The Logo Programming Language
The Logo Programming Language

A teaching language: designed for introductory programming
The Logo Programming Language

A teaching language: designed for introductory programming

One syntactic form for all purposes: invoking a procedure
The Logo Programming Language

A teaching language: designed for introductory programming

One syntactic form for all purposes: invoking a procedure

Only two data types: words and sentences
The Logo Programming Language

A teaching language: designed for introductory programming

One syntactic form for all purposes: invoking a procedure

Only two data types: words and sentences

Code is data: a line of code is a sentence
The Logo Programming Language

A teaching language: designed for introductory programming

One syntactic form for all purposes: invoking a procedure

Only two data types: *words* and *sentences*

Code is data: a line of code is a *sentence*

An elegant tagline: no threshold, no ceiling
The Logo Programming Language

A teaching language: designed for introductory programming

One syntactic form for all purposes: invoking a procedure

Only two data types: words and sentences

Code is data: a line of code is a sentence

An elegant tagline: no threshold, no ceiling

A bit of fun: turtle graphics
The Logo Programming Language

A teaching language: designed for introductory programming

One syntactic form for all purposes: invoking a procedure

Only two data types: words and sentences

Code is data: a line of code is a sentence

An elegant tagline: no threshold, no ceiling

A bit of fun: turtle graphics

Demo
Logo is a Dialect of Lisp
Logo is a Dialect of Lisp

What are people saying about Lisp?
Logo is a Dialect of Lisp

What are people saying about Lisp?

• "The greatest single programming language ever designed."
  - Alan Kay (from the UI video), co-inventor of Smalltalk
Logo is a Dialect of Lisp

What are people saying about Lisp?

• "The greatest single programming language ever designed."
  - Alan Kay (from the UI video), co-inventor of Smalltalk

• "The only computer language that is beautiful."
  - Neal Stephenson, John's favorite sci-fi author
Logo is a Dialect of Lisp

What are people saying about Lisp?

• "The greatest single programming language ever designed."
  - Alan Kay (from the UI video), co-inventor of Smalltalk

• "The only computer language that is beautiful."
  - Neal Stephenson, John's favorite sci-fi author

• "God's programming language."
  - Brian Harvey, Father of CS 61A
Logo is a Dialect of Lisp

What are people saying about Lisp?

- "The greatest single programming language ever designed."
  - Alan Kay (from the UI video), co-inventor of Smalltalk

- "The only computer language that is beautiful."
  - Neal Stephenson, John's favorite sci-fi author

- "God's programming language."
  - Brian Harvey, Father of CS 61A
Logo Fundamentals

Call expressions are delimited by spaces
Logo Fundamentals

Call expressions are delimited by spaces

Logo *procedures* are equivalent to Python functions
Call expressions are delimited by spaces

Logo procedures are equivalent to Python functions
• A procedure takes inputs (arguments) that are values
Call expressions are delimited by spaces

Logo procedures are equivalent to Python functions

• A procedure takes inputs (arguments) that are values
• A procedure returns an output (return value)
Logo Fundamentals

Call expressions are delimited by spaces

Logo procedures are equivalent to Python functions

• A procedure takes inputs (arguments) that are values
• A procedure returns an output (return value)
• A procedure may output None to indicate no return value
Call expressions are delimited by spaces

Logo *procedures* are equivalent to Python functions

- A procedure takes *inputs* (arguments) that are values
- A procedure returns an *output* (return value)
- A procedure may output None to indicate no return value

```python
? print 5
5
```
Call expressions are delimited by spaces

Logo procedures are equivalent to Python functions

• A procedure takes *inputs* (arguments) that are values
• A procedure returns an *output* (return value)
• A procedure may output None to indicate no return value

```
? print 5
5
```

Multiple expressions can appear in a single line
Logo Fundamentals

Call expressions are delimited by spaces

Logo *procedures* are equivalent to Python functions
- A procedure takes *inputs* (arguments) that are values
- A procedure returns an *output* (return value)
- A procedure may output None to indicate no return value

```logo
? print 5
5

Multiple expressions can appear in a single line

? print 1 print 2
1
2```
Nested Call Expressions
Nested Call Expressions

The syntactic structure of expressions is determined by the number of arguments required by named procedures.
Nested Call Expressions

The syntactic structure of expressions is determined by the number of arguments required by named procedures

```
? print sum 10 difference 7 3
14
```
Nested Call Expressions

The syntactic structure of expressions is determined by the number of arguments required by named procedures.

```plaintext
print takes one argument (input)

? (print) sum 10 difference 7 3
14
```
Nested Call Expressions

The syntactic structure of expressions is determined by the number of arguments required by named procedures.

- **print** takes one argument (input)
- **sum** takes two inputs

```
?print(sum 10 difference 7 3)
14
```
Nested Call Expressions

The syntactic structure of expressions is determined by the number of arguments required by named procedures.

- **print** takes one argument (input)
- **sum** takes two inputs

```
? (print) (sum) 10 difference 7 3
14
```
The syntactic structure of expressions is determined by the number of arguments required by named procedures.

- **print** takes one argument (input).
- **sum** takes two inputs.
- **difference** takes two inputs too.

```
?print(sum 10 difference 7 3)
14
```
Nested Call Expressions

The syntactic structure of expressions is determined by the number of arguments required by named procedures.

- print takes one argument (input)
- sum takes two inputs
- difference takes two inputs too

\[
? \text{print} \quad \text{sum} \quad 10 \quad \text{difference} \quad 7 \quad 3 \\
14
\]
Nested Call Expressions

The syntactic structure of expressions is determined by the number of arguments required by named procedures.

- **print** takes one argument (input)
- **sum** takes two inputs
- **difference** takes two inputs too

```
?print(sum(10) difference(7, 3))
14
```
Nested Call Expressions

The syntactic structure of expressions is determined by the number of arguments required by named procedures.

- print takes one argument (input)
- sum takes two inputs
- difference takes two inputs too

```
?print sum 10 difference 7 3
14
```
Nested Call Expressions

The syntactic structure of expressions is determined by the number of arguments required by named procedures.

- `print` takes one argument (input)
- `sum` takes two inputs
- `difference` takes two inputs too

```
? print
sum 10 difference 7 3
14
```
Nested Call Expressions

The syntactic structure of expressions is determined by the number of arguments required by named procedures.

- **print** takes one argument (input)
- **sum** takes two inputs
- **difference** takes two inputs too

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

One nested call expression
Nested Call Expressions

The syntactic structure of expressions is determined by the number of arguments required by named procedures.

- `print` takes one argument (input)
- `sum` takes two inputs
- `difference` takes two inputs too

One nested call expression

*versus*
Nested Call Expressions

The syntactic structure of expressions is determined by the number of arguments required by named procedures.

- **print** takes one argument (input)
- **sum** takes two inputs
- **difference** takes two inputs too

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

One nested call expression

versus

Two expressions on one line
Nested Call Expressions

The syntactic structure of expressions is determined by the number of arguments required by named procedures

print takes one argument (input)

sum takes two inputs

difference takes two inputs too

One nested call expression

versus

Two expressions on one line
Nested Call Expressions

The syntactic structure of expressions is determined by the number of arguments required by named procedures.

- print takes one argument (input)
- sum takes two inputs
- difference takes two inputs too

One nested call expression

versus

Two expressions on one line

? print 1 print 2
1
2

Demo
Data Types and Quotation
Data Types and Quotation

Words are strings without spaces, representing text, numbers, and boolean values
Data Types and Quotation

Words are strings without spaces, representing text, numbers, and boolean values

? print "hello
hello
Data Types and Quotation

Words are strings without spaces, representing text, numbers, and boolean values

```c
? print "hello
hello
? print "sum
sum```

Friday, November 4, 2011
Data Types and Quotation

Words are strings without spaces, representing text, numbers, and boolean values

```ruby
? print "hello
hello
? print "sum
sum
? print "2
2
```
Data Types and Quotation

Words are strings without spaces, representing text, numbers, and boolean values

```
? print "hello
hello
? print "sum
sum
? print "2
2
```

Sentences are immutable sequences of words and sentences
Data Types and Quotation

Words are strings without spaces, representing text, numbers, and boolean values

```python
? print "hello
hello
? print "sum
sum
? print "2
2
```

Sentences are immutable sequences of words and sentences

```python
? print [hello world]
hello world
```
Words are strings without spaces, representing text, numbers, and boolean values

```
? print "hello
hello
? print "sum
sum
? print "2
2
```

Sentences are immutable sequences of words and sentences

```
? print [hello world]
hello world
? show [hello world]
[hello world]
```
Sentence (List) Processing in Logo
Sentence (List) Processing in Logo

Sentences can be constructed from words or sentences
Sentence (List) Processing in Logo

Sentences can be constructed from words or sentences

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Sentence (List) Processing in Logo

Sentences can be constructed from words or sentences

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>sentence</td>
<td>Output a sentence containing all elements of two sentences. Input words are converted to sentences.</td>
</tr>
</tbody>
</table>
### Sentence (List) Processing in Logo

Sentences can be constructed from words or sentences

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>sentence</td>
<td>Output a sentence containing all elements of two sentences. Input words are converted to sentences.</td>
</tr>
<tr>
<td>list</td>
<td>Output a sentence containing the two inputs.</td>
</tr>
</tbody>
</table>
## Sentence (List) Processing in Logo

Sentences can be constructed from words or sentences.

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>sentence</td>
<td>Output a sentence containing all elements of two sentences. Input words are converted to sentences.</td>
</tr>
<tr>
<td>list</td>
<td>Output a sentence containing the two inputs.</td>
</tr>
<tr>
<td>fput</td>
<td>Output a sentence containing the first input and all elements in the second input.</td>
</tr>
</tbody>
</table>
## Sentence (List) Processing in Logo

Sentences can be constructed from words or sentences

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>sentence</td>
<td>Output a sentence containing all elements of two sentences. Input words are converted to sentences.</td>
</tr>
<tr>
<td>list</td>
<td>Output a sentence containing the two inputs.</td>
</tr>
<tr>
<td>fput</td>
<td>Output a sentence containing the first input and all elements in the second input.</td>
</tr>
</tbody>
</table>

**Demo**
Expressions are Sentences
Expressions are Sentences

The run procedure evaluates a sentence as a line of Logo code and outputs its value.
Expressions are Sentences

The run procedure evaluates a sentence as a line of Logo code and outputs its value

```
? run [print sum 1 2]
3
```
Expressions are Sentences

The run procedure evaluates a sentence as a line of Logo code and outputs its value

? run [print sum 1 2]
3

Its argument can be constructed from other procedure calls
Expressions are Sentences

The run procedure evaluates a sentence as a line of Logo code and outputs its value

```
? run [print sum 1 2]
3
```

Its argument can be constructed from other procedure calls

```
? run sentence "print [sum 1 2]
3
```
Expressions are Sentences

The run procedure evaluates a sentence as a line of Logo code and outputs its value

```
? run [print sum 1 2]
3
```

Its argument can be constructed from other procedure calls

```
? run sentence "print [sum 1 2]
3
```

```
? print run sentence "sum sentence 10 run [difference 7 3]
14
```
Expressions are Sentences

The run procedure evaluates a sentence as a line of Logo code and outputs its value

```
? run [print sum 1 2]
3
```

Its argument can be constructed from other procedure calls

```
? run sentence "print [sum 1 2]
3
```

```
? print run sentence "sum sentence 10 (run [difference 7 3])
14
```
Expressions are Sentences

The run procedure evaluates a sentence as a line of Logo code and outputs its value

```
? run [print sum 1 2]
3
```

Its argument can be constructed from other procedure calls

```
? run sentence "print [sum 1 2]
3

? print run sentence "sum sentence 10 (run [difference 7 3])
14
```
Expressions are Sentences

The run procedure evaluates a sentence as a line of Logo code and outputs its value

? run [print sum 1 2]
3

Its argument can be constructed from other procedure calls

? run sentence "print [sum 1 2]
3

? print run sentence "sum sentence 10 (run [difference 7 3])
14
Procedures
Procedures

Procedure definition is a special form, not a call expression
Procedures

Procedure definition is a special form, not a call expression

? to double : x
> output sum : x : x
> end
Procedure definition is a special form, not a call expression

```plaintext
Procedure name

? to{double} : x
> output sum : x : x
> end
```
Procedures

Procedure definition is a special form, not a call expression

```plaintext
? to {double}: x
> output sum : x : x
> end
```
Procedure definition is a special form, not a call expression.

Procedure name

Formal parameter

? to (double): x

Body

> output sum : x : x,
> end
Procedure definition is a special form, not a call expression

```plaintext
Procedure name

? to double : x

> output sum : x : x

> end

? print double 4

8
```
Procedures

Procedure definition is a special form, not a call expression

Procedures are not first-class objects in Logo; they can only ever be referenced by their original procedure name
Procedures

Procedure definition is a special form, not a call expression

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

? print double 4
8
```

Procedures are not first-class objects in Logo; they can only ever be referenced by their original procedure name

Procedure names can be inputs or outputs
Conditional Procedures
Conditional Procedures

If and ifelse are regular procedures in Logo
Conditional Procedures

If and ifelse are regular procedures in Logo

*Meaning*: They do not have a special evaluation procedure
Conditional Procedures

If and ifelse are regular procedures in Logo

*Meaning:* They do not have a special evaluation procedure

They take sentences as inputs and run them conditionally
Conditional Procedures

If and ifelse are regular procedures in Logo

_Meaning:_ They do not have a special evaluation procedure

They take sentences as inputs and run them conditionally

```logo
? to reciprocal :x
> if not :x = 0 [output 1 / :x]
> output "infinity
> end
```
Conditional Procedures

If and ifelse are regular procedures in Logo

*Meaning:* They do not have a special evaluation procedure.
They take sentences as inputs and run them conditionally.

```logo
? to reciprocal :x
> if not :x = 0 [output 1 / :x]
> output "infinity
> end

? print reciprocal 2
0.5
```
Conditional Procedures

If and ifelse are regular procedures in Logo

*Meaning*: They do not have a special evaluation procedure

They take sentences as inputs and run them conditionally

```
? to reciprocal :x
> if not :x = 0 [output 1 / :x]
> output "infinity"
> end

? print reciprocal 2
0.5

? print reciprocal 0
infinity
```
Dynamic Scope
Dynamic Scope

When one function calls another, the names bound in the local frame for the first are accessible to the body of the second
Dynamic Scope

When one function calls another, the names bound in the local frame for the first are accessible to the body of the second

No isolation of formal parameters to function bodies, as we saw with lexical scope
Dynamic Scope

When one function calls another, the names bound in the local frame for the first are accessible to the body of the second

No isolation of formal parameters to function bodies, as we saw with lexical scope

```haskell
? to print_x :x
> print_last_x
> end
```
Dynamic Scope

When one function calls another, the names bound in the local frame for the first are accessible to the body of the second.

No isolation of formal parameters to function bodies, as we saw with lexical scope.

```plaintext
? to print_x : x
> print_last_x
> end

? to print_last_x
> print : x
> end
```
Dynamic Scope

When one function calls another, the names bound in the local frame for the first are accessible to the body of the second

No isolation of formal parameters to function bodies, as we saw with lexical scope

? to print_x :x
> print_last_x
> end

? to print_last_x
> print :x
> end

? print_x 5
5
Logo Examples

Demo
Homework: Huffman Encoding Trees
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).

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>C</td>
<td>1010</td>
<td>E</td>
</tr>
<tr>
<td>B</td>
<td>100</td>
<td>D</td>
<td>1011</td>
<td>F</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>G</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>H</td>
</tr>
</tbody>
</table>
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:

1000 1010
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

Friday, November 4, 2011