Design Specification for CS 2800 Assistant

Tavish Mcdonald
tmm248

Rohit Curucundhi
rc563

Elise Weir
enw23

Alice Pham
atp59

Contents

1 System Description

1.1 Vision

CS 2800 marks the introduction to mathematical foundations of computer science through formal proofs and theoretical abstractions of programming languages. As this material proves difficult, we built CS 2800 learning tool to visualize core course concepts with an emphasis on Turing machines and Kleene algebra. The full scope of the course is not a candidate for implementation, and so in our Final Version, we chose to implement the following:

1. As stated above, functionality for simulation of Turing Machines has been implemented, which allows users to configure a machine setup, and visualize what occurs both on the tape, and in the Finite State Control of the machine.

2. Users can input regular expressions and select rules by which to simplify them. They can also click a button which will determine whether the

current regular expression is actually simpler in form than the one previous to the application of any given step.

3. Finally, we have implemented a walk through of RSA encryption, where users are walked through a process of generating large primes and then are guided through how the algorithm encrypts and decrypts messages.

The initial proposal for the machine was as follows:

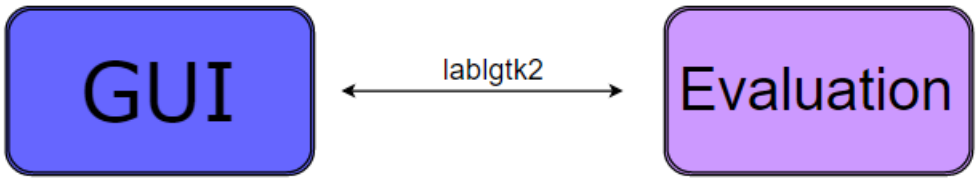
1. Implement a Turing machine that when given a starting input produces and visualizes the individual steps of computation.
2. Implement DFAs and NFAs and visualize computation of a given string with support for the Pumping Lemma.
3. Implement Kleene algebraic rules to simplify regular expressions
4. Time permitting Visualize set theoretic concepts as they relate to functions, cardinality, and probability.

2 Architecture

The overall design of the machine is as a pipe and filter system. We have designed a front-end and back-end system for the purpose of abstraction. Our front-end is the GUI, which serves as the manner in which users interact with the system. The back-end of the system involves evaluation of any necessary components. The following is a listing of the evaluation steps occurring in the back-end of the system:

1. Turing Machines: The `machine.mli` interface provides support to construct a Turing Machine, given all of the necessary fields (described in further detail under system design), and then step that machine one step.
2. RSA Encryption Algorithm: The `rsa.mli` interface allows users to walk through a simulation of the RSA encryption algorithm. It performs the evaluation of all necessary fields for RSA encryption, including the generation of large primes (checking that they are in fact primes using the Miller-Rabin algorithm), the extended Euclidean algorithm, and then finally encryption and decryption.
3. Regular Expression Simplification: The `regex.mli` interface provides support for users to simplify a given regex by one step using a selection of the rules from Kleene Algebra. This involves creating a lexer and parser for regular expressions (another pipe and filter architecture), that then created an AST for regular expressions that was used for all of the computations. Users can also check to see if the step that was taken actually simplified the expression further or not.

C&C Diagram



3 System Design

3.1 Modules

1. **GUI**: This module is the interface for the user to interact with the system. Initially, the GUI opens to a notebook (a type in our GUI Library), which is the home screen for the learning system. From here, users can select from pages that access each of the separate 2800 topics implemented.

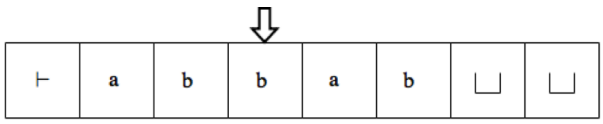


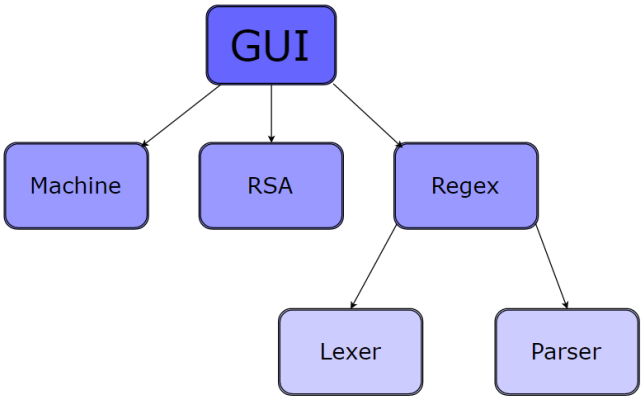
Figure 1: Example of Turing machine GUI representation

2. **Machine:** This module communicates with the GUI when the user selects the Turing Machine simulation window. The module takes the user input of the input alphabet, states, and transition function. The module holds evaluation functions that store user input in an OCaml record and step the machine, updating the machine for visualization as it transforms to the new state.

3. **RSA:** This module contains a suite of functions to interactively describe the RSA encryption and decryption process. The module communicates with the GUI allowing for two modes: One mode allows the user to input their own prime numbers p and q, while the second will generate primes for the user. In the first mode, the Miller-Rabin primality test is employed to probabilistically ensure that the user inputs are indeed prime. This is restricted to small enough inputs. Running in either of the modes outputs a series of guided explanations to the user and displays computations using the user input to the GUI. GUI user input of p, q, and the public key are OCaml strings and are piped to the RSA module. They are then processed according to the RSA Cryptography step and are piped back to the GUI for display.

4. **Regex Simplification** This module communicates with the GUI when the user selects the regex simplification window. The module takes a user input which is a regular expression, and then stores the information as an OCaml string. From here, evaluation is done by first parsing the string to an AST, and then performing any computations necessary with that tree structure. The computations are steps of Kleene Algebra. This module also checks to see if a regular expression is more simplified than a previous version.

3.2 Dependency Diagram



4 Data Types

1. **GUI Type System** In our initial design, we thought it might be a good idea to use JSON objects to transfer information between objects. But given that LablGtk uses OCaml, it was easier and cleaner to pass OCaml types. Instead, our GUI module made direct calls to back-end evaluation modules and used only primitive OCaml types and types that we defined in those modules. The following is a description of the types used in each module.

2. **Turing Machine Type System** The formal definition of a Turing machine is a 7-tuple, $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$
However, not all of these components are essential to Turing Machine evaluation in our implementation. We define type `state` to be a `Q of int`. The `Direction` moved on the tape is a variant type `R | L`. The `input_write` is a `char` and the result of the transition function is the type `step_output` which is a record {`new_state: state; input_write: input_write; direction: direction`}

3. **Regex Type System** The type of a regex is defined as follows:

```
type regex =
  | Empty
  | Char of char
  | Concat of regex * regex
  | Or of regex * regex
  | Star of regex
```

5 External Dependencies

The external libraries and other references we used are listed below by module:

1. **GUI:** We used the LablGTK OCAML GUI package to implement our GUI.
2. **Turing Machine:** No external libraries were used for the implementation of this module. However we relied heavily on *Automata and Computability* by Cornell's own Dexter Kozen to supplement our knowledge of Turing Machines.
3. **RSA:** Again, we did not use any external libraries, but we did cite the following sources:
   (a) http://doctrina.org/How-RSA-Works-With-Examples.html for an overview of the RSA algorithm.
   (b) Wikipedia's article on the Sieve of Eratosthenes for an help implementing an algorithm that would generate large prime numbers for us.
   (c) Rosetta Code's discussion of the Sieve Eratosthenes was used to help develop functions. https://rosettacode.org/wiki/Sieve_of_Eratosthenes
   (d) The lecture "Great Theoretical Ideas in Computer Science", lecture 14 of Carnegie Mellon's CS 15-251 in 2010 for an efficient algorithm of exponentiation.
4. **Regex:** No external libraries were used for the implementation of this module. However, we did heavily cite the following sources:
   (a) http://alleystoughton.us/150-efl/slides-3.3.pdf. This PDF contains the algorithm that we used to implement checking the simplicity of regular expressions.
   (b) The text *Automata and Computability*, which provided a helpful overview of regular expressions and the rules of Kleene Algebra.

6 Testing Plan

Our testing plan involved testing incrementally within each module. We each wrote our own suite of test cases for our modules, except for the GUI modules, which were tested interactively because of the difficulty of writing test

suites for such an environment. All testing was glass-box testing based on the implementation. More individual test plans are outlined in as follows:

1. **Turing Machine:** Testing occurred at milestones, such as after the implementation of a Turing Machine state.

2. **RSA:** When implementing RSA, we first wrote a specific case for one set of given parameters of RSA encryption. After having tested this, we moved on generalizing our approach so that users could provide their own prime numbers, or have them generated for them. This was then tested with a test suite.

3. **Regex:** Testing occurred at milestones again. Here, that meant testing after the implementation of the algorithm for checking the simplicity of a given regex. The rules for Kleene Algebra were then implemented and tested incrementally after each function was written.

Known Bugs: In our implementation of the regular expression simplification, we were not able to find a way to efficiently remove unnecessary parentheses from the final regular expressions. This results in expressions that are hard to read. We also were not able to implement a feature for resetting states inside the GUI, so after each run through of a learning module, the user must close the GUI and restart.

7 Division of Labor

1. **Alice:** Alice handled the implementation of the front end of the system. This involved using the library LablGTK to write the home screen and pages for the different learning modules that we wanted to include in the project. She also wrote the tape display for the Turing Machine, and the design for the input of the fields associated with a Turing Machine. Alice was responsible for integrating the front end and back-end in the module, completing the design. This was approximately 30 hours of work.

2. **Elise:** Elise handled the implementation of the Finite Control visualization for the Turing Machine. She also helped to implement the integration of the front-end of the Turing Machine with the back-end. Elise spent 30 hours on the implementation for this.

3. **Rohit:** Rohit was responsible for the implementation of the regular expression portion of the system. This involved writing a lexer and parser for translating OCaml strings from LablGTK to regex AST's, as defined in the type system above. Rohit was also responsible (along with Tavish), for the implementation of the RSA module. Here, Rohit primarily worked on building the specific case of the RSA algorithm, and designing the structure of the overall module. This was approximately 25 hours of work.

4. **Tavish:** Tavish was responsible for the implementation of the Turing Machine portion of the system. This involved writing a module to first define and organize the structure of a Turing Machine, and then to write functions that would step this Turing Machine to successive states. Tavish

also handled the generalization of the RSA module so that it would take more inputs and added more features. These features handled primality testing, generating prime numbers, and extended Euclidean algorithm This was approximately 43 hours of work.