Code Jam 2009 - Round 1B

Analysis: Decision Tree

The hard part here was parsing the tree. An easy way to do this is by using a technique called *recursive descent*. The tree grammar is defined recursively, so it makes sense to parse it recursively, too. Here is a solution in Python.

```
import re
import sys
inp = sys.stdin

tokens = None
ti = -1

def ReadInts():
  """Reads several space-separated integers on a line.
  """
  return tuple(map(int, inp.readline().strip().split(" ")))

def NextToken():
  """Consumes the next token from 'tokens'.
  """
  global ti
  assert 0 <= ti < len(tokens)
  ti += 1
  return tokens[ti - 1]

def ParseNode():
  """Parses from 'tokens' and returns a tree node.
  """
  global ti
  assert NextToken() == "("
  node = {"weight": float(NextToken())}
  tok = NextToken()
  if tok == ")":
    return node
  node["feature"] = tok
  node[True] = ParseNode()
  node[False] = ParseNode()
  assert NextToken() == ")"
  return node

def ParseTree(s):
  """Initializes 'tokens' and 'ti' and parses a tree.
  """
  global tokens
  global ti
  s = re.compile(r"\(").sub(" ( ", s)
  s = re.compile(r"\)").sub(" ) ", s)
  s = re.compile(r"[ \n]+").sub(" ", " %s " % s)
  tokens = s[1:-1].split(" ")
  ti = 0
  return ParseNode()
```

```
def Evaluate(tree, features):
  ans = tree["weight"]
  if "feature" in tree:
    ans *= Evaluate(tree[tree["feature"] in features], features)
  return ans

if __name__ == "__main__":
  N = ReadInts()[0]
  for prob in xrange(1, N + 1):
    n_lines = ReadInts()[0]
    lines = [inp.readline() for _ in xrange(n_lines)]
    tree = ParseTree(" ".join(lines))
    n_queries = ReadInts()[0]
    print "Case #%d:" % prob
    for _ in xrange(n_queries):
      features = set(inp.readline().strip().split(" ")[2:])
      print "%.7f" % Evaluate(tree, features)
```

For each test case, we read the 'n_lines' lines containing the tree definition, we glue them together using spaces and pass the resulting string to ParseTree().

In ParseTree(), we do some "massaging" to make the string easier to parse. First, we put spaces around each parenthesis by using two simple regular expressions. Next, we replace each sequence of whitespace characters by a single space and make sure there is always exactly one space character at the beginning and the end of the input. Finally, we strip off the leading and trailing spaces and split the rest into tokens.

The ParseNode() function does the rest. It uses the NextToken() function to read one token at a time from the 'tokens' list and returns a simple dictionary representation of a tree node.

Once we have the tree as a dictionary, we then use Evaluate() to do a tree traversal from the root to a leaf and compute the answer for each input animal.

Using the parsers built into dynamic languages

A number of contestants have noticed that there is an even easier way to parse the tree. Most dynamic, interpreted languages give you access to their built-in parser, and by manipulating the input a little bit, it is possible to use make the language's interpreter do the parsing for you! This means using "eval" in JavaScript or Python, or "read" in Lisp. Check out some of the shortest solutions at 1b-a.pastebin.com/d4631e678.