Homework: Huffman Encoding Trees

Efficient encoding of strings as ones and zeros (bits).

```
A 0     C 1010   E 1100   G 1110
B 100   D 1011   F 1101   H 1111
```

Decoding a sequence of bits:

```
1 0 0 0 1 0 1 0
```

Logo Interpreter Architecture

Logo words are represented as Python strings
Logo sentences are represented as Python lists
The Parser creates nested sentences, but does not build full expression trees for nested call expressions

Tracking Positions in Lines

A line is used up as it is evaluated
A Buffer instance tracks how much of a line has been used up.

```
>>> buf = Buffer(['show', '2'])
>>> buf.current
'show'
>>> print(buf)
[ 'show', '2' ]
>>> buf.pop()
'show'
>>> print(buf)
[ 'show' ]
>>> buf.pop()
'2'
```
**Logo Evaluation**

The `logo_eval` function dispatches on expression form:

- **A primitive expression** is a word that can be interpreted as a number, True, or False. Primitives are self-evaluating.
- A **variable** is looked up in the current environment.
- A **procedure definition** creates a new user-defined procedure.
- A **quoted expression** evaluates to the text of the quotation, which is a string without the preceding quote. Sentences are quoted and evaluate to themselves.
- A **call expression** is evaluated with `apply_procedure`.

```python
def logo_eval(line, env):
    """Evaluate the first expression in a line."""
    token = line.pop()
    if isprimitive(token):
        return token
    elif isvariable(token):
        ...
```

**Procedures**

```python
class Procedure():
    def __init__(self, name, arg_count, body, isprimitive=False, needs_env=False, formal_params=None):
        self.name = name
        self.arg_count = arg_count
        self.body = body
        self.isprimitive = isprimitive
        self.needs_env = needs_env
        self.formal_params = formal_params

def logo_apply(proc, args):
    """Apply a Logo procedure to a list of arguments."""
    if proc.isprimitive:
        return proc.body(*args)
    else:
        """Apply a user-defined procedure""
```

**Logo Interpreter**

**Eval/Apply in Lisp 1.5**

```lisp
apply[n;x;a] = [atom[n] = [eq[n;CAR] - car[x];
    eq[n;CDR] = cdr[x];
    eq[n;CONS] = cons[car[x];cdr[x]];]
    eq[n;ATOM] = atom[car[x]];]
    eq[n;EQ] = eq[car[x];cdr[x]];]
    T = apply[eval[n[a];a]];]
    eq[car[n];LAMBDA] = eval[caddr[n];pairlis[caddr[n];x;a]];]
    eq[car[n];LABEL] = apply[caddr[n];x;cons[cons[caddr[n];
        caddr[n]];a]];]

eval[e;a] = [atom[e] = cdr[assoc[e;a]];]
atom[car[e]] = [eq[car[e];QUOTE] = cdr[e];
    eq[car[e];COND] = evcon[cdr[e];a];
    T = apply[car[e];eval[cdr[e];a]];]
    T = apply[car[e];eval[cdr[e];a]];]
```

**Eval/Apply in Logo**

```python
apply[n;x;a] = [atom[n] = [eq[n;CAR] - car[x];
    eq[n;CDR] = cdr[x];
    eq[n;CONS] = cons[car[x];cdr[x]];]
    eq[n;ATOM] = atom[car[x]];]
    eq[n;EQ] = eq[car[x];cdr[x]];]
    T = apply[eval[n[a];a]];]
    eq[car[n];LAMBDA] = eval[caddr[n];pairlis[caddr[n];x;a]];]
    eq[car[n];LABEL] = apply[caddr[n];x;cons[cons[caddr[n];
        caddr[n]];a]];]

eval[e;a] = [atom[e] = cdr[assoc[e;a]];]
atom[car[e]] = [eq[car[e];QUOTE] = cdr[e];
    eq[car[e];COND] = evcon[cdr[e];a];
    T = apply[car[e];eval[cdr[e];a]];]
    T = apply[car[e];eval[cdr[e];a]];]```