Lab 6 Writeup

Justin Cai

April 21, 2018

1 Regular Expression Parser: Recursive Descent Parsing

i. *In your write-up, give a refactored version of the re grammar from Fig- ure 1 that eliminates ambiguity in BNF (not EBNF).*

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

ii. *Explain briefly why a recursive descent parser following your grammar with left recursion would go into an infinite loop.* Consider the simpler grammar of expressions:

$$e ::= n \mid e + n$$

where n are numbers. To parse the string “1+2,” we would check to see if the string matches n, which it does not, so then it would check to see if the string matches $e + n$, which involves checking to see the string matches an e, then $+$, and n. The problem is that the we initally see if the string matches an e, this invokes another recursive check to match an e, an it will infinitely recurse.

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

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

iv. *In your write-up, give the full refactored grammar in BNF without left recursion and new non-terminals like unions for lists of symbols.*

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

2 Regular Expression Literals in JavaScripty

i. *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.*

Typing rules:

$$\textsc{TypeRe} : \frac{re \in \mathscr{L}(re)}{/\hat{}re\$/ : \textbf{RegExp}}$$

$$\textsc{TypeReTest} : \frac{\Gamma \vdash e_1 : \textbf{RegExp} \qquad \Gamma \vdash e_2 : \textbf{string}}{e_1.test(e_2) : \textbf{bool}}$$

TypeRe adds typing for a regular expression literal, with the previous judgement that the re is a valid regular expression. TypeReTest says that the expression $e_1.test(e_2)$ has a type of bool, given that e_1 is a regular expression and e_2 is a string.

Small-step operational semantics:

$$\textsc{SearchReTest} : \frac{e_2 \to e_2'}{/\hat{}re\$/.test(e_2) \to /\hat{}re\$/.test(e_2')}$$

$$\textsc{DoReTest} : \frac{b' = retest(/\hat{}re\$/, s)}{/\hat{}re\$/.test(s) \to b'}$$

SearchCall will step e_1 in $e_1(e_2)$ to $/\hat{}re\$/.test$ (or `Call(GetField(`e_1`,"test"), List(`e_2`))` as an AST node). In order for the interpreter to step this, these two rules need to be implemented. If e_2 is not fully stepped to a string, then SearchReTest will step e_2. Once e_2 is stepped to a string, then DoReTest will perform the test by calling the `retest` function.