UNIVERSITÄT
BERN

Dicto Auto-Complete Engine

a back end for autocompletion

Bachelor Thesis

Radunz, Kenneth
from
3052 Zollikofen BE, Switzerland

Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

2016

Prof. Dr. Oscar Nierstrasz
Andrea Caracciolo

Software Composition Group
Institut für Informatik und angewandte Mathematik
University of Bern, Switzerland

Abstract

Most programming languages have an IDE that supports autocompletion. This feature increases productivity and makes programming an easier and smoother experience. The goal of this project is to add autocompletion to the Dicto language. To ensure reusability, it is implemented in a frontend-backend approach. From this it follows that any new front end only needs to communicate to an already existing back end, which is responsible for the logic. Performance concerns are addressed in the later parts of the thesis.

Contents

1 Introduction

Dicto[6] is a small Domain Specific Language developed at the University of Berne. Its purpose is to provide a simple tool to specify constraints for software projects and automatically check that the implementation does not violate those formulated constraints. As Junit allows its user to ensure the functionality of Java classes, Dicto enables the user to ensure that his project is built according to specifications at the highest level of abstraction. For example, the user wants to ensure that components have a certain dependency structure: When implementing the MVC[1] pattern the model should not depend on the controller.

Most common programming languages are supported by one or more integrated development environments. These IDEs facilitate the work of programmers by bundling many powerful tools with a source code editor. Common features include refactoring tools that can extract methods, rename a variable everywhere it is used, and many other editing features. In addition, IDEs have intelligent code completion, which can be seen as one of the most prominent features. Intelligent code completion facilitates programming by reducing the number of common mistakes the programmer makes, as well as, looking up resources the programmer might need and otherwise has to search for manually. This information is commonly provided using drop downs. For this purpose, this component of the source editor needs to be context aware. This means that it needs to track all methods and variables currently available. In addition, it needs to be aware of the syntax structures that are defined by the programming language chosen by the user.

The task proposed was to provide code completion for the Dicto language. This means not only providing the user with language and statement specific keywords, but also contributing necessary semantic information like variables and variable types, when the user might need it. A short overview of the Dicto language can be found in chapter 2. In addition, the auto completion facility should be reusable, so it could be used as a common back end to different Dicto implemenations.

The solution provides means to model the semantics and syntax of a regular language. In addition, a state machine is provided that uses the specific model to process input and put together useful information that might be needed by the programmer. Also, a web service is provided to give easy access to the state machine. Further information on how the solution works on a conceptual level can be found in

[1] `https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller`

chapter 3. A closer look at the implemented solution is available in chapter 7.

To show that the provided solution is actually usable, load testing needed to be done. The reviewed scenarios involved the comparison of the server and client running on the same machine versus separating both using a local network. The second variable examined was the number of variable and variable. In this case, two different tests were set up, one having a rather small environment only including a few variables and variable types. The second test was run using thousands of types and variables. The test results indicate that of both factors, network and environment, the former has most impact on performance of the implemented solution. The implementation can handle huge input sizes with absurd environments without much time loss. In contrast, using a network decreases the amount of data, the service might process until a delay is experienced, which makes the user lose focus, by a lot. Nevertheless, the tests indicate that the service can process input sizes that are imaginable for a real application of the Dicto language, in a timely fashion.

2 Related Work

2.1 Dicto

Before diving into the details of autocompletion, a small introduction to the language Dicto is required. The Dicto language consists of only two statements: entity and rule.

2.1.1 Entity Statement

The purpose of the entity statement is to define a variable. An example for such a statement would be:

```
Model = Package with name: "org.app.model"
```

The first part of the statement is the variable name. It must satisfy certain restrictions, like variable names in most programming languages, but apart from that it can be chosen freely. The variable name is then separated from the description of its content by the equal sign. The first word after the equal sign specifies the variable type. Possible types could be “Package”, “Class”, or “Website”. Usable Variable types are defined by the implemenation of the dicto engine.
This is then followed by the keyword “with”. Afterwards, the last part of the entity statement consist of a comma separated list of attributes. Each attribute is specified by its name and then followed by a colon and the value given to the attribute. Possible values at the moment are either Strings or Integers. The user cannot choose the attribute names freely. The chosen variable type dictates the available attributes. Not all attributes need to be specified.

2.1.2 Rule Statement

The rule statement is then used to specify behaviour and interactions of the previously defined variables.

```
Model cannot depend on Test
```

Above is the example for a simple rule statement. It consists of four components:

1. subject “Model”

2. mode “cannot”
3. predicate “depend on”
4. object “Test”

The user wants to ensure that his model package does not use anything from the Test Package. The mode and predicate together determine the relation between subject and object. All modes can be used together with any predicate. In addition, modes do not change between engines. They are fixed on a syntactical level. Possible modes are:

1. must
2. cannot
3. can only
4. only can

The last option changes the syntax a bit, because the keyword “only” appears before the subject. For example:

```
only Controller can depend on Model
```

Predicates are specified on a semantic level and depend on the current Dicto engine in use. It is possible to specify more than one subject or object. For example:

```
Model, Controller cannot depend on Test
```

It must be ensured that all subjects of the same statement are of the same type. The same goes for objects. The reason for that is that multiple subjects or objects in one statements is simply a shortened version of the same statement repeated with different subjects (objects). The above statement is equivalent to:

```
Model cannot depend on Test
Controller cannot depend on Test
```

2.2 PetitParser

PetitParser[5] is a scannerless parser library. It is available for Smalltalk, Java, and Dart. In the early stages of exploring the problem, a solution based on the Java version of PetitParser was implemented. This attempt was aborted, because of the library’s exception handling. A parser that would take a keyword would raise an exception if it encountered only half the keyword. This is the desired behaviour in most scenarios. Autocompletion is no such scenario. If half a keyword is encountered, we might want to send a list of suggestions to the user instead of returning an error message. For this reason it was decided to create a new parser library that implemented the desired behaviour naturally instead of hacking it into PetitParser. For further explanation behind this decision read section 3.2.1.

2.3 Ace

Ace[2] is a Javascript framework for web embedded code editors. It supports a variety of programming languages by default. In addition, one can add one’s own language using the framework. It provides many features and is highly customizable. It can be used to send Ajax requests to a back end. For this reason the framework got used for the front end demo.

2.4 Microsoft IntelliSense

This is a feature that handles a lot of convenient tools included in Microsoft's IDEs like Visual Studio. One of the described features is called *List Members*: "A list of valid members from a type (or namespace) appears after you type a trigger character (for example, a period (.) in managed code or :: in C++). If you continue typing characters, the list is filtered to include only the members that begin with those characters." [1]. This feature is similar to this bachelor project, but is more complicated for the following reasons: The languages are more complex and a lot of resources must be scanned and included like other source files or libraries.

3 Architecture and Design

In this chapter the project's structure will be discussed. The first section covers a simple overview on the topmost layer. Then the design of the state machine and its components will be explained in detail, because it contains the most logic. Last, the specific structure of the state machine that was created to process Dicto will be discussed step by step in section 3.3.

3.1 Deployment Architecture

In this section, the complete architecture of the solution is described. The separation of client and server is explained.

3.1.1 Overview

Figure 3.1 displays a general overview on how the complete system was designed. The main idea is that the user sends some form of request containing the user's input. The server then passes the information to the state machine. After the state machine processes the input and generates a result, it is sent back to the client.

3.1.2 Interface

For maximum reusability an HTTP based communication format was chosen. HTTP has good support among most programming languages and a client that sends and receives HTTP messages can be created easily in most environments.
The client sends an HTTP POST request to the url of the server. The message body contains the complete text the user has written thus far. A POST-request was chosen over the GET-request, because of the restriction most HTTP servers have regarding the size of a GET-request. Since, the server demands the full input, it might exceed those limits. Using POST-requests to submit bigger amounts of data is a common way of doing so.
As response the client will receive an JSON-Array. The JSON format was chosen due to its high popularity and its native support in most programming languages. This kind of response format can be adapted easily

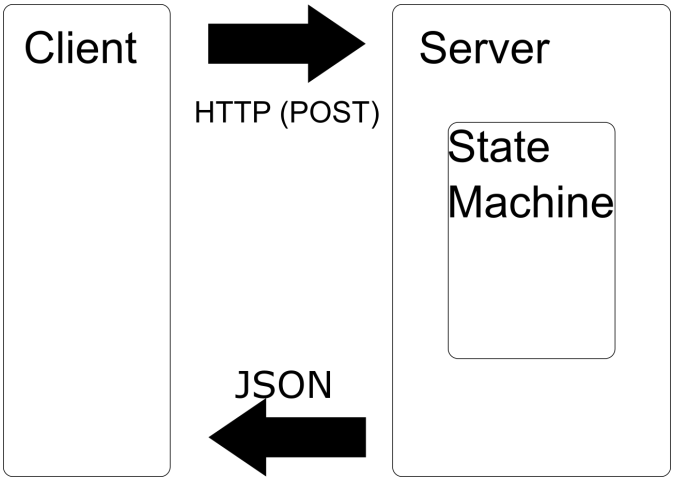

Figure 3.1: An overview of the chosen architecture.

to transmit more and complex types of data. More on how the response can be extended can be read in chapter 5.

3.1.3 Stateless Service

The most important design choice was whether to make the server stateless or state-aware.

Stateless The server is not aware of the clients state and does not store information regarding it. This means that every request to the server can be processed in isolation.

State-aware The server caches the user input and communicates with the client in a incremental fashion. This means the client does not need to send the complete input of the user with each request, but only needs to update the cached version of it. This for example could mean more specifically that the request only sends the information which lines were changed since the last request.

In the end, the server was designed in a stateless way. This was mainly to increase simplicity of both client and server. Specifically, neither client nor server need to keep track of the other side's state. The client does not need to store a version of each request to determine the difference to the current state when it needs to send a new request. Accordingly, the server does not need to cache the request either, to determine the current state of the user's input. This means both client and server are much more simple and straightforward to develop. This design choice comes with a downside. Each time client and server communicate, the client needs to submit its complete state to the server. This can mean sending big amounts of text, each time the user requests the autocompletion feature.

3.2 State Machine Functionality and Components

In this section the state machine's functionality and components are explained. First a new kind of regular expressions parsing library is introduced. Next the transition logic is explained and the needed components are described.

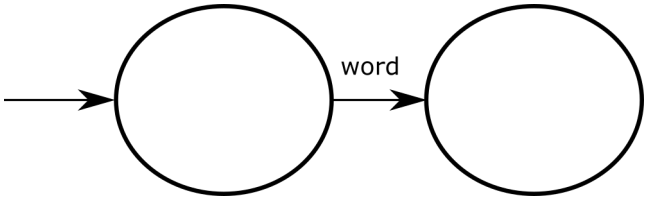

Figure 3.2: A simple state machine.

3.2.1 Tri-valued Acceptors

Consider the example in the figure 3.2. It shows a very basic state machine with only two states. In between is a transition that accepts “word”. Now let’s examine this example under the scenario of autocompletion. Table 3.1 shows a few inputs and the corresponding results we want to achieve.

input	autocompletion suggestions
“”	[“word”]
“wo”	[“word”]
“word”	[]
“number”	PARSING ERROR

Table 3.1: A simple list of inputs and expected results.

We assume that the suggestions we can make correlate to the transitions of the state in which the input terminates. So if the input terminates at the left state, one transition can be looked at and included in our autocompletion suggestions. From this follows that we suggest simply “word”. This seems to be correct for the third input “word”: The state machine transitions from the left to the right state and terminates. Since there is no transition originating from the right state, nothing can be suggested. The fourth input is easily explained as well. The user enters something that the left state cannot assign to any of its transitions and thus raises an error. The interesting situation is when the user types “wo” and then requests autocompletion. The left state again would try to assign the input to its transitions and fail to do so, thus raise another error. But this is not the outcome we want. The state machine is expected to terminate in the left state and return the same suggestion as our first input in the table. A suggested workaround was to create a transition for each character. The solution would look like the state machine in figure 3.3.

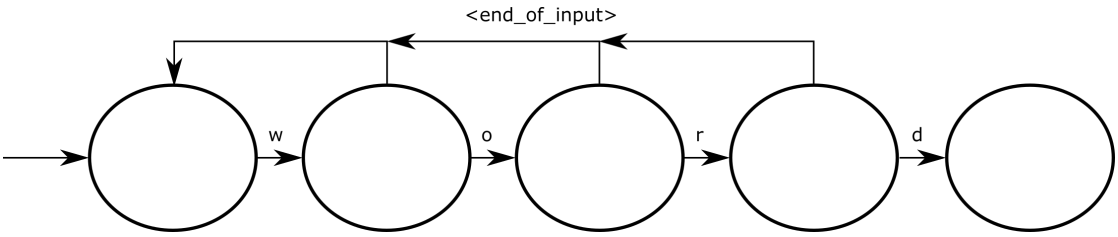

Figure 3.3: Visualization of the suggested workaround.

This seems like a good solution for an example this simple. But regarding a full language specification the amounts of additional states and transition needed to be added raise performance and readability issues. In addition, transitions like “[a-zA-Z][0-9a-zA-Z]*” need to be handled as well. Regarding all those concerns a better and more intuitive solution needs to be devised.

The actual solution is inspired by the multi-valued propositional logic or Lukasiewicz Logic. This logic contains more than the two common truth values: false and true. In addition, it has truth values in between the mentioned two. Specifically, I worked with a three valued version containing 0, $\frac{1}{2}$, 1. An acceptor library was modelled with three different outcomes in mind:

Failure If an acceptor encounters a character that can not be accepted by the state machine.

Partial If an acceptor encounters the end of input before completed.

Success If an acceptor successfully parses the input.

It can be noted that the Success outcome of the new acceptor library corresponds to any standard regex library accepting an input. This means, if the new acceptor accepts an input, the correspondent acceptor in a standard regex library would do so as well and vice-versa. Table 3.2 visualizes how the acceptor outcome is transformed into the state machine outcome. The new acceptor library has functionality similar

input	acceptor result	state machine result	autocompletion suggestions
“”	partial	successful termination at the left state	[“word”]
“wo”	partial	successful termination at the left state	[“word”]
“word”	success	successful termination at the right state	[]
“number”	failure	syntax error	NA

Table 3.2: An extended list of inputs and its outcomes.

to standard regular expression, including:

optional usually noted as “?”. E.g.: a?

or usually noted as “|”. E.g.: gray|black

class usually surrounded with “[” and “]”. E.g.: [a-zA-Z]

one-or-more usually noted as “+”. E.g.: a+

The *or* modifier works very straightforwardly. It tries to match all concatenated sub parsers. If all fail, it fails as well. If one sub parser results in incomplete, the *or* parser returns incomplete. If a sub parser succeeds, so does the *or* parser.

input	standard regex	tri-valued regex
gray	success	success
black	success	success
gr	failure	incomplete
bl	failure	incomplete
ϵ	failure	incomplete
brown	failure	failure
br	failure	failure

Table 3.3: Comparing standard regex and tri-valued regex with the pattern “(gray|black)”

The table 3.3 compares the results of the pattern (gray|black) of a standard library with the tri-valued version. Everytime the standard library successfully matches an input with the pattern, the tri-valued

version does the same. Only if the standard library is not able to match the input with the pattern, the tri-valued version might return a different result. If the given input partly matches any sub pattern, the result of the whole pattern changes from “failure” to “incomplete”. In the example, “bl” can still be completed to “black” and “gr” can still be completed to “gray”. Instead of that, “br” does not show the same property and thus also the tri-valued library returns “failure” for this input.
Classes work similar to the *or* modifier. A *class* matches a single character that depends on the specification inside the brackets. For example, “[abc]” matches either “a”, “b”, or “c”. The class can be seen as a special *or* pattern that only accepts single characters. That means that the result “incomplete” is only possible if the input is empty. Any character not contained in the set will cause the pattern to return “failure”. Negated *classes* are also supported by the tri-valued regex. “[^abc]” accepts any characters, but “a”, “b”, or “c”. Table 3.4 summarizes the difference between a class pattern of a standard regex and a tri-valued regex.

input	standard regex	tri-valued regex
b	success	success
d	failure	failure
ϵ	failure	incomplete

Table 3.4: Comparing standard regex and tri-valued regex with the pattern “[abc]”

input	standard regex	tri-valued regex
ϵ	failure	incomplete
a	failure	incomplete
abc	success	success
abca	failure	incomplete
abcabc	success	success
dbc	failure	failure
abd	failure	failure

Table 3.5: Comparing standard regex and tri-valued regex with the pattern “(abc)+”

The *one-or-more* modifier takes a parser and parses it as many times as possible, but at least one time. It returns “success” if the given sub parser parses the input completely any number of times. “incomplete” is returned, if the last iteration of the sub parser incompletely matches the input. The parser fails if the sub parser, fails on the first iteration. Table 3.5 visualizes possible inputs and their results for the pattern “(abc)+”. As stated, the new implementation only fails if the sub parser, in this case “(abc)” fails on the first try.
As the new regex library is used in a state machine, it is rather configured to support partial matches instead of matching the whole input. This means any parser that returns “success”, specifies a range started at the beginning of the input that it succeeded to parse. For example, the pattern “(abc)+” combined with the input “abcabcbanana” returns that it successfully matched the first six characters opposed to the situation that it failed to match the whole input. This is important as the state machine uses a multitude of acceptors to switch states and those are only needed to match parts of the whole input, to let the state machine switch states.

3.2.2 The Transition

There are four specific cases the state machine needs to account for.

1. Syntax Error
2. Incomplete input
3. Semantic Error
4. Complete input

In the first and third scenario, the state machine will not return a list of suggestions for autocompletion, but a meaningful description where the error is and why it is an error. Consider the example state machine in figure 3.4. Starting with the leftmost state, there are two transitions originating from it. The top one

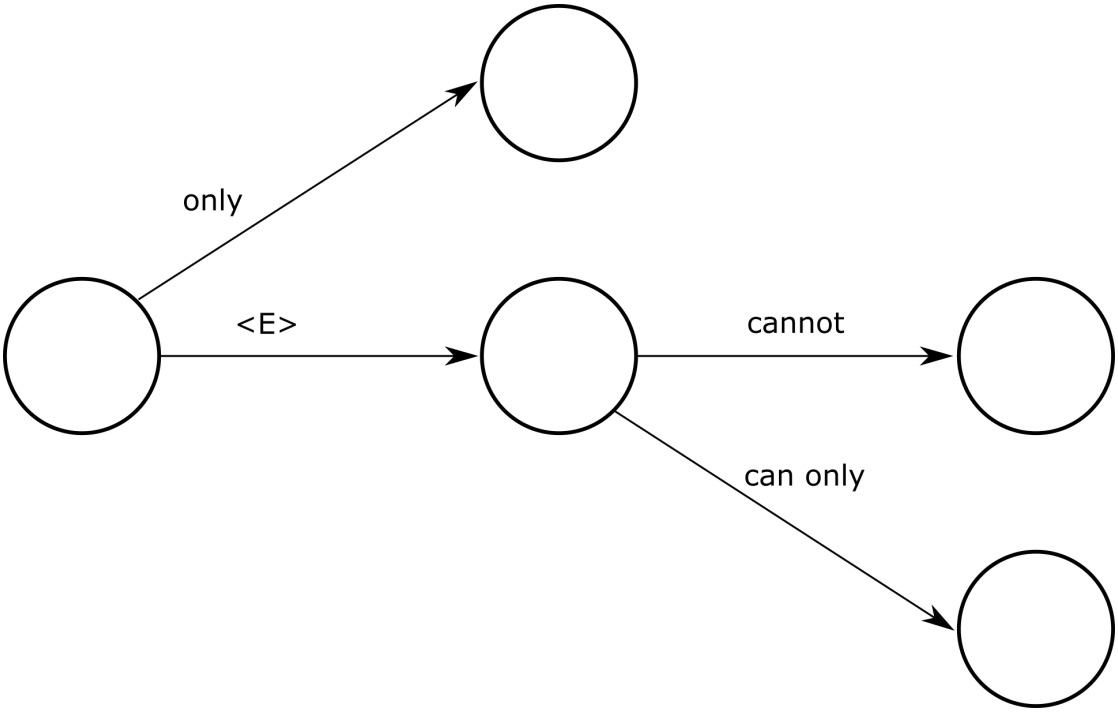

Figure 3.4: An example state machine to illustrate the 4 transition scenarios

exclusively accepts the token “only” as it expects a keyword. The bottom transition accepts an entity, which we assume are already defined variables. If the user types a known variable, he or she can either follow it with “cannot” or “can only”. Considering figure 3.4, the user might type

```
View can only
```

to satisfy the illustrated state machine. Where the first word “View” would satisfy the acceptor of the first bottom transition, and the “can only” would be accepted by the bottom transition of the following state. The first scenario happens, if the state machine is in a state where all transitions fail to accept the current input. For the state machine in 3.4, such a possible input would be “?”, since it is not matched by “only” or the entity, which only matches a combination of letters and digits.
Scenario 2 happens when the remaining input does not satisfy any transition originating from the active state. In most cases the state machine will then raise a syntax error. Consider the input “on”: The point can be made that the acceptor of the top transition will not accept this input and an attempt will be made to

match the second transition. But as we already discussed in section 3.2.1, certain inputs can be completed, with potential input, to actually match Dicto's syntax. So the input "on" should satisfy the top acceptor partially. The result of this scenario should be that the state machine does not switch to the destination state, but stays in the current leftmost state.
A semantic error is raised if the user types something that looks like a variable, but is not defined. If "Window" is typed, but the only previously defined variables are "Controller", "Model", and "View", the second branch will accept the input, as it is syntactically correct, but then will raise a semantic error, when not finding an existing variable with the given name. The last case is very simple. The user types something that satisfies a transition and then no input is remaining. The user could type "only" and the top transition will accept the input, thus the state machine switches the active branch to the destination of said transition.

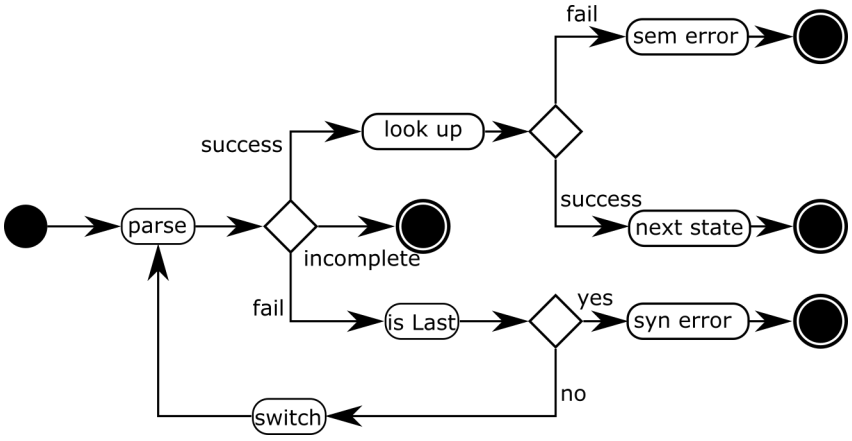

Figure 3.5: How the active state processes the remaining input.

Figure 3.5 visualizes the process how all four described scenarios are incorporated into the state machine logic. The chart illustrates how the input is processed and under which conditions an error is generated.

Figure 3.6 visualizes how an input like "?" is handled and what result it yields, when giving the input to the state machine in figure 3.4. The left half visualizes the decision process involving the top transition, which is checked first. The right half shows what happens when trying the bottom transition. The "parse"-block will try to match the acceptor of the first transition "only" with the input "?", but it fails. The "is last" then returns true if this was the last transition, originating from the active branch that was not tried. In this case, there is still one more transition that can be tried out. But the bottom transition does not accept the input "?" either. In addition, there is no other transition that can be tried. Thus, the process will reach the "syntax error"-block. This terminates the state machine and returns an syntax error that an unexpected character is encountered.
The input "on" covers the second scenario. When trying to match the input with the top acceptor that accepts "only", it will result in a "incomplete" result. This means the input does not satisfy the acceptor, but might match it, if more input was available. Figure 3.7 shows how that is handled. The first condition can split between three values: "success", "failure", and "incomplete", which is further explained in section 3.2.1. The result is that the state machine does not switch to the destination state of the given transition, but terminates at the current state.

The third scenario does not depend on the input entirely, but also on the semantic environment of the state machine, as well as the specific look up that each state performs. For scenario 3, it is assumed that the user input is "Tests" that there is no variable already defined with that name, and that the bottom

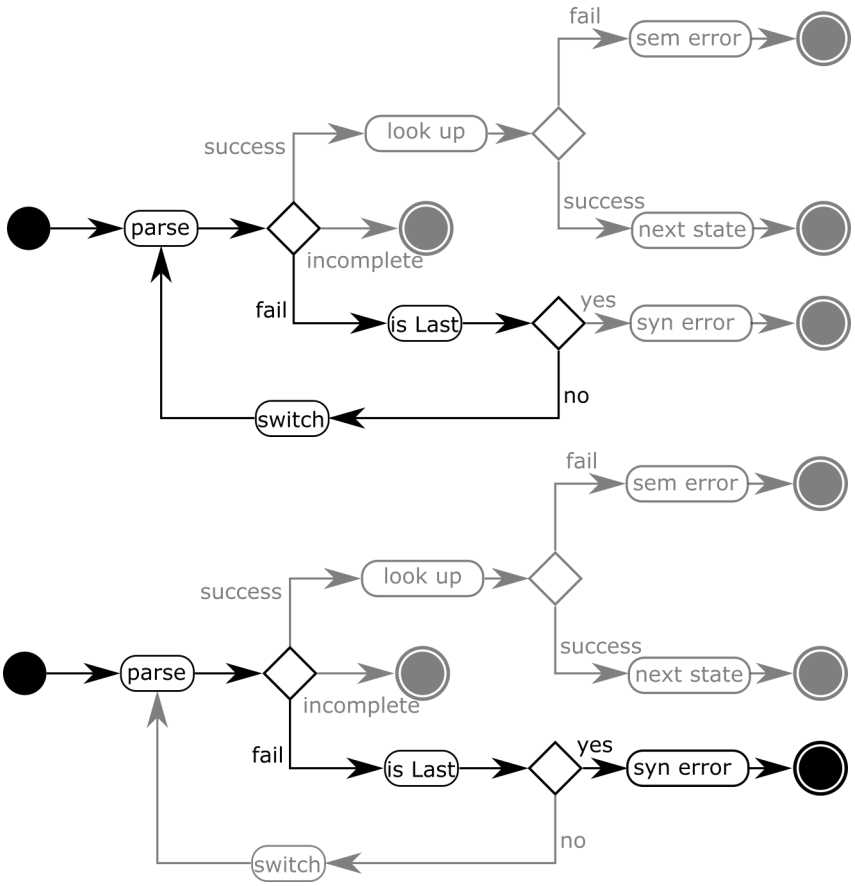

Figure 3.6: How scenario 1 is handled by the state machine.

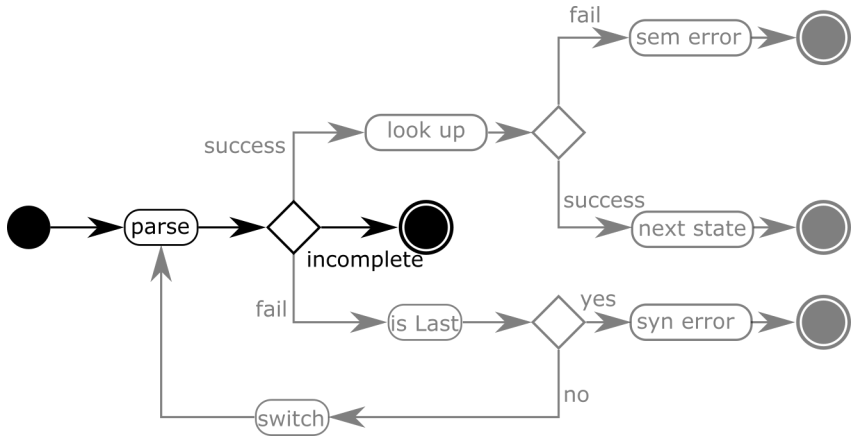

Figure 3.7: How scenario 2 is handled by the state machine.

branch of the current state matches the input against already defined variables. Given the input “Tests” the

first parser fails and switches to the next transition as seen in the left half of figure 3.6. Then, the bottom acceptor matches the input and the process continues to perform a semantic lookup. This post-processing of the transition tries to find a variable with the given name, but fails. Thus, the lookup fails and the state machine terminates while returning a semantic error: “Unknown variable: Tests”

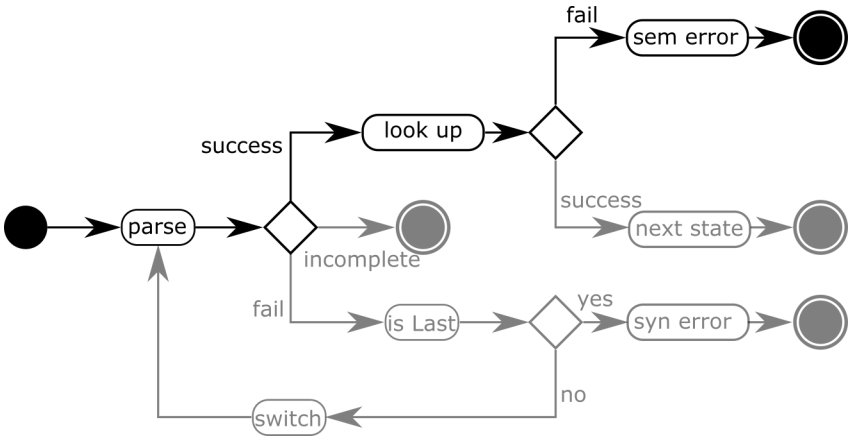

Figure 3.8: How scenario 3 is handled by the state machine.

The last scenario is similar to the third one. It simply differs in the result of the lookup. Consider the same semantic environment, as in the scenario before, but the input is changed to “Controller”. When the input is given to the state machine, the process takes a very similar flow as described in the previous example. The only difference is that the specific semantic lookup connected to the transition, can only match the input with a variable that was already specified. Thus, it does not fail and no semantic error is generated. Instead, the state machine switches its active state to the destination of the transition.

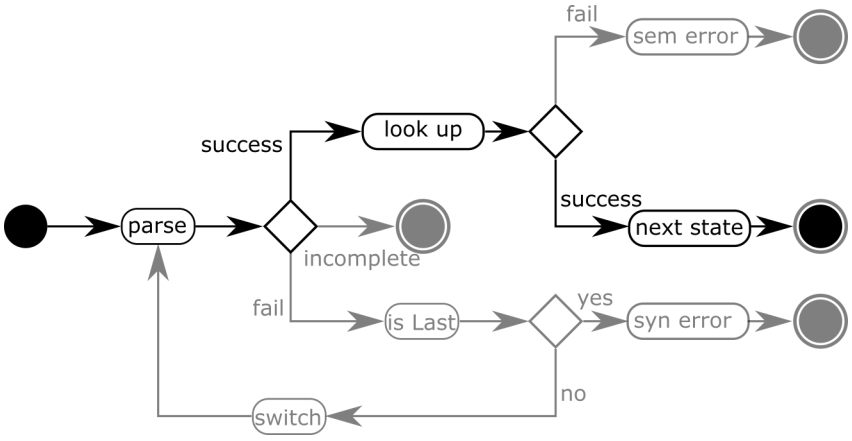

Figure 3.9: How scenario 4 is handled by the state machine.

3.2.3 Transition Post Processor

As already explained in 3.2.2 a semantic look-up is needed to ensure proper functionality. This is added to the state machine in the form of a post process that is added to each transition individually. This is a

simple routine that is executed if a transition's acceptor matches the input of the state machine successfully. The post process has access to the semantic environment, as well as the input parsed by the transition's acceptor. The post process cannot only access the environment, but also modify it, and also change the flow of the state machine. This means that it can cause the state machine to abort and return an error instead of switching the current state to the destination of the transition, which would happen usually if the acceptor matches the input. Examples can be seen in section 3.3 where the configuration of the state machine for Dicto is explained in more detail.

3.2.4 Suggestors

This is the component relevant for the core feature of the state machine: Generating suggestions. This means the state machine returns a set of possible completions for the end of the given input, which is assumed to be the cursor's position. The suggesting part of the solution executes after the state machine has processed the input and terminated at a specific state, without an error occurring. This state and its transition define the suggestions returned by the state machine. For this reason each transition has another component attached, which is referred to as "suggestor" for the rest of the thesis. The function of the suggestor is to generate a set of possible satisfying inputs for the given transition. The state in which the state machine terminated, combines the sets of its originating transition to a final set. Then, this set is returned to the user. The suggestor can be static having its result already specified, when the state machine is created. The other possibility is that the suggestor depends on the semantic environment and generates it's set after the state machine has terminated.

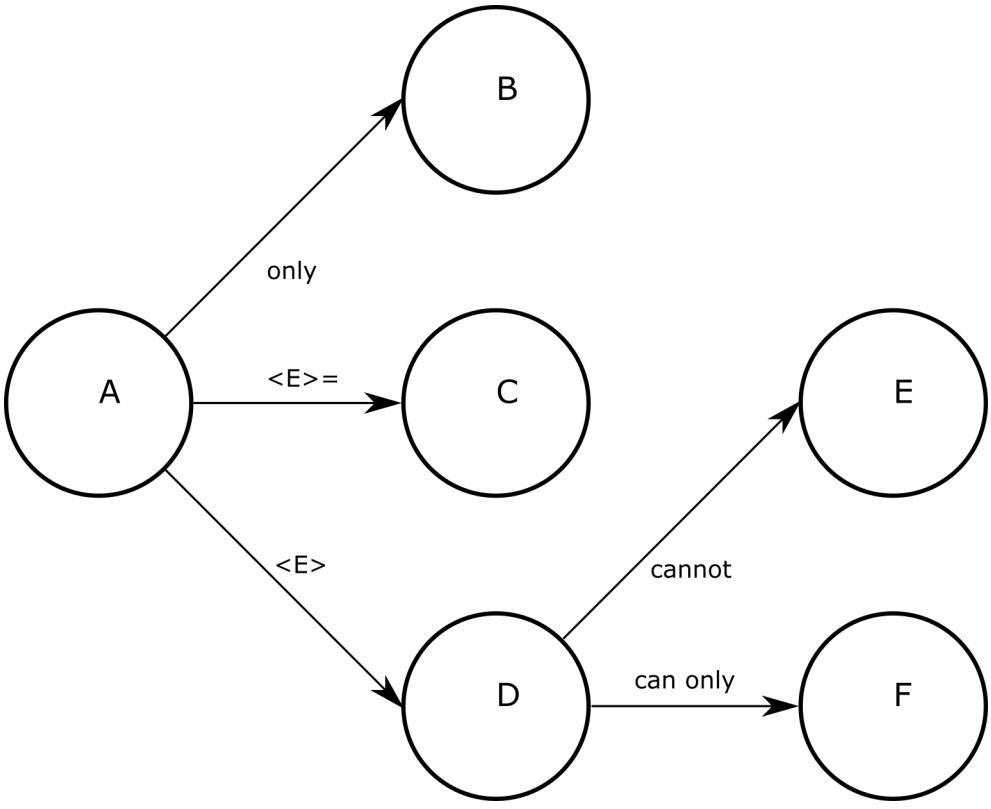

Figure 3.10: An example state machine.

Figure 3.10 represents a state machine that already terminated in state A. There are three transitions originating from the marked state. The first accepts the keyword only. The second accepts an entity followed by a equal sign. On a semantic level, we assume that the transition expects a variable name that is not defined yet. The third transition accepts an entity. Semantically, we assume a name of an already defined variable is expected. To return a meaningful set of suggestions, the state will receive a set from each of the three transitions and then merge the results into one set. This set is returned to the user.

In our example, the top transition's suggestor returns a set, which contains the single element "only". As the transition simply matches a keyword, the suggestor is static. There is no semantic information needed for this suggestor. The second transition matches undefined variable names. This means that there are certain possibilities that we can exclude from the set, namely all defined variable's names. But guessing possible variable names is difficult or nearly impossible. For the simple purpose of this example, the second transition returns an empty set. The last transition matches known variables. This is information stored in the environment of the state machine. Thus, the suggestor can access this information and assemble a set of variable names, as the origin state requests it. Lastly, the state combines these three sets to a single one, containing the keyword "only" and all variable names of already defined variables. A possible outcome of this example can be seen in 3.11.

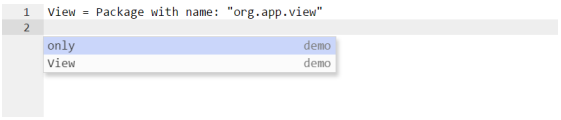

Figure 3.11: A possible outcome of the above example.

3.2.5 Environment

For the process described in 3.2.2 to function, an environment needs to be added to the state machine. The responsibility of this component is to provide semantic information. In the case of Dicto, this involves variable types and their arguments, predicates and the variable types they work with, as well as variables including their types and attributes. In addition, the environment contains a small key-value map that stores Strings. This cache lets post processes and store and read data, to be used by other states and transitions. An example explains the use of the cache in the environment further

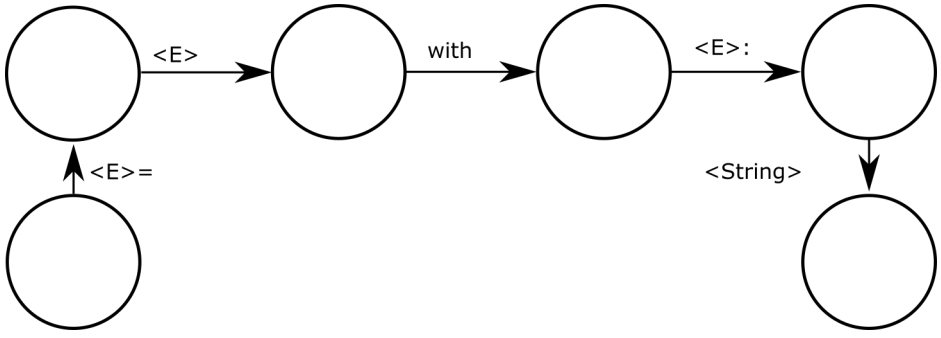

Figure 3.12: An example state machine.

In figure 3.12 a state machine is shown that maps the variable definition statement of Dicto.

```
Controller = Package with name: "org.app.controller"
```

The first transition maps the variable name “Controller” and then stores it in the cache of the environment. The same happens for the variable type “Package”. This continues until the end of the statement. Then all the information is read from the cache and assembled into a variable that then is inserted back into the environment.

3.3 Dicto Specific State Machine

In subsection 3.2 the functionality and components needed for autocompletion are explained. Next, components are assembled to build a state machine that can generate suggestions for the Dicto language. In this chapter it is explained which states and transitions are assembled to the final state machine. For better understandability, this process is separated into two parts, covering the two statements entity and rule separately.

3.3.1 Entity

In this statement a new variable is defined. An example for this kind of statement would be:

```
Model = Package with name: "org.app.model"
```

A variable consists of 3 main parts.

name a unique name given to the variable.

type a variable type that can be chosen out of a known set of existing variable types defined at the construction of the state machine.

attributes a set of key-value pairs. Possible attribute names depend on the variable type.

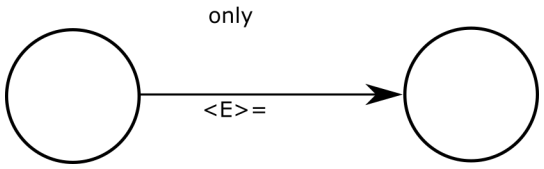

Figure 3.13: The first part of the entity statement mapping the variable name.

Figure 3.13 shows the first transition of this part of the state machine. It matches an entity, namely the variable name, followed by a equal sign as required by the Dicto syntax. The post processor of this transition needs to do two different steps.

1. The post processor needs to check if there is already a variable with the specified name defined. If this is the case, a semantic error must be raised. This ensures that all variables have unique names.

2. The post processor must store the input that is matches with the entity in the cache of the environment. Later, when the statement is complete this information is accessed to assemble the new variable and insert it in the environment.

The suggestor for this transition returns an empty set. As the new variable name is not known to the environment and there is no way to reliably guess it, nothing can be suggested.

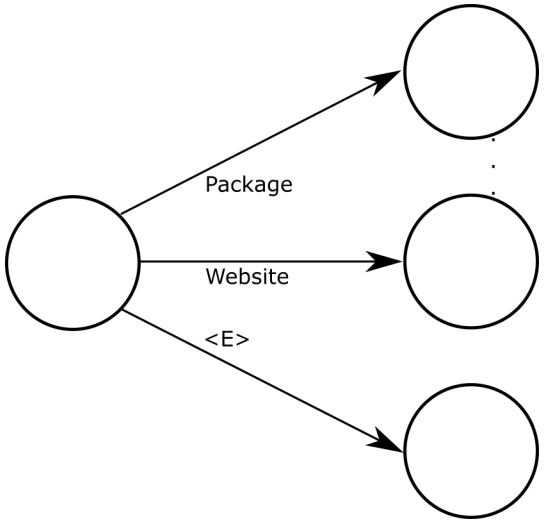

Figure 3.14: The second part of the entity statement mapping the variable name.

Figure 3.14 covers the second part of the statement. Here the user can choose between available variable types. Each variable type gets its own state and transition. These can differ from instance to instance, but remain the same through the life cycle of the state machine. All these transitions have simple post processors and suggestors. Each post processor stores the variable type in the cache of the environment for the same reason as the variable name was already stored. The suggestor for those transitions simply returns a set, only containing the variable type's name as single element. There needs to be a last transition added. This transition is last in the list of transitions and again accepts an entity. The only reason for this transition is to raise an semantic error, if the user uses a unknown variable type. Assume that there are only two variable type: "Package" and "Class". Now without this last transition, if the user types:

```
Controller = Method
```

The state machine would raise an syntax error: "unexpected character: M at index 14." If we now add this transition and let its post processor unconditionally raise an semantic error. The result from the input would be: "unknown variable type: Method", which is the wanted outcome in this scenario.

Following this up is the keyword "with". The corresponding transition can be seen in figure 3.15.

This transition does not need post processing because this part of the state machine is completely syntax dependent. In addition, this means that the suggestor is as well of static nature. It returns a single element set containing only the keyword itself "with". The last part of the statement consists of the attributes of the variable, which are defined in a list using key value pairs separated by commas.

In figure 3.16 can be seen that all attributes are separated using a comma. Attribute name and value themselves are separated using a colon. The only post processor and suggestor worth mentioning belong

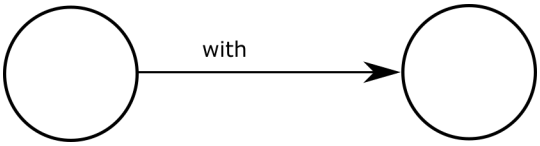

Figure 3.15: The third part of the entity statement mapping the keyword “with”.

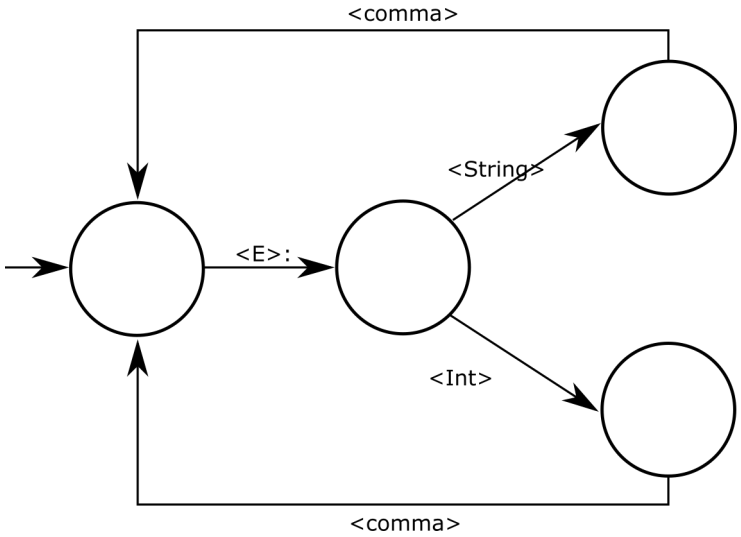

Figure 3.16: The fourth part of the entity statement mapping the variable’s attributes.

to the transition mapping the attribute’s name. The post processor looks up the variable type already stored in the cache and its related attributes. Afterwards it checks if the attributes used in the statement match the ones of the type in use. The corresponding suggestor works similar to the post process, as it simply returns a list of possible attribute names, associated with the already specified variable type, which is accessed the same way as in the post process.

The last transition that needs mentioning is the transition that leads back to the first state. The user finishes the entity statement and then switches to the next line to begin a new statement. This transition that accepts a new line, is very important. Its post process accesses all already stored information in the environment’s cache, which are variable name and variable type. Then it assembles these pieces and inserts it into the environment’s set of defined variables. Afterwards, it cleans the cache so that it can be used again by the next statement’s post processes.

3.3.2 Rule

The rule is the second statement one can write in the Dicto Language. An example is:

```
Controller cannot depend on Model
```

This form of statement consists of 4 parts

1. subject(s)

2. mode
3. predicate
4. object(s)

A rule can contain one or more subjects. If using multiple subjects, it must be ensured that all of them are of the same type. Figure 3.17 shows how the state machine accepts the statement's subjects. The first subject, the user types, is accepted by the left-to-right arrow. It's post process is pretty straightforward: It looks the given subject up in the list of defined variables. If the variable is not in the environment, a semantic error is generated and the state machine terminates. Otherwise, the variable is stored in the environment's cache for later usage. The suggestor of this transition simply assembles a list of all variables that were defined.

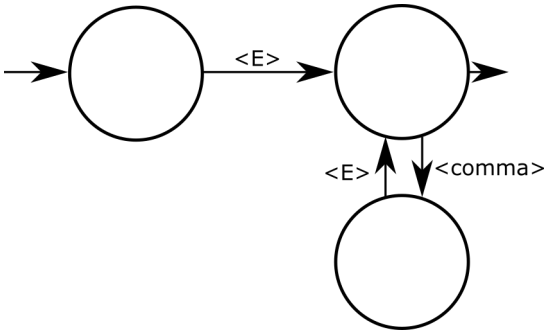

Figure 3.17: The part of the state machine that maps the rule's subjects

The transition downwards accepts a comma. There is no need to add any post process or suggestor. The transition upwards accepts again a subject. The transition differs from the one that was explained first, because it must ensure that its subject's type is identical to the type of the first subject. This means that its post process does everything the first transition's post process does, but additionally accesses the cache to look up the first subject and compare the type. If the comparison fails, again a semantic error is raised and the state machine is terminated. The suggestor of the transition going upwards again returns a set of all defined variables in the environment, but first filters them. It removes all variables from the set that do not share the type with the first subject.
After the user has specified all his subjects, he continues to specify the mode he wants to use. At this point three different modes are available. The related state machine diagram for this part of the statement can be seen in figure 3.18.

These transitions all have similar post processes and suggestors. The post process simply stores the chosen mode in the environment's cache. The suggestor returns a set with the specific mode as sole element. From here the user continues specifying the predicate. The predicates are predefined by the environment and never change as long as the state machine exists. Accepting predicates need to be handled different than accepting entities or keywords. A predicate can consist of multiple, whitespace separated words including special characates as ">". A few examples of predicates are:

```
depend on
contain text
have latency <
```

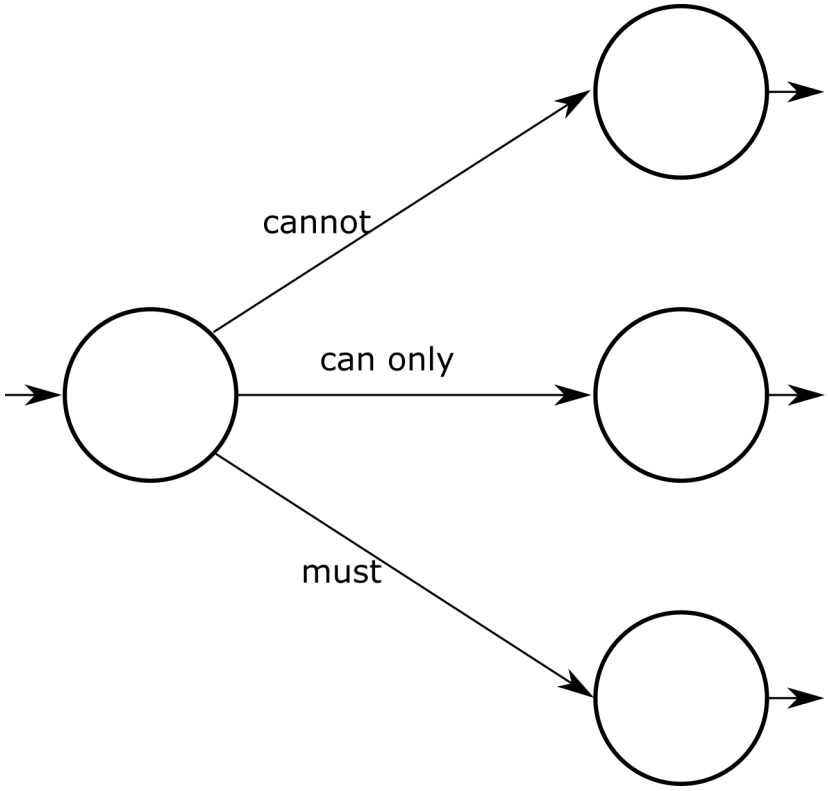

Figure 3.18: The part of the state machine that maps the rule's subjects

This makes the creation of a generic acceptor that can handle all possible predicate names very difficult. For this reason the or acceptor is used that is described in section 3.2.1, which is used to concatenate all predicate names. This is done once, when the state machine is created. It is possible to make this static acceptor, because there is no way to change predicates using a Dicto statement. Using this approach makes the post-process rather simple. It looks up the given predicate and matches it with the subject's variable type. This is necessary because predicates only work with a specified set of variable types. If this fails, a semantic error is raised and the state machine is terminated. The suggestor returns the set of all predicates known to the environment, but filters them using the subject's type. The corresponding diagram can be seen in figure 3.19. The last part of the rule statement consists the objects. These are either integers,

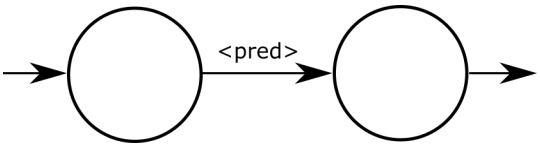

Figure 3.19: The part of the state machine that maps the rule's subjects.

strings, or other variables. Here are a few examples using all possible object types:

```
Model cannot depend on View, Controller
Google must have latency < 1
```

```
BuildFile must contain text "foobarbaz"
```

A relevant point is that each predicate can only have one object type. This means that the states that parse the objects, are quite similar to those, which parse the rule subjects. The relevant part of the state machine is visualized in figure 3.20 The state machine branches into three similar sub branches. Each is responsible

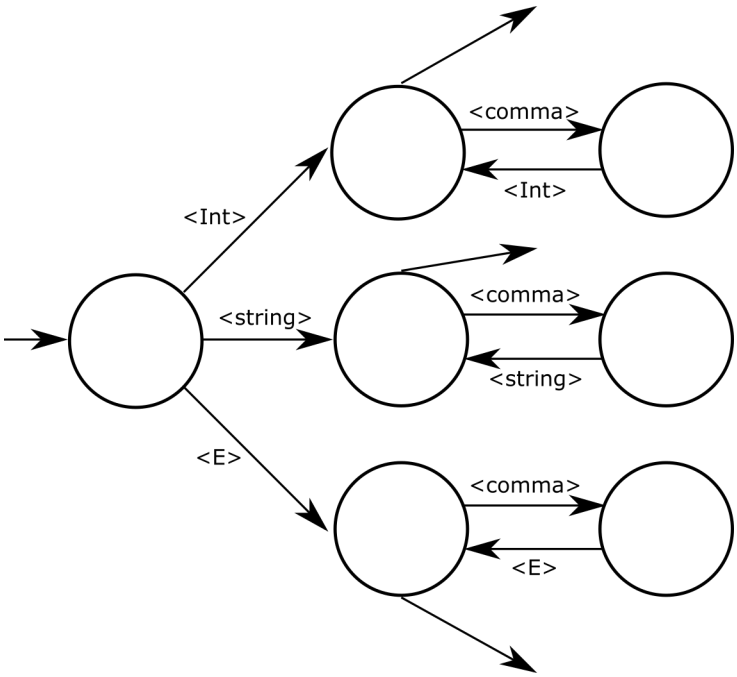

Figure 3.20: The part of the state machine that maps the rule's objects.

to match a specific object type. The first two branches match integers and strings. Those do not need post processes, nor suggestors. The third branch takes variables as objects, as in the first example statement. Again, if multiple objects are given, it must be ensured that all objects share a common variable type. In addition, the predicates request a specific type. For this the first transition parses a variable. The post process of the given transition looks up the specified variable name in the environment. If it can't be found, a semantic error is generated and the state machine is terminated. In addition, it matches the variable type of the given variable against the allowed types, specified by the predicate. If this does not hold, the process of generating suggestions is aborted and a semantic error is returned. Lastly, the variable is stored in the cache of the environment. The suggestor simply returns a set of known variables, which share the type with the allowed types, specified by the predicate. The transition that goes back to the intermediate state works in a very similar way.

3.3.3 Only Can Rule

This is a sub form of the normal rule statement. Its syntax differs, but semantically it is the same. An example for this kind of rule is:

```
only Tests can access Model
```

What is significant in this kind of statement, is that the mode "only can" encapsulates the subject. So we need to extend our state machine to handle this special case. The corresponding diagram is visualized in

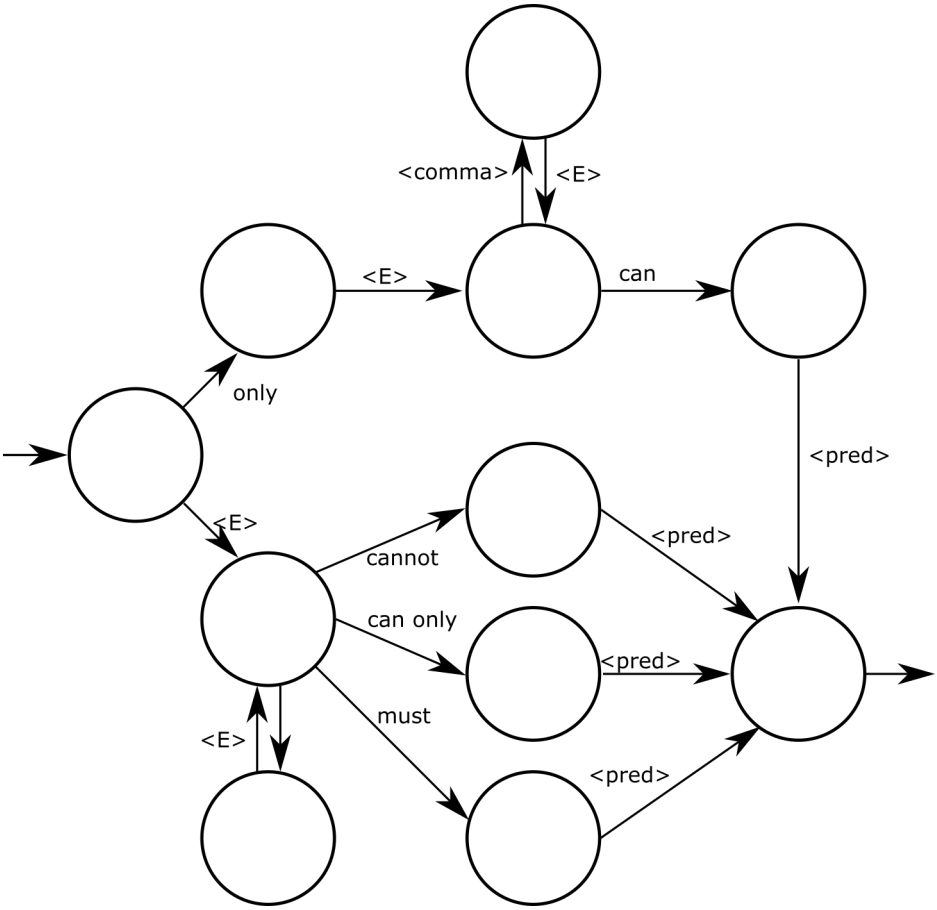

Figure 3.21: The complete state machine.

figure 3.21. The above half of the diagram matches the the beginning of the “only can” rule:

```
only Tests can
```

The transition that matches the keyword “only” has similar post process and suggestor as the transition that matches the keyword “with” in section 3.3.1. In detail, the transition has no post process and the suggestor statically returns a single element set containing “only”. The handling of the subject and the predicate is working exactly the same as the corresponding part in the normal rule statement which can be read in more detail in section 3.3.2.

3.3.4 Complete state machine

Now that all parts of the state machine are explained in detail. Each component can be assembled to the complete state machine seen in figure 3.22.

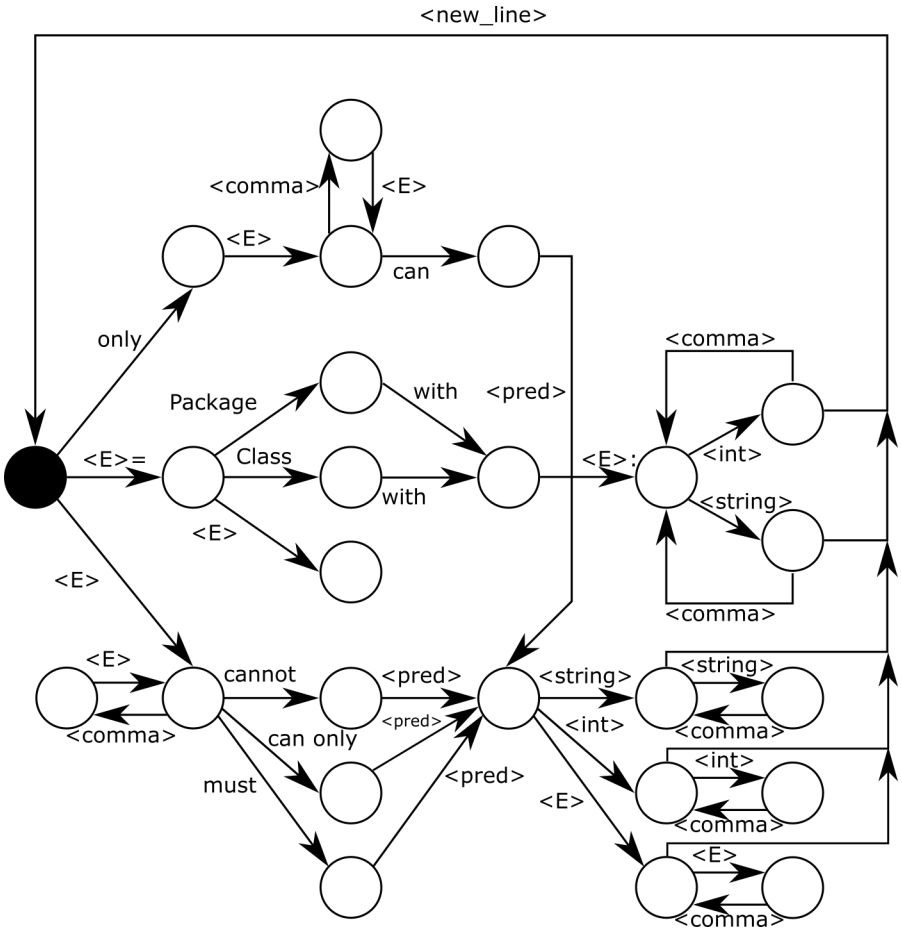

Figure 3.22: The complete state machine.

4 The Validation

Most IDEs can handle autocompletion in a very short amount of time. This means after the user enters his input, the suggestions pop up instantaneously, or with a barely noticeable delay. This implemented solution needs to provide its service in a comparable time frame.
For this purpose, three test scenarios were set up. Each scenario was run a thousand times and the measured times were averaged to ensure meaningful results.

4.1 Scenarios

The load tests were set up in a way that three different variables could be examined. The first variable is the input size. Whenever the client sends a request to the user, it needs to include all the Dicto statements the user has written up to the cursor's position. From this it follows that the delay the user experiences, after the request was sent, increases with the amount the user writes. To ensure that even bigger files can be handled, all tests will be run with a range of inputs, the largest having 50000 statements.
Next, a limiting factor was seen in the semantic environment of the state machine. A larger number of variable types and variables might increase lookup times. Having to do many of these lookups during parsing, the state machine could slow down.
Lastly, the network might limit the solution. Having to send all the text, each time the user needs code completion, might slow down things when client and server are not running on the same machine, but only in the same local area network.

To test the performance of Dicto autocompletion three different scenarios will be introduced. Each scenario will be run using differently sized inputs. The smallest input includes 1000 Dicto statements, whereas the largest input has 50000 statements.

4.1.1 Localhost scenario

This scenario was set up as comparison to the following two scenarios. In this scenario both server and client are set up on the same machine. The reason for this is to eliminate the delay a network connection would cause. When running this scenario the delay will mostly result from the time the state machine needs

to parse the input, and this means that its performance can be evaluated. Additionally, the environment was set up with very few variable types.

4.1.2 Big environment scenario

This scenario was introduced to evaluate the performance loss a bigger amount of variable types would have. For this reason, a test scenario was set up that was similar to the localhost scenario. The difference between the two scenarios is that the number of variable types was increased to 50000. This number is not reasonable, but if the state machine is fast enough in this scenario, it will be able to handle all real work application regarding environment size. Also, the inputs were set up to access the added variable types.

4.1.3 One Hop scenario

In this scenario, the effect of the network will be investigated. Server and client were separated using a standard 1 Gbit ethernet switch, so the tests would use the same environment and inputs that the localhost scenario already uses to ensure a meaningful comparison between them.

4.2 Results

4.2.1 Localhost

Figure 4.1 displays the measured times for the localhost scenario. The graph shows that growing input sizes increase the processing time of the back end in linear fashion. 10000 statements take around 0.005 seconds, while 50000 take 0.02 seconds. Deduced from that, it can be said that every 10000 statements increase the delay by around 0.005 seconds.
The literature indicates that the user perceives delays that does not exceed 0.1 seconds as immediate. In

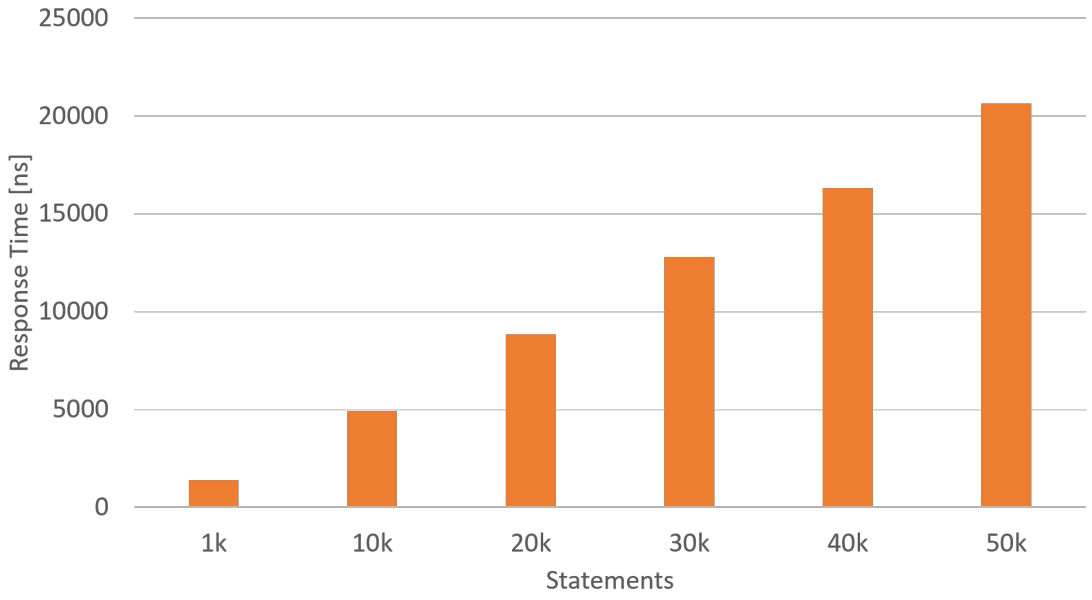

Figure 4.1: The test results of the localhost scenario.

this scenario, this threshold is reached around 250000 statements. This would be a very big input file and not plausible to be reached in any real world application. Even 10000 statements seem to be an exaggeration. In this scenario the solution is surely fast enough to satisfy the user's demand regarding performance.

But those results are not very informative. Firstly, using a very small environment can sugar-coat the results to look better than they actually are and, secondly, without knowing the influence of a network based set up, these test results limit the solution to a very specific and small set of use cases. More tests need to be run and pass before the solution can be considered applicable.

4.2.2 Big environment

In this subsection the second scenario is compared to the localhost scenario. This is needed to understand which influence the size of the environment has on the performance of the solution.
Figure 4.2 includes the results of the second scenario and as well as the results of the localhost scenario.

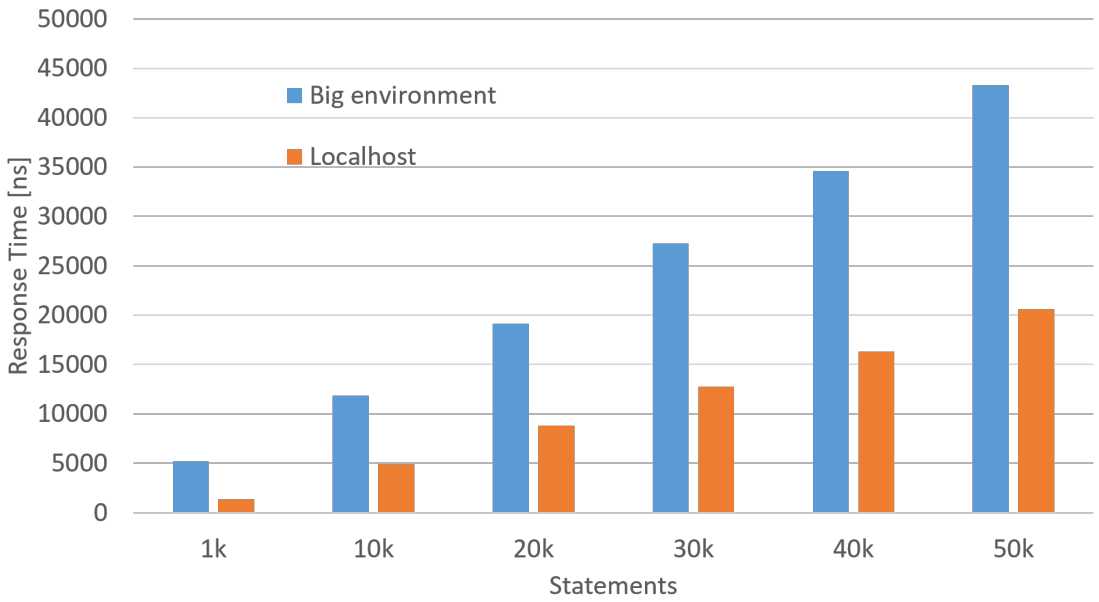

Figure 4.2: The test results of the big environment scenario compared to the localhost scenario

The graph shows that the processing time of the state machine changes significantly if a bigger environment is used. In the second scenario, it takes around 0.01 seconds to process 10000 statements. For 50000 statements it needs around 0.045 seconds. This means that the time needed to process an input increases by around 0.0075 seconds per 10000 statements.

Those numbers are much bigger compared to the localhost scenario, but still very low. In this new scenario, the 0.1 second threshold will be reached at around 100000 to 125000 statements. As already explained, these dimensions of inputs are not in the range of anything expected.

Additionally, it can be argued that the size of the environment is out of range of anything expectable. Real life environments might reach the hundreds, but those would already be considered huge. As conclusion, it

can be said that this second scenario shows that the size of the environment has only a very small impact on the performance of the implemented solution and that it can be neglected.

4.2.3 One Hop network

In this subsection, the effects of a setup, using the network will be discussed. Figure 4.3 visualizes the results of the third test scenario alongside the results of the localhost scenario. On first glance, it can be seen that the network has a serious impact on the performance of the solution.
10000 statements are processed in around 0.075 seconds. For 50000 statements the state machine takes

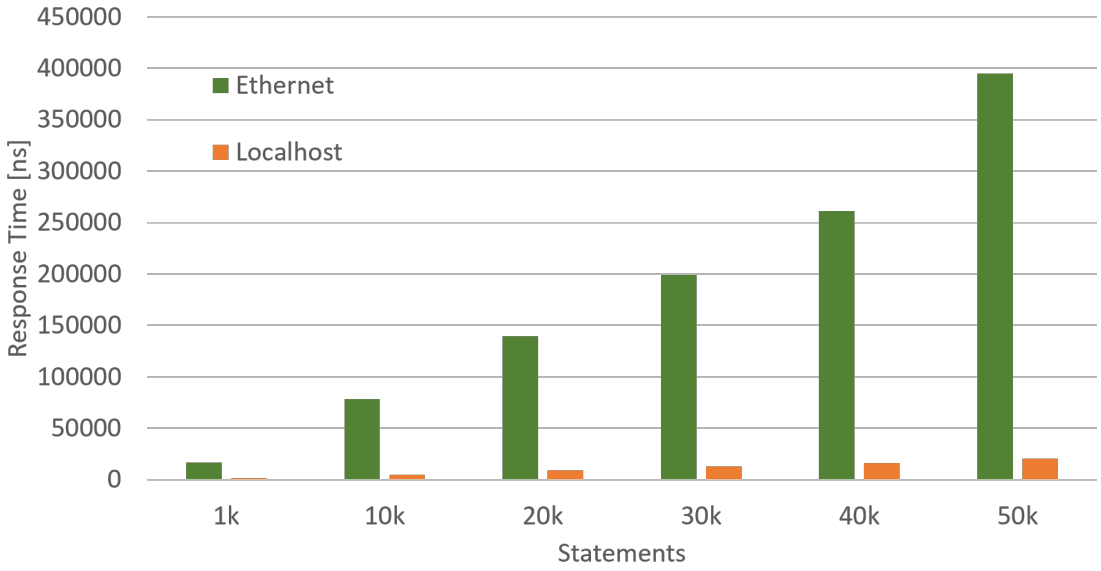

Figure 4.3: The test results of the 1 Hop scenario.

around 0.4 seconds. It can be deduced that every 10000 statements increase the processing time by around 0.08 seconds. This means that the threshold of 0.1 second is reached faster, namely, at 10000 to 150000 statements.
The results for the network scenario are larger compared to the localhost scenario by factor 19.14. In other words, in the third scenario, the network causes 95% of delay. This identifies the network as a main bottleneck of the solution. In addition, the scenario only tests the set up in an unused network with only one intermediate switch. It can be argued that the delay in a real local network will be much larger.

The usability of the solution is mainly restricted by network speed and input size. But delay only gets troublesome when reaching bigger input sizes. Having Dicto documents containing 10000 statements is not really expected at the moment, which means that the results of the load test can be seen positively. The solution should be fast enough for most application in a restricted network environment.

5 Future Work

In this chapter of the thesis future additions to the project will be discussed. For this reason, the chapter is organized into two sections. The first section examines improvements to already existing features. The second part of this chapter talks about features that are needed or wanted, but not implemented at this point in time.

5.1 Improvements

Acceptors need to be provided for every path of the state machine. This means that many different acceptors need to be created in the process of modelling a language. Providing a regular expression language to assist creating the acceptors would bring a few benefits. Firstly, It would shorten the amount of code that is needed to create a state machine, which increases readability. In addition, most programmers are familiar with regular expressions. This would decrease the amount of work a programmer needs to get used to the project.

Additionally, this component is still missing quantifiers and other basic functionality that most regex libraries provide. Adding those would shorten a lot of the acceptors that are created. For example,one must combine the equivalent of "+" and "?" to create "*". Extending the acceptors to support functionality that is equivalent to the regular expressions of "*", "{n}", "{n, m}", or "\w" would improve the usability significantly.

5.2 Additional Features

Here we explain features that were discussed, but not yet implemented.

5.2.1 Response data

This is a basic feature that the user would benefit a lot from. At the moment, if the state machine parses an input successfully, it returns the suggestions as a list of strings. Afterwards, the server then formats that list as a JSON Array and sends this as a response, to the received request. Adding meta data to each

element of this list could improve usability of the solution. This means, instead of just returning a list of suggestions, the response contains information, which element is a keyword, which is a variable, and so on. For variables, the type, and the values of its attributes could be included. Providing this information to the user on the fly could turn out to be very informative and valuable.

5.2.2 External Configuration

This is probably the most important point on this list. The current version of the implementation uses a state machine with a hard coded configuration. This means that the variable types that exist in the environment need to be compiled alongside the other code. This is only a temporary solution and should to be changed as soon as possible. But much more functionality can be added to this feature. For example, the server could support multiple different configurations. The server's API could then provide a list of all configurations and the client specifies the wanted configuration when sending the actual user's input. In addition, a client could add its one configuration remotely to the server. What the actual range of the supported functionality needs to be is still debatable.

5.2.3 Code Highlighting

I mention code highlighting as an example for all the other useful features a normal IDE provides. The user does not want to search for each feature he or she wants individually. He or she wants it as a complete package. Adding more different features might increase the appeal of Dicto and this project to other programmers. Of course, implementing a full IDE is not the scope of this project and providing a network based IDE might not work, but it is still something that needs to be considered.

6 Conclusion

Taking a step back, one could say that the project was a success under the aspect of proving the concept. The validation shows that code completion can work in a client server model. The test results show that the network transmission causes roughly 95% of the delay the user experiences. In conclusion, finding ways that enable the solution communicate in a more efficient fashion, should be the next option to be explored. Otherwise, a few obvious improvements and additions were already mentioned in the previous chapter 5. A challenging task was to keep up with a quickly changing language.

Looking at the project not as a solution, but as a process of development, I can say that I have much to learn. Not only making sure that the project stays organized, but also keeping up morale is a task not to be underestimated. Working on a project this size for the first time generates many points in time, when one realizes that poor decisions were made - probably as often as correct ones. Developing a solution to a specific problem and making sure it does not miss the target, are the main issues that I stumbled upon not only once, but more times that I can count.

In the end, I can say that I learned a lot as programmer and as person: Writing Unit tests as soon as possible can save you a lot of time and frustration, the amount of refactoring needed for a project that I consider to be small is immense, and lastly I improved a lot at focusing on the main features instead of losing myself in little “side quests”. I am very proud that I was able to create my own working project and consider it a success.

7 Anleitung zu wissenschaftlichen Arbeiten

In this chapter I introduce the implemented solution. First, there is an quick overview over most of the classes. Sections 7.2 to 7.5 explain how to build and use a state machine. The last two sections cover the server and a front end demo.

7.1 Overview

Figure 7.1 covers the model component of the implementation. It represents the semantic information a language can have and provides an `Environment` class.

`Variable` represents a variable defined by the Dicto language. It contains a variable type as well as the variable's name.

`VariableType` specifies which arguments a variable of its type can use. In addition, it contains the name of the represented variable type.

`Rule` is an enum representing all the modes available in Dicto: can not, only can, etc.

`Predicate` represents a predicate like "can access" or "depends on". It contains a list of all variable types compatible with the given Dicto predicate.

`Environment` manages all variables and variable types available in a context.

Figure 7.2 shows all classes that make up a state machine.

`StateMachine` represents a complete state machine. It contains a single `State` which is its initial state. It provides a method that takes a `Context` and `Environment` object as input and returns a `StateMachineResult` object.

`StateMachineResult` is returned by the `StateMachine` and indicates whether it accepts the input or not. Depending on that condition this object either contains an error message or a list of suggestions.

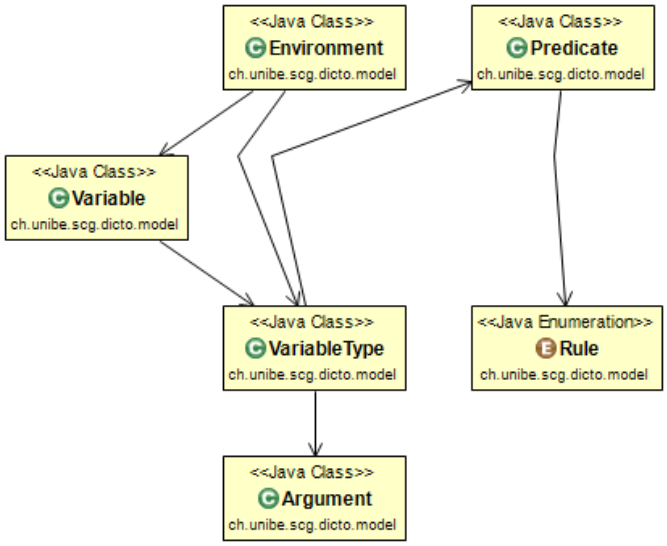

Figure 7.1: The uml class diagram of the model classes.

`State` is a collection of all Paths originating from the given state. It also contains methods that determines which Path accepts a given input.

`AfterSuccessAction` is an interface. An implementation is provided to each Path. The implemented `apply` method is executed after the `Path`'s `Acceptor` accepted an input.

`Suggestor` is an interface. A subclass is given to each Path. When the state machine terminates successfully these are used to provide partial lists for suggestions. These lists are combined to a definitive list by the state the machine was in when it terminated.

`Path` represents a connection between two states. Additionally it provides `Acceptor`, `AfterSuccessAction`, and `Suggestor` objects.

`StateResult` is returned by a state and indicates whether the state machine can transition from the state.

Figure 7.3 contains all classes implemented for the tri-valued parsing component. Section 7.2 covers all the classes and their responsibility in detail.

Figure 7.4 shows all the classes needed to model the Dicto language. The `Dicto` class is a builder for the actual `StateMachine` object and the only method relevant is the `build` method. The remaining classes are implementations of `AfterSuccessAction` or `Suggestor`.

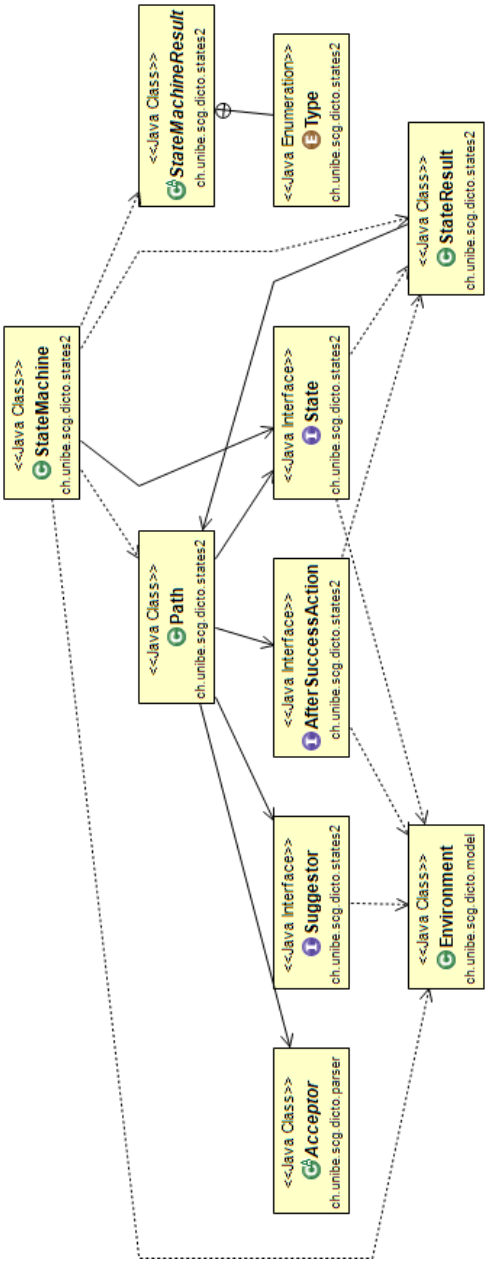

Figure 7.2: The Uml class diagram of the state machine classes.

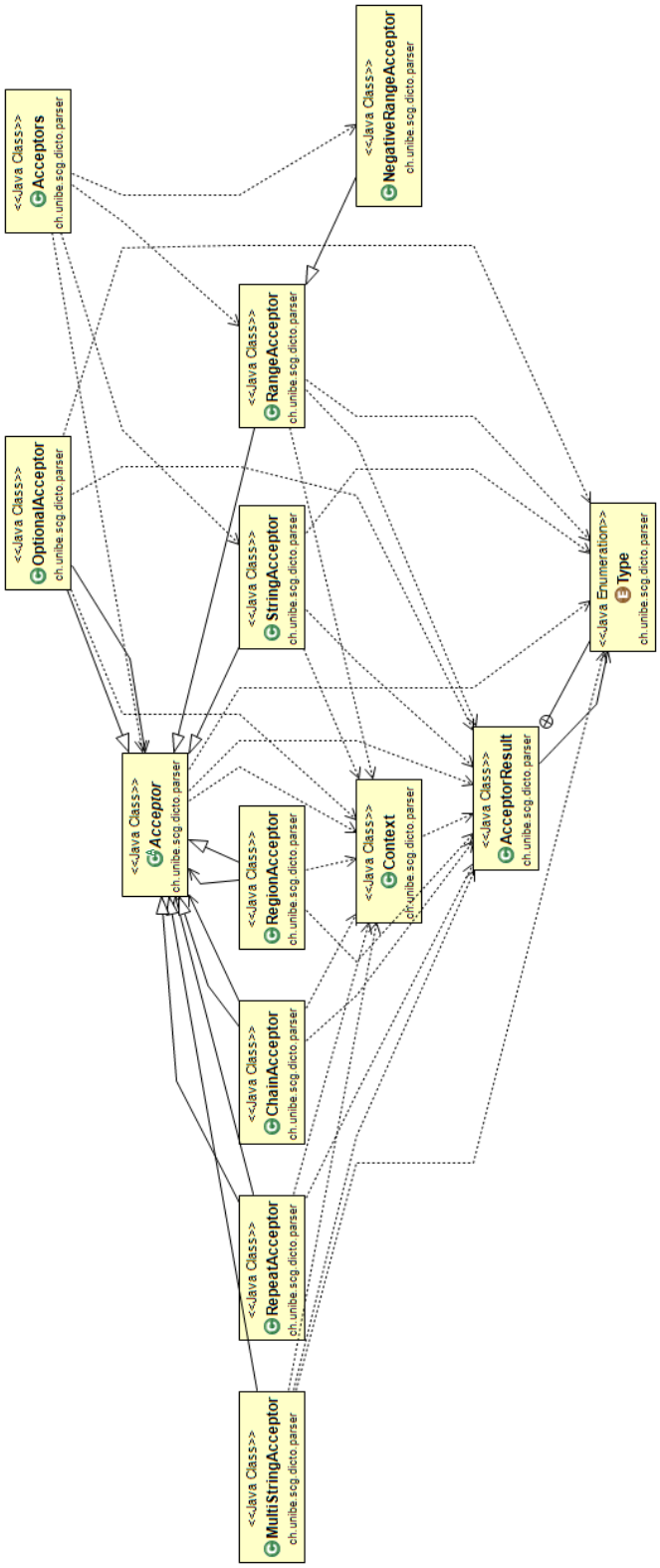

Figure 7.3: The Uml class diagram of the parsers.

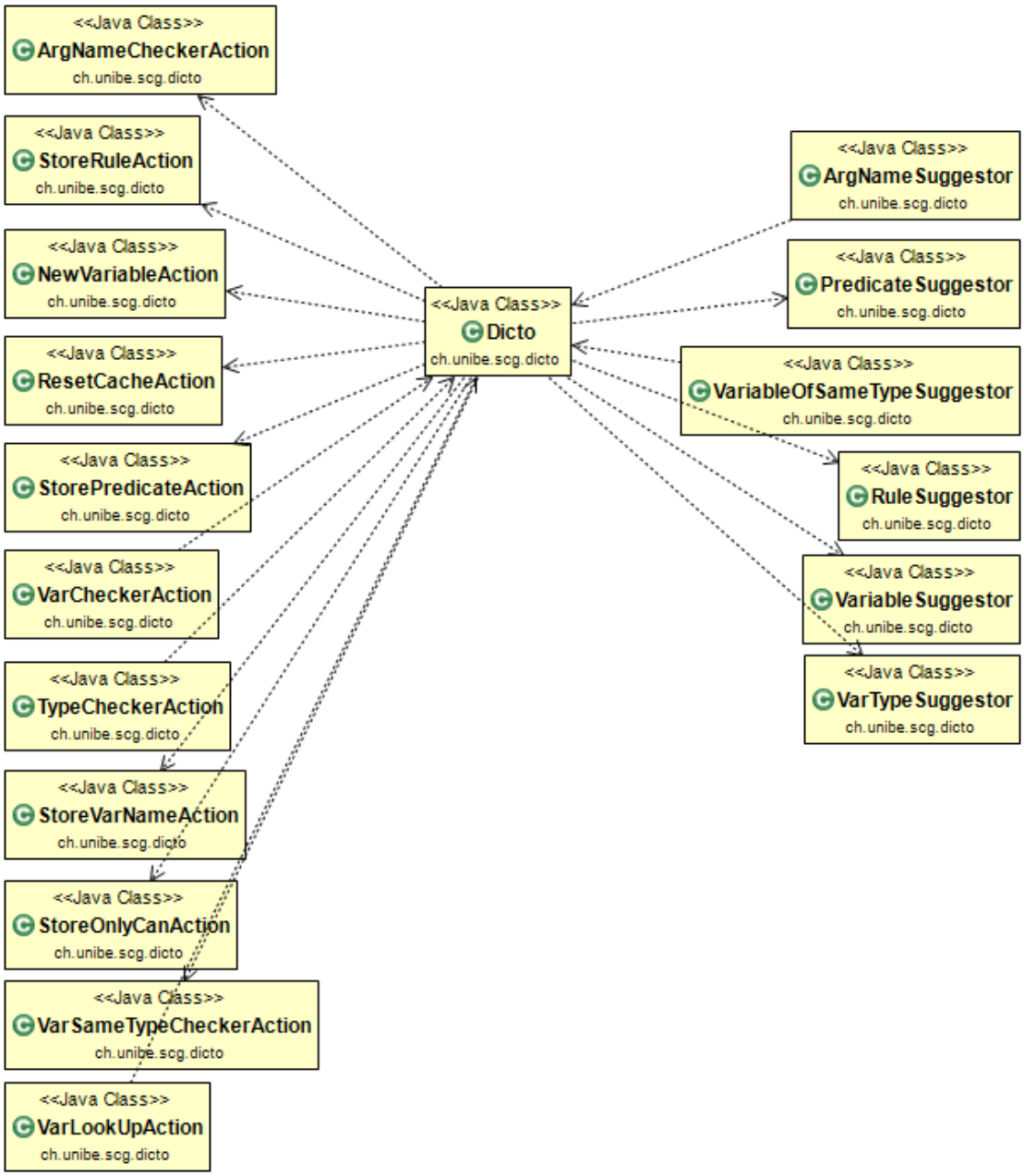

Figure 7.4: The Uml class diagram of the Dicto classes.

7.2 Acceptors

This component is responsible for parsing the input regarding syntactic information. It implements the functionality described in section 3.2.1. The base class for implemented Acceptors has one method of importance, as seen in figure 7.5. The `parseOn` method takes a `Context` object and returns an `AcceptorResult` object. The `parseOn` method has no implementation in the `Acceptor` class. It is implemented differently by all the various subclasses discussed later. The full code can be found at `http://github.com/grushnakk/DAWS`

```
package ch.unibe.scg.dicto.parser;

/* imports */

public abstract class Acceptor {

   public abstract AcceptorResult parseOn(Context context, AcceptorResult result);

   /*
    * ...
    */
}
```

Figure 7.5: An extract of the Acceptor base class.

`Context`, as outlined in figure 7.6, is a simple class that combines a `String` with an index. The `String` is the whole input, while the index indicates where the `Acceptor` needs to start parsing the code. It provides different methods to access the underlying `String` such as `currentChar`, `charAt`, and `substring`.

The `Context` is not updated by the `Acceptor` automatically. This needs to be done by calling the `Context`'s `apply` method passing the given `AcceptorResult`. If the index of the Context object is the same value as the `AcceptorResult`'s begin index, it will set the index to the `AcceptorResult`'s end index.

The `AcceptorResult` holds information about the outcome of the Acceptor's attempt to parse the input. Firstly, it contains the information if the `Acceptor` was able to parse the `Context` starting at its current index. Depending on that, the `AcceptorResult`'s type is set to either FAILURE, SUCCESS, INCOMPLETE, which is according to the described functionality in 3.2.1.

Further, the `AcceptorResult` holds information about how much of the `Context` was parsed by `Acceptor` providing a start position and an end position.

```
package ch.unibe.scg.dicto.parser

/* imports */

public class Context {

   public Context(String content) {/* code */}

   /**
    * returns the character the current index is pointing at.
    */
   public char currentChar() {/* code */}

   /**
    * returns the character at the given index.
    */
   public char charAt(int index) {/* code */}

   /**
    * returns the size of the given content String.
    */
   public int size() {/* code */}

   /**
    * returns the size of the given content String starting at the current index.
    */
   public int sizeLeft() {/* code */}

   /**
    * returns the current index.
    */
   public int getCurrentIndex() {/* code */}

   public String substring(int length) (/* code */}

   public String substring(int start, int end) {/* code */}

   public void apply(AcceptorResult result) {/* code */}
}
```

Figure 7.6: An extract of the Context class.

```
package ch.unibe.scg.dicto.parser;

/* imports */

public class AcceptorResult {

   public static enum Type { FAILURE, SUCCESS, INCOMPLETE; }

   public Type getType() {/* code */}

   public int getBeginPosition() {/* code */}

   public int getEndPosition() {/* code */}

   public String getRegion(String key) {/* code */}

   /*
    * ...
    */
}
```

Figure 7.7: An extract of the AcceptorResult class.

7.2.1 RangeAcceptor and NegativeRangeAcceptor

The first subclasses are the `RangeAcceptor` and `NegativeRangeAcceptor` class. Both parse only a single character. The first of the two classes only accepts a character, which is contained by a given set. The second class does the opposite: It accepts any character except the ones in the given set.

```
public ch.unibe.scg.dicto.example;

import ch.unibe.scg.dicto.parser.*;

public class RangeAcceptorExample {

   public static void main(String[] args) {
      RangeAcceptor rangeAcceptor = new RangeAcceptor("0123456789");
      /*
       * STATEMENT 1
       */
      rangeAcceptor.parseOn(new Context("3")).getType();
      /*
       * STATEMENT 2
       */
      rangeAcceptor.parseOn(new Context("34")).getType();
      /*
       * STATEMENT 3
       */
      rangeAcceptor.parseOn(new Context("")).getType();
      /*
       * STATEMENT 4
       */
      rangeAcceptor.parseOn(new Context("a")).getType();
   }
}
```

Figure 7.8: An example on how the RangeAcceptor works.

The example in 7.8 illustrates how to utilize the `RangeAcceptor` class. Since this type of `Acceptor` only accepts a single character, the first two statements have the same result, when executed: Both succeed by parsing the “3”. Statement 3 returns the type incomplete since the parser tries to accept a character, but there is no input left in the Context. The last statement will fail because the character “a” is not contained by the string passed in the constructor.

7.9 shows an example on how to use the NegativeRangeAcceptor class. It works similar as the RangeAcceptor. The only difference is that this class only accepts a character that is not contained by the String parameter in the constructor.

```
public ch.unibe.scg.dicto.example;

import ch.unibe.scg.dicto.parser.*;

public class RangeAcceptorExample {

   public static void main(String[] args) {
      RangeAcceptor rangeAcceptor = new RangeAcceptor("0123456789");
      /*
       * STATEMENT 1
       */
      rangeAcceptor.parseOn(new Context("3")).getType();
      /*
       * STATEMENT 2
       */
      rangeAcceptor.parseOn(new Context("34")).getType();
      /*
       * STATEMENT 3
       */
      rangeAcceptor.parseOn(new Context("")).getType();
      /*
       * STATEMENT 4
       */
      rangeAcceptor.parseOn(new Context("a")).getType();
   }
}
```

Figure 7.9: An example on how the NegativeRangeAcceptor works.

7.2.2 StringAcceptor

The `StringAccepted` class is initialized using a `String` as constructor input. When parsing, the `Acceptor` succeeds if the `Context` contains the given `String` starting at the `Context`'s index. If the input ends before the whole String can be read, but the characters still match, INCOMPLETE will be returned.

```
public ch.unibe.scg.dicto.example;

import ch.unibe.scg.dicto.parser.*;

public class StringAcceptorExample {

   public static void main(String[] args) {
      StringAcceptor stringAcceptor = new StringAcceptor("Bachelor");
      /*
       * STATEMENT 1 + 2: SUCCESS
       */
      stringAcceptor.parseOn(new Context("Bachelor")).getType();
      stringAcceptor.parseOn(new Context("Bachelor Thesis")).getType();
      /*
       * STATEMENT 3: INCOMPLETE
       */
      stringAcceptor.parseOn(new Context("Bachel")).getType();
      /*
       * STATEMENT 4: FAILURE
       */
      stringAcceptor.parseOn(new Context("Bachek")).getType();
   }
}
```

Figure 7.10: An example on how the StringAcceptor works.

7.2.3 RepeatAcceptor

The `RepeatAcceptor` takes a delegate acceptor and parses it multiple times. It expects that the delegate `Acceptor` will succeed at least once. Otherwise, the `RepeatAcceptor` will return FAILURE. If the last time, the delegate Acceptor is run, returns INCOMPLETE, it will return INCOMPLETE as well.

```
public ch.unibe.scg.dicto.example;

import ch.unibe.scg.dicto.parser.*;

public class RepeatAcceptorExample {

   public static void main(String[] args) {
      RepeatAcceptor repeatAcceptor = new StringAcceptor(new StringAcceptor("abc"));
      /*
       * STATEMENT 1 + 2: SUCCESS
       */
      repeatAcceptor.parseOn(new Context("abc")).getType();
      repeatAcceptor.parseOn(new Context("abcabc")).getType();
      /*
       * STATEMENT 3 - 5: INCOMPLETE
       */
      repeatAcceptor.parseOn(new Context("")).getType();
      repeatAcceptor.parseOn(new Context("ab")).getType();
      repeatAcceptor.parseOn(new Context("abca")).getType();
      /*
       * STATEMENT 6: FAILURE
       */
      repeatAcceptor.parseOn(new Context("d")).getType();
   }
}
```

Figure 7.11: An example on how the RepeatAcceptor works.

7.2.4 OptionalAcceptor

The `OptionalAcceptor` takes a delegate `Acceptor`. It runs the given Acceptor. If the delegate returns FAILURE, the `OptionalAcceptor` simply returns an `AcceptorResult` where begin index and end index are the same and the type is set to SUCCESS. If the delegate returns something else, it simply passes the result on.

```
public ch.unibe.scg.dicto.example;

import ch.unibe.scg.dicto.parser.*;

public class OptionalAcceptorExample {

   public static void main(String[] args) {
      OptionalAcceptor optionalAcceptor =
         new OptionalAccecptor(new StringAcceptor("abc"));
      AcceptorResult result = null;
      /*
       * STATEMENT 1
       */
      AcceptorResult result = optionalAcceptor.parseOn(new Context("abc"));
      result.getType(); //SUCCESS
      result.getBeginIndex(); //0
      result.getSize(); //3
      /*
       * STATEMENT 2
       */
      result = optionalAcceptor.parseOn(new Context(""))
      result.getType(); //SUCCESS
      result.getBeginIndex(); //0
      result.getSize(); //0
      /*
       * STATEMENT 3
       */
      result = optionalAcceptor.parseOn(new Context("bcd"))
      result.getType(); //SUCCESS
      result.getBeginIndex(); //0
      result.getSize(); //0
   }
}
```

Figure 7.12: This example shows how the OptionalAcceptor class works.

7.2.5 ChainAcceptor

The `ChainAcceptor` takes an array of two or more Acceptors when initialized. This `Acceptor` succeeds parsing an input if all delegate parsers succeed. This means that the context is given to the first delegate parser. If that parser fails, parsing aborts and FAILURE is returned. But if it succeeds, the context's index is updated and the second `Acceptor` parses the input. This is done until all delegate parsers succeed or one returns FAILURE or INCOMPLETE. The `Context` passed by the user, will not be altered. Instead the `ChainAcceptor` will create a copy of the given `Context`.

```
public ch.unibe.scg.dicto.example;

import ch.unibe.scg.dicto.parser.*;

public class ChainAcceptorExample {

   public static void main(String[] args) {
      ChainAcceptor chainAcceptor = new ChainAccecptor(
         new RangeAcceptor("abc"),
         new RangeAcceptor("012"));
      AcceptorResult result = null;
      /*
       * STATEMENT 1
       */
      AcceptorResult result = chainAcceptor.parseOn(new Context("a1"));
      result.getType(); //SUCCESS
      /*
       * STATEMENT 2
       */
      result = chainAcceptor.parseOn(new Context("a3"))
      result.getType(); //FAILURE
      /*
       * STATEMENT 3
       */
      result = chainAcceptor.parseOn(new Context("b"))
      result.getType(); //INCOMPLETE
   }
}
```

Figure 7.13: This example shows how the ChainAcceptor class works.

7.2.6 RegionAcceptor

This `Acceptor` does not introduce any parsing functionality, but a way to mark certain parts of the parsed input. The `RegionAcceptor` is handed a key and an acceptor when initialized. When parsing, the `RegionAcceptor` simply delegates to the `Acceptor`. If this `Acceptor` succeeds, the `RegionAcceptor` creates a substring of the input using the result's begin and end index. Then the value is stored in the `AcceptorResult` using the given key.

```
public ch.unibe.scg.dicto.example;

import ch.unibe.scg.dicto.parser.*;

public class RegionAcceptorExample {

   public static void main(String[] args) {
      RegionAcceptor regionAcceptor =
         new RegionAccecptor("Key",  new StringAcceptor("Value"));
      ChainAcceptor chainAcceptor = new ChainAcceptor(
         new StringAcceptor("<"),
         regionAcceptor,
         new StringAcceptor(">"));
      AcceptorResult result = null;
      /*
       * STATEMENT 1
       */
      AcceptorResult result = chainAcceptor.parseOn(new Context("<Value>"));
      result.getType(); //SUCCESS
      result.getRegion("Key"); //Value
   }
}
```

Figure 7.14: This example shows how the RegionAcceptors class works.

7.2.7 Building Acceptors

The abstract `Acceptor` class implements a few methods to facilitate creating complex acceptors as shown in example 7.15. The `repeat` method returns a `RepeatAcceptor` that uses the given ac-

```
public abstract class Acceptor {
   /*
    * ...
    */

   public Acceptor repeat() { /* code */};

   public Acceptor optional() { /* code */};

   public Acceptor region(String key) { /* code */};

   public Acceptor chain(Acceptor... acceptors) { /* code */};
}
```

Figure 7.15: Utility methods of the `Acceptor` class.

ceptor as a delegate. This means that `new RepeatAcceptor(someAcceptor)` is equivalent to `someAcceptor.repeat()`. Methods `optional` and `region` work in a similar fashion.
`chain()` creates a new `ChainAcceptor` that is initialized with an array of delegate acceptors. The first element is the object that method was invoked on. The next elements are the acceptors specified in the parameters. For example, `someAcceptor.chain(acceptor1, acceptor2)` is equivalent to `new ChainAcceptor(someAcceptor, acceptor1, acceptor2)`.
Also included is the `Acceptors` class that supports a few static methods and values to improve writing composite acceptors. It offers static Strings that can be used to initialize a `RangeAcceptor` which cover letters, digits and white space. It also offers factory methods to create `RangeAcceptor`, `NegativeRangeAcceptor` and `StringAcceptor` objects. Example 7.16 features a complex acceptor.
It is important to note that all acceptors are stateless and can be reused within the state machine.

```
public ch.unibe.scg.dicto.example;

import ch.unibe.scg.dicto.parser.*;

public class AcceptorsExample {

   public static void main(String[] args) {
      /*
       * the following composite acceptor parses an simple identifier
       * followed by a equal sign.
       */
      Acceptor idAcceptor = range(RANGE_LETTERS)
         .chain(range(RANGE_LETTERS + RANGE_DIGITS).repeat().optional());
      Acceptor assignAcceptor = idAcceptor.region("REGION_KEY")
         .chain(range(RANGE_WHITESPACE).optional(), string("="));
      AcceptorResult result = null;
      /*
       * STATEMENT 1
       */
      result = assignAcceptor.parseOn(new Context("variable01="));
      result.getType(); //SUCCESS
      /*
       * STATEMENT 2
       */
      result = assignAcceptor.parseOn(new Context("variable01\t\n="));
      result.getType(); //SUCCESS
      /*
       * STATEMENT 3
       */
      result = assignAcceptor.parseOn(new Context("variable01"));
      result.getType(); //INCOMPLETE
      /*
       * STATEMENT 3
       */
      result = assignAcceptor.parseOn(new Context("variable01\t\n"));
      result.getType(); //INCOMPLETE
      /*
       * STATEMENT 4
       */
      result = assignAcceptor.parseOn(new Context("variable01 :"));
      result.getType(); //FAILURE
   }
}
```

Figure 7.16: An example using a more elaborate Acceptor

7.3 Building a state machine

In my implementation state machines are defined in Java. To aid the programmer I created three builder classes: `StateMachineBuilder`, `StateBuilder`, and `PathBuilder`. The most important one of these three is the `PathBuilder` class because Paths have the most room for configuration. To create a new state machine we start by creating a new instance of `StateMachineBuilder`:

```
StateMachineBuilder stateMachineBuilder = new StateMachineBuilder();
```

Each state has a name represented by a `String`. States can't be referenced using its `State` object, because we might reference it before it is created. More precisely, all states are created when the builder's `build` method is called. To grab a `stateBuilder` for a state we want to add to the state machine, we simply call the `state` method of our `StateMachineBuilder` instance passing a String. Whenever passing the same `String`, we will receive the same `StateBuilder` instance. For this reason, I highly recommend using constant fields instead of magic values:

```
StateBuilder stateBuilder = stateMachineBuilder.state(INITIAL_STATE);
```

The next step would be creating a path originating from our `INITIAL_STATE`. We create a `PathBuilder` by calling the `pathTo` method passing a `String` referencing a state. We don't need to have created a corresponding `StateBuilder` before referencing a state because these references will be resolved when the state machine is finalized.

```
PathBuilder pathBuilder = stateBuilder.pathTo(STATE_VARIABLE_TYPE);
```

Using the `PathBuilder`, we can specify multiple attributes of the path. For starters, we might want to define the acceptor:

```
pathBuilder.accept(
   idAcceptor.region(REGION_ID).chain(optionalWhitespace(), string("="))
);
```

All methods of the `PathBuilder` class return the `PathBuilder` instance to support method chaining. Next we can define an action we want to execute after the path's acceptor succeeds. For this purpose we pass a custom implementation of the `AfterSuccessAction` interface to the PathBuilder.

```
pathBuilder.onSuccess(new StoreVarNameAction());
```

Below is the single method that needs to be implemented.

```
//StoreVarNameAction#apply
@Override
public StateResult apply(StateResult result, Environment env) {
   String name = result.getAcceptorResult().getRegion(Dicto.REGION_ID);
   //CHECK IF VAR ALREADY EXISTS
   if(env.isVariableDefined(name))
      return new StateResult("variable already defined: " + name);
   //OK, ITS NEW SO LETS CACHE IT
   env.writeCache(Dicto.CACHE_NEW_VAR_NAME, name);
   return result;
}
```

It passes a `StateResult` containing the `AcceptorResult` and the destination state of the current path, as well as the `Environment` containing variables and variable types, etc. Using this action we can extract the variable name the user wants to use and proceed accordingly. We interrupt parsing if the variable name is already in use and otherwise we write the variable name into the cache for later use.

In addition, we want to add an `Suggestor` for each path. This can be done in two different ways. Either call `suggests` passing a single `String`:

```
pathBuilder.suggests("with");
```

This method can be used when a path represents a keyword. Otherwise, we need to pass a custom implementation of the `Suggestor` interface:

```
pathBuilder.suggest(new Suggestor() {
   @Override
   public List<String> suggestions(Environment env) {
      List<String> suggestions = new ArrayList<String>();
      for(VariableType type : env.getVariableTypes())
         suggestions.add(type.getName());
      return suggestions;
   }
});
```

This would return a list with all variable names belonging to already defined variables. Another possibility would to call neither of these methods. `PathBuilder` uses a default `Suggestor` implementation that returns an empty list.

In additon, `PathBuilder` offers two methods: `startWithOptionalWhitespace` and `startWithWhitespace`. These methods add white space acceptors before the actual acceptor. Otherwise the user would need to do something of the kind:

```
pathBuilder.accept(whitespace()
   .chain(
      idAcceptor.region(REGION_ID).chain(optionalWhitespace(), string("="))
   )
);
```

This would need to be done for almost all paths. The `PathBuilder` uses the `startWithWhitespace` option on default.

After we are done defining our path, the `PathBuilder` instance needs to be told that we are done by invoking the `complete` method. This needs to be done for every path.

The last thing to do is to tell the builder which state we are starting at. This is done by passing a state reference to the `startAt` method:

```
stateMachineBuilder.startAt(INITIAL_STATE);
```

This can be done before the specific state was defined.

After we have defined all States and Paths, we can retrieve the complete state machine using the `build` method of the `StateMachineBuilder` class. Example 7.17 wraps up this chapter.

7.4 Creating an Environment

After we build the state machine, we must provide all variable types and predicates. This is done using the `Environment` class.

```
StateMachineBuilder stateMachineBuilder = new StateMachineBuilder();
stateMachineBuilder.state(INITIAL_STATE).pathTo(STATE_VARIABLE_TYPE)
   .accept(idAcceptor.region(REGION_ID).chain(optionalWhitespace(), string("=")))
   .startWithOptionalWhitespace()
   .onSuccess(new StoreVarNameAction())
   .complete();
StateMachine stateMachine = stateMachineBuilder.build();
```

Figure 7.17: How to build a state machine using `StateMachineBuilder`.

7.4.1 Variable types

A variable type consists of three components: its name, its predicates and its attributes. Creating an instance of `VariableType` thus requires a `String` representing the name, a List of `Rule` objects which represent the predicates, and a list of `Argument` objects representing the attributes.

```
List<Argument> packageArgs = new ArrayList<>();
packageArgs.add(new Argument("name"));
List<Rule> packageRules = new ArrayList<>();
packageRules.add(new Rule("depend on", new ArrayList<Predicate>(){{
   add(MUST);
   add(CANNOT);
   add(CAN_ONLY);
   add(ONLY_CAN);
}}));
packageRules.add(new Rule("access", new ArrayList<Predicate>() {{
   add(MUST);
   add(CANNOT);
   add(CAN_ONLY);
   add(ONLY_CAN);
}}));
VariableType pack = new VariableType("Package", packageArgs, packageRules);
```

The code above creates the variable type “Package” that requires a “name” attribute and can be used with the two predicates “access” and “depend on”. After creating all variable types we pass them to the constructor of the `Environment` class as a list:

```
Environment env = new Environment(variables, types);
```

The `variables` parameter is also a list and should be empty. But it provides a way of adding predefined variables.

7.5 Using the state machine

Now we have our environment and state machine, the next step is to give it input. The `StateMachine` class provides the `run` method for this purpose. It takes a `Context` and a `Environment` object. and returns a `StateMachineResult` object. The code below shows how to use the state machine properly.

```
final Environment env = ...
final StateMachine stateMachine = ...
StateMachineResult result = stateMachine.run(context, env.copy());
```

Note that the state machine is passed a copy of the original environment. The state machine modifies the environment while running. So the `Environment` is implemented as a prototype to ensure reusability of

the same object. The state machine is completely stateless and can be run multiple times even at the same time. The received result is either an instance of `StateMachineError` or `StateMachineSuccess`. The underlying interface provides methods to retrieve either an error message via `getErrorMessage()` or a list of suggestions via `getSuggestions()`.

7.6 Server

I implemented a basic server accepting HTTP POST requests. For this purpose, I used the Java Spark library[4]. It is a Sinatra[1] style HTTP web server in combination with a JSON library[2] provided by Eclipse. The web service expects the Dicto code to be the body of the HTTP POST request. POST was chosen over GET because most HTTP servers limit the data size a GET request can have. Returned will be a list of suggestions as JSON. The return format could be easier, but I used JSON straight away because the return format could be enhanced easily. In the future each element could be accompanied by meta data giving further insight to the writer. The basic web server is the `ch.unibe.dicto.WebService`[3] class located in the server project.

7.7 Demo

A small demo can also be found in the repository. It uses a Javascript library called Ace[2]. It provides tools to implement code editors on web sites. The demo sets up a very basic version and sends requests to a server. It can be tested by running the Java server and then simply opening the HTML file in a web browser.

[1] `http://www.sinatrarb.com/`

[2] `http://eclipsesource.com/blogs/2013/04/18/minimal-json-parser-for-java/`

[3] `http://github.com/grushnakk/DAWS/blob/master/server/src/ch/unibe/dicto/WebService.java`

Bibliography

[1] Microsoft - Using IntelliSense https://msdn.microsoft.com/en-us/library/hcw1s69b.aspx 12 06 2016

[2] Ace - The high performance code editor for the web https://ace.c9.io 12 06 2016

[3] Jakob Nielsen - ResponseT Times: The 3 Important Limits https://www.nngroup.com/articles/response-times-3-important-limits/ 12 06 2016

[4] Spark - A micro framework for creating web applications in Java 8 with minimal effort http://sparkjava.com 12 06 2016

[5] PetitParser Java https://github.com/petitparser/java-petitparser 12 06 2016

[6] Andrea Caracciolo, Mircea Filip Lungu and Oscar Nierstrasz - A Unified Approach to Architecture Conformance Checking http://scg.unibe.ch/archive/papers/Cara15b.pdf 12 06 2016