
CS 320 Principles of Programming Languages, Spring 2019 Homework 2

Notes and sample solutions

Question 1:

---

a) The set of strings of digits 0, 1, 2, ..., 9 that correspond to multiples of 4 without leading zeros. For example, this language includes "0", "4", "8", "12", "16", ..., "124", ... but does not include "00", "3", "13", "121", "1001", "433", ...

Let's write down some more examples of strings in this language to see if we can spot a pattern:

```
 0   4   8  12  16
20  24  28  32  36
40  44  48  52  56
60  64  68  72  76
80  84  88  92  96
...
```

It is immediately clear that the single digit numbers in this list can be described by the following regular expression:

```
r1 = (0|4|8)
```

For the two digit numbers, we can notice that, if the first digit is even (2|4|6|8), then last digit can be any of (0|4|8) and if the first digit is odd (1|3|5|7|9), then the last digit can be any of (4|6). Thus a suitable regular expression for all of the two digit numbers is:

```
r2 = (2|4|6|8)(0|4|8) | (1|3|5|7|9)(2|6)
     \_____________/   \_____________/
```

Finally, any number with three or more digits that is also a multiple of four must end with a pair of two digits that is a multiple of four, but the digits that come before that can be anything because they are all implicitly multiplied by 100, which is a multiple of four. Thus a suitable regular expression for all multiples with 3 or more digits is:

```
r3 = [1-9]    [0-9]*    ((0|2|4|6|8)(0|4|8) | (1|3|5|7|9)(2|6))
     \___/              \_____________________________________/
```

```
        nonzero   other       last two digits are a multiple of 4
        leading   digits
         digit
```

Any natural number that is a multiple of four must have one, two, or more digits, so it will match one of the three regular expressions above, and hence the set of all strings representing multiples of 4 can be described by a regular expression of the form (r1|r2|r3), which is enough to prove that the language must be regular.

---

b) The set of strings of digits 0, 1, 2, ..., 9 that correspond to nonnegative numbers in the range 0-1234, inclusive. For example, this language includes "0", "3", "98", "587", "1000" and "1234" but not "00", "1235", or "12345".

It is clear that all numbers with 0, 1, 2, or 3 digits are included in the language described above. We can describe the set of all such strings with the following regular expression:

```
   s1 =   [0-9]   |   [1-9][0-9]   |   [1-9][0-9][0-9]
        \_______/   \_____________/   \___________________/
         single       two digit      three digit numbers
         digit         numbers
         numbers
```

Can we describe the remaining numbers (in the range 1000-1234) with a regular expression? Of course, all of these numbers begin with 1. If the second digit is 0|1, then we are looking at a number in the range 1000-1199 and the last two digits can be anything. If the first two digits are "12", then either the third digit is 0|1|2 (numbers in the range 1200-1229) or the third digit is 3, in which case the last digit can only be [0-4]. Thus a suitable regular expression for the numbers in the range 1000-1234 is as follows:

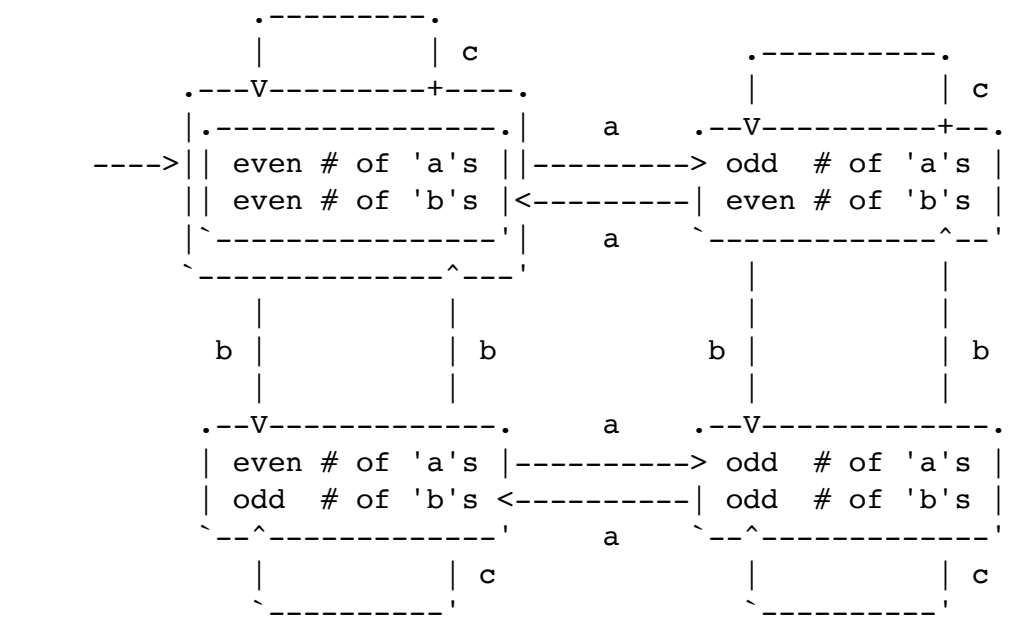
```
   s2 =   1    ( (0|1)[0-9][0-9] | 2(0|1|2)[0-9] | 23[0-4] )
        \___/    \_____________/   \___________/   \_____/
     leading 1       000-199          200-229      230-234
```

The regular expression that would be formed by (s1|s2) will match precisely the strings described in the question corresponding to numbers in the range 0-1234, which is enough to prove that the language is regular.

---

c) The set of strings of characters 'a', 'b', and 'c' that include an even number of 'a's, an even number of 'b's, and any number of 'c's. For example, this language includes "", "aa", "bb", "ccaca", "ababc", ... but not "a", "aaa", "abac", "cab", ...

As we scan through a string containing a sequence of 'a', 'b', and 'c' characters, we can track whether the number of 'a's and 'b's that we have seen so far is even (as it will be before we have seen any characters) or odd. This suggests that we can use a finite automaton with four states:



[Ideally, I would have drawn this by hand with pen and paper, but after the mailing list discussion, I took it as a challenge to draw the diagram in ASCII art :-) Note that you will need to use a fixed width font to view this diagram as intended!]

The key characteristics of this diagram are as follows:

- Every input 'a' character causes an appropriate transition between the left (even # of 'a's) and right (odd # of 'a's) columns.

- Every input 'b' character causes an appropriate transition between the top (even # of 'a's) and bottom (odd # of 'a's) rows.

- Every input 'c' keeps us in the same state as before (the number

of 'a's and 'b's has not changed).

- We can only accept in the top left state, which is precisely
  when the numbers of 'a's and 'b's are both even.

It is clear that this machine recognizes precisely the set of strings described in the question, and so we can conclude that our language is regular (because any language that is recognized by a finite automaton like this must be regular).

Question 2:

---

a) Lists of one or more numbers separated by commas.

We can use the following grammar to describe this language:

```
L -> number
L -> number , L
```

Any derivation using this grammar must begin with some number of steps using the second production, and will terminate immediately as soon as the first production is used. Each of these different derivations will produce a different output string (i.e., the number of comma tokens in each of the resulting sentences/parse trees will be different), so there is no ambiguity in this grammar.

For experimental verification, we can describe the grammar as follows:

```
> parta  = "L -> number ; L -> number , L"
```

Now we can use the Grammar Toolkit to experiment:

```
Main> langSentences 4 parta
1) number
2) number , number
3) number , number , number
4) number , number , number , number

Main> ambigExamples 100 parta
No ambiguity examples found

Main>
```

The first of these two outputs shows that our grammar is generating the expected set of strings, matching the examples in the question;

the second confirms that there are no examples of ambiguity, at
least in the first 100 derivations.

---
b) Bracketed lists of zero or more numbers, separated by commas.

We can describe this language by using the following grammar (for
brevity, I will give most of the remaining examples directly in the
form of Haskell strings that can be used with the Grammar Toolkit):

```
> partb  = "B -> [ ] ; B -> [ L ] ; " ++ parta
```

Note that we reuse the grammar from Part (a) by adding it after the
new productions for B. There is no ambiguity here: we have already
established that the grammar for L, which we are reusing here, does
not allow any ambiguity, and it is clear that the two productions
for B can never match the same input (because L can never be empty).

We can replicate the examples in the question and run a quick test
to check for (the lack of) examples of ambiguity:

```
    Main> grammar partb
    1) B -> [ ]
    2) B -> [ L ]
    3) L -> number
    4) L -> number , L

    Main> langSentences 4 partb
    1) [ ]
    2) [ number ]
    3) [ number , number ]
    4) [ number , number , number ]

    Main> ambigExamples 100 partb
    No ambiguity examples found

    Main>
```

---
c) Lists of zero or more numbers, each followed by a period symbol.

```
> partc  = "S -> ; S -> number . S"
```

Again, we can see that this grammar is unambiguous because there
is only one way to derive a string with some given number N of
"number" tokens: use the second production N times and then finish

up with the first production. Again, we replicate the examples in the question using the Grammar Toolkit:

```
    Main> langSentences 4 partc
    1)
    2) number .
    3) number . number .
    4) number . number . number .

    Main> ambigExamples 100 partc
    No ambiguity examples found

    Main>
```

---

d) Palindromes using terminal symbols a, b, and c.

Every palindrome will be either empty, a single symbol, or a string with at least two symbols in where the first and the last token are the same, and the rest of the input is again a palindrome. We can capture this

```
> partd  = "P -> ; " ++                                -- length==0
>          "P -> a ; P -> b ; P -> c ; " ++            -- length==1
>          "P -> a P a ; P -> b P b ; P -> c P c ;"     -- length>=2
```

Note that it is only possible to construct this as a context free grammar because the set of terminal symbols, {a, b, c}, is finite. This means that we can write a separate production for each possible terminal in the last group of productions above (i.e., the ones for length>=2).

As in each of the previous examples, there is a unique derivation for any given palindrome in this language. If the input string that we want to generate has 0 or 1 symbols, then the derivation contains a single step using the appropriate production from the first two lines above. Alternatively, if the palindrome that we want to match is longer, then the first (and last) symbol will uniquely determine the first production in its derivation, and then we repeat the same argument for the inner part of the palindrome to complete the derivation. (In more formal terms, you can think of this argument as a proof by induction or a definition by recursion; but we're not expecting that level of rigor here!)

Our usual verification steps using the Grammar Toolkit follow. This time, however, we need to generate more sample sentences because

otherwise we wouldn't see enough examples to be sure that all of the symbols a, b, and c are used, and that we can generate palindromes of varying lengths (to save space, however, I have reformatted the output into columns):

```
Main> langSentences 32 partd
1)           9) b b        17) a a a a       25) a c c a
2) a         10) b a b     18) a a a a a     26) a c a c a
3) b         11) b b b     19) a a b a a     27) a c b c a
4) c         12) b c b     20) a a c a a     28) a c c c a
5) a a       13) c c       21) a b b a       29) b a a b
6) a a a     14) c a c     22) a b a b a     30) b a a a b
7) a b a     15) c b c     23) a b b b a     31) b a b a b
8) a c a     16) c c c     24) a b c b a     32) b a c a b

Main> ambigExamples 100 partd
No ambiguity examples found

Main>
```

This time, the output strings do not match all of the examples given in the question. But it is easy to see that all of these strings are palindromes, including all of the examples of length 0, 1, 2, and 3, and an emerging pattern for longer examples.

---

e) Strings of decimal digits that correspond to multiples of three.

[The following documents two different solutions to this problem, and does so in more depth than is expected in student solutions; we include both here for the benefit of students who read these solutions.]

We saw this example previously in the form of a DFA in Week 2, Slides 25-29. That DFA has three states, tracking the sum, modulo 3, of the digits that have been seen so far in a given input string. We will refer to those states here as S0, S1, and S2, where the number after the S is the sum of the digits modulo 3. These three states are connected by an appropriate set of transitions. For example, an input digit 3 does not change the sum modulo 3, and results in a loop at each state. At the same time, a 1 digit will cause a transition from S0 to S1, or from S1 to S2, or from S2 to S0.

We can capture the full DFA in the following grammar using one nonterminal for each of the three states:

```
> parte1 = "S0 -> 0 S0 ; S1 -> 0 S1 ; S2 -> 0 S2 ; " ++
>          "S0 -> 1 S1 ; S1 -> 1 S2 ; S2 -> 1 S0 ; " ++
>          "S0 -> 2 S2 ; S1 -> 2 S0 ; S2 -> 2 S1 ; " ++
>          "S0 -> 3 S0 ; S1 -> 3 S1 ; S2 -> 3 S2 ; " ++
>          "S0 -> 4 S1 ; S1 -> 4 S2 ; S2 -> 4 S0 ; " ++
>          "S0 -> 5 S2 ; S1 -> 5 S0 ; S2 -> 5 S1 ; " ++
>          "S0 -> 6 S0 ; S1 -> 6 S1 ; S2 -> 6 S2 ; " ++
>          "S0 -> 7 S1 ; S1 -> 7 S2 ; S2 -> 7 S0 ; " ++
>          "S0 -> 8 S2 ; S1 -> 8 S0 ; S2 -> 8 S1 ; " ++
>          "S0 -> 9 S0 ; S1 -> 9 S1 ; S2 -> 9 S2 ; " ++
>          "S0 -> "
```

The final production here, "S0 ->", reflects the fact that S0 is an accept state, indicating that the sum of the digits seen so far is a multiple of three.

We can use the Grammar Toolkit again to generate some sample sentences from this (complicated) grammar (with reformatting into multiple columns again so that we can include a good collection of examples in a relatively short space:

```
Main> langSentences 40 parte1
1)            9) 0 9        17) 3 3       25) 5 7       33) 8 1
2) 0          10) 1 2       18) 3 6       26) 6 0       34) 8 4
3) 3          11) 1 5       19) 3 9       27) 6 3       35) 8 7
4) 6          12) 1 8       20) 4 2       28) 6 6       36) 9 0
5) 9          13) 2 1       21) 4 5       29) 6 9       37) 9 3
6) 0 0        14) 2 4       22) 4 8       30) 7 2       38) 9 6
7) 0 3        15) 2 7       23) 5 1       31) 7 5       39) 9 9
8) 0 6        16) 3 0       24) 5 4       32) 7 8       40) 0 0 0

Main> ambigExamples 1000 parte1
No ambiguity examples found

Main>
```

A visual inspection of the output here confirms that we are generating strings corresponding to multiples of three, starting with the one string of length 0, then the strings of length 2 (including some with leading zeros), and so on.

We can use langSentence N parte1 with specific larger values of N to probe for other examples, and confirm that each of those also corresponds to a multi digit number that is a multiple of three. In particular, the question includes "1 6 2" as an example that should be derivable from our grammar. Based on the patterns we see in the

output above, with the three digit numbers starting at Line 40, we can make an educated guess about where "1 6 2" will appear in the full list of sentences:

```
    Main> langSentence (40 + (162 `div` 3)) parte1
    1 6 2

    Main>
```

Intuitively, we can see that the grammar above is unambiguous because, for any given nonterminal, and any single digit, there is a single matching production. Given the complexity of the grammar, however, it is also possible that we might have slipped up and made a mistake in the way the grammar was written. As such, using ambigExamples with a larger number of samples is useful, not just in demonstrating an absence of ambiguity, but also in sanity checking the overall definition.

An alternative, and slightly more compact way to specify the grammar above is to introduce new nonterminals Add0, Add1, and Add2, each of which represents a specific group of digits:

```
> parte2 = "S0 -> Add0 S0 ; S0 -> Add1 S1 ; S0 -> Add2 S2 ;" ++
>          "S1 -> Add0 S1 ; S1 -> Add1 S2 ; S1 -> Add2 S0 ;" ++
>          "S2 -> Add0 S2 ; S2 -> Add1 S0 ; S2 -> Add2 S1 ;" ++
>          "Add0 -> 0 ; Add0 -> 3 ; Add0 -> 6 ; Add0 -> 9 ;" ++
>          "Add1 -> 1 ; Add1 -> 4 ; Add1 -> 7 ;" ++
>          "Add2 -> 2 ; Add2 -> 5 ; Add2 -> 8 ;" ++
>          "S0 -> "
```

Of course, we can test this grammar in the usual way:

```
    Main> langSentences 40 parte2
    1)              9) 9        17) 0 9     25) 3 9     33) 6 9
    2) 0            10) 0 0     18) 3 0     26) 6 0     34) 9 0
    3) 3            11) 0 3     19) 3 3     27) 6 3     35) 9 3
    4) 6            12) 0 6     20) 3 6     28) 6 6     36) 9 6
    5) 9            13) 0 9     21) 3 9     29) 6 9     37) 9 9
    6) 0            14) 0 0     22) 3 0     30) 6 0     38) 9 0
    7) 3            15) 0 3     23) 3 3     31) 6 3     39) 9 3
    8) 6            16) 0 6     24) 3 6     32) 6 6     40) 9 6

    Main> ambigExamples 1000 parte2
    No ambiguity examples found

    Main>
```

This seems to be producing output strings corresponding to multiples of three, and our attempt to find examples of ambiguity comes up empty. But it might be a little disconcerting to see that most of the strings in this list appear two times: could this be an indication that there is some hidden source of ambiguity?

Using sentences 100 parte2, we can generate a list of the first 100 strings that can be derived in this grammar (strings including both terminals and nonterminals), and then we can scan that list to find that the two derivations for the string "0" are at positions 29 and 74. Having figured that out, we can take a look at the actual derivations for these two cases:

```
    Main> treeDerivation 29 parte2
    S0   ==>      S0     ==>      S0     ==>      S0
                   |               |               |
                 .---.           .---.           .---.
                 |   |           |   |           |   |
               Add0 S0         Add0 S0         Add0 S0
                                 |               |   |
                                 0               0

    Main> treeDerivation 74 parte2
    S0   ==>      S0     ==>      S0     ==>      S0
                   |               |               |
                 .---.           .---.           .---.
                 |   |           |   |           |   |
               Add0 S0         Add0 S0         Add0 S0
                                     |           |   |
                                                 0

    Main>
```

As the diagrams here demonstrate, the two derivations end up with the same parse tree structure, so there is no ambiguity here. But there is a familiar difference between the two derivations: the first one (at position 29) is a left-most derivation, while the second (at position 74) is a right-most derivation. Mystery solved, we can move on to the next example!

---

f) Context free grammars in the syntax used by the Grammar Toolkit.

A very close variant of the grammar that we need here was shown in

Week 2, Slide 59! The only difference is that we do not allow G to be empty in the following, a change that is handled by replacing the "G ->" production from the slide with "G -> P":

```
> partf  = "G -> P ; G -> P | G ; " ++
>          "P -> n => W ; " ++
>          "W -> ; W -> n W ; W -> t W"
```

Of course, it is also possible to construct this grammar from first principles. The productions for W do not introduce any ambiguity (as in several of the previous cases, we can see that there is a unique derivation for any string of n and t symbols). And there is only one production for P, and hence no opportunity for ambiguity there. Finally, we can recognize the productions for G as using exactly the same pattern that we saw in Part (a) above. Or we can think of this as a familiar construction for a grammar that defines the | symbol as a right grouping infix operator. Either way, we can see that the grammar is free from ambiguity.

```
Main> langSentences 16 partf
1) n =>          5) n => n t     9) n => n n t   13) n => t n t
2) n => n        6) n => t n     10) n => n t n  14) n => t t n
3) n => t        7) n => t t     11) n => n t t  15) n => t t t
4) n => n n      8) n => n n n   12) n => t n n  16) n => | n =>

Main> ambigExamples 1000 partf
No ambiguity examples found

Main>
```

The first five examples given in the question show up in the list of sentences given above at Positions 1, 2, 3, 14, and 16, respectively. With a little extra experimentation, we can also find a derivation for the sixth example given in the question:

```
Main> langSentence 149 partf
n => n | n => t

Main>
```

These results provide good evidence that our grammar is describing the language as expected.

Question 3:

The starting point for this question is the (ambiguous) grammar that is suggested by the table on Slide 47. (This grammar also appears explicitly, albeit without the case for parentheses, on Slide 59.)

```
R -> c
R -> epsilon
R -> R R
R -> R | R
R -> R *
R -> ( R )
```

a) i) alternatives have lower precedence than sequencing:
For example, "a|bc" should be parsed in the same way as "a|(bc)" so that the sequencing operation takes precedence over the alternatives operation.

ii) sequencing has lower precedence than repetition:
For example, "ab*" should be parse in the same way as "a(b*)" so that the repetition operation takes precedence over the use of sequencing.

iii) repetition has lower precedence than parentheses:
This property is, in fact, already guaranteed by the grammar. there is no ambiguity in an expression that juxtaposes the repetition operator with parentheses; in an example like "(a|b)*", the repetition operation is applied to the full regular expression inside the parentheses.

iv) alternatives group to the right:
For example, "a|b|c" should be parsed in the same way (i.e., should have the same abstract syntax tree) as "a|(b|c)" where the rightmost "|" operator takes precedence over the operator on the left. [Aside: of course, given what we know about the semantics of the alternatives operation, this detail of syntax is not likely to have a significant impact in practice.]

v) sequencing groups to the left:
For example "abc" should be parsed in the same way (i.e., should have the same abstract syntax tree) as "(ab)c" where the use of sequencing on the left has precedence over the use on the right. [Again, we would not expect this to make a difference with the standard semantics for sequencing, but we would be able to observe a difference if we looked at abstract syntax tree structures.]

b) Following the patterns presented in the lecture, we can capture the required precedences ((i) and (ii)) by writing a grammar that describes every regular expression R as:

```
a list of sequences (separated by "|"s);
 each of which (S) is a list of repetitions;
  each of which (P) is an atom followed by 0 or more "*"s;
   each of which (A) is either a single character c, an
    epsilon, or a parenthesized regular expression.
```

The start symbol for our grammar is R, which describes a regular expression as a sequence of "S"s, separated by "|" terminals. The right recursive production in the first line is sufficient to ensure that "|" group to the right, satisfying (iv) (Slide 82):

```
R -> S | R
R -> S
```

Every S is a sequence of repetitions, P, juxtaposed next to one another without an explicit operator symbol. The use of a left recursive production ensures that sequencing groups to the left, as required for property (v) (Slide 82):

```
S -> S P
S -> P
```

Every repetition P is an atom A followed by zero or more "*"s:

```
P -> A
P -> P *
```

And every atom A is either a single character, the epsilon symbol, or a parenthesized R:

```
A -> c
A -> epsilon
A -> ( R )
```

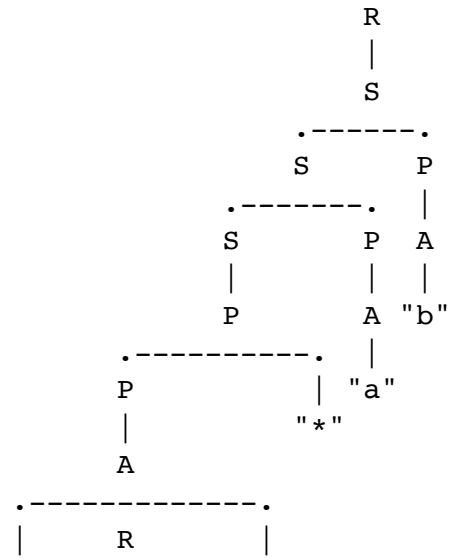
As noted at the relevant points above, the methods that we have used to construct this grammar ensure that it satisfies (i), (ii), (iv), and (v), and the syntax for parentheses ensures that (iii) is also satisfied. The combination of the above ensures that every regular expression has a unique parse tree, and indicates that our grammar is unambiguous.

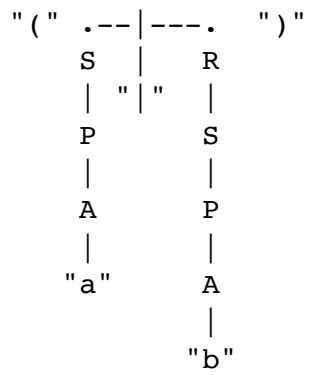
c) A full derivation for "(a|b)*ab" using the grammar above requires 19 steps. (Given the relative complexity of this grammar, it is better to construct this derivation by hand than to try using the Grammar Toolkit to build it.)

```
 1)  R ==> S                 (using R -> S)
 2)    ==> S P               (using S -> S P)
 3)    ==> S P P             (using S -> S P)
 4)    ==> P P P             (using S -> P)
 5)    ==> P * P P           (using P -> P *)
 6)    ==> A * P P           (using P -> A)
 7)    ==> ( R ) * P P       (using A -> ( R ) )
 8)    ==> ( S | R ) * P P   (using R -> S | R)
 9)    ==> ( P | R ) * P P   (using S -> P)
10)    ==> ( A | R ) * P P   (using P -> A)
11)    ==> ( a | R ) * P P   (using A -> c)
12)    ==> ( a | S ) * P P   (using R -> S)
13)    ==> ( a | P ) * P P   (using S -> P)
14)    ==> ( a | A ) * P P   (using P -> A)
15)    ==> ( a | b ) * P P   (using A -> c)
16)    ==> ( a | b ) * A P   (using P -> A)
17)    ==> ( a | b ) * a P   (using A -> c)
18)    ==> ( a | b ) * a A   (using P -> A)
19)    ==> ( a | b ) * a b   (Using A -> c)
```

A lot of the derivation steps here are just replacing one nonterminal with another as we descend through the levels of the grammar (R -> S ; S -> P ; P -> A)

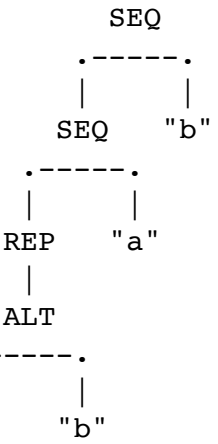
The corresponding parse tree (also hand drawn) is as follows:





Some indications that this is correct include:
1) The root is labeled with R, the start symbol for the grammar.
2) The terminals at the leaves match our input "(a|b)*ab".
3) There are 19 interior nodes, each corresponding to a production
   and to a step in the preceding derivation.

Many of the details in the parse tree can be omitted when building the abstract syntax tree for this regular expression:



In this diagram, we use SEQ and ALT to represent sequencing and alternative nodes, respectively, with two children each; REP to represent repetition nodes with one child; and one character strings for the nodes that represent single characters. The overall tree structure of these two trees is similar, but we do not include all the symbols of the input (such as "(", "|", ")", and "*") in the parse tree, and we do not need to show all the nonterminals that were used in constructing the derivation shown previously.
