HW 4: Implementing Regular Expressions

CS 440: Programming Languages and Translators

Due Sat Mar 2, 11:59 pm

Programming Problems [50 points total]

For this assignment, you'll be modifying and extending our earlier implementation of matching regular expressions against strings. In addition to the expressions we already have: ∈ (empty string), a specified character, alternation, and concatenation, we'll add

- `$` for end of string: a match of $ with the empty string succeeds and matches the empty string; a match of $ against any non-empty string fails.
- `.` for any single character: dot matches any one character in the alphabet
- `?` for optional expression: *exp* `?` is like (*exp* | ∈): The expression or empty string (note it always succeeds).
- Kleene star: *exp** is like (*exp exp**) | ∈. However, if *exp* matches but returns the empty string as the match, then just succeed with that. This is to avoid infinite recursion on expressions like ∈*.

In addition, there's a change to how *exp1* | *exp2* should behave: If both sides succeed, we want the result with the longer matching string. E.g., a match of (`"a"` | `"ab"`) `"abc"` should succeed with `"ab"` (and leftover `"c"`). The way we were doing it before (skip *exp2* if *exp1* succeeds) can fail to return the longest match, which is now our goal.

Part 1. Backtracking Search

This is a extension and modification to the earlier backtracking search program. There are two data structures

```
data RegExp = ... (the regular expressions)
data Mresult = FAIL | OK String String deriving (Eq, Show)
```

An unsuccessful match should return *FAIL*. A successful result `OK` *str1 str2* means that *str1* matched the expression and *str2* was leftover. Note *str1* ++ *str2* should equal the string passed to match. E.g., matching `(Rch 'a')` against `"abc"` should yield `OK "a" "bc"`.

There's a skeleton program `HW04_part1.hs` -- it compiles but its `match` always returns `FAIL`. You should replace the stub code with your implementation.

Part 2. Nondeterministic Search

For this version of match, instead of returning a single result `FAIL` or `OK`, you should return a list of all successful search results. E.g., matching `(a|ab)` with `"abc"` should succeed with both disjuncts and return `[OK "a" "bc", OK "ab" "c"]`. (In part 1, you should return `OK "ab" "c"`, the longer of the two match results.)

The regular expression data structure stays the same as in part 1. The result type changes:

```
data Mresult = OK String String deriving (Eq, Show)
type Mresults = [Mresult]
```


So `match` returns an `Mresults`, which is a list of `OK` … …. A list of empty indicates a match failed, so we don't need `FAIL` anymore.

Except for the slightly different data structures, many of the cases behave the same as in Part 1, but not all:

- Alternation, *exp1* | *exp2*, becomes easy: Run match on both expressions and concatenate the lists of results.
- Concatenation, *exp1 exp2*, however, is much harder now.
  1. Say matching *exp1* returns 2 results, [`OK` *r1 s1*, `OK` *r2 s2*].
  2. For `OK` *r1 s1*, we have to concatenate *r1* with each result of matching *exp2* with *s1*. I.e., for each result `OK` *r3 s3* of *exp2* with *s1*, we generate `OK` (*r1* ++ *r3*) *s3*.
  3. Similarly, for `OK` *r2 s2*, we have to concatenate *r2* with each result of matching *exp2* with *s2*.
  4. Finally, we can concatenate the results of steps 2 and 3.
- **Example**: match `(a|ab) b` with `"abbc"`.
  - The recursive match of `(a|ab)` with `"abbc"` returns two results, [`OK "a" "bbc"`, `OK "ab" "bc"`].
  - Take `OK "a" "bbc"` and match for `b` with `"bbc"`. This gives us [`OK "b" "bc"`], which we extend to [`OK "ab" "bc"`].
  - Take `OK "ab" "bc"` and match for `b` with `"bc"`. This gives us [`OK "b" "c"`], which we extend to [`OK "abb" "c"`].
  - So match `(a|ab) b` with `"abbc"` returns [`OK "ab" "bc"`, `OK "abb" "c"`]
- Note in this example, the match of *exp2* returned only one result. Remember, if it returns many results, then *each* of those results has to be concatenated with the match of the particular *exp1* search that the *exp2* search follows.
- **Example**:
  - Matching `(a|aa)` with `"aaaaaa"` yields [`OK "a" "aaaaa"`, `OK "aa" "aaaa"`]
  - matching `(a|aa)(aa|a)` with `"aaaaaa"` yields [`OK "aa" "aaaa"`, `OK "aaa" "aaa"`, `OK "aaa" "aaa"`, `OK "aaaa" "aa"`].
  - Note `OK "aaa" "aaa"` appears twice here -- don't worry about it. (Don't look for duplicate results.)

Once again, there's a skeleton, `HW04_part2.hs` with stub code that you should replace. (Currently, its `match` always returns the empty list of results.)

