Behind the Scenes of the Puzzle Game

Following is a look behind the scenes of the Puzzle Game within ToonTalk. Are you wondering how you are learning computer programming while having fun? Take a look at the explanations of each puzzle and you'll be surprised at how much you've learned or can learn while playing ToonTalk.

Level #1 - Numbers (mostly powers of 2)

Puzzle #1 – Marty needs a box with a 1 and 2 inside. This introduces ToonTalk boxes that are for holding things. Computer scientists call these *data structures*. Boxes closely resemble what they call *arrays*, *vectors*, and *tuples*. The numbers are an example of what computer scientists call *atomic data types*. To solve this puzzle you need to figure out how to put things in boxes (i.e. how to *initialize elements of a vector*).
Puzzle #2 – Marty needs a 4. This puzzle introduces Bammer the Mouse who does arithmetic and other basic operations. In solving this puzzle you will discover how to express the addition of two numbers.
Puzzle #3 – Marty needs a box with 8, 16, and 32 inside. This introduces a way to combine boxes to make bigger boxes. It also illustrates how to express addition of a number that is in a box. In computer science terminology you learn how to *concatenate vectors* and how to *operate on an element of a data structure*.
Puzzle #4 – Marty needs a number bigger than 1,000. This puzzle introduces the ToonTalk magic wand which is used to copy ToonTalk objects. Like all ToonTalk tools, the wand can be used as part of a program and as a tool within the programming environment. This puzzle is interesting mathematically since it relies upon repeated doubling to grow from 1 to 1,024 in only 10 steps. It is intentionally a bit tedious, to provide motivation for automating tasks like this later.
Puzzle #5 – Marty needs a zero. This puzzle introduces Dusty the Vacuum. Dusty is a tool used here to remove things. The zero is buried under things that only Dusty can remove.
Puzzle #6 – Marty needs a –1. To solve this puzzle the player needs to guess (or receive hints from Marty) that the button on Dusty's nose can be changed by clicking on it. This use of buttons to change the behavior of tools is used throughout ToonTalk. An observant player might notice that this use of Dusty to suck things up and then later spit them out corresponds very closely to *cut and paste* in many window-based interfaces.
Puzzle #7 – Marty needs a blank number pad. This puzzle introduces blank number pads which are used in later puzzles. A blank number pad is a way of expressing *the type of a data structure.* In this case, it indicates some data whose *type* is *number*.
Puzzle #8 – Marty needs a box with two zeroes. In training the robot in this puzzle, you are constructing your first program. The program constructed is equivalent to a textual program like this:

```
while (sizeOf(box) = 1 and box[0] = 1) do
    box := concat(box, copy(box));
endwhile;
```

The reason that even young children can construct this program without help, is that the puzzle constrains the world so that the amount of searching a player needs to do to find a solution isn't too large. Here the robot's memory is limited to remembering only two steps, and the robot already has a magic wand.
Puzzle #9 – Marty needs a number greater than a billion. The solution to this puzzle builds upon the experience of solving puzzles 4, 7, and 8. To solve this puzzle you need to train a robot to do what you did manually in puzzle #4. During puzzle #4 you had to repeat a sequence of actions 10 times. Here you simply train the robot to do one sequence and he'll do it the needed 30 times. In order to get the robot to work repeatedly, you need to learn about how to make the robot less fussy about the kind of box he'll work on. In computer science jargon this is called *relaxing the predicate of the conditional*. The textual program equivalent of the robot trained in this puzzle is:

```
repeat 30
    if (sizeOf(box) = 1 and isNumber(box[0])) then
        Box[0] := box[0] + copy(box[0]);
```

```
endif;
```

Puzzle #10 - Marty needs a box inside a box. To solve this puzzle, you need to place a box inside a box inside a box. This is what computer scientists call *a recursive data structure, since elements of the structure can be of the same type as the whole structure*. If you place one of the boxes in the wrong hole, then Marty informs you that the elements need to be in the correct location. This is because a box is *an ordered indexed collection*.

Puzzle #11 - Marty needs a box with 3 zeros. Like puzzle #2, you need to connect (*concatenate*) two boxes (*vectors*). Here you will discover that boxes cannot be connected when they are a part of another box. You discover that boxes can be removed, connected, and then put back. One solution corresponds to the code fragment:

```
temp1 := box[0];
temp2 := box[1];
temp1 := concat(temp1,temp2);
box[1] := temp1;
```

Puzzle #12 - Marty needs a box with 6 zeros. This puzzle resembles puzzle #11. The operation of connecting boxes is repeated 3 times. This is to prepare you for the next puzzle where a robot needs to be trained to repeatedly connect boxes.

Puzzle #13 - Marty needs a box with 10 zeros. In solving this puzzle, you train a robot to repeatedly extend a box. This introduces *a programming technique commonly used when incrementally creating a data structure*. The trained robot corresponds to the textual program:

```
repeat 4
    if (sizeOf(box) = 2 and sizeOf(box[0]) = 1 and box[0][0] = 0) then
        temp1 := copy(box[0]);
        temp1 := concat(temp1,box[1]);
        box[1] := temp;
    endif;
```

The robot in this puzzle can remember up to 20 steps but the puzzle can be solved with only 4.

Puzzle #14 - Marty needs a box containing 1, 2, 4, 8, 16, and 32. Solving this puzzle involves repeatedly extending a box with numbers that are twice the size of the previous number.

Puzzle #15 - Marty needs a box containing 1, 2, 4, 8, 16, and so on all the way up to 1,073,741,824. Solving this puzzle builds upon puzzles #9, #13, and #14. It is a good example of how one often needs to combine different programming techniques to reach a goal. Here is the equivalent textual program:

```
repeat 30
    if (sizeOf(box) = 2 and sizeOf(box[0]) = 1 and isNumber(box[0][0])) then
        box[0][0] := box[0][0] + copy(box[0][0]);
        temp1 := copy(box[0]);
        temp2 := box[1];
        temp2 := concat(temp2,temp1);
        box[1] := temp2;
    endif;
```

Puzzle #16 - Marty needs the year you were born. The powers of 2 are *a basis set of the positive integers*. In other words, any whole number can be expressed as the sum of numbers which are the power of 2 and no number in the sum occurs more than once. Hence the lack of the magic wand in this puzzle.

Puzzle #17 - Marty needs the year you were born in binary. You need to do exactly the same thing here as he or she did when solving puzzle #17. This actually can be a much easier puzzle than #16. The trick is to notice that wherever you need a 1, find a number with a 1 in that position.
Puzzles #18, #19, and #20 - Intentionally left blank.

Level #2 - Letters and Words

Puzzle #21 - Marty needs a box with A, B, and C. This puzzle introduces text pads (*a new data type*). In solving this puzzle, you will discover what the addition operation does when applied to letters.
Puzzle #22 - Marty needs a box with A, B, C, D, E, and F. This puzzle is similar to puzzle #14 but the same technique is now applied to letters rather than numbers.
Puzzle #23 - Marty needs a box with A, B, C, and so on up to Z. This puzzle builds upon puzzles #15 and #22. Like #15, you needs to train a robot to do one step in such a way that when he's done he'll be ready to do the next step. The equivalent textual program is:

```
repeat 20
    if (sizeOf(box) = 3 and box[0] = 1 and sizeOf(box[1]) = 1 and isText(box[1][0])) then
        temp1 := copy(box[1]);
        box[1][0] := box[1][0] + copy(box[0]);
        temp2 := box[2];
        temp2 := concat(temp2,temp1);
        box[2] := temp2;
    endif;
```

Puzzle #24 - Marty needs a box with a, b, c, and so on up to z. This puzzle is trivial because the robot that was trained in the previous puzzle can create the box needed here. All the robot needs, is to be given a different box to begin with. A robot is what is called *a procedure* by computer scientists. And *this procedure* has *one argument* - the box that is given to the robot. A very important aspect of a procedure is that it can be reused by *passing* it different *arguments*.
Puzzle #25 - Marty needs a box with the alphabet that is small enough to see all at once. This puzzle introduces a new tool - Pumpy The Bike Pump. Pumpy changes the size of things. This is useful in 'Free Play' for managing screen real estate. It is also an easy and direct way to change the size of pictures.
Puzzle #26 - Marty needs a box with a period, question mark, and comma. This puzzle illustrates that text pads are not limited to letters of the alphabet but include punctuation as well. Letters, punctuation, and special symbols (like $) are called *characters* in most programming languages. This puzzle also dramatically illustrates that the size of objects in ToonTalk doesn't influence on how they behave (their *semantics*).
Puzzle #27 -Marty needs the box with punctuation to be big enough to see its contents. This puzzle introduces a *keyboard accelerator* - in this case, a way to create a sound to call for Pumpy to jump into your hand. Keyboard accelerators are common in many different kinds of user interfaces.
Puzzle #28 - Marty needs the word "start". This puzzle introduces what computer scientists call *strings* - i.e. *sequences of characters*. To solve this puzzle you need to discover that letters and strings can be concatenated to form longer strings.
Puzzles #29 and #30 - Intentionally left blank.

Level #3 - Mathematics of Time

Puzzle #31 - Marty needs a box with your birthday. To solve this puzzle, you needs to discover that the keyboard can be used to change the value of strings and numbers. This is a kind of *direct manipulation of data*.
Puzzle #32 - Marty wants to know the number of minutes in a day. This puzzle illustrates that repeated addition is multiplication. The equivalent textual program is:

```
repeat 24
```

```
if (sizeOf(box) = 2 and isNumber(box[0]) and isNumber(box[1])) then
    box[1] := box[1] + copy(box[0]);
endif;
```

Puzzle #33 - Marty wants to know the number of seconds in an hour. The robot from puzzle #32 is re-used here, as the only difference is how many times the instructions should be repeated.

Puzzle #34 - Marty needs a box showing a set of scales that shows two numbers are the same. This puzzle introduces scales which are a way of expressing *numeric comparisons*. It also introduces the use of negative numbers for subtraction.

Puzzle #35 - Marty needs another box with a scale showing that two numbers are the same. This puzzle differs from the previous, because now a robot is doing the work. What is important to note here is that the robot stops when the scale is no longer titled to the left. This example shows how to use *a comparison predicate in a conditional*. Since this robot has a wand with unlimited magic, he stops only when the box no longer matches the box in his thought bubble. This is what computer scientists call a *while loop*. The textual equivalent is:

```
while (sizeof(box) = 4 and box[1] = '>' and isNumber(box[0]) and isNumber(box[2]) and
isNumber(box[3])) then
    box[2] := box[2] + copy(box[3]);
endwhile;
```

The expression "box[1] = '>'" is unusual. In this case it is equivalent to the more common "box[0] > box[2]", since the scale is always displaying the relationship between its neighboring data.

Puzzle #36 - Marty needs a box with 24 zeros. This puzzle combines puzzles #13 and #35. It shows how to use the *while loop* like a *for statement*. This is an important technique for repeating something a number of times when you don't know how many times it will need to be repeated until the program runs.

Computer scientists analyze programs to find *invariants*. These are relationships that hold after every cycle. Here there is invariant that the first hole of the box is a number which is the size (i.e. number of holes) of the box in the sixth hole. The textual equivalent of the robot is:

```
while (sizeof(box) = 6 and box[1] = '<' and isNumber(box[0]) and isNumber(box[2]) and
isNumber(box[3])) do
    temp1 := copy(box[4]);
    temp2 := box[5];
    temp2 := concat(temp2,temp1);
    box[5] := temp2;
    box[0] := box[0] + copy(box[3]);
endwhile;
```

or equivalently:

```
if (sizeof(box) = 6 and box[1] = '<' and isNumber(box[0]) and isNumber(box[2]) and isNumber(box[3]))
then
    for (; box[0] < box[2]; box[0] := box[0] + copy(box[3])) do
        temp1 := copy(box[4]);
        temp2 := box[5];
        temp2 := concat(temp2,temp1);
        box[5] := temp2;
```

endfor;

endif;

Puzzle #37 - Marty wants to know how many hours there are in a year. This is similar to puzzle #32 except that now, you are training a more generally useful robot. That is because this robot computes the product of the first 2 numbers and keeps it in the sixth hole. The *invariant* in this program is that the number in the sixth hole is the product of the numbers in the first and fourth holes. The robot stops when the numbers in the second and fourth hole are the same. The textual equivalent of this is:

while (sizeof(box) = 6 and box[1] > box[3] and isNumber(box[0]) and isNumber(box[1]) and isNumber(box[3]) and isNumber(box[4]) and isNumber(box[5])) do

box[5] := box[5] + copy(box[0]);

box[3] := box[3] + copy(box[4]);

endwhile;

Puzzle #38 - Marty wants to know how many seconds are in a day. To solve this puzzle, you use the robot trained in the previous puzzle to multiply 60 times 24 resulting in 1,440. The player then uses the robot again to multiply 60 times 1,440. This can be done by placing the 60 in either the first or second hole and the 1,440 in the other. The robot will compute the product correctly in either case but it will be much faster if the 1,440 is in the first hole. This happens, because the robot multiplies by repeatedly adding the first number. The amount of work the robot must do is proportional to the number in the second hole. In computer science terminology, we say *the program's complexity is linear with the second argument.*

Puzzle #39 - Marty wants to know how many seconds are in a year. Like the previous puzzle, this one repeatedly uses the robot trained in puzzle #37 to compute 365x24x60x60.

Puzzle #40 - Intentionally left blank.

Level #4 - Computing the Time

Puzzle #41 - Marty needs the days of the week. This puzzle introduces birds and their nests. To a computer scientist a bird and her nest is *a communication channel*. A bird is *the right or capability to send message*s *on a channel* and her nest *the right to receive messages on that channel*. This example also illustrates that messages in ToonTalk are *queued* in a *first-in first-out fashion*.

Puzzle #42 - Marty needs a box with a nest with integers starting from 2. The robot trained in solving this puzzle is what computer scientists call a *generator*. Here the robot generates *a stream of integers*. This generator is *an infinite generator* since it doesn't stop. The textual equivalent of this robot is:

while (sizeOf(box) = 3 and isNumber(box[0]) and isNumber(box[1]) and isSendCapability(box[2])) do

transmit(box[2],copy(box[0])); *// transmit a copy of box[0] on the channel of box[2]*

box[0] := box[0]+copy(box[1]);

endwhile;

Puzzle #43 - Marty needs the sum of the numbers in the nest. Here you train a robot to be a *consumer*. Many *stream-oriented programs* involve *consumers* and *generators*. This puzzle illustrates how to *receive messages*. The textual equivalent is:

while (sizeOf(box) = 2 and isNumber(peek(box[0])) and isNumber(box[1])) do

// we "peek" at the communication channel in box[0] to see if a number is there

box[1] := box[1] + receive(box[0]);

// "receive" removes the top element in the queue and returns it

endwhile;

Puzzle #44 - Marty needs a box with 3 numbers that aren't changing. Here, you are introduced to sensors. A sensor is updated on every cycle so it displays the most recent value of what it is sensing. What this sensor is sensing is intended to be a mystery until puzzle #48. This puzzle teaches a ToonTalk programming technique of "freezing a sensor" by dropping it on a zero.

Puzzle #45 - Marty needs a box with a nest full of numbers that aren't changing. This puzzle combines puzzles #42 and #44 to produce *a stream of sensor values*. The textual equivalent is:

```
while (sizeOf(box) = 3 and isNumber(box[0]) and isSendCapability(box[1]) and box[2] = 0) do
    temp1 := copy(box[2]);
    temp1 := temp1 + copy(box[0]);
    transmit(box[1],temp1);
endwhile;
```

Puzzle #46 - Marty wants to know the sum of the numbers in the nest. The robot trained in puzzle #43 works fine with this stream of numbers.

Puzzle #47 - Marty wants the number that results from waiting 8 seconds. Here you run the robot from puzzle #43 and the robot from puzzle #45 *in parallel*, in other words, at the same time.

Puzzle #48 - Marty wants the number that results from waiting 14 seconds. To solve this puzzle, you need to *spawn a new process* by loading the truck with a robot and box. This is the preferred method of running programs in parallel - doing it within a single house is harder to control and gets messy quickly. Here you will learn that the sensor measures the number of milliseconds since the last ToonTalk cycle. Hence the sum of the numbers measures how much time has passed. If you left for exactly 8 seconds on puzzle #47 the number would have been 8,000. Here if you went away for exactly 14 seconds the number would be 14,000. The reason you need to get out of sight is that the birds slow down for your benefit so you can watch them. Slow birds interfere here with measuring time accurately.

Puzzle #49 - Marty wants you to get rid of the other house. To solve this puzzle you need to discover how to *terminate a process*.

Puzzle #50 - Marty wants the results from waiting 10 seconds. This time you need to train a single robot to do what the robots in puzzles #47 and #48 accomplished. Programmers often make a special program that does the same as the combination of two general programs. They do this because the computer needs to do fewer steps to run the special program. Here the generator and consumer processes can be combined into a faster and much simpler process. The textual equivalent is:

```
while (sizeOf(box) = 2 and isNumber(box[1]) and isNumber(box[2])) do
    box[1] := box[1] + copy(box[0]);
endwhile;
```

Puzzle #51 - Marty needs the box after the 3 second timer goes off. This puzzle combines puzzles #35 and #50 to measure time until some comparison is no longer true. The reason the box has 6 holes rather than 4 becomes apparent later. Here is the textual form:

```
while (sizeOf(box) = 6 and isNumber(box[0]) and isNumber(box[1]) and isNumber(box[3])and box[1] <
box[3]) do
    box[3] := box[3] + milliseconds_since_last_cycle();
endwhile;
```

Puzzle #52 - Marty needs -10. This puzzle illustrates the repeated use of -1 to *decrement a counter*. It is equally valid to view this as adding a negative number or subtracting a positive one. The textual equivalent is:

```
while (sizeOf(box) = 3 and isNumber(box[0]) and isNumber(box[1])) do
    box[0] := box[0] + copy(box[1]);
endwhile;
```

Puzzle #53- Marty wants the secret word. This puzzle introduces teams of robots. When a box is given to a robot and that box doesn't match the box in the robot's thought bubble then the robot will leave the box for the next robot in the team, if there is one. In procedural programming languages, this corresponds to what computer scientists call *if-then-else statements*. In object-oriented programming languages, a team corresponds to *the behavior of an object* where each robot corresponds to *a method*. In logic programming languages, a team corresponds to *a predicate* and each robot to *a clause*. The team constructed in this puzzle is equivalent to the following procedural textual program:

```
procedure team(Box box)
```

```
    if (sizeOf(box) = 3 and box[0] = 0 and box[1] = -1 and box[2] = 'a') then

        run_secret_procedure(box);

        team(box);

    else if (sizeOf(box) = 3 and isNumber(box[0]) and isNumber(box[1])) then

        box[0] := box[0] + copy(box[1]);

        team(box);

    endif;
endprocedure;
```

Puzzle #54 - Marty wants an alarm clock. Here the robot trained for puzzle #51 is used together with the one trained here to implement a message send delayed by a number of seconds. The equivalent textual program is:

```
procedure team(Box box)
    if (sizeOf(box) = 6 and isNumber(box[0]) and isNumber(box[1]) and isNumber(box[3])and
    box[1] < box[3]) then

        box[3] := box[3] + milliseconds_since_last_cycle();

        team(box);

    else if (sizeOf(box) = 6 and isNumber(box[0]) and isNumber(box[1]) and isNumber(box[3]) and
    isSendCapability(box[4]) and isText(box[5])) then

        transmit(box[4],box[5]);

        team(box);

    endif;
endprocedure;
```

If you don't vacuum away the scale in the thought bubble of the new robot then we won't work in the rare case where box[1] = box[3]. The odds of this happening are 1 out of the average cycle duration which ranges between 10 and 100 depending upon the speed of the computer involved.

Puzzles #55, #56, #57, #58, #59,and #60 - Intentionally left blank.

Level #5 - Building a Clock

Puzzle #61 - Marty wants a number that keeps getting bigger. The solution to this puzzle is the same as puzzle #50. Here, however, the number being changed has a special property so that changes to it show up on the other number as well. This is what computer scientists call *shared state*. Shared state is known to cause problems in concurrent programs. In ToonTalk, however, remote controls work only within a single house so these problems are avoided.

Puzzle #62 - Marty wants the number to increase by 1,000 every second. The solution to this puzzle requires placing the robot and his box on the back of the number. Many software development systems provide user interface objects that can have programs associated with them. In ToonTalk, you can put robots on the back of pictures to give them any behavior you program. The reason that the solution to this puzzle keeps time correctly, while the solution to the previous one didn't, is that in the previous puzzle the robot was going slow so you could observe him.

Puzzle #63 - Marty wants a number that increases by 1 every second. The solution to this builds upon the programming techniques used in level #4. One of the *invariants* of this robot is that the number in the third hole should be 1 less than 1,000 times the number in the fourth hole. Another *invariant* is that the fourth number will be 1/1000th of the value of the first number. The textual form of this robot is:

```
while (sizeof(box) = 6 and box[0] > box[2] and isNumber(box[0]) and isNumber(box[2]) and
isNumber(box[3]) and isNumber(box[4]) and isNumber(box[5])) do
    box[2] := box[2] + copy(box[5]);

    box[3] := box[3] + copy(box[4]);

    update_display(box[3]); // update the display to show the new value of box[3]
endwhile;
```

Puzzle #64 - Marty wants a number that increases by 1 every minute. This puzzle reuses the robot from the previous puzzle. The numbers have been changed so that the robot ensures that the fourth number is 1/60th of the first number.

Puzzle #65 - Marty needs a box with hours, minutes, and seconds timers. Constructing the hour timer is the same as constructing the minute timer in the previous puzzle.

Puzzle #66 - Marty wants the seconds timer to go back to 0 when it reaches 60. After the robot runs the first number will be the remainder after dividing what was there by 60. Computing the remainder of the division is implemented by repeated subtraction. It is worth noting that this process is *running in parallel* with the earlier constructed process that makes the number be 1/1000th of the milliseconds timer. This robot is equivalent to:

```
while (sizeOf(box) = 4 and box[0] > box[2] and isNumber(box[0]) and isNumber(box[2]) and
isNumber(box[3]) do
    box[0] := box[0] + copy(box[3]);
endwhile;
```

Puzzle #67 - Marty wants the minutes timer to go back to 0 when it reaches 60. The solution to this puzzle requires noticing that the same robot is needed here as with the previous puzzle and that you can use the magic wand to copy the needed robot.

Puzzle #68 - Marty wants the hours timer to go back to 0 when it reaches 24. The solution to this puzzle is the same as the previous puzzle and additionally you must change the numbers so that the robot behaves correctly. This tests your understanding of the previous two puzzles.

Puzzle #69 - Marty wants the digital clock to show the right time. To solve this puzzle, you simply *initialize* the values of the timers to the current time.

Puzzle #70 - Marty wants a nicer looking digital clock. Most modern software not only needs to compute correctly, but it also should display information to users in an appealing and effective manner. Here, you need to make the clock look more attractive.

Puzzle #71 - Marty wants a box showing the time of your birth. This is needed for the next puzzle.

Puzzle #72 - Marty wants to know how many seconds until you are a round number of millions of seconds old. Here, you should drop the number showing your age in seconds on the zero to freeze the number. If you try to subtract with the timer then the number continues to get larger even when it is negative. Neglecting to do so will only affect the answer by a few seconds.

Puzzle #73 - Marty wants to know how many minutes until you are a round number of millions of seconds old. This puzzle illustrates how ToonTalk, like most programming languages, can perform division *primitively*. Since the ship's computer was broken before you couldn't directly multiply or divide numbers and had to program those operations using addition and subtraction.

Puzzle #74 - Marty wants to know how many days until you are a round number of millions of seconds old. Here, you need to type the division operation rather than use a pre-defined one.

If you have reached the end of the puzzle game, you should have learned enough to build a wide variety of programs in 'Free Play'. You can learn more by watching some of the demo movies ('See Demos'). Please share what you build with others.