Jacob Berman
Lab6 Write Up
7/28/18

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:*

re ::= union
union ::= union `|` intersect | intersect
intersect ::= intersect & concat | concat
concat ::= concat not | not
not ::= ~ not | star
star ::= star+ | star* | star? | atom
atom ::= c | # | . | !

II. ***Exercise:** Explain briefly why a recursive descent parser following your grammar with left recursion would go into an infinite loop.*

A recursive descent parser following our grammar with left recursion would go into an infinite loop because it would never reach any terminals. It would always match on the non-terminals and never reach an exit.

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:*

re ::= union
union ::= intersect {`|` intersect}
Intersect ::= concat {`&` concat}
concat ::= not {not}
not ::= {‘~’} star
star ::= atom | atom {‘*’} | atom {‘+’} | atom {‘?’)
atom ::= ‘!’ | ‘#’ | ‘.’ | ‘c’

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.

re ::= union
union ::= intersect unions
unions ::= € | `|` intersect unions
intersect ::= concat intersects
intersects ::= € | `&` concat intersects
concat ::= not concats
concats ::= € | not concats
not ::= nots star
nots ::= € | '~' nots
star ::= atom stars
stars ::= € | stars {'*'} | stars {'+'} | stars {'?'}
atom ::= 'c' | '#' | '.' | '!' | `(`re`)`

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.

TypeRegExp:

[nothing goes here]

Γ ⊢ /^ re1 $/ : [RegExp]

This is the base case rule that makes sure that that re1 a regular expression is the proper type.

TypeTest:

Γ ⊢ e1 : [RegExp]   Γ ⊢ e2 : [string]

Γ ⊢ [e1.test(e2)] : [bool]

TypeTest working by making sure e1 is a RegExp and e2 is a string. TypeTest will return a Boolean. The Boolean will be true if we test RegExp and a string.

$e_1 \rightarrow e_1'$

------------------------------------ SearchTest1

e_1.test(e_2)→ e_1’.test(e_2)

SearchTest1 is stepping on e_1 until it reaches a value.

e_2→e_2’

------------------------------------ SearchTest2

/^re$/.test($e_2$)→ /^re$/.test(e_2’)

SearchTest2 is stepping on e_2 until it reaches a value.

r→/^re$/ s → str b = retest(r,s)

--DoRegTest

r.test(s)→ b

In DoRegTest the first step test is called on a regular expression with a string as an argument. We will then call our retest function. We are stepping the expression to a Boolean that is returned by retest.