COMS W4115 Tandem Tutorial

Team 11

3/21/12

Abdullah Al-Syed, aza2105

Jenee Benjamin, jlb2229

Patrick De La Garza, pmd2130

Donald Pomeroy, dep2127

Varun Ravishankar, vr2263

An Introduction to Tandem

State machines are fundamental to how we program, from using regular expressions to creating simulations. However, it is often difficult to create state machines without creating spaghetti code. State machine code is riddled with 3 or more nested levels of if-else statements, calling functions that might be anywhere in the code. While libraries have attempted to solve this problem, most programmers prefer to avoid creating state machines, even when it might be a natural description for the algorithm at hand. Tandem attempts to solve these problems by taking inspiration from finite state automata and abstract state machines by making the node a central feature in the language, and letting the programmer focus on the inputs to be handled, the outputs that are produced, and transitions to the next state. This leads to a easy-to-read syntax for state machines and makes it easy to introduce concurrency to handle the large simulations that will be created on multicore machines in the near future.

Tandem is well-suited for simulations of any type, including hardware, networks, and physics simulations. Evolutionary programs can be described as a set of states with a feedback loop, converging on a final output; genetic algorithms and neural networks can be described with simple Tandem programs. Finally, algorithms that are best described with state machines, like the error recovery in TCP network programming, are awkward to program in most languages. Tandem, however, can handle the coupling between chains of states without confusing the programmer about the transitions between states. If a program can be described as a series of transitions from one state to another, it is suited for Tandem. Programmers will find that using Tandem produces easy-to-read and maintainable code without forcing them to use abstractions that are not natural to the problem.

Getting Started

This tutorial provides a brief introduction to the Tandem programming language. It will focus on demonstrating the basics of the language: nodes, expressions, conditionals, loops, etc. It will also demonstrate how to use a few of the most basic built-in functions. In the last section of the tutorial, we will examine a few complex programs that demonstrate how Tandem is used as state machines and for simulations.

Hello, World!

Below is the most simplistic way to write a program in Tandem that prints the words Hello, World!:

```
print Hello, World!
```

This code must be written in a program in a file whose name ends in .td, such as hello.td. Then, we compile it with our Tandem compiler:

```
$ tdc hello.td
```

Print is one of Tandems built-in system nodes, so we can simply call this node to print the desired text. A node is Tandems fundamental trait which makes it possible to create objects that have some behavior. Literal input to a node always immediately follows the calling of the node. (We will formally go into what defines a literal later in the tutorial.) The input to print is *Hello*, *World*!, and when we called print, it acts on this input to print the text to the output channel. *Print* automatically terminates the text with the escape symbol. Therefore, this program will simply print the text to the screen:

```
$ tandem ./hello
Hello, World!
$
```

Below is an alternate hello world program:

```
node hello()
        print Hello, World!
end

hello
```

In this program, we defined a node called *hello*. *hello* simply calls the *print* node to output the text. We always define a node with the keyword **node**, then the name of the node. In this example *hello* is the name of the node. The parentheses following the name of the node allows a user to pass parameters to the node. Parameters (also known as input variables) can be manipulated or used inside the node definition to perform some behavior. In this program, *hello* does not take in any parameters, so there is nothing within the parentheses. After we call the *print* node inside the definition, we denote the end of the node with the keyword **end**. Every node definition must signal its completion with the **end** keyword.

The last action of the program is to call the node that we just defined. Since *hello* takes no input, simple writing hello will call the node to print *Hello*, *World*. Then the program terminates.

Also, it is recommended to keep your code tidy by using the proper indents, so it can be easier to see where one node definition begins and ends - this becomes more useful when we start nesting nodes. Nonetheless, proper spacing is not necessary for compiling the code.

Below is a more intricate version of the hello world program:

```
node hello2()
        node myWords(text)
                return text
        end

        myWords Hello, World! | print
end

hello2
```

Above, we have demonstrated the ability to define nodes within nodes.

Now, the node *hello*2 is defined by an inner another node called *myWords* and the following print statement.

We also demonstrated the ability to pass input to nodes.

*myWords*is a node that has a input variable. We use the variable *text* to represent the input that *myWords* accepts. In the node definition of *myWords*, we simply return the value of the parameter passed. The return value of a node is the nodes output. The output of *myWords* is therefore text.

In the last line of the *hello*2 node definition, we call the node that was just defined with the input Hello World. *myWords* returns Hello, World! as its output. The pipe after this node call indicates that the output of the node to the left of the pipe will be input to the node on the right. Node inputs (as opposed to literal inputs, which always immediately follow the node call) are always denoted in this fashion. Therefore *Hello*, *World*! is input to *print*, and the *print* node will display the text. We will go into detail about pipelines later.

Another important aspect of Tandem to note is that the **return** keyword is optional. No matter what, the last expression of the node definition would be what the node returns, and pipelines are expressions!. If a node does not have any expressions, the node will return *null* (print Hello, World! is an expression that returns *null*).

Variables, Literals, Comments

In the previous section, we introduced variables as inputs to nodes. We can also define variables in any part of a program - outside a node or inside a node - and variables may or may not be assigned a value. Using the variable age, the following program prints a number to the output channel.

```
node n()
                node myAge()
                age = 21
        end

        myAge | print
        # prints the output of myAge
end
```

= is the assignment operator and assigns the value on the right of the equals sign to the variable on the left. In this case, we assign the number 21 to *age*. By default, in Tandem, a number without any decimal value or exponential value following it is an integer, or an int. Therefore 21 is an int. More specifically, 21 is an integer literal because it represents a fixed value. Similarly, in our first few hello world programs, *Hello, World*! was a String literal because the text represented its own fixed value.

Variables in Tandem are assigned types based on the values that they are given; therefore in assigning age to be 21, we assign the type of age to an int. Variables can change values at any time, and a variable can represent any single type at a given moment. For example, we can reassign age to be 21.0, which is a double type. We can even reassign age to be *hi*, which is a string type.

Another important aspect of variables are their scopes. Because we define age inside the *myAge* node, it cannot be seen by any nodes or statements outside myAge. So, after the **end** keyword, the scope of *age* is finished and the node n will not know what *age* is.

The last line before the node *n*s end is initiated by a hash symbol, #. This denotes a comment. The rest of the line following the # is not code to be compiled and can contain any notes that the user wants. It is recommended to put comments in code to make it easy to understand to readers of the code what different parts are doing.

To explore more kinds of literals in Tandem, lets take a look at another sample program:

```
node sample()
        node makeList()
                a = [1,3,5,7,9,9]
                element = a[0] # element is equal to 1
        end

        node makeSet()
                a = {0,2,4,6,8}
                anotherElement = a[3] # anotherElement is equal to 6
        end

        finished = true
end
```

The *makeList* node returns *a*, which is a list of odd single digit positive integers. (Remember from before that the **return** keyword is optional, and a node always returns the value of the last expression.)

As shown in the program, a list is declared by an identifier followed by a single equals sign followed by an open square bracket, followed by any number of literals separated by commas, or a range operator, followed by a close square bracket. Literals do not need to be unique, and we can have any number of elements more than once in the list. For example, 9 appears twice in our list *a*.

The *makeSet* node returns a, which is a list of even single digit positive integers. (As mentioned before, the scope of the a in the *makeList* node is of that node, so the a that we define here in *makeSet* is completely different.) As shown, a set is declared by an identifier followed by a single equals sign followed by an open curly bracket , followed by any number of unique literals separated by commas, followed by a close curly bracket.

Accessing individual elements of lists, as we do in the lines following each types declaration, is simple. We access the element using its index number, which is surrounded by brackets next to the list or set. For the index of list or set, a positive index from 0 to the arrays size - 1, inclusive, returns the corresponding element of the list. Negative values are also allowed, and indices go from $-$size to -1. Anything outside of these ranges returns null.

In Tandem, booleans are either $true$ and $false$. In the sample node, we created the variable finished with a boolean value of $true$. *sample* will return $true$ as it finishes.

Arithmetic Expressions

In this next part, we will demonstrate how arithmetic operations are performed in Tandem.

The following program:

```
2+3
```

is a valid Tandem program, and will return the value 5. The two literals 2 and 3 are evaluated as ints and are added - one of the basic operations that Tandem can perform (along with subtraction, multiplication, division, exponentiation, modulo, shift right, shift left, XOR, AND, OR)

Now, lets look at the following program:

```
node sillyMath(x, y)
        temp = x + y
        temp -= y
        answer = temp - x
end
```

This node always returns the value of *answer*, which will always be 0 when numbers are passed in. This program demonstrates a few arithmetic operations and assignments in Tandem. We have two inputs to the node $sillyMath$: x and y, separated by a comma in the node declaration. We create the variable *temp* is the sum of x and y and whose type defaults to the type of x or y. If x or y is a list, however, this can produce the empty list. Just be careful and remember that people may not pass in the types that you expected. You should always document what types you expect, but write code that is flexible enough to work with multiple types.

The next expression is $temp- = y$, which is a shorthand expression for $temp = temp - y$. This line subtracts y from the current value of *temp* and assigns the difference to the value of *temp*.

Any operation with the format <operator>= does performs the operation in a similar manner: i.e.

$num\ <operator>= num2$means $num1 = num1\ <operator>\ num2$.

The last line creates the variable *answer* and its value is *temp* minus x. It is implicit that $sillyMath$ returns answer.

Note how we never defined the types of x and y. Therefore, nothing stops sillyMath from having ints and doubles as inputs. The only requirement is that the inputs must be able to perform the operations that are in the node definition.

Among other arithmetic operations that Tandem can perform are multiplication ($*$), division (/), modulo (mod, %), exponentiation ($**$),bitwise complement ( ), bitwise shift-left ($<<$), bitwise shift-right ($>>$), bitwise and ($/\backslash$), bitwise xor ($\hat{}$), bitwise or ($\backslash/$).

Other than arithmetic expressions, relational and logical expressions are very important is manipulating data in Tandem. We will briefly go over their usage next.

Relational and Equality Expressions

Relational expressions are used to perform comparisons between variables, and as such, evaluate to a boolean value. Equality expressions check for equality between two variables and also evaluate to a boolean value. There are several operators that can be used inside a relational expression: <, <=, >, >=. For equality expressions, we can use == or ! =. An example with an int is illustrated below:

x >= 2

The above expression compares the magnitude of the variable with 2, and returns true if x is greater than or equal to 2; otherwise, it returns fails.

The types of the variables being compared should be the same. For instance, comparing a string and a double using one of the relational or equality operators above will cause a compile-time error.

Take note that these operators are overloaded for sets: they return whether the compared sets are supersets or subsets, correspondingly.

Logical Expressions

Logical expressions take the following form

boolean [operator] boolean

The operator can be one of: && (Boolean AND), || (Boolean OR), both of which have higher precedence compared to the low precedence Boolean operators: **not**, **and**, **xor**, **or**.

A boolean can be *true*, $false$ or a variable/expression that evaluates to a boolean value.

Consider the following examples:

```
true AND false # this evaluates to true.
true and false || true
# This also evaluates to true because || has higher precedence than and.
```

Conditionals

Conditionals can be specified in a number of ways. A conditional can be nested inside another conditional, with the keyword **end** denoting the end of each conditional block. Three examples illustrating the use of conditionals are included below:

Example 1:

```
if expression
        statement1
else
        statement2
end
```

Tandem does not allow for an **if** statement without the accompanying **else** statement. The expression should return a boolean value, which if true, will cause *statement*1 to be executed; otherwise, *statement*2 will be executed.

Example 2:

```
unless expression
        statement1
end
```

An **unless** should be understood as an if not. As before, expression returns a boolean value. If false, this will cause *statement*1 to execute. If true, *statement*1 will not execute.

Cond expressions are fundamentally like switch/case statements in C-like languages. They evaluate a condition, and if the condition is true, evaluate the corresponding statements. There are implicit breaks between each condition, so only the first condition that is found to be true will run. Finally, you can specify a default case by making sure your condition always evaluates to true.

Example 3:

```
cond
        condition1
        # Execute this code and return if condition1 is met
        # do not deal with the other two
        end

        condition2
                # return this one
        end
```

```
        condition3
                # return this one
        end

        true
                # return this one
        end
end
```

Here, if *expression*1 evaluates to true, *statement*1 is executed; otherwise, if *expression*2 evaluates to true, then *statement*2 is executed. The same holds for *condition*3. If none of these are true, the last condition, *true*, returns true, and will run as the default condition.

Loops

In Tandem, we can construct loops in the nodes with the keywords for and while. Loops can be very useful when we need nodes to perform the same operations on different data values. Lets say we have a file called fib.td that contains the following code:

```
node fibonacci(input)
        node iterative(number)
                prev1 = 0
                prev2 = 1

                for x in 0..number
                        savePrev1 = prev1
                        prev1 = prev2
                        prev2 = savePrev1 + prev2
                end

                return prev1
        end

        iterative input
end
```

In the node *iterative*, which is a subnode of *fibonacci*, we have a **for** loop. This is only one of different ways to create a for loop. In this case, we want to iterate through the numbers from 0 to the value of *number* and perform whatever is in the for loops body until we iterate passed the value of *number*.

Loops can take many other formats. Below are other types of loops and the format in which they are used:

Generic loops are done as follows:

```
loop
```

```
        # do things
        # possibly (hopefully) break under some condition
end
```

For-loops can be done as follows:

```
for item in group
        # do stuff here
end
```

While loops are used to loop as long as condition holds:

```
while condition
        # do stuff
end
```

A do-while-loop:

```
do
        # do things
while condition
end
```

The difference between a while loop and a do-while loop is that the while evaluates its condition first, and then proceeds to the loop body, while the do-while loop runs the loop body, and then checks the corresponding condition.

Until-Loops are like "while not" loops. They loop until some condition is met, at which point they stop.

```
until condition
        # do stuff
end
```

Recursion

As mentioned before, a node can have any number of subnodes. In our fib.td program, we can add another node called recursive. This node will perform the fibonacci sequence recursively and will demonstrate the capability of a recursive loop in Tandem.

```
node fibonacci(input)
        node iterative(number)
                prev1 = 0
                prev2 = 1

                for x in 0..number
                        savePrev1 = prev1
                        prev1 = prev2
                        prev2 = savePrev1 + prev2
```

```
                end

                return prev1
        end

        node recursive(number)
                if number < 2
                        return number
                else
                        (recursive number-1) + (recursive number-2)
                end
        end

        iterative input
        recursive input
end
```

We previously demonstrated the ability to call nodes within nodes with *print*. Whereas *print* is a Tandem built-in node, *recursive* is a node that we defined in the program. We call *recursive* on input $number - 1$ and also call *recursive* on input $number - 2$. We add these two values and *recursive* returns that sum. The program loops and loops on the input until the base case condition is met.

Pipelining

As mentioned previously, Tandem is a state machine language. Outputs of one node are fed as inputs to other nodes. We perform this with pipelines- the value of the node on the left of the pipe is always input to the node on the right of the pipe. The pipes represent the transition of states with the nodes input as the previous nodes output. Consider the following program.

```
from "fib.td" import fibonacci.*

public node N()
        node f(x)
                x+1
        end

        cond
                a > 0
                        f a | recursive | print
                true
                        g a | iterative | print
                end
        end
```

```
        node g(x)
                x
        end
end
```

Before we discuss the pipeline, lets draw our attention to the first line of the code where the import utility is being used. Imports allow you to access nodes form other files or libraries. You can import every node in a file in one of these two ways:

```
import filename
from filename import *
```

Using *, our example imports all the nodes in fibonnacci, allowing us to use recursive and iterative.

The pipelines in lines 8 and 10 make use of the imported functions. Remember from the fib.td program where *recursive* and *iterative* were defined, that both nodes took in a number as their input. The input to this node call comes from the output of the node to the left of the pipeline. In this same way, the input to the print node call comes from the output of *recursive* (or *iterative* in line 10).

Remember that pipelines are used to connect the output of *nodes* to the input to other nodes. If a node is manipulating a literal, the standard [node] [input] format can be used.

More Complicated Uses of Tandem

We have provided two sample programs here. The first models a 4-bit shift register, a hardware device that in real life consists of a cascade of flip flops, which has the output of any one but the last flip-flop connected to the data input of the next one in the chain. This is useful when storing a series of values and then extracting them one-at-a-time, and was used heavily when computers were first being built to store data.

```
node bit0(d)
        cond
                d = 0
                        return 0
                end

                d = 1
                        return 1
                end
        end
end
```

```
node bit1(d)
        cond
                d = 0
                        return 0
                end

                d = 1
                        return 1
                end
        end
end

node bit2(d)
        cond
                d = 0
                        return 0
                end

                d = 1
                        return 1
                end
        end
end

node bit3(d)
        cond
                d = 0
                        return 0
                end

                d = 1
                        return 1
                end
        end
end

bit0 1 | bit1 | bit2 | bit3
```

The next program is a bit more complicated, but only uses knowledge discussed in this tutorial. Try and see if you can understand it, and feel free to refer back to the tutorial or the Language Reference Manual for more details on syntactic constructs.

The following program takes an analog signal and encodes it in the time domain the same way a neuron does.This can be really useful. With the implementation of fork in the future, can feed the signal through a gamma-tone filter-bank and encode it simultaneously with several neurons. This is great for

modeling the human ear.

```
from Math import sin abs min

#This function encodes an analog signal as an integrate and fire neuron would
#The signal is a list of values of the same length as the list of times.
#dt specifies the distance between each time, bias is used to make the
#signal have a posotive integral
#threshold is the threshold value at which the neuron will output a spike
#and reset it's value
node IAF_encode(signal,times,dt,bias,threshold)
        y=[] #integral of the signal over time
        spike=[] #records spike times

        y[0]=dt*(signal[0]+bias) #set the initial value of the integral

        spikecount = 0 # will keep track of the number of spikes produced and
        for time in times
                unless time is 0.0
                        i =  time / dt
                        j = i - 1
                        y[i] = y[j] + dt * (signal(i) + bias)
                        #Here we approximate the integral for each time step

                        #if the threshold is passed, then we make a spike
                        if y[i] >= threshold
                                spike[spikecount] = time
#store the spike data
                                y[i] -= d
#reset the variable
                                spikecount += 1
#increment the spike index
                        else
                                continue
                        end

        end
        end

        return spike #return the spike times/time-encoded signal
end

dt= 1 / 1000000 #seconds
duration = .25 #seconds
times=none
t=0
```

```
#assign the times
do
        times[t]=t*dt
        t += 1
while (t*dt) < duration
end

Bandwidth = 25
#create bandlimited signal
signal = 2 * Bandwidth * (sin (2 * Bandwidth * times))
bias = abs signal | min

#determined experimentally
threshold = 0.007

#print the spiketimes that represent the signal
IAF_encode signal times dt bias threshold | print
```