CSCI 3155: Lab 6 (due April 20, 2018)

Ian Smith & Eric Minor

3(b) i. ***Exercise.*** *In your write-up, give a refactored version of the re grammar from Figure 1 that eliminates ambiguity in BNF (not EBNF). Use the following template for the new non-terminal names:*

$$
\begin{aligned}
re &::= union \\
union &::= union \text{ `|' } intersect \mid intersect \\
intersect &::= intersect \text{ `\&' } concat \mid concat \\
concat &::= concat\ not \mid not \\
not &::= \sim not \mid star \\
star &::= star* \mid star+ \mid star? \mid atom \\
atom &::= ! \mid \# \mid . \mid c
\end{aligned}
$$

ii. A recursive descent parser following this grammar would go into an infinite loop because on evaluating a regular expression *re* the parser would look for something that matches the non-terminal production of *re*, *union*. It would then look at the productions of *union* and try to match against the first production, which is something that looks like *union*. But *union* is defined in terms of *union* and there is no base case to match against, so the parser would continue trying to match against something that looks like *union* on the left and would go into an infinite loop.

iii. ***Exercise.*** *In your write-up, give a refactored version of the re grammar that replaces left-associative binary operators with n-ary versions using EBNF using the following template:*

$$
\begin{aligned}
re &::= union \\
union &::= intersect \{\text{ `|' } intersect\} \\
intersect &::= concat \{\text{ `\&' } concat\} \\
concat &::= not \{not\} \\
not &::= \{\sim\} star \\
star &::= atom \mid atom\{*\} \mid atom\{+\} \mid atom\{?\} \\
atom &::= ! \mid \# \mid . \mid c
\end{aligned}
$$

iv. ***Exercise.*** *In your write-up, give the full refactored grammar in BNF without left recursion and new non-terminals like unions for lists of symbols. You will need to introduce new terminals for intersects and so forth.*

$$
\begin{aligned}
re &::= union \\
union &::= intersect\ unions \\
unions &::= \epsilon \mid \text{`|'}\ intersect\ unions \\
intersect &::= concat\ intersects \\
intersects &::= \epsilon \mid \text{`\&'}\ concat\ intersects \\
concat &::= not\ concats \\
concats &::= \epsilon \mid not\ concats \\
not &::= nots\ star \\
nots &::= \epsilon \mid \sim nots \\
star &::== atom\ stars \\
stars &::= \epsilon \mid stars\{*\} \mid stars\{+\} \mid stars\{?\} \\
atom &::= ! \mid \# \mid . \mid c \mid \text{`('}re\text{`)'}
\end{aligned}
$$

Because we need parentheses in the parser that we are implementing, I added parentheses to this final grammar, so that it is clear how they should be implemented in my Scala parser.

3(c) i. ***Exercise.*** *In your write-up, give typing and small-step operational semantic rules for regular expression literals and regular expression tests based on the informal specification given above. Clearly and concisely explain how your rules enforce the constraints given above and any additional decisions you made.*

$$\frac{}{\Gamma \vdash /\hat{}re\$/ : \text{TReg}}\text{TypeRegEx}$$

$$\frac{\Gamma \vdash e_1 : \text{TReg} \quad \Gamma \vdash e_2 : \text{TString}}{\Gamma \vdash e_1.test(e_2) : \text{TBool}}\text{TypeRETest}$$

This first type rule enforces the base case rule that a regular expression is of type TReg, this is an axiom. The second type rule says that testing whether a string matches a Regular Expression return a type bool. This is true so long as we test a TReg against a TString, otherwise the typefield and regular call rules would apply.

$$\frac{r = /\hat{}re\$/ \quad s = str \quad b = matcher(r, s)}{\langle M, r.test(s)\rangle \rightarrow \langle M, b\rangle}\text{DoRegTest}$$

$$\frac{\langle M, e_1\rangle \rightarrow \langle M', e_1'\rangle}{\langle M, e_1.test(e_2)\rangle \rightarrow \langle M', e_1'.test(e_2)\rangle}\text{SearchRegTest1}$$

$$\frac{r = /\hat{}re\$/ \quad \langle M, e_2\rangle \rightarrow \langle M', e_2'\rangle}{\langle M, r.test(e_2)\rangle \rightarrow \langle M', r.test(e_2')\rangle}\text{SearchRegTest1}$$

The first step rule DoRegTest says that when test is called on a regular expression with a string type argument, we should immediately call our matcher function (in our Scala interpreter this is our `retest()` function). This process does not change memory so we simply step our expression to the boolean that is returned by the matcher function. The matcher function checks whether the string matches the regular expression. This DoRegTest inference rule avoids conflicting with GetField step rules by requiring that r is a regular expression and that s is a string.
The second step rule SearchRegTest1 simply steps expressions of the form, $e_1.test(e_2)$ by stepping e_1 until it is a value, this rule would also work to step GetField() expressions, which are already stepped in this manner. This inference rule just makes it explicit that this is done for ReGex test.
The final rule SearchRegTest2, takes an expression $r.test(e_2)$ where r is a regular expression value already and then steps the second argument to a value. If this e_2 is eventually stepped to a string then we can perform the Do rule, otherwise we get stuck, which is okay because our language does not support calling test on a regular expression with an argument that isn't a string.

Note: I used `http://cs.txstate.edu/~ch04/webtest/teaching/courses/5318/lectures/slides2/s4-amb-assoc-prec.pdf` as a resource to reference when writing my BNF grammar unambiguously.