A real-time lexical analyzer in Haskell over network stream

Rtlex (Real-time Lexical Analyzer) is a lexical scanner over network stream (or any kind of real-time streams) that executes a monadic action whenever a pattern matches some text from the stream. To work on real-time streams, it is implemented on the basis of a very efficient NFA algorithm that can keep matching multiple patterns simultaneously and without backtracking, and as such, does not rely on the stream being recoverable (or bufferable) by using things like `unget()`.

Rtlex uses (self- and mutual- as well) recursive regular expressions for specifying patterns to match, which are easier to write and better to fit in a lexical analyzer than the context-free grammar. Rtlex takes extended regular expressions into which we can embed Haskell variables and arbitrary (in-line) Haskell functions. An embedded variable can refer to other regular expressions including its own regular expression that contains it, and an embedded Haskell function can be used as a zero-width assertion that determines the success or failure of its match based on the (sub-)string that has been partially matched up to the position of the zero-width assertion in the regular expression. Embedded functions are also used to convert the partially matched (sub-)string into another string, in the middle of matching a regular expression. Regular expressions in rtlex are specified wrapped in *quasi quotes* and thus compiled at the compile-time of rtlex.

Actions that will run when their associated patterns are matched are ordinary Haskell monadic functions under some user-specified monad such as `IO`. They are given as an argument the matched input string by the associated patterns. As all actions share the same monad, they can communicate with each other through the monad; we can use, for example, simple mutable reference like `IORef`, or a transformed monad like `StateT u IO` with some user state type `u`.

Why to analyze in real-time?

Legacy lexical analyzer like `flex` is based on finding *the earliest and the longest submatch* and so tries to match its patterns in such a way that: - it tries to match its patterns from top to bottom and one by one (or simultaneously if it is "smart" enough), - if it finds a pattern matches a string that it has read so far from input, it memorizes the matched pattern and the position of input, and then it repeatedly tries other patterns (including the currently matched pattern as well) for any possible longer matches, and - if there are no more patterns that successfully match as it reads characters from input, it finally declares a match with the last pattern that has been remembered to match successfully, and recovers the input to the position corresponding to the last match, to continue the next lexical analysis for further input.

For example, if a legacy lexical analyzer has the following two rules,

```
%%
helloworld { printf("1st action\n"); }
hello      { printf("2nd action\n"); }
```

and if we give it the input, "helloworld", it will not execute either action until it completes to read the whole input string, and then it will execute only the first action. At the moment it reads 'o' in the middle of input, it knows the second pattern can match, but it does not execute the second action immediately, because it wants to match patterns with further input characters as long as possible. And finally when it knows the first pattern matches with the whole input string, it executes the first action, ignoring the second action. What if we give it the input, "helloworks"?

This time, the second action will be executed as expected, but will be executed only when it reads the character 'k', which is the first clue of no success of matching with the first pattern. That means, depending on other patterns, the action corresponding to a matched pattern may not be executed immediately.

Note that most lexical analyzers (including the `flex`) are not so "smart" enough to match multiple patterns simultanenously, and when they get to know the first pattern does not match "helloworks" at the character 'k' they simply backtracks and try to match the second pattern back from the start of the input. (In fact, if they are to be "smart", they need to include actions into their associated regular expressions (regular expression ASTs, to be accurate) and then combine all those regular expressions across rules into a single regular expression, before starting to match them.)

So, to facilitate the analysis of real-time streams, rtlex features:

- immediate matching of patterns, rather than lazy matching to find out any possible longer matches, and
- simultaneous matching of all patterns, rather than trying patterns one by one. (By simultaneous, I do not mean that every pattern is matched through a separate thread, but that patterns are matched in such an interleaved way that there will be no backtracking when a pattern fails to match and then another pattern is tried.)

An example

Here is a simple code that detects "sheer", "she", "he", and "he*r" when given the input string, "sheerEnd".

```
{-# LANGUAGE QuasiQuotes #-}

import Parser
import Rtlex
import Control.Monad (when)

main :: IO Int
main =
    stream (-1) "sheerEnd"  -- main returns (-1) if stream runs out.

    $$ yyLex (const $ return ())
        -- A simple analyzer here does nothing with the resulting reports from actions
        -- that are executed when corresponding patterns match some input.

    $$ rules [
        rule [regex|End|]   $ \s -> yyReturn 0,  -- main returns 0 if "End" is reached.
        rule [regex|sheer|] $ \s -> do putStrLn s; yyReject,
        rule [regex|she|]   $ \s -> do putStrLn s; yyReject,
        rule [regex|he|]    $ \s -> do putStrLn s; yyReject,
        rule [regex|he*r|]  $ \s -> do
            when ( s == "her" ) $
                putStrLn s
```

```
            yyReject
        ]
```

It can match “she” and “he” while matching “sheer”.

```
*Main> main
she
he
sheer
0
```

Another example

By having `StateT (Map String Int) IO` instead of the `IO` monad, we can count occurrences of words with the State monad.

```
{-# LANGUAGE QuasiQuotes #-}

import Parser
import Rtlex
import Control.Monad.State
import qualified Data.Map as Map

main :: IO ()
main = do  -- in the IO monad
    m <- flip execStateT Map.empty $
        -- execStateT returns the final state and discards the final value, which is ()
        -- returned from stream ().

        stream () "ha ha ho hoo hi ha"

        $$ yyLex (\s -> do  -- in the "StateT (Map String Int) IO" monad
            modify $ Map.insertWith (+) s 1  -- stores occurrences of each word in a Map
            lift $ putStrLn s)               -- and prints each word as well.

        $$ rules [
            rule [regex|ha|ho|hi|] $ \s -> yyAccept s  -- reports each word to the yyLex.
            ]

    print $ Map.toList m  -- finally prints the counts in the Map.
```

```
*Main> main
ha
ha
ho
ho
hi
ha
[("ha",3),("hi",1),("ho",2)]
```

A network-stream version of this code is introduced at the **Network streams** section near the end of this document.

About general form

Rtlex has the following general form.

```
{-# LANGUAGE QuasiQuotes #-}

import Parser
import Rtlex

analyzer :: m r
analyzer =
    stream r0 "string"              -- r0 :: r

    $$ yyLex yacc                   -- yacc :: a -> m b

    $$ rules [
        rule [regex|re1|] action1,  -- action :: String -> m (ActionResult r a)
        rule [regex|re2|] action2,
        ...
        rule [regex|reN|] actionN
        ]
```

- `QuasiQuotes` language extension is for specifying regular expressions within `[regex|...|]`. And regular expressions are compiled (that is, encoded into regular expression ASTs) at the compile-time of source code. So, if there is an error in a regular expression, it will be reported at compile-time. The syntax and semantics for regular expressions are described below.
- `import Parser` imports the `[regex|...|]` quasi quoter and the regular expression engine.
- `import Rtlex` imports: `Stream`, `stream`, `stream0`, `yyLex`, `rules`, `($$)`, `rule`, `yyReturn`, `yyAccept`, and `yyReject`.
- The `analyzer` consists of three sections, `stream`, `yyLex`, and `rules`, separated by `$$` operator. Each section is actually implemented as a coroutine that interacts with each other internally using yield (and resume), and the `$$` operator plays the role of binding those coroutines. The middle coroutine `yyLex` acts as a proxy between the upper and the lower coroutines, and reads each input character from the input stream and passes it to `rules`. The `rules` tries to match each rule in its rules list with the given character, runs actions that correspond to successfully matched patterns, and reports the results from such actions to `yyLex` one by one. Then with each result from `rules`, `yyLex` calls `yacc` that is present as its argument, before `yyLex` repeats the next cycle of reading another character from the stream and so on.
- `stream` introduces an input stream and takes two arguments, `r0` and a string. `r0` can be any user-determined value of type `r`, and will be returned as a result from `analyzer` when the end of stream is reached. String is a list of characters to be served as the stream. Instead of a string, a bytestring or anything from an instance of `Stream` class can be used as well.

- `yyLex` takes a user-defined function called `yacc`, which has type of `a -> m b`. `yacc` is a monadic function, taking each `a` that results from actions when their corresponding patterns are matched, and returning `b` under the user-determined monad `m`. The monad `m` will usually be the `IO` monad or some transformed monad of `IO`, but any monad will be ok. The monad result `b` is currently not used and can be anything, but is reserved for a future extension and will be possibly used to communicate with the stream to control it.
- `rules` introduces rules in a list. As such, each rule in the list must be separated with a comma, "`,`".
- `rule` combines a quasi-quoted regular expression and a user-defined `action` function into a rule. Each pattern of rules is matched as characters are read from the input stream, and if a pattern successfully matches a string from the stream up to the current character, the corresponding action is called with the matched string by the pattern as an argument. (More details about the matching algorithm are explained below.) Every `action` has type of `String -> m (ActionResult r a)`, where `ActionResult` type is defined as:

  ```
  data ActionResult r a
      = Return r  -- to finish the lexical analyzer immediately with value "r"
      | Accept a  -- to accept the current match and report value "a" to lexical analyzer
      | Reject    -- to reject the current match and try other actions
  ```

  As you see, there are two user-determined types involved, `r` and `a` that are already introduced above. `a` is for reportng a value to `yyLex` and thus `yacc`, and `r` is for stopping and exiting the `analyzer` immediately with the return value of a `r`. So, before reaching the end of stream, we can early exit from `analyzer` using the `Return`. `Accept` is used to accept the current match and report an `a` to `yyLex`, blocking further actions that also have their patterns matched from being executed. `Reject` just passes control over to the next matched action of having an associated pattern matched. (See the details below.) Note that the `action` functions and `yacc` function run under the same shared monad `m`, which means they can interact with each other through the monad. To make it convenient to use those constructors of `ActionResult` under the monad, three short-cuts are provided:

  ```
  yyReturn :: Monad m => r -> m (ActionResult r a)
  yyReturn = return . Return

  yyAccept :: Monad m => a -> m (ActionResult r a)
  yyAccept = return . Accept

  yyReject :: Monad m => m (ActionResult r a)
  yyReject = return Reject
  ```

Regular expressions

In rtlex, patterns are written in regular expressions instead of in context-free grammar. Since the regular expressions here are extended to embed any Haskell expressions (as well as variables) that lead to other regular expressions, we will see these regular expressions are more powerful than context-free grammar in terms of the recognizable languages. Moreover, the engine for matching such regular expressions is implemented with the Glushkov NFA algorithm, which runs efficiently

in *O(nm)* where n is the length of the input and m the size of the regular expression. The algorithm does not involve any backtracking to match every possible alternative in regular expressions, and thus works best with real-time streams that are hard to take back characters that have already been consumed.

The LL grammar for regular expressions that rtlex takes is:

```
Regex        = ParseAlt
ParseAlt     = ParseAlt "|" ParseAnd | ParseAnd              (left associative)
ParseAnd     = ParseAnd "&" ParseSeq | ParseSeq             (left associative)
ParseSeq     = ParseSeq ParseTerm | ParseTerm               (left associative)
ParseTerm    = <a character>
             | "."
             | "[" ["^"] <characters> "]"
             | "${" [<a variable>] "}"
             | "{" <a Haskell function> "}"
             | "(" ParseAlt ")"
             | ParseTerm { "?" | "*" | "+" }
```

- `"|"` (*alternation*): `[regex|foo|bar|]` matches "foo" and matches "bar", and `[regex|land|island|]` matches "land" and matches "island".

  ```
  main :: IO ()
  main =
      stream () "island"
      $$ yyLex (const $ return ())
      $$ rules [
          rule [regex|land|island|] $ \s -> do putStrLn s; yyAccept ()
          ]
  -- Output will be:
  -- *Main> main
  -- land
  -- island
  ```

  Note, unlike the case with `[regex|foo|bar|]` that has no occurrences of matching both "foo" and "bar", `[regex|land|island|]` matches both alternatives at the moment it reads the character 'd' from input, and in this case, the corresponding action is executed for each matched string. Also note that in such a case with `"|"` operator, we have no way of executing the action for one alternative over another. Whereas if we write the alternatives in separate rules we can choose one of the corresponding actions by using `yyAccept` (See the details below).

  ```
  main :: IO ()
  main =
      stream () "island"
      $$ yyLex (const $ return ())
      $$ rules [
          rule [regex|island|] $ \s -> do putStrLn s; yyAccept (),
          rule [regex|land|]   $ \s -> do putStrLn s; yyAccept ()
          ]
  -- Output will be:
  ```

```
-- *Main> main
-- island
```

- `"&"` (*and*): `[regex| & |]` matches a string that is matched with both and *at the same time*. So, `[regex|foo.*&.*bar|]` matches a string from input that starts with "foo" and ends with "bar".

  Among words separated by spaces, if we want to choose the words that contain a number, we can use:

```
main :: IO ()
main =
    stream () " abc de fgh1 ijk 23lm "
    $$ yyLex (const $ return ())
    $$ rules [
        rule [regex|.*[0123456789].*& [^ ]+ |] $ \s -> do putStrLn s; yyAccept ()
        ]
-- Output will be:
-- *Main> main
-- fgh1
-- 23lm
```

- (*Operator precedence*): All operators are listed from the highest precedence to the lowest as:

```
?, *, +                             (postfix, same precedence)
sequencing (juxtaposition)          (binary, left-associative)
&                                   (binary, left-associative)
|                                   (binary, left-associative)
```

- `"."`: `"."` matches any single character (including whitespaces and control characters such as a new-line). So, `[regex|a.*b|]` matches strings of any length between "a" and "b" including "a" and "b". (However, be careful in using `".*"` in a pattern, because as a real-time analyzer, rtlex matches "." with any character includig a new-line, and `".*"` will match the whole input stream if not accompanied by proper boundary expressions.)

- `"[...]"` (*character class*) and `"[^...]"` (*negated character class*): `"[...]"` matches any single character from input that is listed inside the bracket, and `"[^...]"` matches one that does not listed inside it. (As for now, the character classes do not recognize character ranges such as `"[0-9]"`, so every character should be listed literally like as `"[0123456789]"`.)

- "*char*" (*escaped character*): Meta characters used as regular expression operators can be escaped with a preceding backslash `"\"`, and single-character escape codes such as `"\n"` can be used the same as in the Haskell and the C languages.

- `"*"` (*Kleene closure*), `"+"` (*positive closure*), and `"?"` (*options*): `[regex| *|]` matches zero or more times, `[regex| +|]` matches one or more times, and `[regex| ?|]` matches zero or one time, that is, matches once but optionally.

- `"${var}"` and `"${}"` (*reference to other regular expression*): A regular expression can contain references to other regular expressions and its own regular expression as well. `"${}"` represents the whole regular expression that is currently being defined, and it is used to make a self-recursive regular expression. `"${var}"` embeds a reference to other regular expression through

a variable name. However, `"${var}"` actually can take on any Haskell expression that leads to a regular expression.

We can specify a regular language of {a^n b^n | n >= 0} using the expression, `"X = (aXb)?"`, which cannot be described with the ordinary (non-recursive) regular expressions.

```
main :: IO ()
main =
    stream () "aaaaaabbbaaabb"
    $$ yyLex (const $ return ())
    $$ rules [
        rule [regex|(a${}b)?|] $ \s -> do putStrLn s; yyAccept ()
        --or we could also use: rule (let x = [regex|(a${x}b)?|] in x) $ \s -> ...
    ]
-- Output will be:
-- *Main> main
-- ab
-- aabb
-- aaabbb
-- ab
-- aabb
```

Using a function returning a regular expression, we can even make a more powerful expression that recognizes {a^n b^n c^n | n >= 1}, which is known that it cannot be described by the context-free grammar.

```
main :: IO ()
main =
    stream () "aaaabbbcccc"
    $$ yyLex (const $ return ())
    $$ rules [
        rule [regex|${abc nul}|] $ \s -> do putStrLn s; yyAccept ()
    ]

    where
    nul    = [regex|()|]  -- "()" represents  that matches an empty string.
    abc bc = let bc' = [regex|b${bc}c|] in
             [regex|a(${bc'}|${abc bc'})|]
    -- To avoid left-recursion, we have used:
    -- abc | aabbcc | aaabbbccc | ... == a(bc | a(bbcc | a(bbbccc | a(...)))).
-- *Main> main
-- aaabbbccc
```

However, left recursion should be avoided or it will lead to an infinite loop. Every recursive call has to be guarded by a (non-terminal) symbol just as with parser combinators. For example, the expression, `"a*"` can be represented in context-free grammar as either `"X = (aX)?"` or `"X = (Xa)?"`, but the directly translated expression, `[regex|(${}a)?|]` corresponding to the latter will lead to an infinite loop. (Left recursions can be eliminated by left-factoring the grammar. See here.)

- `"{fun}"`: For a zero-width assertion, We can embed a Haskell function into regular expressions.

Functions should be of type `String -> [String]`, and they are provided as an argument with a partially matched string up to each position of them inside regular expressions. As each function acts as an assertion, it can decide whether it matches or not on its position based on the given partially matched string. For example, the following code matches a "B" only after a new-line. The first symbol in the pattern is "." and matches any single character, so the function inside `{...}` takes as the argument s, the single-character string matched by ".". If s is a new-line, the function declares that it also matches by returning a non-empty list, `[""]`, otherwise it returns an empty list, which makes the current match fail and does not give a chance to the next symbol "B" to match with further character from input (the further character has to be matched from the start, with the first symbol ".").

```
main :: IO ()
main =
    stream () "A is A.\nB is B.\nC is C.\n"
    $$ yyLex (const $ return ())
    $$ rules [
        rule [regex|.{\s -> if s == "\n" then [""] else []}B|] $
            \s -> do putStrLn s; yyAccept ()
    ]
-- *Main> main
-- B
```

The assertion function also acts as a converter, and so it can convert the partially matched string up to its position into another string. In fact, in the above example, the assertion function actually converts a new-line character into the null string, `""`, otherwise the finally matched string that is printed on the screen would be "\nB" instead of "B". Also note that it returns the converted string in a list rather than as is. Actually, an assertion function can convert a string into multiple strings, and when some (or all) of the strings finally pass through all the remaining patterns behind the assertion function, they will be fed to their corresponding action one be one.

```
main :: IO ()
main =  -- finds ".he" only if it is "she", and converts it into "SHE".
    stream () "he she and they"
    $$ yyLex (const $ return ())
    $$ rules [
        rule [regex|.he{\s -> if s == "she" then ["SHE"] else []}|] $
            \s -> do putStrLn s; yyAccept ()
    ]
-- *Main> main
-- SHE
```

Note that unlike the `yacc` function and `action` functions, assertion functions are pure and do not run under the user-specified monad. So, they cannot interact with each other, nor with `yacc` or `action` functions.

- `"(...)"`: Regular expressions can be grouped in parentheses to limit the scope of operators. The empty parentheses `"()"` represents that matches an empty string. So, `" ?"` is an equivalent expression to `" |()"`.
- For optimal code generation, zero-width assertions `"{fun}"` and pattern `"()"` are supported

only when the `"-DZERO_WIDTH_ASSERTION"` pre-processor constant is defined in `ghc` or `ghci`.

Details about matching rules

Every possible submatch rather than the earliest and the longest submatch

Whereas legacy lexical analyzers are eager to find *the earliest and the longest submatch*, rtlex tries to find *every possible submatch*. For that, rtlex keeps matching all patterns constantly and simultaneously. *Constantly*, because every time rtlex reads an input character from a stream rtlex tries to match all patterns with it, thinking as if the character can start a new match with any of the patterns. *Simultaneously*, because rtlex considers the possibilities of successful matching for all the patterns at a time, so that it does not need to backtrack and reconsider some unconsidered patterns later.

Here, while reading the second occurrence of "aba" from input, rtlex considers it as a new match with the `[regex|abac|]` pattern, even though rtlex is in the middle of considering the first occurrence of "aba" with the same pattern.

```
main :: IO ()
main =
    stream () "ababac"
    $$ yyLex (const $ return ())
    $$ rules [
        rule [regex|abac|] $ \s -> do putStrLn s; yyAccept ()
    ]
-- *Main> main
-- abac
```

In the code below, rtlex considers the two patterns simultaneously while reading "ab" from input.

```
main :: IO ()
main =
    stream () "abd"
    $$ yyLex (const $ return ())
    $$ rules [
        rule [regex|abc|] $ \s -> do putStrLn s; yyAccept (),
        rule [regex|abd|] $ \s -> do putStrLn s; yyAccept ()
    ]
-- *Main> main
-- abd
```

Using the nature of finding "every possible submatch", we can detect the start and the end of a pattern in real-time when the pattern occurs consecutively.

```
main :: IO ()
main =
    stream () "aaabbbbbbccc"
    $$ yyLex (const $ return ())
    $$ rules [
        -- repeated b's are discarded
```

```
        rule [regex|bb|]    $ \s -> yyAccept (),

        -- catch the first b
        rule [regex|b|]     $ \s -> do putStrLn "Start of b's"; yyAccept (),

        -- catch the last b
        rule [regex|b[^b]|] $ \s -> do putStrLn "End of b's"; yyAccept ()
    ]
-- *Main> main
-- Start of b's
-- End of b's
```

Actions are executed from top to bottom, but selectively.

The `rule` combines a quasi-quoted regular expression and a user-defined action into a rule. Each pattern of rules is matched as characters are read from the input stream, and if a pattern successfully matches a string from the stream up to the current character, the associated action is called with the matched string by the pattern as an argument. Every action has type of `String -> m (ActionResult r a)`, where `ActionResult` type is defined as:

```
data ActionResult r a
    = Return r  -- to finish the lexical analyzer immediately with value "r"
    | Accept a  -- to accept the current match and report value "a" to lexical analyzer
    | Reject    -- to reject the current match and try other actions

yyReturn :: Monad m => r -> m (ActionResult r a)
yyReturn = return . Return

yyAccept :: Monad m => a -> m (ActionResult r a)
yyAccept = return . Accept

yyReject :: Monad m => m (ActionResult r a)
yyReject = return Reject
```

However, unlike patterns are tried to match constantly and simultaneously, not all matched actions (that is, actions associated with matched patterns) are executed always. As monads, actions are executed in the order of top-to-bottom, and each action is executed only if its all previous actions give it a way by returning `Reject`. In other words, if an action returns `Accept`, the actions below it are not executed. (Note, however, returning `Reject` or `Accept` affects only the execution of actions, and has nothing to do with matching patterns; all patterns are tried matching always!)

```
main :: IO ()
main =
    stream () "she"
    $$ yyLex (const $ return ())
    $$ rules [
        rule [regex|she|] $ \s -> do putStrLn s; yyReject,
        rule [regex|he|]  $ \s -> do putStrLn s; yyAccept ()
    ]
```

```
-- *Main> main
-- she
-- he

main :: IO ()
main =
    stream () "she"
    $$ yyLex (const $ return ())
    $$ rules [
        rule [regex|she|] $ \s -> do putStrLn s; yyAccept (),
        rule [regex|he|]  $ \s -> do putStrLn s; yyAccept ()
    ]
-- *Main> main
-- she
```

A pattern can pass multiple strings to the corresponding action in a match.

The last example above can be rewritten with a single pattern as follows. As the `[regex|he|she|]` pattern can match "he" and "she" at the same time when reading 'e' from the input string, "she", it passes both of them over to its action, calling the action with each of them as the argument. And in this case, we cannot control the multiple executions of the same action using `Accept` or `Reject`, as they only affect the execution of actions that come below.

```
main :: IO ()
main =
    stream () "she"
    $$ yyLex (const $ return ())
    $$ rules [
        rule [regex|he|she|] $ \s -> do putStrLn s; yyAccept ()
    ]
-- *Main> main
-- he
-- she
```

Be careful when using quantification operators, `"*"` and `"+"`, that rtlex will match all the possible (sub-)strings from input stream.

```
main :: IO ()
main =
    stream () "aaa"
    $$ yyLex (const $ return ())
    $$ rules [
        rule [regex|a*|] $ \s -> do putStrLn s; yyAccept ()
    ]
-- *Main> main
-- a
-- a
-- aa
-- a
```

```
-- aa
-- aaa
```

The results from the example above may look redundant, but if we think of the input string as "a1a2a3" they correspond to:

Then, we might have a question, if an action is called multiple times in a match, for each matched string by the corresponding pattern, and if the action returns `Accept` for some of the strings and `Reject` for others, how does it affect the action that follows it? The answer is that the action behaves just the same when it is called separately for different matches from its pattern, so we do not need to consider this case specially.

```
import Data.Char (toUpper)
main :: IO ()
main =
    stream () "abc"
    $$ yyLex putStrLn
    $$ rules [
        rule [regex|ab|b|] $ \s -> if s == "b" then yyAccept s else yyReject,
        rule [regex|.b{\s -> [map toUpper s]}|] $ \s -> yyAccept s
    ]
-- *Main> main
-- b
-- AB
```

In the above example, at the moment the character 'b' is reached, the first pattern `[regex|ab|b|]` matches both "ab" and "b" at the same time, and the corresponding action returns `yyAccept "b"` for "b" and `yyReject` for "ab". Although this case is processed in a rather complex way internally, we can easily think that "b" is accepted and gets passed to `putStrLn` through `yyLex`, and at the same time, since "ab" is ignored by the first action the second action is executed and converts it into its capitalized string, which is then passed over to `putStrLn` as is `yyAccept`ed.

So, we can just think that an action is called for each match regardless of whether the matches occur at the same time in a pattern or not.

`yyLex` is also called for each match, but with some user-determined value than the matched string.

The `yyLex` is called whenever a matched action returns something with `yyAccept`, and then `yyLex` calls the `yacc` that is specified as its argument and feeds the returned result from the action to `yacc`. So, whenever there is a match, a corresponding action is called and then `yacc` is also called. But, whereas action is given the matched string as its argument, `yacc` is given the result that is returned by such an action. In a sense, we can think of actions as converters from String into a value of a user-determined type `a`.

```
action :: String -> m (ActionResult r a)
yacc   :: a -> m b
```

The `yacc` returns `m b`, a value wrapped in a user-determined monad `m`. However, since `yacc` is called each time there is a match from a pattern, and since it is not called at once with all the matches together, the intermediate results of `m b` cannot be used outside of `yacc`, and they are

simply ignored (actually, reserved for future use) outside of `yacc`. That's actually what `yacc` is for. If we want to process the result from an action each time a pattern matches, we put the procedure into `yacc`. However, if we want the results from actions at once at the end of lexical analysis, we need to gather those results using some monad.

```
import Control.Monad.State

gather :: IO [String]
gather =  -- in the IO monad
    flip execStateT [] $
    -- execStateT returns the final state and discards the final value, which is ()
    -- returned from stream ().

    stream () "ha ha ho ho hi ha"

    $$ yyLex (\s -> do  -- in the "StateT [String] IO" monad
        modify $ (s:))  -- stores each word in a list

    $$ rules [
        rule [regex|ha|ho|hi|] $ \s -> yyAccept s  -- reports each word to the yyLex.
        ]

-- *Main> gather
-- ["ha","hi","ho","ho","ha","ha"]
```

About `r`

The `r` is the type of the final result from our lexical analyzer. It can be returned by either `stream` or an action. `stream` returns an `r` when it has reached the end of its stream. An action can use `yyReturn` to return an `r`, making our lexical analyzer stop immediately. If our lexical analyzer needs to keep running to the end of a stream, chances are we don't need to `yyReturn` it in any action unless there is an exceptional case in the stream. But, if we expect our lexical analyzer to exit early when encountering a certain pattern in the stream, we can make use of it.

Network streams

Instead of `stream`, a more generic stream-reading function `stream0` is provided to work with any stream of an instance of `Stream` class, which is very simple because it is not assumed to be recoverable. And, by defining a proper `getc` method for a custom stream, we can use it with `stream0`.

```
class Stream s m r c | s -> r c where -- needs FunctionalDependencies
    getc :: s -> m (Either r (c, s))
    -- getc will return either r in case of an error in the stream, or (c, s) otherwise.
```

For example, we can make an example server program that was introduced earlier to count occurrences of some words.

```haskell
{-# LANGUAGE QuasiQuotes, FlexibleInstances, MultiParamTypeClasses #-}

import Parser
import Rtlex
import Control.Monad.State
import qualified Data.Map as Map
import Network
import System.IO
import System.IO.Error

-- Handle from IO monad as a Stream instance
instance MonadIO m => Stream Handle m Int Char where
    -- needs FlexibleInstances, MultiParamTypeClasses
    getc handle = liftIO $
        catchIOError
        (do c <- hGetChar handle; return $ Right (c, handle))
        (\e -> return $ Left $ if isEOFError e then 0 else -1)
        -- r == 0 if EOFError, -1 for other errors

main :: IO ()
main = withSocketsDo $ do  -- in the IO monad
    sock <- listenOn $ PortNumber 3001
    putStrLn "Starting server ..."
    (handle, host, port) <- accept sock

    m <- flip execStateT Map.empty $
         stream0 handle

         $$ yyLex (\s -> do  -- in the "StateT (Map String Int) IO" monad
            modify $ Map.insertWith (+) s 1  -- stores occurrences of each word in a Map
            lift $ putStrLn s)               -- and prints each word as well.

         $$ rules [
            rule [regex|ha|ho|hi|] $ \s -> yyAccept s  -- reports each word to the yyLex.
            ]

    hClose handle
    print $ Map.toList m  -- finally prints the counts in the Map.
    putStrLn "Server closed."
```

When we run the above program and run some client program, we will get things like:

```
$ nc localhost 3001
ha ha ho hoo hi ha
^D

*Main> main
Starting server ...
ha
```

```
ha
ho
ho
hi
ha
[("ha",3),("hi",1),("ho",2)]
Server closed.
```