

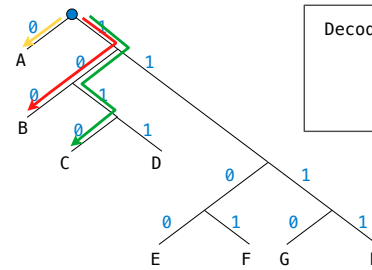
61A Lecture 29

Monday, November 7

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
 B A C

Logo Refresher

Data types: Words and sentences (immutable sequences)

Syntactic forms: Call expressions, literals, and to-statements

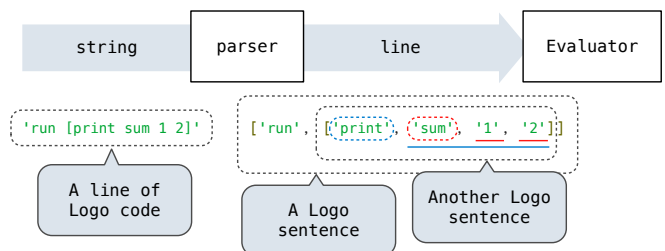
```
? print sum 10 difference 7 3
14

? run [print sum 1 2]
3

? to double :x
> output sum :x :x
> end

? print double 4
8
```

Logo Interpreter Architecture



Logo words are represented as Python strings

Logo sentences are represented as Python lists

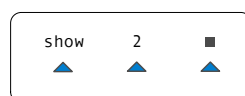
The Parser creates nested sentences, but **does not** build full expression trees for nested call expressions

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 >> 2 ]
>>> buf.pop()
'2'
```



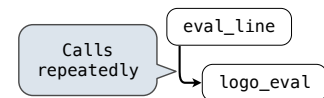
Demo

Evaluating Lines

Evaluating a line of Logo involves evaluating each expression

Evaluate a line

Evaluate the next expression



```
? print 1 print 2
1
2
```

logo_eval	Argument	Effect
first call	[>> print, 1, print, 2]	prints 1, returns None
second call	[print, 1 >> print, 2]	prints 2, returns None

Logo Evaluation

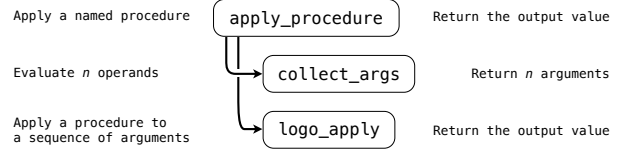
The `logo_eval` function dispatches on expression form:

- 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`.

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

The expression form can be inferred from the first token

Evaluating Call Expressions



```
[ print >> 2 ]
Popped by logo_eval
```

```
[ print, 2 >> ]
Popped by logo_eval also
(recursive call)
```

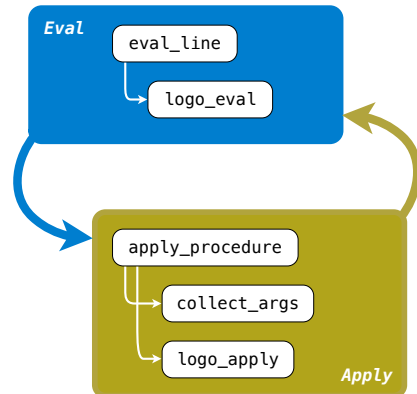
1. Collect 1 argument via `logo_eval` (`collect_args`)
2. Apply `print` procedure to the argument '2' (`logo_apply`)

Procedures

```
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

```
apply[fn;x;a] =
    [atom[fn] -> [eq[fn,CAR] -> caar[x];
    eq[fn,CDR] -> cdar[x];
    eq[fn,CONS] -> cons[car[x];cadr[x]];
    eq[fn,ATOM] -> atom[car[x]];
    eq[fn,EQ] -> eq[car[x];cadr[x]];
    T -> apply[eval[fn;a];x;a]];
    eq[car[fn];LAMBDA] -> eval[caddr[fn];pairlis[caddr[fn];x;a]];
    eq[car[fn];LABEL] -> apply[caddr[fn];x;cons[cons[caddr[fn];
    caddr[fn]];a]]

eval[e;a] = [atom[e] -> cdr[assoc[e;a]];
    atom[car[e]] ->
        [eq[car[e],QUOTE] -> cadr[e];
        eq[car[e],COND] -> evcon[cdr[e];a];
        T -> apply[car[e];evlis[cdr[e];a];a]];
    T -> apply[car[e];evlis[cdr[e];a];a]]
```

Eval/Apply in Logo

