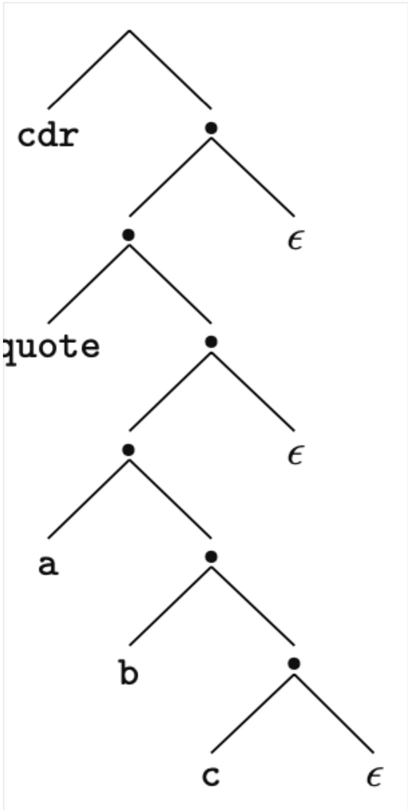
HW4

4.1, 4.5 and 4.13

4.1

Basic results from automata theory tell us that the language L = a^n b^n c^n = ε, 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



parse tree, such that R.ok = true if and only if the string corresponding to the fringe of the tree is in *L*.

$L \to A\ B\ C$

$\triangleright$ $L.valid = A.cnt == B.cnt == C.cnt$

$A \to A a$

$\triangleright$ $A.cnt = A.cnt + 1$

$\to \epsilon$

$\triangleright$ $A.cnt = 0$

$B \to B b$

$\triangleright$ $B.cnt = B.cnt + 1$

$\to \epsilon$

$\triangleright$ $B.cnt = 0$

$C \to C c$

$\triangleright$ $C.cnt + 1$

$\to \epsilon$

$\triangleright$ $C.cnt = 0$

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

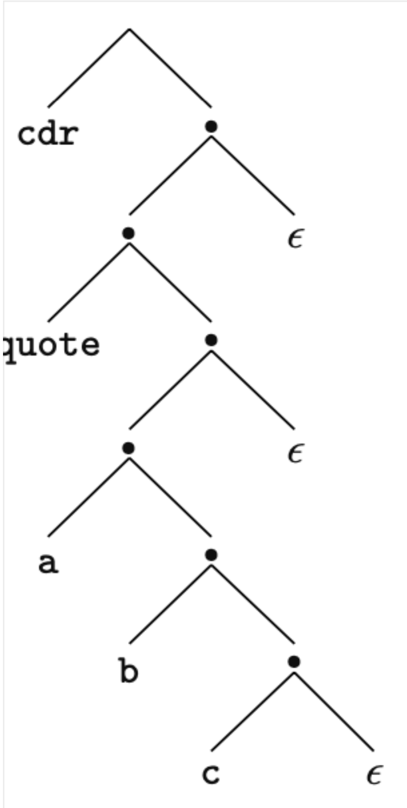


Figure 4.16. (The notation ,L is syntactic sugar for (quote L).)
Extend the CFG of

$$
\begin{array}{lll}
P & \longrightarrow & E\ \$\$ \\
E & \longrightarrow & \texttt{atom} \\
 & \longrightarrow & \texttt{'}\ E \\
 & \longrightarrow & \texttt{(}\ E\ Es\ \texttt{)} \\
Es & \longrightarrow & E\ Es \\
 & \longrightarrow &
\end{array}
$$

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.

Answer:

P -> E$$

=> P.v := E.v

E -> atom

=> E.v := atom.v

Es -> ' E2

=> Es.v := cons(‘ , E2.v))

E1 -> (E2 Es)

=> E1.v := cons(E2.V, Es.v)

Es1 -> E Es2

=> Es1.v := cons(E.v, Es2.v)

Es -> ε

=> Es.v := null

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

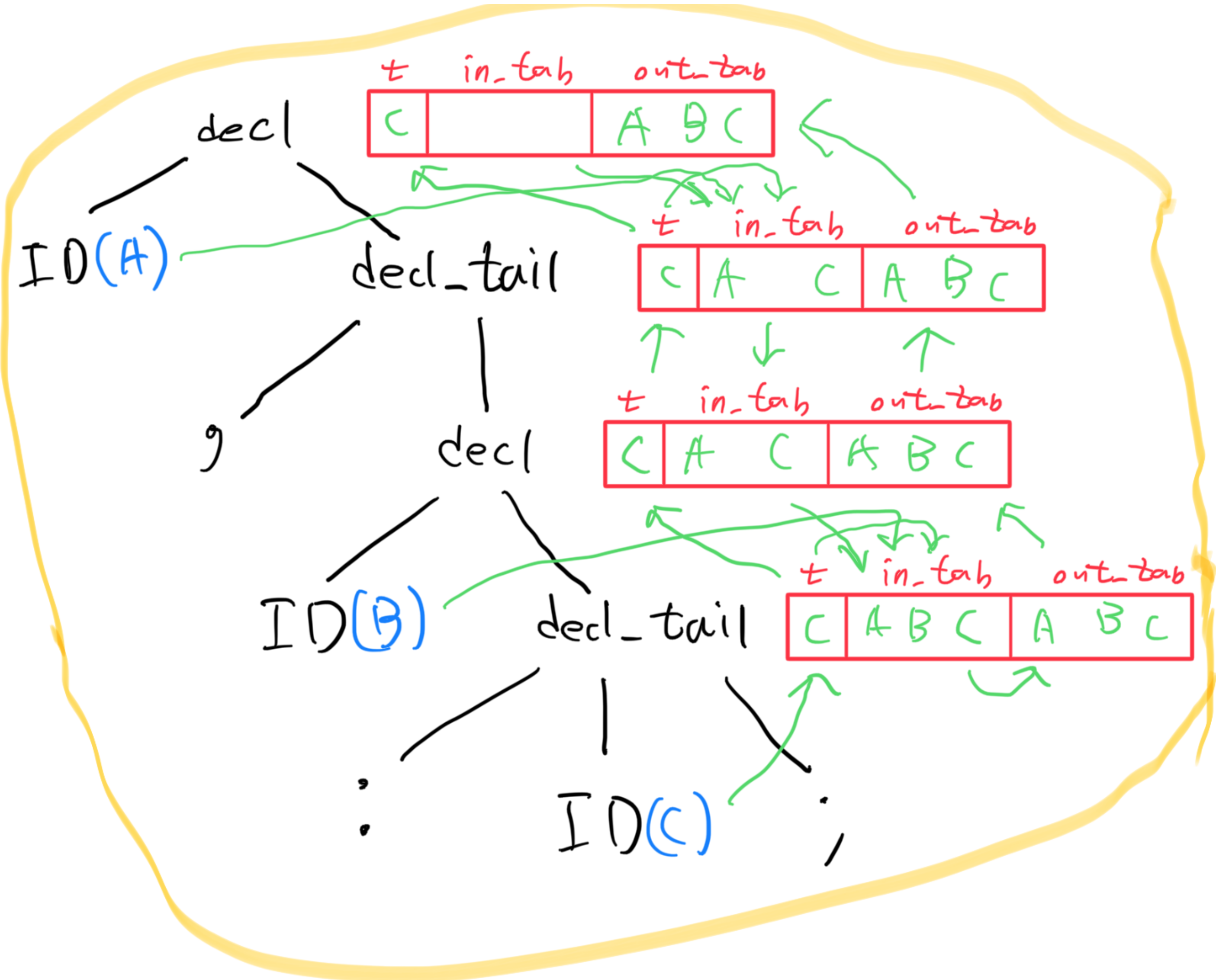
decl_tail.out_tab := decl.out_tab

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.)


decl
ID(A)
decl_tail
,
decl
ID(B)
decl_tail
:
ID(C)
;
t
in_tab
out_tab
C
A B C
C
A C
A B C
C
A C
A B C
C
A B C
A B C