Chapter 3: Programming

The Vector

- We can build (or at least imagine!) lots of circuits.
- We can even think about state machines that use circuits to do various things over time.
- We’re headed towards using these ideas to create a programmable computer.
- Hillis has us take a detour to talk about programming first.
Programming

• Many different languages have been devised for programming.

• Each provides a way of writing a kind of script along with rules for the computer to interpret the script as instructions.

• They can all do pretty much the same things, but make it easier to say some things than others.

• How many languages can you name?

Scratch

• Developed by the “Lifelong Kindergarten” group at MIT.

• Allows users to make media-rich programs by clicking together blocks.

• Share your creations a la YouTube.

• My miniNim program is an example.
Things

- Performance area
  - stage (background)
  - sprites (objects)
    - scripts (behavior)
    - costumes (appearance)
    - sounds (available sounds)
More Things

- Run/Stop
- Script inventory
  - Motion
  - Looks
  - Sound
  - Pen
  - Control
- Sensing
- Numbers
- Variables

Shapes

- Trigger
- Statement
- Ending statement
- Boolean value
- Numeric value
Motion Inventory

- move 10 steps
- turn 15 degrees
- turn 25 degrees
- point in direction 90
- point towards
- go to x: 0 y: 0
- go to
- glide 1 secs to x: 0 y: 0
- change x by 10
- set x to 0
- change y by 10
- set y to 0
- if on edge, bounce

Looks Inventory

- switch to costume costume2
- next costume
- costume #
- say Hello! for 2 secs
- say Hello!
- think Hmm... for 2 secs
- think Hmm...
- change color effect by 25
- set color effect to 0
- clear graphic effects
- change size by 10
- set size to 100%
- size
- show
- hide
- go to front
- go back 1 layers

- color, fisheye, whirl, pixelate, mosaic, brightness, ghost
Sound Inventory

play sound meow until done
stop all sounds
play drum 48 for 0.2 beats
rest for 0.2 beats
play note 60 for 0.5 beats
set instrument to 1

change volume by -10
set volume to 100%

change tempo by 20
set tempo to 60 bpm

large collection of synthesized sounds

Pen Inventory

clear
down
pen up
set pen color to
change pen color by 10
set pen color to 0
change pen shade by 10
set pen shade to 50
change pen size by 1
set pen size to 1
stamp
Control Inventory

- when flag clicked
- when space key pressed
- when Sprite clicked
- wait 1 secs
- forever
- repeat 10
- if
- when I receive
- forever if
- if
- stop script
- stop all

Sensing Inventory

- mouse x
- mouse y
- mouse down?
- key space pressed?
- touching ?
- touching color ?
- color is touching ?
- distance to
- reset timer
- timer
- x position of Sprite1
- loudness
- loud?
- slider sensor value
- sensor button pressed?
Numbers Inventory

Variables Inventory
Logo in Scratch

- Logo is a language invented to help people (kids?) learn to program.
- Scratch is the language we’ve been using for examples and demos in this class. It’s a descendent of Logo, in many ways.
- Hillis uses Logo for his examples in this chapter.
- I’ll translate them into Scratch so it fits better with our other examples.

Drawing Commands

![Drawing Commands](image-url)
New Command

when I receive square
    pen down
    move 100 steps
    turn ↘ 90 degrees
    move 100 steps
    turn ↘ 90 degrees
    move 100 steps
    move 100 steps

Do It!

broadcast square and wait
Use It!

Command With Variable
Conditionals

- Based on the random number, two possible outcomes.
- Executes only one block of code, depending on the Boolean condition.

If.  If is good.

(Infinite) Looping

Never never never stops!

(bug in book)
Kinds of Loops

• Infinite loop

• “For” loop: Repeat a set number of times.
  ‣ Flexible.

• “Repeat until” loop: Until a condition holds.
  ‣ More Flexible.

• Recursion: Repeat substructure.
  ‣ Most Flexible.

Just Four, Thanks

• Sets a repeat count

• Stops after that number of repetitions
**Unroll The Loop**

- It’s as if the statements inside the loop are repeated four times.

**More Power**

- “For” (repeat) loops are great if you know how many times you will be repeating.
- Sometimes need something more powerful.
- “While” (repeat-until) loops keep going until a condition becomes true.
- Can behave like for loops...
Four Square Countdown

Unroll, With Conditions
Even More Power

- “While” (repeat-until) loops are great if repetitions are sequential.
- Sometimes need something more powerful.
- “Recursion” can allow control to proceed in multiple directions at once!
- But, can also behave like for loops...

Infinite Recursion

- Each “pattern” message spawns another.
- Ad infinitum
Proper Recursion

• “Base case” says what to do when the counter runs down.
• In this case, it stops when the counter reaches zero.

Not A Simple Loop

• Loop proceeds in two directions at once.
• Can’t do that with for or while.
(In fact, can’t do it at all in Scratch, but it is commonplace in other programming languages.)
I Think That I Shall Never See...

- Each tree is made up of a trunk and two smaller trees.
- Base case keeps the structure from growing infinitely.
- Makes shapes known as “fractals”.
- Other examples?

Recap

- Using logic gates, we know how to do a bunch of things with bits:
  - test equality
  - if-then-else gate
  - select one bit from a set (universal gate)
What Can We Do?

- Lots: Any function of bits, we can specify with logic gates.
- But, creating dedicated circuitry for every new problem is daunting and inefficient.
- Would like a way of using a fixed set of circuits to act like any circuitry we might want.
- We can use the state-machine idea to trade gates for time...

Christmas Light Architecture
Consider Reprogramming

\[ A = (A \land \neg(B \land C)) \lor (\neg A \land (B \land C)) \]

- Reprogramming the Christmas lights required that we create a bunch of circuits including the one above.
- It’s got 2 “not” gates, 4 “and” gates, and 1 “or” gate wired in a particular pattern.
- It would be nice if we didn’t have to play with gates and wires to change behavior.
Circuit, Visualized

- \( A = (A \text{ and not } (B \text{ and } C)) \text{ or } (\text{not } A \text{ and } (B \text{ and } C)) \)
- Separate circuits for \( B \) and \( C \).
- Ready for blinking!

Concrete Example: Adding

<table>
<thead>
<tr>
<th></th>
<th>( x )</th>
<th>( y )</th>
<th>( z )</th>
<th>( c )</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>+ 10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

- We want to compute the sum of \( x \) and \( y \) (2-bit numbers). \( z \) (3 bits) is the answer and \( c \) (2 bits) is the carry.
- \( z_0 = (x_0 \text{ and not } y_0) \text{ or } (\text{not } x_0 \text{ and } y_0) \)
- \( c_0 = (x_0 \text{ and } y_0) \)
Truth Table That Adds Bits

- Basic step:
  - 3 bits in
  - 2 bits out
- sum bit
- new carry bit

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>c</th>
<th>c'</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Concrete Example: Adding

\[
\begin{array}{cccc}
10 & c_1 & c_0 \\
11 & x_1 & x_0 \\
+ 10 & y_1 & y_0 \\
101 & z_2 & z_1 & z_0 \\
\end{array}
\]

- \( z_1 = (x_1 \text{ and not } y_1 \text{ and not } c_0) \text{ or } (\text{not } x_1 \text{ and } y_1 \text{ and not } c_0) \text{ or } (\text{not } x_1 \text{ and not } y_1 \text{ and } c_0) \text{ or } (x_1 \text{ and } y_1 \text{ and } c_0) \)

- \( c_1 = z_2 = (x_1 \text{ and } y_1 \text{ and not } c_0) \text{ or } (\text{not } x_1 \text{ and } y_1 \text{ and } c_0) \text{ or } (\text{not } x_1 \text{ and not } y_1 \text{ and } c_0) \text{ or } (x_1 \text{ and } y_1 \text{ and } c_0) \)

Adding Bytes

- Computing \( z_i \) and \( c_i \) from \( x_i, y_i, \) and \( c_{i-1} \) can be carried out with 6 ands, 3 ors, 4 nots.

- The previous slide uses 16 ands, 6 ors, and 9 nots (not as good).

- This operation is called a “full adder”.

- By chaining together one full adder per bit, we can make a circuit that adds any number of bits (4, 8, 16, 32, 64, etc.).
Hardware

- Any function we want to implement from bits to bits can be carried out by constructing the right circuit of and/or/nots.
- Creating a circuit solves the problem “in hardware”.
- The advantage of hardware solutions are that they are fast.
- The disadvantage is that they are inflexible.

Software

- The lovely thing about a computer is that the hardware does not have to change for the computer to change its behavior.
- A fixed set of circuits can actually change its behavior to represent any desired function!
- Build one, reprogram into anything.
- Disadvantage of the software approach: Can be much slower.
Key Insight

- Make a language for expressing operations.
- Complex enough to capture the important functions.
- Simple enough to be implementable in hardware.

Machine Language

Break it Down

- \( A = (A \text{ and not } (B \text{ and } C)) \text{ or } (\text{not } A \text{ and } (B \text{ and } C)) \)

<table>
<thead>
<tr>
<th>Equation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>acc = B</td>
<td>acc now holds “B”</td>
</tr>
<tr>
<td>acc = acc and C</td>
<td>acc now holds “B and C”</td>
</tr>
<tr>
<td>E = acc</td>
<td>E now holds “B and C”</td>
</tr>
<tr>
<td>acc = not A</td>
<td>acc now holds “not A and (B and C)”</td>
</tr>
<tr>
<td>acc = acc and E</td>
<td>acc now holds “not A and (B and C)”</td>
</tr>
<tr>
<td>F = acc</td>
<td>F now holds “not A and (B and C)”</td>
</tr>
<tr>
<td>acc = not E</td>
<td>acc now holds “not (B and C)”</td>
</tr>
<tr>
<td>acc = acc and A</td>
<td>acc now holds “A and not (B and C)”</td>
</tr>
<tr>
<td>acc = acc or F</td>
<td>acc now holds “(A and not (B and C)) or (not A and (B and C))”</td>
</tr>
<tr>
<td>A = acc</td>
<td>A holds the new value of the equation</td>
</tr>
</tbody>
</table>
**Instruction Set: 7 Bits**

- **V** in 0000...1111 (variables A- P)
- 000V: \( \text{acc} = \text{acc or V} \)
- 001V: \( \text{acc} = \text{acc and V} \)
- 010V: \( \text{acc} = V \)
- 011V: \( \text{acc} = \text{not V} \)
- acc: special temporary variable
- 100V: \( V = \text{acc or V} \)
- 101V: \( V = \text{acc and V} \)
- 110V: \( V = \text{acc} \)
- 111V: \( V = \text{not acc} \)

Idea

- We write a little program that would perform the same function as the circuit.
- We make a circuit that can execute any program.
- Reduction!
Memory

• Need a place to store the various quantities we’re working with.

• Main memory is like a giant filing cabinet, where each drawer is numbered consecutively and can store one value.

• Need to be able to store and retrieve values.

Variables

• Let’s say we need to store 100 numbers.

• Can name them:
  ‣ apple, asparagus, artichoke, apricot, banana, blueberry, blackberry, cantaloupe, ..., zucchini

• Tedious to assign names to them all.
A List of Variables
• For convenience, if nothing else, use numbers to name the variables.
  ‣ item 1 of var, item 2 of var, ..., item 100 of var.

Indirection
• Naming the variables with numbers gives us some additional power!
• Can use a variable to name another variable.
Memory Circuit

- We’ll have 32 memory locations, for instructions.
- Each one has a 5-bit name (0-31) called its “address”.
- Memory circuit takes the contents of memory (32 x 7 bits) and an address, 229 bits in all, & outputs the data stored at the corresponding address.

Writing to Memory

- Similar circuit allows memory cells to be altered.
- mem[address] = newval
- If needed for future processing, copied back up at the end of the cycle.
Persistence of Memory

• We can use this memory idea to store the Boolean variables (A-P).

• We can also use another set of memory locations to store the series of instructions to be executed (program).

• How is the instructions stored?

Bits For One Instruction

\[
\begin{array}{cccccc}
\text{b6} & \text{b5} & \text{b4} & \text{b3} & \text{b2} & \text{b1} & \text{b0} \\
\end{array}
\]

- load/store (1 bit)
  - 0: load; 1: store

- instruction (2 bits)
  - 00: acc or V
  - 01: acc and V
  - 10: acc (load) / V (store)
  - 11: not acc (load) / not V (store)

- variable name (4 bits)
  - 1011000
    - store = 1
    - instruction = 01
    - constant = 1000 = I
    - So, “I = acc and I”
**Series of Instructions**

- **Address**
- **Contents (decimal)**
- **Contents (binary)**

<table>
<thead>
<tr>
<th>Contents (decimal)</th>
<th>Contents (binary)</th>
</tr>
</thead>
<tbody>
<tr>
<td>051</td>
<td>acc = not D</td>
</tr>
<tr>
<td>0110011</td>
<td></td>
</tr>
<tr>
<td>017</td>
<td>acc = acc and B</td>
</tr>
<tr>
<td>0010011</td>
<td></td>
</tr>
<tr>
<td>211</td>
<td>P = acc</td>
</tr>
<tr>
<td>1101111</td>
<td></td>
</tr>
<tr>
<td>49</td>
<td>acc = not B</td>
</tr>
<tr>
<td>0110011</td>
<td></td>
</tr>
<tr>
<td>419</td>
<td>acc = acc and D</td>
</tr>
<tr>
<td>0010011</td>
<td></td>
</tr>
<tr>
<td>515</td>
<td>acc = acc or P</td>
</tr>
<tr>
<td>0001111</td>
<td></td>
</tr>
<tr>
<td>6104</td>
<td>I = acc</td>
</tr>
<tr>
<td>11101000</td>
<td></td>
</tr>
<tr>
<td>733</td>
<td>acc = B</td>
</tr>
<tr>
<td>0100001</td>
<td></td>
</tr>
<tr>
<td>819</td>
<td>acc = acc and D</td>
</tr>
<tr>
<td>0010011</td>
<td></td>
</tr>
</tbody>
</table>

- **Program counter:** which address’s instruction to process next

- **Registers:**
  - **Boolean variables and their values**
  - **Accumulator:** Special register

- **Michael Littman’s Mini Logic Machine Language (ML³)**

---

**von Neumann Architecture**

- A computer is just a big state machine.
- **Input:** registers, memory, input devices
- **Output:** new values for registers, memory, output devices
- **PC** = Program counter, the address of the statement to be executed.

- **CPU** = Central Processing Unit
- **mem**
- **acc**
- **PC**
- **reg**

- **mem**
  - 7x32 bits
  - 1x16 bits
- **acc**
  - 5 bits
- **CYCLE**
- **PC**
  - 1 bit
- **reg**
  - 246 bits total
Cycle: A Whole Computer

Instruction Sets

- ML³ used a particular design that made it relatively easy to fit in a lecture slide while handling 2-bit addition.
- Computer manufacturers have different goals in mind: cost, speed, ease of running modern programs.
- Some quick examples:
### x86: Intel’s Old Set

#### Z80: My First

- **INC**: Increment
- **ADD**: Add
- **ADC**: Add with Carry
- **ADDX**: Add with Extended
- **ADDZ**: Add with Zero Fault
- **NEG**: Negate
- **SUBL**: Subtract
- **SBB**: Subtract with Borrow
- **SBBX**: Subtract with Borrow Extended
- **SUBX**: Subtract with Extended
- **NOT**: Invert
- **XOR**: Exclusive OR
- **ORL**: OR (Logical)
- **ORLX**: OR (Logical)
- **AND**: AND
- **ADC**: Add with Carry
- **ADDX**: Add with Extended
- **ADDZ**: Add with Zero Fault
- **NEG**: Negate
- **SUBL**: Subtract
- **SBB**: Subtract with Borrow
- **SBBX**: Subtract with Borrow Extended
- **SUBX**: Subtract with Extended
- **NOT**: Invert
- **XOR**: Exclusive OR
- **ORL**: OR (Logical)
- **ORLX**: OR (Logical)
- **AND**: AND
- **ADC**: Add with Carry
- **ADDX**: Add with Extended
- **ADDZ**: Add with Zero Fault
- **NEG**: Negate
- **SUBL**: Subtract
- **SBB**: Subtract with Borrow
- **SBBX**: Subtract with Borrow Extended
- **SUBX**: Subtract with Extended
- **NOT**: Invert
- **XOR**: Exclusive OR
- **ORL**: OR (Logical)
- **ORLX**: OR (Logical)
- **AND**: AND
- **ADC**: Add with Carry
- **ADDX**: Add with Extended
- **ADDZ**: Add with Zero Fault
- **NEG**: Negate
- **SUBL**: Subtract
- **SBB**: Subtract with Borrow
- **SBBX**: Subtract with Borrow Extended
- **SUBX**: Subtract with Extended
- **NOT**: Invert
- **XOR**: Exclusive OR
- **ORL**: OR (Logical)
- **ORLX**: OR (Logical)
- **AND**: AND
- **ADC**: Add with Carry
- **ADDX**: Add with Extended
- **ADDZ**: Add with Zero Fault
- **NEG**: Negate
- **SUBL**: Subtract
- **SBB**: Subtract with Borrow
- **SBBX**: Subtract with Borrow Extended
- **SUBX**: Subtract with Extended
- **NOT**: Invert
- **XOR**: Exclusive OR
- **ORL**: OR (Logical)
- **ORLX**: OR (Logical)
- **AND**: AND
- **ADC**: Add with Carry
- **ADDX**: Add with Extended
- **ADDZ**: Add with Zero Fault
- **NEG**: Negate
- **SUBL**: Subtract
- **SBB**: Subtract with Borrow
- **SBBX**: Subtract with Borrow Extended
- **SUBX**: Subtract with Extended
- **NOT**: Invert
- **XOR**: Exclusive OR
- **ORL**: OR (Logical)
- **ORLX**: OR (Logical)
- **AND**: AND
- **ADC**: Add with Carry
- **ADDX**: Add with Extended
- **ADDZ**: Add with Zero Fault
- **NEG**: Negate
- **SUBL**: Subtract
- **SBB**: Subtract with Borrow
- **SBBX**: Subtract with Borrow Extended
- **SUBX**: Subtract with Extended
- **NOT**: Invert
- **XOR**: Exclusive OR
- **ORL**: OR (Logical)
- **ORLX**: OR (Logical)
- **AND**: AND
- **ADC**: Add with Carry
- **ADDX**: Add with Extended
- **ADDZ**: Add with Zero Fault
- **NEG**: Negate
- **SUBL**: Subtract
- **SBB**: Subtract with Borrow
- **SBBX**: Subtract with Borrow Extended
- **SUBX**: Subtract with Extended
- **NOT**: Invert
- **XOR**: Exclusive OR
- **ORL**: OR (Logical)
- **ORLX**: OR (Logical)
- **AND**: AND
- **ADC**: Add with Carry
- **ADDX**: Add with Extended
- **ADDZ**: Add with Zero Fault
- **NEG**: Negate
- **SUBL**: Subtract
- **SBB**: Subtract with Borrow
- **SBBX**: Subtract with Borrow Extended
- **SUBX**: Subtract with Extended
- **NOT**: Invert
- **XOR**: Exclusive OR
- **ORL**: OR (Logical)
- **ORLX**: OR (Logical)
- **AND**: AND
- **ADC**: Add with Carry
- **ADDX**: Add with Extended
- **ADDZ**: Add with Zero Fault
- **NEG**: Negate
- **SUBL**: Subtract
- **SBB**: Subtract with Borrow
- **SBBX**: Subtract with Borrow Extended
- **SUBX**: Subtract with Extended
- **NOT**: Invert
- **XOR**: Exclusive OR
- **ORL**: OR (Logical)
- **ORLX**: OR (Logical)
- **AND**: AND
- **ADC**: Add with Carry
- **ADDX**: Add with Extended
- **ADDZ**: Add with Zero Fault
- **NEG**: Negate
- **SUBL**: Subtract
- **SBB**: Subtract with Borrow
- **SBBX**: Subtract with Borrow Extended
- **SUBX**: Subtract with Extended
- **NOT**: Invert
- **XOR**: Exclusive OR
- **ORL**: OR (Logical)
- **ORLX**: OR (Logical)
- **AND**: AND
- **ADC**: Add with Carry
- **ADDX**: Add with Extended
- **ADDZ**: Add with Zero Fault
- **NEG**: Negate
- **SUBL**: Subtract
- **SBB**: Subtract with Borrow
- **SBBX**: Subtract with Borrow Extended
- **SUBX**: Subtract with Extended
- **NOT**: Invert
- **XOR**: Exclusive OR
- **ORL**: OR (Logical)
- **ORLX**: OR (Logical)
CARDIAC (1968)

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>INP</td>
<td>Input – take a number from the input card and put it in a specified memory cell.</td>
</tr>
<tr>
<td>1</td>
<td>CLA</td>
<td>Clear and add – clear the accumulator and add the contents of a memory cell to the accumulator.</td>
</tr>
<tr>
<td>2</td>
<td>ADD</td>
<td>Add - add the contents of a memory cell to the accumulator.</td>
</tr>
<tr>
<td>3</td>
<td>TAC</td>
<td>Test accumulator contents – performs a sign test on the contents of the accumulator.</td>
</tr>
<tr>
<td>4</td>
<td>SFT</td>
<td>Shift – shifts the accumulator x places left, then y places right.</td>
</tr>
<tr>
<td>5</td>
<td>OUT</td>
<td>Output – take a number from the specified memory cell and write it on an output card.</td>
</tr>
<tr>
<td>6</td>
<td>STO</td>
<td>Store – copy the contents of the accumulator into a specified memory cell.</td>
</tr>
<tr>
<td>7</td>
<td>SUB</td>
<td>Subtract – subtract the contents of a specified memory cell from the accumulator.</td>
</tr>
<tr>
<td>8</td>
<td>JMP</td>
<td>Jump - jump to a specified memory cell</td>
</tr>
<tr>
<td>9</td>
<td>HRS</td>
<td>Halt and reset – stop program execution, move bug to cell 00.</td>
</tr>
</tbody>
</table>

CARDboard Illustrative Aid to Computation

Evaluating Expression Trees

• I’m now going to switch gears into a topic that bridges the gap between programs and circuits.

• It will also give us a different view of logical expressions.
  ‣ Expression Trees
Expression Trees: Small

- True
  - True
- False
  - False
  - and
    - True
    - False
- True and False
  - not
    - not True
      - True
- not True

Expression Trees: Bigger

- True and True
  - and
    - True
    - True
  - or
    - not
      - not False
      - True
- not True or False
Expression Trees: Combined

- (True and True) and (not True or False)

Trees and Subtrees

- Just as we can build more complex expressions out of simpler ones, we can build more complex trees out of simpler ones.
- An expression tree is: ET =
  
True or False or not or and or or
Circular Definition?

• In a sense, this definition looks broken because it is defining an expression tree in terms of expression trees.
• This circularity is safe, because the definition also provides us a way to stop (True/False).
• It’s also necessary, because there’s an infinitely large set of possible expressions.

Some Tree Terminology

• root: The top node of the tree (“True”, “False”, “not”, “and”, or “or”).
• subtree: A tree underneath the root.
• left subtree: The subtree to the left.
• right subtree: The subtree to the right.
• leaf: A tree with no subtrees of its own.
• depth: number of nodes between the root and the farthest leaf.
Evaluate Bottom Up

- Evaluate a tree whenever all its subtrees are evaluated.

Evaluate Top Down

- Start at the root.
- Ask a friend to evaluate the subtrees.
- Do the root.
Not That Different, Really

• In a sense, you have to start at the bottom.
• But, what recursion (self delegation?) does is let you focus on what happens at the top and the lower-down stuff just works itself out.
• Can make for much cleaner code.

evaluateTree Function

• Takes a tree as input, returns True/False.
• In some sense, very literal!
• But, uses recursion to handle the messy lower-level stuff.
• Somehow, extremely natural and extremely mind bending.

```python
def evaluateTree(tree):
    if root(tree) == 'True': return True
    if root(tree) == 'False': return False
    if root(tree) == 'not':
        val = evaluateTree(subtree(tree))
        return not val
    if root(tree) == 'and':
        v1 = evaluateTree(leftSubtree(tree))
        v2 = evaluateTree(rightSubtree(tree))
        return v1 and v2
    if root(tree) == 'or':
        v1 = evaluateTree(leftSubtree(tree))
        v2 = evaluateTree(rightSubtree(tree))
        return v1 or v2
```
Expressions With Variables

- evaluateTree: Took an expression tree (with Trues and Falses) as input and returned its value.
- What about an expression with variables: “not A or B”?
- If we know A and B’s values, can substitute them in and use evaluateTree! Interpreter.

Tree ⇒ Program

- If A and B are not known, but we still want to do something useful, we can convert the expression tree into a program that, given A and B, produces the value of the expression!

```
acc = not A
acc = acc or B
{ answer in acc }
```
Compiler

- An interpreter takes a program as input and makes it happen.
- A compiler takes a program as input and creates a machine-language program as output.
- A program that converts a program into a program—twisted, but useful!

Game Plan

- We’ll develop two schemes that compute the value of the expression.
  - One leaves the final value in “acc”.
  - The other leaves it in a variable in memory.
- Both are given a list of variables they can use for temporary storage.
- Mutually recursive (oooh).
Code Generation: Var

- Expression: A

  answer in acc
  acc = A

  answer in P
  acc = A
  P = acc


Code Generation: not

- Expression: not EXPR

  answer in acc (temp N)
  generate code for EXPR, answer in N
  acc = not N

  answer in P
  generate code for EXPR, answer in acc
  P = not acc

- Note: We keep a pool of temporary variables to use as needed (not just N).
Code Generation: or

- Expression: \((\text{EXPR}_1 \text{ or } \text{EXPR}_2)\)

  answer in acc (temp \(N\))
  generate code for
  \(\text{EXPR}_1\), answer in \(N\)
  generate code for
  \(\text{EXPR}_2\), answer in acc
  \(\text{acc} = \text{acc or } N\)

- Note: “and” is handled the same way.

---

Example Expression

- Expression: \(\neg A \text{ or } B\)

  answer in acc
  generate code for
  \(\neg A\), answer in \(N\)
  generate code for
  \(B\), answer in acc
  \(\text{acc} = \text{acc or } N\)
  \(\text{or}\)

  or

  \(\text{ET}_1\)

  \(\text{ET}_2\)

  answer in P
  generate code for
  \(\neg A\), answer in \(P\)
  generate code for
  \(B\), answer in acc
  \(\text{acc} = \text{acc or } P\)

  or

  \(\text{not}\)

  \(\text{B}\)

  \(\text{A}\)

  \(\text{P} = \text{acc or } P\)
Assembler

- An assembler handles the last little step of translating the series of instructions to a series of numbers.

<table>
<thead>
<tr>
<th>answer in acc</th>
<th>answer in P</th>
</tr>
</thead>
<tbody>
<tr>
<td>acc = A 32</td>
<td>acc = A 32</td>
</tr>
<tr>
<td>N = not acc 125</td>
<td>P = not acc 127</td>
</tr>
<tr>
<td>acc = B 33</td>
<td>acc = B 33</td>
</tr>
<tr>
<td>acc = acc or N 13</td>
<td>P = acc or P 79</td>
</tr>
</tbody>
</table>

Complete Code

```python
def compileToVar(tree, temps, target):
    if root(tree) == 'V': return [LOAD(ACC(VAR(tree)))] + [STORE(ACC(target))]
    if root(tree) == 'not':
        return compileToAcc(subtree(tree), temps+[target]) + [STORE(NOT(target))]
    if root(tree) == 'or':
        finalCmd = STORE(OR(temps[0]))
    if root(tree) == 'and':
        finalCmd = STORE(AND(temps[0]))
    return compileToVar(leftSubtree(tree), temps, target) + compileToAcc(rightSubtree(tree), temps) + [finalCmd]

def compileToAcc(tree, temps):
    if root(tree) == 'V': return [LOAD(ACC(VAR(tree)))]
    assert(len(temps)>0)
    if root(tree) == 'not':
        return compileToVar(subtree(tree), temps[1:], temps[0]) + [LOAD(NOT(temps[0]))]
    if root(tree) == 'or':
        finalCmd = LOAD(OR(temps[0]))
    if root(tree) == 'and':
        finalCmd = LOAD(AND(temps[0]))
    return compileToVar(leftSubtree(tree), temps[1:], temps[0]) + compileToAcc(rightSubtree(tree), temps) + [finalCmd]
```
Inefficiency

- (not A or B)
- Automatically generated machine code:

  ```
  answer in acc
  acc = A  32
  N = not acc  125
  acc = B  33
  acc = acc or N  13
  ```

- By hand:

  ```
  answer in acc
  acc = not A  48
  acc = acc or B  1
  ```

- Often more than one way to do it!

Optimizations

- Many ways of speeding up compiled code have been developed.
- Want to minimize steps, temporary variables.
- I’ll describe two important ones:
  - Shared subexpressions
  - Logical equivalence
**Automatic Code (13 inst.)**

- $E = ((A \land B) \land C) \lor ((A \land B) \land D)$
  - $E = (A \land B) \land C$
    - $E = A \land B$
      - acc = A
      - E = acc
    - acc = B
    - E = acc and E
  - acc = C
  - E = acc and E

- acc = (A \land B) \land D
  - $N = A \land B$
    - N = A
    - acc = A
    - N = acc
    - acc = B
    - N = acc and N
    - acc = D
    - acc = acc and N
  - E = acc or E

- 1 temps, 13 instructions

**Shared Subexpression**

- $E = ((A \land B) \land C) \lor ((A \land B) \land D)$
  - $E = (N \lor C) \land D$
    - $E = N \land C$
      - acc = N
      - E = E and acc
  - acc = N
    - acc = acc and O
    - E = acc or O

- 2 temps, 13 instructions
Logical Equivalence

- \( E = ((A \text{ and } B) \text{ and } C) \text{ or } ((A \text{ and } B) \text{ and } D) \)
- \( E = (A \text{ and } B) \text{ and } (C \text{ or } D) \)
  - \( E = A \text{ and } B \)
  - \( acc = A \)
  - \( E = acc \)
  - \( acc = B \)
  - \( E = E \text{ and } acc \)
  - \( acc = C \text{ or } D \)
  - \( acc = C \)
- \( N = acc \)
- \( acc = D \)
- \( acc = acc \text{ or } N \)
- \( E = E \text{ and } acc \)
- 1 temps, 9 instructions

A Compiler

- A program that translates computer programs that people write into a machine language instructions for the computer to execute.

(Adapted from notes by Barbara Ryder.)
Interpreters

- *Compiler* translates program to machine lang.
- *Interpreter* translates statements by executing equivalent commands
  - No real translation step
- Interpretation requires programming language have a defined meaning for its statements---*semantics*
  - Sometimes defined mathematically, sometimes in English.

Recap: Reduction

<table>
<thead>
<tr>
<th>Level</th>
<th>Examples</th>
<th>Alternatives</th>
</tr>
</thead>
<tbody>
<tr>
<td>software libraries</td>
<td>graphics, animation, robotics</td>
<td>networking, security, mathematics</td>
</tr>
<tr>
<td>high-level language</td>
<td>Python</td>
<td>C, Java, C++, Logo, LISP, Fortran, ML</td>
</tr>
<tr>
<td>machine language</td>
<td>ML³</td>
<td>x86, CARDIAC, Z80</td>
</tr>
<tr>
<td>logic gates</td>
<td>equal, ifthenelse, add</td>
<td>memlookup, memwrite</td>
</tr>
<tr>
<td>basic logic gates</td>
<td>and, or, not</td>
<td>nor, nand, xor</td>
</tr>
<tr>
<td>physical bits</td>
<td>0,1 via high/low voltage</td>
<td>water pressure, kinetic energy</td>
</tr>
</tbody>
</table>
Does It End There?

- Of course not.
- We can continue to build sophisticated programs out of simpler programs.
- The idea of **subroutines** (procedures, functions) makes this work.

Subroutines

- **A lot** of research in computer science is about designing and creating just the right set of subroutines, sometimes called **libraries**.
- You don’t have enough background yet to weigh in on these problems.
- But, there is an analogous set of problems where you are already an expert...
Sing-A-Long Programs

Gilligan’s Island Theme

Just sit right back and you’ll hear a tale, a tale of a fateful trip.
That started from this tropic port, aboard this tiny ship.
The mate was a mighty sailin’ man, the skipper brave and sure.
Five passengers set sail that day, for a three hour tour, a three hour tour...
The weather started getting rough, the tiny ship was tossed.
If not for the courage of the fearless crew,
the Minnow would be lost; the Minnow would be lost.
The ship took ground on the shore of this uncharted desert isle,
with Gilligan, the Skipper too, the Millionaire, and his Wife,
the Movie Star, the Professor and Mary Ann, here on Gilligan’s Isle.

Scratch Code

```
when Sprite1 clicked
play sound Just sit right back
wait 6 secs
play sound That started from
wait 4 secs
play sound The mate was a
wait 11 secs
play sound The weather started
wait 12 secs
play sound The ship set ground
wait 13 secs
play sound The movie star
wait 8 secs
stop all sounds
```
Clementine

In a cavern, in a canyon,
Excavating for a mine,
Dwelt a miner forty niner,
And his daughter Clementine.

Oh my darling, oh my darling,
Oh my darling, Clementine!
Thou art lost and gone forever
Dreadful sorry, Clementine.

Light she was and like a fairy,
And her shoes were number nine,
Herring boxes, without topses,
Sandals were for Clementine.

Oh my darling, oh my darling,
Oh my darling, Clementine!
Thou art lost and gone forever
Dreadful sorry, Clementine.

Drove she ducklings to the water
Ev'ry morning just at nine,
Hit her foot against a splinter,
Fell into the foaming brine.

Oh my darling, oh my darling,
Oh my darling, Clementine!
Thou art lost and gone forever
Dreadful sorry, Clementine.

Ruby lips above the water,
Blowing bubbles, soft and fine,
But, alas, I was no swimmer,
So I lost my Clementine.

Oh my darling, oh my darling,
Oh my darling, Clementine!
Thou art lost and gone forever
Dreadful sorry, Clementine.

How I missed her! How I missed her,
How I missed my Clementine,
But I kissed her little sister,
I forgot my Clementine.

Oh my darling, oh my darling,
Oh my darling, Clementine!
Thou art lost and gone forever
Dreadful sorry, Clementine.

Chorus Structure

Clementine

In a cavern, in a canyon,
Excavating for a mine,
Dwelt a miner forty niner,
And his daughter Clementine.

Oh my darling, oh my darling,
Oh my darling, Clementine!
Thou art lost and gone forever
Dreadful sorry, Clementine.

Light she was and like a fairy,
And her shoes were number nine,
Herring boxes, without topses,
Sandals were for Clementine.

Oh my darling, oh my darling,
Oh my darling, Clementine!
Thou art lost and gone forever
Dreadful sorry, Clementine.

Drove she ducklings to the water
Ev'ry morning just at nine,
Hit her foot against a splinter,
Fell into the foaming brine.

Oh my darling, oh my darling,
Oh my darling, Clementine!
Thou art lost and gone forever
Dreadful sorry, Clementine.

Ruby lips above the water,
Blowing bubbles, soft and fine,
But, alas, I was no swimmer,
So I lost my Clementine.

Oh my darling, oh my darling,
Oh my darling, Clementine!
Thou art lost and gone forever
Dreadful sorry, Clementine.

How I missed her! How I missed her,
How I missed my Clementine,
But I kissed her little sister,
I forgot my Clementine.

Oh my darling, oh my darling,
Oh my darling, Clementine!
Thou art lost and gone forever
Dreadful sorry, Clementine.

Scratch Code

when Miner clicked
play sound In a Cavern, in a
play sound Dwelt a miner and wait
play sound Oh my darling and wait
play sound Thou art lost and wait
play sound Light she was and wait
play sound Herring boxes and wait
play sound Oh my darling and wait
play sound Thou art lost and wait
play sound Drove she duckings and wait
play sound Hit her foot against and wait
play sound Ruby lips above the and wait
play sound But, alas, I was no and wait
play sound Oh my darling and wait
play sound Thou art lost and wait
play sound How I missed her and wait
play sound But I kissed my little and wait
play sound Oh my darling and wait
play sound Thou art lost and wait
A Subroutine

- Now, “Chorus” is its own little program, or a new statement in the language, that we can call whenever we need it.
- Calling the subroutine does all the steps, then the program resumes from where the call took place.
- Simplifies the program, easier to maintain, extend, fix.

Shared Structure

All My Loving
Lennon/McCartney

Close your eyes and I'll kiss you,
Tomorrow I'll miss you,
Remember I'll always be true,
And then while I'm away,
I'll write home every day,
And I'll send all my loving to you.
I'll pretend I am kissing,
The lips I am missing,
And hope that my dreams will come true,
And then while I'm away,
I'll write home every day,
And I'll send all my loving to you.
All my loving, I will send to you,
All my loving, darling, I'll be true.
Close your eyes and I'll kiss you,
Tomorrow I'll miss you,
Remember I'll always be true,
And then while I'm away,
I'll write home every day,
And I'll send all my loving to you.
All my loving, I will send to you,
All my loving, darling, I'll be true,
All my loving. All my loving
Woo, all my loving, I will send to you

What subroutines would you define?
Deeper Nesting

All My Loving
Lennon/McCartney

Close your eyes and I'll kiss you,
Tomorrow I'll miss you,
Remember I'll always be true,
And then while I'm away,
I'll write home every day,
And I'll send all my loving to you.

I'll pretend I am kissing,
The lips I am missing,
And hope that my dreams will come true,
And then while I'm away,
I'll write home every day,
And I'll send all my loving to you.

All my loving, I will send to you,
All my loving, darling, I'll be true.

More Structure

Eight Days a Week
(Beatles)

Ooh I need your love babe, guess you know it’s true,
Hope you need my love babe just like I need you,

Hold me, love me,
Hold me, love me,
Ain’t got nothin’ but love babe,
Eight days a week.

Love you ev’ry day girl, always on my mind,
One thing I can say girl, love you all the time,

Hold me, love me,
Hold me, love me,
Ain’t got nothin’ but love babe,
Eight days a week.

Eight days a week I love you,
Eight days a week is not enough to show I care.

Hold me, love me,
Hold me, love me,
Ain’t got nothin’ but love babe,
Eight days a week.

Eight days a week, Eight days a week.
**Building Structure**

**Old Macdonald had a farm, E-I-E-I-O**
And on his farm he had a cow, E-I-E-I-O

<table>
<thead>
<tr>
<th>With a “moo-moo” here and a “moo-moo” there</th>
</tr>
</thead>
<tbody>
<tr>
<td>Here a “moo” there a “moo”</td>
</tr>
<tr>
<td>Everywhere a “moo-moo”</td>
</tr>
</tbody>
</table>

**Old Macdonald had a farm, E-I-E-I-O**

**Old Macdonald had a farm, E-I-E-I-O**
And on his farm he had a pig, E-I-E-I-O

<table>
<thead>
<tr>
<th>With a (snort) here and a (snort) there</th>
</tr>
</thead>
<tbody>
<tr>
<td>Here a (snort) there a (snort)</td>
</tr>
<tr>
<td>Everywhere a (snort-snort)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>With a “moo-moo” here and a “moo-moo” there</th>
</tr>
</thead>
<tbody>
<tr>
<td>Here a “moo” there a “moo”</td>
</tr>
<tr>
<td>Everywhere a “moo-moo”</td>
</tr>
</tbody>
</table>

**Old Macdonald had a farm, E-I-E-I-O**

**Old Macdonald had a farm, E-I-E-I-O**
And on his farm he had a horse, E-I-E-I-O

<table>
<thead>
<tr>
<th>With a “neigh, neigh” here and a “neigh, neigh” there</th>
</tr>
</thead>
<tbody>
<tr>
<td>Here a “neigh” there a “neigh”</td>
</tr>
<tr>
<td>Everywhere a “neigh-neigh”</td>
</tr>
</tbody>
</table>

**But Notice...**

- Each time we “snort”, we follow it with “moo”.
- Each time we “neigh”, we follow it with “snort”.
- Each time we “cluck”, we follow it with “neigh”.
- Let’s put the earlier noises into the later subroutines...
Top Level Song

- At the highest level, the song consists of one verse per animal.

Cow

- Needs to know two things
  - Sing the verse
  - Sing the animal sounds (and highlight)
Pig

- Animal-sound subroutine oinks
  - then calls the cow’s sound to finish the verse!

Horse

- Animal-sound subroutine neighs
  - then calls the pig’s sound
    - which calls the cow’s sound to finish the verse!
Chick

- Animal-sound subroutine clucks
  - then calls the horse’s sound
    - which calls the pig’s
    - which calls the cow’s to finish the verse!

Subroutines So Far

- Can be used to gather up repeated code in one place.
- Can be deeply nested to capture structure when necessary.
- Next, parameters greatly increase the power of the subroutine idea...
Near Miss...

**Magical Mystery Tour** (Beatles)

Roll up, roll up for the mystery tour. 
Roll up, roll up for the mystery tour. 
Roll up, roll up for the mystery tour. 
Roll up, roll up for the mystery tour. 
The magical mystery tour is __waiting__ to take you away, __waiting__ to take you away.

Roll up, roll up for the mystery tour. 
Roll up, roll up for the mystery tour. 
Roll up, roll up for the mystery tour. 
The magical mystery tour is __hoping__ to take you away, __hoping__ to take you away.

Roll up, roll up for the mystery tour. 
Roll up, roll up for the mystery tour. 
Roll up, roll up for the mystery tour. 
The magical mystery tour is __coming__ to take you away, __coming__ to take you away.

Roll up, roll up for the mystery tour. 
Roll up, roll up for the mystery tour. 
Roll up, roll up for the mystery tour. 
The magical mystery tour is __dying__ to take you away, __dying__ to take you away, take you today.

• Verse structure is nearly repetitive. Can’t quite define a single subroutine that covers all three.

• If we could fill in the blank, we could reuse the same routine...

---

In The Verse...

• Plug in the word at the proper time.
Main Song

- Do each verse with a different setting of “word”.
- Then, sing the conclusion.

Recursion...

- We saw in McDonald and other examples that subroutines can call other subroutines.
- Sometimes it is useful for a subroutine to call itself.
- This is called recursion.
- First, some song examples, then more general...
A Recursive Song!

- Sing a verse
- Update bottles
- Sing the next verse

Everyday Recursion

- Some people think recursion is scary, but it’s all around us: language, nature, toys, music, family trees, mathematical expressions.

- If you like self-reference, you’ll love recursion!
Recursion Gone Wrong

- Featured in “Lambchop’s Play-along”
- Canceled.

Paintcan

A colorful song about recursion

**Tune:** Free Fallin’ by Tom Petty

**Lyrics:** Michael Littman

For CS105, Fall 2007
Rutgers University