Building a parse tree

Let's say we have a grammar $S \rightarrow aSx \mid bSx \mid \lambda$, which has First(S) = {a,b} and Follow(S) = {x,$}. A recursive descent parsing function can be written following the pattern seen in class, as follows.

```
def parseS(toks):
    tok = toks.next()
    if tok in ('a', 'b'):
        toks.match(tok)
        parseS(toks)
        toks.match('x')
    elif (tok == None) or (tok == 'x'):
        pass
    else:
        raise Exception
```

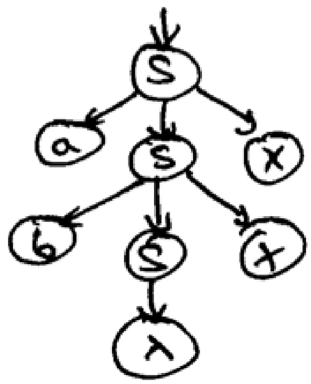
In an actual computer application, like a compiler, the application is likely going to want to do more than recognize inputs. This usually means building a parse tree in memory and returning a reference to its root. The following code has each call to parseS return a reference to a tree node. Study it carefully.

```
# A tree node is a piece of data along with an optional list of children.
# If the _children field is None it indicates the node is a leaf
class node:
    def __init__(self, data):
        self.data = data
        self.children = None

    def add_child(self, child):
        if self.children == None:
            self.children = []
        self.children.append(child)

    def is_leaf(self):
        return self.children == None

def parseS(toks):
    tok = toks.next()
    # Create node to be returned and label it "S"
    rval = node('S')
    if tok in ('a', 'b'):
        # Match tok and add first child, a leaf node labeled tok
        toks.match(tok)
        rval.add_child(node(tok))
        # parseS and have resulting subtree be our second child
        rval.add_child(parseS(toks))
        # Match "x" and add third child, a leaf node labeled "x"
        toks.match('x')
        rval.add_child(node('x'))
    elif (tok == None) or (tok == 'x'):
        # S is nullable, and $ and x both indicate S -> lambda
        # Make child be a leaf node labeled "" (empty string)
        rval.add_child(node(''))
    else:
        # Unexpected token, so throw an exception
        raise Exception
    return rval
```

In every case, in this example, if parseS is successful, it returns a reference to a node labelled "S". Depending on which parsing branch the code takes, the S node will have 1 or 3 children. it is parseS's responsibility to create a node for itself and one for each of its terminal children. Calls to further parsing functions required by the production return a reference to the node they create. ParseS embeds all these references into its own node and returns it.

For example, in the case of $S \to aSx$, the S node gets 3 children, one for a, one for S, and one for x. The function creates a node for a and x and they are given labels "a" and "x". The middle child is the reference returned by the recursive call to parseS. If, on the other hand, parseS sees that the next token is end-of-input or "x", then $S \to \lambda$ is the right production to use, and only a child node labeled as an empty string is created for the parse tree.

If this parseS is called with toks containing "abxx", it builds the following tree and returns a reference to the root. The initial call to parseS creates the topmost S, creates the a node, adds it as a child, calls parseS, when that call is done and returns a reference, it's added as S's second child, then the x node is created and added as a third child. Something similar happens when the second parseS is called. When the third parseS is called, "x" is the next token, so no token is consumed and we create a node with an empty string label.



Walking a tree

When given a tree, you often want to visit each of the nodes. This is called "traversing the tree" or sometimes called "walking the tree".

The idea is simple. To walk a tree you begin at the root and recursively walk each of the subtrees rooted at each child. So, if the root is Node R and it has two children Node A and Node B, then to walk the tree beginning at Node R, you walk the tree beginning at Node A and when that is done you walk the tree beginning at Node B. Since you started at Node R, you have now visited every node in the tree. In pseudocode:

```
walk(noderef):
    // Do something with noderef's data if you want
    for child = eachof noderef's children
        walk(child)
    // Do something with noderef's data if you want
```

If you want to do something special at each leaf, you can detect that a node is a leaf by seeing that it has no children list. For example, here is a tree walker that works on a tree built of nodes and prints the contents of each leaf.

```
def print_leaves(tree_node):
    if tree_node.is_leaf():
        print(str(tree_node.data))
    else:
        for child in tree_node.children:
            print_leaves(child)
```