Generating Tokenizers with Flat Automata

Hans de Nivelle
School of Engineering and Digital Sciences,
Nazarbayev University, Nur-Sultan City, Kazakhstan
hans.denivelle@nu.edu.kz

Dina Muktubayeva
School of Engineering and Digital Sciences,
Nazarbayev University, Nur-Sultan City, Kazakhstan
dina.muktubayeva@nu.edu.kz

We introduce flat automata for automatic generation of tokenizers. Flat automata are a simple representation of standard finite automata. Using the flat representation, automata can be easily constructed, combined and printed. Due to the use of border functions, flat automata are more compact than standard automata in the case where intervals of characters are attached to transitions, and the standard algorithms on automata are simpler. We give the standard algorithms for tokenizer construction with automata, namely construction using regular operations, determinization, and minimization. We prove their correctness. The algorithms work with intervals of characters, but are not more complicated than their counterparts on single characters. It is easy to generate C++ code from the final deterministic automaton. All procedures have been implemented in C++ and are publicly available. The implementation has been used in applications and in teaching.

1 Introduction

This paper is part of a project to obtain a programming language for the implementation of logical algorithms. Logic is special because its algorithms operate on trees that have many different forms with different subtypes. Algorithms need to distinguish the form of the tree, and take different actions dependent on this form. Intended applications of our language are parts of theorem provers, or interactive verification systems. For the interested reader, we refer to ([11]). Part of this project is to obtain a working compiler. We have looked at existing tools for the generation of the parser and the tokenizer, but none of them fulfilled our needs. In particular, there was no bottom-up parser generation tool available that supports modern C^{++}, and existing tokenizer generation tools are not flexible enough. Existing tokenizer generators like LEX ([9]) and RE2C ([3]) generate the complete tokenizer, which makes them unsuitable for our language. Our language uses Python-style indentation, which requires that the tokenizer must generate a token when the indentation level changes. Detecting a change of indentation level is quite complicated, and it cannot be represented by regular expressions. Lack of flexibility is a general problem, for example C and C^{++} require that the tokenizer has access to type information, so that different tokens can be generated for identifiers that represent a type name or a template name. C^{++}-11 allows use of `>>` to close two template arguments at once (for example in `std::vector<std::pair<int,int>>`. In that case, `>>` must be tokenized as two separate `>`. In order to do this correctly, the tokenizer needs to know if the parser is currently parsing a template argument.

In order to obtain the required flexibility, we created a new implementation that does not generate the complete tokenizer, but which only cuts the input in small chunks, and classifies them by type. We discuss details of our implementation in Section 8. In this paper, we concentrate on the representation of finite automata used by our implementation. We use so-called border functions to represent interval-based transitions. Instead of storing transitions of form $([\sigma_1, \sigma_2], q)$, (for characters between σ_1 and σ_2, go to state q) we store only the points where the behavior of the transition changes, i.e. the borders, so instead we store $(\sigma_1, q), (\sigma + 2, \#)$, with # denoting 'getting stuck'. For every character, the transition

Submitted to:
GandALF 2022

is determined by the greatest border that is not greater than the character itself. When implementing transformations on automata, border functions are much easier to deal with than intervals, because there is no need to distinguish between the beginning and the end of an interval. All that needs to be looked at, are the borders.

In addition to the use of border functions, we store the automata in an array (vector) using relative state references. This removes the need to represent automata as graphs, and the combinations that correspond to regular operators become trivial. In most cases, the automata can be just concatenated with the addition of a few ε transitions.

These two modifications result in a representation that is easy to explain and implement, and whose automata are easy to read. This is useful both for teaching and for debugging. Big automata representing complete tokenizers tend to be local, and our transformations preserve this locality.

In general, our automaton representation is somewhat more complicated than the standard representation, some of the correctness proofs become a bit more complicated, but the operations themselves are equally complicated. The extra effort in defining the automata and proving the operations correct pays off when the automata are applied: The standard representation must be further adapted in order to make it work in practice, while ours works without further adaptation. We have implemented flat automata in C^{++}, and the implementation is available from [12].

In the next section, we will define alphabets and border functions. In Section 3, we define acceptors, which are automata that can only accept or reject. In Section 4 we explain how to obtain acceptors by means of regular operations. We do not define regular expressions as separate entities, instead we directly construct the automata. In Section 5, we define classifiers, which are obtained by pairing acceptors with token names. In Section 6 we adapt the standard determinization procedure to automata with border functions. The border functions make it possible to keep the algorithm simple. In Section 7 we adapt state minimization to our representation of automata. The algorithm can be kept simple (as simple as for single characters) because of the border functions. We use Hopcroft's algorithm ([7]) with an adaptation of a filter from [10]. In Section 8 we draw some conclusions, and sketch possibilities for future work.

2 Preliminaries

We will assume that alphabets are well-ordered sets. In the usual case where the alphabet is finite, it is sufficient that there exists a total order on the alphabet.

Definition 2.1 *An* alphabet *is a pair* $(\Sigma, <)$, *s.t.* Σ *is a non-empty set, and* $<$ *is a well-order on* Σ. *We define* $c_{\perp} = \min(\Sigma)$. *If* $\{c' \in \Sigma | \; c < c'\}$ *is non-empty, then we write* c^{+1} *for* $\min\{\, c' \in \Sigma \mid c < c' \,\}$.

As far as we know, all alphabets in use, including ASCII and Unicode ([5]) satisfy the requirements of Definition 2.1 or can be adapted in such a way that they do.

Our aim is to define automata by means of intervals, because in practice, many tokens (like for example numbers or identifiers) use intervals in their definitions. Another advantage of use of intervals is that is becomes possible to use large alphabets, like Unicode.

Dealing with intervals becomes easier if one removes the distinction between start and end of interval. This can be done by storing only the points where a new value starts, and creating a special value # denoting 'not in any interval'. For example, when defining identifiers, one may want to define a transition to some state q, for $\sigma \in \{A, \dots, Z\} \cup \{a, \dots, z\}$, because all letters usually behave the same. This can be represented as $\{\, (A,q),\, (Z+1,\#),\, (a,q),\, (z+1,\#) \,\}$. Here $A, Z+1, a, z+1$ are the borders where the behavior changes. In order to determine the transition for a given symbol one needs to find the largest border that is not greater than the symbol at hand. We will call a function, that is defined in this way, a border function.

Definition 2.2 *Let* $(\Sigma, <)$ *be an alphabet, let* D *be an arbitrary, non-empty set. A* border function ϕ *on* $(\Sigma, <)$ *is a partial function from* Σ *to* D*, defined for a finite subset of* Σ*, but at least for* $c_\perp$*. We write* $\text{dom}(\phi)$ *for the set of symbols for which* ϕ *is defined. We call the set* D *the* range *of* ϕ*. We will write border functions as sets of ordered pairs, whenever it is convenient.*

Definition 2.3 *For a given* $\sigma \in \Sigma$*, we first define* $\sigma^{\leq} = \max\{\, \sigma' \in \text{dom}(\phi) \mid \sigma' < \sigma \text{ or } \sigma' = \sigma \,\}$*. After that, we define* $\phi^{\leq}(\sigma) = \phi(\sigma^{\leq})$*.*

It can be easily checked that $\phi^{\leq}(\sigma)$ always exists and is uniquely defined, because ϕ is finite and has $c_\perp$ in its domain.

Definition 2.4 *Let* ϕ_1 *and* ϕ_2 *be two border functions on the same alphabet* $(\Sigma, <)$*. We say that* ϕ_1 *and* ϕ_2 *are* equivalent *if for all* $\sigma \in \Sigma,\ \phi_1^{\leq}(\sigma) = \phi_2^{\leq}(\sigma)$*.*
We call ϕ minimal *if there exists no equivalent* $\phi' \subset \phi$*. We define* the minimization of ϕ *as the* $\subseteq$*-minimal border function that is equivalent to* ϕ*.*

Definition 2.4 uses the fact that border functions can be viewed as sets of ordered pairs. It can be easily checked that border functions can be minimized. If $\phi(\sigma_1) = \phi(\sigma_2)$, and there is no σ' with $\sigma_1 < \sigma' < \sigma_2$ in the domain of ϕ, then $\phi(\sigma_2)$ can be removed from ϕ.

If for example both $\phi(1) = \phi(3) = 4$, and 2 is not in the domain of ϕ, then removing 3 from the domain will not have effect on $\phi^{\leq}$. Assume that $\Sigma = \{-100, -99, \ldots, 99, 100\}$. Assume that $\phi(-100) = -1,\ \ \phi(-4) = 3,\ \ \phi(2) = 8$, and $\phi(6) = 4$, then $\phi^{\leq}(-100) = \phi^{\leq}(-3) = -1,\ \ \phi^{\leq}(-4) = \phi^{\leq}(1) = 3$, $\phi^{\leq}(2) = \phi^{\leq}(5) = 8$, and $\phi^{\leq}(6) = \phi^{\leq}(100) = 4$.

Definition 2.5 *Let* ϕ_1 *and* ϕ_2 *be two border functions over the same alphabet* $(\Sigma, <)$*. Let* D_1 *be the range of* ϕ_1 *and let* D_2 *be the range of* ϕ_2*. We define the* product $\phi_1 \times \phi_2$ *as the border function*

$$\{\ (\ \sigma, (\phi_1^{\leq}(\sigma), \phi_2^{\leq}(\sigma))\ )\mid \sigma \in \text{dom}(\phi_1) \cup \text{dom}(\phi_2)\ \}.$$

The range of $\phi_1 \times \phi_2$ *is* $D_1 \times D_2$*.*

Definition 2.6 *Let* ϕ *be a border function over alphabet* $(\Sigma, <)$*. Let* D *be the range of* ϕ*. Let* f *be a function from* D *to some set* D'*. We define* the application of f on ϕ *as the minimization of*

$$\{\ (\sigma, f(\phi(\sigma)))\mid \sigma \in \text{dom}(\phi)\ \}.$$

We will write $f(\phi)$ *for the application of* f *on* Φ*.*

3 Acceptors

We distinguish two types of automata which we call *acceptor* and *classifier*. An acceptor can only accept or reject a word, while a classifier is able to classify words. A complete tokenizer is a classifier, while single tokens are defined by acceptors. A classifier is obtained by associating acceptors with token classes.

Although acceptors can be directly defined in code through initializers, it is inconvenient to do this, and we will construct them from regular expressions. We do not view regular expressions as independently existing objects. Instead we view regular operators as operators that work directly on acceptors. We have no data structure for regular expressions.

Acceptors are standard finite automata. We represent them in such a way that the regular operations are easy to present and to implement. In order to obtain this, we use a flat, linear representation which we will introduce shortly. In the literature, finite automata are traditionally represented by graphs whose vertices are states and whose edges are labeled with symbols. (See for example [1, 14]). This is implementable, but we believe that our representation is simpler. There is no problem of memory management, and printing automata is easy. When we print an automaton, the states are printed absolute instead of relative.

Definition 3.1 *Let $(\Sigma, <)$ be an alphabet. An* acceptor $\mathscr{A}$ *over Σ is a finite sequence*

$$\mathscr{A} = (\Lambda_1, \phi_1), \ldots, (\Lambda_n, \phi_n) \ \ (n \geq 0),$$

where each $\Lambda_i \subseteq \mathscr{Z}$, and each ϕ_i is a border function from Σ to $\mathscr{Z} \cup \{\#\}$.

Each Λ_i denotes the set of epsilon transitions from state i, while each ϕ_i represents the set of non epsilon transitions from state i.

We call $\mathscr{A}$ deterministic *if all Λ_i are empty. We often write $\|\mathscr{A}\|$ instead of n for the size of $\mathscr{A}$.*

We use the following conventions:

- # means that no transition is possible. Note that $\phi(\sigma) = \#$ should not be confused with '$\phi(\sigma)$ is undefined'. Due to the use of border functions, one has to explicitly state that $\phi(\sigma)$ has no transition, because otherwise $\phi^{\leq}(\sigma)$ would 'inherit' a transition from a $\sigma' < \sigma$.
- The initial state is always 1, and the accepting state is always $n+1$, just outside of the acceptor.
- State references in a Λ_i or ϕ_i are always relative to i. That means that i itself is represented by 0, $i+1$ is represented by 1, while $i-1$ is represented by -1, etc.
- There are no transitions to states < 2 or states $> n+1$.

Note that the last condition stipulates that the acceptor cannot return to the first state during a run. Most of the constructions for combining acceptors become simpler with this condition. Forbidding transitions to the initial state of an automaton is common in the literature, see for example [8]. An automaton with this property is usually called *committing*.

In addition, it is usually required that there is exactly one accepting state, and that there are no transitions going out of the accepting state. These conditions are automatically fulfilled by our representation. Acceptors can be non-deterministic, but all non-determinism must be inside the Λ_i, i.e. in the form of ε-transitions. All acceptors constructed by the regular operations of Section 4 have this form. If one wants to represent a general non-deterministic automaton, one has to remove transitions from the same state with overlapping intervals. For example, a state can have transitions to different states for the intervals $[a, \ldots, z]$, and $[a, \ldots, d]$. In this case, the original state can be split into two states connected by an ε-transition. During this process, the number of states and ε-transitions can increase, but it will not become more than the total number of borders in the step functions of the original automaton.

We will now formally define when $\mathscr{A}$ accepts a word w.

Definition 3.2 *Let $\mathscr{A}$ be an acceptor over alphabet Σ. We define a* configuration *of $\mathscr{A}$ as a pair (z, w), with $1 \leq z \leq \|\mathscr{A}\|$ and $w \in \Sigma^*$.*
We define the transition relation $\vdash$ *between configurations as follows:*

- *If $j \in \Lambda_i$ and $w \in \Sigma^*$, then $(i, w) \vdash (i+j, w)$.*
- *If $w \in \Sigma^*$, $\sigma \in \Sigma$, and $\phi_i^{\leq}(\sigma) = j$ with $j \neq \#$, then $(i, w) \vdash (i+j, w\sigma)$, where $\phi_i^{\leq}(\sigma)$ is the border function of state i applied on σ.*

We define $\vdash^i$ and $\vdash^*$ between configurations as usual.
We say that $\mathscr{A}$ accepts *$w \in \Sigma^*$ if $(1, \varepsilon) \vdash^* (\,\|\mathscr{A}\| + 1, w\,)$.*
We write $\mathscr{L}(\mathscr{A})$ for the language $\{w \in \Sigma^* \mid \mathscr{A}$ accepts $w\}$.

Example 3.3 *We give an acceptor that accepts standard identifiers (starting with a letter, followed by zero or more letters, digits, or underscores). The first column, which numbers the states, is not part of the automaton.*

$$\begin{array}{lll} 1: & \{\,\} & \{\,(c_\perp,\#),\ (A,1),\ (Z^{+1},\#),\ (a,1),\ (z^{+1},\#)\,\} \\ 2: & \{1\} & \{\,(c_\perp,\#),\ (0,0),\ (9^{+1},\#),\ (A,0),\ (Z^{+1},\#),\ (\_,0),\ (\_^{+1},\#),\ (a,0),\ (z^{+1},\#)\,\} \end{array}$$

The initial state is 1. From state 2, there is one epsilon transition to state $2+1=3$, which is the accepting state. If $\sigma \in \{a,\ldots,z\} \cup \{A,\ldots,Z\} \cup \{\_\}$, there is a transition from state 2 to state $2+0=2$.

Example 3.4 *The following acceptor accepts the reserved word "while". The accepting state is 6.*

$$\begin{array}{lll} 1: & \{\,\} & \{\,(c_\perp,\#),(w,1),(w^{+1},\#)\,\} \\ 2: & \{\,\} & \{\,(c_\perp,\#),(h,1),(h^{+1},\#)\,\} \\ 3: & \{\,\} & \{\,(c_\perp,\#),(i,1),(i^{+1},\#)\,\} \\ 4: & \{\,\} & \{\,(c_\perp,\#),(l,1),(l^{+1},\#)\,\} \\ 5: & \{\,\} & \{\,(c_\perp,\#),(e,1),(e^{+1},\#)\,\} \end{array}$$

It may seem from Examples 3.3 and 3.4 that acceptors can be easily written by hand, but unfortunately that is not the case in general, because one needs to know the order of the alphabet. One must remember that upper case letters come before lower case letters in ASCII, and the relative positions of special symbols. We initially thought that it would be doable, but it turned out impossible to write non-trivial acceptors by hand. Despite this, automata are easily readable if one prints the states in transitions as absolute, and uses the following printing convention: In the transition function, pairs of form $(\sigma,\#)$ where σ is the successor of a symbol τ, are printed in the form $(\tau^{+1},\#)$. Without this convention, for example $(A,0),(Z^{+1},\#)$ would be printed as $(A,0),(\,[,\#)$, which is a bit hard to read.

4 Obtaining Acceptors by Regular Operations

As explained below Example 3.4, writing down acceptors directly by hand is unpractical. The standard approach in the literature and in existing systems, is to obtain automata by means of regular expressions ([1, 14]). We follow this approach, but we will not view regular expressions as independently existing objects. Rather we define a set of regular operators on automata that construct acceptors at once.

Definition 4.1 *Let $(\Sigma,<)$ be an alphabet. In the current definition, we will construct border functions with range $\{\mathbf{f},\mathbf{t}\}$. We define $\phi_\emptyset = \{(\sigma_\perp,\mathbf{f})\}$, and $\phi_\Sigma = \{(\sigma_\perp,\mathbf{t})\}$. We define*

$$\phi_{\geq\sigma} = \ \mathit{if}\ (\sigma = \sigma_\perp)\ \mathit{then}\ \{(\sigma_\perp,\mathbf{t})\}\ \mathit{else}\ \{(\sigma_\perp,\mathbf{f}),(\sigma,\mathbf{t})\}.$$

For $\sigma \in \Sigma$, let $C_{>\sigma} = \{\sigma' \in \Sigma \mid \sigma' > \sigma\,\}$, the set of symbols greater than σ. We define

$$\phi_{\leq\sigma} = \ \mathit{if}\ (C_{>\sigma} = \emptyset)\ \mathit{then}\ \{(\sigma_\perp,\mathbf{t})\}\ \mathit{else}\ \{(\sigma_\perp,\mathbf{t}),(\min(C_{>\sigma}),\mathbf{f})\}$$

We define $\phi_1 \cap \phi_2 = I(\phi_1 \times \phi_2)$, with $I(\,(d_1,d_2)\,) = \ \mathit{if}\ (d_1 = \mathbf{t}\ \mathit{and}\ d_2 = \mathbf{t})\ \mathit{then}\ \mathbf{t}\ \mathit{else}\ \mathbf{f}$, and we define $\neg\phi = N(\phi)$, with $N(d) = \ \mathit{if}\ (d = \mathbf{t})\ \mathit{then}\ \mathbf{f}\ \mathit{else}\ \mathbf{t}$. Other Boolean combinations, like $\phi_1 \cup \phi_2$, and $\phi_1 \backslash \phi_2$ can be defined analogously.

Definition 4.2 *We define the following ways of constructing acceptors over* Σ :

- *The acceptor* $\mathscr{A}_\varepsilon$*, which accepts exactly the empty word, is defined as* $(\,)$.
- *Let* $f_\#$ *be the function defined from* $f_\#(\mathbf{f}) = \#$*, and* $f_\#(\mathbf{t}) = 1$*. Then, if* ϕ *is a border function with range* $\{\mathbf{f}, \mathbf{t}\}$*, we define* $\mathscr{A}[\phi]$ *as the acceptor* $(\,(\{\,\}, f_\#(\phi)\,)$*. (We are using Definition 2.6.)*

$\mathscr{A}[\phi]$ accepts exactly the symbols (as words) for which $\phi^{\leq}$ returns $\mathbf{t}$. Using $\mathscr{A}[\phi]$, it is easy to construct acceptors for Boolean combinations of intervals. For example, an acceptor that accepts exactly letters can be defined as $\mathscr{A}[\ (\phi_{\geq a} \cap \phi_{\leq z}) \cup (\phi_{\geq A} \cap \phi_{\leq Z})\ ]$. An acceptor that accepts all letters except X can be defined as $\mathscr{A}[\ \phi_\Sigma \cap \neg(\phi_{\geq X} \cap \phi_{\leq X})\ ]$. The acceptor that accepts nothing can be defined as $\mathscr{A}_\emptyset = \mathscr{A}[\phi_\emptyset]$.

Definition 4.3 *Let* $\mathscr{A} = (\Lambda_1, \Phi_1), \ldots, (\Lambda_n, \Phi_n)$ *and* $\mathscr{A}' = (\Lambda'_1, \Phi'_1), \ldots, (\Lambda'_n, \Phi'_{n'})$ *be acceptors. We define the* concatenation $\mathscr{A} \circ \mathscr{A}'$ *as* $(\Lambda_1, \Phi_1), \ldots, (\Lambda_n, \Phi_n), (\Lambda'_1, \Phi'_1), \ldots, (\Lambda'_{n'}, \Phi'_{n'})$.

Operation $\circ$ simply concatenates acceptors.

Theorem 4.4 *Let* $\mathscr{A}_1$ *and* $\mathscr{A}_2$ *be acceptors.* $\mathscr{L}(\mathscr{A}_1 \circ \mathscr{A}_2) = \{\, w_1 w_2 \mid w_1 \in \mathscr{L}(\mathscr{A}_1)$ *and* $w_2 \in \mathscr{L}(\mathscr{A}_2)\,\}$.

Proof
Throughout the proof, we define $n_1 = \|\mathscr{A}_1\|$ and $n_2 = \|\mathscr{A}_2\|$.

Let $w \in \mathscr{L}(\mathscr{A}_1 \circ \mathscr{A}_2)$. By definition, $(1, \varepsilon) \vdash^* (n_1 + n_2 + 1, w)$. There exists at least one prefix w' of w, s.t. $(1, \varepsilon) \vdash^* (n', w') \vdash^* (n_1 + n_2 + 1, w)$ having $n' > n_1$ because w itself satisfies this condition. Let w_1 be the smallest such prefix. By the last condition of Definition 3.1, n' must be equal to $n_1 + 1$, hence $w_1 \in \mathscr{L}(\mathscr{A}_1)$. Let w_2 be the rest of w, so we have $w = w_1 w_2$. Because $(n_1 + 1, w_1) \vdash^* (n_1 + n_2 + 1, w)$, it follows that $(n_1 + 1, \varepsilon) \vdash^* (n_1 + n_2 + 1, w_2)$. Note that this sequence still uses $\mathscr{A}_1 \circ \mathscr{A}_2$. Since $\mathscr{A}_2$ has no transitions to states < 2, and all transitions originate from $\mathscr{A}_2$, the configurations (n'', w'') in the sequence $(n_1 + 1, \varepsilon) \vdash^* (n_1 + n_2 + 1, w_2)$ must have $n'' \geq n_1 + 1$. Since transitions are relative, we have $(1, \varepsilon) \vdash^* (n_2 + 1, w_2)$ in $\mathscr{A}_2$.

Now assume that $w_1 \in \mathscr{L}(\mathscr{A}_1)$ and $w_2 \in \mathscr{L}(\mathscr{A}_2)$. We have $(1, \varepsilon) \vdash^* (n_1, w_1)$ in $\mathscr{A}_1$, and $(1, \varepsilon \vdash^* (n_2, w_2)$ in $\mathscr{A}_2$. The second sequence can be easily modified into $(n_1 + 1, \varepsilon) \vdash^* (n_1 + n_2 + 1, w_2)$ in $\mathscr{A}_1 \circ \mathscr{A}_2$, which in turn can be modified into $(n_1 + 1, w_1) \vdash^* (n_1 + n_2 + 1, w_1 w_2)$ in $\mathscr{A}_1 \circ \mathscr{A}_2$.

Definition 4.5 *We first define an operation that adds* ε *transitions to acceptor. Let* $\mathscr{A} = (\Lambda_1, \Phi_1), \ldots, (\Lambda_n, \Phi_n)$ *be an acceptor. We define* $\mathscr{A}\{i \to^\varepsilon j\}$ *as* $(\Lambda_1, \Phi_1), \ldots, (\Lambda_i \cup \{j - i\}, \Phi_i), \ldots, (\Lambda_n, \Phi_n)$*. We add* $j - i$ *instead of just* j *to* Λ_i *because transitions are relative.*
The union $\mathscr{A}_1 \,|\, \mathscr{A}_2$ *of* $\mathscr{A}_1$ *and* $\mathscr{A}_2$ *is defined as*

$$(\mathscr{A}_1 \circ \mathscr{A}_\emptyset \circ \mathscr{A}_2)\{\ 1 \to^\varepsilon \|\mathscr{A}_1\| + 2,\ \ \|\mathscr{A}_1\| + 1 \to^\varepsilon \|\mathscr{A}_1\| + \|\mathscr{A}_2\| + 2\ \}.$$

In this definition, we use $\mathscr{A}_\emptyset$ as defined below Definition 4.2, namely $\mathscr{A}_\emptyset = (\{\,\}, \{(c_\perp, \#)\})$. We prove that union behaves as expected:

Theorem 4.6 *For every two acceptors* $\mathscr{A}_1$ *and* $\mathscr{A}_2$*, we have* $\mathscr{L}(\mathscr{A}_1 \,|\, \mathscr{A}_2) = \mathscr{L}(\mathscr{A}_1) \cup \mathscr{L}(\mathscr{A}_2)$.

Proof
As before, we use $n_1 = \|\mathscr{A}_1\|$ and $n_2 = \|\mathscr{A}_2\|$. Assume that $w \in \mathscr{L}(\mathscr{A}_1 | \mathscr{A}_2)$. By definition, $(1, \varepsilon) \vdash^* (n_1 + n_2 + 2, w)$. If state $n_1 + 2$ does not occur in this sequence, it must be the case that the state $n_1 + 1$ occurs in the sequence, because the accepting state is reachable only from $n_1 + 1$ or from states $\geq n_1 + 2$.. This implies that $w \in \mathscr{L}(\mathscr{A}_1)$. Similarly, if state $n_1 + 2$ occurs in the sequence, then we note that $n_1 + 2$ originates from the initial state of $\mathscr{A}_2$. It follows that $w \in \mathscr{L}(\mathscr{A}_2)$. As a consequence, we have $\mathscr{L}(\mathscr{A}_1 | \mathscr{A}_2) \subseteq \mathscr{L}(\mathscr{A}_1 \cup \mathscr{A}_2)$.

Now assume that $w \in \mathscr{L}(\mathscr{A}_1) \cup \mathscr{L}(\mathscr{A}_2)$. If $w \in \mathscr{L}(\mathscr{A}_1)$, we have $(1,\varepsilon) \vdash^* (n_1+1,w)$ in $\mathscr{A}_1$. In $\mathscr{A}_1|\mathscr{A}_2$, this sequence can be extended to $(1,\varepsilon) \vdash^* (n_1+1,w) \vdash (n_1+n_2+2,w)$.

If $w \in \mathscr{L}(\mathscr{A}_2)$, we have $(1,\varepsilon) \vdash^* (n_1+1,w)$ in $\mathscr{A}_2$. In $\mathscr{A}_1|\mathscr{A}_2$, this sequence becomes $(1,\varepsilon) \vdash (n_1+2,\varepsilon) \vdash^* (n_1+n_2+2,w)$. This implies that $\mathscr{L}(\mathscr{A}_1 \cup \mathscr{A}_2) \subseteq \mathscr{L}(\mathscr{A}_1|\mathscr{A}_2)$.

Definition 4.7 *The Kleene star $\mathscr{A}^*$ of $\mathscr{A}$ is defined as*

$$(\mathscr{A}_\emptyset \circ \mathscr{A} \circ \mathscr{A}_\emptyset)\{1 \to^\varepsilon 2,\ 2 \to^\varepsilon \|\mathscr{A}\|+3,\ \|\mathscr{A}\|+2 \to^\varepsilon 2\}.$$

Theorem 4.8 *For every acceptor $\mathscr{A}$, the following holds:*

$$w \in \mathscr{L}(\mathscr{A}^*) \textit{ iff there exist } w_1,\dots,w_k\ (k \geq 0),\ \textit{s.t. } w = w_1 w_2 \cdots w_k \textit{ and each } w_i \in \mathscr{L}(\mathscr{A}).$$

Proof
In this proof, let $n = \|\mathscr{A}\|$. First assume that $(1,\varepsilon) \vdash^* (n+3,w)$. By separating out the visits of state 2, we can write this sequence in the following form:

$$(1,\varepsilon) \vdash (2,\varepsilon) \vdash^* (2,v_1) \vdash^* (2,v_2) \vdash^* \cdots \vdash^* (2,v_{k-1}) \vdash^* (2,v_k) \vdash^* (n+3,w),$$

where each subsequence $\vdash^*$ contains no visits to state 2. For simplicity, set $v_0 = \varepsilon$. Then, for i with $1 \leq i \leq k$, the word v_{i-1} is a prefix of v_i. For $1 \leq i \leq k$, define the difference w_i such that $v_{i-1}\,w_i = v_i$.

By construction of $\mathscr{A}^*$, state 2 originates from the original acceptor $\mathscr{A}$. Hence $(2,v_{i-1}) \vdash^* (2,v_i)$ implies that $(2,v_{i-1}) \vdash^* (2+n,v_i) \vdash (2,v_i)$. Since the sequence $(2,v_{i-1}) \vdash^* (2+n,v_i)$ must be completely within $\mathscr{A}$, it follows that $w_i \in \mathscr{L}(\mathscr{A})$. For the final sequence $(2,v_k) \vdash^* (n+3,w)$, it can be easily checked that the only transition to $n+3$ is an ε-transition from state 2. Hence, we have $(2,v_k) \vdash (n+3,w)$ and $v_k = w$. Since we have $w = w_1 \cdots w_k$, this completes one direction of the proof.

For the other direction, assume we have $w_1,\dots,w_k$, s.t each $w_i \in \mathscr{L}(\mathscr{A})$ for some $k \geq 0$. By definition, we have $(1,\varepsilon) \vdash^* (n+1,w_i)$ in $\mathscr{A}$, which implies that for every word $v' \in \Sigma^*$, we have $(1,v') \vdash^* (n+1,v'w_i)$ in $\mathscr{A}$.
In $\mathscr{A}^*$, we have $(2,v') \vdash^* (n+2,v'w_i)$. By combining and properly instantating the v', we obtain

$$(1,\varepsilon) \vdash (2,\varepsilon) \vdash^* (2,w_1) \vdash^* (2,w_1w_2) \vdash^* \cdots \vdash^* (2,w_1w_2\cdots w_k) \vdash (n+3,w_1w_2\cdots w_k),$$

which completes the proof.
At this point, we can define all other common regular operations. For example $\mathscr{A}^+$ can be defined as $\mathscr{A} \circ \mathscr{A}^*$, and $\mathscr{A}^?$ can be defined as $\mathscr{A}|\mathscr{A}_\varepsilon$. Since direct construction results in slightly smaller acceptors, we still give the following definitions:

Definition 4.9 *Let $\mathscr{A}$ be an acceptor. We define the non-empty repetition $\mathscr{A}^+$ as*

$$(\mathscr{A}_\emptyset \circ \mathscr{A} \circ \mathscr{A}_\emptyset)\{\ 1 \to^\varepsilon 2,\ \|\mathscr{A}\|+2 \to^\varepsilon 2,\ \|\mathscr{A}\|+2 \to^\varepsilon \|\mathscr{A}\|+3\ \}.$$

We define the optional expression $\mathscr{A}^?$ as $\mathscr{A}\{\ 1 \to^\varepsilon \|\mathscr{A}\|+1\ \}$.

The construction of $\mathscr{A}^?$ relies on the fact that $\mathscr{A}$ is committing.

Theorem 4.10 *For every acceptor $\mathscr{A}$, the following holds:*

$$w \in \mathscr{L}(\mathscr{A}^+) \textit{ iff there exist } w_1,\dots,w_k\ (k \geq 1),\ \textit{s.t. } w = w_1 w_2 \cdots w_k \textit{ and each } w_i \in \mathscr{L}(\mathscr{A}).$$

$$\mathscr{L}(\mathscr{A}^?) = \mathscr{L}(\mathscr{A}) \cup \{\varepsilon\}.$$

Instead of the automaton in example 3.3, we can now write:

$$\mathscr{A}[(\phi_{\geq a} \cap \phi_{\leq z}) \cup (\phi_{\geq A} \cap \phi_{\leq Z})] \circ \mathscr{A}[(\phi_{\geq a} \cap \phi_{\leq z}) \cup (\phi_{\geq A} \cap \phi_{\leq Z}) \cup (\phi_{\geq 0} \cap \phi_{\leq 9}) \cup (\phi_{\geq \_} \cap \phi_{\leq \_})]^*.$$

5 Classifiers

In order to obtain a complete tokenizer, it is not sufficient to accept or reject a given input. Instead one must classify input into different groups. We call an automata that can classify a *classifier*. Contrary to standard text books, like for example [14], we define determinization and minimization on classifiers, not on acceptors.

Definition 5.1 *Let* $(\Sigma, <)$ *be an alphabet. Let* T *be a non-empty set of* token classes. *A* classifier over Σ *into* T *is a non-empty, finite sequence*

$$\mathscr{C} = (\Lambda_1, \phi_1, t_1), \ldots, (\Lambda_n, \phi_n, t_n) \ \ (n \geq 1),$$

where each $\Lambda_i \subseteq \mathscr{Z}$*, each* ϕ_i *is a border function from* Σ *to* $\mathscr{Z} \cup \{\#\}$*, and each* $t_i \in T$*. We will often write* $\|\mathscr{C}\|$ *for the size of* $\mathscr{C}$*. We call* $\mathscr{C}$ *deterministic if all* Λ_i *are empty.*

For representing transitions, we use the same conventions as for acceptors, namely that transitions are stored relative, and $\phi_i(\sigma) = \#$ means that no transition is possible. In contrast to acceptors, we allow transitions to state 1, and we forbid transitions to state $n+1$. Intuitively, a classifier is a non-deterministic automaton, which looks for the longest run possible, and classifies as t_i when it gets stuck in state i. We will make this more precise soon.

In order to obtain a classifier, we start with a trivial classifier that classifies every input as error (actually, this classifier defines what is an error), and add pairs of acceptors and token classes.

We always assume that state 1 defines the error class. This is a reasonable choice, because no classifier can classify ε as a meaningful token.

Definition 5.2 *Let* T *be a token class. Let* $e, t \in T$ *and let* $\mathscr{A} = (\Lambda_1, \phi_1), \ldots, (\Lambda_n, \phi_n)$ *be an acceptor. We define* $\mathscr{A}[e,t]$ *as the classifier* $(\Lambda_1, \phi_1, e), \ldots, (\Lambda_n, \phi_n, e), (\{\ \}, \{(\sigma_\perp, \#)\}, t)$, *i.e. as the classifier that classifies words accepted by* $\mathscr{A}$ *as* t*, and all other words as* e*.*

Let $e \in T$*. We define* $\mathscr{C}_e = (\{\ \}, \{(\sigma_\perp, 0)\}, e)$, *i.e. as the classifier that classifies every word as* e*.*

For a classifier $\mathscr{C}$ *with first classification* t_1*, acceptor* $\mathscr{A}$*, and* $t \in T$*, we define* $\mathscr{C}[\, t : \mathscr{A}\,]$ *as*

$$\mathscr{C}\{\ 1 \rightarrow^{\varepsilon} \|\mathscr{C}\| + 1\ \} \circ \mathscr{A}[t_1, t].$$

Here $\circ$ *denotes concatenation of acceptors.*

The construction of $\mathscr{C}[\, t : \mathscr{A}\,]$ appends $\mathscr{A}$ to $\mathscr{C}$ in such a way that words accepted by $\mathscr{A}$ will be classified as t. Since acceptors accept by falling out of the automaton, we need to add an additional state without outgoing transitions, which will classify words that are able to reach it as t. We also add an ε transition from the first state to the added acceptor. Words that cannot reach an accepting state of any of the acceptors will be classified as t_1, because the classification of the first state is used as error classification.

Example 5.3 *Assume that we want to construct a classifier that classifies identifiers as I with the exception of 'while', which should be classified as W. Using the acceptors of Examples 3.3 and 3.4, we can* *construct* $\mathscr{C}_E[\, I : \mathscr{A}_{\text{id}},\ W : \mathscr{A}_{\text{while}}\,]$ *as*

$1:$	$\{1,4\}$	$\{\ (c_\perp,0)\ \}$	E
$2:$	$\emptyset$	$\{\,(c_\perp,\#),\ (A,1),\ (Z^{+1},\#),\ (a,1),\ (z^{+1},\#)\,\}$	E
$3:$	$\{1\}$	$\{\,(c_\perp,\#),\ (0,0),\ (9^{+1},\#),\ (A,0),\ (Z^{+1},\#),\ (\_,0),\ (\_^{+1},\#),\ (a,0),\ (z^{+1},\#)\,\}$	E
$4:$	$\emptyset$	$\{\,(c_\perp,\#)\,\}$	I
$5:$	$\emptyset$	$\{\,(c_\perp,\#),(w,1),(w^{+1},\#)\,\}$	E
$6:$	$\emptyset$	$\{\,(c_\perp,\#),(h,1),(h^{+1},\#)\,\}$	E
$7:$	$\emptyset$	$\{\,(c_\perp,\#),(i,1),(i^{+1},\#)\,\}$	E
$8:$	$\emptyset$	$\{\,(c_\perp,\#),(l,1),(l^{+1},\#)\,\}$	E
$9:$	$\emptyset$	$\{\,(c_\perp,\#),(e,1),(e^{+1},\#)\,\}$	E
$10:$	$\emptyset$	$\{\,(c_\perp,\#)\,\}$	W

Without further restrictions, the classifier above can classify ‘while’ either as I or as W. In order to avoid such ambiguity, we always take the classification of the maximal (using $<$ on natural numbers) reachable state that is not an error state. In the current case, after reading ’while’ the reachable states are $1,3,4$ and 10. Since 10 is the maximal state and its label is not $t_1 = E$, the classifier classifies as W.

Other solutions for solving ambiguity do not work well. In particular using an order $<$ on T is unpleasant. If T is an enumeration type, it is difficult to control how T is ordered. If T is a string type, its order is determined by the lexicographic order, and it is tedious to override it.

Before we can make classification precise, we need to introduce one technical condition. By default, the first state defines the error state t_1. If from the first state it is possible to reach a state i with $t_i \neq t_1$, we could possibly classify the word as non-error. Whenever we encounter such a situation in real, it is due to a mistake, mostly due to writing $\mathscr{A}^*$ where $\mathscr{A}^+$ would have been required. Hence, we will forbid such automata.

Definition 5.4 *A classifier* $\mathscr{C}$ *is* well-formed *if it does not allow a sequence* $(1,\varepsilon) \vdash^* (i,\varepsilon)$ *with* $t_i \neq t_i$.

The automaton in Example 5.3 is well-formed. Changing t_2 into $t_2 = I$ would make it ill-formed.

The following definition makes classification precise:

Definition 5.5 *For classifiers, we define configurations as in Definition 3.2. We also define* $\vdash$ *and* $\vdash^*$ *in the same way.*

We define classification*: Classifying a word* $w \in \Sigma^*$ *means obtaining a maximal prefix* w' *of* w *that is not classified as error* (t_1)*, together with the preferred classification of* w'*. Let* $\mathscr{C}$ *be a classifier, let* $w \in \Sigma^*$*. Let* w' *be a maximal prefix of* w*, s.t. there exists a state* i *of* $\mathscr{C}$ *with* $(1,\varepsilon) \vdash^* (i,w')$ *and* $t_i \neq t_1$.

If no such state i *exists, then the classification of* w *equals* (ε,t_1).

If such a state exists, assume that i *is the largest state for which* $(1,\varepsilon) \vdash^* (i,w')$ *and* $t_i \neq t_1$*. In this case, the classification equals* (w',t_i).

6 Determinization

It is possible to run a non-deterministic classifier directly, but it is inefficient in the long run when many input words need to be classified. As with standard automata, a non-deterministic classifier can be transformed into an equivalent, deterministic classifier. The construction is almost standard (See for example [1, 8, 14]), but there are a few differences: We perform the construction on classifiers instead of acceptors, because that is what will be used in applications, and we get generalization to character intervals for free, because of the use of border functions. The advantage of border functions is that there is no need to distinguish between starts and ends of intervals. The only points that need to be looked at are the borders. As a result the construction is only slightly more complicated than the standard approach, while at the same time working in practice without adaptation. The following definition is completely standard:

Definition 6.1 *Let $\mathscr{C}$ be a classifier. Let S be a subset of its states. We define* the closure *of S, written as* $\text{CLOS}_{\mathscr{C}}(S)$ *as the smallest set of states S' with $S \subseteq S'$, and whenever $i \in S'$ and $j \in \Lambda_i$, we have $i + j \in S'$.*

As said before, during determinization one only needs to consider the borders:

Definition 6.2 *Let $\mathscr{C}$ be a classifier defined over alphabet $(\Sigma, <)$. Let S be a non-empty set of states of $\mathscr{C}$. We define*

$$\text{BORD}_{\mathscr{C}}(S) = \{\, \sigma \in \Sigma \mid \sigma \textit{ is in the domain of a } \phi_i \textit{ with } i \in S \,\}.$$

$\text{BORD}_{\mathscr{C}}(S)$ is the set of symbols where the border function of one of the states in S has a border. These are the points where 'something happens', and which have to be checked when constructing the deterministic classifier. In the classifier of Example 5.3, we have $\text{CLOS}_{\mathscr{C}}(\{1\}) = \{1,2,5\}$ and

$$\text{BORD}_{\mathscr{C}}(\{1,2,5\}) = \{\, c_{\perp}, A, Z^{+1}, a, w, w^{+1}, z^{+1} \,\}.$$

Before we describe the determinization procedure, we need a way of extracting classifications from sets of states:

Definition 6.3 *Let $\mathscr{C}$ be a classifier. Let S be a subset of its states. We define* $\text{CLASS}_{\mathscr{C}}(S)$ *as follows: If for all $i \in S$, one has $t_i = t_1$, then* $\text{CLASS}_{\mathscr{C}}(S) = t_1$. *Otherwise, let i be the maximal element in S for which $t_i \neq t_1$. We define* $\text{CLASS}_{\mathscr{C}}(S) = t_i$.

In example 5.3, $\text{CLASS}_{\mathscr{C}}(\emptyset) = \text{CLASS}_{\mathscr{C}}(\{1,2,3,5,6,7,8,9\}) = E$, $\text{CLASS}_{\mathscr{C}}(\{4,6,7\}) = I$, and $\text{CLASS}_{\mathscr{C}}(\{3,4,10\}) = W$.

Now we are ready to define the determinization procedure. It constructs a deterministic classifier $\mathscr{C}_{\text{det}}$ from $\mathscr{C}$.

Definition 6.4 *The determinization procedure maintains a map H that maps subsets of states of $\mathscr{C}$ that we have discovered into natural numbers. It also maintains a map S_i that is the inverse of H, so we always have $S_{H(S)} = S$.*

1. *Start by setting* $H(\ \text{CLOS}_{\mathscr{C}}(\{1\})\ ) = 1$, *and by setting* $S_1 = \text{CLOS}_C(\{1\})$.
2. *Set* $\mathscr{C}_{\text{det}} = (\,)$.
3. *As long as* $\|\mathscr{C}_{\text{det}}\| < \|H\|$, *repeat the following steps:*
4. *Let* $i = \|\mathscr{C}_{\text{det}}\| + 1$. *Append* $(\{\,\},\{\,\},\text{CLASS}_{\mathscr{C}}(S_i))$ *to* $\mathscr{C}_{\text{det}}$.
5. *For every* $\sigma \in \text{BORD}_{\mathscr{C}}(S_i)$, *do the following:*
   - *Let* $S' = \{\, s + \phi_s^{\leq}(\sigma) \mid s \in S \textit{ and } \phi_s^{\leq}(\sigma) \neq \# \,\}$. *($\phi_s$ is the border function of state s.)*
   - *If $S' = \emptyset$, then extend ϕ_i by setting $\phi_i(\sigma) = \#$. Skip the remaining steps.*
   - *Set* $S'' = \text{CLOS}_{\mathscr{C}}(S')$.
   - *If S'' is not in the domain of H, then add $H(S'') = \|H\| + 1$ to H, and set $S_{\|H\|+1} = S''$.*
   - *At this point, we are sure that $H(S'')$ is defined. Extend ϕ_i by setting $\phi_i(\sigma) = H(S'')$.*

As usual, H and S can be discarded when the construction of $\mathscr{C}_{\rm det}$ is complete. It is easily checked that $\mathscr{C}$ is deterministic, because all its Λ_i are empty.

Theorem 6.5 *Let $\mathscr{C}$ be a classifier that is well-formed, and $\mathscr{C}_{\rm det}$ be the classifier constructed from $\mathscr{C}$ by using the determinization procedure of Definition 6.4. For every word $w \in \Sigma^*$, if $\mathscr{C}$ classifies w as (w', t'), and $\mathscr{C}_{\rm det}$ classifies w as (w'', t''), then $w' = w''$ and $t' = t''$.*

Proof
The proof is mostly standard, and we sketch only the points where it differs from the standard proof. Because $\mathscr{C}$ is well-formed, we have $t_1 = t_{det,1}$, which means that both classifiers will use the same token class as error class.
For every word $w \in \Sigma^*$, define the set $R_w = \{\, r \in \{\,1,\dots,\|\mathscr{C}\|\,\} \mid (1,\varepsilon) \vdash^* (w,r)\,\}$. These are the set of states that classifier $\mathscr{C}$ can reach while reading w.
Also define the relation $\delta_{det}(w,i)$ as $(\varepsilon,1) \vdash^* (w,i)$. (Classifier $\mathscr{C}_{det}$ reaches state i while reading w.)
It can be proven by induction, that

1. if $R_w \neq \emptyset$, then $\delta_{det}(w,i)$ implies $i = H(R_w)$. If $R_w = \emptyset$, then there is no i, s.t. $\delta_{det}(w,i)$.
2. if $R_w \neq \emptyset$, then $\delta_{det}(w,i)$ implies $t_{det,i} = \text{CLASS}(R_w)$.

Now we can look at the classification of an arbitrary word $w \in \Sigma^*$. If for all prefixes w' of w, we have $\text{CLASS}(R_{w'}) = t_1$, then $\mathscr{C}$ will classify w as $(\emptyset, t_1)$. If for some prefix there exists an i', s.t. $\delta_{\rm det}(w',i')$ holds, we have $t_{det,i'} = \text{CLASS}(R_{w'}) = t_1$ by **(2)**, so that $\text{CLASS}(R_{w'}) = t_{\rm det,1}$. It follows that $\mathscr{C}_{det}$ also classifies w as $(\emptyset, t_1)$.

If there exists a prefix w' of w for which $\text{CLASS}(R_{w'}) \neq t_1$, then let w' be the largest such prefix. There exists exactly one i', s.t. $\delta_{\rm det}(w',i')$ holds, and by **(2)** again, we have $t_{det,i'} = \text{CLASS}(R_{w'})$, which is not equal to $t_{det,1}$.

Because w' was chosen maximal, it follows that for all words $w'' \neq w'$ s.t. w' is a prefix of w'' and w'' is a prefix of w, either we have $R_{w''} = \emptyset$ or $\text{CLASS}(R_{w''}) = t_1$. In both cases, there is no i'', s.t. $(1,\varepsilon) \vdash (i'', w'')$ and $t_{det,i''}$ in classifier $\mathscr{C}_{det}$. In the former case, no i'' exists exists at all, and in the latter case, $\delta_{\rm det}(w'',i'')$ holds, and we have $t_{det,i''} = \text{CLASS}(R_{w''}) = t_1$.

As a consequence, both $\mathscr{C}$ and $\mathscr{C}_{det}$ will classify w as $(w', \text{CLASS}(R_{w'})\,)$.

7 State Minimization

It is well-known that for every regular language there exists a unique deterministic automaton with minimal number of states (See [1] Section 3.9, or [8] Section 4.4.3). The minimal automaton can be obtained in time $O(n.\log(n))$ from any deterministic automaton by means of Hopcroft's algorithm ([7]).

Altough it probably has minimal impact on performance, minimization has a suprising effect on the size of the classifier. It turns out that on classifiers obtained from realistic programming languages, the number of states decreases by 30/40%.

It is straightforward to adapt Hopcroft's algorithm to classifiers. We sketch the implementation below. The algorithm takes a deterministic classifier $\mathscr{C}$ as input, and constructs the smallest (in terms of equivalence classes) partition on the states of $\mathscr{C}$, s.t. $i \equiv j$ implies $t_i = t_j$ and for every $\sigma \in \Sigma$, $i + \phi_i^{\leq}(\sigma) \equiv j + \phi_j^{\leq}(\sigma)$. (We are implicitly assuming that $\# \equiv \#$ and $\# \neq i$.) Once one has the partition, the automaton can be minimized by selecting one state from each partition.

Definition 7.1 *We use an array $(P_1,\dots,P_p)$ for storing the current state partition. We have $\bigcup_{1 \leq i \leq p} P_i = \{\,1,\dots,\|\mathscr{C}\|\,\}$ and $i \neq j \Rightarrow P_i \cap P_j = \emptyset$.*

In addition to the partition $(P_1,\ldots,P_p)$, we use an index map I that maps states to their partition, i.e. for every state $i\,(1 \leq i \leq \|\mathscr{C}\|)$, we have $i \in P_{I_i}$.

The initial partition is obtained from a function f with domain $\{1,\ldots,\|\mathscr{C}\|\}$ and arbitrary range. States i and j are put in the same class iff $f(i) = f(j)$.

We tried two initialization strategies: The first strategy is simply taking $f(r) = t_r$, which means that two states will be equivalent if they have the same classification. The second is an adaptation of a heuristic in [10] that takes paths to possible future classifications into account. We discuss it in more detail shortly.

Due to the use of border functions instead of intervals, Hopcroft's algorithm needs only minor adaptation for classifiers in our representation. We give the algorithm:

Definition 7.2 *First create an array B of back transitions. For every state i with $1 \leq i \leq \|\mathscr{C}\|$, $B(i)$ is the set of states that have a transition into i, i.e.*
$B(s) = \{\, j \mid 1 \leq j \leq \|\mathscr{C}\|$ s.t. there exists a $\sigma \in \Sigma^*$, s.t. $\phi_j(\sigma) \neq \#$ and $j + \phi_j(\sigma) = i\,\}$.
Construct the initial partition $P = (P_1,\ldots,P_p)$ from the chosen initialization function f. Initialize the index array I from P. Create a stack $U = (1,\ldots,p)$ of indices. The variable name U stands for unchecked.

1. *As long as U is non-empty, pop an element from U, call it u, and do the following:*
2. *Construct $S = \bigcup_{i \in P_u} B(i)$. This is the set of states that have a transition into a state $i \in P_u$.*
3. *For every $\sigma \in \mathrm{BORD}_{\mathscr{C}}(S)$ do: Construct $F_\sigma = \{\, i \in S \mid \phi_i^{\leq}(\sigma) \neq \#$ and $i + \phi_i^{\leq}(\sigma) \in P_u\,\}$. Refine P, I, U with F_σ.*
   (F_σ is the set of states whose σ-transition goes into a state in P_u)

The refinement operation *is defined as follows: Assume that we want to refine P, I, U by a set of states F. For every P_i, s.t. $P_i \cap F \neq \emptyset$ and $P_i \not\subseteq F$, do the following:*

1. *Construct $N = P_i \backslash F$ and replace P_i by $P_i \cap F$.*
2. *If this results in $\|P_i\| < \|N\|$, then exhange N and P_i.*
3. *Append N to $(P_1,\ldots,P_i,\ldots,P_p)$, and assign $I(i) = p+1$, for $i \in N$. Add $(p+1)$ to U.*

The intuition of refinement is the fact that if some P_i partially lies inside F and partially outside F, then P_i needs to be split.

When the final partition $(P_1,\ldots,P_p)$ has been obtained, it is trivial to construct the quotient classifier $\mathscr{Q} = \mathscr{C}/(P_1,\ldots,P_p)$.

It is essential that $1 \in P_1$ because Definition 5.1 and Definition 5.5 treat t_1 as the error state. This can be easily obtained by sorting $(P_1,\ldots,P_p)$ by their minimal element before constructing the quotient classifier. An additional advantage of sorting is that it improves readability, because it preserves more of the structure of the original classifier.

Both [2] and [13] agree that U should be implemented as stack, as opposed to a queue.

Although Hopcroft's algorithm is theoretically optimal, it can be improved by a preprocessing stage. In the early stage of the algorithm, all states that classify as error will be in a single equivalence class. This equivalence class is gradually refined into smaller classes dependent on possible computations originating from these classes. Although the number of steps is limited by the number of states in the class, it may still be costly because the initial class is big.

The initial refinements can be removed by using a preprocessing stage. In [10], a filter for simple, deterministic automata is proposed that marks states with the shortest distance towards an accepting state. This can be done in linear time. In order to adapt this approach to classifiers, one has to include the accepted token in the markings.

Definition 7.3 *Let $\mathscr{C}$ be a classifier from alphabet $(\Sigma, <)$ into token set T. A* reachability function ρ *is a total function from $\{1, \ldots, \|\mathscr{C}\}$ to partial functions from T to $\mathscr{N}$.*

Intuitively, $\rho(i)(t) = n$ means that there exists a path of length n from i to a state j with $t_j = t$.

Although theoretically, the total size of ρ could be quadratic in the size of $\mathscr{C}$, in all cases that we encountered, all states except for the initial state, can reach only a few token classes.

Our goal is to compute the optimal reachability function and use it to initialize the first partition. This can be done with Dijkstra's algorithm.

Definition 7.4 *Start by setting $\rho(i) = \{\, (t_i, 0)\,\}$, for every state i that has $t_i \neq t_1$. (Every state can reach its own classification in 0 steps.) Set $\rho(i) = \{\,\}$ for the remaining states (that classify as error). Create a stack $U = (\,1, \ldots, \|\mathscr{C}\|\,)$ of unchecked states.*

- *While U is not empty, pick and remove a state from U, call it u, and do the following:*
- *For every $i \in B(u)$, for every $(t,n) \in \rho(u)$ do the following: If $\rho(i)(t)$ is undefined, insert $(t, n+1)$ to $\rho(i)$. Otherwise, if $\rho(i)(t) = n'$, set $\rho(i)(t) = \min(n', n+1)$.*

  If this results in a change of $\rho(i)$, then add i to U.

Using ρ to initialize the partition in Definition 7.2 works well in practice. In most cases, the first partition is also the final partition. We end the section with an example of a reachability function for a simple classifier that classifies identifiers and the reserved word 'for':

Example 7.5 *Consider the following deterministic classifier that classifies identifiers (for simplicity only lower case and digits), and the reserved word 'for':*

$$\begin{array}{llll}
1: & \emptyset & \{\,(c_\perp,\#),\,(a,1),\,(f,2),\,(g,1),\,(z^{+1},\#)\,\} & E \\
2: & \emptyset & \{\,(c_\perp,\#),\,(0,2),\,(9^{+1},\#),\,(a,3),\,(z^{+1},\#)\,\} & I \\
3: & \emptyset & \{\,(c_\perp,\#),\,(0,1),\,(9^{+1},\#),\,(a,2),\,(o,3),\,(p,2),\,(z^{+1},\#)\,\} & I \\
4: & \emptyset & \{\,(c_\perp,\#),\,(0,0),\,(9^{+1},\#),\,(a,1),\,(z^{+1},\#)\,\} & I \\
5: & \emptyset & \{\,(c_\perp,\#),\,(0,-1),\,(9^{+1},\#),\,(a,0),\,(z^{+1},\#)\,\} & I \\
6: & \emptyset & \{\,(c_\perp,\#),\,(0,-2),\,(9^{+1},\#),\,(a,-1),\,(r,1),\,(s,-1),\,(z^{+1},\#)\,\} & I \\
7: & \emptyset & \{\,(c_\perp,\#),\,(0,-3),\,(9^{+1},\#),\,(a,-2),\,(z^{+1},\#)\,\} & F
\end{array}$$

This classifier was constructed by the determinization procedure. If one initializes the partition with $f(i) = t_i$, the initial partition will be $(\{1\}, \{2,3,4,5,6\}, \{7\})$. The optimal reachability function has

$$\begin{array}{ll}
\rho(1) = & \{\,(I,1),\,(F,3)\,\} \\
\rho(2) = \rho(4) = \rho(5) = & \{(I,0)\,\} \\
\rho(3) = & \{\,(I,1),\,(F,2)\,\} \\
\rho(6) = & \{\,(I,1),\,(F,1)\,\} \\
\rho(7) = & \{\,(I,1),\,(F,0)\,\}
\end{array}$$

The minimal classifier has 5 states, so the initial partition based on ρ is already the final partition.

8 Conclusions and Future Work

We have introduced a way of representing finite automata which uses relative state references and border functions. Border functions make it possible to concisely represent interval-based transition functions. Our representation is more complicated than the standard representation in text books (like [1, 14]) and the proofs are slightly harder, but the algorithms are not, and the representation can be used in practice without further adaptation. We have implemented our representation and used it in practice. We gave a presentation about it, together with our parser generation tool, at the C^{++} Now conference. The implementation is available from [12].

On the practical level, we make the threshold for using our automated tools as low as possible. In the simplest case, one compiles the library, defines a classifier in code by means of regular expressions, and calls a default function for classification. Constructing classifiers in code has the advantage that the user does not need to learn a dedicated syntax, and that construction of classifiers has full flexibility.

Our implementation does not construct a complete tokenizer. This is important, because in our experience this is the obstacle that stopped us from using an existing tokenizer generator tool. There is always something in the language that cannot be handled by an automatically generated tokenizer. Therefore, in our implementation, we automated only the classification process, and leave all remaining implementation to the user. In practice, not much additional code needs to be written. If one needs efficiency, one can create an executable classifier in C^{++}. Both the default classifier and the C^{++} classifier can be compiled with any input source which satisfies a small set of interface requirements.

In the future, we plan to look into full Boolean operations (extend regular expressions with intersection and negation), or more advanced matching techniques, as specified by POSIX.

The final point that needs consideration is the use of compile time computation. Compile time computation was introduced in C^{++}-11 with the aim of allowing more general functions in declarations, primarily for the computation of the size of a fixed-size array. Since then, the restrictions on compile time computation have gradually been relaxed, and nowadays, it is possible to convert a regular expression represented as an array of characters into a table-based DFA at compile-time. This was implemented in the CTRE library ([6]). We did not try to make our implementation suitable for compile time computation, because it would result in reduced expressivity in the code that constructs the acceptors. In addition, the experiments with RE2C imply that directly coded automata are an order of magnitude faster than table-based automata ([4]).

9 Acknowledgements

This work gained from comments by Witold Charatonik and Cláudia Nalon. We thank Nazarbayev University for supporting this research through the Faculty Development Competitive Research Grant Program (FDCRGP) number 021220FD1651.

References

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi & Jeffrey D. Ullman (2007): *Compilers (Principles, Techniques and Tools)*. Pearson, Addison Wesley.

[2] Manuel Baclet & Claire Pagetti (2006): *Around Hopcroft's Algorithm*. In Oscar Ibarra & Hsu-Chun Yen, editors: *Implementation and Application of Automata*, LNCS, Springer Verlag, pp. 114–125, doi:10.1007/11812128_12.

[3] Markus Boerger, Peter Bumbulis, Dan Nuffer, Ulya Trofimovich & Brian Young (2003-2021): *re2c System*. `https://re2c.org/`.

[4] Klaus Brouwer, Wolfgang Gellerich & Erhard Ploederer (1998): *Myths and Facts about the Efficient Implementation of Finite Automata and Lexical Analysis*. In K. Koskimies, editor: *Compiler Construction (CC 1998)*, LNCS 1383, Springer, pp. 1–15, doi:10.1007/BFb0026419.

[5] The Unicode Consortium: *Unicode*. `https://home.unicode.org/`.

[6] Hana Dusíková (2019-): *CTRE (Compile-Time Regular Expressions) Library*. `https://compile-time.re/`.

[7] John E. Hopcroft (1971): *An n.log(n) algorithm for minimizing the states in a finite automaton*. The theory of machines and computations 43, pp. 189–196, doi:10.1016/B978-0-12-417750-5.50022-1.

[8] John E. Hopcroft, Rajeev Motwani & Jeffrey D. Ullman (2006): *Introduction to Automata Theory, Languages, and Computation*, 3d edition. Pearson, Addison Wesley.

[9] Michael E Lesk & Eric Schmidt (1975): *Lex: A lexical analyzer generator*.

[10] Desheng Liu, Zhiping Huang, Yimeng Zhang, Xiaojun Guo & Shaojing Su (2016): *Efficient Deterministic Finite Automata Minimization Based on Backward Depth Information*. PLOS ONE 11(11), pp. 59–78, doi:10.1371/journal.pone.0165864.

[11] Hans de Nivelle (2021): *A Recursive Inclusion Checker for Recursively Defined Subtypes*. Modeling and Analysis of Information Systems 28(4), pp. 414–433, doi:10.18255/1818-1015-2021-4-414-433. Available at `https://www.mais-journal.ru/jour/article/view/1568`.

[12] Hans de Nivelle & Dina Muktubayeva (2021): *Tokenizer Generation*. `http://www.compiler-tools.eu/`.

[13] Andrei Păun, Mihaela Păun & Alfonso Rodríguez-Páton (2009): *On the Hopcroft's minimization technique for DFA and DFCA*. theoretical computer science, pp. 2424–2430, doi:10.1016/j.tcs.2009.02.034.

[14] Michael Sipser (2013): *Introduction to the Theory of Computation (Third Edition)*. CENGAGE Learning.