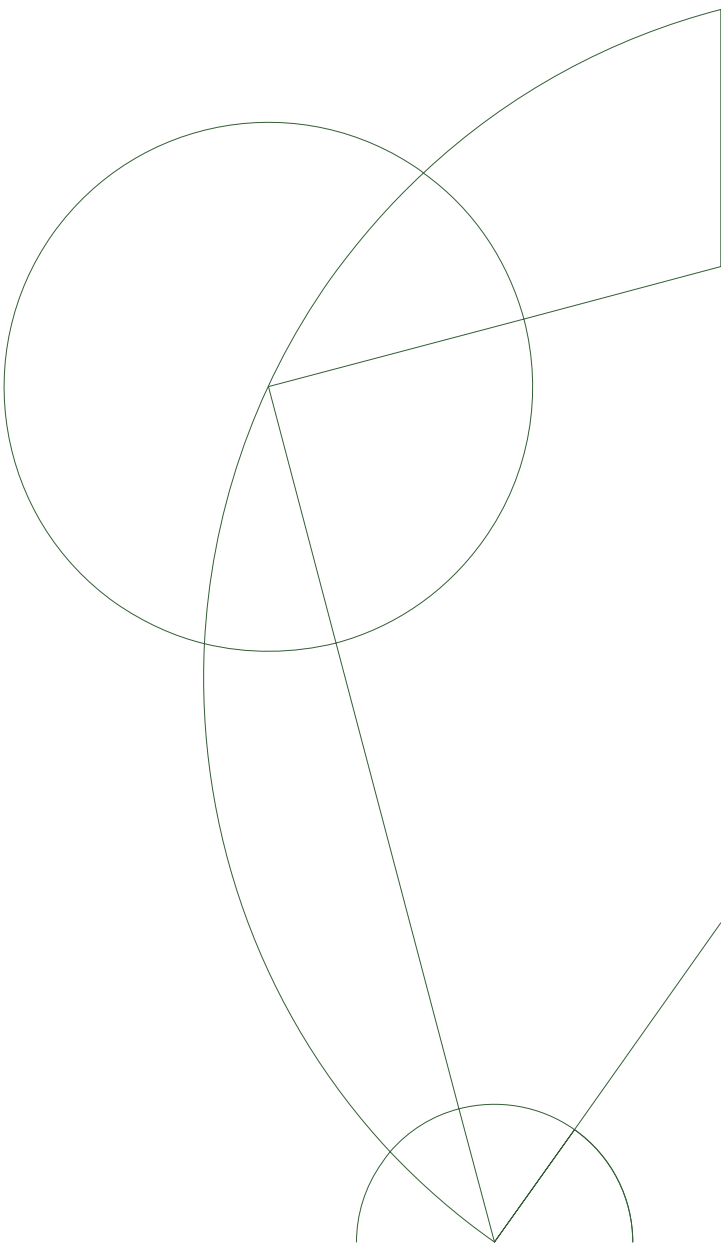
FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

Bachelor's Thesis

Lars Vadgaard & Anders Jørgensen

Reversible Programming Languages
— and Program Transformations



Robin Kaarsgaard

Juni 2018

Abstract. Each report must include an abstract that summarizes the results.Recommended length is at most 150 words. The abstract should notcontain any references or displayed equations.

Contents

Introduction

Many sequential programming languages today are forward deterministic; a given instruction in a program uniquely defines the next state. Most of these languages, however, may discard information along the execution path. Thus, one given instruction in a program may not uniquely define the previous state. These languages are therefore not backward deterministic; they are irreversible.

There are, however, reversible programming languages that do not allow this loss of information. A given instruction still uniquely defines the next state, but it also uniquely defines the previous. Two such languages, proposed in [YAG16], are RL and SRL.

Landauer's Principle, described in [Lan61], states that erased information must be dissipated as heat. That is, the lower limit of power consumption in a microprocessor depends, in some way, on erasure of information. Thus, reversible computation, which does not allow loss of information, can in theory circumvent this lower bound and improve upon the power efficiency of modern computers.

We will in this project, both as proof of concept and for educational purposes, implement an interpreter for each of the proposed languages RL and SRL along with some interesting program transformations, namely program inversion for each of the two languages and program translation *between* the two languages. Furthermore, we will implement — and demonstrate — a web based, graphical user interface for the two interpreters.

Background

In this chapter, we will briefly discuss the background of the implementation project. We will dive into the definitions, and some of the theory, presented in [YAG16].

Reversible Flowcharts

Flowcharts are a natural way to model imperative programs. Similarly, *reversible* imperative programs can be naturally modeled by reversible flowcharts. The two models are similar — however, to actually guarantee reversibility in reversible flowcharts, we may define some additional attributes. More specifically, we may want any join point between two edges in a flowchart to be acknowledged through an assertion which can determine the origin of the control flow. The definition as seen in [YAG16] says that

"A *reversible flowchart* F is a finite directed graph with three types of nodes, each of which represents an *atomic operation*"

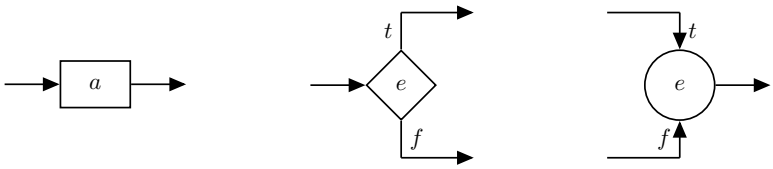
T. Yokotama et al. move on to describe the three atomic operations:

(a) the *step* operation performs some injective (i.e. reversible) operation and passes the control flow to the outgoing edge,

(b) the *test* passes the control flow to one of two outgoing edges depending on a predicate e, and

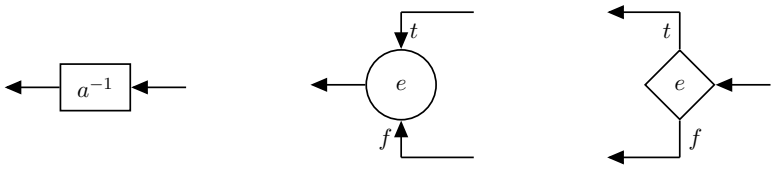
(c) the *assertion* joins two edges and passes the control flow depending on a predicate e.

The atomic operations can be seen in Figure 1. Having these as elementary building blocks of our reversible flowcharts, it is trivial to invert a program; simply invert each edge and, naturally, any step operation in the flowchart. This, in turn, means that the inversion of a test is an assertion and vice versa. The inversion of each atomic operation can be seen in Figure 2.



(a) Step (b) Test (c) Assertion

Figure 1: The three atomic operations of reversible flowcharts



(a) Step (b) Assertion (c) Test

Figure 2: Inversion of the three atomic operations of reversible flowcharts

This is the general idea of reversible flowcharts, and two incarnations of this model are RL and SRL.

RL and SRL

Reversible Language and Structured Reversible Language — RL and SRL for short, repectively — are two reversible flowchart languages proposed in [YAG16]. RL is an assembly-like language with unstructured jumps, while SRL, as the name indicates, is a similar language but with control structures and no jumps. The two languages share a common subset of features — namely their expressions and step operations. The grammar of the two languages along with the common structures are given in [YAG16] and can be seen in Figure 3.

As we see, RL, described in Figure 3b, consists of one or more (RL-)blocks, where each block is (uniquely) defined by a label and has a come-from assertion, zero or more step operations and an unstructured jump to any block in the program. Note that a well-formed RL program must contain exactly one entry and one exit.

SRL is more linear; as we see in Figure 3a, a program consists of exactly one (SRL-)block, where a block may either be an atomic step operation, a control structure — that is, an `if-` or a `do-until` structure — or recursively a sequence of two blocks. Sequences of blocks are executed in order, and thus, an SRL program looks a lot like a program written in common

programming languages such as Python or C, to name a few. Note that a block in RL is different from a block in SRL.

As mentioned, the two languages share expressions and step operations along with the basic value types. A brief overview:

Values and types

- A variable can either be mapped to an integer or to a list of integers.
- We only have integer literals; to obtain a certain list of integers, one has to bind each of the wanted values to a variable and push it to a list.
- Boolean values are represented by integers. A non-zero integer value will evaluate to true, and a zero integer value will evaluate to false.

Step operations

- To modify a variable, we have the variable update. The variable update only supports reversible operators, namely addition and subtraction. Indexing is supported in a single dimension. One rule is that the identifier being updated must not occur in the right-hand expression since such an update is irreversible.
- You can push the value of a variable onto a list with the `push` step operation. The variable being pushed must be zero-cleared afterwards.
- You can pop the top value from a list into a variable with the `pop` step operation. The variable must be zero-cleared beforehand.
- The `skip` step operation does nothing.

Expressions

- We can have any arbitrary binary operator between two expressions, recursively.
- An expression terminates with an integer literal or a variable lookup.
- The `top` expression is unary and returns the top of the operand.
- The `empty` predicate is unary and returns true if the operand is an empty list and false otherwise.

$$
\begin{array}{lcl@{\qquad}lcl}
p & ::= & b & b & ::= & a \mid \texttt{if } e \texttt{ then } b \texttt{ else } b \texttt{ fi } e \\
 & & & & \mid & b\ b \mid \texttt{from } e \texttt{ do } b \texttt{ loop } b \texttt{ until } e
\end{array}
$$

(a) Structured reversible language *SRL*.

$$
\begin{array}{lcl@{\qquad}lcl@{\qquad}lcl}
q & ::= & d^+ & k & ::= & \texttt{from } l & j & ::= & \texttt{goto } l \\
d & ::= & l : k\ a^*\ j & & \mid & \texttt{fi } e \texttt{ from } l \texttt{ else } l & & \mid & \texttt{if } e \texttt{ goto } l \texttt{ else } l
\end{array}
$$

(b) Unstructured reversible language *RL*.

$$
\begin{array}{lcl@{\qquad}lcl}
a & ::= & x \oplus = e & e & ::= & c \mid x \mid x[e] \mid e \otimes e \mid \texttt{top } x \mid \texttt{empty } x \\
 & \mid & x[e] \oplus = e & c & ::= & 0 \mid 1 \mid \cdots \mid 4294967295 \\
 & \mid & \texttt{push } x\ x & \otimes & ::= & \oplus \mid * \mid / \mid \cdots \\
 & \mid & \texttt{pop } x\ x & \oplus & ::= & + \mid - \mid \hat{} \\
 & \mid & \texttt{skip} & & &
\end{array}
$$

(c) Reversible step operations and expressions.

$$
\begin{array}{lllllll}
\texttt{SRL}: & p \in \texttt{SRL} & b \in \texttt{Blk} & & & & \\
\texttt{RL}: & q \in \texttt{RL} & d \in \texttt{RLBlk} & j \in \texttt{Jump} & k \in \texttt{From} & l \in \texttt{Label} \\
\texttt{SRL}, \texttt{RL}: & a \in \texttt{Step} & e \in \texttt{Exp} & c \in \texttt{Const} & x \in \texttt{Var} & \oplus, \otimes \in \texttt{Op}
\end{array}
$$

(d) Syntax domains of *SRL* and *RL*.

Figure 3: Syntax of the two reversible flowchart languages.

Implementation

We have, as was the goal of the project, successfully implemented the interpreters for the reversible programming languages RL and SRL. Furthermore, both implementations support program inversion and -translation. Finally, a web based user interface for the two languages has been designed.

The Interpreters

The interpreters are at the very core of this project.

Changes

Step operations added: swap init free

Update operators added: multiplication division

exp operators added: Binary: Power Equality Inequality Less than Less than or equal to Greater than Greater than or equal to Modulo Logical or Logical and

Unary: Negation Sign Logical negation Size Null

Features: Explicit type variable type declaration Indexing in an arbitrary number of dimensions

Syntax: Shorthands: Skip - . Empty - ? Not - !

It is important to note that we have made a few changes to the languages. We have modified some of the syntax described in Figure 3, but also added some new step operations and core features.

Common

RL

SRL

Program Inversion

Common

RL

SRL

Program Translation

SRL to RL

RL to SRL

The Web Interface

The purpose of the web interface is to grant a simpler way to users for interacting with the interpreters. This is split up in two parts: The server, which handles the routing of the web server and serves the client interface, along with api-calls for the actual interaction with the Command-Line Interpreters; The client part, which is the actual web interface that the user sees. The implementation of these are described in the following subsections.

Client Interface

Features

Before explaining the implementation of the client interface, the features of the interface will be described. If we look at the User Interface seen in Figure 4, we have 3 rectangular areas: The toolbar at the top; the code editor, located below the toolbar on the left hand side; the result area, located below the toolbar on the right hand side.

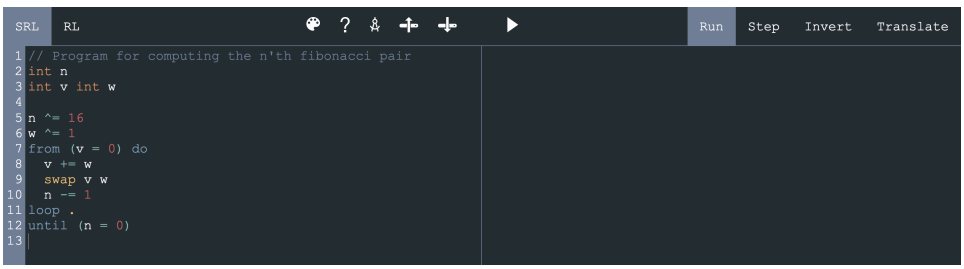


Figure 4: Screenshot of web client UI.

At the far left of the toolbar a radio button group with the options SRL and RL. The highlighted option indicates how the code in the code editor should be interpreted. If RL is chosen, the code is interpreted as if it were a RL program; likewise for SRL.
On the left hand side of the middle, the toolbar has some icons:

Themes
By hovering over the icon a dropdown menu, for choosing the color scheme/ theme of the client interface, appears as illustrated in Figure 5.
Figure 6 shows the color scheme when the white theme is chosen.

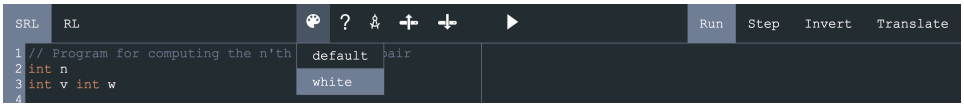


Figure 5: Web client themes dropdown menu.

Figure 6: Web client UI in white theme color scheme.

? Help
By clicking on this icon a modal window opens, illustrated in Figure 7, with help for the web interface, the modes and the languages.

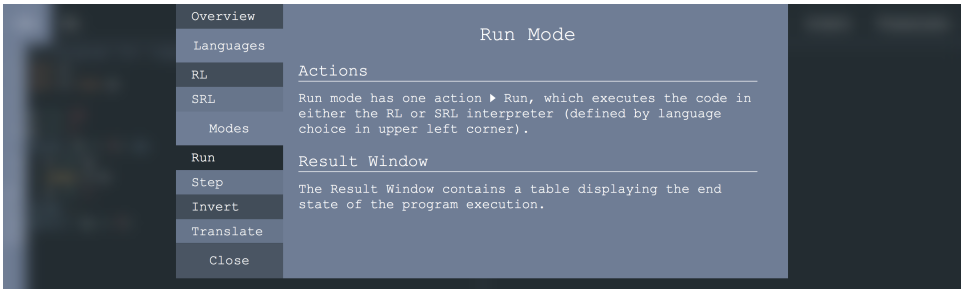


Figure 7: Web client help modal window displaying the Run mode help page.

Templates
By hovering over this icon a dropdown menu, where template programs for both RL and SRL can be loaded into the code editor, appears as illustrated in Figure 8.

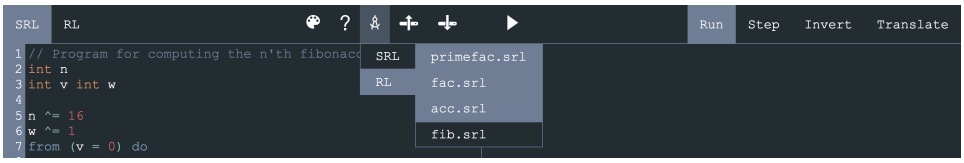


Figure 8: Web client templates dropdown menu.

Open
By click on this icon a modal window opens, as illustrated in Figure 9. Here it is possible to import and open previously stored programs.

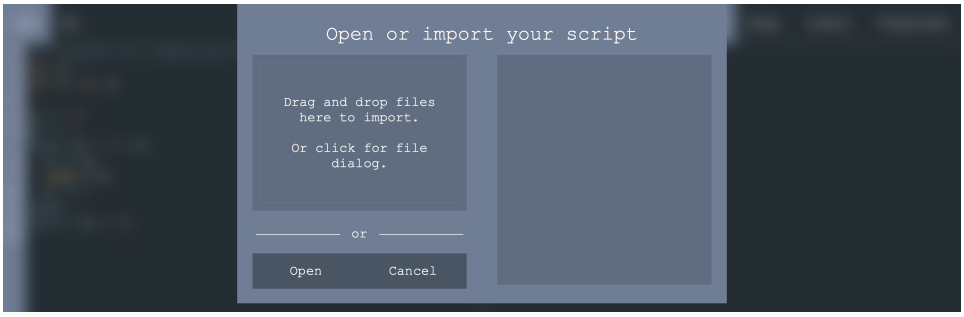


Figure 9: Web client open modal window.

Save
By clicking on this icon a modal window opens, as illustrated in Figure 10. Here it is possible to export, share and store the code currently loaded into the code editor.

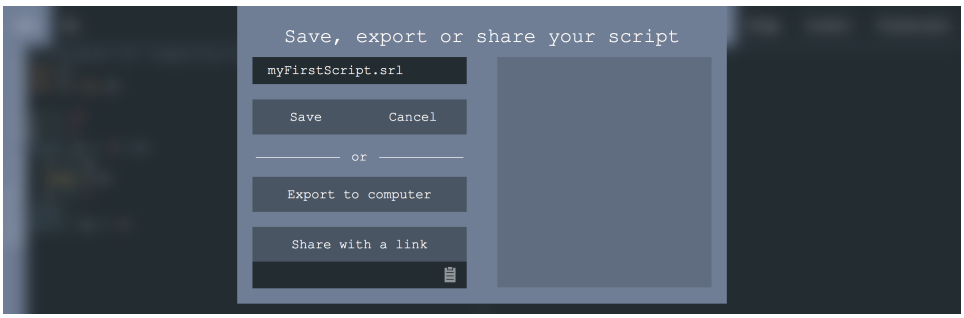


Figure 10: Web client save modal window.

The right hand side of the toolbar is dedicated to modes and their associated actions. There are 4 modes which are chosen from the radio button group at the far right:

Run

The Run mode utilises the run mode from the Command-Line Interface. The only action associated with the Run mode is ▶ **run**, which executes the code written in the code editor, and displays the final program-state in the result area.

Step

The Step mode utilises the run mode as the Run mode, but sets the log flag, to get the execution log instead of only the final state. This way, it is easy to step through the program step by step operation, and see how the individual statements alters the state. The execution of the last step operation can either result in successful or erroneous program termination. When choosing the Step mode, it is not yet activated. For activating the mode, the action ▶ **begin stepping** is used. This displays either an error or the start state of the program (all variables set to empty). When activated 5 actions are available: ◀ **previous step** undoes the last executed step operation; ▶ **next step** executes the next step operation; ◀| **reset** undoes all executed step operations; |▶ **end** executes all remaining step operations; ■ **stop** stops the execution and deactivates the Step mode.
While Step mode is activated, all other features are disabled.

Invert

The Invert mode utilises the invert mode from the Command-Line Interface. The only action associated with the Invert mode is ▶ **invert**, which inverts the code and displays it in the result area.

Translate

The Translate mode utilises the translate mode from the Command-Line Interface. The only action associated with the Translate mode is ▶ **translate**, which translates the code and displays it in the result area.

All the above modes have the ability to fail. Figure 11 illustrates a case where a parse error occurs.

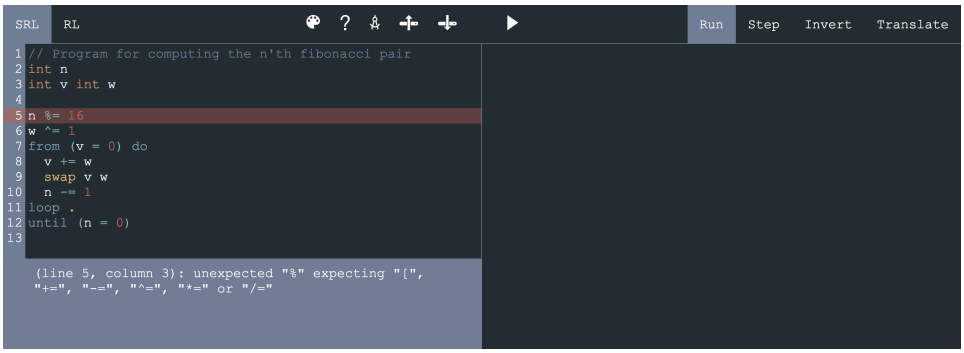


Figure 11: Web client with parse error.

Implementation

The web client interface is a nodeJS project written in ECMAScript 6, using the library ReactJS, for creating modularised components. We chose to use ReactJS, instead of vanilla javascript and html, since it allows component rendering based on state. This way focus was primarily on state and logic, instead of design and propagating state to the application. To separate state and logic from UI even further, we chose to handle state through Redux. This way we could move state and alteration to state outside of the ReactJS components. Hereby giving the individual components access to only the absolute necessary state variables, and state alterations methods. This way the possibilities where a specific state alteration can fail is limited to the components with access, and hereby making it easier to debug and fix.

The web client interface uses babel and webpack to compile and bundle scripts and resources. Webpack uses babel to convert JSX to React components and transpile ECMAScript 6 to ECMAScript 5, to allow for import statements. Webpack itself is used to precompile SASS to CSS, append prefixes to the CSS for better cross-browser support and bundling scripts, styles and assets for minimal HTTP requests.

The root of the web client is `/web/client`. Configuration files for node package dependencies and the webpack build flow are found here. Under the subdirectory `src/` all source files and resources are located. SASS files are found under `src/styles` and the ReactJS components are found under `src/components`.

TODO: CodeMirror

TODO: Project important packages

TODO: Project structure

Server & API

The web server is an independent node project, when built uses the Client Web Interface and the Command-Line Interface as dependencies. These are build and copied into the web server project folder. The executables for the Command-Line Interfaces are placed under `/web/server/bin` and the Web Client Interface is placed under `/web/server/client`.

The server is written in ECMAScript 6, but since nodeJS (version 9 and below) does not support `import` statements natively, the Babel transpiler is used for running the ECMAScript 6 code as ECMAScript 5. This is done in the entry-file `babel-server.js`, as a wrapper for the actual server configuration in `server.js`. We run the transpiled code with node, and uses the Express package for opening ports, setting up the server and handling web-routes. All communication between the API server and the client-side, is formatted as JSON; as is the results from the interpreters. This way it is easy to parse data from the interpreter to the client interface via the API.

The server has Cross Origin Resource Sharing (CORS) enabled, to allow for separation of the api server and the client web interface. This allows for the server to be hosted at one address, while the interface is hosted at another. CORS is enabled by setting the appropriate headers shown in Listing 1.

```
10 // Enable CORS
11 app.use((req, res, next) => {
12   res.header('Access-Control-Allow-Origin',  '*');
13   res.header('Access-Control-Allow-Headers', 'Origin, X-
       Requested-With, Content-Type, Accept');
14   next();
15 });
```

Listing 1: CORS headers from /web/server/server.js.

The routing has two responsibilities: To handle API calls, which can be seen in Figure 13; serve static files, when the client interface is requested. The defined routes are listed in Figure 12. Routes are divided into 3 categories: API routes, root requests and static files; where static files are the bundled css and javascript files, that the client interface requires.

Route	Responsibility
`/`	Fetch the built client interface index page from (`/web/server/client/index.html`).
`/api*`	Here `*` is everything after `/api`, which is forwarded to the API router. The API routes are described in Figure 13.
`/:filename.:ext`	Here `:filename.:ext` matches a static resource, e.g. `bundle.js`. All static files are fetched from `/web/server/client`. Thus only files that the client interface uses can be requested.

Figure 12: Routes for web server.

If the client interface is hosted at a different address than the api, the `/web/server/client` folder can be removed, causing the route for the index page and the static pages to respond with a 404 (page not found) status and page.

TODO: API - features

Route	Method	Responsibility
`/run/:language`	`POST`	Here `:language` can either be `rl` or `srl`. This API-call executes the code that is posted along with the request, and returns the end-state of the program. The code submitted should be wrapped in a JSON object, with the attribute `code`.
`/run/log/:language`	`POST`	Is equivalent to `/run/:language`, but instead of returning the end-state alone, it also returns the json-formatted execution-log.
`/invert/:language`	`POST`	Here `:language` can either be `rl` or `srl`. This API-call inverts the code. The code submitted should be wrapped in a JSON object, with the attribute `code`.
`/translate/:language`	`POST`	Here `:language` can either be `rl` or `srl`. This API-call translates the code from the specified language to its counterpart. The code submitted should be wrapped in a JSON object, with the attribute `code`.
`/template/list`	`GET`	Returns a list of SRL and RL files, as a JSON-object. The object has the attributes `srl` and `rl`, where each corresponding value is a list of template-names for that language.
`/template/:file`	`GET`	Here `:file` is the template requested. A JSON-object, with the attribute `code`, which contains the code of the requested template file, is returned.

Figure 13: API routes.

TODO: Security For accessing the Command-Line interface, listing and fetching template-files, the interface execFile(file, [arguments], callback) from the node module

```
execFile(cmd,
        [mode, code, flags],
        {maxBuffer, timeout},
        (err, stdout, stderr) => {
  // ... callback content
});
```

Further Improvements

Interpretation

TODO: faster interpretation, optimising the code

Parsing

TODO: not ignoring newlines, generates better error messages and allows more syntactic sugar

Optimisation

TODO: optimisation of Common: merge updates, compute constants, remove redundant stuff, remove skips TODO: optimisation of RL: AST as a cyclic graph, reduce the graph

Documentation

Installation

Both the interpreters for RL and SRL and the web interface can be found at the same github repository: KvelaGorrrrnio/BscProject.
To get the code, clone the repository with: `$ git clone https://github.com/KvelaGorrrrnio/BscProject.git`

The different parts of the project use different build systems. To simplify the build-flow Makefiles are used as a wrapping build-system, to supply a uniform build interface for the project.

Command-Line Interface

The Interpreters and the Command-Line Interface are located under the `/src` directory, but can be build from the project root directory. The underlying dependency manager and build system for the Command-Line Interface is Stack, which can be installed by one of following:

```
$ brew install haskell-stack cabal-install ghc # MacOS
$ curl -sSL https://get.haskellstack.org/ | sh # Unix
$ wget -qO- https://get.haskellstack.org/ | sh # Unix alternative
```

When standing at the project root, there are two ways of installing the Command-Line Interface.
First option is to `$ make src`, which builds the `rl` and `srl` executables into `/src/bin`. Copy the executables to your local bin, for system wide usage.
Second option is to `$ make install`, which installs the `rl` and `srl` executables directly to the stack local bin, for system wide usage. This requires that the stack local bin is in your `$PATH`.

Web Interface

The web client interface is found under `/web/client` and the web server is found under `/web/server`. Both the server and the client application is build and runned with NodeJS. NodeJS can be installed by one of the following:

```
$ brew install node               # MacOS
$ sudo apt-get install nodejs npm # Ubuntu
$ sudo pacman -S nodejs npm       # Arch
```

The default build command, soon to be described, combines the interpreters and the client web interface with the web server, such that the server has access to the interpreters and the client interface is accessible through the running server.
By running `$ make web`, when standing at the project root, the web servers dependencies are installed, the Command-Line Interface and the web client interface is built and copied to the web server. If builded successfully the last line should be `"Web server has been built."`.

Usage

Command-Line Interface

The Command-Line Interface for `rl` and `srl` are almost identitical, the only real difference is the naming (rl instead of srl, and vice versa). Thus only the interface for `rl` will be described. Both Command-Line Interfaces has `-help` flags, which displays the different modes, options and flags, of which can be chosen and set.

TODO: Describe modes

Web Interface

For starting the web server, use `$ make server`. This should yield an output ending with

```
1  Server has started.
2  Web interface is running at http://localhost:3001.
```

After starting the server and the above-shown output has been printed, every api-call and index page requests to the server is logged to `stdout`. The log contains for each request timestamp and the requested path - along with information describing the outcome (erroneous or succesful). For shutting down the server, use `Ctrl-C`.
TODO: Hosting on two different addresses. (webpack.prod.config.js)

Testing

When located at the root of the project, all tests, for both interpreters/cli and the web interface can be executed by `$ make test`. For only getting output relevant for testing, build all targets with `$ make` before running the tests.

Command-Line Interface

TODO: CLI testing

Web Interface

TODO: Web client testing
TODO: Web server testing

Appendices

Appendix A. Source Code

1. **Interpreter**
2. **Inversion**
3. **Translation**

References

[1] R. Landauer. “Irreversibility and heat generation in the computing process”. In: *IBM Journal of Research and Development* 5.3 (1961), pp. 183–191.

[2] T. Yokoyama, H. B. Axelsen, and R. Glück. “Fundamentals of Reversible Flowchart Languages”. In: *Theoretical Computer Science* 611 (2016), pp. 87–115.