4.1 Basic results from automata theory tell us that the language $L = a^n b^n c^n = \epsilon$, abc, aabbcc, aaabbbccc, ... is not context free. It can be captured, however, using an attribute grammar. Give an underlying CFG and a set of attribute rules that associates a Boolean attribute ok with the root R of each

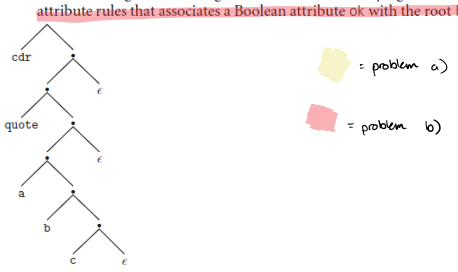

Figure 4.16 Natural syntax tree for the Lisp expression `(cdr '(a b c))`.

a) CFG

S → ABC | ε
A → a A | ε
B → b B | ε
C → c C | ε

b) Attribute rules

To make sure a, b, c symbols are equal we need to introduce something to count symbols
↳ each non terminal A B C will carry an attribute to count number of occurrences
↳ The root S will have boolean ok to indicate a = b = c

1. Base case:
   For S → ε, S.ok = true
2. Recursive case:
   For A → a A', A.count = 1 + A'.count
   For B → b B', B.count = 1 + B'.count
   For C → c C', C.count = 1 + C'.count
3. Validation
   at the end counting ABC we set S.ok = (A.count = B.count = C.count)

4.5 Lisp has the unusual property that its programs take the form of parenthesized lists. The natural syntax tree for a Lisp program is thus a tree of binary cells (known in Lisp as cons cells), where the first child represents the first element of the list and the second child represents the rest of the list. The syntax tree for `(cdr '(a b c))` appears in Figure 4.16. (The notation 'L is syntactic sugar for `(quote L)`.)

Extend the CFG of Exercise 2.18 to create an attribute grammar that will build such trees. When a parse tree has been fully decorated, the root should have an attribute v that refers to the syntax tree. You may assume that each atom has a synthesized attribute v that refers to a syntax tree node that holds information from the scanner. In your semantic functions, you may assume the availability of a cons function that takes two references as arguments and returns a reference to a new cons cell containing those references.

a) Extend the context free grammar

2.18 CFG LL(1)

P → E $$
E → atom
  → 'E
  → (E Es)
Es → E Es
   →

⇒

P → E $$	P.v = E.v
E → atom	E.v = create Atom Node(atom)
→ 'E	E.v = create Quote Node(E.v)
→ (E Es)	E.v = cons(E.v, Es.v)
Es → E Es	Es.v = cons(E.v, Es.v)
→	Es.v = create Nil Node()

This allows us to represent the structure as Lisp cons cells

4.13 Consider the following attribute grammar for variable declarations, based on the CFG of Exercise 2.11:

decl ⟶ ID *decl_tail*
  ▷ decl.t := decl_tail.t
  ▷ decl_tail.in_tab := insert (decl.in_tab, ID.n, decl_tail.t)
  ▷ decl.out_tab := decl_tail.out_tab
decl_tail ⟶ , *decl*
  ▷ decl_tail.t := decl.t
  ▷ decl.in_tab := decl_tail.in_tab
  ▷ decl_tail.out_tab := decl.out_tab
decl_tail ⟶ : ID ;
  ▷ decl_tail.t := ID.n
  ▷ decl_tail.out_tab := decl_tail.in_tab

Show a parse tree for the string `A, B : C;`. Then, using arrows and textual description, specify the attribute flow required to fully decorate the tree. (Hint: Note that the grammar is *not* L-attributed.)

A, B : C;