Course Project 4
The Mire Programming Language

CSE 30151 Spring 2017

Version of April 25, 2017

In this last project, we'll extend regular transduction expressions (RTEs) into a full-fledged programming language, Mire (match iterated regular expression). A Mire program is a sequence of RTEs, separated by semicolons (`;`). The expressions are executed in sequence, with the output of each feeding into the input of the next (just like the composition operation in CP3).

Regular transductions are closed under composition, so adding composition to RTEs doesn't increase their power. But one more small change does: If an expression is surrounded by curly braces ({...}), then that expression is executed zero or more times, like a while loop. As you showed in HW7, this makes our expression language equivalent to Turing machines! Indeed, *nondeterministic* Turing machines are fairly easy to simulate in Mire.

Disclaimer: Unlike `more`, which was actually a genuinely useful tool, Mire is an example of a *Turing tarpit*, a language "in which everything is possible but nothing of interest is easy."[1] Well, the nondeterminism in the language makes some interesting things surprisingly easy, but I doubt that you'd ever want to write a real program in it.

Getting started

We've made some updates, so please have one team member run the commands

```
git pull https://github.com/ND-CSE-30151-SP17/theory-project-skeleton
git push
```

and then other team members should run `git pull`.

You will need a correct solution for CP3 for this project. You may use the official solution or another team's solution, as long as you properly cite your source.

[1] Alan Perlis, 1982. Epigrams on programming. *ACM SIGPLAN Notices* 17(9):7–13.

Examples

The transformation `(1(1:))*` only accepts an even number of `1`'s and deletes every other `1`. If we wrap a loop around this, `{(1(1:))*)}` halves the number of `1`'s repeatedly. If we require that a single `1` remains afterwards, then the number of `1`'s must have originally been a power of two. Thus, the following program accepts the language $\{1^{2^n} \mid n \geq 0\}$:

```
// This is examples/log2a.mire
{(1(1:))*};                    // delete every other 1
1                              // must be exactly one 1 left
```

Try the following (user input in blue):

```
$ python/mire -f examples/log2a.mire
1111                input string
1                   output string
```

The `-f` flag means to read the program from a file instead of the command line. To understand better what's going on, turn on the `-v` flag for verbose output:

```
$ python/mire -v -f examples/log2a.mire
1111                input string
[1]         11      delete every other 1
[1;1]       1       delete every other 1
[1;1;2]     1       must be exactly one 1 left
1                   output string
```

The verbose output shows each step of the run. For example, the line

```
[1;1;2]    1
```

means that steps 1, 1, and 2 were applied to the input string (`1111`) to yield the string `1`. So, what's happening above is that the first step applies twice, transforming the string to `11`, then `1`. At that point, the string matches the second step, so the program exits the loop and accepts (with output `1`).

But if the input string is `111111`, then the first line will transform it to `111`, and at that point, the first line can't repeat anymore (because it only matches even-length strings) and the second line can't match either, so the program rejects.

Let's extend the above program to not just output `1`, but to output 1^n, where n is the log (base 2) of the input.

```
// This is examples/log2u.mire
1*(:#);                        // append counter
{(1(1:))*#1*(:1)};             // delete every other 1, increment counter
(1#:)1*                        // remove input and separator
```

Here's an example run, on the same input string:

```
$ python/mire -v -f examples/log2u.mire
1111              input string
[1]       1111#   append counter
[1;2]     11#1    delete every other 1, increment counter
[1;2;2]   1#11    delete every other 1, increment counter
[1;2;2;3] 11      remove input and separator
11                output string
```

As a final example, recall from CP3 that the RTE `(0|1)*(0:1)(1:0)*` increments a binary number. We can use this trick to write the input and output in binary instead of unary.

```
// This is examples/log2b.mire
10*(:#0*);                      // append counter
{10*(0:)#(0|1)*(0:1)(1:0)*};  // delete 0, increment counter
(1#:)(0|1)*                     // remove input and separator
```

Example run:

```
$ python/mire -v -f examples/log2b.mire
100               input string
[1]       100#    append counter
[1;2]     10#1    delete 0, increment counter
[1;2;2]   1#10    delete 0, increment counter
[1;2;2;3] 10      remove input and separator
10                output string
```

This is similar to the previous example, but notice how the counter is initialized in the first step. This is because we don't know yet how many bits we need for the counter, so we simply append `0*`, for zero or more `0`s. This means that there are multiple possible intermediate strings, and the verbose output just shows one possibility. So when the second line increments the counter, it magically widens as needed!

There's one more example in the repository, `examples/factor.mire`, which illustrates some more techniques.

1 Exercises

The test script `tests/test-cp41.sh` provides tests for the following Mire programming exercises.

a. Write a Mire program that accepts any string $w \in \{\mathtt{a}, \mathtt{b}\}^*$ and outputs w^R. Please name your program `cp4/reverse.mire`.

b. Write a Mire program that accepts any string ww where $w \in \{\texttt{a}, \texttt{b}\}$ and outputs w. Please name your program `cp4/uncopy.mire`.

c. Write a Mire program that accepts any string $w \in \{\texttt{a}, \texttt{b}\}^*$ that can be permuted into a string w' matching `a*b*`, and outputs w'. Call your program `cp4/sort.mire`.

2 Parser

2.1 Preprocessing

To improve readability, a Mire program can have extra whitespace and comments. Write a preprocessing function:

- Argument: string α
- Return: copy of α preprocessed as follows
  1. Delete every occurrence of `//` and everything to its right, to the end of the line.
  2. Remove leading and trailing whitespace from each line.
  3. Join lines together, without any separator.

For example,

```
(na( na)*      // comment
 ( hey jude))*
```

should become `"(na( na)*( hey jude))*"`.

2.2 Composition and loops

Write code to extend the syntax to handle composition (`;`) and loops ($\{\ldots\}$). For simplicity, assume that a loop can contain only a single expression; a loop can't contain a composition or another loop. (You're welcome to try a more general case if you want to.)

You can do this by extending the grammar and its recursive-descent parser, but because the new syntax is so simple, it's also fine just to split the string on semicolons and check whether each expression begins/ends with curly braces.

2.3 Testing

Write a program `parse_program` that can be run in two ways:

`parse_program` *program*
`parse_program -f` *filename*

In the first form, the program is given on the command line; in the second form, the program is contained in the specified file. For each step of the program, `parse_program` should output a string representing the parse of that step, as in CP2 and CP3. If a program step is a loop, then that step's parse should be surrounded by `loop(...)`. For example, `./parse_program "{(1(1:))*};1"` should output

```
loop(star(concat(symbol(1),transduce(symbol(1),epsilon()))))
symbol(1)
```

2.4 Interface with interpreter

To ready your parser for the next part of the assignment, have it output a data structure representing a Mire program. A suggested interface for the parser function is:

- Argument: string α
- Return: a list $[(M_1, \ell_1), (M_2, \ell_2), \ldots, (M_m, \ell_m)]$, where each M_i is a NFT and each ℓ_i is a boolean indicating whether M_i is in a loop.

3 Interpreter

The interpreter for the program should take a program P and input string w, and run the steps of P. Just as a NFT can have multiple output strings, a Mire program can also have multiple output strings. As in CP3, we're just going to take one arbitrary output string.

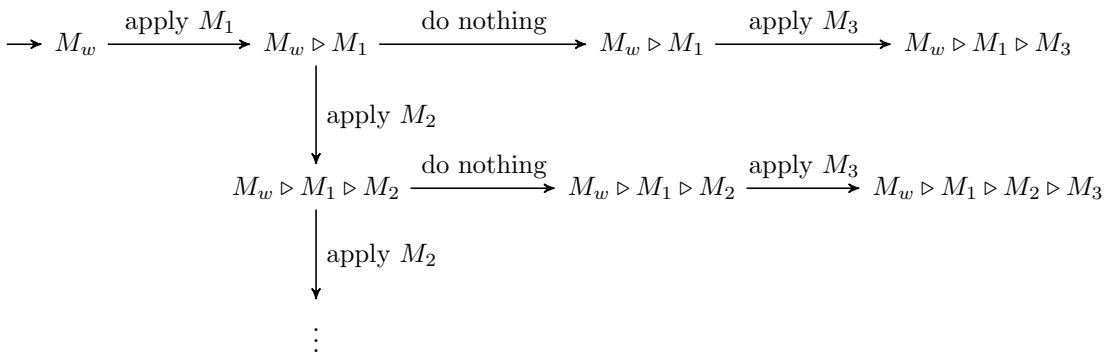
If the program has no loops, interpreting it would be straightforward:

function run-noloop(P,w) ▷ Don't implement this
  $M = \mathsf{singleton}(w)$
  for $i = 1$ **to** m **do**
    $M = M \triangleright M_i$
  return an arbitrary output string of M

You could visualize a program without loops as a graph; for example, a three-step program without loops (where $M_w = \mathsf{singleton}(w)$):

$$\longrightarrow M_w \xrightarrow{\text{apply } M_1} M_w \triangleright M_1 \xrightarrow{\text{apply } M_2} M_w \triangleright M_1 \triangleright M_2 \xrightarrow{\text{apply } M_3} M_w \triangleright M_1 \triangleright M_2 \triangleright M_3$$

But with loops, interpretation becomes more complex. We can visualize the program as a graph whose edges are the steps M_i. For example, a three-step program $M_1;\{M_2\};M_3$ can be visualized as:



At each of the nodes in the second column, the program has a choice: either repeat the loop again (down arrow) or quit the loop (right arrow). In order to navigate all these choices, we do a breadth-first search on this graph. The pseudocode is below:

function run(P, w)
  let A be an empty queue
  $M_w =$ singleton(w)
  push $(M_w, 0)$ to A
  while $|A| > 0$ **do**
    pop (M, i) from A
    if M has an accepting path **then**
      if $i = m$ **then**
        return M ▷ end of program
      else if ℓ_{i+1} **then** ▷ updated 2017/04/25
        push $(M, i+1)$ to A ▷ do nothing and go to next step
        push $(M \triangleright M_{i+1}, i)$ to A ▷ run step $(i+1)$ and repeat loop
      else
        push $(M \triangleright M_{i+1}, i+1)$ to A ▷ run step $(i+1)$ and go to next step
  return emptyset()

The return value of this function is a NFT M that only accepts w as input, and outputs zero or more strings. If there are no output strings, that means the program rejects w. Otherwise, the program accepts w, and we can print out an arbitrary output string of M.

4 Putting it together

Put all of the above into a command-line interpreter called `mire` that can be invoked in one of two ways:

`mire` *program*
`mire -f` *filename*

In the first form, *program* is a program. In the second form, *filename* contains a program. In either case, the program should be preprocessed for comments and whitespace and parsed into a form P that can be passed to `run`.

Then, for each input string w read from stdin:

- $M = \mathsf{run}(P, w)$
- If M has an accepting path, print one arbitrary output string.

Test your interpreter by running `test-cp4.sh`.

Submission instructions

Your code should build and run on `student`*nn*`.cse.nd.edu`. The automatic tester will clone your repository, `cd` into its root directory, run `make -C cp4`, and run `tests/test-cp4.sh`. You're advised to try all of the above steps and ensure that all tests pass.

To submit your work: If you are working in a branch, please merge to `master`. Push your repository to Github and then create a new release with tag version `cp4` (note that the tag version is not the same thing as the release title). If you are making a partial submission, then use a tag version of the form `cp4-1234`, indicating which parts you're submitting.

Rubric

Exercises	9
Preprocessor	3
Parser	6
Interpreter	9
Main program	3
Total	30