Course Project 4
The Mire Programming Language

CSE 30151 Spring 2017

Version of January 19, 2017

In this final project, we'll extend finite transducers to be equivalent to Turing machines using one seemingly simple change: allow a subexpression to be composed with itself an unbounded number of times. We call this programming language Mire (match iterated regular expression). Mire extends regular relation expressions with a new loop construction, denoted by curly braces. This is like a nondeterministic while loop, where before every iteration we can either stop or do another iteration. Slightly more formally, the semantics of $\{\alpha\}$ should be equivalent to

$$\Sigma\texttt{*}\texttt{|}\alpha\texttt{|}\alpha\texttt{;}\alpha\texttt{|}\alpha\texttt{;}\alpha\texttt{;}\alpha\texttt{|}\cdots$$

That is, the transformation α can apply zero times (Σ`*`), once (α), twice (α`;`α), and so on, up to an arbitrary number of times.

For example, 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 $\{\texttt{1}^{2^n} \mid n \geq 0\}$:

```
{(1(1:))*}; // repeatedly delete every other 1
1           // must be exactly one 1 left
```

See `examples/*.mire` for more examples.

To keep the implementation simpler, let's only allow loops to occur at the top level of the program (that is, not inside any parentheses). You're welcome to drop this restriction if you want to. But even under this restriction, it's possible to write a (nondeterministic or deterministic) Turing machine simulator in Mire.

Disclaimer: Unlike the previous project, which was actually a genuinely useful tool, this project has no conceivable use other than to make you think really hard about nondeterminism.

Getting started

The project repository includes the following files:

```
bin/
  parse_program
  mire
examples/
  binary.mire
  factor.mire
tests/
  test-cp4.sh
doc/
  cp4.pdf
cp4/
```

Please place the programs that you write into the `cp4/` subdirectory.

1 Warmup exercises

a. [3 points] Write a Mire program equivalent to the CFG G_1 in the book (page 102).

b. [6 points] Write a Mire program equivalent to the TM M_1 in the book (Figure 3.10).

2 Data structure

In CP3, the semantics of a regular expression was a NFT. But because NFTs are not closed under the loop operation, we can no longer represent a program containing a loop as a NFT. Remember that we are restricting the loop operator to only occur at the top level of the program. So, let's say that the semantic representation of a program is a sequence of steps, each of which is either a NFT or a loop containing a NFT. Write a data structure to represent this.

3 Preprocessor

To improve readability, when a Mire program is written in a file, it can have extra white-space 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))*"`.

4 Extend the parser

I think the simplest way to extend the grammar is to add the following rules (and set $S = \mathsf{Program}$):

$$\mathsf{Program} \to \mathsf{Program}\ \boxed{;}\ \mathsf{Step}$$
$$\mathsf{Program} \to \mathsf{Step}$$
$$\mathsf{Step} \to \boxed{\{}\ \mathsf{Cross}\ \boxed{\}}$$
$$\mathsf{Step} \to \mathsf{Cross}$$

The following rule's lookahead needs to be modified:

$$\mathsf{Concat} \to \mathsf{Concat}\ \mathsf{Unary} \qquad \text{if not followed by } \boxed{\}}\boxed{;}\boxed{:}\boxed{|}\boxed{)} \text{ or end of string}$$

Write a new top-level parsing function that is just like parseCompose except it doesn't perform composition; it just builds a sequence of steps:

function parseProgram()
  $M \leftarrow$ parseStep()
  append M to P
  while next token is $\boxed{;}$ **do**
    read $\boxed{;}$
    $M \leftarrow$ parseStep()
    append M to P
  return P

function parseStep()
  if next token is $\boxed{\{}$ **then**
    read $\boxed{\{}$
    $M \leftarrow$ parseCross()

```
        read }
        return loop M
    else
        M ← parseCross()
        return M
```

To test your parser extension, write a program `parse_program` that takes an expression defined by the above grammar and outputs a *sequence* of ASTs, one step per line. For example, the expression `"{(1(1:))*};1"` should output

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

5 Loop operation

The execution of the program is yet another breadth-first search.

```
function run(P, w)
    let A be an empty queue
    let M be such that R(M) = {(w, w)}
    push (M, 0) to A
    while |A| > 0 do
        pop (M, i) from A
        if M is not empty then
            if i = |P| then
                return M
            if M is not a loop then
                push (M ▷ P_{i+1}, i + 1) to A          ▷ run step P_{i+1}
            else
                push (M, i + 1) to A                    ▷ exit loop
                push (M ▷ P_{i+1}, i) to A              ▷ run step P_{i+1} and repeat
    return emptyset()
```

This computes a NFT M that only accepts w as input, and outputs zero or more strings. If $R(M) = \emptyset$, that means the program definitely rejects w. If $R(M) \neq \emptyset$, that means the program definitely accepts w. However, the output language of M is only a subset of the true output language. But that's good enough, because we only want to print one arbitrary output string.

6 Putting it together

Write a function that puts all the above functions together:

- Arguments: regular expression α, string w
- Return: one arbitrary output string if α matches w; raises an exception or returns a sentinel value otherwise

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

`mire` *program*
`mire -f` *progfile*

In the first form, *program* is a program. In the second form, *progfile* contains a program and should be preprocessed for comments and whitespace as described above. The interpreter should read lines from stdin. For each input line that matches, it should print out 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, run `make` in subdirectory `cp4`, and then 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, please push your repository to Github and then create a new release with tag `cp4`. If you need to re-submit, create another release with a new tag starting with `cp4`, like `cp4.1`.

Rubric

Exercises	9
Data structure	3
Preprocessor	3
Parser	6
Loop operation	6
Main program	3
Total	30