61A Lecture 29

Monday, November 7
Homework: Huffman Encoding Trees

Efficient encoding of strings as ones and zeros (bits).
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
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
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:
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
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
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
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
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
Data types: Words and sentences (immutable sequences)
Logo Refresher

Data types: Words and sentences (immutable sequences)

Syntactic forms: Call expressions, literals, and to-statements
Data types: Words and sentences (immutable sequences)

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

? print sum 10 difference 7 3
14
Logo Refresher

**Data types:** Words and sentences (immutable sequences)

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

```logo
? (print) sum 10 difference 7 3
14
```

Monday, November 7, 2011
Data types: Words and sentences (immutable sequences)

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

? (print sum 10 difference 7 3)
14
Data types: Words and sentences (immutable sequences)

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

? print sum 10 difference 7 3
14
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
```
Data types: Words and sentences (immutable sequences)

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

```logo
? (print sum 10 difference 7 3)
14

? run [print sum 1 2]
3

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

parser
Logo Interpreter Architecture

parser

Evaluator
Logo Interpreter Architecture

parser

Evaluator
Logo Interpreter Architecture

string → parser → Evaluator
Logo Interpreter Architecture

string -> parser -> Evaluator

'run [print sum 1 2]'
Logo Interpreter Architecture

string -> parser -> line -> Evaluator

'run [print sum 1 2]'
Logo Interpreter Architecture

string parser line Evaluator

'run [print sum 1 2]' ['run', ['print', 'sum', '1', '2']]
Logo Interpreter Architecture

Logo words are represented as Python strings
Logo Interpreter Architecture

string parser line Evaluator

'run [print sum 1 2]' ['run', ['print', 'sum', '1', '2']]

Logo words are represented as Python strings
Logo sentences are represented as Python lists
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
Logo Interpreter Architecture

'run [print sum 1 2]'  ['run', ['print', 'sum', '1', '2']]

A line of Logo code

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
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.
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.
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
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
Tracking Positions in Lines

A line is used up as it is evaluated
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.
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.

```python
>>> buf = Buffer([\'show\', \'2\'])
```
A line is used up as it is evaluated

A Buffer instance tracks how much of a line has been used up.

```python
>>> buf = Buffer(['show', '2'])
```

```plaintext
show 2
```
A line is used up as it is evaluated

A Buffer instance tracks how much of a line has been used up.

```python
>>> buf = Buffer(['show', '2'])
>>> buf.current
'show'
```
A line is used up as it is evaluated

A Buffer instance tracks how much of a line has been used up.

```python
>>> buf = Buffer(['show', '2'])
>>> buf.current
'show'
>>> print(buf)
[ 'show', 2 ]
```

Monday, November 7, 2011
A line is used up as it is evaluated

A Buffer instance tracks how much of a line has been used up.

```python
>>> buf = Buffer(['show', '2'])
>>> buf.current
'show'
```
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.

```python
>>> buf = Buffer(["show", '2'])
>>> buf.current
'show'
>>> print(buf)
[  >> show, 2 ]
>>> buf.pop()
'show'
>>> print(buf)
[ show >> 2 ]
```
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.

```python
>>> buf = Buffer(['show', '2'])
>>> buf.current
'show'
>>> print(buf)
[ show >> 2 ]
>>> buf.pop()
'show'
>>> print(buf)
[ show >> 2 ]
>>> buf.pop()
'2'
```
A line is used up as it is evaluated

A Buffer instance tracks how much of a line has been used up.

```python
>>> buf = Buffer(['show', '2'])
>>> buf.current
'show'
>>> print(buf)
[ 'show', 2 ]
>>> buf.pop()
'show'
>>> print(buf)
[ 'show', 2 ]
>>> buf.pop()
'2'
```
Evaluating Lines
Evaluating Lines

Evaluating a line of Logo involves evaluating each expression.
Evaluating Lines

Evaluating a line of Logo involves evaluating each expression

Evaluate a line

eval_line
Evaluating Lines

Evaluating a line of Logo involves evaluating each expression

Evaluate a line

eval_line

Evaluate the next expression

logo_eval
Evaluating Lines

Evaluating a line of Logo involves evaluating each expression

Evaluate a line

Evaluate the next expression
Evaluating Lines

Evaluating a line of Logo involves evaluating each expression

Evaluate a line

Evaluate the next expression

Calls repeatedly

eval_line

logo_eval
Evaluating Lines

Evaluating a line of Logo involves evaluating each expression

Evaluate a line

Evaluate the next expression

Calls repeatedly

? print 1 print 2
1
2
Evaluating Lines

Evaluating a line of Logo involves evaluating each expression

Evaluate a line

Evaluate the next expression

Calls repeatedly

? print 1 print 2

1

2

logo_eval Argument Effect

first call

second call
Evaluating Lines

Evaluating a line of Logo involves evaluating each expression

Evaluate a line

Evaluate the next expression

Calls repeatedly

? print 1 print 2
1
2

<table>
<thead>
<tr>
<th>logo_eval</th>
<th>Argument</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>first call</td>
<td>[ &gt;&gt; print, 1, print, 2 ]</td>
<td></td>
</tr>
<tr>
<td>second call</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Monday, November 7, 2011
Evaluating Lines

Evaluating a line of Logo involves evaluating each expression

Evaluate a line

Evaluate the next expression

? print 1 print 2
1
2

<table>
<thead>
<tr>
<th>logo_eval</th>
<th>Argument</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>first call</td>
<td>[       &gt;&gt; print, 1, print, 2 ]</td>
<td>prints 1, returns None</td>
</tr>
<tr>
<td>second call</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Evaluating Lines

Evaluating a line of Logo involves evaluating each expression

Evaluate a line

Evaluate the next expression

? print 1 print 2
1
2

<table>
<thead>
<tr>
<th>logo_eval</th>
<th>Argument</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>first call</td>
<td>[ print, 1 &gt;&gt; print, 1, print, 2 ]</td>
<td>prints 1, returns None</td>
</tr>
<tr>
<td>second call</td>
<td>[ print, 1 &gt;&gt; print, 2 ]</td>
<td></td>
</tr>
</tbody>
</table>
Evaluating Lines

Evaluating a line of Logo involves evaluating each expression

Evaluate a line

Evaluate the next expression

\[
\text{? print 1 print 2}
\]

1

2

<table>
<thead>
<tr>
<th>logo_eval</th>
<th>Argument</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>first call</td>
<td>[ print, 1, print, 2 ]</td>
<td>prints 1, returns None</td>
</tr>
<tr>
<td>second call</td>
<td>[ print, 1, print, 2 ]</td>
<td>prints 2, returns None</td>
</tr>
</tbody>
</table>
Logo Evaluation
Logo Evaluation

The logo_eval function dispatches on expression form:
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.
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.
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.
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.
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.
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):
```
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.""
```
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()
```
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()
```

The expression form can be inferred from the first token.
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):
```
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
```

The expression form can be inferred from the first token.
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):
        pass
```

The expression form can be inferred from the first token.
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):
        ...
```

The expression form can be inferred from the first token.
Evaluating Call Expressions
Evaluating Call Expressions

Apply a named procedure  

apply_procedure
Evaluating Call Expressions

Apply a named procedure

Evaluate $n$ operands

apply_procedure

collect_args
Evaluating Call Expressions

Apply a named procedure

Evaluate \( n \) operands

Apply a procedure to a sequence of arguments

apply_procedure

collect_args

logo_apply
Evaluating Call Expressions

Apply a named procedure

Apply a procedure to a sequence of arguments

Evaluate $n$ operands

apply_procedure

collect_args

logo_apply

Return the output value
Evaluating Call Expressions

Apply a named procedure

apply_procedure

Return the output value

Evaluate $n$ operands

collect_args

Return $n$ arguments

Apply a procedure to a sequence of arguments

logo_apply
Evaluating Call Expressions

Apply a named procedure

apply_procedure

Return the output value

Evaluate \( n \) operands

collect_args

Return \( n \) arguments

Apply a procedure to a sequence of arguments

logo_apply

Return the output value
Evaluating Call Expressions

Apply a named procedure

apply_procedure

Return the output value

Evaluate $n$ operands

collect_args

Return $n$ arguments

Apply a procedure to a sequence of arguments

logo_apply

Return the output value

[ print >> 2 ]
Evaluating Call Expressions

Apply a named procedure

Apply a procedure to a sequence of arguments

Evaluate $n$ operands

apply_procedure

collect_args

logo_apply

Return the output value

Return $n$ arguments

Return the output value

[ print >> 2 ]

Popped by logo_eval
Evaluating Call Expressions

Apply a named procedure

Apply a procedure to a sequence of arguments

Evaluate $n$ operands

$\text{apply\_procedure}$

Return the output value

$\text{collect\_args}$

Return $n$ arguments

$\text{logo\_apply}$

Return the output value

$\text{print} \gg 2$

Popped by $\text{logo\_eval}$

1. Collect 1 argument via $\text{logo\_eval}$

$(\text{collect\_args})$
Evaluating Call Expressions

Apply a named procedure

Evaluate \( n \) operands

Apply a procedure to a sequence of arguments

apply_procedure

Return the output value

collect_args

Return \( n \) arguments

logo_apply

Return the output value

[ print >> 2 ]

Popped by logo_eval

1. Collect 1 argument via logo_eval (collect_args)

[ print, 2 >> ]
Evaluating Call Expressions

Apply a named procedure

Apply a procedure to a sequence of arguments

Evaluate $n$ operands

apply_procedure

Return the output value

collect_args

Return $n$ arguments

logo_apply

Return the output value

1. Collect 1 argument via logo_eval (collect_args)

[ print >> 2 ]

Popped by logo_eval

[ print, 2 >> ]

Popped by logo_eval also (recursive call)
Evaluating Call Expressions

Apply a named procedure

Apply a procedure to a sequence of arguments

Evaluate \( n \) operands

**apply\_procedure**

Return the output value

**collect\_args**

Return \( n \) arguments

**logo\_apply**

Return the output value

1. Collect 1 argument via **logo_eval** (**collect\_args**)
2. Apply print procedure to the argument '2' (**logo\_apply**)

\[
\begin{align*}
\text{[ print } & \gg 2 \text{ ]} \\
\text{Popped by logo\_eval}
\end{align*}
\]

\[
\begin{align*}
\text{[ print, 2 } & \gg \text{ ]} \\
Popped by logo\_eval also (recursive call)
\end{align*}
\]
Procedures
```python
class Procedure():
```

Monday, November 7, 2011
class Procedure():
    def __init__(self, name, arg_count, body, isprimitive=False, needs_env=False, formal_params=None):
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
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."""
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)
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
Eval/Apply in Lisp 1.5
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[cadr[fn];x;a]];]

eq[car[fn];LABEL] → apply[caddr[fn];x;cons[cons[cadr[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

**Eval**

- `eval_line`
  - `logo_eval`

**Apply**

- `apply_procedure`
  - `collect_args`
    - `logo_apply`
Eval/Apply in Logo

Eval

- eval_line
- logo_eval
  - Call expressions

Apply

- apply_procedure
  - collect_args
  - logo_apply
Eval/Apply in Logo

**Eval**
- eval_line
- logo_eval
  - Call expressions

**Apply**
- apply_procedure
  - collect_args
  - logo_apply
  - Operand expressions
Eval/Apply in Logo

Eval

- eval_line
- logo_eval
- Call expressions

Apply

- apply_procedure
- collect_args
- logo_apply

User-defined procedures
Operand expressions