Christian-Albrechts-University of Kiel

An API Search Engine for Curry

submitted by
Sandra Dylus

Bachelor Thesis
Programming Languages and Compiler Construction
Prof. Dr. Michael Hanus
Department of Computer Science
Christian-Albrechts-University of Kiel

Advised by Prof. Dr. Michael Hanus
M. Sc. Björn Peemöller

Erklärung der Urheberschaft

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit ohne Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oderähnlicher For in keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Ort, Datum Unterschrift

Contents

List of Figures

1

Introduction

This thesis examines the development of an API search engine for the functional logic programming language Curry [4] and presents the results of this development. Currently, online documentations are the only way to get an overview of the functions and data types that Curry provides. More precisely, PAKCS[1] and KiCS2[2], two implementations of Curry, offer such online documentations for the modules that are part of the distribution. Furthermore, these documentations are generated by a tool named *CurryDoc* that is also part of PAKCS as well as KiCS2. Unfortunately, the generated documentation is presented as a list of modules; under these circumstances, the search for a specific function is complicated if one has to scan every module, albeit one knows where to start searching.

The functional programming language Haskell [7] offers a similar documentation system. The hackageDB[3] is a collection of released Haskell packages; such packages mostly consist of several modules. In the course of time, the amount of released and uploaded packages has increased. Figure 1.1 shows this increase in the time between 2006 and the beginning of 2012. Currently, the database consists of more than 4,000 packages and 40,000 functions. In 2004, Neil Mitchell started to work on Hoogle[4], a search engine for Haskell packages, which was written in Haskell. Hoogle browses the documentation of Haskell modules available on hackageDB. These documentations are generated by Had-

[1] `http://www.informatik.uni-kiel.de/~pakcs/`
[2] `http://www-ps.informatik.uni-kiel.de/kics2/`
[3] `http://hackage.haskell.org/packages/hackage.html`
[4] `http://haskell.org/hoogle`

dock[5] and the provided modules are part of the Glasgow Haskell Compiler[6] (GHC), an open source compiler and interactive environment for Haskell. Unfortunately, Hoogle browses *only* these modules, but as we have learned, there are about 4000 of these packages that consist of even more modules.

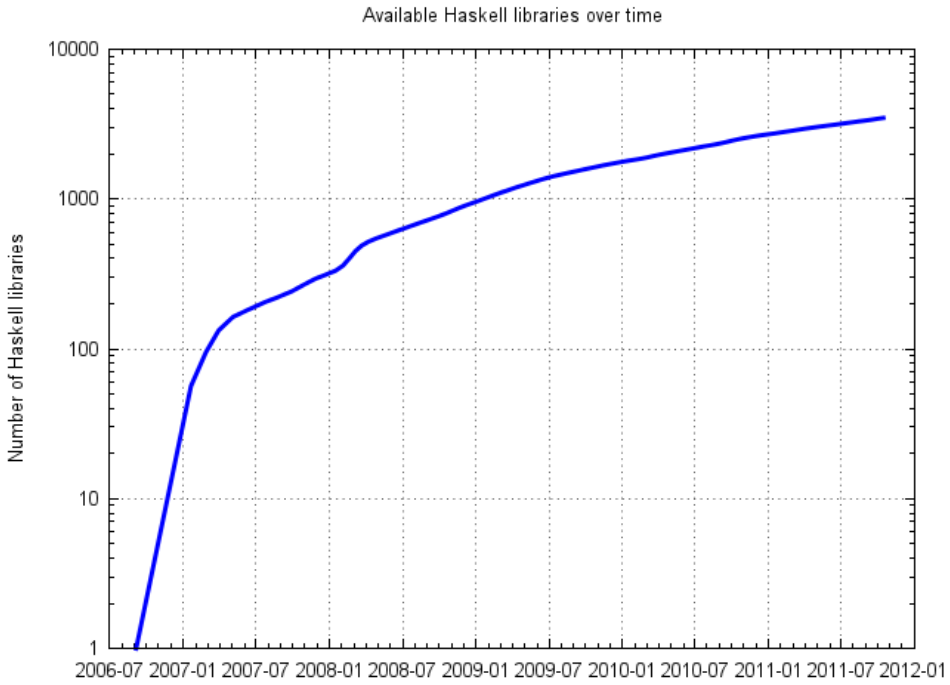


Figure 1.1.: Increasing Data Stock of hackageDB over the Past Five Years

In 2008, the FH Wedel decided to build *Hayoo!*[7], a new search engine written in Haskell that searches information about all available packages and corresponding modules of hackageDB. We think that such a search engine simplifies the work with a language like Haskell because it allows us to get a quick overview of functions that already exist and use them in our implementations.

More precisely, the FH Wedel implemented a framework to build highly-flexible search engines and *Hayoo!* was the first search engine application that was build with this framework. Thus, this is the starting point of this thesis: inspired by Hayoo! and Hoogle,

[5] `http://www.haskell.org/haddock/`
[6] `http://haskell.org/ghc`
[7] `http://holumbus.fh-wedel.de/hayoo/hayoo.html`

we decided that Curry needs its own search engine. The search engine can improve the work with Curry and forms a good addition to the existing documentation. Figure 3.1 shows Curr(y)gle - the web application of the search engine that we implemented.

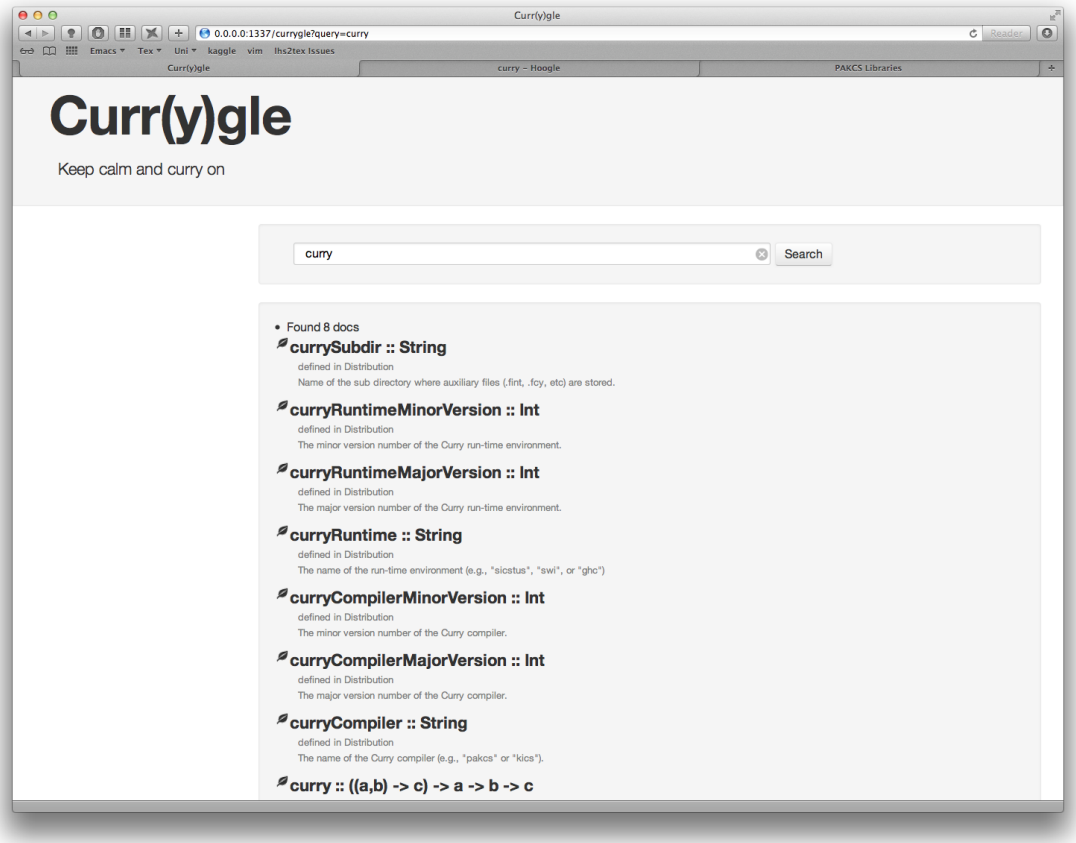


Figure 1.2.: Curr(y)gle - An API Search Engine for Curry

The further chapters of this thesis are organized as follows. At first, Chapter 2 presents the preliminiaries of this thesis. This includes basic information about the programming language Curry, the introduction of CurryDoc – a tool to generate a HTML-documentation of a given Curry source code that is quite similar to javadoc[8]. Further, we introduce the Holumbus framework that is used to build the search engine. In Chap-

[8] `http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html`

ter 3, we analyse the requirements to create an API search engine for Curry. The chapter outlines the first ideas for the following implementation. Chapter 4 focusses on the most important implementation ideas and decisions. At the very end, we discuss the results of this development. Also, we give a short outlook on features and ideas to expand the given result.

2

Preliminaries

This chapter gives a brief introduction to fundamentals that are necessary to comprehend the following chapters. The first section of this chapter gives an introduction to the programming language Curry. It outlines main concepts and features of the language. Furthermore, we present CurryDoc [3], a tool to generate documentation that is distributed with PAKCS, in the second section. The last section introduces the Holumbus[1] framework, a library written in Haskell, to configure and build search engines.

2.1 The programming language Curry

Curry is a functional logic programming language created by an international development team to provide a platform for research and teaching mostly. As the description suggests, it offers features of both programming paradigms: functional and logic.

As in Haskell, a Curry program consists of function definitions and data structures. A module *Test* is a program that is saved in a file named *Test.curry*. The syntax of a Curry program is quite similar to the syntax of Haskell.

$$addTwo\ x = x + 2$$

The left-hand side of this function $addTwo\ x$ is evaluated to the right-hand side $x + 2$, i.e., the call $addTwo\ 3$ yields $3 + 2 = 5$.

In general, an expression is evaluated by replacing the left-hand side of a definition by the right-hand side. The evaluation proceeds one replacement after another until

[1] `http://holumbus.fh-wedel.de/trac`

it yields a value. A value is an expression that does not contain function calls but only literals or data structures. If the last replacement does not result in a value, the evaluation fails. Moreover, it is also possible that the amount of replacement steps are infinite and the evaluation never succeeds nor fails. If an evaluation has more than one possible replacement step, so-called subexpression can be evaluated. Curry uses *lazy evaluation*, this means that such a subexpression is only evaluated, if its result is necessary to continue the evaluation and every expression is evaluated just once. A functional programming language can offer lazy evaluation because of referential transparency, this means that the value of an expression only depends on the values of its subexpression and does not depend on the time of the evaluation.

Furthermore, Curry supports function definition with pattern matching, which is often used in functional and logic programming languages. Pattern matching is a concept that allows variables and data constructors, like *True* and *False*, to occur in the arguments on the left-hand side of a definition in order to use them on the right-hand side. The boolean operation *not* is a good example for a definition with pattern matching. The definition distinguishes between more than one input value, so we have to write one rule for each possible input value.

$$not :: Bool \to Bool$$
$$not\ False = True$$
$$not\ True = False$$

The first line is the type signature and says that the function expects a boolean value as its argument and yields a boolean value as its result. The last two lines of the definition are rules, that describe that the application *not False* yields *True*, whereas *not True* yields *False*. There are no more possible values for the argument of the function *not*, since Curry is a strongly-typed language and *True* and *False* are the only possible values of boolean type. In addition, Curry also allows polymorphic functions. For instance, the identity function returns the argument that it is applied to, regardless of the input value's type. That is, you can apply the function to all types of values, because the type is not considered in the implementation. Hence, polymorphic types are indicated by type variables. The following code presents the type signature and definition of the identity function.

$$id :: a \to a$$
$$id\ value = value$$

Furthermore, functions are first-class citizens in Curry. This means that they can appear as argument or a result of an expression as well as in a data structure. The most popular use-case is the manipulation of all elements of a list by a given function. An exemplar of a higher-order-function is the function *map*, which also exists in Haskell, that takes two arguments, a function and a list and returns a list. It is important that the type of the function's argument matches the types of the elements of the list. For example, a function that converts an integer to a character can be applied to a list of integers and yields a list of characters. The following code presents the definition of the function *map*.

$$map :: (a \to b) \to [a] \to [b]$$
$$map\ func\ [\,] = [\,]$$
$$map\ func\ (elem : list) = func\ elem : map\ func\ list$$

The first line presents the type signature. The function $f :: (a \to b)$ takes a value of polymorphic type and returns this polymorphic type. The second argument is a list of elements with the type the function *func* expects. Furthermore, the resulting list of *map* contains elements of the same type as the resulting type of the function *func*. The definition of *map* says that an empty list yields an empty list. In case of a list that contains at least one element, *func* is applied to the each element of the list recursively.

Besides function definitions, a Curry program consists of data types definitions. PAKCS provides predefined data types, like the boolean type that we mentioned above.

$$\textbf{data}\ Bool = True \mid False$$

This code defines a data type with the name *Bool* that has two nullary constructors *False* and *True* with type *Bool*. Another interesting predefined data type is the polymorphic list.

$$\textbf{data}\ [a] = [\,]\ a \mid a : [a]$$

The syntax for lists is the same as in Haskell. $[\,]$ is the empty list, whereas $1 : 2 : 3 : [\,]$ is the same as $[1, 2, 3]$ or $1 : [2, 3]$. The latter representation is often used in pattern matching, to use the first element of the list in the right-hand side of the definition.

In addition to the already mentioned functional characteristics, Curry also offers logical variables and non-deterministic functions. Logic programming languages consist

of rules, for example, we can define a constant function that represents my favourite number,

$$favouriteNumber = 7$$

or in case of two favourite numbers, we define:

$$favouriteNumber = 3$$
$$favouriteNumber = 7$$

This function is non-deterministic because it returns different values for the same input. The pattern of this function overlaps in functional programming languages and, therefore, only the first definition will be used. Fortunately, Curry's ability to search for results allows to define those non-deterministic functions.

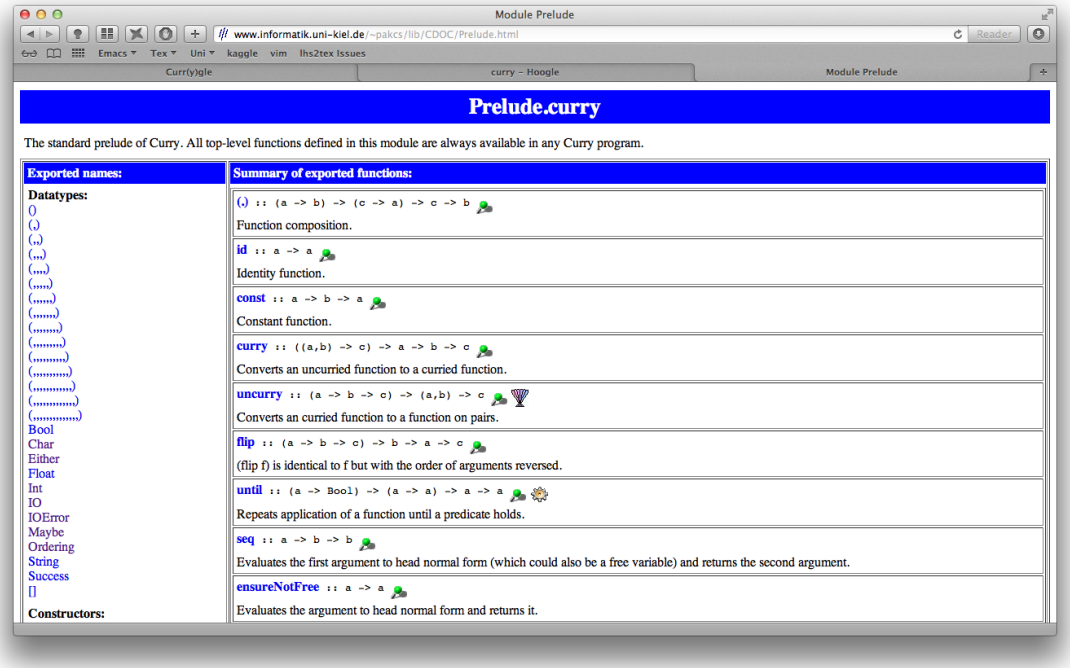
Curry also offers logic variables. A variable is called logic if it appears on the right-hand side but not on the left-hand side of a rule. These variables are unbound values, that are instantiated if the evaluation of an expression needs to know the value of the logical variable in order to proceed. Furthermore, these variables can be bound to different values because Curry computes all possible solutions of an expression.

Curry provides two different approaches to evaluate an expression with logical variables. The first approach suspends the evaluation in the hope that the logical variable will be bound by a concurrent evaluation of an expression. If there is no other expression to bind the value, the evaluation fails. This approach is called residuation and Curry uses it for boolean operators, like the equality operator $\equiv$. The second approach, called narrowing, guesses a value for an unbound value. Constraint operators, like the boolean constraint operator $=:=$, use narrowing for evaluation. In this context, Curry distinguishes two types of operators: flexible operators that use narrowing and rigid operators that use residuation. For example, arithmetic (i.e. $+$, $-$, $*$ etc) and other primitive operations are rigid. However, these distinctions do not have any significance for expressions without logical variables, so called *ground expressions*. As mentioned in the previous section, Curry evaluates ground expressions with lazy strategy.

In summary, Curry has a strong type system and allows polymorphism. A Curry program is a collection of function definition and data types. Furthermore, Curry provides higher-order-functions, algebraic data structures and pattern matching. Logic variables and non-determinism are logic features that Curry supports.

2.2 CurryDoc

CurryDoc is a tool to generate documentation for a program written in Curry. The current version can generate either a HTML or LaTeX file as output. CurryDoc works similar to code generating tools like javadoc, as it uses the comments in the source code, which are provided by the programmer. It also provides the type signatures of functions; if the signature is not given in the source code, Curry can use its type inference algorithm to provide the information. In addition, the CurryDoc tool analyses the program's structure and approximates the run-time behavior to gain further information [3]. This analysis includes information about (in-)completeness, overlapping pattern matches and (non-)deterministic functions. The following screenshot shows the online documentation of the Curry module *Prelude* that is generated by CurryDoc.



CurryDoc is implemented in Curry and it uses the meta-programming module *FlatCurry*[2] that provides an intermediate language representation of the Curry program to analyse the special function properties. A FlatCurry program consists mainly of a list of functions, a list of types and information about the module itself.

[2] `http://www.informatik.uni-kiel.de/~pakcs/lib/CDOC/FlatCurry.html`

2.3 The Holumbus Framework

The Holumbus Framework is a Haskell library created by students of FH Wedel in context of two master's theses [5][8]. Holumbus is a framework to build and configure customizable search engines. The main idea of the framework is to collect data with a specific structure, like a cooking recipe or an API of a programming language, and to take advantage of this structure in order to improve the search results. In addition to the framework, Hayoo! [1] was developed, an API search engine for the functional programming language Haskell.

The framework supports three phases to create a working search engine: the crawling (1), the indexing (2) and the searching phase (3).

1. In the process of crawling, the framework can automatically browse web pages of given URIs in order to collect all data and information, respectively. The retrieved pages are then passed to the indexer in order to preprocess the data.

2. In the indexing process the pages are analysed and stored in a data structure to provide a fast information retrieval. The indexer preprocesses the data in order to filter all relevant search criteria, e.g., an indexer could filter only PDF documents. Additionally, instead of keeping a copy of the original document, e.g., a web page, a characteristical data structure is created.

3. Finally, the fast information retrieval is the heart of the search engine. One can access the information of the index by phrasing a search query. Commonly, the search query is compared to the information stored in the index and convenient matches, i.e., documents, are returned as results.

For the implementation of Curr(y)gle, we do not use the crawler part of Holumbus because we do not want to browse and parse web pages. We rather want to make use of the existing CurryDoc and the information of Curry modules that the generated documentation provides. Therefore, we focus on the latter two features of the Holumbus framework. Holumbus provides several modules for the to construct and access the index; the directory structure of the framework is illustrated in Figure 2.1. The module *Common* provides implementations for the data structure to represent the searchable information. This data structure is composed of two parts: search keys and documents.

In Holumbus, the structure to hold the preprocessed data is called *index* and *documents* is the name for the data structure that holds the crawled information. Further data structures that are used in the indexing process are stored in the folder *Common*, whereas *Inverted* consists of the implementation for the data structure of the index. Holumbus uses a structure called *inverted filed* or *inverted index*. This structure stores the occurrences of each search criterion and a reference to its corresponding document. Furthermore, Holumbus provides a mechanism to form and process a search request, and a data structure to represent the result; additionally, the result can be subjected to a ranking.

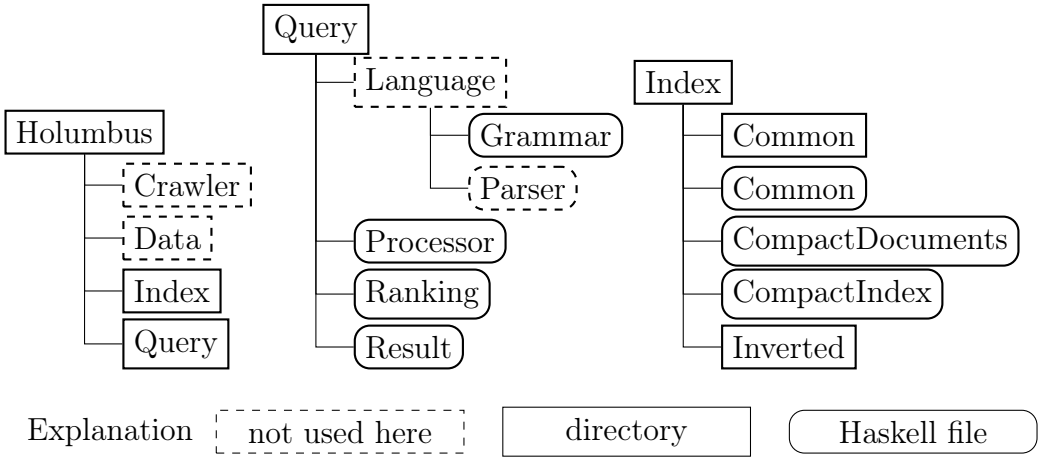


Figure 2.1.: Directory Structure of the Holumbus Framework

3

Analysis

This chapter looks into the requirements to build and run Curr(y)gle, an API search engine for Curry. The first section deals with the creation of an index that manages the information that we want to provide, whereas the second and third section address the process of searching for a query. In this context, we take a closer look at the Holumbus framework and its features related to searching and evaluate the criteria to accomplish a user-friendly search mechanism. Furthermore, we discuss the syntax Curr(y)gle should provide to specify a search query.

Before we present the first ideas in more detail, we want to give an overview of the structure of Curr(y)gle. The Figure 3.1 illustrates all relevant components that work together in order to provide the search engine.

The first components, including the parser and the Holumbus framework, describe the functionality of the web application, i.e., the searching phase. At first, the parser analyses a given search term, like `"map"`, and constructs a query data structure for the Holumbus framework. Holumbus searches for the given search term in the provided index and, in case of matching entries in the index, returns the documents that contain the search term. At last, we prepare the results, in form of the documents, for the web application.

We highlighted the second process of Curr(y)gle with dashed arrows. This process describes the construction of the index that we pass to Holumbus in order to provide the data that can be searched for. The main idea is to use the information generated by CurryDoc. We prepare this data in order to fit the required data structure that

Holumbus provides. More precisely, we split the data in to search keys and documents and pass this data to Holumbus as source for the index.

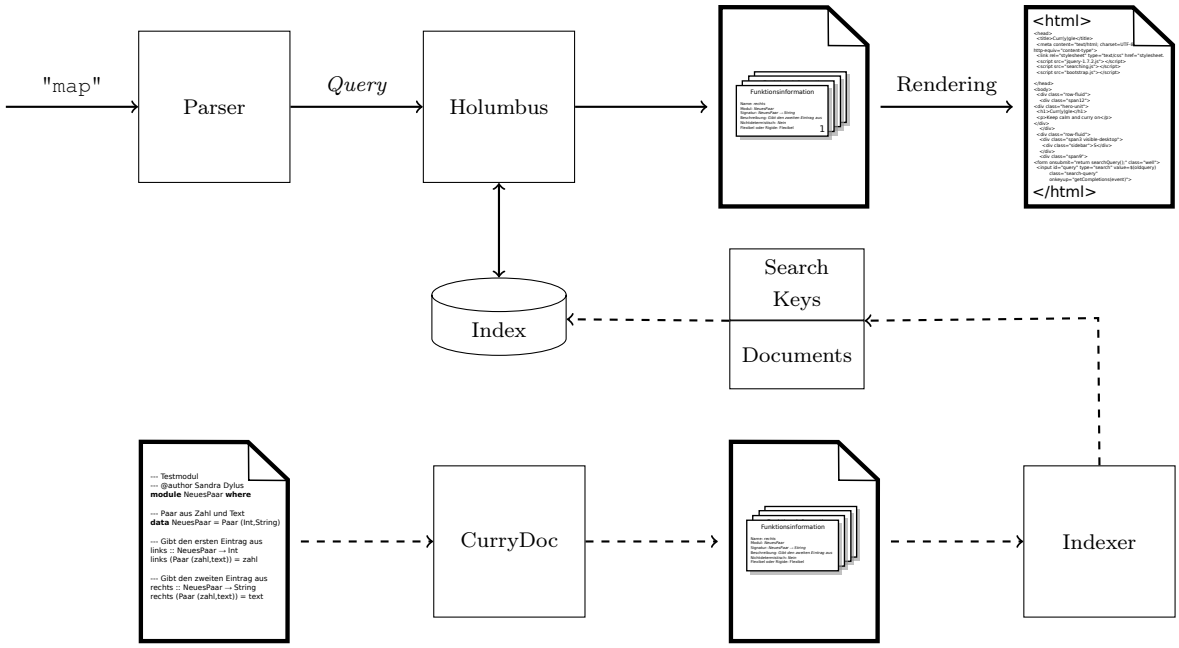


Figure 3.1.: Structure of Curr(y)gle

In the following section, we elaborate on the index construction, including the function of Holumbus and the usage of CurryDoc as source of the information in the index. Furthermore, we outline the main functions of the parser that we implemented in order to analyse the user queries.

3.1 Extracted Information

Search engines look up information, hence, we need to collect data that we can search for. For Curr(y)gle, we want to collect data that provides information about the API of Curry. Therefore, we first need to think what kind of information we can provide and decide what we want to search for. Secondly, we introduce the idea behind the construction of the index and the managed data that is traversed in the process of searching.

Usually, a web crawler is applied to browse web pages for data. But since Curry is currently organized by the module documentation generated by CurryDoc, we already have a mechanism to gain the information about a Curry module that we want to provide for the index. In fact, we have even more specific function-related information because we know if a function definition is non-deterministic or deterministic and if a given function is flexible or rigid. In the following, we refer to these characteristics as context and the following table presents the information or, more precisely, the contexts that we want to provide:

	Module	Function	Type
Name	✓	✓	✓
Author	✓		
Type Signature		✓	✓
Description (in comments)	✓	✓	✓
Defining Module		✓	✓
Rigid/Flexible characteristics		✓	
(Non-)-deterministic Definition		✓	

In general, we want to distinguish between three kinds of information in a Curry module: the defined data types and functions, and the module itself. We want to provide the name and description for all three kinds; for the module we also store the author's name. Additionally, we gain a lot of information for functions that we can add to the index: nondeterministic vs. deterministic definition, flexible vs. rigid definition, the corresponding type expression and the name of the module that defines the function. For types, we add the corresponding module and the type signature for its constructors.

Currently, CurryDoc processes Curry modules and generates documentation in form of HTML or LaTeX output. For our index, we do not want to use a document markup language but the pure information about the Curry module, otherwise we would have to parse the HTML-structures in order to filter the relevant information. This observation leads to the idea of adding a new data structure that is generated by CurryDoc. For this CurryDoc extension, we take advantage of the FlatCurry representation of a Curry module to access the information we mentioned above. We discuss the actual implementation of the CurryDoc extension in the next chapter.

We use the information, that we are generating with the CurryDoc extension, as source for the index that we want to provide for Holumbus. As requirement to utilize the framework, we have to use specific data structures. These structures arrange the index to be split into two components: the search keys that we want to provide, and the documents that correspond to the given search keys. Furthermore, a search keys consists of the actual word that we want to search for and a context to differ between several kinds of information. For example, if we have a Curry modul with information about the module's name and description, we provide a document structure and the corresponding search keys consisting of context and word:

ModuleInformation
Name: *Duck*
Description: *It's a duck*

context	search word
module	"Duck"
description	"It's a duck"

Thus, if we search for the word "Duck", Holumbus scans the given search words and, in case of a hit, returns the corresponding document structure.

In summary, we want to extend the current CurryDoc implementation to generate a new parseable data structure that contains information about a given Curry module. This information covers, among other things, function and data structure definitions, descriptions in the form of user comments, and meta information about the module.

3.2 Searching

After creating the data for the index, we want to actually use this information in a search query. Thus, the first step is to think about the structure of a query, in the second step we process the query, and lastly we need a representation of the results of the processed query for further use.

Fortunately, these are all features the Holumbus framework provides. At first, we take a look at the search mechanism and the query representation. The data structure *Query* allows to search for a word or a phrase, both case-insensitive and case-sensitive.

```
data Query = Word String
           | Phrase String
           | CaseWord String
           | CasePhrase String
           | ...
```

Since the search depends on user-input, the framework also allows *fuzzy searching* to scan for results with spelling errors like transposed letters. Since the index data structure of Holumbus uses pairs of words and contexts, a special mechanism to search for one or more contexts is given. Furthermore, the structure provides binary operators to combine multiple queries; supported operators are conjunctions, disjunctions and negations.

```
data Query   = ...
             | FuzzySearch String
             | Specifier [Context] Query
             | BinQuery BinOp Query Query
type Context = String
data BinOp   = And | Or | But
```

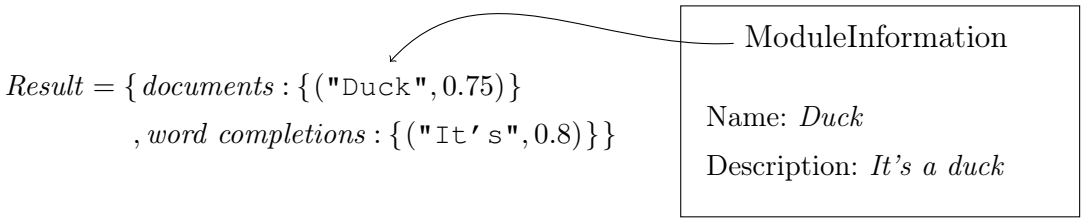
After the construction of the query data structure, we pass the index, document and query to Holumbus, by calling the function *processQuery* that, as the name suggests, processes the query; the function is provided by Holumbus. When processing the query, Holumbus only searches for prefixes of the given word or phrase in a query, we need to keep this restriction in mind when we create the index in Section 4.2.

The return value of the successfully processed query is a data structure *Result* that consists of the matching documents as well as possible word completions. Additionally, each document and word completion has a calculated score between 0 and 1 that determines the relevance of the result. For documents, this score is calculated by the number of occurrences of the search query in the document by default and represented as a float. However, Holumbus also provides a mechanism to apply a customized ranking function to calculate the score.

Let us proceed the examples of the previous section. We can construct the query

```
Specifier ["description"] "It"
```

and after processing, we get the following result:



All modules, i.e. documents, whose description contain the word "it" and all possible word completions (we have added random values as scores to the example for completeness) are the result of the search. The word `"Duck"` stands for the document that we already discussed in the previous section and the curly brackets { } symbolise the possibility of several documents and word completions as result. More precisely, the data structure *Result* can hold a list of documents and word completions.

Summing up, we have discussed the mechanism to evaluate a query with the Holumbus framework. The provided mechanism includes the data structures to represent a query, which can be processed to a data structure consisting of the matching documents and possible word completions.

3.3 Parsing User-Queries

The next question is how to construct the query for a given user-input. At first, we have to decide about the criteria users can search for. Since the index provides the pairs of contexts and search words, we can use these contexts to restrict the search results. More precisely, we are able to construct the *Query* data structure with the *Specifier* constructor to search in the given context, only. This structure of the index allows us to search for modules, functions, types, signatures, and all other contexts we use during the creation of the index.

The search mechanism (as part of the user-experience) is supposed to be as simple as possible. The right use of a specific language can increase the usability. A good example is the search term `"IO"`, since in Curry *IO* is the name for a module, a type and a constructor. Thus, the search for *IO* results in a great amount of hits. We can restrict the search to a specific context to reduce the number of hits. Therefore, we provide

specifiers to combine a context with a search term, for example `":function IO"` searches for *IO* in the context of function names only. But this special syntax restricts the user in the use of the search engine, if the language gets more complex. Thus, in order to provide a user-friendly search engine, we have to make a compromise between a human-readable language and a language that can be parsed.

Besides these specifiers, we want to parse type expressions of Curry functions and data types. During our test phase, we also studied Hayoo!, the first search engine that was build with the Holumbus Framework. We recognized that Hayoo! is not able to parse redundant parenthesized type expressions. In our opinion, this parsing behavior can be irritating in the usage of the search engine, therefore, we want to address the problem of redundant parentheses with great care when we parse signatures. For instance, let's assume a beginner searches for a function with the type expression `"IO -> (IO Int)"`. In this case, the type *IO Int* and unary type constructors in general do not need parentheses, but as beginner you might think that they do. Thus, we want to support parenthesized type expressions and parenthesized query parts in general.

Last but not least, we want to provide logical conjunctions like *AND*, *OR* and *NOT*. On the one hand, a combination of more search words is desirable because popular search engines, like Google, support binary operations as feature. The popularity increases the probability that users assume that binary operations are standard features and expect search engines to provide the conjunction of several search terms. On the other hand, if the desired result is still vague, a combination of more search words by a disjunction *OR* helps to narrow down the search results. Additionally, *AND* and *NOT* can help to search for a specific search term.

In the end, we want to provide an intuitive but powerful syntax for the search engine. With specifiers to restrict the search results to a given context and with binary operators to narrow down or extend the search, we want to provide a simple language for the user queries. Additionally, we want to recognize type expressions, including function, constructor and simple types as well as redundant parenthesized type expressions. Thus, in order to reach this goal, we need to analyse the user input and construct an expression of our *Query* data structure. Appendix C.1 describes the parser's language as *Extended Backus-Naur Form* (EBNF) and in Section 4.3 we discuss our actual implementation.

4

Implementation

This chapter presents the implementation of the search engine on the basis of some code examples and the corresponding design ideas and decisions. At fitst, we take a look at our extension for the current version of CurryDoc. In this context, we illustrate the interaction between this extension and the index creation. The latter will be depicted in detail in the second section. We specify the index and document data in more detail and state some difficulties that arose due to the structure of the Holumbus framework. The third section addresses the search process. We outline the function of the parser that converts the user input into a query that can be processed by Holumbus. We illustrate the general idea and implementation approach of a parser to introduce into the subject and we give an excerpt of the implemented parser. Furthermore, we explain the documental retrieval that Holumbus provides.

The last step to our search engine is the application itself. In order run the search engine as a web application, we use the *Snap Framework*[1]. Snap is a Haskell library to simplify web development. We do not go into detail on this topic, albeit, Appendix B shows how to run the application.

4.1 CurryDoc Extension

In the previous chapter, we discussed the general idea of an extension for CurryDoc to extract the information about a module into a data structure. Later in the process, this data structure serves as source for the index creation. In this section, we take a look at

[1] `http://snapframework.com/about`

the implementation of the extension of CurryDoc.

Since CurryDoc is written in Curry, we implemented our extension in Curry as well. With this decision we benefit from already implemented functionalities and, on the other hand, using the same programming language simplifies the integration of our implementation with the current CurryDoc version.

CurryDoc uses the meta-programming module FlatCurry to gain an intermediate data structure of a Curry module. We can use this data structure for our purposes. Additionally, we can reuse functions already provided by CurryDoc. For example, CurryDoc supports a special comment syntax to annotate the author and version of a module. Furthermore, the arguments and the return value of a function can be described as well as general descriptions.

At first, we discuss which information we want to provide in our data structures. The following data structure

data *CurryInfo* =
  CurryInfo ModuleInfo [*FunctionInfo*] [*TypeInfo*]

represents a Curry module, that consists of several function and data type definitions and meta-information about the module itself. As next step, we want to describe *ModuleInfo*, *FunctionInfo* and *TypeInfo*.

data *ModuleInfo* = *ModuleInfo*
  String -- name
  String -- author
  String -- description

Like the name suggests, *ModuleInfo* represents the data corresponding to a Curry module. The main information about a module consists of its name, author and description. We could also provide the version number of the implementation or the imported modules, but we decided against it. The latter seems to be useless information for the search engine, since the Curry modules are highly interrelated. Thus, searching for a module results in a great amount of hits, since every correlating module will be shown as well. Furthermore, we think the version number is not a significant characteristic for a module. Therefore, we decided to focus on the three mentioned properties only.

```
data FunctionInfo = FunctionInfo
  String          -- name
  TypeExpr        -- type signature
  String          -- corresponding module
  String          -- description
  Bool            -- True if the function is non-deterministic
  FlexRigidResult -- flexible/rigid characteristic, conflicted or unknown value
```

FunctionInfo contains the characteristics for a given function like the function's name and description. Additionally, we decide to add the corresponding module to provide a connection between the function and its module. This decision is based on the cause that we do not keep the *CurryInfo* data structure as whole for the index construction, but the three arguments consisting of the list of functions, the list of types and the module information. Thanks to FlatCurry, we can also access function characteristics like nondeterministic vs. deterministic, along with the information if a given function is rigid or flexible. The data structure for the latter is defined as

```
data FlexRigidResult = UnknownFR | ConflictFR | KnownFlex | KnownRigid
```

analogue to the definition in the *FlatCurry* module. Since these are important characteristics to distinguish Curry functions, *FunctionInfo* stores this information as a property. In addition, FlatCurry provides a data structure *TypeExpr* to describe type expressions of function definitions (see last chapter), hence, we can access the actual definition of a function.

We use a function's type signature as part of the *FunctionInfo* data structure, but decided against the usage of a function's definition since we could not think of a relevant use-case for our search engine.

The data structure for types looks quite similar to *FunctionInfo*.

```
data TypeInfo = TypeInfo
  String                  -- name
  [(QName, [TypeExpr])]   -- constructors and their signatures
  [Int]                   -- type variables
  String                  -- corresponding module
  String                  -- description
  Bool                    -- True for type synonyms, false for data types
```

TypeInfo consists of a type's name, description and corresponding module. Since the module FlatCurry provides type expressions for functions, we also get information about constructors for a given type. Therefore, we store a list of *TypeExpr* representing the type's constructors. Additionally, *TypeInfo* holds a list of integers to represent possible type variables. The decision to use integers corresponds to the definition of *TypeExpr*, where type variables are represented as integers as well.

In the end, we feed the *CurryInfo* data structure with the specific module, function and type information of a given Curry program, and our CurryDoc extension writes the data structure into a file. For example, if we want to index the *Prelude*, thus, we get a file named *Prelude.cdoc*. The final CurryDoc version allows two mechanisms to generate the *CurryInfo* structure. You can generate the *.cdoc*-file only or you use the HTML generation, where the *.cdoc*-file is also part of the output. We provide further instructions for the usage of CurryDoc in Appendix A.

Due to the similar syntax, we can use the same data structures in Haskell as in Curry to exchange those information about a Curry module. More precisely, we can parse the *.cdoc*-file within our Haskell implementation and work with the data structure. In order to do use this structure for the construction of the index, we need a Haskell program that defines all the data structures used in *CurryInfo* including the nested structures.

Appendix A.2.1 shows an exemplary Curry module and the resulting data structures that are generated by the CurryDoc extension.

4.2 CurryIndexer Implementation

This section illustrates how to create the index for the search engine. In our analysis, we only discussed about the source of our information that is provided by the CurryDoc extension. In the following, we present the data structures that Holumbus provides in order to create the index. Furthermore, we talk about the advantages and disadvantages of using the Holumbus framework and describe the implementation of the index creation in more detail.

In the previous section, we have already described the required preparation in order to create the index. On the Haskell side, we need to define the data structures that we

use to build the *CurryInfo* structure that are already defined in the implementation of the CurryDoc extension. Then, we can read the file, that is produced by the CurryDoc extension, for further usage. Next, we process these data to fit the data structures provided by Holumbus. In the end, we can either create a new index by writing each structure to a file to store our information or update an existing index with additional Curry modules. In order to update an existing index, we load the index and document files and merge them with new data. Due to lazy evaluation, we cannot read and write to the same file; it is not assured that we finish reading before we start to rewrite the file. Therefore, we have to write temporary files and rename these files afterwards to guarantee a clean outcome.

During the testing phase of the indexer, we noticed problems regarding duplicate data. In particular, when we add a Curry module to the index twice, there is no mechanism to detect the duplicated data. For this reason, we also manage a list of the modules, which are stored in the index. Thus, every time we update the index with a given module, we check if it already exists in the saved list of modules that are stored in the index. We start the processing of the data if the module does not occur in the list and add the module's name to the list.

4.2.1 Index Construction

After we decided about the contents of the index, we need to discuss the data structure to hold the information about a given Curry module. We make use of the Holumbus framework that provides data structures to manage the collected data and implementations to operate on these structures. Holumbus uses two structures to store the data, one for the documents we are indexing and the other one for the actual search keys that we traverse when a search is performed. At first, we want to discuss the search keys.

Simply put, the search keys consist of pairs of strings, where the second component contains the actual *search word* and the first component contains the context of this word. In order to provide an example, let us assume that we have a Curry module *Duck* and the following information about this module:

```
("name","Duck")
("author","Donald")
("description",
 "If it quacks like a duck & walks like a duck, it's a duck.")
```

As addition to the design, the identifier of the document is stored in the index data structure to provide an association between the search key and the document the data was indexed from.

In fact, this concept of referencing to the documents in order to separate the data is called *inverted index* or *inverted file*, which explains the name of the data type. Thus, in the end, the data we pass to Holumbus is a collection of search keys and these search keys consist of triples: the context, the actual search word and the reference to the corresponding document.

Inverted =

```
("name","Duck",1)
("author","Donald",1)
("description","If it quacks [...] it's a duck.",1)
```

Holumbus' data structure *Inverted* maps the words to their location, in our case, the document that we describe later. More precisely, *Inverted* maps the search words and contexts to their corresponding document. Additionally, Holumbus provides a data structure that represents the position of the word in the document. We can use these positions of words to reconstruct a phrase in a document. For example, if we have the description "It's a duck", the data structure represents this phrase with

```
("description","It's",(1,1))
("description","a",(1,2))
("description","duck",(1,3))
```

and with the information of the positions of the search word in the document, we can reconstruct the phrase by searching for the document with identifier 1 and the positions 1 til 3. However, we would only need these positions if we wanted to provide phrase queries. Since our main goal is to search for type expressions or function, module and type names, we decided against the support of phrase queries. Therefore, we do not use the position when indexing our data. Furthermore, the words are stored in prefix trees,

which only allow prefix search, like we mentioned before. Due to the prefix search, we came across some difficulties, which we discuss later.

When we index the *CurryInfo* structure, we examine its substructures *ModuleInfo*, *FunctionInfo* and *TypeInfo* to gain the characteristic information. In this process, the information is paired with a context and the contexts' names correspond to the information in these substructures. For instance, in *ModuleInfo* we keep the module's name, author and description. Table 4.1 illustrates the provided contexts and shows the corresponding property of the data structure *ModuleInfo*. We do not have much to do for

ModuleInfo	
property	context name
name	"module"
author	"author"
description	"description"

Figure 4.1.: Contexts for a *ModuleInfo* Data Structure

a module's name, but since a module can be written by several authors, we have to add a context for each author stored in *ModuleInfo*. Since we only have a representation as *String*, we need to process the string containing the author information.

In fact, the representation is not the main problem; the prefix search prohibits the usage of the whole string as a word for our index. If we search for an author named *"Duck"*, we will not find *"Donald Duck"* since the search word *"Duck"* is not a prefix of *"Donald Duck"*. Therefore, we have to spilt the string on whitespaces to gain all subparts of the string and add each part individually. A similar problem applies to the description, thus, we split the description on whitespaces, too. In addition, we filter words that are shorter than two characters to minimize redundant or, more precisely, unserviceable data in the index.

We need the same preparation for the name of a function and its description in *FunctionInfo* and the type's name and description in *TypeInfo*. In addition, both structures hold their corresponding module's name. In a first version, we used the same context for a function or type's defining module as for the name in *ModuleInfo*. However, we want to distinguish between a search for a module's name and a function or data structure in a specific module. Therefore, we decided to use two different contexts:

one for *ModuleInfo* that we already introduced and the second one in the context of a corresponding module for a function or data structure; for the latter we use *"inModule"*. Next, we take a look at type expressions, where we have the same problems due to the prefix search.

The FlatCurry module provides a data structure that represents type expressions used in function signatures and for constructors in data type definitions.

$$\begin{aligned}&\textbf{data}\ \mathit{TypeExpr} = \mathit{TVar}\ \mathit{TVarIndex}\\ &\qquad\qquad\quad |\ \mathit{FuncType}\ \mathit{TypeExpr}\ \mathit{TypeExpr}\\ &\qquad\qquad\quad |\ \mathit{TCons}\ \mathit{QName}\ [\mathit{TypeExpr}]\end{aligned}$$

It consists of three constructors to distinguish between

- a function type - $\mathit{FuncType}\ \mathit{TypeExpr}\ \mathit{TypeExpr}$
- a type variable - $\mathit{TVar}\ \mathit{TVarIndex}$

and

- a type constructor application - $\mathit{TCons}\ \mathit{QName}\ [\mathit{TypeExpr}]$

For the latter, the list of *TypeExpr* represens the type arguments of the defined constructors. Furthermore, *TVarIndex* is just a type synonym for *Int* and *QName* is a type synonym for a tuple $(\mathit{String}, \mathit{String})$; *QName* represents a qualified name consisting of the module's name and the function's or type constructor's name. A type like *Bool* is represented as a type constructor with an empty list, i.e. without any type arguments. The following expression

$$\begin{aligned}&\mathit{FuncType}\ (\mathit{TCons}\ (\mathit{Prelude}, \mathit{Bool})\ [\,])\\ &\qquad\qquad (\mathit{FuncType}\ (\mathit{TCons}\ (\mathit{Prelude}, \mathit{Int})\ [\,]))\\ &\qquad\qquad\qquad\qquad (\mathit{TCons}\ (\mathit{Prelude}, \mathit{Int})\ [\,]))\end{aligned}$$

is a function type and represents the type $\mathit{Bool} \to \mathit{Int} \to \mathit{Int}$. Next, we see the function type

$$\begin{aligned}&\mathit{FuncType}\ (\mathit{TCons}\ (\mathit{Prelude}, \mathit{IO})\ [\mathit{TVar}\ 97])\\ &\qquad\qquad (\mathit{TCons}\ (\mathit{Prelude}, \mathit{IO})\ [\mathit{TVar}\ 97])\end{aligned}$$

that consists of two unary type constructors with type variables; the corresponding type expression is $IO\ a \to IO\ a$. As last example, we show the structure

$$TCons\ (Prelude, Maybe)\ [(TCons\ (Prelude, IO)\ [(TCons\ (Prelude, [])\ [(TCons\ (Prelude, Char))])])]$$

that corresponds to the nested constructor type $Maybe\ (IO\ String)$.

In order to add the type expression to the index, we convert the *TypeExpr* we store in *FunctionInfo* and *TypeInfo* into a valid type expression that is consistent with the Curry syntax, more precisely, we pretty print the data structure. This conversion yields a *String* for further processing. Now let's assume we want to search for a function that yields the type *HTMLExp*. The first idea is to search for *":type HTMLExp"* and look for further information in the documentation. The problem is, we do not want the user to look for information in the documentation, we rather want to provide a mechanism to cover this scenario. Since we provide type expressions, the user can search for *HTMLExp* in the context of signatures to get information about all the functions (and data structures) that contain the type *HTMLExp*.

Unfortunately, in order to provide this behaviour, we have to modify the type expressions, too. Due to the prefix search, we can only search for type signatures that begin with *HTMLExp*, but a function's type signature that yields the type *HTMLExp* ends with *HTMLExp*. The first idea is to split a given type string on function arrows $\to$ and add each part to the context. For the type expression

$$(Int \to String) \to String \to HTMLExp$$

we get the partitions "(*Int*", "*String*)", "*String*", "*HTMLExp*". As first consequence, we lose the function type $(Int \to String)$. Secondly, we cannot only add simple or constructor types (like *String* or *Maybe Int*) to the index if we want to provide the search for type expressions with at least one function arrow. Thus, we decided not to convert the *TypeExpr* to the corresponding string, but to decompose the type expression into all its valid suffixes first. In this way, each suffix is paired with the context *"signature"*, converted into a type expression represented as *String* and added to the index. For example, if we want to add the type expression

$$(Int \to String) \to String \to HTMLExp$$

to the index, we actually add the following search keys:

```
("signature","(Int -> String)"),
("signature","String"),
("signature","HTMLExp")
```

Since *TypeInfo* stores a list of *TypeExpr* representing its constructors' type signatures, we have to apply the same splitting mechanism to all elements of the list of *TypeExpr*. Additionally, we have to add the constructor name to the index manually because it is not part of the *TypeExpr*s, since its usage is rather similar to a function than to a type. Table 4.2 summarizes the contexts of *TypeInfo*, whereas there are still contexts of *FunctionInfo* left to be discussed. We distinguish between non-/deterministic and

TypeInfo	
property	context name
name	"type"
constructor	"function"
corres. module	"inModule"
signature	"signature"
description	"description"

Figure 4.2.: Contexts for a *TypeInfo* Data Structure

flexible vs. rigid functions. For each characteristic that applies to a function, we add the given context to the index. In case of a deterministic and flexible function, we add (`"flexible",""`) and (`"det",""`). The summary of all contexts concerning a function is shown in Table 4.3.

Summing up, we process *CurryInfo*'s substructures *ModuleInfo*, *FunctionInfo* and *TypeInfo* in order to pair their information with a context.

4.2.2 Document Construction

As the next step, we take a look at the second structure of the index: the documents. In this context, we discuss the document's role and value as part of the triple.

FunctionInfo	
property	context name
name	"function"
corres. module	"inModule"
signature	"signature"
description	"description"
flexible/rigid	"flexible" "rigid"
non-/deterministic	"nondet" "det"

Figure 4.3.: Contexts for a *FunctionInfo* data structure

Each document, that we want to pass to the Holumbus framework, needs to be stored in the data structure

$$\begin{aligned}\textbf{data}\ \mathit{Document}\ a = \mathit{Document}\ \{\,&\mathit{title} :: \mathit{Title}\\ ,\,&\mathit{uri}\ \ :: \mathit{URI}\\ ,\,&\mathit{custom} :: (\mathit{Maybe}\ a)\}\end{aligned}$$

that consists of a title, URI and customizable information. The latter has the polymorphic type a and determines the type for a document. For example, if we have a document for the *CurryInfo* structure, we use *Document CurryInfo* as data structure.

For the creation of the index, we have to feed these documents with actual data. As mentioned before, we can read the *CurryInfo* structure and use it in the process. The first idea is to construct a document with *CurryInfo* as data structure for the custom information. This idea is easy to implement since we just use the unmodified data structure that CurryDoc produces. As consequence, all the information in the corresponding index maps to this document only. When we search our index for an information, we can relate a given search result to the corresponding *CurryInfo*, only; we cannot distinguish if the search result is associated to the module, function or type information of the given *CurryInfo* structure.

In order to provide a more differentiated representation of a Curry module in the index, we choose not to use *CurryInfo* but its substructures *ModuleInfo*, *FunctionInfo* and *TypeInfo* as document types, i.e., the custom information. However, if we want to

distinguish between these three sources of information, we need to store three types of documents. Nevertheless, this decision allows us to relate the information of a Curry module to its functions, types or information about the module itself.

Furthermore, we decide that a Curry module is converted into more than this three documents. We rather want to create a document for each function and data structure of a given Curry module plus the module's general information. In the end, we trace back a search result to the exact function, data structure or module information since we can take advantage of the devision of the *CurryInfo* data structure.

This design already determines the decision regarding the title and the URI. The title corresponds to the name of the function, type or module. The value of the URI is an argument to fill by the user when generating the index; the URI can point to a local or online source for documentation. We designed the URI representation according to the HTML-documentation provided by CurryDoc because the current main source for Curry documentation, that can be accessed online[2], is generated via CurryDoc. Since this HTML-structure of a Curry module documentation provides anchors to the module's defined functions and data structures, we use this link mechanism for our URIs as well. Thus, the URIs are build according to the following schema:

$$
\begin{aligned}
\mathit{moduleURI} &= \mathit{baseURL} +\!\!+ \mathit{moduleName} +\!\!+ \texttt{".html"} \\
\mathit{functionURI} &= \mathit{baseURL} +\!\!+ \mathit{moduleName} +\!\!+ \texttt{".html"} +\!\!+ \texttt{"\#"} +\!\!+ \mathit{funcName} \\
\mathit{typeURI} &= \mathit{baseURL} +\!\!+ \mathit{moduleName} +\!\!+ \texttt{".html"} +\!\!+ \texttt{"\#"} +\!\!+ \mathit{typeName}
\end{aligned}
$$

4.2.3 Example

In the following, we list the steps of an example index construction for one Curry program; we only consider the information about the functions of a given Curry module.

- At first, we extract the list of *FunctionInfo* of the given *CurryInfo* structure.
- For each *FunctionInfo*, we construct the pair of contexts and words and return the list of pairs.

[2] `http://www.informatik.uni-kiel.de/~pakcs/lib/CDOC`

```
  -- contexts
[("function", functionName),
  ("inModule", functionModule),
  ("signature", functionSignature),
    ("flexible", functionFlexibleRigidStatus)
  or ("rigid", functionFlexibleRigidStatus),
    ("nondet", functionNonDetStatus)
  or ("det", functionNonDetStatus),
  ("description", functionDescription)
]
```

- Next, for each *FunctionInfo*, we need to construct the *Document FunctionInfo* structure to store the data structure.

```
  -- new document
Document { title = functionName
         , uri   = uriPath
         , custom = Just theFunctionInfoStructure }
```

- We add all documents to construct the collection *Documents FunctionInfo* and the first part of the index is complete.
- When we add each document to the collection, we can access the unique identifiers. We use these identifiers to add the required document reference for the lists of context-word-pairs.
- In order to construct the *Inverted* structure, we use a function Holumbus provides to build this structure from a list of the required triples.
- In the end, we merge all *Inverted* structures.
- In case of (the first module of) a new index construction, the process is complete and we gained a structure representing the index that is written into a file.
- Otherwise, we need to merge the new constructed index structure with an existing index.

4.2.4 Conclusion

We implemented a mechanism to create an index data structure that is traversed, when we perform a search query, and one mechanism to store the corresponding document that is linked to the index structure again. We create these index structures for the three substructures of *CurryInfo* and, together, they represent the index. In the end, each index structures – for modules, functions, types – is written into two files (separating search key and document structure again); these files serve as index.

Since we do not want our search engine to be built on just one Curry program, we can update the index with new data. We already adressed the problem concerning lazy evaluation but we ran into another problem as well. When we write the index structure into a file, the data is compressed into structures named *CompactInverted* and *SmallDocuments*. This means, when we load our files again, the search keys as well as the document structures do not harmonize with the original data structures, *Inverted* and *Documents*, anymore. We solve this problem by creating the *Inverted* and *Documents* data structures as usually, and, as second step, we convert *Inverted* into the *CompactInverted* data structure and *Documents* into the *SmallDocuments* data structure. We cannot merge the structures just yet, since we created new documents with identifiers starting with 1. We need to adapt the *DocId*s such that the minimum of the new structure is the maximum of the old one. Thereafter, we can merge the documents and index structure and write the new file.

In the end, there are two ways to build an index with at least one given *.cdoc*-file of a Curry program. One can start the index creation for the given file and the URI to the corresponding documentation to either create a new index or to update an existing one. We also provide a mechanism to read a *.txt*-file consisting of pairs of paths to *.cdoc*-files and the corresponding URI. Further information about the usage is provided in Appendix B.

4.3 Searching

In the main part of this section, we discuss the general idea of a parser, connect this idea with our search engine and develop a simple parser as an example. The example

addresses the problem of parsing expressions that can occur with and without parentheses. We choose this example because we want to support the same functionality for the user queries in our search engine. Furthermore, we introduce *Parsec*[3], a Haskell library to build a fast parser. In this context, we present some code examples taken from the implementation of Curr(y)gle and outline the advantages of using Parsec as well as some problems that occurred during the development of the parser for Curr(y)gle.

The process of searching mainly consists of parsing the query since the Holumbus framework already takes care of other requirements and tasks, like processing the query and finding the result structure in the index. The only task left is to submit a score calculation to Holumbus and preprocess the given result for the web application. Therefore, we mainly focus on the query parser and just give a brief overview of the document retrieval.

4.3.1 General Idea and Usage of Parsers

A parser is used to analyse a given input and compose a new data structure depending on the information of the input. Parsing is an important topic among functional programmers and hence many papers discuss the development of parsers. The most popular approaches use monadic parsers [6] but there are also alternatives using higher-order functions [2].

In general, we want a parser to take an input value and return a new structure. Thus, a parser can be described as follows:

type $Parser\ \sigma\ \alpha = \sigma \rightarrow \alpha$

With the type definition above, we can run one parser on a given input. Commonly, this input is of type *Char*, thus, we will use the type for the following parsers. The main idea behind parsing is to apply several parsers and combine the results. More precisely, we want to use a second a parser for the characters that the first parser did not consume. In order to combine the results, the parser type needs a pair consisting of the parsing result and the rest of the input, which has not been consumed. Additionally, we need to consider that the parsing of an input may be ambiguous. This means, there can be more than one way to parse the input and it is also possible that the input cannot be parsed at all. Thus, we extend the result type to a be a list of pairs, representing the different

[3] `http://legacy.cs.uu.nl/daan/parsec.html`

parsing alternatives or the empty list, if the parser fails. The idea behind this approach was introduced by Philip Wadler [9]. Additionally, if we think about combining several parsers, we have to adjust the applied argument. Instead of just one *entity* of the input type, we want to parse *several* input values with *several* parsers. Therefore, we apply the parser to a list of *Char*, i.e, *String*, and get the following type for the corresponding parser.

$$\textbf{type } \mathit{Parser}\ \alpha = \mathit{String} \to [(\alpha, \mathit{String})]$$

Summing up, in general, a parser is parametrized with the type of the elements to parse, in the following we use *Char*, and the resulting type α. The parser takes a *String* and returns a list of possible parsed structures and the remaining characters. These structures are pairs, where the first entry is the result and the second entry represents the remaining input. An empty list denotes failure, whereas a non-empty list stands for success.

In order to make use of more than one parser, we need parser combinators. Parser combinators take two or more parsers as arguments in order to construct a new parser that behaves like the combination of the given parsers. At first, we take a look at a combinator for alternatives (also: choice combinator).

$$(\oplus) :: \mathit{Parser}\ \alpha \to \mathit{Parser}\ \alpha \to \mathit{Parser}\ \alpha$$
$$p \oplus q = (\lambda ts \to p\ ts +\!\!+ q\ ts)$$

For a given input, we apply parser p first and concatenate the resulting list with the result of the application of parser q to the input. We get all possible combinations that can be parsed by parser p or parser q.

Additionally, we want to combine parsers sequentially. There are three possibilities to combine two parses, the first two are similar to the monadic operator $\gg$ and the third one follows the same idea as the monadic bind operator $\gg\!=$. We start with the latter combinator. The following operator allows us to use the parsers sequentially and combine their results.

$$(\circledast) :: \mathit{Parser}\ (\alpha \to \beta) \to \mathit{Parser}\ \alpha \to \mathit{Parser}\ \beta$$
$$p \circledast q = \lambda ts \to [(f\ x, ts_2) \mid (f, ts_1) \leftarrow p\ ts, (x, ts_2) \leftarrow q\ ts_1]$$

The idea behind sequential application is to combine two parsers in order to gain a new parser. More precisely, we run the first parser on an input and can run the second parser

on the remaining input that the first parser did not consume. At first, we apply the parser p to the input and gain a tuple consisting of $f :: \alpha \to \beta$ and the rest of the input $ts_1 :: [\sigma]$. Then we apply parser q to the remaining input, which results in a further tuple $(x, ts_2) :: (\alpha, String)$. Finally, we apply the gained function $f\ x :: \beta$ and return it as pair with the remaining input ts_2.

In order to actually use the binding operator $\circledast$, we have to apply a function to combine the results.

$$(\circledS) :: (\alpha \to \beta) \to Parser\ \sigma\ \alpha \to Parser\ \sigma\ \beta$$
$$f\ \circledS\ p = \lambda ts \to [(f\ x, ts_1) \mid (x, ts_1) \leftarrow p\ ts]$$

The operator $\circledS$ works similar to the function *map* because the first argument is a function and the second argument of $\circledS$ is applied to this function. At first, the function runs the parser p for a given input, applies the given function to the first entry of the resulting pair and returns the modified result.

We define the other two sequence combinators to show the interaction of the two functions $\circledast$ and $\circledS$. The other two sequence combinators also take two parsers, but discard one parser's result and return the other one's result.

$$(*\!>) :: Parser\ \alpha \to Parser\ \beta \to Parser\ \beta$$
$$p\ *\!>\ q = (\lambda\_\ qResult \to qResult)\ \circledS\ (\lambda ts \to (p \circledast q)\ ts)$$

This function discards the result of the first parser and returns the result of the second. The function $(<\!*)$ works the other way around: the second result is discarded and we return the result of the first parser.

These functions illustrate the fundamentals to build combinatorial parsers. In order to get a better idea of the usage, we present a small example.

At first, we want to construct a parser to read a character and return it in the resulting pair. Otherwise the parser returns an empty list for an empty *String* as input.

```
parsePredicate :: (Char → Bool) → Parser Char
parsePredicate predicate ts =
  case ts of
    [] → []
    (t : tss) → if predicate t then [(t, tss)] else [([], t : ts)]
```

With this function we can define simple parsers like *parseAlphaNum* that parses any alphanumeric characters

$$
\begin{aligned}
&\mathit{parseAlphaNum} :: \mathit{Parser\ Char}\\
&\mathit{parseAlphaNum} = \mathit{parsePredicate\ isAlphaNum}
\end{aligned}
$$

or parseT that parses the character t.

$$
\begin{aligned}
&\mathit{parseT} :: \mathit{Parser\ Char}\\
&\mathit{parseT} = \mathit{parseSymbol}\ (t \equiv)
\end{aligned}
$$

We use the same scheme as above in order to construct a parser that reads a left parenthesis and a parser that reads a right parenthesis.

$$
\begin{aligned}
&\mathit{parseLeft} :: \mathit{Parser\ Char}\\
&\mathit{parseLeft} = \mathit{parsePredicate}\ (\texttt{'('} \equiv)\\
&\mathit{parseRight} :: \mathit{Parser\ Char}\\
&\mathit{parseRight} = \mathit{parsePredicate}\ (\texttt{')'} \equiv)
\end{aligned}
$$

Next, we need to define a parser for the word between the parentheses. Since we already have a parser for characters and we want to parse a string, the main idea is to parse a sequence of characters, i.e., to combine a sequence of *parsePredicate*.

$$
\begin{aligned}
&\mathit{parseWord} :: \mathit{Parser\ String}\\
&\mathit{parseWord}\ \mathit{input} =\\
&\quad (\lambda(c, cs) \rightarrow c : cs)\ \textcircled{\scriptsize S}\ (\mathit{parseAlphaNum} \circledast \mathit{parseWord})\ \mathit{input}
\end{aligned}
$$

For a given input string we run *parseAlphaNum* that reads the first character. Next, *parseWord* is executed with the remaining input, hence we read one character after another until the parsing fails or the input is read entirely. The parsed characters are then composed to a list and the parser returns the parsed string, i.e., the sequence of alphanumeric characters that we parsed.

Finally, we want a parser to read a parenthesized expression and discard the surrounding parentheses.

$$
\begin{aligned}
&\mathit{parseParenWord} :: \mathit{Parser\ String}\\
&\mathit{parseParenWord}\ \mathit{input} =\\
&\quad (\lambda\_\ \mathit{expr}\ \_ \rightarrow \mathit{expr})\ \textcircled{\scriptsize S}\ (\mathit{parseLeft} \circledast \mathit{parseWord} \circledast \mathit{parseRight})\ \mathit{input}
\end{aligned}
$$

With these parser combinators, we read the left parenthesis first, then the expression and finally the right parenthesis. The function to combine the three results just returns the

result of the function *parseWord*. If we are more precisely, the parser above does not parse an expression with *or* without parentheses like we promised before. Instead, the parser *parseParenWord* only parses a parenthesized sequence of alphanumeric characters. In order to fix the parser, we need to add the choice combinator. Additionally, we want to present a second implementation to discard the read parentheses.

$$
\begin{aligned}
&\mathit{parseParenWord2} :: \mathit{Parser\ String}\\
&\mathit{parseParenWord2} =\\
&\quad \lambda ts \rightarrow (\mathit{parseLeft} \mathbin{*\!\!>} \mathit{parseParenWord2} \mathbin{<\!\!*} \mathit{parseRight})\\
&\qquad \oplus\ \mathit{parseWord}
\end{aligned}
$$

Summing up, we introduced the basic idea of parsers and implemented a small parser that can read a word with or without parentheses. In the following, we have to keep in mind that we build a parser by combining several parses that parse substructures.

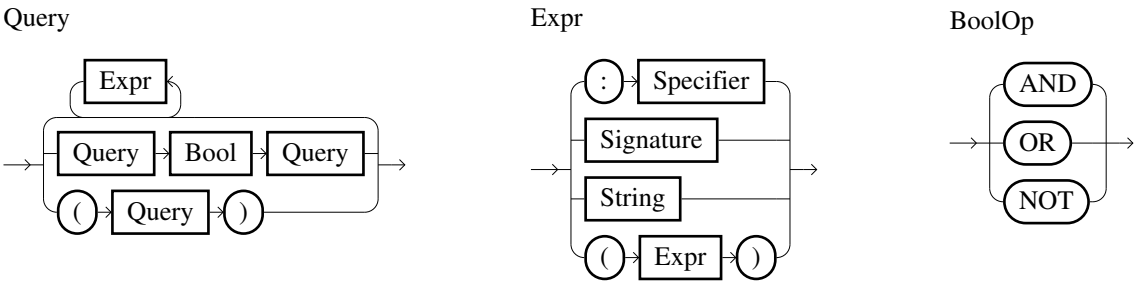
4.3.2 Parsing User Queries

In our implementation we use the Haskell library Parsec to build the parser to read and analyse the user queries. The main idea of Parsec is to use parser combinators instead of parser generators. The latter can generate a parser for a given grammar, whereas the combination of several parsers correlates to the idea we presented above.

The Parsec library called our attention, since the Holumbus framework uses the library to provide a simple parser. The parser handles binary operations like *AND*, *OR*, and *NOT* for user queries. In the previous chapter we already mentioned our requirements on a parser for user queries. The parser that Holumbus provides, does not fulfill these requirements, therefore, we implemented our own parser. In Appendix C, we give an overview of the language that we provide for the user queries. After the previous introduction of the basic ideas of a parser, we want to give an overview of the query parser that Curr(y)gle provides. Following the overview, we want to present an excerpt of our implementation.

The following syntax diagram illustrates the parseable queries.

A query can consists of context specifiers (`":module IO"`, `":function map"` etc.), type expressions (`"Int -> Int"`) and *simple* search terms (`"map"`) and the parenthesized version of all three components. Additionally, a query can be a sequence



of the previous components or these components can occur in combination with binary operators, like *AND*, *OR*, *NOT*. The presented parsers in the following section already consider any trailing whitespaces, such that, the top-level parser for queries needs to parse all leading whitespaces.

On the first layer, we apply the parser for the binary operations, since we want to combine specifiers, signatures and *simple* search words. For search queries that consist of several context specifications and a simple search word, the parser implicitly combines the substructures with conjunctions, i.e., "*:function map* $a \to b$" searches for a function starting with *map*, whose type consists of the type expression $a \to b$. The parser for signatures is executed in case of the context for type expressions and also for any word starting with an upper-case character that forms a valid type expression. This means, the search terms "*:signature IO*" and "*IO*" yield the same *Query*-structure, hence, the same search results. On the contrary, type variables alone are not parsed implicitly because we also want to search for words without context, i.e., we search without a context for any word starting with a lower-case character.

Additionally, the query parser always tries to parse as much words as possible. For example, if a valid expression is followed by nonsense, we parse the input as far as possible and discard the remaining input, i.e., "*IO* 1 _!@#^" yields the same results as "*IO*". We also parse unbalanced parentheses that occur in the beginning or end of a search query, e.g., the search quert $((Int \to Int)$ yields the same as $(Int \to Int))$ and $(Int \to Int)$. We expect an intuitive usage of the search engine of this underlying parser and all the features the parser provides. In addition to the EBNF that we have already mentioned in the previous chapter, we also provide all figures of the following section and the additional parsers that represent the query parser altogether (see Appendix C.2).

4.3.3 Parsing Type Expressions

The following code focuses on our implementation of the parser for type expressions, since such expressions are the main reason we decided to build a new parser. Furthermore, the implementation uses some special features Parsec provides that we want to introduce.

At first, we take a look at all kinds of type expression that can compose a type signature for a function or constructors of data types.

- type variables – a
- simple types – Int, $Bool$
- type constructors – $Maybe\ Int$, $IO\ ()$
- function types – $Int \to Bool$
- list types – $[Int]$
- tuple types – $(String, Int)$

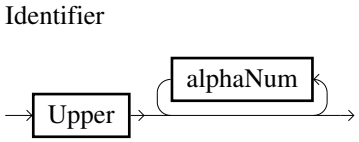
We use these substructures to build our parser for type expressions. Each item on the list needs to be analysed by its own parser and step by step, we combine all parsers according to valid Curry syntax for type expressions.

If you take a closer look at the list of type expressions, you see that all identifiers we have to parse, start with an upper-case letter, except for type variables. Thus, at first, we need a parser that reads all valid identifiers. Luckily, Parsec provides a feature to define tokens that are used by the language and compose a parser for these tokens. Tokens are constructs like whitespace, comments, identifiers, reserved words, numbers or strings.

The module *ParsecToken* exports a function *makeTokenParser* that takes such a language definition as argument and returns a record with a set of lexical parsers. Every lexical parser already considers trailing whitespaces, hence we do not need to consider whitespaces when we use a parser of the language record. The following code shows our language definition for type expressions:

```
signatureDef = emptyDef {identStart      = upper
                        , identLetter    = alphaNum
                        , reservedNames = ["AND", "NOT", "OR"]}
```

In order to define such a language, we take an empty definition record and update the fields that we want to use. Identifiers start with an upper case (we handle type variables separately) and consist of alphanumeric characters. The reserved names are words that



are not allowed to occur as name in a type expression. Our parser also knows the binary operations *AND*, *NOT*, and *OR* and since they are starting with an upper character, they are potential type expressions. Therefore, we need the parser to fail on these reserved names when parsing type expressions.

We use the defined lexical parsers in our implementation, including *sigIdentifier*, *whitespace*, *aSymbol* and *lexemer*, where the latter takes a parser as argument and parses trailing white spaces; *aSymbol* does the same, but takes a string as argument. Additionally, we have parsers *paren* and *bracket* for parentheses and brackets.

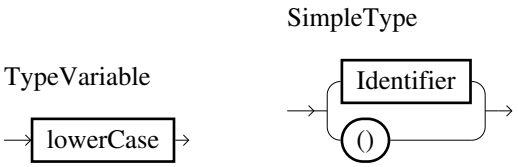
Before we examine the first type expression, we need to consider the type of our parser. We are parsing a *String* into a *TypeExpr*, thus, in the definitions we used above, the type of our parser corresponds to:

```
type TypeExprParser = Parser String TypeExpr
```

In contrary to the parser structure we presented above, Parsec does not return a list of possible results. The data structure follows a different approach to determine if the parser application was successful or not. On success, the parser returns the longest possible match. The first substructures we want to parse are type variables and simple types. Therefore, we first take a look at the following syntax diagrams.

A type variables is a single lower-case character.

```
varParser :: TypeExprParser
varParser = var Ⓢ lower <* notFollowedBy alphaNum)
```



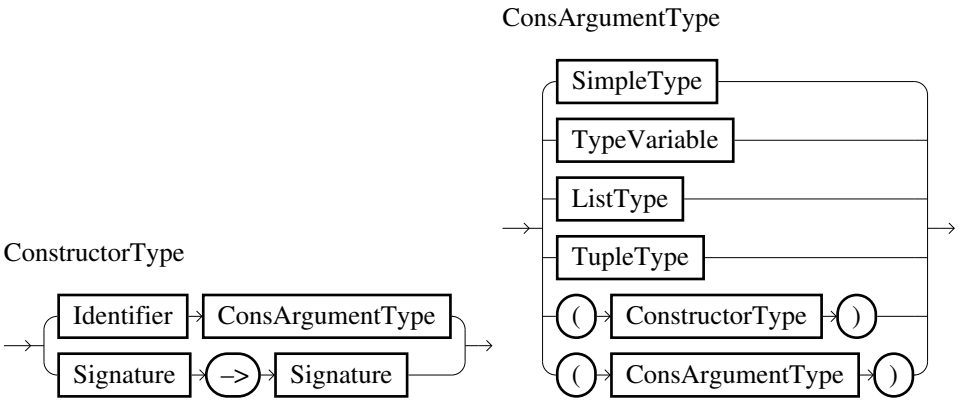
Thus, we do not allow type variables like *abc*. Thus, for a tye expression like *IO abc* the function *varParser* fails on *abc* and we only read *IO* as a simple type. After parsing, the function *var* converts the character into a *TypeExpr*.

The next parser handles simple types like *Int* and *Bool*, but also the unit type (). For simple types, we need to parse one identifier and consider the unit type () as special constructor.

$$primParser :: TypeExprParser$$
$$primParser = prim \circledS\ (sigIdentifier \oplus aSymbol\ \texttt{"()"})$$

The function *prim* wraps this identifier into a *TCons*-structure with no type arguments, e.g., *Int* is represented as *TCons* (`"Prelude"`, `"Int"`) [].

In case of n-ary type constructors, we parse at least one identifier, a whitespace, and another type expression. For a better understanding, the following figures illustrate the idea of a parser for constructor types.



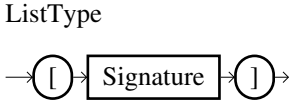
At first, we parse an identifier like we do for primitive types and additionally, we need whitespace and another type expression to follow.

```
consParser :: TypeExprParser
consParser =
  ((λconstr _ expr → cons constr expr)
    Ⓢ sigIdentifier ⊛ whitespace ⊛ sepBy1 (signatureTerm False) whitespace)
```

The function *sepBy1* takes two parsers as arguments, where the second one parses a separator that occurs between the input of the first parser . At least one type expression (but no whitespace) needs to occur, otherwise the parser *sepBy1* fails and altogether *consParser* fails. Thus, in the *consParser* definition above, we parse at least two type expressions separated by a white space or just one type expression. The parser *signatureTerm* handles all the substructures we listed above, except for function types, but we discuss the parser for function types later. The boolean value in the function call *signatureTerm* indicates, if a constructor type may occur without parentheses. In the definition above, constructor types after the first separator are only allowed to occur with parentheses.

For example, we want to parse constructor types with one argument, like *Maybe a* and *IO a*. If we combine these type constructors to one type expression, we get *Maybe* (*IO a*) or *IO* (*Maybe a*). We have to parenthesize the inner type constructor because otherwise constructions like *IO Maybe a* and *Maybe IO a* suggest that both constructors take two type arguments instead of one.

Next, let's take a look at the parsers for list types and tuple types, starting with lists.



A list can be any valid type expression enclosed by brackets.

```
listParser :: TypeExprParser
listParser =
  (λtexpr → cons "[]" [texpr]) Ⓢ bracket (signatureParser True)
```

We wrap the result in a type constructor with [] as constructor and the type expression as type arguments, e.g., *TCons* (`"Prelude"`, `"[]"`) [*TCons* (`"Prelude"`, `"Int"` [])] for a list of *Int*. Whereas *signatureTerm* is the parser for any type expression besides function types, *signatureParser* combines *signatureTerm* and function types, thus, it the

main parser for type expressions. The boolean value in the function call *signatureParser* indicates again, if a constructor type may appear without surrounding parentheses.

Next, we present the parser for tuple types in three steps. The first (and main) part

looks quite similiar to the parser for constructor types. Only this time we parse any kind of type expressions seperated by a comma instead of a white space.

```
parseTuple =
  (λitem _ itemList → item : itemList) Ⓢ
    (signatureParser True)
    ⊛ aSymbol ","
    ⊛ sepBy1 (signatureParser True) (aSymbol ",")
```

We combine the first type expression and the list of following type expressions to a list. This list represents the type arguments for the tuple constructor. Next, we need to build the tuple constructor because the name of the constructor depends on the number of arguments: $(,)$ is a tuple constructor for a pair, whereas for a triple the name of the constructor is $(,,)$.

```
tupleCons list = "(" ++ replicate (length list − 1) ',' ++ ")"
```

Depending on the length of the list of type arguments, we construct the tuple constructor. And in the end, we combine the functions *parseTuple* and *tupleCons* to gain a parser for tuples:

```
tupleParser :: TypeExprParser
tupleParser =
  ((λtuple → cons (tupleCons tuple) tuple)
    Ⓢ paren parseTuple)
```

The last parser we need to discuss handles function types like $Int \to Int$. For an infix operator, we always need to look ahead after parsing an identifier, if we come across the

→-operator next. Luckily, the Parsec library provides a mechanism simplify the parsing of such operators.

At first, we define the fixity and associativity of the operator and assign a function that determines the result of the parsing.

```
signatureTable =
  [[Infix ((λ_ → FuncType) Ⓢ (aSymbol "->" x)) AssocRight]]
```

When parsing a function arrow →, we return the partial application of *FuncType* because the two arguments of this constructor are the type expression to the left and right of the operator. In order to use this parser, we use Parsec's function *buildExpressionParser* that takes such an definition table and a parser as arguments; the arguments of the function arrow are then applied to the passed parser.

```
signatureParser :: TypeExprParser
signatureParser =
  buildExpressionParser signatureTable (signatureTerm True)
```

In the definition of *signatureParser*, we assign *signatureTerm* as parser for the arguments of the function types. We need to parametrize the function *signatureTerm* in order to indicate if a constructor types nees to be parenthesized. A list or a tuple can consist of functions types or type constructors without parentheses, whereas for a valid constructor type, a function type or a nested contructor type as argument needs to be parenthesized. Additionally, we want to parse redundant parenthesized type expressions as well. We can offer this functionality with the following definition of signatureTerm:

```
signatureTerm :: Bool → TypeExprParser
signatureTerm allowConsParser =
  (guard allowConsParser ≫ try consParser)
  ⦶ try tupleParser
  ⦶ paren (signatureParser False)
  ⦶ listParser
  ⦶ primParser
  ⦶ varParser
```

In order to guarantee that we can parse tuples, parenthesized function types and redundant parentheses, we cannot consume any characters in case *tupleParser* fails, otherwise

the parser for parenthesized expressions fails, too. Therefore, we have to try *tuplesParser* first; if it fails, the result is discarded and we can try for parenthesized expressions next. We do not need a try for the parenthesized expression because there are no other valid expressions that start with an opening parenthesis, since we already ruled out the possibility of tuples. Therefore, the parser *paren* (*signatureParser False*) either fails on the first character of an input or parses a valid parenthesized expression.[4] The same logic holds for the following three parsers, since they all expect distinct first characters. For lists, the first input character has to be a left bracket, *primParser* only accepts an upper-case character and *varParser* the opposite, a lower-case character.

In the end, we can parse any valid type expression, which is then converted into a *Query*-Structure in order to be passed to Holumbus and to start the searching process.

4.3.4 Document Retrieval

Up to this point, we already constructed the index with information about Curry modules, functions and types. This information are given by search terms and the corresponding documents. We can also parse a given user query in order to construct the required data structure to communicate with Holumbus. The last part that we want to look at, is the scanning of the search words and the retrieval of the documents. This part is realized by Holumbus, therefore, we only want to sketch the idea instead of discussing implementation details.

Let us assume, we have the index illustrated in Figure 4.4, consisting of search keys and documents. When we search for `"New"`, Holumbus scans the column *search word* of the given table. In our example, Holumbus finds two rows, where this search word occurs: the module with the name *New* and the type with the name *NewPair*. Thus, Holumbus retrieves the documents that fit the identifiers that are recorded in these two rows, i.e., the documents with the identifier 1 and 2, respectively. If we restrict our query and search for `":module New"`, Holumbus only considers search words that are combined with the context *module*. Thus, in our example, Holumbus returns the document of the module information with the identifier 1.

[4] Here, we only consider well-balanced parentheses.

context	search word	ID
module	New	1
author	Sandra	1
author	Dylus	1
description	Testing	1
description	CurryDoc	1
type	NewPair	2
signature	(Int, String)	2
description	Number	2
description	Text	2

ModuleInformation

Name: *New*
Author: *Sandra Dylus*
Description: *Testing CurryDoc*
1

TypeInformation

Name: *NewPair*
Signature: *(Int,String)*
Description: *Number & Text*
2

Figure 4.4.: Example Index Consisting of Search Keys and Documents

In summary, we prepare and provide the information that Holumbus traverses when it processes a search query and provides us with the document that fits the given query.

5

Conclusion

In the final chapter, we complete this thesis with a conclusion and give an outlook on future works. At first, we review the progress of our implementation and tie in with our introduction and the goals, requirements, and ideas we had in the beginning. Furthermore, we provide an overview of ideas for future works.

5.1 Summary and Results

The main goal of this thesis was to build a search engine for the functional logic programming language Curry in order to provide a simple way to look up functions, modules, and data types of Curry libraries, e.g., the libraries distributed with PAKCS. Therefore, we presented the CurryDoc tool to generate informative documentations for a given Curry program. In Chapter 4.1, we demonstrated an extension of CurryDoc in order to create the source of the search engine's traversed data; the information is extracted from Curry modules. This source produces a data structure that represents the given Curry program that is then analysed in the following section in order to create an index for the search engine. The index consists of the data we want to search for, i.e., the API of the mentioned Curry libraries. The data we are indexing can be paired with a "context" or, more precisely, every information we are indexing is categorized corresponding to its "role" in the Curry program, e.g., a function's name, the description of the module, a data type's signature. These pairs of contexts and words allow us to improve our search engine because we can restrict a search term to one of these contexts. These restrictions can minimize unwanted results and, therefore, make a contribution to a user-friendly

search engine. In order to provide the usage of such specifiers to restrict a search term, we need a non-ambiguous language for the search queries. Therefore, we have presented a parser to analyse the search term in the last section of Chapter 4. In the end, we have build a search engine to run as a web application.

5.2 Outlook

Last but not least, we want to discuss some ideas to extend this project. One functionality we do not provide is an *intelligent* way to search for polymorphic types like Hoogle, the search engine for Haskell, does. For example, let's assume we search for a function that concatenates two lists of integers. The best idea is to search for the signature of this function, i.e., `"[Int] -> [Int]"`. Unfortunately, we will not find any results, since there is no function with this specific type. However, there is a function to concatenate two lists of elements with an arbitrary type, i.e., $[a] \to [a]$. Like we mentioned, Hoogle uses a mechanism to search for more general types in order to find the suitable function for our example. In our opinion, there is one simple approach to gain this functionality and another one that requires more resources. The first approach just replaces each appearance of a type with a fixed type variable. For our example, we can transform $[Int] \to [Int]$ to $[a] \to [a]$ if we look at the first type Int and replace this and all following occurrences of Int with the type variable a. The next different type expression is then replaced by the type variable b et cetera. Another idea is to unify each type expresion, that we have in the index, with the searched signature and return the successful unifications along with the matched type expressions. In the development period of this search engine, we simply did not find the time to concern ourselves with this topic. Albeit, we think such a feature would be an excellent addition of the already existing functionality.

Furthermore, we think that the widely-used "did you mean?" suggestion can make Curr(y)gle even more user-friendly. In case of misspellings, this feature helps to find words that are lexicographical similar with the search term in order to proceed the search. The Holumbus framework already provides a fuzzy search that can help to find the best result for misspelled words.

At last, we want to mention, that the interaction between the CurryDoc extension and the Haskell implementation could be more modular. We tried to adjust the search engine to the existing CurryDoc and its function of generating documentation. For example, we compose the URI to provide the reference between a result entry and the documentation in the implementation of the search engine. Another approach would be to provide this kind of information in the CurryDoc extension because, then, we can automatically consider adjustments in the documentation process, like changes in the URI construction. The used contexts are another aspect; maybe a more flexible adjustment of the context names is desirable. One method could be to construct the *CurryInfo* data structures with several tuples, where the first entry represents the context and the second entry the value. For example, *ModuleInfo* could look like this:

data *ModuleInfo* =
  (*String*, *String*) -- (context, name)
  (*String*, *String*) -- (context, author)
  (*String*, *String*) -- (context, description)

These are just some ideas we had in the end of our development. However, we had to differ between requirements and features and, of course, we decided to focus on the requirements. We think that these features, that we just mentioned, are purely optional and our search engine meets the requirements we made in the beginning, provides a good user-experience and is a good starting point for further developments. During the implementation of the CurryDoc extension, we wanted to look up a lot of functions and suffered from the lack of a Hoogle-equivalent. We got really frustrated when browsing the online documentation and reading each module in search of the convenient function. Hence, we hope to do some good with this search engine and help to provide a more efficient work-environment for Curry.

A

CurryDoc Instruction

A.1 Installation

The installation requires a running Curry, we recommend the PAKCS or KICS2[1] implementation.

Next, you need to modify the included *Makefile*, the path directories have to be adjusted to your system. Afterwards, you can execute the Makefile in order to compile CurryDoc.

A.2 How-to-use

You can either generate the *.cdoc*-file itself or the HTML representation, since the latter generates the *.cdoc*-file as well. Furthermore, you can pass the directory of the Curry module an its name, if the module is not located in your current directory.

```
$ currydoc [--nomarkdown] [--html|--tex|--cdoc] <module_name>
$ currydoc [--nomarkdown] [--html|--tex|--cdoc]
   <doc directory> <module_name>
$ currydoc [--nomarkdown] --noindexhtml
   <doc directory> <module_name>
$ currydoc --onlyindexhtml <doc directory> <module_names>
```

[1] http://www-ps.informatik.uni-kiel.de/kics2/

A.2.1 Example

The following code presents the data structures that is generated for a given Curry module with the implemented extension for CurryDoc.

```
(CurryInfo
  (ModuleInfo "NewPair"
    "Sandra Dylus"
    "Testing the .cdoc generation of CurryDoc\n")
  [(FunctionInfo  "left"
                  (FuncType (TCons ("NewPair","NewPair") [])
                            (TCons ("Prelude","Int") []))
                  "NewPair"
                  "Returns the first entry of a NewPair\n"
                  False
                  KnownFlex),
    (FunctionInfo "right"
                  (FuncType (TCons ("NewPair","NewPair") [])
                            (TCons ("Prelude","[]")
                              [(TCons ("Prelude","Char") [])]))
                  "NewPair"
                  "Returns the second entry of a NewPair\n"
                  False
                  KnownFlex)]
[(TypeInfo "NewPair"
           [(("NewPair","Pair"),[(TCons ("Prelude","(,)")
                                  [(TCons ("Prelude","Int") []),
                                    (TCons ("Prelude","[]")
                                    [(TCons ("Prelude","Char") [])])])])]
           []
           "NewPair"
           "Pair of a number and a text\n"
           False)])
```

The data structures was generated with the following exemplary Curry module *NewPair*, which contains two function definitions and a single definition of a data type.

```
  -- Testing the .cdoc generation of CurryDoc
  -- @author Sandra Dylus
module NewPair where
  -- Pair of a number and a text
data NewPair = Pair (Int, String)
  -- Returns the first entry of a NewPair
left :: NewPair → Int
left (Pair (int, str)) = int
  -- Returns the second entry of a NewPair
right :: NewPair → String
right (Pair (int, str)) = str
```

B

Installation and Usage of Curr(y)gle

B.1 Installation

The distributed source directory contains a *.cabal* file; for the installation you simply have to run cabal.

```
$ cabal configure
$ cabal build
$ cabal install
```

Alternatively, you can use the provided *Setup.hs*.

```
$ runhaskell Setup.hs configure
$ runhaskell Setup.hs build
$ runhaskell Setup.hs install
```

B.2 How-to-use

The installation process creates two binaries: *curryIndexer* and *curryServer*. The first creates the index for a given directory that contains at least one *.cdoc*-file and a corresponding URI. The *.cdoc*-file can be generated with the extended CurryDoc for a given Curry module, the URI is the corresponding HTML documentation that can also be generated with CurryDoc. This package provides an example directory with three Curry modules generated with CurryDoc that provide some *.cdoc*-files. So you can generate an

index with these examples and the documentation provided by PAKCS. The curryIndexer can either generate a new index or update the existing one. In order to distinguish between these options, you can use the flag $--n$ for generating a new index and $--u$ for updating the index.

```
$ curryIndexer ./example/CDOC_HTML
   http://www.informatik.uni-kiel.de/~pakcs/lib/CDOC/ --n
$ curryIndexer ./example/CDOC_URL
   http://www.informatik.uni-kiel.de/~pakcs/lib/CDOC/ --u
$ curryIndexer ./example/CDOC_XML
   http://www.informatik.uni-kiel.de/~pakcs/lib/CDOC/ --u
```

Since it can take a very long time to add each single module, you can pass a *.txt*-file as argument that consists of pairs of *.cdoc-file* and URI to generate the index.

```
$ curryIndexer ./example/test.txt --n
```

If you have generated an index, you can run the web application on your localhost on a given port with the curryServer.

```
$ curryServer -p 1337
```

C

User-Query Syntax

C.1 Extended Backur Naur-Form of the Parser

Query ::= { *Expr* } | *Expr Bool Expr* { *Bool Expr* } | "(" *Query* ")"

Expr ::= ":" *Specifier* | *Signature* | *String* | "(" *Expr* ")"

Bool ::= "AND" | "OR" | "NOT"

String ::= *IdentStart* { *IdentLetter* } | { *Operator* }

IdentStart ::= "_" | *lower*

IdentLetter ::= *alphaNum* | "'"

Operator ::= ":" | "!" | "#" | "$" | "%" | "&" | "*" | "+" | "."
| "/" | "<" | "/" | "<" | "=" | ">" | "?" | "@" | "\\"
| "^" | "|" | "-" | "~" | "_"

Signature ::= *ConstructorType* | *ConsArgumentType*

ConstructorType ::= *Identifier ConsArgumentType* | *Signature* "->" *Signature*

ConsArgumentType ::= *PrimitiveType* | *TypeVariable* | *ListType* | *TupleType*
| "(" *ConstructorType* ")" | "(" *ConsArgumentType* ")"

ListType ::= "[" *Signature* "]"

TupleType ::= "(" *Signature* "," *Signature* { "," *Signature* } ")"

PrimitiveType ::= *Identifier* | "()"

TypeVariable ::= *lowerCase*

Identifier ::= *Upper* { *alphaNum* }

Specifier ::= *SignatureSpecifier* | *ModuleSpecifier* | *FunctionSpecifier*
| *TypeSpecifier* | *AuthorSpecifier* | *InModuleSpecifier*
| *FlexibleSpecifier* | *RigidSpecifier*
| *NonDetSpecifier* | *DetSpecifier*

SignatureSpecifier ::= `"signature"` [*Signature*] | `"s"` [*Signature*]

ModuleSpecifier ::= `"module"` [*alphaNum*] | `"m"` [*alphaNum*]

FunctionSpecifier ::= `"function"` [*alphaNum*] | `"f"` [*alphaNum*]

TypeSpecifier ::= `"type"` [*alphaNum*] | `"t"` [*alphaNum*]

AuthorSpecifier ::= `"author"` [*alphaNum*] | `"a"` [*alphaNum*]

InModuleSpecifier ::= `"inModule"` [*alphaNum*] | `"in"` [*alphaNum*]

FlexibleSpecifier ::= `"flexible"` | `"fl"`

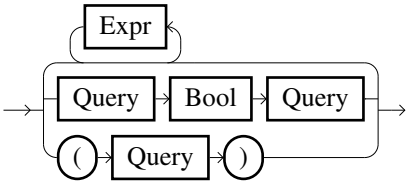
RigidSpecifier ::= `"rigid"` | `"r"`

NonDeterSpecifier ::= `"nondet"` | `"nd"`

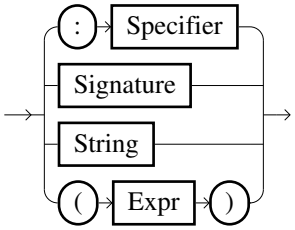
DetSpecifier ::= `"det"` | `"d"`

C.2 Syntax Diagrams

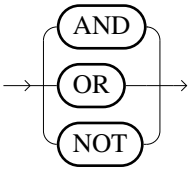
Query



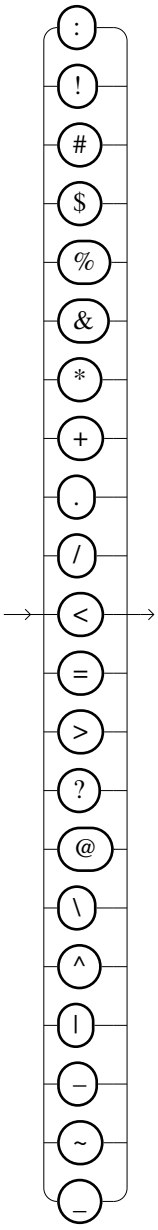
Expr



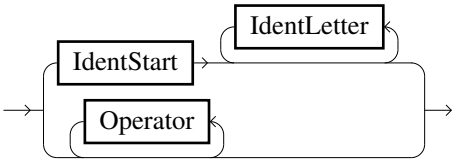
BoolOp



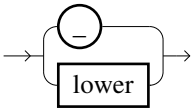
Operator



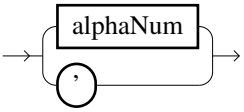
String



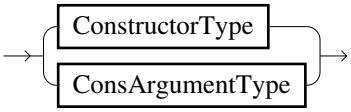
IdentStart



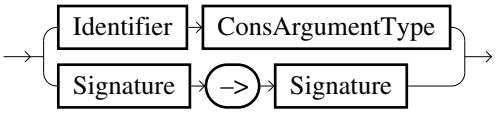
IdentLetter



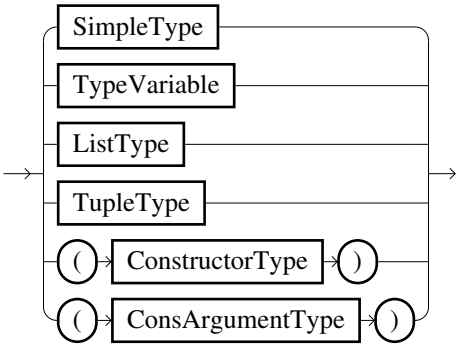
Signature



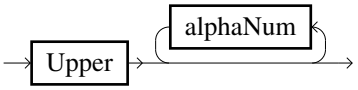
ConstructorType



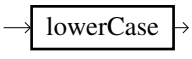
ConsArgumentType



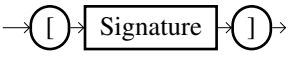
Identifier



TypeVariable

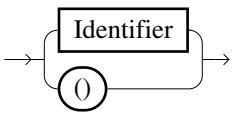


ListType

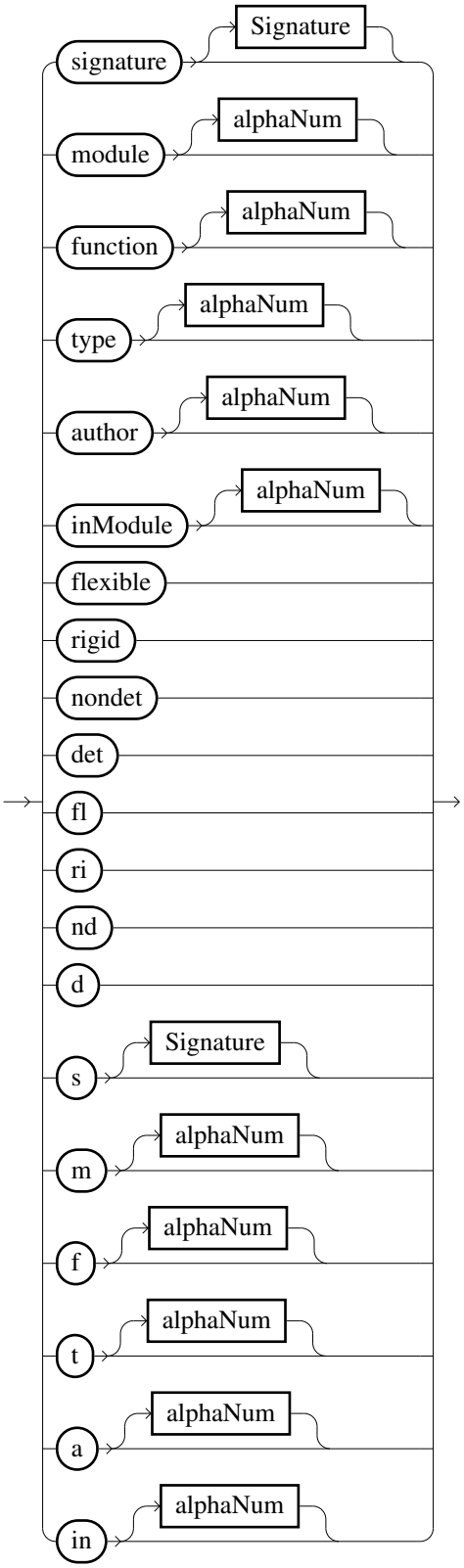


TupleType

SimpleType



Specifier


signature
Signature
module
alphaNum
function
alphaNum
type
alphaNum
author
alphaNum
inModule
alphaNum
flexible
rigid
nondet
det
fl
ri
nd
d
s
Signature
m
alphaNum
f
alphaNum
t
alphaNum
a
alphaNum
in
alphaNum

Bibliography

[1] Hayoo! Homepage. `http://holumbus.fh-wedel.de/hayoo/hayoo.html`.

[2] Jeroen Fokker. Functional parsers. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, London, UK, UK, 1995.

[3] M. Hanus. Currydoc: A documentation tool for declarative programs. In *Proc. of the 11th International Workshop on Functional and (Constraint) Logic Programming*, pages 225–228. Research Report, 2002.

[4] M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.8.2). Available at `http://www.curry-language.org`, 2006.

[5] Timo B. Hübel. The holumbus framework: Creating fast, flexible and highly customizable search engines with haskell. Master's thesis, FH Wedel University of Applied Sciences, 2008.

[6] Graham Hutton and Erik Meijer. Monadic Parsing in Haskell. *Journal of Functional Programming*, pages 437–444, 1998.

[7] Simon L. Peyton Jones, editor. *Haskell 98 Language and Libraries—The Revised Report*. Cambridge University Press, 2003.

[8] Sebastian M. Schlatt. The holumbus framework: Creating scalable and highly customized crawlers and indexers. Master's thesis, FH Wedel University of Applied Sciences, 2008.

[9] Philip Wadler. How to replace failure by a list of successes. In *Proc. of a conference on Functional programming languages and computer architecture*, pages 113–128, New York, NY, USA, 1985.