Bachelor's Thesis Computing Science

Model Learning of Lexers

A study about model learning 'black-box' lexers

Rowan van Rooijen
s1079968

August 18, 2024

First supervisors/assessors:
Dr. Mart Lubbers & Dr. Jurriaan Rot

Second assessor:
Prof. dr. Frits Vaandraager

Radboud University

Abstract

Lexers are crucial components of compilers. In lexers, finite state machines play a significant role. In this thesis, we approach the lexer as a black box and utilize a technique called Model Learning to learn the underlying automata. We start with a general overview of lexers where we discuss their role in compilers, how to generate one in C by using the tool `lex` and how we can see them as state machines using transducers. We then give an overview of model learning by talking about the goal and purpose, discussing the L^* algorithm and how to use it in practice using a Java library named LearnLib. Finally, we apply the knowledge from the previous chapter in a practical prototype where we learn the automata from a simple lexer. This research successfully produces a prototype that learns the finite state machines of a lexer. We also find that we can learn a Mealy machine and are able to transform this into a transducer.

Contents

Chapter 1

Introduction

Lexers are an important component of compilers, they are responsible for the first step of compiling the code, turning code into tokens. Lexers segments the input into words whereas the parser turns the words into sentences. These words are called Tokens. Tokens are a data structure that the parser uses to analyze and interpret the sentences. Lexers are often generated by a lexer generator, one such is a tool called `lex` for C. The code underlying a lexer is often based on finite state machines. Finite state machines allow lexers to fast and efficiently segment the input into the output words, hence finite state machines play a significant role in this part of the compiler.

Another important theory for this research is Model Learning. Model Learning is a process that is used to reverse engineer a machine back into a state machine [13]. There are many algorithms developed to do this, one of the more famous ones is L^* by Angluin [1].

Now let us say we have a lexer and want to study the underlying automaton to understand the lexer or improve it, but do not have the source code available, then you can use Model learning to learn the automaton of the lexer. In this thesis, we will make use of LearnLib [8], a Java Library for model learning, and `lex` [9]. By treating the generated code of lex as a *black-box* lexer we learn the model behind this lexer. This model can then be used to verify whether the lexer behaves as it should.

The answer to this problem helps other researchers and developers in improving and further studying lexers. Using the learned state machines of lexers we can analyze the behaviour of lexers in more detail and use this to find bugs or study the behavior of lexers where the source is not available.

This thesis consists of eight chapters, we start with Chapter 2 which covers the preliminaries, here we discuss the basics of finite-state machines and transducers. In Chapter 3 we cover lexers, how to generate lexers in C and how to see lexers as transducers. In Chapter 4 we discuss Model learning by talking about the theory of model learning, the L^* algorithm and the LearnLib library. In Chapter 5 we apply the theory discussed in the previous chapters to build a prototype able to learn the state machine underlying a lexer whilst treating the generated code as a black box. Finally, in Chapter 6, and Chapter 7 we discuss related work, conclusions and future work.

Chapter 2

Preliminaries

In this chapter, we review the basics of finite-state transducers and Mealy machines. We start by going over the definition of finite-state transducers and elaborate upon them using an example. After that we discuss Mealy machines, which are a special kind of transducers.

2.1 Finite-state transducers

A finite state transducer is an extension of the deterministic finite automaton (DFA) [11]. A DFA over alphabet A is defined as the tuple (Q, q_0, F, δ) where:

– Q is a finite set of states.

– $q_0 \in Q$ is the initial state.

– $F \subseteq Q$ is the set of final states.

– δ: $Q \times A \to Q$ is the transition function.

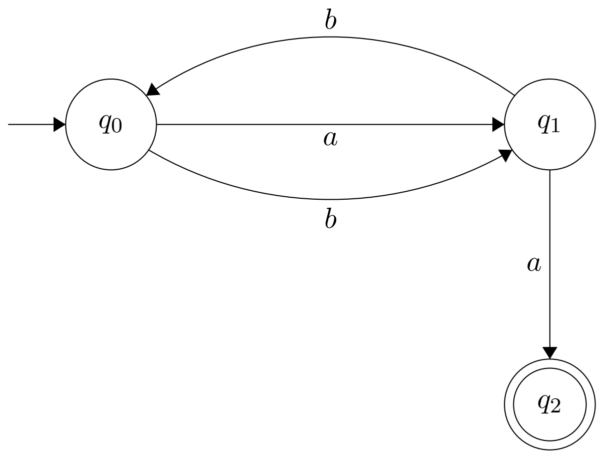

Figure 2.1: An example of a DFA.

With the definition above a DFA only processes an input alphabet and does not produce any output. This is where the difference with a finite-state transducer comes in. A transducer is similar to a DFA. It has states with transitions which are labelled with letters from the alphabet. A transducer however has two alphabets, an input and an output alphabet. Because transducers are also deterministic there cannot be multiple transitions with the same character or a transition with the empty word λ. Each transition is also labelled with an output word, which does not have to be unique for the transitions leaving a state. The output label can also be the empty word λ, in this case, the label is often omitted. So is it not possible to have two transitions both with input label a leaving state q_1, but two transitions with output label a leaving from q_1 is allowed. The output alphabet can be different from the input alphabet.

Formally, a transducer T is defined as the tuple $(Q, q_0, F, \Sigma, \Delta, \delta, \sigma)$ where:

- Q is a finite set of states.
- $q_0 \in Q$ is the initial state.
- $F \subseteq Q$ is the set of final states.
- Σ and Δ are the input and output alphabets respectively.
- δ: $Q \times \Sigma \rightarrow Q$ is the state transition function.
- σ: $Q \times \Sigma \rightarrow \Delta^*$ is the output function.

With this definition, we can explain the transducer in Figure 2.2. In this transducer, we have the following values:

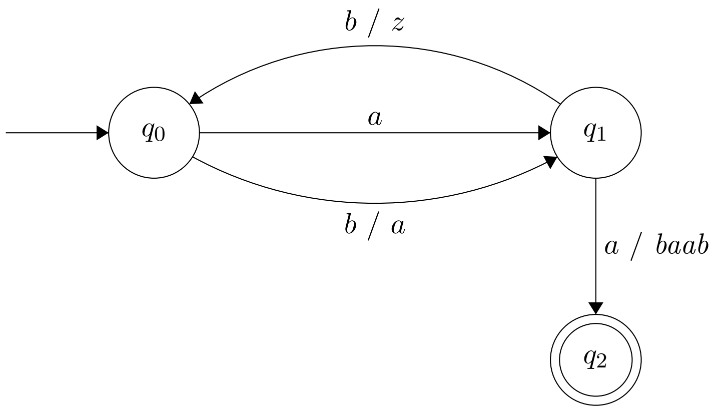
$$\begin{tikzcd}
\arrow[r] & q_0 \arrow[rr, "a"] \arrow[rr, bend right, "b / a"'] & & q_1 \arrow[ll, bend right, "b / z"'] \arrow[d, "a / baab"] \\
& & & q_2
\end{tikzcd}$$

Figure 2.2: An example of a transducer.

- $Q = \{q_0, q_1, q_2\}$
- $i = q_0$
- $F = \{q_2\}$
- $\Sigma = \{a, b\}$
- $\Delta = \{a, b, z\}$

From the initial state q_0 we can see two outgoing transitions. For the transition labelled with a, nothing is written to the output, for the transition labelled with b an a is written. From q_1 we can see two more transitions, note that for the transition labelled with b z is written, z is not in the input alphabet. Finally, we can look at the transition labelled a coming from q_1, when this transition is performed, the character sequence *baab* is written to the output. With this sequence, we can see the property of a transducer being able to write multiple characters to the output in a single transition. When the final state is reached, for lexers this is often after an end of file is sent to the lexer, the automaton behaves the same as a DFA. Unlike a regular DFA, a finite-state transducer computes an output word.

2.1.1 Mealy Machines

A Mealy Machine is a special kind of transducer. A transducer is a mealy machine when it complies with the following two conditions:

1. The set of final states, i.e. $F = \emptyset$
2. The output function σ maps to a word of exactly one character of length, i.e. there cannot be a transition which adds zero or more than one character to the output.

With this, we can define a Mealy machine with the following tuple $(Q, i, \Sigma, \Delta, \delta, \gamma)$ where:

- Q is a finite set of states.
- $i \in Q$ is the initial state.
- Σ and Δ: are the input and output alphabets respectively.
- δ: $Q \times \Sigma \to Q$ is the state transition function.
- σ: $Q \times \Sigma \to \Delta$ is the output function.

Note that with a Mealy machine, it is not possible to write the empty word.

Chapter 3

Lexers

In this chapter, we first lay the foundation by explaining the purpose and use of lexers. Here we talk about their role in compilers, how they generally work and their importance. After that, we focus on how to generate a lexer in C using the tool `lex` and how to use it for basic tasks by using examples. Finally, we talk about how we can see lexers as a finite state machine using transducers.

3.1 Lexers

In computer science, lexers are often used in compiler design [5]. Simply said, a lexer is a tool used in the parsing phase of a compiler. Parsing code can be split up into two parts, turning code into tokens, and tokens into a program. A token is a data structure that the compiler can use to simplify parsing the code.

Let us say we have a simple line of Java code:

```
String hello = "world";
```

We can parse this string using a parser only as parsers can parse context-free grammers. This would mean the parser needs to handle each character and deal with white spaces. This process, however, becomes very complex rather quickly and makes the compiler difficult to maintain. Making use of a lexer makes this process easier. Instead of directly parsing the string, we first turn it into tokens. For this line instead of letting the parser find out the tokens and have to process each of the 23 characters, we can use a lexer which would result in only five tokens remaining:

1. Identifier(value=String)
2. Identifier(value=hello)
3. =

4. StringLiteral(stringCharacters=world)

5. ;

As you can see, not important parts of the code, like whitespace, are completely left out and do not have to be dealt with by the compiler anymore. With these tokens, the parsing of the code becomes much easier.

Lexers are generally generated by a lexer generator. A lexer generator takes a grammatical definition and turns it into code. Often these tools can perform more tasks than just lexical analysis. For instance, ANTLR4 [12], a language recognition tool in Java, supports lexical analysis but also generates interfaces to parse the input content.

3.2 Lexical Analysis in C

When we want to do lexical processing in C the `lex` [10] tool is frequently used. The program `lex` is a compiler design tool that can be found on all POSIX-compliant systems [9]. The tool makes use of user-defined regular expressions and generates C code to process a character stream using these definitions into meaningful tokens. The `lex` tool is often combined with `yacc` [14], in this duo `lex` performs the task of turning an input stream into tokens and `yacc` the task of parsing using grammar rules. In this research, we only focus on `lex`.

Using lex we define the language in a separate source file. This file exists of three primary parts:

1. First, you start with the definitions. This can include C code which is placed at the top of the generated file. To use C code in this section you have to surround it with `%{ %}`.

2. Second, you define the lexer rules itself.

3. Third, you define user subroutines. These include functions `lex` specific functions like `int yywrap()`. Additionally, you can define the `int main()` here if you want to be able to compile the generated source code directly.

These three parts are separated by the delimiter `%%`. An example of such a ruleset is:

```
%{
  enum Token {
    ERROR = -1,
    NUMBER = 1,
    WHITESPACE = 2,
    IDENT = 3
  };
%}

DIGIT [0-9]
LETTER [a-zA-Z]

%%
DIGIT* { return NUMBER; }
[ \t]     { return WHITESPACE; }
[a-z]LETTER* { return IDENT; }
%%

int yywrap()
{
    return 1;
}
```

In this example, we can see the three parts of a lex program. First, we see the definition of an enum using C code. C code wrapped with `%{ %}` in this section. Secondly, we see two aliases defined, DIGIT and LETTER. After that, we see the `%%` delimiter indicating the start of the lex rules. In this next section three lex rules are defined, two of which use the aliases specified in the first section. After each rule we can see more C code between curly braces, this C code is executed when the rule is matched. After the last rule we see another `%%` delimiter indicating the end of the lex rules and the start of the last section. In the last section we can see the `yywrap` subroutine. `yywrap` is a function which determines whether the end of the input is found.

The three rules in this ruleset, match the following:

- `DIGIT*` which is an alias for `[0-9]*` is returned for a number.
- `[ \t]` is matched for a space or tab character.
- `[a-z]LETTER*` which is an alias for `[a-z][a-zA-Z]*` is matched for a string starting with lowercase.

If we give the following input to this lexer

```
number is 123
```

we get the following as a result:

```
IDENT
WHITESPACE
IDENT
WHITESPACE
NUMBER
```

3.3 Lexer as Transducer

The automaton behind a lexer can be best compared with a transducer. A transducer is a kind of automaton that is able to write any amount of characters of the output alphabet whilst reading one character from the input alphabet. Let us say we have the following three rules for a lexer:

```
[1-9][0-9]* { return NUMBER; }
[a-z][a-zA-Z]* { return IDENT; }
[ ] { return WHITESPACE; }
```

We can construct a transducer as follows:

- First we start by determining our input alphabet, this alphabet consists of all characters that this lexer can read. For these three rules the alphabet consist of $\{a..z\} \cup \{A..Z\} \cup \{0..9\} \cup \{' \;\; '\}$.
- Secondly we determine the output alphabet, which consists of all possible output tokens. For this lexer, the alphabet can be defined as $\{N, I, W\}$. For convenience, the items of the alphabet are abbreviated to the first letter of the tokens.
- Thirdly, we define an initial state q_0. From this state, we create all rules.
- Finally, we create the states and transitions for the rules.

To start off making the transitions we start with the easiest one, the whitespace. When the lexer reads a whitespace from the first state, it instantly writes a W token and remains in this state. The other rules are more com-

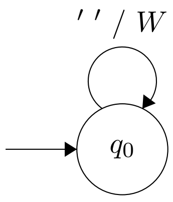

Figure 3.1: The states and transitions for the whitespace rule.

plex. For rules with two or more characters, we have to make an intermediary state as the transducer is only able to read a single character at once. If we look at the number rule in the lexer we first have to read a number between one and nine, followed by zero or more numbers between zero and nine. To convey this to transitions we first have to make a transition for `[1-9]` to intermediate state q_1 and from there we can read any amount of `[0-9]` by creating a loop. Both these transitions do not write a token yet, this is because the token should only be written once the entire number has been read. This happens in two cases:

1. A whitespace is read, or
2. a letter is read.

For the whitespace it is simple, we read the whitespace and write N and W to the output. Because we are back at the state where the lexer is not in the process of reading a token we go back to q_0.

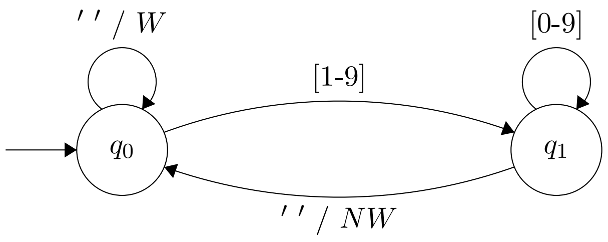

Figure 3.2: The states and transitions for number rule.

When a letter is read we start reading a new ident. Before we can add this transition we add the transitions from the initial state for reading an ident.

For reading an ident the intermediate state and transitions are similar to when reading a number. We start by reading a lowercase letter, followed by any amount of letters, either lower or uppercase. An ident is entirely read in one of the two cases:

1. A whitespace is read, or
2. a number is read.

For the whitespace, the transition is also similar.

Now we can add the transitions for the second case. To start off when the lexer is reading a number, if a lowercase letter is read we first write N to the output, and then start reading an ident. This is done by adding a transition from q_1 to q_2 labelled with `[a-z]` which writes N to the output. The same is done when the lexer is reading a letter. If a digit from one to nine is read, write I to the output and start reading a number.

With these transitions added to the transducer, we can read the entire language.

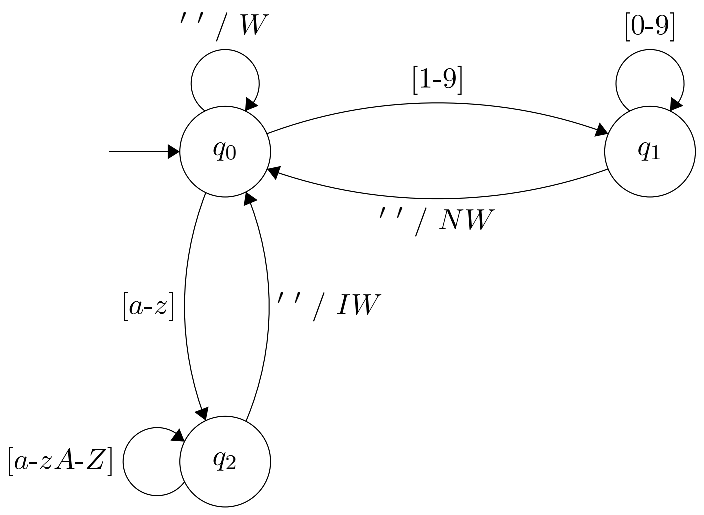

Figure 3.3: The states and transitions for the letter rule.

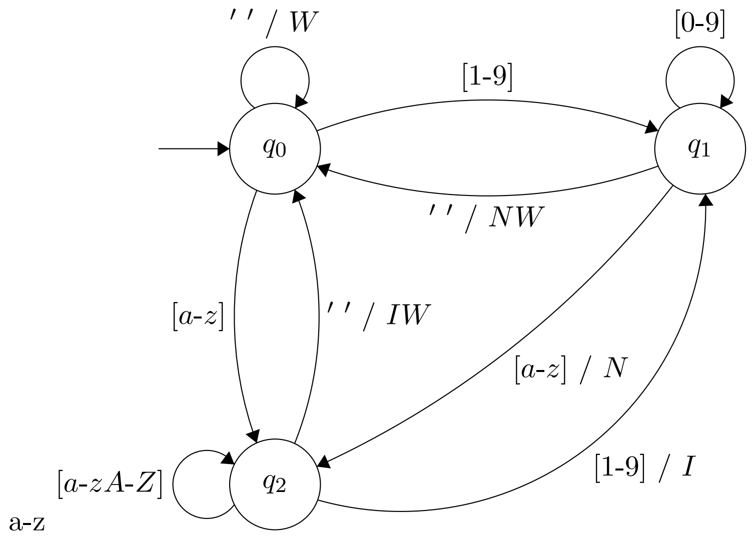

Figure 3.4: The full transducer of the example lexer.

Chapter 4

Model Learning

In this chapter, we go over the foundation of model learning. We first discuss the goal and purpose of model learning followed by the L^* algorithm. Finally, we talk about LearnLib, a Java library for model learning.

4.1 Model learning

Whilst the concept of state machines is known to many, how to reverse engineer a machine or software component back into a state machine is not. Model Learning [13] can be seen as reverse engineering a (black-box) machine back into a state diagram. An article published by Angluin [1] in 1987 shows that finite automata can be built using two types of queries: membership and equivalence queries. These two queries can both be implemented on a black-box lexer which is the oracle that can answer these queries. The first query, the membership query supplied with a string t, is answered with *yes* or *no* based on whether t is a member of the set being learned. The second query, the equivalence query supplied with a regular set S answers *yes* if S is equivalent to the automata being learned, or *no* together with a counter-example if this is not the case. These queries can be structured using many algorithms.

4.2 Angluin's L^*

The pioneer of learning algorithms is Angluin's L^* [1]. L^* learns the automaton underlying a machine by keeping track of an observation table and populating it by asking membership queries and testing the contents of the table with equivalence queries. In L^* we know the input alphabet and use this to make queries to learn the automata. In an observation table, L^* keeps information to classify a finite collection of strings over the input alphabet as members or nonmembers of the automata.

An observation table consists of three columns, a non-empty set of prefixes, A non-empty finite prefix-closed set S of strings, a non-empty set of suffixes and a finite function T mapping the union of the prefixes and the dot product between the prefixes and the input alphabet to 1 or 0. The goal is to build a hypothesis that can be validated by using a membership query. To build such a hypothesis, the current observation table needs to fulfil two properties, namely *closed* and *consistent* [1]. When this is the case, we can build a hypothesis. To do this we need to build the state machine as follows:

- We populate Q with each row.
- We define q_0, the initial state, as the row with the empty word.
- We populate F, the final states, with the rows where the transition function results in 1.
- We populate the transitions $\delta(row(s), a) = row(s \cdot a)$ where the state with string s where character a is added transitions to the state of s with a appended.

This will result in a well-defined hypothesis as proven in Angluin's paper. With this hypothesis, we can ask the oracle an equivalence query. If it is equivalent, we have learned the underlying automata of this machine, otherwise, a counter-example is returned and is added to the observation table using membership queries. This is also proven correct because the teacher answers whether the proposed automata is equivalent to the one to be learned.

4.3 LearnLib

To be able to make use of model learning, a few popular libraries exist. One of these libraries is LearnLib [8]. LearnLib is an open-source framework for automata learning in Java. It features many algorithms to learn various types of machines including L^* and Mealy Machines. To learn an automata from an unknown machine you have to supply LearnLib with the input alphabet. This alphabet can be a set of functions and parameters. Let us say you want to learn the automata behind a class in Java that represents a bounded string queue . The class has two methods:

```java
public void offer(String s)
```

to add a string to the queue and

```java
public @Nullable String poll()
```

to get the next element in the queue. We can then provide these methods using a `SimplePOJOTestDriver`, which stands for a Simple Plain Old

Java Objects Test Driver. We can instantiate it by supplying the class as parameter:

```
SimplePOJOTestDriver driver =
            new SimplePOJOTestDriver(BoundedStringQueue.class);
```

After we instantiated it we can get two `Method` objects using Java's reflection library:

```
Method mOffer =
   BoundedStringQueue.class.getMethod("offer", String.class);
Method mPoll =
   BoundedStringQueue.class.getMethod("poll");
```

This specific driver outputs a MethodInput, which is a class to handle the in- and output of methods, when adding the methods to the driver:

```
// offer
MethodInput offerA = driver.addInput("offer_a", mOffer, "a");
MethodInput offerB = driver.addInput("offer_b", mOffer, "b");

// poll
MethodInput poll = driver.addInput("poll", mPoll);
```

These methods will be the *input* alphabet of the learner. The output alphabet will be the outputs of these functions. To actually learn the automata behind the class we have to create an *system under learning* (SUL) . An SUL is used to make single steps in the various learning algorithms. To learn the `BoundedStringQueue` class, we will be using a ResetCounterSUL which is an SUL which can also reset it's counter, this is useful when the teacher gives a counter-example. We can instantiate the SUL as follows:

```
StatisticSUL<MethodInput, MethodOutput> statisticSul =
        new ResetCounterSUL<>("membership queries", driver);
```

LearnLib also supports caching of the SUL, this allows the SUL to store the membership queries done and save resources when the same membership query is performed again. An example of how to do this can be found in Appendix A.2. Next to the SUL for the learner, we also need to define the SUL of the teacher. We can do this by using the wrapper class SULOracle . This wrapper class will later be used to determine what kind of machine we are going to learn. We can define it as follows:

```
SULOracle<MethodInput, MethodOutput> mqOracle =
        new SULOracle<>(effectiveSul);
```

After this is done we need to define our set of initial suffixes. We can do this by using the *words* added to the input alphabet earlier.

```
List<Word<MethodInput>> suffixes = new ArrayList<>();
suffixes.add(Word.fromSymbols(offerA));
suffixes.add(Word.fromSymbols(offerB));
suffixes.add(Word.fromSymbols(poll));
```

Now we are going to define the learner. LearnLib provides a wide range of implementations for different learning algorithms which can be found in the implementations of the LearningAlgorithm interface. In this example we'll make use of the L^* algorithm thus will use the ExtensibleLStarMealy class. This class is a subclass of the MealyLearner, there are also variants for L^* for DFA's and Moore machines. We can define the Learner as follows:

```
MealyLearner<MethodInput, MethodOutput> lstar =
  new ExtensibleLStarMealyBuilder<MethodInput, MethodOutput>()
    .withAlphabet(driver.getInputs())
```

Alongside the Learner we also have to define the teacher, as we want to learn a Mealy machine we make use of a MealyEquivalenceOracle . LearnLib also provides different implementations of the teacher using different algorithms for checking the proposed automata. In this example we will make use of the RandomWalkEQOracle . This teacher validates the proposed model by randomly walking over the automata until it has found a counter-example or walked a fixed number of steps. We can define this as follows:

```
MealyEquivalenceOracle<MethodInput, MethodOutput> randomWalks =
    new RandomWalkEQOracle<>(driver,
                             RESET_PROBABILITY,
                             MAX_STEPS,
                             false,
                             new Random(RANDOM_SEED)
                );
```

As the final step, we have to define the learning experiment. LearnLib provides two kinds of experiments, a DFAExperiment and a MealyExperiment . As we want to learn a MealyMachine we make use of the latter one. To define the experiment we supply it with the learner, the oracle and the input alphabet as follows:

```
MealyExperiment<MethodInput, MethodOutput> experiment =
        new MealyExperiment<>(
           lstar,
           randomWalks,
           driver.getInputs()
        );
```

Now we can run the experiment by using the `experiment.run()` method. After running the experiment with a `BoundedStringQueue` with a maximum

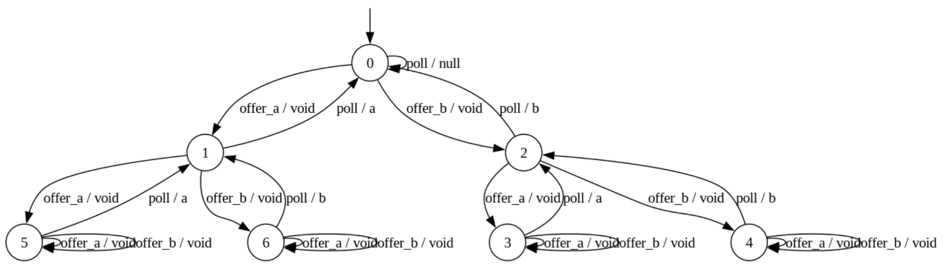

Figure 4.1: The learned automata of BoundedStringQueue with max size of two.

size of two we get the automata in Figure 4.1 as the result. In Figure 4.1, we can see the following from the class we learned. From the initial state 0 we have three outgoing transitions, one for each method we can call on this class. We can `poll`, `offer_a` or `offer_b`. In the initial case the list is empty, so polling returns `null`. We can offer either an `a` or `b`. When offering an `a` we transition into state 1. From this state we can do the same three methods, however, when polling instead of `null` being returned the offered `a` is returned. After we polled from the list, the item is removed and we transition back into state 0. Lastly, the offer does not change the state if we offer when the list has two items in it already. This correlates with the behavior of a list with a size limit, which the class we learned is an example of.

Chapter 5

Experiment & Prototype

In this chapter we start by going over the foundation of the implementation and how we call the generated C code by `lex` in Java. After that, we describe the steps to give LearnLib the input alphabet to learn the state machine behind the lexer. Finally, we generate a state machine for an example lexer.

5.1 Interacting with the lexer in Java

The `lex` tool generates C code, however, the model learning framework we want to use, LearnLib, uses Java. To be able to interact with the C code in Java, we make use of the Java Native Interface [6]. This Java interface allows us to interact with C methods and structures as Java objects and methods. To make use of this interface we use the `native` keyword provided in Java. After creating a class with native methods in it we can use the `-h` flag to create the C header file. Using this we can create an implementation for the method in the class. For the lexer generated in C, we make use of a simple class with three native methods in it, one to create a new lexer instance, one to close the lexer and finally, one to send one character to the lexer. The code for this class is found in Appendix B.1.

When using the methods want to receive the lexed tokens as output. Because it is not guaranteed that a token is given after each input character we cannot simply wait for a token to be returned, instead, we have to wait until the lexer is done processing. To do this we make use of a separate thread where the lexer is run from. When a new lexer is created using the `Lexer.create()` method we start such a thread. To lex a character we use a custom `lex_one_chart(struct lex_state *cookie, int c)` function where we read the lexer state before returning the tokens.

The returned object can consist of none, one or two tokens. To be able to return this we make use of a struct that has a `first` and `second` field. This struct is converted to a `Tokens` record in Java which then can be used in LearnLib.

5.2 Setting up LearnLib

Now we can interact with the lexer in Java, we have to set up LearnLib to be able to learn the state machine behind the lexer. To do this we define a `LexerDriver` class that extends the `ContextExecutableInputSUL` class. This class is used for learning a language where the SUL requires context, in this case, the lexer instance, and an executable input, which are the different characters in the input alphabet. Because we make use of a context and input, we also have to implement a `ContextHandler` as well as a `ContextExecutableInput`.

To create the `ContextHandler` for the `Lexer` class we have to implement two methods:

- `public Lexer createContext()`
- `public void disposeContext(Lexer lexer)`

Implementing these methods is easy as we can simply use `Lexer.create()` and `lexer.close()`. The full class can be found in Appendix C.1.

Next, we have to implement the `ContextExecutableInput`. This class takes two generic arguments, the output type and the context type. For the output type, we have a `String` consisting of which tokens are written by the lexer, as for the context type we have the `Lexer` class. To implement this class we have to implement one method:

- `public String execute(Lexer lexer)`

We implement this method by using the `lexer.lex(char c)` method provided by the `Lexer` class. This method takes a character to be given to the lexer and outputs a `Tokens` object that consists of zero, one or two output tokens. To turn this `Tokens` object to a string we make use of a `token.toStringWithDict(Map<Integer, String> tokenDict)` function. This makes a string from the two tokens and allows us to provide a dictionary to translate the numeric token given by the lexer into a human-readable version of this token. The code for this class can be found in Appendix C.2.

Lastly, we implement the `ContextExecutableInputSUL` which is the driver for learning this language. This driver handles creating and disposing of the context and executing the inputs in this context. the `ContextExecutableInputSUL` takes three generic arguments:

- Firstly, the input type, which is `LexerInput` for learning a lexer.
- Secondly, the output type, which is `String` as it has to match the output of the `LexerInput`.
- And lastly, the context type, which is `Lexer`.

To use this driver we have to instantiate the superclass using an instance of the context handler. Besides using this class to create and dispose of context, we can use it to build the input alphabet. We can do this by adding a list of symbols to this class. The code for this class can be found in Appendix C.3.

We can now use these classes to learn a Mealy machine as described in Section 4.3.

5.3 State machine from lexer

To finally learn a state machine from the lexer we first have to define the input library. We can do this by using the lexer defined in the previous section. First, we have to give the driver defined in the previous section our input alphabet. To make the final state machine a little easier to grasp we only use one letter and number, and the characters to open and close a comment.

```
LexerDriver driver = new LexerDriver();

driver.addSymbol('a');
driver.addSymbol('1');
driver.addSymbol('/');
driver.addSymbol('*');
driver.addSymbol(' ');
```

We then this driver as explained in Section 4.3 and start the experiment.

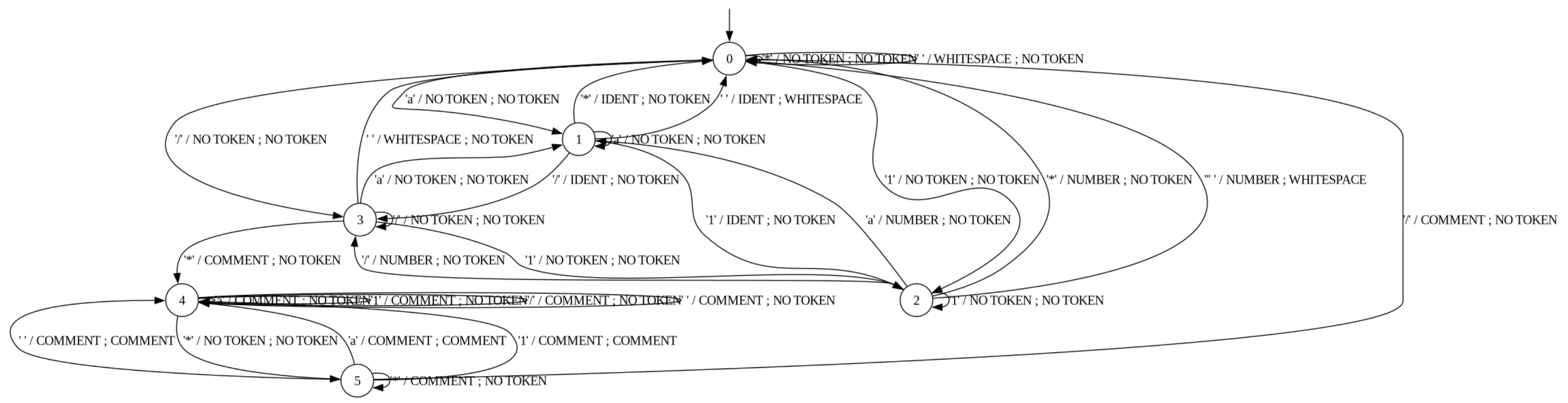

Figure 5.1: Learned State Machine from Lexer

In Figure 5.1 we can see the state machine learned. The state machine we learned is not a transducer but a Mealy machine.

5.4 Mealy Machine to Transducer

To turn the learned Mealy machine into a transducer we use the properties of a lexer: The fact that it outputs at most two tokens at once. Using this property we can transform a Mealy Machine into a transducer using the following steps:

- Replace the combined characters with a word containing those characters.
- Replace the empty character `NO_OUTPUT` with an empty word λ.

Applying these steps to the Mealy machine from Figure 5.1 with output alphabet $\{N, W, I, S, C\}$ to denote numbers, whitespace, idents, strings and comments respectively the following changes:

- `NO TOKEN ; NO TOKEN` becomes writing nothing.
- `WHITESPACE ; NO TOKEN` becomes `W`
- `IDENT ; NO TOKEN` becomes `I`
- `IDENT ; WHITESPACE` becomes `IW`
- `NUMBER ; NO TOKEN` becomes `N`
- `NUMBER ; WHITESPACE` becomes `NW`
- `COMMENT; NO TOKEN` becomes `C`
- `WHITESPACE ; NO TOKEN` becomes `W`
- `COMMENT ; COMMENT` becomes `CC`

Applying these changes to the state machine we get Figure 5.2 as a result.

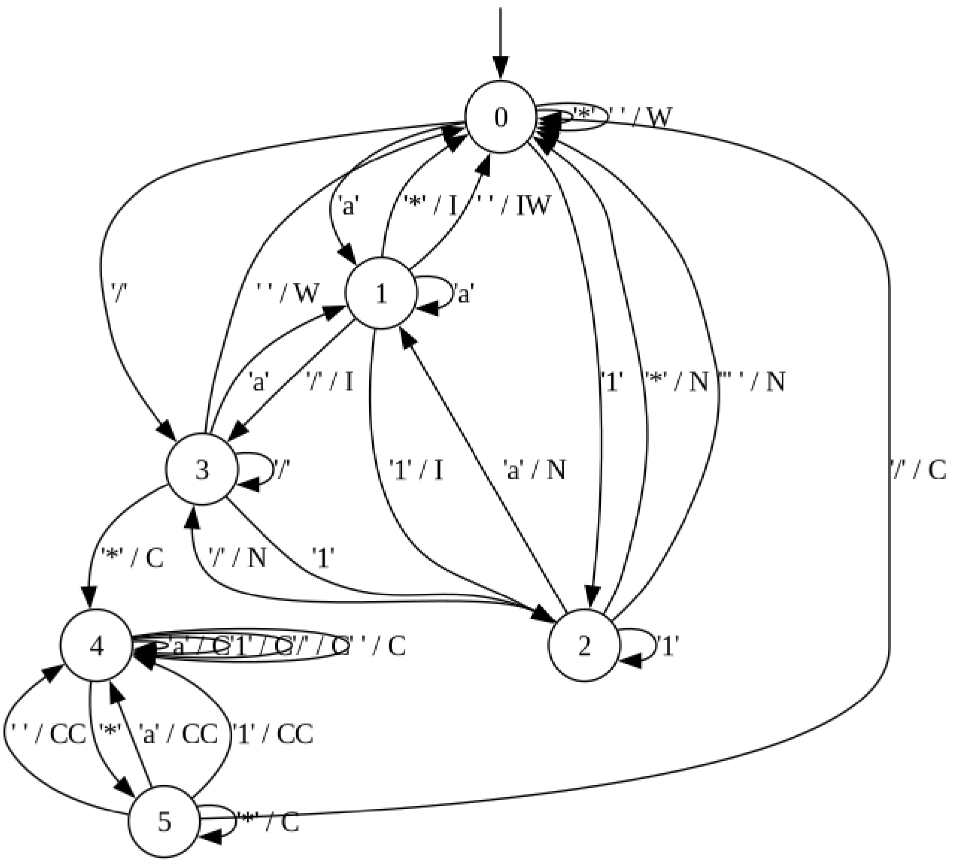

Figure 5.2: The learned state machine from the lexer.

Chapter 6

Related Work

There are many papers which discuss case studies on model learning, however none focus on lexers. A paper authored by Vaandraager [13] about model learning focuses on the theory of model learning and discusses examples of applications from other papers such as smartcards, network protocols and legacy software. Another case study published by Bastini et al. [2] gets closer to the topic of learning lexers and discusses the GLADE algorithm that can be used to learn context-free grammar from a black-box oracle that answers membership queries. This paper does only cover grammar and does not talk about lexing.

There are also papers about model learning of transducers. Take the paper about active learning of sequential transducers by Berthon et al. [3]. In this paper, they show that there exists an algorithm to learn subsequential string transducers and accelerate this process by using knowledge about the domain. Whilst this does touch the area of learning transducers, the focus in this paper is developing a learning algorithm that can use the domain information rather than applying one software component or machine that can be interpreted as a transducer such as a lexer. Another paper that develops an algorithm for learning transducers is the paper about Σ^* [4]. The Σ^* algorithm can be seen as an extension of L^* but with two ideas added onto it. Σ^* is able to discover the input alphabet and instead of equivalence checking against the program, the algorithm builds an over-approximation that Σ^* can use to check whether the proposed algorithm is equivalent or not. Whilst this can be used to learn lexers, it does not show that this algorithm can be used on lexers.

Another topic that relates to this paper is the visualization of the automata of lexers. A paper by Jorgensen et al. about a tool they created named Vlex [7] touches on visualising a lexer's state machine. This tool does that by supplying it with the rules of the lexer. It does not support reading the lexer rules from the lex files or the generated output making it different from this thesis. The paper about Vlex focuses on helping students understand the concept of lexers and not on how you can learn the state machines from a black-box lexer.

Chapter 7

Conclusions

In this study, we investigated the feasibility of model learning lexers and have successfully built a prototype. From this, we conclude that learning the automata underlying lexers is feasible when treating the source code as a 'black box'. Using the LearnLib library we have learned the underlying automaton of a lexer. The library does not have access to the source code and can learn the automata by sending single characters to the lexer and observing the corresponding outputs.

Through this process, we have shown that model learning can accurately learn the automaton of a lexer. This contributes to the field of compiler design. The ability to study the automata of lexers without access to the source code helps with more efficient debugging, optimization, and further analysis of lexers. Additionally, this approach can potentially benefit the automated testing of these software components.

Several areas for future work can be taken from this research. This thesis revealed that model learning of finite state transducers is not implemented in LearnLib and currently requires an intermediary Mealy machine. Future work can focus on applying the work on learning finite state transducers in LearnLib. Papers that can be used to achieve this could be the Σ^* algorithm by Botinčan and Babić [4] or the paper about learning sequential transducers with side information about the domain by Berthon et al. [3].

Another area for future work is the analysis of lexers where the source code is unavailable. By creating a Java class that interacts with the binary of the lexer. This aids the study and verification of legacy lexers, researchers can apply these model learning concepts to the binaries of lexers. This can provide insights into their functionality.

Future research can also focus on applying the results of this thesis to bigger and more complex lexers. Investigating how the concepts of this research perform with more sophisticated lexers could reveal scalability issues and potential optimizations. This would help in understanding the limitations and strengths of this approach.

Bibliography

[1] Dana Angluin. "Learning regular sets from queries and counterexamples". In: *Information and Computation* 75.2 (Nov. 1987), pp. 87–106. ISSN: 0890-5401. DOI: 10.1016/0890-5401(87)90052-6. URL: https://www.sciencedirect.com/science/article/pii/0890540187900526 (visited on 05/15/2024).

[2] Osbert Bastani et al. "Synthesizing program input grammars". In: *SIGPLAN Not.* 52.6 (June 2017), pp. 95–110. ISSN: 0362-1340. DOI: 10.1145/3140587.3062349. (Visited on 06/27/2024).

[3] Raphaël Berthon et al. "Active Learning of Sequential Transducers with Side Information About the Domain". In: *Developments in Language Theory: 25th International Conference, DLT 2021, Porto, Portugal, August 16–20, 2021, Proceedings.* Berlin, Heidelberg: Springer-Verlag, Aug. 2021, pp. 54–65. ISBN: 978-3-030-81507-3. DOI: 10.1007/978-3-030-81508-0_5. (Visited on 07/18/2024).

[4] Matko Botinčan and Domagoj Babić. "Sigma*: symbolic learning of input-output specifications". In: *SIGPLAN Not.* 48.1 (Jan. 2013), pp. 443–456. ISSN: 0362-1340. DOI: 10.1145/2480359.2429123. (Visited on 07/18/2024).

[5] Bruce Hahne and Hiroyuki Satō. "Using yacc and lex with C++". en. In: *ACM SIGPLAN Notices* 29.12 (Dec. 1994), pp. 94–103. ISSN: 0362-1340, 1558-1160. DOI: 10.1145/193209.193231. (Visited on 05/14/2024).

[6] S. Fouzi Husaini. "Using the Java Native Interface". en. In: *XRDS: Crossroads, The ACM Magazine for Students* 4.2 (Nov. 1997), pp. 18–23. ISSN: 1528-4972, 1528-4980. DOI: 10.1145/332100.332105. (Visited on 06/17/2024).

[7] Alisdair Jorgensen, Rob Economopoulos, and Bernd Fischer. "VLex: visualizing a lexical analyzer generator – tool demonstration". In: *Proceedings of the Eleventh Workshop on Language Descriptions, Tools and Applications.* LDTA '11. New York, NY, USA: Association for Computing Machinery, Mar. 2011, pp. 1–6. ISBN: 978-1-4503-0665-2. DOI: 10.1145/1988783.1988795. (Visited on 05/14/2024).

[8] *LearnLib.* en. URL: https://learnlib.de/ (visited on 06/01/2024).

[9] M. E. Lesk and E. Schmidt. “Lex—a lexical analyzer generator”. In: *UNIX Vol. II: research system (10th ed.)* USA: W. B. Saunders Company, Mar. 1990, pp. 375–387. ISBN: 978-0-03-047529-0. (Visited on 08/11/2024).

[10] *lex(1p) - Linux manual page.* URL: https://man7.org/linux/man-pages/man1/lex.1p.html (visited on 03/26/2024).

[11] Mehryar Mohri. “Finite-state transducers in language and speech processing”. In: *Computational Linguistics* 23.2 (June 1997), pp. 269–311. ISSN: 0891-2017.

[12] Terence Parr. *The Definitive ANTLR 4 Reference.* 2nd ed. Raleigh, NC: Pragmatic Bookshelf, 2013. ISBN: 978-1-93435-699-9. URL: https://www.safaribooksonline.com/library/view/the-definitive-antlr/9781941222621/.

[13] Frits Vaandrager. “Model learning”. en. In: *Communications of the ACM* 60.2 (Jan. 2017), pp. 86–95. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/2967606. (Visited on 05/15/2024).

[14] *yacc(1p) - Linux manual page.* URL: https://man7.org/linux/man-pages/man1/yacc.1p.html (visited on 05/21/2024).

Appendix A

LearnLib Example Code

A.1 Class to learn

```
public static class BoundedStringQueue {
    // capacity
    public static final int MAX_SIZE = 3;
    // storage
    private final Deque<String> data = new ArrayDeque<>(3);
    // add a String to the queue if capacity allows
    public void offer(String s) {
        if (data.size() < MAX_SIZE) {
            data.offerFirst(s);
        }
    }
    // get next element from queue (null for empty queue)
    public @Nullable String poll() {
        return data.poll();
    }
}
```

[1]

[1]This code is adapted from the examples by LearnLib: https://github.com/LearnLib/learnlib/blob/develop/examples/src/main/java/de/learnlib/example/Example2.java

A.2 Caching of SUL

```
// oracle for counting queries wraps sul
StatisticSUL<MethodInput, MethodOutput> statisticSul =
        new ResetCounterSUL<>("membership queries", driver);

SUL<MethodInput, MethodOutput> effectiveSul = statisticSul;
// use caching in order to avoid duplicate queries
effectiveSul =
  SULCaches.createCache(driver.getInputs(), effectiveSul);
```

Appendix B

Java Classes to Interact with Lexer

B.1 Lexer Class

```
public class Lexer {
    static {
        if (!LibraryLoader.load(Lexer.class, "lexer"))
            System.loadLibrary("lexer");
    }

    public native static Lexer create();

    public native void close();

    public native Tokens lex(char input);
}
```

B.2 C code of lexer class

```
#include "jni_Lexer.h"
#include "test.c"
#include "test.h"

struct lex_state *cookie = NULL;

/*
 * Class:     jni_Lexer
 * Method:    create
 * Signature: ()Ljni/Lexer;
 */
```

```c
JNIEXPORT jobject JNICALL Java_jni_Lexer_create
        (JNIEnv *env, jclass lexerClass) {
    if (cookie != NULL) {
        // throw IllegalStateException
        jclass exceptionClass =
          (*env)->FindClass(env, "java/lang/IllegalStateException");
        (*env)->ThrowNew(env, exceptionClass, "Lexer already created");
        return NULL;
    }

    cookie = lexer_setup();
    // Create a new Lexer object
    jmethodID constructor =
      (*env)->GetMethodID(env, lexerClass, "<init>", "()V");
    jobject lexer = (*env)->NewObject(env, lexerClass, constructor);

    return lexer;
};

/*
 * Class:     jni_Lexer
 * Method:    close
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_jni_Lexer_close
        (JNIEnv *env, jobject jobj) {
    lex_one_char(cookie, EOF);
    lexer_destroy(cookie);
    cookie = NULL;
};

/*
 * Class:     jni_Lexer
 * Method:    internalLex
 * Signature: (Ljava/lang/String;)Ljni/Token;
 */
JNIEXPORT jobject JNICALL Java_jni_Lexer_lex
        (JNIEnv *env, jobject obj, jchar jchar){
    char c = (char) jchar;
    struct Tokens tokens = lex_one_char(cookie, c);

    // Find the Tokens class
    jclass tokensClass = (*env)->FindClass(env, "jni/Tokens");
    // Find the constructor that takes two integers
    jmethodID constructor =
      (*env)->GetMethodID(env, tokensClass, "<init>", "(II)V");

    // Create a new Tokens object
    jobject token = (*env)->NewObject(
      env,
      tokensClass,
      constructor,
      tokens.first,
      tokens.second
    );

    return token;
};
```

Appendix C

LearnLib class for the Lexer

C.1 The context handler

```
static class LexerContextHandler implements ContextHandler<Lexer> {
    @Override
    public Lexer createContext() {
        return Lexer.create();
    }

    @Override
    public void disposeContext(Lexer lexer) {
        lexer.close();
    }
}
```

C.2 The lexer input

```
static class LexerInput implements ContextExecutableInput<String, Lexer> {
    public static Map<Integer, String> tokenDict = new HashMap<>();

    private final Character symbol;

    public LexerInput(Character symbol) {
        this.symbol = symbol;
    }

    @Override
    public String execute(Lexer lexer) {
        Tokens token = lexer.lex(symbol);
        return token.toStringWithDict(tokenDict);
    }
    @Override
    public String toString() {
        return String.valueOf(symbol);
    }
}
```

C.3 The lexer driver

```
static class LexerDriver
        extends ContextExecutableInputSUL<LexerInput, String, Lexer> {
    private final List<LexerInput> symbols = new ArrayList<>();

    public LexerDriver() {
        super(new LexerContextHandler());
    }

    public Alphabet<LexerInput> getAlphabet() {
        return Alphabets.fromList(symbols);
    }

    public LexerInput addSymbol(Character symbol) {
        LexerInput input = new LexerInput(symbol);
        symbols.add(input);
        return input;
    }
}
```