RegularIMP: an imperative calculus to describe regular languages

Soroush Aghajani, Emma Kelminson, and
Tiago Cogumbreiro[0000−0002−3209−9258]

University of Massachusetts Boston
{soroush.aghajani001,emma.kelminson001,tiago.cogumbreiro}@umb.edu

August 4, 2024

Abstract. Formal Languages and Automata (FLA) is a fundamental course of the undergraduate computer science degree. A challenge of this curriculum is conveying theoretical intuitions to students more interested in practical aspects of programming. Our approach is to reframe FLA material in terms of a programming language that is familiar to students. To this end, we propose RegularIMP, an imperative programming language that expresses regular languages. We present the syntax and an operational semantics. We also show how to transform a RegularIMP statement into a nondeterministic finite automaton, and a regular expression into a RegularIMP statement.

1 Introduction

Formal Languages and Automata (FLA) is a core subject of the undergraduate degree curriculum in Computer Science [6]. There are multiple techniques aimed at improving the instruction of this material. Interactive FLA editors, such as [9, 14, 19], allow instructors and students to design an automaton by dragging and dropping states and transitions and then test the automaton against a set of inputs. Theory courses benefit from relating the material being taught to other parts of the curriculum. John MacCormick explores the idea of emphasizing search problems written in a real programming language rather than decision problems [13]. Similarly, Blanco Arbe *et al.* propose an approach of creating web forms using regular expressions in JavaScript [2].

This paper takes a novel approach to present regular languages using practical use cases. We argue that finite automata are too abstract to convincingly relate have students perceive an automaton or as a program. Similarly, although matching regular expressions are used commonly in software development, this technique is too specialized to make the point that regular languages represent classes of programs. This paper presents an imperative language, such as C or Python, that can describe a regular language.

Education. We introduce RegularIMP, a core language with a single program variable, a primitive to read a character from the standard input akin to function `getchar` from the C standard library, assertions, a source of nondeterminism akin to function `rand` from the C standard library, and the usual

control-flow primitives of imperative languages (such as loops and conditionals). *Programs in our language* behave *like imperative programs that students already know*, unlike finite automata and regular expressions. We want students to exercise *forward reasoning* (show that a program consumes a certain input) and *backward reasoning* (given that a program consumed a certain input show that the input). The goal of RegularIMP is to have students reason about the behavior of programs, not to serve as an alternative theory to regular expressions, or to declare string matching algorithms.

Theory. In this paper, we present a translation from RegularIMP to nondeterministic automata and from regular expressions to RegularIMP. Our translation is the first step towards showing that a regular language can indeed be expressed with RegularIMP.

Functional programming. We present a tool called `RegularC`[1] that implements our translation function. By converting a RegularIMP program into a nondeterministic finite automaton, students can visualize the behavior of programs, which can uncover surprising misunderstandings.

In summary, our paper makes the following contributions.

- We formalize the syntax of RegularIMP (§4) and give a big-step operational semantics (§5).
- We present a translation from RegularIMP into nondeterministic-finite automata (NFA) (§6).
- We present a translation from a regular expression into RegularIMP (§7).
- We introduce `RegularC`, a compiler-based framework to study computability, written in OCaml, and discuss some implementation details (§8).

The remainder of our paper is as follows. Section 2 details the state of the art, Section 3 overviews our approach, and Section 9 concludes our paper.

2 Related Work

Graphical FLA workbenches One effective way to teach students about FLA is with interactive editors. JFLAP [9, 19], OpenFLAP [14], ComVis [10], and JCT [18] allow declaring, visualizing, and exercising the inputs of automata. RegeXex [3] is an interactive system to help students write regular expressions, by providing examples about which strings are not being matched and which strings should be matched. GUItar [1] provides a GUI to interactively draw state diagrams and export the automaton to multiple well known file formats.

Application Programming Interface (API) Another approach to teaching FLA is to offer a programming interface. Students can program design an automaton or a regular expression using a programming language and then apply common operations and transformations. There are many works for symbolic manipulation of FLA with an API [4, 15, 16, 17, 21, 22, 23].

[1] `https://gitlab.com/umb-svl/regular-c`

Language-based approaches While our approach is to offer a programming model that can express regular languages, other works directly offer a domain-specific language to declare an automaton. Knuth and Bigelow introduce a programming language to program a stack automaton [11]. Chakraborty *et al.* develop a toolkit that takes an automaton specification language called Finite Automaton Description Language (FADL) and offers simulation and conversion to different kinds of automata. Cogumbreiro and Blike present a domain-specific language to style and visualize automata [5].

3 Programs to discuss FLA

In this section we motivate the use of programs and program behavior to discuss FLA, with an emphasis on using imperative programs to describe regular languages. We also motivate the design principles behind our calculus, RegularIMP, by means of examples.

Figure 1 lists a C program that reads a string from the standard input and returns true if, and only if, the last character of the input is `'0'`. Note how Figure 1 relies only on features taught at the introductory-level of any programming course. An FLA instructor can use the program in Figure 1 to:

- ask the student to reason about the set of all possible outputs given an input;
- ask the student to reason about the set of all possible inputs given an output;
- visually illustrate the behavior of programs, as depicted by the state diagram in Figure 1;
- motivate the use of regular expressions and finite automata as a concise description of program behavior (and consequently of formal languages);
- show that the techniques we use to *debug* programs are same techniques we use to prove the correctness of programs (*i.e.*, reasoning about the set of possible inputs and outputs);

Figure 2 lists a program that only executes successfully, that is, without aborting, when the last character of the input is `'0'`. Importantly, the program may abort even when the last character of the input is `'0'`, unlike the program in Figure 1. An instructor can compare both programs and reflect on a key difference between a *deterministic* finite automaton and a *nondeterministic* finite automaton: in the former there is a single path (derivation) per input; in the latter there may exist multiple paths (derivation) per input.

The program in Figure 1 is more realistic than Figure 2. It is more common to use the `return` keyword to communicate the success of a computation instead of `assert`. Moreover, Figure 2 may fail even with a "good" input. However, if we only consider the set of "successful" computations, then both programs accept the same inputs. Since this paper focuses on introducing a small calculus, called RegularIMP, that can express a regular language, and since formalizing `assert` leads to fewer rules than formalizing `return`, our paper focuses on the style of programs given in Figure 2.

```c
#include <stdbool.h>
#include <stdio.h>
int main() {
  int i;
  i = getchar();
  while(i != EOF) {
    if (i == '0') {
      i = getchar();
      if (i == EOF) {
        return true;
      }
    } else {
      i = getchar();
    }
  }
  return false;
}
```

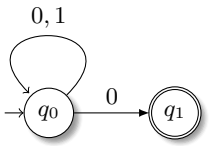

Fig. 1. On the left-hand side, we list a C program that returns `true` only when the last character of the standard input is `'0'`. Function `getchar` reads a character from the standard input. Constant `EOF` denotes that the there are no more characters available in the standard input. On the right-hand side, we depict a state diagram of a nondeterministic diagram that recognizes the set of strings that end with character 0.

4 Syntax

Figure 3 introduces the syntax of expressions and statements. Let Σ be a set of atoms.[2] Let c range over the set $\Sigma \cup \{\bot\}$ such that $\bot \notin \Sigma$, where $\bot$ denotes the end of the input. The syntax of expressions $e \in \mathcal{E}$ consists of: a boolean $b \in \{\texttt{tt}, \texttt{ff}\}$, the conjunction operator, the disjunction operator; $\texttt{i} = c$ tests whether a global variable `i` equals some character c; `rnd` yields a random boolean. The syntax for a statement s is as follows. Statement $\texttt{i} \leftarrow \texttt{read}$ consumes one character from the input and assigns global variable `i` to the consumed character. An $\texttt{assert}(e)$ aborts when expression e evaluates to false. The remaining statements are standard in imperative programming languages.

Example 1 (Statement that consumes strings that terminate with 0*).*

$$\texttt{while}(\texttt{rnd})\{\texttt{i} \leftarrow \texttt{read}\}; \texttt{assert}(\texttt{i} = 0); \texttt{i} \leftarrow \texttt{read}; \texttt{assert}(\texttt{i} = \bot)$$

In Example 1, we rewrite the code in Figure 2 using our syntax. Statement $\texttt{while}(\texttt{rnd})\{\texttt{i} \leftarrow \texttt{read}\}$ consumes an arbitrary number of characters from the input, as `rnd` returns a random boolean each time it is evaluated. At each iteration the assignment overwrites variable `i` and consumes another character

[2] Note that the abstract syntax of RegularIMP differs considerably from the concrete syntax of C listed in Section 3.

```c
#include <stdio.h>
#include <assert.h>
int main() {
  int i;
  while(rand() % 2) {
    i = getchar();
  }
  assert(i == '0');
  i = getchar();
  assert(i == EOF);
}
```

Fig. 2. A C program executes successfully only when character `'0'` appears in the last position of the input. Expression `rand() % 2` returns a random boolean. Function `assert` aborts the program execution when the given expression returns false.

$$
\begin{aligned}
e &::= b \mid \texttt{i} = c \mid \texttt{rnd} \mid \texttt{not}(e) \mid \texttt{and}(e, e) \mid \texttt{or}(e, e) \\
s &::= \texttt{i} \leftarrow \texttt{read} \mid \texttt{assert}(e) \mid s; s \mid \texttt{if}(e)\{s\}\texttt{el}\{s\} \mid \texttt{while}(e)\{s\}
\end{aligned}
$$

Fig. 3. Syntax of expressions and statements.

from the input. After the loop terminates, we ensure that the last read is 0. Finally, we read another character and ensure that the input is empty.

5 Semantics

In this section we give a semantics for expressions and another for statements. The semantics for statements captures *consuming* the input.

Expressions In Figure 4, we give a big-step operational semantics of expressions with judgement $(e, c) \Downarrow b$ where given an expression e and a character c (the state of variable `i`), we obtain a boolean b. The rules are straightforward. A boolean b evaluates to itself. Rules CHAR-EQ states that when the state of `i` is c and we are testing `i` against c, then the test yields true. Otherwise (Rule CHAR-NEQ), the expression yields false. Expression `rnd` returns an arbitrary boolean b, which makes the semantics of expressions nondeterministic. The negation, conjunction, and disjunction are straightforward.

Statements Figure 5 introduces the semantics of statements. Let $I ::= [] \mid c :: I$ denote a sequence of characters where $[]$ denotes an empty sequence and $c :: I$ denotes adding a character c to sequence I. We say that statement s takes (c, I) and outputs (c', I'), notation $(s, c, I) \Downarrow (c', I')$, where character c represents the state of `i`, sequence of characters I represents the input available, character c'

$$\text{BOOL}\ \frac{}{(b, c) \Downarrow b} \qquad \text{CHAR-EQ}\ \frac{}{(\mathtt{i} = c, c) \Downarrow \mathtt{tt}} \qquad \text{CHAR-NEQ}\ \frac{c_1 \neq c_2}{(\mathtt{i} = c_2, c_1) \Downarrow \mathtt{ff}} \qquad \text{FLIP}\ \frac{b \in \{\mathtt{tt}, \mathtt{ff}\}}{(\mathtt{rnd}, c) \Downarrow b}$$

$$\text{NOT}\ \frac{(e, c) \Downarrow b}{(\mathtt{not}(e), c) \Downarrow \neg b} \qquad \text{AND}\ \frac{(e_1, c) \Downarrow b_1 \qquad (e_2, c) \Downarrow b_2}{(\mathtt{and}(e_1, e_2), c) \Downarrow b_1 \wedge b_2} \qquad \text{OR}\ \frac{(e_1, c) \Downarrow b_1 \qquad (e_2, c) \Downarrow b_2}{(\mathtt{or}(e_1, e_2), c) \Downarrow b_1 \vee b_2}$$

Fig. 4. Semantics of expressions $\boxed{(e, c) \Downarrow b}$.

$$\text{READ-1}\ \frac{}{(\mathtt{i} \leftarrow \mathtt{read}, c_1, c_2 :: I) \Downarrow (c_2, I)} \qquad \text{READ-2}\ \frac{}{(\mathtt{i} \leftarrow \mathtt{read}, c, []) \Downarrow (\bot, [])}$$

$$\text{ASSERT}\ \frac{(e, c) \Downarrow \mathtt{tt}}{(\mathtt{assert}(e), c, I) \Downarrow (c, I)} \qquad \text{SEQ}\ \frac{(s_1, c_1, I_1) \Downarrow (c_2, I_2) \qquad (s_2, c_2, I_2) \Downarrow (c_3, I_3)}{(s_1; s_2, c_1, I_1) \Downarrow (c_3, I_3)}$$

$$\text{IF-T}\ \frac{(e, c_1) \Downarrow \mathtt{tt} \qquad (s_1, c_1, I_1) \Downarrow (c_2, I_2)}{(\mathtt{if}(e)\{s_1\}\mathtt{el}\{s_2\}, c_1, I_1) \Downarrow (c_2, I_2)} \qquad \text{IF-F}\ \frac{(e, c_1) \Downarrow \mathtt{ff} \qquad (s_2, c_1, I_1) \Downarrow (c_2, I_2)}{(\mathtt{if}(e)\{s_1\}\mathtt{el}\{s_2\}, c_1, I_1) \Downarrow (c_2, I_2)}$$

$$\text{WHILE-T}\ \frac{(e, c_1) \Downarrow \mathtt{tt} \qquad (s; \mathtt{while}(e)\{s\}, c_1, I_1) \Downarrow (c_2, I_2)}{(\mathtt{while}(e)\{s\}, c_1, I_1) \Downarrow (c_2, I_2)} \qquad \text{WHILE-F}\ \frac{(e, c) \Downarrow \mathtt{ff}}{(\mathtt{while}(e)\{s\}, c, I) \Downarrow (c, I)}$$

Fig. 5. Semantics for statements $\boxed{(s, c, I) \Downarrow (c, I)}$.

represents the next state of `i`, and, finally, sequence I' represents the remaining input. Rule READ-1 states that when the input is $c_2 :: I$ where first character of the input is character c_2, we assign `i` to c_2 setting the remaining input as I. Rule READ-2 governs that when the input is empty $[]$, then we assign `i` to $\bot$, leaving the input unchanged. The semantics allows for multiple reads after the input is empty. Rule ASSERT only executes $\mathtt{assert}(e)$ when expression e evaluates to `tt`. Rule SEQ proceeds as usual, sequencing s_1 amounts to executing s_1 and obtaining a character c_2 and an input I_2 that is then used to execute s_2. Rule IF-T executes the body of the conditional s_1 when condition e evaluates to true; otherwise, executes s_2 (Rule IF-F). Similarly, Rule WHILE-T unfolds the loop body s when the condition of a loop e evaluates to `tt`; otherwise, when e evaluates to `ff` the value of `i` and the input is left unchanged.

For instance, we have that $\texttt{while}(\texttt{rnd})\{\texttt{i} \leftarrow \texttt{read}\}$ takes $(\bot, [1,0])$ and outputs $(0, [])$.

$$\dfrac{\dfrac{\overline{(\texttt{i} \leftarrow \texttt{read}, \bot, [1,0]) \Downarrow (1, [0])} \qquad \dfrac{\cdots}{(\texttt{while}(\texttt{rnd})\{\texttt{i} \leftarrow \texttt{read}\}, 1, [0]) \Downarrow (0, [])}}{(\texttt{i} \leftarrow \texttt{read}; \texttt{while}(\texttt{rnd})\{\texttt{i} \leftarrow \texttt{read}\}, \bot, [1,0]) \Downarrow (0, [])}}{(\texttt{while}(\texttt{rnd})\{\texttt{i} \leftarrow \texttt{read}\}, \bot, [1,0]) \Downarrow (0, [])} \tag{1}$$

Additionally, we have that

$$\dfrac{(\texttt{i} = 0, 0) \Downarrow \texttt{tt} \qquad \dfrac{\dfrac{\overline{(\texttt{i} \leftarrow \texttt{read}, 0, []) \Downarrow (\bot, [])} \qquad \dfrac{(\texttt{i} = \bot, \bot) \Downarrow \texttt{tt}}{(\texttt{assert}(\texttt{i} = \bot), \bot, []) \Downarrow (\bot, [])}}{(\texttt{i} \leftarrow \texttt{read}; \texttt{assert}(\texttt{i} = \bot), 0, []) \Downarrow (\bot, [])}}{(\texttt{assert}(\texttt{i} = 0); \texttt{i} \leftarrow \texttt{read}; \texttt{assert}(\texttt{i} = \bot), 0, []) \Downarrow (\bot, [])} \tag{2}$$

Thus, we can show that our running example consumes $(\bot, [1,0])$ and outputs $(\bot, [])$.

$$\dfrac{\begin{array}{c}(\texttt{while}(\texttt{rnd})\{\texttt{i} \leftarrow \texttt{read}\}, \bot, [1,0]) \Downarrow (0, []) \\ (\texttt{assert}(\texttt{i} = 0); \texttt{i} \leftarrow \texttt{read}; \texttt{assert}(\texttt{i} = \bot), 0, []) \Downarrow (\bot, [])\end{array}}{(\texttt{while}(\texttt{rnd})\{\texttt{i} \leftarrow \texttt{read}\}; \texttt{assert}(\texttt{i} = 0); \texttt{i} \leftarrow \texttt{read}; \texttt{assert}(\texttt{i} = \bot), \bot, [1,0]) \Downarrow (\bot, [])} \tag{3}$$

We are now ready to define the notion of accepting an input. A statement s accepts an input I, which starts with $\texttt{i}$ assigned to $\bot$, and outputs some character c and some input I'.

Definition 1 (Accepting input). *We say that statement s accepts input I if $(s, \bot, I) \Downarrow (c, I')$ holds for some I' and c.*

From Equation (3), we have that our running example accepts input $[1,0]$. Importantly, accepting an input does not require the consumption of that input, *i.e.*, it is easy to show that statement $\texttt{assert}(\texttt{tt})$ accepts any input without consuming any characters.

6 From RegularIMP to NFA

To translate a RegularIMP program into an NFA, we start from a graph-representation of RegularIMP known as a control-flow graph (CFG). Deriving a CFG directly from the semantics in Figure 5 is beyond the scope of this paper; we refer the reader to some recent developments of this approach in [12].

6.1 Control-flow Graphs

A CFG is a labelled and directed graph $G = (V, \Sigma, A, v_0, B, F)$ that consists of a nonempty finite set of vertices V, a finite set of the alphabet Σ, a finite set of arcs A, an initial vertex v_0, a set of branch vertices B (where $B \subseteq V$), and a

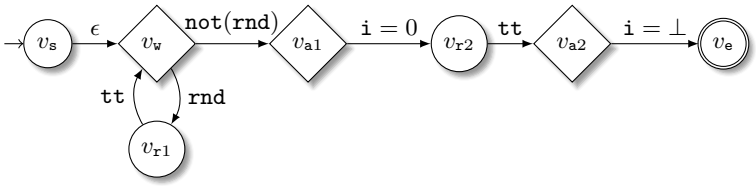

Fig. 6. A CFG of our running example.

set of return vertices F (where $F \subseteq V$). Let v range over the set of vertices $\mathcal{V}$, where $V \subseteq \mathcal{V}$. An arc $a = (v, l, v')$ is directed from the *head* v to the *tail* v' and labelled by l We have that meta-variable l ranges over set $\mathcal{E} \cup \{\epsilon\}$ (either an expression or ϵ). A branch-vertex $v \in B$ is either: an if, a while, or an assert.

Figure 6 illustrates a CFG of our running example. A vertex with an incoming arrow denotes the initial node ($v_{\mathtt{s}}$). Vertices with a diamond shape are branch vertices. Vertex $v_{\mathtt{w}}$ represents the while loop. Vertex $v_{\mathtt{r1}}$ represents the loop body $\mathtt{i} \leftarrow \mathtt{read}$. Vertex $v_{\mathtt{a1}}$ represents $\mathtt{assert}(\mathtt{i} = 0)$. Vertices with a double-circle are final, for instance, $v_{\mathtt{e}}$. The labels define the conditions that allow the execution to go from one state to another. An ϵ allows linking the initial vertex to multiple vertices — much like what we have with finite automata. In Figure 6, the initial vertex $v_{\mathtt{s}}$ only links to $v_{\mathtt{w}}$, which signifies that the execution effectively starts at $v_{\mathtt{w}}$. Every other edge is labelled with an expression e. We use $\mathtt{tt}$ to signify an unconditional transition. For instance, the loop body $v_{\mathtt{r1}}$ continues unconditionally to the beginning of the loop $v_{\mathtt{w}}$, so we have a label $\mathtt{tt}$.

We now detail the transformation from a CFG into a normalized CFG which removes all branch vertices. Let $A|_v$ select every arc that mentions vertex v, defined as

$$A|_v = \{(v_1, l, v_2) \mid (v_1, l, v_2) \in A \wedge (v_1 = v \vee v_2 = v)\}$$

Let $\mathit{skip}(A, v)$ return the set of all arcs that skip v. The intuition is if v_1 arrives at v with label l_1, and v_2 departs from v with label l_2, then we generate an arc that connects v_1 to v_2 with a conjunction of both labels l_1 and l_2.

$$\begin{aligned}
\mathit{skip}(A, v) &= \{(v_1, \mathtt{and}^\star(l_1, l_2), v_2) \mid (v_1, l_1, v) \in A \wedge (v, l_2, v_2) \in A\} \\
\mathtt{and}^\star(e_1, e_2) &= \mathtt{and}(e_1, e_2) \\
\mathtt{and}^\star(\epsilon, l) &= \mathtt{and}^\star(l, \epsilon) = l
\end{aligned}$$

Finally, we introduce a recursive definition of the normalization of CFGs that iteratively removes each branch vertex until there are none left.

$$\begin{aligned}
\mathit{norm}((V, \Sigma, A, v_0, \emptyset, F)) &= (V, \Sigma, A, v_0, \emptyset, F) \\
\mathit{norm}((V, \Sigma, A, v_0, B, F)) &= \mathit{norm}(V \setminus \{v\}, \Sigma, A', v_0, B \setminus \{v\}, F) \quad \text{if } v \in B \\
&\qquad \text{where } A' = A \setminus A|_v \cup \mathit{skip}(A, v)
\end{aligned}$$

For instance, given the CFG in Figure 6 we obtain the normalized CFG illustrated in Figure 7. Note that vertex $v_{\mathtt{e}}$ is an unreachable state in the resulting NFA. Unreachable states are not depicted in state diagrams.

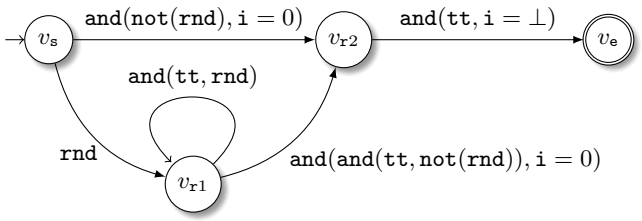

Fig. 7. A normalized CFG of our running example.

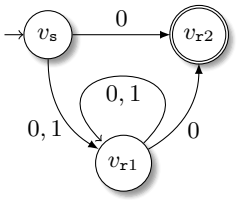

Fig. 8. An NFA of our running example where $\Sigma = \{0, 1\}$.

6.2 Generating a Nondeterministic Finite Automaton

We are now ready to define a function that translates a normalized CFG into a Nondeterministic Finite Automaton (NFA). We follow the usual notion of an NFA [20]. An NFA is defined as $N = (V, \Sigma, \delta, v_0, F)$ where V is a finite set of states, Σ is the finite alphabet of the NFA, $\delta \colon V \times (\Sigma \cup \{\epsilon\}) \mapsto \mathcal{P}(V)$ is a transition function, $v_0 \in V$ is the initial state, and $F \subseteq V$ is the set of final states, where $\mathcal{P}(\cdot)$ is the power set function.

Function $[\![G]\!]$ takes as input a normalized CFG G and outputs an NFA N, defined as follows. Any arc (v_1, e, v_2) where the labelled expression e evaluates to true when $\texttt{i} = c$ is a transition in N. There is a self transition in N for any return vertex $v_1 \in F$ in the CFG. Any arc in the CFG labelled with an ϵ becomes an ϵ-transition in N. Any final vertex in the CFG is a final state in the NFA. Additionally, any read vertex v_1 that reaches a final vertex $v_2 \in F$ is also a final state in N when its expression e evaluates to true when $\texttt{i} = \bot$.

$$
\begin{aligned}
[\![(V, A, v_0, \emptyset, F)]\!] &= (V, \Sigma, \delta, v_0, F') \\
\text{where } \delta(v_1, c) &= \{v_2 \mid (v_1, e, v_2) \in A \wedge c \in \Sigma \wedge (e, c) \Downarrow \texttt{tt}\} \cup (\{v_1\} \cap F) \\
\delta(v_1, \epsilon) &= \{v_2 \mid (v_1, \epsilon, v_2) \in A\} \\
F' &= F \cup \{v_1 \mid v_2 \in F \wedge (v_1, e, v_2) \in A \wedge (e, \bot) \Downarrow \texttt{tt}\}
\end{aligned}
$$

Figure 8 is known as a *state diagram* and depicts the NFA that results from translating the normalized CFG in Figure 7.

7 From Regular Expressions to RegularIMP

We introduce a translation from regular expressions into statements of RegularIMP. Let the following define the syntax of regular expressions.

$$R ::= \epsilon \mid c \mid \emptyset \mid R + R \mid R \cdot R \mid R^\star$$

We have ϵ denote the empty string, c denote the string with a single character c, $\emptyset$ rejects all strings, $R_1 + R_2$ (set union) accepts any string from R_1 and from R_2, $R_1 \cdot R_2$ concatenates any string accepted by R_1 with the strings accepted by R_2, finally, $R^\star$ denotes the Kleene star operator. We now define a translation function $[\![\cdot]\!]$ from regular expression R into statements s.

$$\begin{aligned}
[\![\epsilon]\!] &= \texttt{assert}(\texttt{tt}) \\
[\![c]\!] &= \texttt{i} \leftarrow \texttt{read}; \texttt{assert}(\texttt{i} = c) \\
[\![\emptyset]\!] &= \texttt{assert}(\texttt{ff}) \\
[\![R_1 + R_2]\!] &= \texttt{if}(\texttt{rnd})\{[\![R_1]\!]\}\texttt{el}\{[\![R_2]\!]\} \\
[\![R_1 \cdot R_2]\!] &= [\![R_1]\!]; [\![R_2]\!] \\
[\![R_1^\star]\!] &= \texttt{while}(\texttt{rnd})\{[\![R]\!]\}
\end{aligned}$$

We have that ϵ yields the program that leaves its input intact. When translating c the program reads one character and ensures that c is read. We translate $\emptyset$ by aborting the computation. Union becomes a conditional on an arbitrary boolean. Concatenation of regular expressions becomes the sequencing. Finally, the Kleene star operator becomes a loop runs an arbitrary number of iterations.

Given that the semantics of RegularIMP accepts any input even without consuming it, we must ensure that after translating a regular expression with $[\![\cdot]\!]$ we reach the end of the input. Thus, we introduce function $rex(\cdot)$ that takes a regular expression and produces a statement that recognizes the same inputs.

$$rex(R) = [\![R]\!]; \texttt{i} \leftarrow \texttt{read}; \texttt{assert}(\texttt{i} = \bot)$$

We conclude by showing the translation of a regular expression that also accepts inputs that end with 0, *i.e.*, our running example.

$$\begin{aligned}
& rex((0+1)^\star \cdot 0) \\
=& [\![(0+1)^\star \cdot 0]\!]; expect(\bot) \\
=& [\![(0+1)^\star]\!]; [\![0]\!]; expect(\bot) \\
=& \texttt{while}(\texttt{rnd})\{[\![(0+1)]\!]\}; expect(0); expect(\bot) \\
=& \texttt{while}(\texttt{rnd})\{\texttt{if}(\texttt{rnd})\{[\![0]\!]\}\texttt{el}\{[\![1]\!]\}\}; expect(0); expect(\bot) \\
=& \texttt{while}(\texttt{rnd})\{\texttt{if}(\texttt{rnd})\{expect(0)\}\texttt{el}\{expect(1)\}\}; expect(0); expect(\bot) \\
& \text{where } expect(c) = \texttt{i} \leftarrow \texttt{read}; \texttt{assert}(\texttt{i} = c)
\end{aligned}$$

8 `RegularC`: a framework to study computability

In this section, we discuss our tool `RegularC` that takes as an input a RegularIMP statement and allows students to visualize its behavior. Our tool implements the translation presented in Section 6 as a compiler, a pipeline of transformations takes a RegularIMP statement as input and outputs a nondeterministic finite automaton. `RegularC` implements the following pipeline:

1. transform a statement into a control-flow graph
2. normalize a control-flow graph
3. for each arc remove `rnd`
4. for each arc replace $\texttt{i} = \bot$ by `ff`; add accepting nodes that result from $\texttt{i} = \bot$
5. converts each expression without `rnd` and $\texttt{i} = \bot$ into a set of characters (which represents the characters of a state transition)

Students can visualize any step of the pipeline, Steps (1–4) are variations of a control-flow graph, Step (5) outputs a state diagram of the automaton. Step (2) implements function $norm(G)$. Steps (3–5) implement $[\![G]\!]$, each step gradually transforms the CFG until we obtain the finite automaton.

The main challenge of implementing $[\![G]\!]$ stems from the generation of non-ϵ transitions, that we recall below:

$$\delta(v_1, c) = \{v_2 \mid (v_1, e, v_2) \in A \wedge c \in \Sigma \wedge (e, c) \Downarrow \texttt{tt}\} \cup (\{v_1\} \cap F)$$

Proposition $(e, c) \Downarrow \texttt{tt}$ evaluates nondeterministically, due to `rnd`, so an implementation must decide if there is at least one derivation that returns `tt`. Step (3) replaces each expression e in the CFG by a simpler expression e' that can be evaluated deterministically, where $(e, c) \Downarrow \texttt{tt} \iff (e', c) \Downarrow \texttt{tt}$ for any c.

Step (3) implements the simplification of expressions with `remove_rnd`. Function `iter` generates a sequence of expressions, one per valuation of `rnd`, which are combined with a disjunction by function `remove_rnd`.

```
let remove_rnd e =
  let rec iter =
    function
    | b -> List.to_seq [b]
    | i = c -> List.to_seq [i = c]
    | rnd -> List.to_seq [tt; ff]
    | not(e) -> iter e |> Seq.map (fun e' -> not(e'))
    | and(e1, e2) -> Seq.product (iter e1) (iter e2)
                     |> Seq.map (fun (e1', e2') -> and(e1', e2'))
    | or(e1, e2) -> Seq.product (iter e1) (iter e2)
                     |> Seq.map (fun (e1', e2') -> or(e1', e2'))
  in
  iter e |> Seq.fold_left (fun (e1' e2') -> or(e1', e2')) ff
```

For instance, `remove_rnd` $\texttt{and}(\texttt{i} = c, \texttt{rnd})$ outputs $\texttt{or}(\texttt{and}(\texttt{i} = c, \texttt{tt}), \texttt{and}(\texttt{i} = c, \texttt{ff}))$.

It is important to note that after Step (3), since `rnd` is absent, the implementation of $(e, c) \Downarrow \texttt{tt}$ becomes straightforward:

```
let rec eval_true c =
  function
  | b -> b
  | i = c' -> c = c'
  | not(e) -> not (eval_true e)
  | and(e1, e2) -> eval_true e1 && eval_true e2
  | or(e1, e2) -> eval_true e1 || eval_true e2
```

9 Conclusion and Future Work

We present RegularIMP, a core imperative calculus that expresses regular languages. We introduce the syntax and operational semantics of RegularIMP. To establish that RegularIMP is indeed able to describe regular languages, we give a translation function to convert a statement of RegularIMP into an equivalent nondeterministic finite automaton, and from a regular expression to a RegularIMP statement. We present `RegularC`, an implementation of the translation written in OCaml that allows students to visualize the behavior of RegularIMP.

Future work includes developing well known verification techniques to RegularIMP, with the goal of introducing software verification to undergraduate students. For instance, we are using RegularIMP to teach symbolic execution: we are showing how to implement a symbolic execution engine that can execute a RegularIMP statement. Additionally, we want to introduce model checking of RegularIMP statements by following [7, 8].

Acknowledgments. We thank the anonymous reviewers of the 2024 edition of Trends in Functional Programming in Education (TFPIE'24) for their insightful feedback on earlier versions of this work. This material is based upon work supported by the National Science Foundation under Grant No. 2204986.

Bibliography

[1] Almeida, A., Almeida, M., Alves, J., Moreira, N., Reis, R.: FAdo and GUItar: Tools for automata manipulation and visualization. In: Proceedings of CIAA. pp. 65–74. Springer (2009). https://doi.org/10.1007/978-3-642-02979-0_10

[2] Blanco Arbe, J.M., Ortega, A.S., Mártinez-Conde, J.I.n.: Formal languages through web forms and regular expressions. SIGCSE Bulletin **39**(4), 100–104 (2007). https://doi.org/10.1145/1345375.1345424

[3] Brown, C.W., Hardisty, E.A.: RegeXeX: An interactive system providing regular expression exercises. In: Proceedings of SIGCSE. p. 445–449. ACM, New York, NY, USA (2007). https://doi.org/10.1145/1227310.1227462

[4] Champarnaud, J.M., Hansel, G.: AUTOMATE: a computing package for automata and finite semigroups. Journal of Symbolic Computation **12**(2), 197–220 (1991). https://doi.org/10.1016/S0747-7171(08)80125-3

[5] Cogumbreiro, T., Blike, G.: Gidayu: Visualizing automata and their computations. In: Proceedings of ITiCSE. pp. 110–116. ACM, New York, NY, USA (2022). https://doi.org/10.1145/3502718.3524742

[6] Joint Task Force on Computing Curricula, A.f.C.M.A., Society, I.C.: Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science. ACM (2013). https://doi.org/10.1145/2534860

[7] De Ferro, C.M., Cogumbreiro, T., Martins, F.: Formalizing model inference of MicroPython. In: Proceedings of VERDI. pp. 283–289 (2023). https://doi.org/10.1109/DSN-W58399.2023.00069

[8] de Ferro, C.M., Cogumbreiro, T., Martins, F.: Shelley: A framework for model checking call ordering on hierarchical systems. In: Proceedings of COORDINATION. pp. 93–114. Springer (2023)

[9] Hung, T., Rodger, S.H.: Increasing visualization and interaction in the automata theory course. In: Proceedings of SIGCSE. p. 6–10. ACM (2000). https://doi.org/10.1145/330908.331800

[10] Jovanović, N., Miljković, D., Stamenković, S., Jovanović, Z., Chakraborty, P.: Teaching concepts related to finite automata using ComVis. Computer Applications in Engineering Education **29**(5), 994–1006 (2021). https://doi.org/10.1002/cae.22353

[11] Knuth, D.E., Bigelow, R.H.: Programming language for automata. Journal of the ACM **14**(4), 615–635 (oct 1967). https://doi.org/10.1145/321420.321421

[12] Koppel, J., Kearl, J., Solar-Lezama, A.: Automatically deriving Control-Flow Graph generators from operational semantics. Proceedings of the ACM on Programming Languages **6**(ICFP) (2022). https://doi.org/10.1145/3547648

[13] MacCormick, J.: Using computer programs and search problems for teaching theory of computation. Communications of the ACM **63**(10), 33–35 (2020). https://doi.org/10.1145/3382036

[14] Mohammed, M., Shaffer, C.A., Rodger, S.H.: Teaching formal languages with visualizations and auto-graded exercises. In: Proceedings of the 52nd ACM Technical Symposium on Computer Science Education. p. 569–575. SIGCSE '21, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3408877.3432398, https://doi.org/10.1145/3408877.3432398

[15] Møller, A.: dk.brics.automaton: Finite-state automata and regular expressions for Java (2021), http://www.brics.dk/automaton/

[16] Raymond, D., Wood, D.: Grail: A C++ library for automata and expressions. Journal of Symbolic Computation **17**(4), 341–350 (1994). https://doi.org/10.1006/jsco.1994.1023

[17] Riley, M., Allauzen, C., Jansche, M.: OpenFst: An open-source, weighted finite-state transducer library and its applications to speech and language. In: Proceedings of NAACL-Tutorials. p. 9–10. ACL, USA (2009)

[18] Robinson, M.B., Hamshar, J.A., Novillo, J.E., Duchowski, A.T.: A Java-based tool for reasoning about models of computation through simulating finite automata and turing machines. In: Proceedings of SIGCSE. pp. 105–109. ACM, New York, NY, USA (1999). https://doi.org/10.1145/299649.299704

[19] Rodger, S.H., Bressler, B., Finley, T., Reading, S.: Turning automata theory into a hands-on course. In: Proceedings of SIGCSE. p. 379–383. ACM (2006). https://doi.org/10.1145/1121341.1121459

[20] Sipser, M.: Introduction to the theory of computation. Cengage Learning (2012)

[21] Stoughton, A.: Experimenting with formal languages using Forlan. In: Proceedings of FDPE. p. 41–50. ACM (2008). https://doi.org/10.1145/1411260.1411267

[22] Veanes, M., Bjørner, N.: Symbolic automata: The toolkit. In: Flanagan, C., König, B. (eds.) Proceedings of TACAS. pp. 472–477. Springer (2012). https://doi.org/10.1007/978-3-642-28756-5_33

[23] Watson, B.W.: FIRE lite: FAs and REs in C++. In: Proceedings of WAI. pp. 167–188. Springer, Berlin, Heidelberg (1997)