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 StringTokenizer 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 \( p, q, r, x, y, z, \ldots \)
    - 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*

<table>
<thead>
<tr>
<th>( p )</th>
<th>( q )</th>
<th>( \text{and}(p,q) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Alternate notations: \( p \land q \)
\( pq \)
\( p \& \& q \)
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;
  ```

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:

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

Truth Tables for **or**

<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
<th>or(p, q)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Alternate notations:  \( p \lor q \)  
\( p + q \)  
\( p \mid \mid q \)

Negation

<table>
<thead>
<tr>
<th>p</th>
<th>not(p)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Alternate notations:  \( \neg p \)  
\( p' \)  
\( \bar{p} \)  
\( !p \)
(Material) Implication

<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
<th>implies(p,q)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

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

Rex code for `implies`

- 4-case definition:

```plaintext
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 \land b) \lor (c \land \neg d)
  
  ab + cd'$

- Precedence: not, and, or
- To start, we'll use

  $\land \lor \neg \rightarrow \equiv$

- When we discuss circuits, we'll use

  $\cdot + \neg \Rightarrow \equiv$
Logical Equivalences

• Commutative Rules
  \[a \land b = b \land a\]
  \[a \lor b = b \lor a\]

• Associative Rules
  \[(a \land b) \land c = a \land (b \land c)\]
  \[(a \lor b) \lor c = a \lor (b \lor c)\]

• Distributive Rules
  \[(a \lor b) \land c = (a \land c) \lor (b \land c)\]
  \[(a \land b) \lor c = (a \lor c) \land (b \lor c)\]

More Logical Equivalences

\[(a \land 0) = 0\]
\[(a \land 1) = a\]
\[(a \lor 0) = a\]
\[(a \lor 1) = 1\]
\[\neg(a \land b) = (\neg b \lor \neg a)\]
\[\neg(a \lor b) = (\neg b \land \neg a)\]

Equivalences for Implication

\((a \rightarrow b) = (\neg a \lor b)\)
\((a \rightarrow b) = \neg(a \land \neg b)\)
\[(0 \rightarrow b) = 1\]
\[(1 \rightarrow b) = b\]
\[(a \rightarrow 0) = \neg a\]
\[(a \rightarrow 1) = 1\]
\[(a \rightarrow bc) = (a \rightarrow b) \land (a \rightarrow c)\]
\[((a \rightarrow b) \land (b \rightarrow c)) \rightarrow (a \rightarrow c)\]
\[(a \rightarrow b) = (\neg b \rightarrow \neg a)\]

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 \((a \rightarrow b) = (\neg b \rightarrow \neg a)\)
  - 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: → 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 \(N\),
  \[\log_2(N)\] 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 outputs.

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

**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]]
```