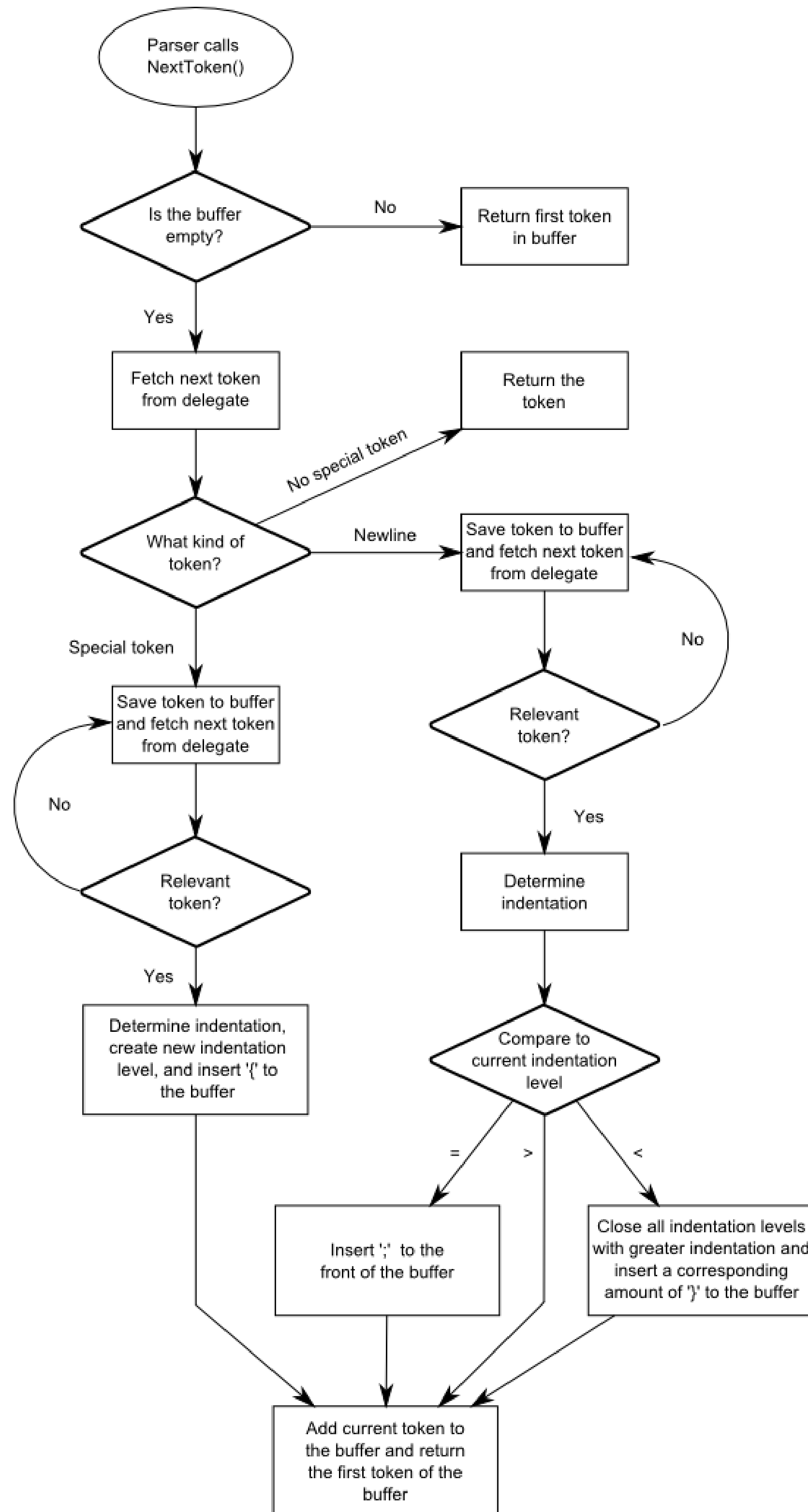


Figure 5.5: Outline of the simple version of the algorithm used in *ManipulatedToken-Source*.

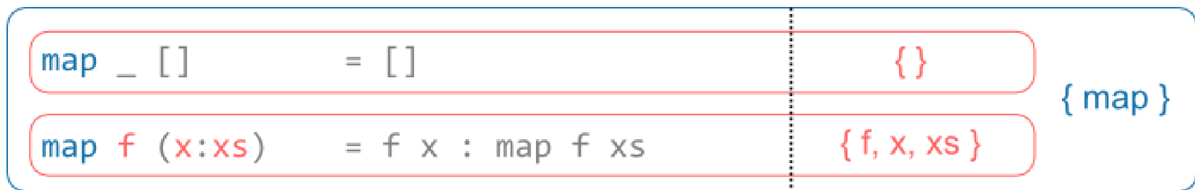


Figure 5.6: Visualization of the module scope (blue) and the local scopes (red) for a sample program.

equations, that is, in expressions. The modified Xtext grammar containing all of those cross-links is listed in Appendix A.

However, this modification is not sufficient because, in Curry, some elements are only referable in a special region of the code, such regions are called *scopes*. This allows the definition of local elements that should not be visible globally [Han12, §2.4]. The good thing is that this is a common concept in programming languages and DSL's, so that the developers of Xtext have included an easy to adapt scoping mechanism that allows to satisfy local scopes.

In the following, the linking mechanism is described in detail. Xtext employs a so called *linking service* that is responsible to find the right target element for cross-links. Therefore, it has to determine the scope of the element that contains the cross-reference. To do so, it asks the *scope provider* for the appropriate scope. In turn, the scope provides a list of all matching elements that are visible. The linking service may choose the right element from this list, if the list contains more than one element. Xtext provides default implementations of every component involved in the linking process. For Curry, these default implementations are not suitable, hence, we replace them with custom implementations. Again, this is supported by Xtext by the use of appropriate interfaces and dependency-injection. The corresponding interfaces are *ILinkingService*[8], *IScopeProvider*[9], and *IScope*[10].

5.4.1 Scoping

To implement the scopes with good performance, we use scope inheritance and scope nesting. For particular elements a new nested scope is created that contains all elements from its parent scope. An element which has not been assigned a particular scope, inherits the scope of its parent element. This approach allows to hide global definitions locally, called *shadowing*. Later, we will implement a warning for such situations.

We consider some examples to demonstrate the scoping mechanism of Curry.

[8] package org.eclipse.xtext.linking
[9] package org.eclipse.xtext.scoping
[10] package org.eclipse.xtext.scoping

Figure 5.6 illustrates the scopes that are created for a sample Curry program, the code snippet is taken from the standard prelude. The module scope is colored blue and contains the function declaration of *map*. The local scopes that are created for every equation are colored red. They inherit the elements from the module scope and extend it by its locally defined elements. This example illustrates two different ways of selecting the appropriate elements for a particular scope, we talk about *search strategies* in the following. For the module scope (blue), all function and data declarations have to be picked from the module body. Whereas, for the local scope of a function equation, the variables introduced in patterns on the left-hand side of an equation are added to the local scope. We call these strategies *ModuleStrategy* and *PatternStrategy*, respectively.

Local Definitions

The scoping gets more complex when local definitions come into play. A sample code with local definitions and corresponding scopes is depicted in Figure 5.7. Again, the blue rectangle represents the module scope and the red and green rectangles visualize local scopes. We can see that for the local scope of the function *take* (red) not only the variables are added, as described in the previous example, but also locally defined functions. The search strategy that is necessary to find local definitions is broadly similar to the *ModuleStrategy*, however, they differ in detail and, hence, we introduce a new strategy, namely the *LocalDefinitionStrategy*. For both strategies it is important not to pick elements from inner local definitions. In contrast to declarations on module level, local definitions can contain definitions of constants by pattern matching. The names introduced by these declarations are visible in the expression and the right-hand side of the local declarations (cf. [Han12]). Hence, the *LocalDefinitionStrategy* has to add them to the appropriate scope. Besides function equations, there are more cases where local definitions can occur and the *LocalDefinitionStrategy* is applied, they will be discussed later. For the function equations within the local definitions, the scope and search strategies are similar to declarations on module level.

Do Notation

In Curry, the *do notation* provides another syntax for I/O sequences (cf. [Han12]). Thus, the statements in *do-expressions* have to be treated in a special way. In Figure 5.8, the scoping for *do-expressions* is illustrated. Every assignment statement introduces a new scope for the following statement or the final expression of the do-expression. Hence, it is necessary to introduce a special search strategy for do-expressions (*DoExpression-Strategy*) as well as modify the creation of scopes for do-expressions.

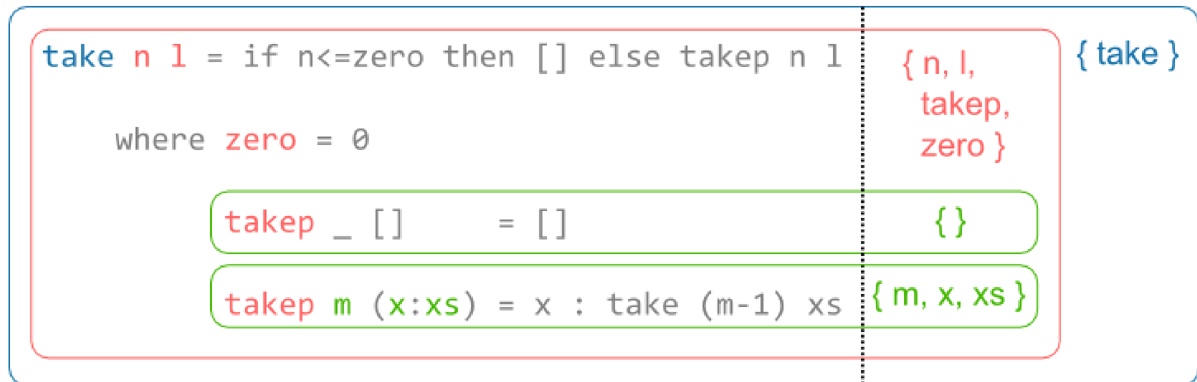


Figure 5.7: Visualization of the module scope (blue) and nested local scopes (red and green) for a sample program with local definitions.

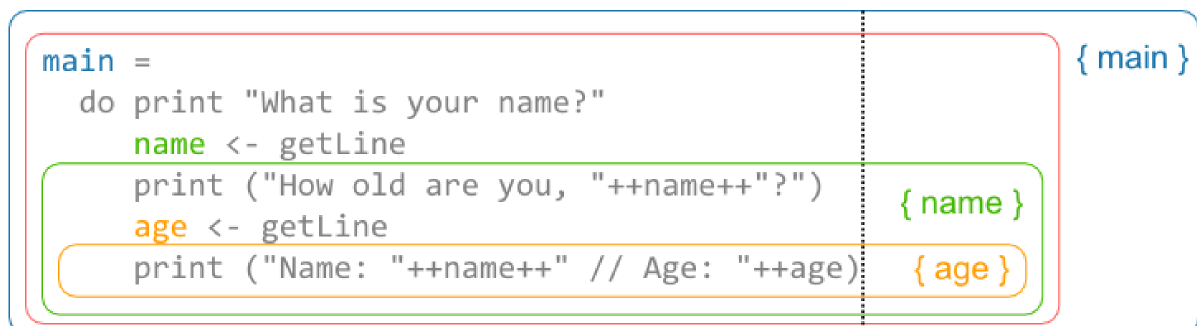


Figure 5.8: Visualization of the special scoping for *do-expressions*.

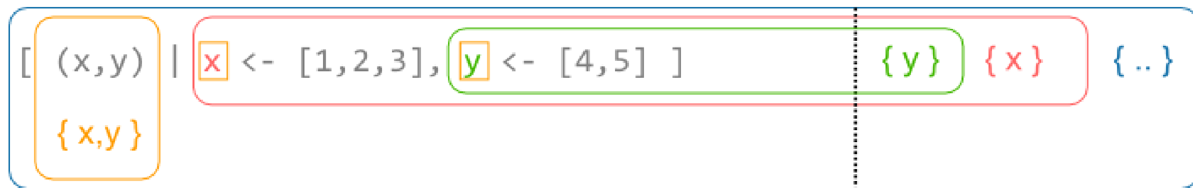


Figure 5.9: Visualization of the special scoping for *list comprehensions*.

List Comprehension

List comprehension are another special case, since they are a compact notation for lists, as described in Chapter 2. Figure 5.9 illustrates the corresponding scopes for this sample. The blue scope represents any parent scope of the expression. The scope for the expression e contains the variables that are introduced by the generators, that is why a special search strategy is necessary (*ListComprehensionStrategy*). The first qualifier's parent scope is the parent scope of the whole list comprehension expression. Hence, the qualifiers do not share a scope with the expression e. The qualifiers scopes are nested similar to the scopes of statements within do-expressions.

5.4.2 Implementation

To model Curry's scoping, we have to define *when* to create a new scope and *which* elements are contained. In Xtext, the scope provider is responsible for both concerns. It has to introduce a new scope for the following elements:

- Modules
- Data declarations
- Type synonym declarations
- All elements that contain local definitions
- Let expressions
- Lambda expressions
- Statements and expressions in do-expressions
- Expressions and qualifiers in list comprehensions

All other entities inherit the scope of their parent entity.

The scope provider employs the different strategies described above to collect the right elements that belong to a particular scope. Xext's scoping API provides the interface *IDefaultResourceDescriptionStrategy*[11] for custom implementations of such strategies.

In Table 5.1, the single scopes are listed and associated with the appropriate search strategy and search regions. We can see that the basic search strategies *Module-*, *Pattern-*, and *LocalDefinitionStragety* are sufficient to find all elements in every scope. However, depending on the scope the search strategies have to be applied to different parts of the AST. Therefore, we implement a custom strategy for every scope by selecting the appropriate region(s) of the AST and applying the basic strategies.

Table 5.1: The strategies used to find the elements within particular scopes.

Scope	Strategy	Search Region
Module	ModuleStrategy	Module body
Data declaration	PatternStrategy	Left-hand side
Type synonym declaration	PatternStrategy	Left-hand side
Equation	PatternStrategy LocalDefinitionStrategy	Left-hand side Local definitions
Lambda expression	PatternStrategy	Left-hand side
Let expression	LocalDefinitionStrategy	Local definitions
Alternative	PatternStrategy LocalDefinitonStrategy	Pattern Local definitions
Statement, expression (in do-expression)	PatternStrategy	Search in previous statement, if any
Expression in list comprehension Qualifier in list comprehension	PatternStrategy PatternStrategy	All qualifiers Previous qualifier, if any

Scope Caching

The linking process can be very expensive due to the calculation of the scope for every element of the input program. To determine the scope for a particular element the scope provider needs the parent scope to model the scope nesting. The result is that the scopes for some elements might be calculated many times. To prevent the scope provider from creating scopes for one particular element again and again, the scopes that have already been determined are stored in a cache. This approach is crucial to make the IDE usable at all. Xtext provides a special cache for scopes of a particular file (or resource) that is

[11] package org.eclipse.xtext.resource

invalidated on changes appropriately.

5.4.3 Import- and Export Mechanism

To provide a mechanism for encapsulation and reuse of definitions, Curry modules can import other modules. For a finer granularity, the imports can be restricted to either import particular elements only or hide particular elements from the imported module. Likewise, the elements which should be exported by a module can be restricted, to be able to model private declarations. In Curry, a module can even export elements which are imported from another module, in this case, we talk about *reexporting*.

As described in Chapter 4, Curry allows the usage of qualified names to eliminate ambiguities of identifiers. Again, this is a common approach adopted by many programming languages. Xtext supports this by using the data type *QualifiedName*[12] to identify elements. The default implementation provided by Xtext uses the dot to separate the segments of a qualified name, like it is the case in Curry. It is possible to use a custom implementation of *IQualifiedNameProvider*[13] to modify the construction of qualified names for particular entities. This is really handy, since the default implementation shipped with Xtext is not able to handle infix operators that do not consist of letters. By default, a qualified identifier like "Prelude.Maybe" is split by the dot, so that the qualified name consists of two segments: "Prelude" and "Maybe". Considering the infix operator '.', the qualified name provider would create an empty qualified name.

Xtext supports import and export functionality by the concept of global scopes. It provides the interface *IGlobalScopeProvider*[14] and uses special structures used to store exported elements of a resource, which in our case is a Curry module. An *IResourceDescription*[15] contains information about the resource itself which consists of its *URI*[16], a list of exported elements in the form of *IEObjectDescriptions*[17] as well as information about outgoing cross-references and qualified names it references. The cross-references contain resolved references only, while the list of imported qualified names also contains those names which couldn't be resolved. This information is leveraged by Xtext's indexing infrastructure in order to compute the transitive hull of dependent resources.

The *IEObjectDescription*[18] of an exported element contains the *URI* of the actual element and its *QualifiedName*. In addition, one can export arbitrary information using a *user data* map. This will be handy when dealing with the export of data types and whether its data constructors are also exported or not.

[12]package org.eclipse.xtext.naming
[13]package org.eclipse.xtext.naming
[14]package org.eclipse.xtext.scoping
[15]package org.eclipse.xtext.resource
[16]package org.eclipse.emf.common.util
[17]package org.eclipse.xtext.resource
[18]package org.eclipse.xtext.resource

The *IResourceDescriptionManager*[19] is responsible to compute the list of exported *IEObjectDescriptions*. Analogous to local scoping, we use different search strategies to find the appropriate elements to export for a particular module. Therefore, we introduce two search strategies, one for modules without export restrictions and another for modules with restricted exports. The search strategy for modules without export restrictions is broadly similar to the *ModuleStrategy*, with the difference that we make use of the *user data* to add a reference to the corresponding data type for data constructors. We will need this information when import restrictions come into play. The strategy used for modules with export restrictions simply ensures that the list of exported elements is in agreement with the restrictions. The Curry IDE contains appropriate implementations for the global scope provider, resource description manager, and both export strategies.

The global scope provider creates the export scope for a particular resource, which should contain a Curry module. We still need to define how to determine the resource for an import which is specified by the module name only. This is basically done by searching a file with the name of the imported module and an appropriate file extension (".curry" or ".lcurry"). This search is realized by the *CurryImportHelper*[20] and is discussed in detail later.

To enable the import and export mechanism in the Curry IDE, the linking service is enhanced as follows: To find a target element by a qualified name, it looks in the appropriate local scope for the target element; if no matching element has been found, the linking service starts to search the imported elements for the target element. This is done by asking the global scope provider for the export scope of the single imported modules. The linking service has to handle import aliases by modifying the qualified name of the search appropriately. When the linking service has found a potential candidate for the qualified name, it checks if the finding is in agreement with the import restriction defined by the importing module.

5.4.4 External Paths and Libraries

Up to now, we did not define which paths are used to find the resource containing the definition for an import by the module name. We start by describing the project structure used in the Curry IDE. A Curry project is basically a directory which contains any number of Curry modules and has an arbitrary subdirectory structure. All Curry modules can be imported by their module name from any other module within the Curry project regardless of their project relative path.

However, qualified names are allowed for modules to represent the project structure. Additionally, the Curry IDE associates every Curry project with one (or none) Curry library. In most cases, this should be the standard library provided by the Curry runtime

[19]package org.eclipse.xtext.resource
[20]package de.kiel.uni.informatik.ps.curry.utils

system that is used for compilation and execution. All modules of the associated Curry library (including those in subfolders) will be available.

In addition, external paths can be added to a Curry project to make all containing Curry modules available for the project. The first difference between an external path and a Curry library is that sub folders of external paths are not available by default and have to be added explicitly. The second difference is that all external paths are handed over to the employed Curry runtime system via an environment variable - this procedure is discussed later. Whereas the Curry library associated to a project should be the standard library of the employed Curry runtime system and, hence, it is available for compilation and execution anyway. Additionally, the following ordering is used for the different kind of paths, i.e., the first occurrence of a module in this search path is imported:

1. Curry project directory
2. The external paths that are specified for the project
3. The directories of the Curry library associated with the project

Implementation

To implement the global scoping of Curry, we use the specific components provided by Xtext for such scenarios which are described in the Xtext documentation [xte]. Xtext ships with an index which remembers all *IResourceDescription* and their *IEObjectDescription* objects. The index is updated by an incremental project builder. The global index state is held by an implementation of *IResourceDescriptions* (note the plural form) which is even aware of unsaved editor changes, such that all linking happens to the latest maybe unsaved version of the resources.

The index is basically a flat list of instances of *IResourceDescription* and does not know about any visibility constraints. They are defined by means of so called *IContainers*[21]. The *IContainerManager*[22] is responsible to determine which container a resource belongs to. For a given *IResourceDescription*, the *IContainerManager* provides the corresponding *IContainer* as well as a list of all *IContainers* which are visible for the corresponding resource. Xtext provides the implementations *StateBasedContainerManager*[23] and *WorkspaceProjectsState*[24] which keep track of the set of available containers and their relationships. They are based on plain Eclipse projects and let each project act as a container and the project references (which can be set in the UI) are the visible

[21]package org.eclipse.xtext.resource
[22]package org.eclipse.xtext.resource
[23]package org.eclipse.xtext.resource.containers
[24]package org.eclipse.xtext.ui.containers

containers. We use this implementation as the basis for the *CurryContainerManager*[25] which adds all external paths as well as the Curry library (if any has been associated with the corresponding project) according to the ordering described above.

Finally, the *CurryImportHelper* can use the *CurryContainerManager* to find the appropriate resource for a particular module import. Note that the *CurryContainerManager* is part of the UI project, hence, running the Curry parser generated by Xtext in isolation will not provide the full import mechanism of Curry. This is due to the fact that the necessary information regarding external paths and Curry libraries are available in the context of the UI only and the parser is not meant to run without the UI. However, we do not insert a dependency from the parser project to the UI project (cf. Section 4.4), because the *CurryContainerManager* is injected by the UI and the *CurryImportHelper* uses the corresponding interface only.

5.5 Finishing

Now that we have a functioning basis of the Curry IDE including syntax checking and linking, we are able to realize the features defined in Chapter 3. Xtext provides a basic implementation for most of them which has to be customized only, however, we will enhance the generated IDE with additional features as well.

5.5.1 Outline

Xtext creates an outline view for Curry modules and already integrates it into the Curry IDE. All we have to do is customize the information that should be displayed. This is done by the *CurryOutlineTreeProvider*[26] that builds the appropriate tree structure for Curry modules. This includes the insertion of virtual nodes for imports and exports and custom textual representations for all declarations.

5.5.2 Labeling and Documentation

The label provider is used to determine the textual representation of elements in various places including content proposals, find-dialogs and tooltips (of elements). For the Curry IDE, the custom implementation *CurryElementLabelProvider*[27] is employed to produce the textual representation for Curry elements. Besides, we want to integrate CurryDocs to display additional information as described in Chapter 3. We modify the grammar by adding the following terminal class:

[25] package de.kiel.uni.informatik.ps.curry.ui.container
[26] package de.kiel.uni.informatik.ps.curry.ui.outline
[27] package de.kiel.uni.informatik.ps.curry.ui.provider

```
1 terminal CURRY_DOC_COMMENT:
2   '---' !('\r' | '\n')*;
```

However, the CurryDoc comments are not part of the language, hence, we hide it in the entry rule:

```
1 Module hidden(WS, INLINE_TAB, COMMENTS, CURRY_DOC_COMMENT, ...
```

Note that the *ManipulatedCurryTokenSource* has to be modified slightly as well to ignore CurryDoc comments.

Additionally, we implement and integrate the *CurryDocumentationProvider*[28] which is used to create information that is displayed as tooltips for Curry elements in the editor. It simply looks for a preceding CurryDoc comment and uses it as the information to display. If no appropriate CurryDoc comment could be found, no description is provided at all.

5.5.3 Additional validation

The Curry IDE provides full syntactic validation, but the semantic validation is currently restricted to linking, as described above. Curry imposes some additional constraints to valid Curry programs. A major feature of Curry is that it is a strongly typed language [Han12]. The types of particular language elements have to be determined by type inference. The implementation of a type inferencer is complex. Besides, we made some simplifications regarding the grammar so that parts of the AST have to be modified or processed before the type inference can be started. For instance, consider infix expressions which are parsed as a flat list regardless of the associativity of the single infix operators. To determine the type of such expressions, the AST has to be modified according to the associativities.

These are the reasons why this thesis does not consider Curry's type system and the Curry IDE does not check if programs are well-typed [Han12]. However, the Curry IDE can be extended by type validation in the future.

To demonstrate the definition of additional validations, we implement a check for module names. The name of modules has to be equal to the name of the file in which they are stored. The module names can also be qualified, in this case, the qualification prefix of the module name has to correspond to the project relative path of the file containing the modules definition. Xtext generates and integrates a validator (*CurryValidator*[29]) by default. Further, Xtext provides an easy way to define errors and warnings that are integrated into the editor appropriately.

[28] package de.kiel.uni.informatik.ps.curry.ui.documentation
[29] package de.kiel.uni.informatik.ps.curry.validation

It is possible to add validations in a declarative way adding appropriate methods for every custom validation. Such methods have to be annotated with *@Check* and take one argument of any Curry data model type. The data model allows to traverse the whole AST to gather needed information.

For instance, the function for the module name check looks as follows:

```
1 @Check
2 public void checkModuleName(Module module) ...
```

Additionally, we add *warnings* for particular situations that indicate possible programming errors. For instance, we create warnings for unused variables or declarations of elements that shadow other elements. The corresponding implementation can also be found in the class CurryValidator.

There are more situations where warnings might be helpful for programmers, these can be added in the future.

Content Assist

Eclipse's concept of code completion is called *content assist*. In Chapter 3 the desired proposals are described.

Basically, the ability to make good suggestions for code completions consists of two steps. At first, the *context* of the proposal has to be determined. It contains information about the textual position, the AST elements involved, and the grammar.

In the second step, meaningful proposals depending on this context are generated. This is done by using appropriate scopes to find matching elements.

Accordingly, the implementation is split into two parts, the context is determined in *CurryContentAssistContextFactory*[30] and the proposals are created in *CurryProposalProvider*[31].

5.5.4 Curry Perspective

The Eclipse platform allows definition of so called *perspectives*. They can be created by the users or provided by plug-ins. A perspective defines the visible views, their arrangement, the set of available actions, and visible shortcuts. To provide a good user experience, the Curry IDE comes with a custom *Curry* perspective. The perspective is configured to show the custom components tailored for *Curry*.

[30]package de.kiel.uni.informatik.ps.curry.ui.contentassist
[31]package de.kiel.uni.informatik.ps.curry.ui.contentassist

Preferences and Project Properties

To manage various Curry libraries, a preference page is added to the Curry category (which is created by Xtext). The preference page is called *Libraries* and allows to add and remove arbitrary paths as Curry libraries. The characteristics and differences to external paths have been mentioned above. Moreover, a default Curry library can be chosen which is used as the associate library for new Curry projects automatically. The preference page is defined in the class *CurryLibrariesPreferencePage*[32]. The Curry libraries added on this page are stored in the configuration settings of the eclipse instance. This allows to persist the settings permanently. The storage functionality is encapsulated in the class *CurryUiHelper*[33] to be accessible from other parts of the UI.

Moreover, the *CurryUiHelper* stores the associated Curry library and all external parts of Curry projects as *persistent project properties*. Persistent properties have string values which are stored on disk across platform sessions. The value of a persistent property is a string which should be short (i.e., under 2KB).

Wizards

The Curry IDE is shipped with custom wizards that allow easy creation of new Curry projects and Curry modules. The Curry project wizard configures the project appropriately, i.e., the default Curry library (if defined) is associated with new Curry projects. Additionally, the wizard switches to the Curry perspective when it has been finished. The new-module-wizard adds the appropriate file extension (".curry") if not specified by the user, and may be extended in the future. For instance, the header could be created automatically for the new module.

Curry Project Explorer

The *Curry Project Explorer* shows Curry projects only. It adds a virtual layer to each project to display referenced projects, external paths and the associated Curry library. Moreover, appropriate actions are added to the context menu to add and remove external paths or referenced projects and change the associated Curry library.

Integrate Curry Runtime Systems

To integrate existing Curry runtime systems into the Curry IDE, they are executed as *external programs* with a special launch configuration. A launch configuration is a description of how to launch a program. The *CurryRuntimeSystemsLaunchConfigura-*

[32]package de.kiel.uni.informatik.ps.curry.ui.preferences
[33]package de.kiel.uni.informatik.ps.curry.ui.helper

tionTab[34] provides an input field for the command that should be used to start the Curry runtime system.

The settings from the launch configuration are passed to a so called launch configuration delegate. The custom implementation *CurryLaunchConfigurationDelegate*[35] starts a process whose in- and output streams are redirected to a console view within the Curry IDE. The launch delegate ensures that the environment variable *CURRYPATH* is set according to the subfolders and external paths of the Curry project to which the selected module belongs. Afterwards, the process executes the command specified in the launch configurations. The associated console view in the Curry IDE allows arbitrary interaction with the runtime system.

Eclipse Update Site

Eclipse Update Sites are repositories of plug-ins that allow easy installation and updating of Eclipse plug-ins. We create a separate Eclipse Update Site project where we add the *feature* of the Curry IDE that is created by Xtext and can be found in the project *de.kiel.uni.informatik.ps.curry.CurryIDE.sdk*. Now we can easily publish new versions of the Curry IDE via the update site. Moreover, the Curry IDE can be easily installed using the built-in software manager of Eclipse ("Help" → "Install New Software…") and updated using the corresponding Eclipse mechanism ("Help" → "Check for Updates") as well.

5.6 Testing

Now that we have implemented the Curry IDE, we want to make sure that it works as expected. We avoid any formal verification of the parser due to its immense effort, i.e., we do not know whether the generated parser will accept all Curry programs and decline all non-Curry inputs. As well, the syntax tree created during parsing process may not be built as expected.

To reduce this uncertainty, we consider the standard Curry libraries as test cases and try to parse them correctly. The sources of these libraries are available at:

- `https://git-ps.informatik.uni-kiel.de/curry/curry-libs` (last visited December 2, 2012).

We import the sources of these libraries (commit *a4fbbeb*) into the Curry IDE. After the workspace has been built, we see that the Curry IDE has detected 126 errors. At first sight, this seems to be a lot, but investigating the particular errors, we realize that most errors are directly or indirectly caused by incompatible case-modes. The libraries are

[34]package de.kiel.uni.informatik.ps.curry.ui.launch.tabs
[35]package de.kiel.uni.informatik.ps.curry.ui.launch

conform to the (default) case-mode, which is *free*, whereas the Curry IDE just supports the *haskell mode*.

As a consequence, we can ignore this kind of error and state the parser of the Curry IDE to pass the test. Furthermore, all UI components work as expected.

6 Integration of Curry Analysis Tools

The specification of the Curry IDE contains the requirement to integrate the *Curry Analysis Tool*. The usage of the tool via socket communication and the corresponding protocol have already been discussed in Chapter 3. The interesting part with respect to the Curry IDE is how the analysis can be started and how their results are visualized. The analyses as well as their output types are loaded dynamically into the IDE. Consequently, the Curry IDE has to integrate them generically.

6.1 Concept

At first, we abstract from the communication with the *Curry Analysis Tool* by implementing an appropriate client that understands the corresponding protocol. In addition, we add a generic context menu to the Curry editor that contains all dynamically loaded analyses. The analyses are grouped by the kind of analysis so that the context menu consists of one submenu for every kind of analysis containing all output types that are available for this kind of analysis. If there is more than one visualization available for one particular output type, the visualization can be chosen through another submenu. If no visualization is available for a particular result type, the plain result text will be displayed. This way, the user can start every Curry analysis provided by the *Curry Analysis Tool* and select the desired output type and visualization, respectively. However, due to the way the analyses are executed, it is reasonable to run them in a separate thread to avoid UI freezes. Therefore, we need a queueing mechanism to delay analysis requests occurring while another analysis request is pending. As well, we add a custom view to the Curry IDE, the *Curry Analysis View*, that displays either the state of the associated analysis request or its visualization of the analysis results. However, we want the Curry IDE to be as flexible as possible in terms of the visualization of the analysis results. The common case seems to be a visualization that processes the result and displays it within the *Curry Analysis View*. For instance, there might be a visualization for the output type *graph* that loads an image from the file system that has been created by an external tool and simply displays it in the *Curry Analysis View*. Likewise, it is easy conceivable that the information provided by an analysis should be displayed directly in the editor by means of warnings or errors.

To provide this kind of flexibility for the visualization of particular output types, we make use of the capabilities of the Eclipse platform. We define an interface for analysis

visualizations that supports our requirements to flexibility and define an *extension point* where its implementations can be hooked into the Curry IDE. If an Eclipse plug-in defines an extension point, it allows other plug-ins to add functionality based on the contract defined by the extension point. Consequently, the analysis visualizations are Eclipse plug-ins that implement the specified interface and add a contribution to the corresponding extension point. To handle plug-in dependencies properly, the interface and extension point are encapsulated in a separate Eclipse plug-in that is part of the Curry IDE.

6.2 Implementation

The implementation consists of two separated parts. At first the encapsulated Eclipse plug-in containing an interface for analysis visualizations as well as defining an appropriate extension point is introduced, we call this plug-in *Curry Analysis SDK*. Afterwards, we extend the Curry IDE to integrate the *CurryAnalysisTool* and the visualizations.

6.2.1 The Curry Analysis SDK

The Interface for Visualizations

The basis for the integration of Curry analyses is the interface *ICurryAnalysisVisualization*[1] that is depicted in Figure 6.1. Every visualization should provide a name that is used as the textual representation in the UI, hence, it does not have to be unique, though a good name helps the user to identify the visualization. The output type, which is a *String*, is used to match the visualization with corresponding analyses. If the *Curry Analysis Tool* provides any analysis with the same output type (the String compare is case-sensitive) as a visualization, it will be available to display the result of the analysis. Though, it is not accessible, if no analysis with the corresponding output type is available.

The *CurryAnalysisView* employs a tabbed view to display various analyses, each in a single tab. This makes the development of visualizations very easy, because developers do not have to care about the integration into the UI. Moreover, the tab is created when the analysis has been requested and an appropriate message indicating the status (e.g., loading, success, or error) of the analysis is displayed. For more flexibility this "hosting" mode within the *CurryAnalysisView* can be disabled by the appropriate implementation of the method *isStandalone()*. When the analysis has been finished successfully, the *CurryAnalysisView* checks whether the corresponding visualization is a *standalone* visualization by calling the method *isStandalone()*. Depending on the result value, the visualization is executed by calling the method *processResult(String[],Composite)* (hosted

[1] package de.kiel.uni.informatik.ps.curry.curryide.analysis.contract

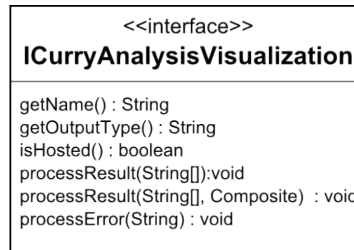


Figure 6.1: The interface for Curry analysis visualizations.

visualization) or *processResult(String[])* (standalone visualization). For hosted visualizations an additional argument of type *Composite*[2] is handed over, which is a reference to the ui-element that serves as the container for visualization.

There is a separate method, *processError(String)*, that is called when any error occurred during the execution of the requested analysis. The argument of type *String* contains an error message with additional information.

The Extension Point

Eclipse's concept of extensions and extension points offers great flexibility. An extension point is a declaration made by a plug-in to indicate that it is open to being extended with new functionality in a particular way. It is found in the *plugin.xml* file for the plug-in. The extension point for Curry analysis visualizations consists of the following definitions:

- The extension point ID: *de.kiel.uni.informatik.ps.CurryIDE.analysis.visualization*
- The human-readable name: *CurryAnalysisVisualization*
- An XML schema document, which defines the structure of the meta-data that extensions are required to supply.

The XML schema document contains the constraint that contributions to this extension point have to be implementations of the interface *ICurryAnalysisVisualization*.

6.2.2 Curry IDE Integration

Now, we discuss the integration of the Curry analyses into the Curry IDE. Figure 6.2 gives an overview of the Java classes involved, to which project they belong, and how they interact. An arrow indicates that the Java class at its source knows the class it points at. The *CurryAnalysisTool* is depicted as a cloud, this is meant to illustrate

[2]package org.eclipse.swt.widgets.Composite

that we do not have any knowledge about this tool except that it supports the specified protocol. The single components of the implementation are described in detail in the following.

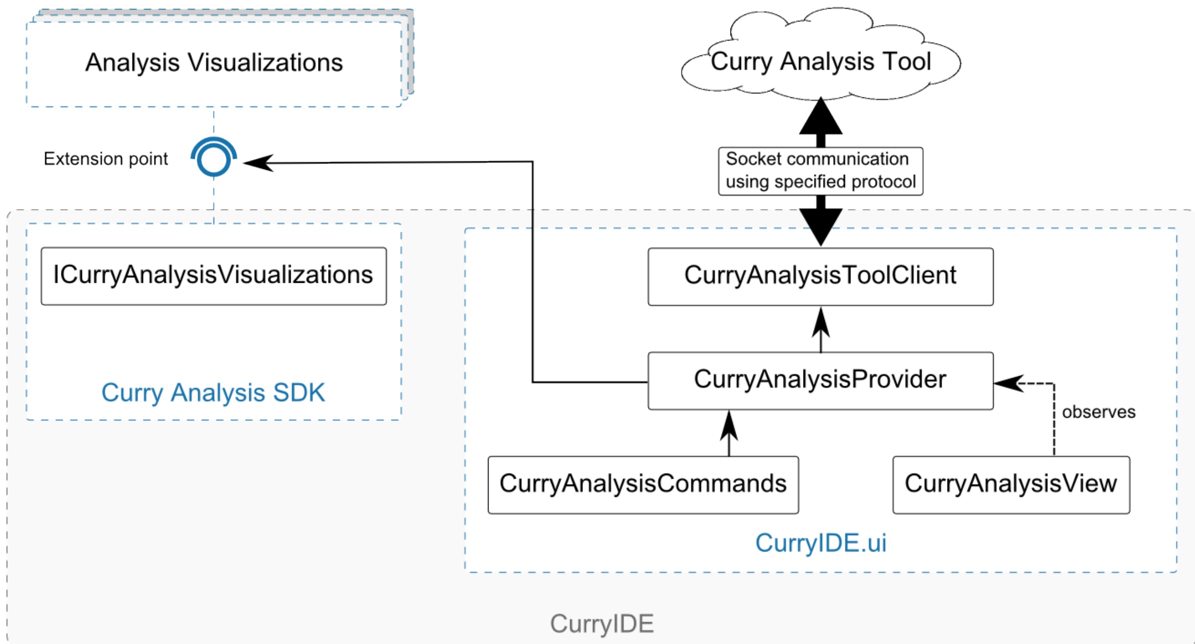


Figure 6.2: Overview of the relevant Java classes involved in the integration of the Curry Analysis Tool.

CurryAnalysisToolClient

The client used to abstract from the communication with the *Curry Analysis Tool* is implemented by the *CurryAnalysisToolClient*[3]. It is responsible for handling the socket connection and send valid commands in the sense of the protocol specified in Chapter 3.

CurryAnalysisProvider

To simplify the usage of the analyses in code, we introduce an additional abstraction layer, the *CurryAnalysisProvider*[4]. It is responsible for loading all available Curry analyses as well as all Curry analysis visualizations and link the output types of the analyses

[3]package de.kiel.uni.informatik.ps.curry.ui.curryanalysis
[4]package de.kiel.uni.informatik.ps.curry.ui.curryanalysis

to the corresponding visualization(s). Therefore, the analyses are loaded from the *CurryAnalysisToolClient*, they simply consist of a name and its output type.

The visualizations are loaded from the extension point specified above by using the *Extension Registry* provided by the Eclipse platform. This part is very simple, however, the visualizations are Java classes that we have to instantiate generically. Moreover, handling these contributions has to be done with some caution, because they consist of foreign code which might contain errors.

The *CurryAnalysisProvider* does also extend the class *Observable*[5] and fires change notifications when a new Curry analysis has been started. We will use this mechanism to keep the *CurryAnalysisView* up to date.

For the reasons mentioned above, the analysis requests are started asynchronously, however, we do not want to care about this fact when starting an analysis from the code. Hence, the *CurryAnalysisProvider* queues incoming analysis requests and employs a worker thread that executes them subsequently in the background using the *CurryAnalysisToolClient*. The worker is also responsible for setting the curry path of the analysis tool (cf. protocol in Chapter 3) before an analysis is executed.

CurryAnalysisCommands

In the next step, we implement a dynamic menu to start an analysis as described above and add it to the context menu of the Curry project explorer. The implementation of the dynamic menu can be found in the class *CurryAnalysisCommands*[6]. It creates the menu structure specified in Chapter 3 which is shown in Figure 6.3. The submenu consists of actions that call the *CurryAnalysisProvider* to start the selected Curry analysis with the selected output type and visualization.

Moreover, it has to determine which Curry element has been selected, this can be a module, a function, a data type, or a data constructor. Eclipse provides a *SelectionService* that can be used to easily get the current selection. For the editor this consists of a text selection, which does not contain any information about the type of the selected element. Hence, we have to find the corresponding model element in the current AST of the editor content. The model element provides all information necessary to start the appropriate analysis. It is passed to the *CurryAnalysisProvider* that extracts necessary information including the module and element name as well as the curry path that has to be set in advance (cf. protocol in Chapter 3).

CurryAnalysisView

Now, we consider the *CurryAnalysisView*, as mentioned above, it is used to host the visualizations of the Curry analyses. It consists of a tab view with one tab per visual-

[5]package java.util

[6]package de.kiel.uni.informatik.ps.curry.ui.menu

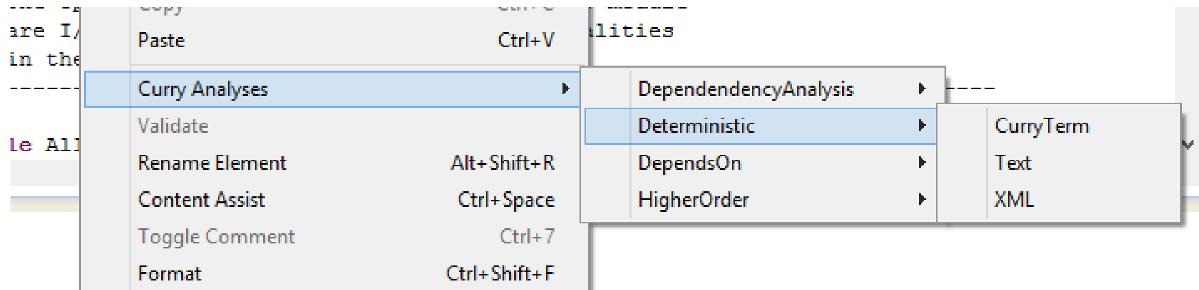


Figure 6.3: The context menu for some sample Curry analyses.

ization. The *CurryAnalysisView* is registered to observe the *CurryAnalysisProvider* so that it is notified when a new analysis is started.

6.3 A Sample Curry Analysis Visualization

In this Section, we demonstrate how to develop a new Curry analysis visualization. At first, we describe the prerequisites for development, afterwards, we create and, finally, deploy a visualization.

6.3.1 Prerequisite

Curry analysis visualizations are Eclipse plug-ins, hence, an appropriate Eclipse installation is necessary. The Curry IDE is based on Eclipse Juno (4.2). There are various packages available for downloading on the official website[7], for the development of Eclipse plug-ins the package *Eclipse for RCP and RAP Developers* is a good choice.

Moreover, the *Curry Analysis Visualization SDK* has to be accessible, either through an *Update Site* or by a *Jar-file*[8].

6.3.2 Development

Having a running Eclipse instance, the next step is to create a new *Plug-in Project* (from the category *Plug-in Development*). To access the interface and the extension point for the Curry analysis visualization, the *Curry Analysis Visualization SDK* has to be available. If it is provided by an *Update Site*, it can simply be installed using Eclipse's built-in mechanism ("Help" -> "Install New Software..") similar to the installation of the Eclipse IDE. If the *Curry Analysis Visualization SDK* is available as a *Jar*, the *Jar*-file has to be copied to the "plugins" directory within the Eclipse distribution that

[7] `http://www.eclipse.org/downloads/`, last visited December 3, 2012
[8] Java Archive

should be used to develop the Curry analysis visualization. Afterwards, the SDK can be added as a dependency to the project as follows: Go to the "plugin.xml", switch to the tab "Dependencies", in the area "Required Plug-ins" click the "Add..." button. The plug-in "de.kiel.uni.informatik.ps.curry.CurryIDE.analysis" should be available in the list of plug-ins (tip: type "curry" into the input field to find it), select it and save the "plugin.xml".

Define the Extension

Now, switch to the tab "Extensions" in the "plugin.xml" and click the "Add..." button within the area "All Extensions". Select the extension:

- de.kiel.uni.informatik.ps.curry.CurryIDE.analysis.visualization

(Again, the filter might help you find the extension by typing "curry".) Click "Finish", the extension should be added to the list of extension in the "plugin.xml". Right click on the extension and select "New"->"Curry Analysis Visualization". Select the item that has been created and choose any implementation of *ICurryAnalysisVisualization* for the *class* property in the area "Extension Element Details". If there is no implementation yet, click on the hyperlink "class*" to open a wizard to create an appropriate class. Figure 6.4 shows a snippet of the *Extension* tab from the "plugin.xml" containing an extension definition as described.

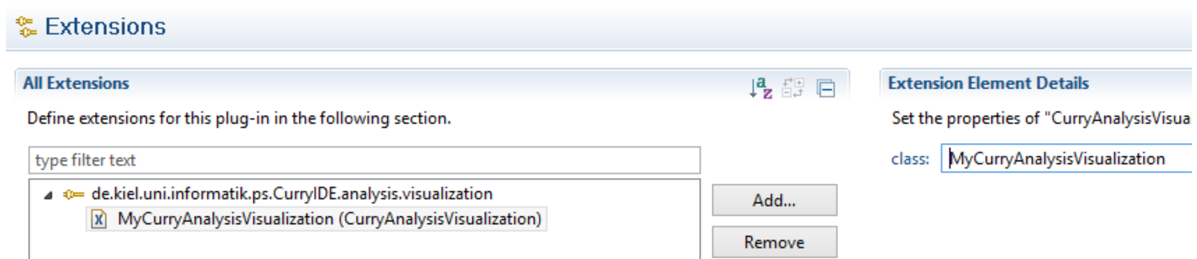


Figure 6.4: The screenshot shows a sample definition for an extension of the extension point from the *Curry Analysis Visualization SDK*.

Implement the Visualization

Now, the code of the visualization is the last thing that is missing. We demonstrate this by a sample implementation for a simple textual visualization of analysis results that is displayed in the *CurryAnalysisView*. We call this visualization "Text Visualization" and associate it with the output type "Text":

```
1 public String getName() {
2   return "Text Visualization";
3 }
4 public String getOutputType() {
5   return "Text";
6 }
```

The basic visualization is defined in the body of the method *processResult* (Note that we let the *isStandalone* method return 'false'). We can use the argument of type *Composite* to create an SWT based ui representation of our visualization and create a label for every line of the result and add it to the parent *Composite*:

```
1 public void processResult(String[] result, Composite parent) {
2   parent.setLayout(new RowLayout(SWT.VERTICAL));
3   for (String str : result) {
4     Label l = new Label(parent, SWT.NONE);
5     l.setText(str);
6   }
7 }
```

The visualization is completed and we can turn to the deployment to use it in the Curry IDE.

6.3.3 Deployment

To be able to use the visualization in the Curry IDE, we have to deploy it as an Eclipse plug-in. Therefore, we use the export wizard provided by Eclipse. For instance, it can be opened by right click on the visualization project in the *Package Explorer* and choose "Export...". On the first page of the wizard, the export destination has to be specified, we select "Deployable plug-ins and fragments" from the category "Plug-in Development". On the next page the plug-in has to be selected and the destination directory has to be defined. The export process can be completed by pressing the button "Finish". After a short period of time, the specified destination directory should contain a new folder called "plugins" which contains a Jar-file with the same name of the plug-in project.

This Jar-file is a runnable Java archive containing the *Curry Analysis Visualization* plug-in. To load it into the Curry IDE, this Jar-file has to be copied into the "plugins" directory of the Eclipse instance that hosts the Curry IDE.

The next time this Eclipse instance is started, the visualization is loaded and, from this time on, available for usage in the Curry IDE.

7 Conclusions

In this chapter, we summarize the development process of the Curry IDE (Section 7.1), discuss the overall result in Section 7.2, and propose future work in Section 7.3.

7.1 Summary

In the first part of this thesis, we used the framework Xtext to implement an IDE for Curry. To do so, we transformed the grammar presented in the Curry Report [Han12] into an LL(*)-grammar that serves as a valid input for Xtext. Moreover, we modified the generated lexer to support Curry's layout. Subsequently, we realized basic semantic validation using Xtext's linking mechanism, which includes the definition of the data model representing parsed input programs. We modified the UI components of the generated IDE as well as added new custom components to the UI.

The second part of this thesis deals with the integration of existing Curry analyses into the Curry IDE. We realized this using a generic approach that allows to load analyses from an external tool as well as corresponding visualizations using Eclipse's plug-in mechanism. Further, we showed how to implement such an analysis visualization and how to add it to the Curry IDE.

7.2 Discussion

The result of this thesis is a usable IDE for Curry in compliance with the specification defined in Chapter 3. The restriction of the supported case-modes is a limitation of the recognition strength but seems to be a reasonable decision to achieve the stated goals. As well, type checking is an important validation that is currently not available in the Curry IDE.

Though, the Curry IDE provides a lot of features that improve the development process of Curry programs. The initial parse times of complete projects including referenced libraries and external paths are acceptable and subsequent parse times of single modules are fast, i.e., errors are displayed instantly while typing.

The integration of Curry analyses and their visualizations is completely generic so that arbitrary contributions are possible during runtime of the Curry IDE.

Another great advantage is that the Curry IDE is based on Eclipse. This allows using the whole set of Eclipse-based tools within the Curry IDE. For instance, there is a Git[1] tool, called *EGit*, which integrates the version control system into Eclipse.

Overall, the objective of this thesis has been achieved and the Curry IDE improves the development process of Curry programs. It is possible to implement an IDE for Curry without any limitations or restrictions by not using Xtext. However, the development effort would increase significantly, because Xtext does not just generate large parts of the UI, but serves as an extensive library as well.

An installation guide of the Curry IDE can be found in Appendix B.

7.3 Future Work

In the future, the Curry IDE should be extended by type checking which provides important information for programmers. A short description of how to implement type checking is given in Section 5.5.3. Further, the implementation of all case-modes supported by Curry is desirable.

Being based on Eclipse, the Curry IDE can be extended in various ways to continuously improve the user experience of Curry programmers in the future.

[1]Git is a free and open source distributed version control system.

Appendices

A Complete Xtext Grammar

```
1  import "http://www.eclipse.org/emf/2002/Ecore" as ecore
2  generate curry
3      "http://www.kiel.de/uni/informatik/ps/curry/Curry"
4
5  Module hidden(WS, INLINE_TAB, COMMENTS, NL, ANY_OTHER):
6    {Module} ('module' moduleId=ModuleID
7                  (exports=Exports)? 'where')?
8              block=Block;
9
10 Exports : {Exports}'(' members+=Export?
11                      (',' members+=Export)* ')';
12
13 Export:
14   {FunctionExport}
15   (function=[FunctionOrVariable|FunctionID]
16   |infixExport?='('
17        function=[FunctionOrVariable|InfixOpIDWithSpecialCases]
18             ')')
19   |{DataTypeExport} dataType=[SimpleType|QTypeConstrID]
20         ('(' exportAllConstructors?=DOT_DOT ')')?
21   | {ModuleExport}'module' moduleId=ModuleID;
22
23 Block:  {Block}
24     INDENT
25       (ImportDeclarations+=ImportDecl END_OF_LINE)*
26       (FixityDeclarations+=FixityDeclaration END_OF_LINE)*
27       (BlockDeclarations+=BlockDeclaration END_OF_LINE)*
28     DEDENT;
29
30 ImportDecl:
31   {ImportDecl} 'import' (isQualified?='qualified')?
32                importedModuleId=ModuleID
33                (hasAlias?=AS_KEYWORD importName=ModuleID)?
```

```
34              (Restrictions+=ImportRestr)?;
35
36 ImportRestr:  {ImportRestr} (isHideMode?='hiding')?
37              '(' imports+=Import (',' imports+=Import )*
38                  ')';
39
40 Import:
41   {FunctionImport}
42       importedFunction=[FunctionOrVariable|FunctionID]
43   | {InfixOpImport}
44       '('
45           op=[FunctionOrVariable|InfixOpIDWithSpecialCases]
46         ')'
47   | {DataTypeImport}
48       importedDataType=[SimpleType|TypeConstrID]
49           ('('importAllConstructors?=DOT_DOT')')?;
50
51 BlockDeclaration:
52   TypeSynonymDecl
53   | DataDeclaration
54   | FunctionDeclaration;
55
56 TypeSynonymDecl:
57   'type' SimpleType '=' originalDataType=TypeExpr;
58
59 SimpleType:
60   {SimpleType} name=TypeConstrID typeVariables+=TypeVarID*;
61
62 DataDeclaration:
63   {DataDeclaration} 'data' type=SimpleType
64              ('=' constructors+=ConstrDecl
65                ('|' constructors+=ConstrDecl)*)?;
66
67 ConstrDecl: {DataConstructor}
68   name=DataConstrID typeExprs+=SimpleTypeExpr*;
69
70 TypeExpr:
71   {TypeExpr} simpleTypeExpr+=SimpleTypeExpr
72       ('->' simpleTypeExpr+=TypeExpr)?;
73
```

```
74  SimpleTypeExpr:
75    {ConstructTypeExpr} typeRef=[SimpleType|QTypeConstrID]
76                   =>nextTypeExpr+=SimpleTypeExpr*
77    | {VarTypeRef} name=TypeVarID
78    | {PlaceholderTypeExpr} '_'
79    | {TypeParenExpr}'(' (typeExprs+=TypeExpr
80                    (',' typeExprs+=TypeExpr)*)? ')'
81    | {TypeListExpr}'[' typeExprs+=TypeExpr ']';
82
83  FixityDeclaration:
84    infixType=FixityKeyword infixPrio=NATURAL
85        ops+=InfixOpIDWithSpecialCases
86          ( ',' ops+=InfixOpIDWithSpecialCases)*;
87
88  FixityKeyword:    'infixl' | 'infixr' | 'infix';
89
90  FunctionDeclaration:
91    Signature | Equat;
92
93  Signature:
94    {Signature} functions=FunctionNames '::' type=TypeExpr;
95
96  FunctionNames:
97    functions+=FunctionName (',' functions+=FunctionName)*;
98
99  FunctionName returns FunctionOrVariable:
100   =>'(' name=InfixOpIDWithSpecialCases ')' | name=FunctionID;
101
102 Equat:
103   {Equation} leftHand=FunLHS
104               ('=' rightHand=Expr | condExpr=CondExprs)
105               ('where' localDefs=LocalDefs)?
106   | {ExternalFunctionEquation}
107     function=[FunctionOrVariable|FunctionID] 'external'
108   | {ExternalInfixEquation}
109     '('
110       infixOp=[FunctionOrVariable|InfixOpIDWithSpecialCases]
111       ')' 'external';
112
113 FunLHS returns FunctionOrVariable:
```

```
114   {InfixOpLHS} leftPattern=SimplePat
115                name=InfixOpIDWithSpecialCases
116                rightPattern=SimplePat
117   | {FunctionLHS} name=FunctionID simplePatterns+=SimplePat*;
118 
119 Pattern:
120   {PatternDataConstructor}
121       name=[DataConstructor|QDataConstrID]
122       simplePatterns+=SimplePat+
123       (=>':' concPattern=Pattern)?
124   | {PatternSimple}
125       name=SimplePat
126       (=>':' concPattern=Pattern)?;
127 
128 SimplePat:
129   VariablePattern
130   | {EmptyPattern}'_'
131   | {DataPattern} constuctor=[DataConstructor|QDataConstrID]
132   | {LiteralPattern} Literal
133   | ParenPattern
134   | {ListPattern}
135       '[' (patterns+=Pattern (',' patterns+=Pattern)*)? ']'
136   | {AsPattern} varId=VariableID '@' simplePattenr=SimplePat;
137 
138 //This has to be a separate rule to be able
139 // to instantiate VariablePatterns from ValueDeclarations.
140 VariablePattern returns FunctionOrVariable:
141   {VariablePattern} name=VariableID;
142 
143 ParenPattern:
144   {ParenPatter}
145       ('(' (patterns+=Pattern (',' patterns+=Pattern)*)? ')');
146 
147 LocalDefs:
148   {LocalDefinition}
149       INDENT
150         valueDeclarations+=ValueDeclaration
151         ( END_OF_LINE valueDeclarations+=ValueDeclaration )*
152       DEDENT;
153 
```

```
154 ValueDeclaration:
155   {PatternDeclaration}
156     =>name=Pattern '=' expr=Expr
157       ('where' localDefs=LocalDefs)?
158   | FunctionDeclaration
159   | {FreeVariableDeclaration}
160       freeVariables+=VariablePattern
161       (',' freeVariables+=VariablePattern)* 'free';
162
163 CondExprs:
164   '|' cond=Expr '=' expr=Expr nextCondExpr=CondExprs?;
165
166 Expr:
167 ( {LambdaExpression}
168   '\\' simplePatterns+=SimplePat+ '->' expr=Expr
169   | {LetExpression}
170     'let' localDefinitions=LocalDefs 'in' expr=Expr
171   | {IfExpression}
172     'if' cond=Expr 'then' expr=Expr 'else' elseExpr=Expr
173   | {CaseExpression}
174     'case' expr=Expr 'of' INDENT
175       (alternatives+=Alt (END_OF_LINE alternatives+=Alt)*)?
176       DEDENT
177   | {FCaseExpression}
178     'fcase' expr=Expr 'of' INDENT
179       (alternatives+=Alt (END_OF_LINE alternatives+=Alt)*)?
180       DEDENT
181   | {DoExpression}
182     'do' INDENT (=>statements+=Stmt END_OF_LINE)*
183       expr=Expr DEDENT
184   | {UnaryMinusExpression} DASH expr=Expr
185   | {FunctionExpression} expr=FunctExpr)
186 ((=>infixOp=QInfixOpID | infixOp=':') infixOpExpr=Expr)?;
187
188 FunctExpr:      expressions+=BasicExpr+;
189
190 BasicExpr:
191   {VariableOrFunctionExpr}
192     funcOrVar=[FunctionOrVariable|QFunctionOrVariableID]
193   | {AnonymousFreeVariable} '_'
```

```
194   | {ConstructorExpr} constr=[DataConstructor|QDataConstrID]
195   | {EmbracedInfixOpExpr}
196     '(' (op=[FunctionOrVariable|QInfixOpID] | ':' | ',') ')'
197   | {LiteralExpression} Literal
198   | {ParenExpression}
199     '(' ((expr=Expr
200         ((',' tupleExprs+=Expr)+
201           | (op=[FunctionOrVariable|QInfixOpID]| ':')
202             rightExpr=Expr?)?)
203         | (op=[FunctionOrVariable|QInfixOpID]| ':')
204           rightExpr=Expr)? ')'
205   | {BracketExpression}
206     '[' (exprs+=Expr ('|' quals+=Qual (',' quals+=Qual)*
207         | (=>',' exprs+=Expr)?
208           (DOT_DOT Expr? | (',' exprs+=Expr)*)))? ']';
209
210 Alt:
211   {Alternative} pattenr=Pattern gdAlts=GdAlts? '->' expr=Expr
212     ('where' localDefinitions=LocalDefs)?;
213
214 GdAlts:
215   '|' leftExpr=Expr '->' rightExpr=Expr nextGdAlts=GdAlts?;
216
217 Qual:
218   {ExpressionStatement} expr=Expr
219     | {LetStatement}=>'let' localDefinitions=LocalDefs
220     | {AssignmentStatement}=>pattern=Pattern '<-' expr=Expr;
221
222 Stmt:       Expr
223           | =>'let' LocalDefs
224           | =>Pattern '<-' expr=Expr;
225
226 Literal:
227   STRING | CHAR | NATURAL | FLOAT;
228
229 ModuleID:           QUALIFIER* ID_UPPER;
230
231 QFunctionOrVariableID: QUALIFIER* (ID_LOWER | AS_KEYWORD);
232
233 QVariableID:        QUALIFIER* VariableID;
```

```
234 QFunctionName:       '(' InfixOpIDWithSpecialCases ')'
235                      | QFunctionID;
236 QFunctionID:         QUALIFIER* FunctionID;
237 QTypeConstrID:       QUALIFIER* TypeConstrID;
238 QDataConstrID:       QUALIFIER* DataConstrID;
239
240 TypeConstrID:        ID_UPPER;
241 DataConstrID:        ID_UPPER;
242 TypeVarID:           ID_LOWER| AS_KEYWORD;
243 FunctionID:          ID_LOWER| AS_KEYWORD;
244 VariableID:          ID_LOWER| AS_KEYWORD;
245
246
247 QInfixOpID:          QUALIFIER* InfixOpIDWithSpecialCases;
248
249 terminal CHAR:
250   "''"
251   |"'"!("'"|'\\')"'"
252   |"'"'\\'('b'|'t'|'n'|'f'|'r'|'u'|"\\"|'"'|"'")"'"
253   |"'"'\\'('0'..'9')('0'..'9')('0'..'9')"'"
254   |"'"'\\''x'('a'..'f'|'A'..'F'|'0'..'9')
255             ('a'..'f'|'A'..'F'|'0'..'9')"'";
256 terminal STRING:
257   '"'('\\'('b'|'t'|'n'|'f'|'r'|'u'|'"'|"'"|'\\')
258   |'\\'('0'..'9')('0'..'9')('0'..'9')
259   |'\\''x'('a'..'f'|'A'..'F'|'0'..'9')
260             ('a'..'f'|'A'..'F'|'0'..'9')
261   |!('\\'|'"'))*'"';
262 terminal DOT_DOT:     '..';
263 terminal DOT:         '.';
264 terminal AS_KEYWORD:  'as';
265 terminal DASH:        '-';
266 terminal QUALIFIER:   (ID_UPPER | ID_LOWER) DOT;
267 terminal FLOAT_SUFFIX:
268   DOT NATURAL ('e' DASH? NATURAL)?;
269
270 FLOAT:
271   NATURAL FLOAT_SUFFIX;
272
273 InfixOpIDWithSpecialCases:
```

```
274   INFIX_OP_ID
275   | INFIX_OP_HELPER
276   | DOT
277   | DASH;
278
279 terminal INFIX_OP_HELPER:
280   '~'|'!'|'@'|'#'|'$'|'%'|'^'|'&'|'*'|'+'|'='
281     |'<'|'>'|'?'|':'|'/'|'|'|'\\';
282 terminal INFIX_OP_ID:
283   DASH? (INFIX_OP_HELPER|DOT) (INFIX_OP_HELPER|DASH|DOT)*
284   | '`' (ID_LOWER|ID_UPPER) '`';
285 terminal ID_UPPER:
286   'A'..'Z' ('a'..'z'|'A'..'Z'|'_'|'0'..'9'|'\'')*;
287 terminal ID_LOWER:
288   ('a'..'z') ('a'..'z'|'A'..'Z'|'_'|'0'..'9'|'\'')*;
289 terminal NATURAL: ('0'..'9')+;
290 terminal CURRY_DOC_COMMENT:
291   '---' !('\r' | '\n')*;
292 terminal COMMENTS:
293   '--' !('\r' | '\n')*
294   | '{-'->'-}';
295 terminal WS:              ' ';
296 terminal INDENT:          '{';
297 terminal DEDENT:          '}';
298 terminal END_OF_LINE:     ';';
299 terminal INLINE_TAB:      '\t';
300 terminal NL:              ('\r'|'\n')+;
301 terminal ANY_OTHER:       . ;
```

B Installation Guide

B.1 Prerequisite

The Curry IDE is an Eclipse plug-in, that is why an appropriate Eclipse installation is necessary to run it. The Curry IDE is based on Eclipse Juno (4.2). There are various packages available for downloading on the official website[1], it is recommended to use the *Eclipse IDE for Java Developers*.

Moreover, the Curry IDE has to be accessible through an *Update Site* or JAR-archive and an internet connection is required to download additional required Eclipse plug-ins during the installation process.

[1] `http://www.eclipse.org/downloads/`, last visited December 3, 2012

B.2 Installation

B.2.1 Step 1

Start Eclipse and select “Help” → “Install New Software..” from the menu as illustrated in Figure B.1.

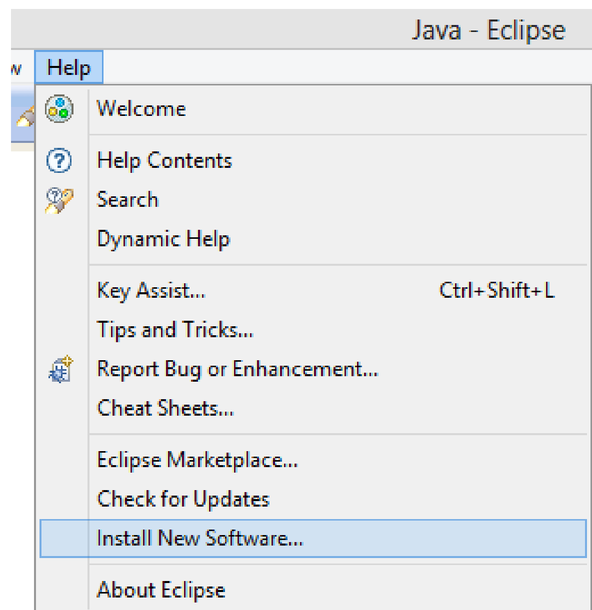


Figure B.1: Step 1: Select “Help” → “Install New Software..” from the menu.

B.2.2 Step 2

Next, the source of the Curry IDE has to be specified. Figure B.2 shows how this can be done. There are various ways to make Eclipse plug-ins accessible. We are using a local Update Site to demonstrate one of these possibilities.

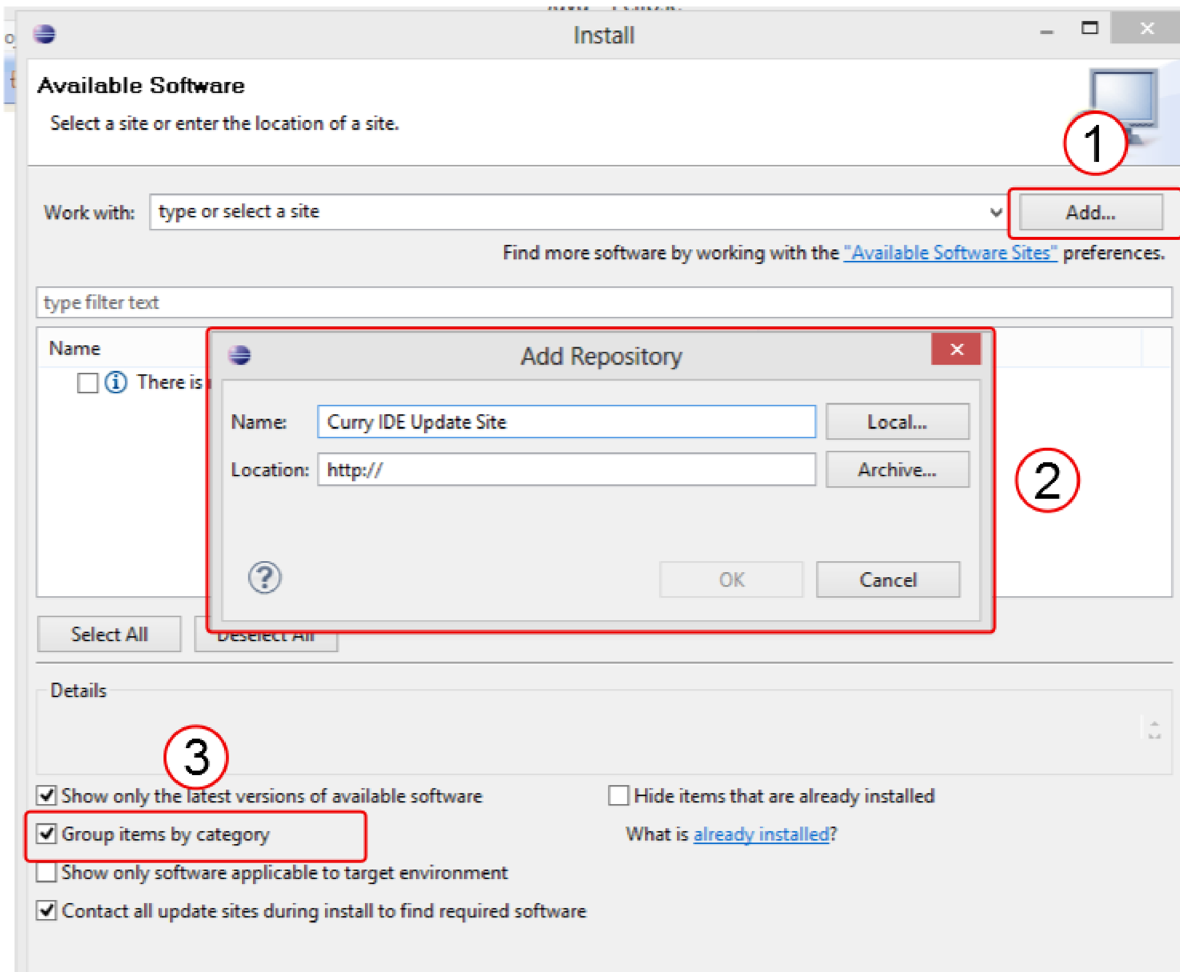


Figure B.2: Step 2: Add the source of the Curry IDE.

- Click the button “Add. . . ” (marked as (1) in Figure B.2).
- In the popup “Add Repository” (marked as (2) in Figure B.2) the source has to be chosen. Click the button “Local. . . ”, if you want to use a local Update Site, and select its root directory. Use any *name* to identify this repository and confirm using the “OK” button.
- The category grouping mechanism does not work for the Curry Update Site, hence, it is necessary to *uncheck* the option “Group items by category” (marked as (3) in Figure B.2).

B.2.3 Step 3

Select the repository that has been added in Step 2 from the drop-down list labelled “Work with:“ (marked as (1) in Figure B.3). The “Curry SDK Feature” should be visible (marked as (2) in Figure B.3), check the corresponding check-box and click “Next” (at the bottom of the dialog). Eclipse starts to calculate dependencies and requirements of the Curry IDE, there should be no conflicts. Start the installation process by pressing the button “Finish” (at the bottom of the dialog). During the installation process some additional plug-ins that are required to run the Curry IDE are downloaded and installed as well.

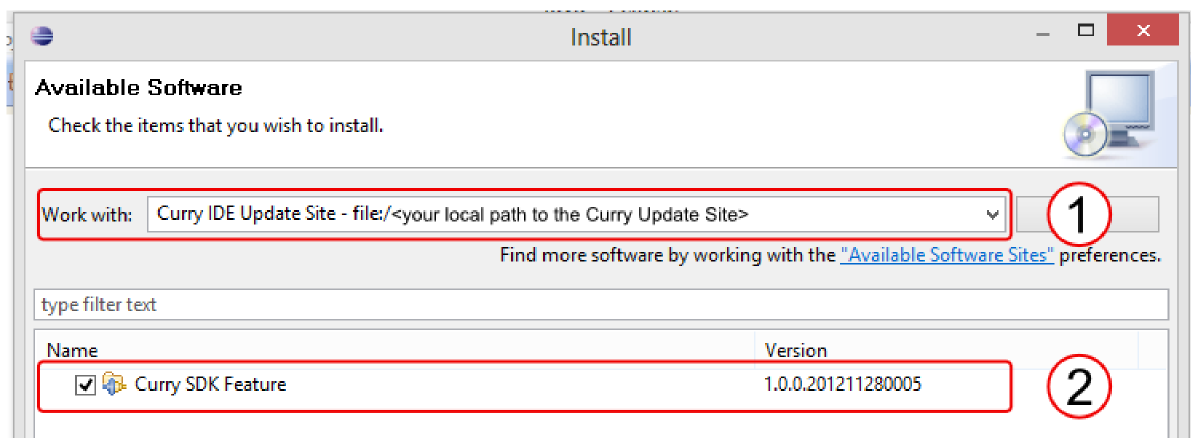


Figure B.3: Step 3: Select the repository and the *Curry SDK Feature*.

During the installation process a security warning is displayed, because the plug-ins of the Curry IDE are not signed. Confirm the warning using the “OK” button to continue the installation process.

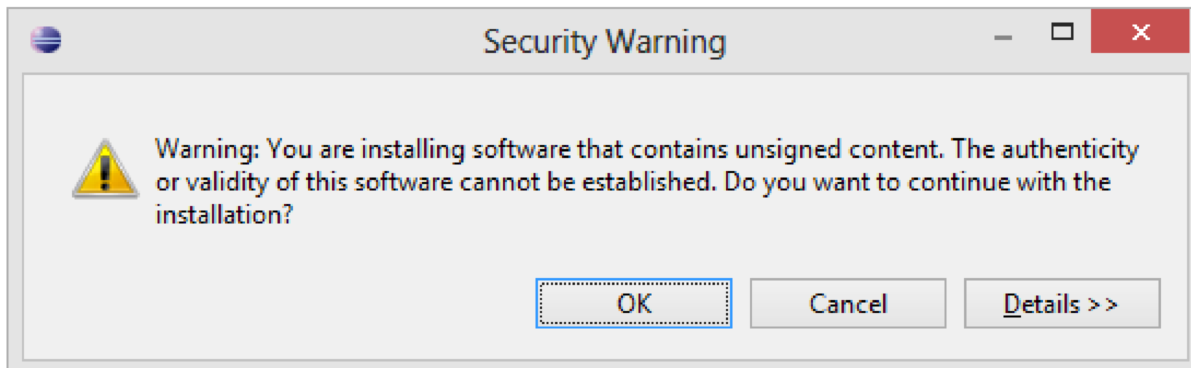


Figure B.4: During the installation process a security warning is displayed.

To finish the installation, Eclipse has to be restarted. Eclipse should automatically ask you for a restart. Afterwards, the Curry IDE should be ready to use.

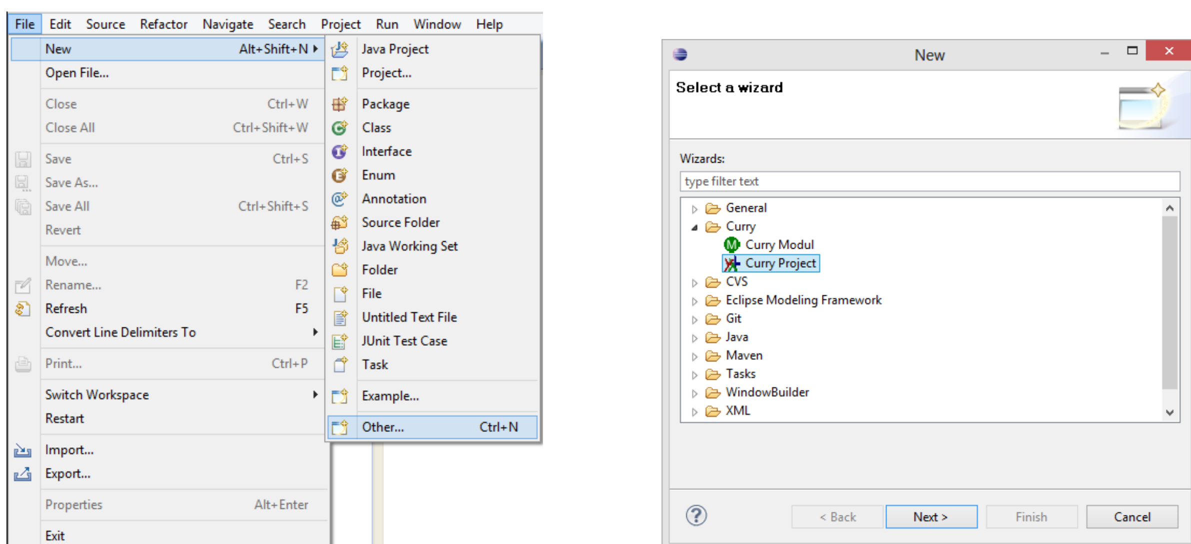


Figure B.5: Start the wizard to create a new Curry project.

You can verify that the installation has succeeded by creating a new Curry project. Select “File” → “New” → “Other…” from the menu. In the dialog that is displayed, a category name “Curry” should be visible containing the wizards to create a Curry project and module (see Figure B.5). After a Curry project has been created, Eclipse automatically asks to switch to the *Curry perspective*.

B.3 Update

If an Update Site is available for the Curry IDE, it can be updated using Eclipse update mechanism. Select "Help"→ "Check for Updates" to check if a new version of the Curry IDE is available. Follow the instructions of the update wizard to start the update process.

C Project structure

C.1 Overview

C.1.1 de.kiel.uni.informatik.ps.curry.CurryIDE

This project contains the components for language recognition. The most important packages are the following:

de.kiel.uni.informatik.ps.curry

This package contains the Xtext grammar, the generation workflow, and the configuration for dependency-injection.

de.kiel.uni.informatik.ps.curry.description

The *CurryDescriptionManger* can be found in this package.

de.kiel.uni.informatik.ps.curry.description.strategies

This package contains all search strategies that are used to determine the elements of particular scopes.

de.kiel.uni.informatik.ps.curry.linking

The custom linking service for Curry which contains the im- and export logic.

de.kiel.uni.informatik.ps.curry.naming

Contains a custom qualified name provider for Curry.

de.kiel.uni.informatik.ps.curry.parser

This package contains the manipulated token source to support Curry's layout and the modified Curry parser that uses it instead of the token source that is generated by Xtext.

de.kiel.uni.informatik.ps.curry.scopes

Contains the data model for Curry scopes.

de.kiel.uni.informatik.ps.curry.scoping

This package contains the local and global scope providers that can be used to determine the scope of a particular Curry element.

de.kiel.uni.informatik.ps.curry.utils

The *CurryImportHelper* can be found in this package.

de.kiel.uni.informatik.ps.curry.validation

The validator containing additional semantic validation.

C.1.2 de.kiel.uni.informatik.ps.curry.CurryIDE.ui

This is the UI project containing all generated and custom UI components of the Curry IDE, it depends on the project *de.kiel.uni.informatik.ps.curry.CurryIDE*. The *plugin.xml* can be found in the project's root folder, it contains a lot of configuration for the Curry IDE.

de.kiel.uni.informatik.ps.curry.CurryIDE.ui

Contains the configuration for dependency-injection.

de.kiel.uni.informatik.ps.curry.CurryIDE.ui.container

The *CurryContainerManager* manages the set of visible resource for Curry modules.

de.kiel.uni.informatik.ps.curry.CurryIDE.ui.contentassist

Contains the implementation of custom content assist.

de.kiel.uni.informatik.ps.curry.CurryIDE.ui.curryanalysis

This package contains the implementation of the client for the Curry analysis tool as well as the additional abstraction layer called *CurryAnalysisProvider*.

de.kiel.uni.informatik.ps.curry.CurryIDE.ui.helper

The *CurryUiHelper* can be found in this package, it provides some functionality to handle properties of Curry projects.

de.kiel.uni.informatik.ps.curry.CurryIDE.ui.views

Contains the custom views for the Curry IDE including the project explorer and the analysis view.

de.kiel.uni.informatik.ps.curry.CurryIDE.ui.views

The implementation of the wizards to create new Curry projects and modules can be found in this package.

C.1.3 de.kiel.uni.informatik.ps.curry.CurryIDE.analysis

The *Curry Analysis SDK* contains the interface for Curry analysis visualizations and provides the extension point to contribute visualizations to the Curry IDE.

C.1.4 de.kiel.uni.informatik.ps.curry.CurryIDE.sdk

This project is an Eclipse *feature project* and defines the plug-ins that should be part of the Curry IDE package.

C.1.5 CurryUpdateSite

The *Curry Update Site* allows to make the Curry IDE accessible for installation and updating. The “site.xml” contains information about versions that are available and can be used to build new versions.

Bibliography

[AHLT05] Sergio Antoy, Michael Hanus, Jimeng Liu, and Andrew Tolmach. A virtual machine for functional logic computations. pages 108–125, 2005.

[Ant07] Sergio Antoy. Curry: A tutorial introduction, December 2007.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[BCHH04] Bernd Braßel, Olaf Chitil, Michael Hanus, and Frank Huch. Observing functional logic computations. In *Sixth International Symposium on Practical Aspects of Declarative Languages*, pages 193–208, 2004.

[CF08] Jan Christiansen and Sebastian Fischer. Easycheck – test data for free. In *FLOPS '08: Proceedings of the 9th International Symposium on Functional and Logic Programming.* Springer LNCS 4989, 2008.

[FK02] Bryan Ford and M. Frans Kaashoek. Packrat parsing: a practical linear-time algorithm with backtracking, 2002.

[HAB+12] Michael Hanus, Sergio Antoy, Bernd Braßel, Martin Engelke, Klaus Höppner, Johannes Koj, Philipp Niederau, Ramin Sadre, and Frank Steiner. *PAKCS: The Portland Aachen Kiel Curry System*, 2012.

[Han02] Michael Hanus. Currydoc: A documentation tool for declarative programs. In *11th International Workshop on Functional and (Constraint) Logic Programming*, pages 225–228, 2002.

[Han06] M. Hanus. Currybrowser: A generic analysis environment for curry programs. In *Proc. of the 16th Workshop on Logic-based Methods in Programming Environments (WLPE'06)*, pages 61–74, 2006.

[Han12] Michael Hanus. Curry - an integrated and functional logic and language. In *Curry - An Integrated and Functional Logic and Language*, 2012. Version 0.8.3.

[HK01] M. Hanus and J. Koj. An integrated development environment for declarative multi-paradigm programming. In *Proc. of the International Workshop on Logic Programming Environments (WLPE'01)*, pages 1–14, Paphos (Cyprus), 2001. Also available from the Computing Research Repository (CoRR) at http://arXiv.org/abs/cs.PL/0111039 (last visited December 3, 2012).

[Mar] Simon Marlow. Haskell 2010 language report.

[MLA10] Jeff McAffer, Jean-Michel Lemieux, and Chris Aniszczyk. *Eclipse Rich Client Platform*. Addison-Wesley Professional, 2nd edition, 2010.

[PF11] Terence Parr and Kathleen Fisher. Ll(*): The foundation of the antlr parser generator. *SIGPPLAN Not.*, 47(6):425–436, June 2011.

[PQ94] Terence J. Parr and Russell W. Quong. Adding semantic and syntactic predicates to ll(k): pred-ll(k). In *In Computational Complexity*, pages 263–277. Springer-Verlag, 1994.

[xte] Xtext 2.3 documentation. http://www.eclipse.org/Xtext/documentation/ (last visited December 3, 2012).

[You67] D. Younger. Recognition and parsing of context-free languages in time n^3. *Information and Control*, pages 189–208, February 1967.

Declaration

I hereby declare that I have completed the present thesis independently, making use only of the specified literature and aids. Sentences or parts of sentences quoted literally are marked as quotations; identification of other references with regard to the statement and scope of the work is quoted. The thesis in this form or in any other form has not been submitted to an examination body and has not been published.

Kiel, December 4, 2012

Marian Palkus