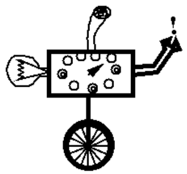
CS 60



Logic

November 5, 2001

Assignment 8

```
program -> { command }

command -> cs          // clear screen
         | home        // home turtle
         | pu          // lift pen up
         | pd          // put pen down
         | fd number   // move forward
         | rt number   // right-turn (deg.)
         | repeat number [ program ]
```

Input: *Tokens*

```
// excerpts from the Logo.java
interface Token
class StringToken implements Token
class LeftBracketToken implements Token
class EOFToken implements Token
...
class LogoTokenizer {
  Token getToken()
  bool nextTokenIsNumber()
  double getNumber()
  ...
}
```

Output: *Program*

```
// from the Logo.java
class Program {
  void addCommand(Command c)
  void execute()
  ...
}
interface Command { ... }
class PUCommand implements Command
class FDCommand implements Command
class RepeatCommand implements Command
...
```

Why Study Logic?

- A basis for computer hardware
- A basis for computer programming
- A basis for specification
- A basis for verification and testing

- In a certain sense

Computing *is* Logic

Propositional Logic

- Built up from
  - `0` (a.k.a. `false`)
  - `1` (a.k.a. `true`)
  - propositional variables `p, q, r, x, y, z,...`
    - These variables may be true or false
  - and, either
    - functions (functional view)
    - connectives (expression view)

Some Statements

- True Statements:

- False Statements:

Truth Tables for `and`

p	q	and(p,q)
0	0	0
0	1	0
1	0	0
1	1	1

and	0	1
0	0	0
1	0	1

Alternate notations: $p \wedge q$
pq
`p && q`

Rex code for and

- 4-case definition:

```
and(0, 0) => 0;
and(0, 1) => 0;
and(1, 0) => 0;
and(1, 1) => 1;
```

- Using convention that cases are sequential:

```
and(1, 1) => 1;
and(p, q) => 0;
```

Truth Tables for or

p	q	or(p,q)
0	0	**0**
0	1	**1**
1	0	**1**
1	1	**1**

or	0	1
0	0	1
1	1	1

Alternate notations: $p \vee q$
p+q
p || q

Rex code for or

- 4-case definition:

```
or(0, 0) => 0;
or(0, 1) => 1;
or(1, 0) => 1;
or(1, 1) => 1;
```

- Using convention that cases are sequential:

Negation

p	not(p)
0	1
1	0

```
not(0) => 1;
not(1) => 0;
```

Alternate notations: $\neg p$
p'
$\overline{p}$
!p

(Material) Implication

p	q	implies(p,q)
0	0	1
0	1	1
1	0	0
1	1	1

imp	0	1
0	1	1
1	0	1

Alternate notations: $p \supset q$
$p \Rightarrow q$
$p \rightarrow q$

Rex code for `implies`

- 4-case definition:

```
implies(0, 0) => 1;
implies(0, 1) => 1;
implies(1, 0) => 0;
implies(1, 1) => 1;
```

- Using convention that cases are sequential:

Expression Forms

- Use for greater readability of certain equalities
- Similar to ordinary discourse

Logical Expressions

- Example (These mean the same thing)
  $(a \wedge b) \vee (c \wedge \neg d)$
  $ab + cd'$
- Precedence: not, and, or
- To start, we'll use
  $\wedge \quad \vee \quad \neg \quad \rightarrow \quad \equiv$
- When we discuss circuits, we'll use
  $\cdot \quad + \quad ' \quad =$

Logical Equivalences

- Commutative Rules

$$a \wedge b = b \wedge a$$
$$a \vee b = b \vee a$$

- Associative Rules

$$(a \wedge b) \wedge c = a \wedge (b \wedge c)$$
$$(a \vee b) \vee c = a \vee (b \vee c)$$

- Distributive Rules

$$(a \vee b) \wedge c = (a \wedge c) \vee (b \wedge c)$$
$$(a \wedge b) \vee c = (a \vee c) \wedge (b \vee c)$$

More Logical Equivalences

$$(a \wedge 0) = 0$$
$$(a \wedge 1) = a$$

$$(a \vee 0) = a$$
$$(a \vee 1) = 1$$

$$\neg (a \wedge b) = (\neg b \vee \neg a)$$
$$\neg (a \vee b) = (\neg b \wedge \neg a)$$

Equivalences for Implication

$$(a \rightarrow b) = (\neg a \vee b)$$
$$(a \rightarrow b) = \neg(a \wedge \neg b)$$

$$(0 \rightarrow b) = 1$$
$$(1 \rightarrow b) = b$$
$$(a \rightarrow 0) = \neg a$$
$$(a \rightarrow 1) = 1$$

$$(a \rightarrow bc) = (a \rightarrow b) \wedge (a \rightarrow c)$$
$$((a \rightarrow b) \wedge (b \rightarrow c)) \rightarrow (a \rightarrow c)$$
$$(a \rightarrow b) = (\neg b \rightarrow \neg a)$$

Checking Relations using the **Boole-Shannon Principle**

- Relations hold iff they hold for any substitution of 0 and 1 for the variables (uniformly throughout the expression)
- Therefore, a relation holds if, choosing *any* variable V, it holds for V = 0 and for V = 1.
- But substituting 0 or 1 for a variable often yields simplifications that make the relation obvious.

Example

- Verify $(a \to b) = (\neg b \to \neg a)$
  - Choose a as the variable;
  - Check when a=0 and when a=1

Tautologies

- An expression that always evaluates to 1 (true) regardless of what value each variable is assigned is called a ***tautology***.
- The property of being a tautology can be checked using:
  - Truth-table construction
  - Boole-Shannon Principle, recursively
- Example of a tautology checker (applet):

`http://www.cs.hmc.edu/~keller/javaExamples/taut/taut.html`

Encodings

- In order to use logic to build computers and other devices, we need to represent or encode general finite domains into the logic domain.
- At a sufficiently low-level, most information in a digital system is encoded into strings (or tuples) of **bits.**

Encodings

- Let $\{0, 1\}^n$ mean the set of all n-tuples of 0's and 1's, e.g.
- $\{0,1\}^3 = \{000, 001, 010, 011, 100, 101, 110, 111\}$
- An *encoding* of a set S is a function from S into $\{0,1\}^n$ for some n (the "number of bits").
  - Function should be *injective* (a.k.a. *one-to-one*)

Encoding Examples
(note: $\rightarrow$ is *maps to*, not *implies*)

- Encode the set {red, green, blue, black}
  - Encoding #1:
    - red → 00, green → 01, blue → 10, black → 11
  - Encoding #2:
    - red → 01, green → 10, blue → 11, black → 00
  - Encoding #3 (called **"one-hot" encoding**)
    - red → 1000, green → 0100, blue → 0010 , black → 0001

How many bits are enough?

- To encode a set of size `N`,

$\lceil \log_2(N) \rceil$ bits are needed, at a minimum

- $\lceil K \rceil$ is the smallest integer $\geq$ `K`

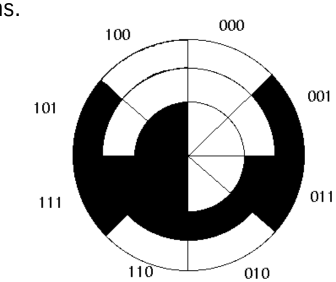
  (read the "ceiling" of `K`).

More Encoding Examples

- Encode the set {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15}
  - Encoding #1 (straight binary encoding)
    - `0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111`
  - Encoding #2 (Gray encoding)
    - `0000, 0001, 0011, 0010, 0110, 0111, 0101, 0100, 1100, 1101, 1111, 1110, 1010, 1011, 1001, 1000`

Gray Code

- Invented by Frank Gray.
- U.S. Patent 2 632 058, March 17, 1953.
- Many important applications.
  - E.g., shaft-position encoder
  - No glitches!



Example of a Commercial "Shaft Encoder"

Position information is provided as parallel **Gray Code** or Natural Binary, serial RS422, or 0-10V and/or 4-20mA analog ouputs.

Available Configurations
FD-850 Incremental Digital
FD-850A 8 to 12 bit Absolutes

Building Gray Codes

- A one-bit Gray code is easy:
- To get an n-bit Gray Code:
  - Take two copies of the `(n-1)`-bit Gray code
  - Reverse the second copy
  - Prepend 0 to the elements of the first copy
  - Prepend 1 to the elements of the second copy

Building Gray Codes

Gray code generator in rex

```
gray(0) => [[]];

gray(n) =>
   prev = gray(n-1),
   append(map((X) => [0|X], prev),
          map((X) => [1|X], reverse(prev)));


gray(3) ==>
     [[0,0,0], [0,0,1], [0,1,1], [0,1,0],
      [1,1,0], [1,1,1], [1,0,1], [1,0,0]]
```