Regular Expressions, part 2

CS 440: Programming Languages and Translators, Spring 2020

2/5: pp.4,5

A. An Implementation of Regular Expressions that Recognizes matching input

- Let's look at a concrete implementation of regular expressions.
- We'll work on a `match` program that takes a regular expression and string and tries to match the expression with (some initial segment of) the string.
  - On success, it returns the leftover part of the input string. It doesn't return the actual input, so this is a **recognizer** (tells you whether an input is good or not).
  - It's a backtracking search algorithm: Handling alternation has us trying different alternatives if we encounter a failure to match.
- We'll start with the following datatype for regular expressions and look at the part of the `match` function that handles constants, AND, and OR.

```
-- Regular expressions
data RegExpr a
    = RE_const a
    | RE_or [RegExpr a]
    | RE_and [RegExpr a]
    | RE_star (RegExpr a)
    | RE_any                    -- dot   - any one symbol
    | RE_in_set [a]             -- [...] - any symbol in list
    | RE_end                    -- $  - at end of input
    | RE_empty                  -- epsilon - empty (no symbols)
        deriving (Eq, Read, Show)

match :: Eq a => RegExpr a -> [a] -> Maybe [a]
....
```

Overview of RegExpr Matching

- Since `RegExpr` is parameterized, we can handle input that's a sequence of characters (i.e., strings) or a sequence of more-complicated symbols. I'll just use characters and strings below.
- The `match` function takes a regular expression and input and checks to see if a prefix of the input matches the expression. If it does, the match succeeds and we return `Just` *suffix* (the leftover input after removing the prefix). E.g., if the expression is *ab* and the input is *abc*, then the leftover suffix is *c*. If the match fails, we return `Nothing`. So on strings, `match :: RegExpr Char -> [Char] -> Maybe [Char]`.
- For a regular expression on characters, the specific type for match is `RegExpr Char -> [Char] -> Maybe [Char]`, so `match re` *string* = `Nothing` (if the match fails) or `Just` *string* (if it succeeds). On success,

the match routines removes the matching substring from the head of the input and returns a string with the leftover suffix.

Constants, Sequences (AND), and Alternations (OR)

- The basic regular expression is a constant (`RE_const` *symbol*), which checks the head of the input against *symbol* and removes it from the input if it's there. E.g., matching `(RE_const 'a')` with input `"abc"` would succeed and return `"bc"`. Matching the same expression witn input `"xyz"` returns `Nothing`.
- More specifically, if the input is `(head_inp:input')` and `head_inp` == *symbol*, then the match succeeds and we return `Just input'` (the leftover input). If `head_inp` /= *symbol* or the input is empty, the match fails and we return `Nothing`.
- We can build up more complicated regular expressions with alternation (`RE_or`) and conjunction (`RE_and`). We'll go into more detail but for an example, the reg expr `ab|c` is represented as

```
re = RE_or [ RE_and [map RE_const "ab"], RE_const 'c' ]
```

- Note that `RE_or` and `RE_and` both take lists of regular expressions. This lets us represent expressions like *e1 e2 e3 e4* as `RE_and` [*e1*, *e2*, *e3*, *e4*] instead of having to nest something like (`RE_and` *e1* (`RE_and` *e2* (`RE_and` *e3 e4*))).
- For example: (Recall `re` represents `ab|c`.)

```
match re "abcd" == Just "cd"  -- the ab in ab|c matched, leaving cd
match re "cxyz" == Just "xyz" -- the c in ab|c matched, leaving xyz
match re "qrst" == Nothing    -- both ab and c failed to match
match re "acde" == Nothing    -- the a matched but b didn't match cde
                              -- and c didn't match acde
```

- **More on RE_or**: The `RE_or` constructor takes a list of regular expressions and tries them one after another against the input list. The first match ends the search and `match` returns the result of that match. If none of the expressions match, `match` returns `Nothing`. The `Just` leftover list comes from whichever match succeeded. Some examples:

```
or1 = RE_or (map RE_const "abc")
match or1 "axy" == Just "xy"
match or1 "bcd" == Just "cd"
match or1 "ccd" == Just "cd"
match or1 "dba" == Nothing

or2 = RE_or (map RE_const ["hello","goodbye"])
match or2 ["hello", "and", "goodbye"] == Just ["and", "goodbye"]
match or2 ["goodbye", "and", "hello"] == Just ["and", "hello"]
match or2 ["aloha"] == Nothing
```

- To implement `RE_or`, first we look at the list of regular expressions: If it's empty, the match fails. Otherwise we try to match the head of the expression list against the input. If that succeeds, we're done. If it fails (returns `Nothing`), we recursively search the same input using `RE_or` on the tail of the expression list.

- **More on RE_and**: With `RE_and`, we try to match a sequence of regular expressions against the input. First we match the head expression against the input; if that fails, then we fail. If it succeeds (returns `Just` *leftover*), we continue the match using the tail of the expression list on the leftover input. So we stop as soon as the first expression fails or when we've used up all the expressions and succeeded.
- Here are some examples:

```
abc = RE_and $ map RE_const "abc"    -- look for "a" then "b" then "c"
match abc "abcd" == Just "d"  -- "d" left after dropping "a", "b", "c"
match abc "ab"   == Nothing   -- "a" and "b" ok but matching "c" fails

-- First "abc" leaves Just "abcz"; second "abc" leaves Just "z"
match (RE_and [abc,abc]) "abcabcz" == Just "z"

-- First or2 matches "hello", second matches "goodbye"
match (RE_and [or3,or3]) ["hello","goodbye","okay?"] == Just ["okay?"]

-- First or3 matches "hello" but second or3 doesn't match "nope", so we fail
match (RE_and [or3,or3]) ["hello","nope"] == Nothing
```

Skeleton code for match

- The code for `match` is short but (like basically all Haskell programs) needs to be studied carefully.
- The skeleton attached contains code for `RE_const`, `RE_or`, and `RE_and`. (You'll be implementing other cases.)

Activity Questions, Lecture 7

Study the code in the attached skeleton, `Lec_07_skeleton.hs`.

1. Load / Enter the code in the skeleton into ghci and run the tests from `Lec_07_tests.hs`. They should all come back as true.

For Problems 2 -4, you are to implement three of the unimplemented expressions: `.` (dot; any one symbol), `$`, and ε (empty).

```
data RegExpr a
     = ....
     | RE_any                    -- dot    - any one symbol
     | RE_end                    -- $   - at end of input
     | RE_empty                  -- epsilon - empty (no symbols)
```

2. `RE_empty` is the easiest to implement: matching `RE_empty` on some input always succeeds and returns the input unchanged. (The empty expression consumes no input, so the "leftover" part is the same as the input before the match.)
3. Matching `RE_end` succeeds if the input is the empty list; since there's no leftover input, matching returns `Just` the empty list. If the input is not empty, matching fails (returns `Nothing`).
4. Matching `RE_any` succeeds if the input is nonempty; it consumes the head symbol and the leftover input is `Just` the tail of the input. If the input is empty, matching `RE_any` fails.

For Problems 5 - 9 [2/4], translate some of the regular expressions from Lecture activity 6 into `RegExpr` format. We haven't covered implementation of Kleene *, which means we can't just use `RE_star` (*reg expr*). A temporary hack for something like [0-9]* is *digits* = [0-9] *digits* | ε, which translates to [2/5]

```
digits = RE_or [RE_and [RE_or (map RE_const "0123456789"), digits], RE_empty]
```

Feel free to give names to other patterns.

5. (`0|[1-9]\d*`): Integer constants ≥ 0 that don't begin with a leading zero except for `0` itself
6. `-?(0|[1-9]\d*)` Same as the previous problem but with an optional minus sign. [2/5]
7. (`0[0-7]*`): Octal constants ≥ 0 that begin with a leading zero 0, including `0`
8. (`\d+\.\d+`): Floating point constants that include a dot and at least one digit before and after the dot. You can include constants that consist of only zeros (plus that dot)
9. (`\d+\.\d*|\.\d+`): Floating point constants that include a dot and at least one digit before **or** after the dot (or both).

Solutions to Activity Questions

1. (Do an experiment)

2. (Match ε)

```
match RE_empty input = Just input -- matching empty string always succeeds
```

3. (Match end of input)

```
match RE_end [] = Just [] -- Side question: how is (match RE-end "") different?
match RE_end _  = Nothing
```

4. (Match any one symbol)

```
match RE_any (_ : input') = Just input'
match RE_any _ = Nothing
```

5. (Integer constants ≥ 0 that don't begin with a leading zero except for `0` itself.) To make things easier to read, I'll declare a couple of patterns first:

```
digits = RE_or [RE_and [RE_or (map RE_const "0123456789"), digits], RE_empty]
one_to_nine = RE_or (map RE_const "123456789")
```

Then,

```
p5 = RE_or [RE_const '0', RE_and [one_to_nine, digits]]
```

6. (Like problem 5 but with optional leading hyphen)

```
p6 = RE_and [ RE_or [RE_const '-', RE_empty], p5]
    -- where p5 = answer to problem 5
```

7 - 9. (Omitted)