Programming Languages: Recitation 3

Goktug Saatcioglu

02.14.2019

1 HW1 Problem 1

(a) We split the regular expression into two parts. An identifier has to start with a letter A to Z, a to z or and underscore _. Let $\ldots$ represent a range such that $A \mid \ldots \mid Z$ is the alternation regular expression of strings A through Z and $a \mid \ldots \mid z$ is the alternation regular expression of strings a through z. Thus, we get the following regular expression for a valid identifier start

$$(A \mid \ldots \mid Z \mid a \mid \ldots \mid z \mid \_).$$

Then zero or more letters, underscores and digits follow the start. So we can use our starting regular expression and augment it with digits and then enclose the whole expression in paranthesis to use the Kleene star to get zero or more. Thus, we get the following regular expression for the rest of the identifier

$$(A \mid \ldots \mid Z \mid a \mid z \mid \_ \mid 0 \mid \ldots \mid 9)^*.$$

Bringing everything together gives us the answer:

$$(A \mid \ldots \mid Z \mid a \mid \ldots \mid z \mid \_)(A \mid \ldots \mid Z \mid a \mid \ldots \mid z \mid \_ \mid 0 \mid \ldots \mid 9)^*.$$

(b) We split the problem into smaller parts. Notice that the smallest part involves digits that exclude 0 and 1 and digits that do not exclude 0 and 1. So we define two new regular expressions. Let d be the regular expression for digits 0 through 9 meaning

$$d = 0 \mid \ldots \mid 9.$$

Let e be the regular expression for digits 2 through 9 meaning

$$e = 2 \mid \ldots \mid 9.$$

Now we proceed with the area code. An area code may not begin with 0 or 1 and consists of three digits so we can use our smaller expressions d and e getting us

$$edd.$$

Furthermore, we know that an area code may be enclosed by paranthesis or not enclosed at all. This gives us the following solution for area codes

$$(edd) \mid edd.$$

Side note: Why are solutions that try to attempt to check if an initial paranthesis was opened impossible to implement? Next, the cental office code is three digits and cannot begin with a 0 or 1 and proceeds with all digits. This gives us the same solution as for the area code (minus the parantheses)

$$edd.$$

Then, the subscriber number is 4 digits with no restrictions giving us

$$dddd.$$

Finally, the numbers are either seperated by either a dot or a dash which gets us the answer

$$[[(edd) \mid edd] \; - \; edd \; - \; dddd] \mid [[(edd) \mid edd] \, . \, edd \, . \, dddd].$$

We use square brackets [and] to group regular expressions together and avoid confusion between parantheses which are part of the alphabet and note that the dot . does not refer to the wildcard regular expression in this case (why?).

(c) The solution to this question may seem complicated but we can solve it by breaking the question into parts. We begin by noticing that our regular expression should be of the form

$$\texttt{"}\ldots\texttt{"}$$

where we have to fill in the blank area ... (i.e. between the quotes). We start with point (i). If we start with \then there are certain rules we must follow. Thus, \is followed by either b, t, n, f, r, ", ', \, unicode escape or octal escape. Unicode escapes are of the form $uhhhh$ where h is a hexadecimal digit. Let

$$\texttt{hdigit} = A \mid \ldots \mid F \mid a \mid \ldots \mid f \mid \mid 0 \mid \ldots \mid 9$$

be the regular expression for a single hexadecimal digit. Thus, a unicode escape will be of the form

$$\texttt{u hdigit hdigit hdigit hdigit}.$$

However, this is not exactly correct. We have to avoid the newline and linefeed character. Define the following new regular expressions

$$\texttt{nzhdigit} = A \mid \ldots \mid F \mid a \mid \ldots \mid f \mid \mid 1 \mid \ldots \mid 9$$

where nz means not zero,

$$\texttt{nfzhdigit} = A \mid \ldots \mid E \mid a \mid \ldots \mid e \mid \mid 1 \mid \ldots \mid 9$$

where nfz means not f and not zero,

$$\texttt{nfhdigit} = A \mid \ldots \mid E \mid a \mid \ldots \mid e \mid \mid 0 \mid \ldots \mid 9$$

where nf means not f, and

$$\texttt{nahdigit} = B \mid \ldots \mid E \mid b \mid \ldots \mid b \mid \mid 0 \mid \ldots \mid 9$$

where na means not a. Now we can igore the newline and linefeed as follows (and let's name it μ since it's long):

$$\mu = u \; \texttt{nzhdigit hdigit hdigit hdigit} \mid 0 \; \texttt{(nzhididgt hdigit hdigit} \; \mid$$

$$0 \; \texttt{(nfzhdigit hdigit} \mid 0 \; \texttt{nahdigit} \; \mid (f \mid F) \; \texttt{nfhdigit)))}$$

(notice how we considered the ways of achieving the disallowed characters and then avoided them). Let

$$o = 0 \mid \ldots \mid 7$$

be the regular expression for a single octal digit and

$$t = 1 \mid 2 \mid 3.$$

Thus, an octal escape will be as follows (and we name it ω)

$$\omega = o \mid oo \mid too.$$

Bringing everything together the regular expression for valid backslash escapes are

$$\beta = \backslash(b \mid t \mid n \mid f \mid r \mid \texttt{"} \mid \texttt{'} \mid \backslash \mid \mu \mid \omega)$$

(and we named it β for ease of use going forward). Note that β covers the regular expression necessary for (ii) so we go on to point (iii). Parts of β cover (iii) so we only need to consider the cases of newline and linefeed characters. Using the hint we see that this is easy to achieve since we can represent the newline character using '$\backslash n$' meaning we can do

$$[\hat{}\,\texttt{'}\backslash n\texttt{'},\ \texttt{'}\backslash r\texttt{'}\ ]$$

to avoid the two illegal characters. At this point we have the following regular expression for valid all valid strings

$$\texttt{"}(\beta \mid [\hat{}\,\texttt{'}\backslash n\texttt{'},\ \texttt{'}\backslash r\texttt{'}\ ])^*\texttt{"}$$

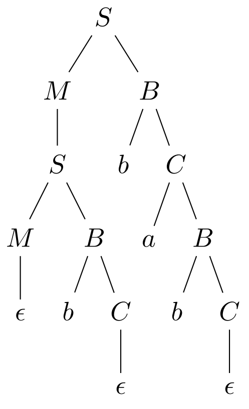
which is almost the correct answer. Since β starts with \and we are not allowed to use a " without a \we almost also avoid those characters if we do not consider escaped characters which gives us the correct solution

$$\texttt{"}(\beta \mid [\hat{}\,\texttt{'}\backslash n\texttt{'},\ \texttt{'}\backslash r\texttt{'}, \backslash, \texttt{"}\ ])^*\texttt{"}.$$

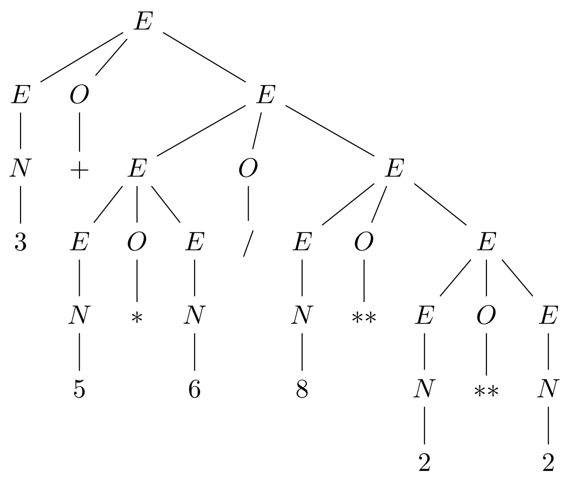
2 HW1 Problem 2

(a) $\mathcal{L}(G)$ = more b's than a's. How do we figure this out? We see that the the start symbol S generates MB. M then generates $S \mid \epsilon$ meaning M can generate another S or just the empty string ϵ and end there. Next, B either generates a b and then goes to rule C or generates an a and two more B's. Either way B generates more b's then a's. Then, C either generates ϵ meaning we have more b's than a's or C generates either bA or aB meaning C generates the same amount of b's and a's. Finally, we consider A where A generates more a's then b's such that $\#(a) = \#(b) + 1$. However, we can get to A only through C which adds one more b ensuring that upon "exiting" A we will have $\#(a) = \#(b)$. Similarly, before "entering" C B generates one more b ensuring that $\#(b) > \#(a)$ upon "exiting" C with an ϵ or A or $\#(a) = \#(b)$ upon "exiting" C with a B where B then again ensures $\#(b) > \#(a)$. Thus, we conclude that the language of the grammar is all strings with b's and a's such that the number of a's is greater than the number of b's and this holds true for all strings derived from B meaning even if we create many S's using M the property will still hold true (for all substrings).

(b) One parse tree is given below.



(c) The unique parse tree is given below.



3 Class 3

- Side effects: The evaluation of an expression influences subsequent computation in some other way besides returning a value.
- Statement: Expressions whose sole purpose is their side effect and whose result value does not matter.
- Imperative languages: Computing by means of side effects.
- Purely functional languages: No side effects. They are referentially transparent (replace function by its return value and get the same behavior).
- L-value: Expressions that denote memory locations.
- R-value: Expressions that denote values stored at memory locations.
- Value model: Same expression can be an l-value or r-value depending on its context.
  - Variable on the left side of an assignment is l-value.
  - Variable on the right side of an assignment is r-value.
  - Example (C/C++):

    ```
    int a = 0;  \\ a is l-value
    a = a + 1;  \\ first a is l-value (refers to a memory location)
                \\ second a is r-value (refers to value stored in a)
    ```

- Reference model: Every variable is an l-value and to get the value at a named location we must dereference.
- Java mixes these. Primitives are value model while Objects are reference model.