COMS W4115 Project Report

Team 11

5/5/12

Varun Ravishankar, vr2263, Project Manager

Patrick De La Garza, pmd2130, Language Guru

Jeneé Benjamin, jlb2229, System Architect

Donald Pomeroy, dep212, System Integrator

Contents

Part I

Introduction

1 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 three 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.

2 Main Features of Tandem

Tandem is based upon the concept of the node, representing a state that the program is in. Nodes can be linked together to form chains of nodes, which in turn can be led to be thought of as nodes. This leads to easy modularity and object-oriented programming. Tandem lends itself to parallel programming, with no data being shared between nodes and easy simulation of non-deterministic state machines leading to massively concurrent programs.

2.1 Nodal

The Tandem Language is nodal. This is Tandem's fundamental trait and makes possible the language's capabilities. In Tandem, everything is a Node. Tandem code can be thought of as a directed graph of functions or states (nodes). Every program written in Tandem is itself a node, even if it is composed of a larger chain of nodes. The nodal logical-structure underlying Tandem makes possible a combination of traits that allows the user to comfortably tackle a wide variety of problems and work in a paradigm that is conducive to easy and innovative development.

2.2 Object-oriented

Tandem is an object-oriented language. Each node in Tandem forms an object with its own set of variables and functions hidden from other nodes that it can call on to process its input. Nodes can also be linked into a collection of interconnected nodes, or a super node, that can also be treated as a single node object that takes a set of inputs, processes them, and produces some output. Only the output of the nodes operation and the users input is ever passed from one node to another. Tandem has a built-in sense of encapsulation and polymorphism, enabled by the use of Java, that allows the user to make re-usable and extendable code.

2.3 Modular

Tandem allows the user to write modular programs. Every node represents a separate function and contains all of the information needed to execute that function. This allows for compartmentalization and separation of concerns. This makes code in Tandem very reusable and allows for large projects to be broken down into more manageable sub-projects where a team working on one sub-project needs to know very little about what other teams are working on. Additionally, developers can create libraries by combining nodes to form super nodes, and users can use these libraries by considering the inputs each super node takes and possible outputs that it can produce. The end result is that users can use modules written by others without worrying about the minute manipulations going on inside an individual state.

2.4 Immutable

Tandem heavily emphasizes immutability. All data in a node is encapsulated within that node, which means nodes cannot share any state. This requires a shift in the programmer's mindset, but allows a developer to not have to worry about other nodes affecting the node at hand. While variables internal to a node may be re-assigned, Tandem does not let users create global variables that can later be manipulated. This means that the programmer does not need to worry about mutability changing any data structures and instead can focus on all possible inputs that may be received and any possible outputs that should be sent to the next state.

2.5 Massively Parallel

Tandem programs allow for massively parallel programs written naturally. As systems are developed with more and more cores, it becomes important to be able to program on each of these cores without having to worry about threads and locks, and the parallel nature of Tandem allows for this. Multiple transitions should be allowed for the same output, with non-determinism being simulated by threads. Nodes can quickly branch out to simulate all the different possibilities for an algorithm, and immutability means that nodes do not have to worry about inadvertently creating race conflicts or altering some shared mutable state. The ease with which Tandem will run these concurrent and completely separate processes on a system makes the language support massively concurrent algorithms, and in fact makes programs embarrassingly parallel. Finally, because each thread can run at the same time with its own copies of data, and because there is no communication between the processes, Tandem allows for the fast and safe execution of programs.

3 Practical Aspects of Tandem

3.1 Compiled

Tandem is built using Java and then compiled into intermediate code, such as Ruby code. This intermediate code is then interpreted into bytecode (using JRuby by default), which in turn is interpreted by a virtual machine, such as the Java Virtual Machine (JVM). Compiling to the JVM enables the user to attain high performance for their code, allows the user to not have to worry about allocating or freeing memory, and provides a stable and secure environment on which Tandem code can run. However, this does create the drawback that users will need to compile their results before they can see any speed benefits, and takes away the high turnover-rate of pure modern dynamic languages like Python or Ruby, where code can be run and tested quickly without having to wait for it to compile first. The inclusion of an interpreter later on can help mitigate this cost.

3.2 Dynamic Type System

Tandem is a dynamic programming language. It does not force the user to specify types statically, like in C or Java, nor do users have to worry about fixing type errors simply to satisfy the compiler. While this does introduce some complexity for us, the compiler's writers, users gain the benefit of being able to focus only on the code and nothing extraneous to the task at hand.

3.3 Architecture-Independent

Tandem is architecture-independent. The language is compiled to run upon the Java Virtual Machine when using the default JRuby backend, and thus inherits the benefits of the JVM, including architecture-independence and portability. An architecture-independent language runs the same regardless of the type of assembly instructions that an individual machine supports, and runs the same on 32-bit and 64-bit systems. Architecture-independent languages also have the benefit of being easy to port across multiple operating systems, so a programmer who writes on a Linux machine will have no problem debugging or maintaining the code on Windows or OS X. If a machine supports the JVM, then Tandem can run on that machine.

3.4 Threaded

Because Tandem consists of nodes that can transition to multiple states, we can make use of Javas multi-threaded capabilities with the Thread class to simulate the various transitions that can be made, especially to simulate non-determinism. The functionality of the class allows for threads to run concurrently without disturbing each other, which makes processes safe and stable as long as individual threads do not alter shared state. Because each node in Tandem is immutable, as a node in a thread transitions and executes some function, it operates on its own working copy of a variable. The language puts restrictions on what shared state, if any, can be accessed from within an individual node and lets us create as many threads as necessary to simulate states without worrying about corrupted memory or race-conditions.

Part II
Language Tutorial

4 An Introduction to Tandem

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.

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.

5 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.

5.1 Installation

To run the Tandem compiler, Ant, Java, and Bash must be installed on your computer. To see more see: Tandem Compiler

If you have git, you can check out the file using:

```
$ git clone git://github.com/vrdabomb5717/Tandem.git
$ cd Tandem
```

If you do not have git, go to the repository, Tandem Compiler, and click the downloads tab. Download and extract the file, and then navigate to that directory.

5.2 Hello, World!

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

Listing 1: hello.td

```
Println "Hello, World!"
```

This code must be written in a program in a file whose name ends in .td, such as hello.td. This file must be called with the complete path, as seen below. Then, we run it with our Tandem compiler:

```
$ ./tandem /home/thedonald/Tandem/test/tutorial/hello.td
Hello, World!
$
```

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.

Below is an alternate hello world program:

Listing 2: alt-hello.td

```
node Hello()
        Println "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, which must begin with a capital letter. 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!" | Println
end

Hello2
```

Above, we have demonstrated the ability to define nodes within nodes. Nodes can only be called after they are defined.

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*).

5.3 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.

Listing 3: n.td

```
node N()
                node MyAge()
                        age = 21
                end

        MyAge | Println
        # 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:

Listing 4: dsliterals.td

```
require "set"
node DSLiterals()
        node MakeList()
                a = [1,3,5,7,9,9]
                element = a[0] # element is equal to 1
        end

        node MakeSet()
```

```
                a = Set.new 0 2 4 6 8
                anotherElement = a[3] # anotherElement is
                    equal to 6
        end

        finished = true
end

DSLiterals
```

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 Set.new followed by a space separated list of elements.

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.

5.4 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:

Listing 5: sillymath.td

```
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 (/\), bitwise xor (ˆ), bitwise or (\/).

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

5.5 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.

5.6 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.
```

5.7 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.

5.8 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:

Listing 6: FirstFibonacci.td

```
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
```

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

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

5.9 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.

Listing 7: Fibonacci.td

```
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.

5.10 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.

Listing 8: pipeline.td

```
import "Fibonacci.td"

public node N()
	node F(x)
```

```
5                 x+1
6         end
7
8         node G(x)
9                 x
10        end
11
12        cond
13                a > 0
14                        F a | Recursive | Println
15                true
16                        G a | Iterative | Println
17                end
18        end
19 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 this way:

```
import filename
```

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.

5.11 More Complicated Uses of Tandem

We have provided three 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.

Listing 9: fourbitshiftregister.td

```
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 | Println
```

The next program is a bit more complicated, but only uses knowledge discussed in this tutorial. geometry.td calculates basic mathematical properties of three dimensional shapes, such as cylinders, cones, prisms, spheres, and tori. Notice that this file uses the pipeline and also uses PI, which is translated as $Math.pi$. In general, Ruby code can be included by requiring the appropriate module and calling the class within a pipeline.

Listing 10: geometry.td

```
#geometry_operations.td - returns numerical values
#for the properties of common shapes such as area and volume
#Donald Pomeroy

node Hypotenuse_length(leg1, leg2)
        temp1 = leg1**2 + leg1**2
        return temp1**(0.5)
end

node Circle_area(radius)
        return (PI * (radius**2))
end

node Circle_perimeter(radius)
        return (2*PI*radius)
end

node Square_perimeter(side_len)
        return (side_len*4)
end

node Rectangle_area(len,width)
        return (len*width)
end

node Rectangle_perimeter(len,width)
        return ((2*len)+(2*width))
end

node Square_area(side_len)
        return side_len * side_len
end

node Cylinder_volume(radius, height)
        return (Circle_area radius)*height
end

```

```
node Cube_Volume(side_len)
        return side_len ** 3
end

node Cone_Volume(radius, height)
        return (1.0/3.0)*(Circle_area radius)*height
end

node Cube_surface_area(side_len)
        return (Square_area side_len)*6
end

node Cylinder_surface_area(radius,height)
        return 2*(PI*(radius**2)) + (2*PI*r*h)
end

node Sphere_surface_area(radius)
        return 4*(Circle_area radius)
end

node Sphere_volume(radius)
        return (4.0/3.0)*(Circle_area radius)*(radius)
end

node Cone_surface_area(height,radius)
        PI * radius * (Hypotenuse_length height radius) *
            (Circle_area radius)
end

node Rectangular_prism_volume(length,width,height)
        return length*width*height
end

node Rectangular_prism_surface_area(length,width,height)
        return (2*(length*width)) + (2*(length*height))
            +(2*(height*width))
end

node Trapezoid_area (base1,base2,height)
        return (1.0/2.0)*(base1 + base2)*height
end

node Trapezoidal_prism_volume(base1,base2,baseHeight,
    height)
        return height * (Trapezoid_area base1 base2
            baseHeight)
```

```
end

node Triangle_area(base, height)
        return (1.0/2.0)*(base*height)
end

node Regular_pentagon_area(side_length)
        return (side_length**2)*1.7
end

node Regular_hexagon_area(side_length)
        return (side_length**2)*2.6
end

node Regular_octagon(side_length)
        return (side_length**2)*4.84
end

node Regular_icosahedron_volume(side_length)
        return (side_length**3)*2.18
end

node Regular_icosahedron_surface_area(side_length)
        return 8.66*(side_length**2)
end

node Torus_volume(minorRadius,majorRadius)
        return 2*PI*majorRadius * (Circle_area
            minorRadius)
end

node Torus_surface_area(minorRadius,majorRadius)
        return (2*PI*minorRadius)*(2*PI*majorRadius)
end

node Regular_tetrahedron_volume(side_length)
        return (2**(1.0/2.0))*(1.0/12.0)*(side_length**3)
end

node Regular_tetrahedron_surface_area(side_length)
        return (3**(1.0/2.0))*(side_length**2)
end
```

Try and see if you can understand this next program, 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.

Listing 11: IAFencode.td

```
#Patrick De La Garza

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

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

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

```
                                        the spike index
                        else
                                continue
                        end

        end

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

dt= 1.0 / 1000000.0 #seconds
duration = 0.25 #seconds
times=[]
t=0

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


bandwidth = 25
#create bandlimited signal
signal=[]
for index in 0..((times.size)-1)
        c=2*bandwidth*times[index]
        a=Math.sin c
        signal[index] = 2 * bandwidth * a
end
#########


bias = signal.min
bias=bias.abs

#determined experimentally
threshold = 1.4

#Get the spiketimes that represent the signal
spiketimes = IAF_encode signal times dt bias threshold

#Print the spiketimes to a file
f = File.open "spiketimes.dat" "w"
f.write spiketimes
```

```

#Print the Neuron's estimated firing rate over the
    interval
Print "Neuron firing rate:"
numspikes = spiketimes.size
rate=numspikes*4
Print rate
Println "HZ"
```

Part III

Language Reference Manual

6 Program Definition

The program consists of import statements, node definitions, which contain function and variable definitions, the main consists of code not within the body of a node definition. The main method is defined as any code (with the exception of import statements) outside of node definitions, it does not necessarily have to be written at the bottom of the file. However, it is good style to put all of the main code at the end of the file following the node definitions. Yet, if there are import statements, they must be provided before any other any code.

The basic structure of the program is as follows:

```
<<import statements>>
<<node definitions>>
<<main method>>
```

7 Lexical Conventions

7.1 Whitespace

Tokens must be separated by whitespace, which can include spaces or tabs. Newlines act as separators for statements, so they cannot generally be used as whitespace.

7.2 Comments

is a single line comment.

// is also a single comment.

Example:

```
# this is a single line comment
// this is also a single line comment
```

7.3 Identifiers

Identifiers, used for variable names, consist of strings of numbers, letters, underscores. The first character of an identifier must be an underscore or lowercase letter. The identifiers for nodes, on the other hand, must start with a capital letter.

7.4 Reserved Words

The following words are reserved:

Keywords
true
false
import
node
end
cond
public
while
for
loop
in
until
not
or
xor
break
continue
is (equality)
assert
unless
if
else
mod
null

The following are reserved but have no function in the language. They may be added in the future.

Reserved Keywords
fork
from
try
catch
finally
with
lambda
private
is
some
none

7.5 Types and Literals

All variables in Tandem are dynamically typed. The programmer does not need to specify types when he or she uses a variable, but does need to have a value assigned to a variable before it can be used. Use of variables that have not been declared will throw an error.

7.5.1 Integer

An integer, or int, is declared as follows: an identifier followed by a single equals sign, followed by zero or more digits. Integers can have sign, the range of possible integers is determined by Ruby, which converts integers to large integers when they overflow.

Example:

```
a = 42
```

7.5.2 Double

A double is declared as follows: an identifier followed by a single equals sign, followed by one or more digits, or the same followed by a single decimal point . followed by one or more digits. Doubles can have signed, the range of possible doubles is determined by the IEEE 754 standard.

Example:

```
a = 45.01
```

7.5.3 Boolean

A boolean is declared as follows, an identifier followed by a single equals sign followed by *true* or $false$. All values in Tandem are true, except for $false$ and *null*, which represents an undefined value. You should not use *null* to represent empty values; future versions of Tandem will introduce a *none* type to represent empty values.

Example:

```
a = true
```

7.5.4 Strings

A string is declared as follows, an identifier followed by a single equals sign followed by one double quotation mark followed by zero or more characters followed by a single double quotation mark.

Example:

```
a = "hello world"
```

Strings supports most of the Java escape sequences, and a few other special escape sequences.

Escape Sequence	Use
\x	Equivalent to the character x by itself, unless x is a line terminator or one of the special characters $abefnrstv$. This syntax is useful to escape the special meaning of the \,#, ′, and ” characters.
\t	The TAB character (ASCII code 9).
\s	The Space character (ASCII code 32).
\a	The BEL character (ASCII code 7). Rings the console bell.
\b	The Backspace character (ASCII code 8).
\e	The ESC character (ASCII code 27).
\n	The Newline character (ASCII code 10).
\r	The Carriage Return character (ASCII code 13).
\f	The Form Feed character (ASCII code 12).
\v	The vertical tab character (ASCII code 11).

7.5.5 List

A list is declared as follows, an identifier followed by a single equals sign followed by a square bracket, followed by any number of literals separated by commas.

Example:

```
a=[1,2,3,4,5]
```

7.5.6 Hash

A hash is declared as follows, an identifier followed by a single equals sign followed by a curly bracket, followed by any number of unique literals followed by a fat comma (=>), followed by any literal, each separated commas.

Example:

```
a = {1=>2, 2=>3, 4=>5}
```

7.5.7 Binary Literal

A binary is declared as follows, an identifier followed by a single equals sign followed a 0, the letter b, and one or more 0s or 1s. The maximum binary size and length is determined by Ruby. In most cases, this will be when Ruby runs out of stack space.

Example:

```
a =  0b10010
```

7.5.8 Hexadecimal Literal

A hexadecimal is declared as follows, an identifier followed by a single equals sign followed a 0, followed an X, followed by one or more of these character, $0, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f$ (case insensitive to the letters). The maximum size and length of the literal is determined by Ruby. In most cases, this will be when Ruby runs out of stack space.

Example:

```
a = 0xFFFFFFFF
```

2.5.9 Complex Numbers

A complex number is declared as follows, the node Complex followed by the real part, followed by the imaginary part. The maximum size is determined by Ruby. In most cases, this will be when Ruby runs out of stack space. this actually calls the Ruby Complex class, which you may do by requiring the appropriate class and then using it within a pipeline. For more information, see (11.2.1).

Example:

```
Complex 2 3 #=> (2+3i)
```

7.6 Operators

7.6.1 Operators, associativities, and precedences

Operator(s)	Associativity	Operation
{...}	N	Hash and hash comprehensions
[...]	N	List, list comprehension
(...)	L	expression
x.*attr*	L	attribute reference
x[*i*]	L	Indexing
$\vert$	L	Pipeline operator
**	R	Exponentiation
~!	R	bitwise complement; Boolean NOT
−+	R	unary minus; unary plus
∗ / %	L	Multiplication; division (true), modulo
+−	L	Addition (or concatenation); subtraction, set difference
<< >>	L	Bitwise shift-left; bitwise shift-right
/\	L	Bitwise AND, set intersection
^	L	Bitwise XOR, set symmetric difference
\/	L	Bitwise OR, set union
<<= >= >	L	magnitude comparison, set subset and superset, value equality operators
==! =	N	Equality testing, comparison
&&	L	Boolean AND
$\vert\vert$	L	.. N Range creation (inclusive)
x..y	N	Range creation (inclusive)
= ** = ∗ = / = % = + = − = <<= >>= && = $\vert\vert$ = \/ = /\ = ^ =	R	Assignment
x *mod* y	L	Modulo (low precedence)
x *is* y, x *is not* y	N	Identity tests
x *in* y, x *not in* y	N	Membership
not x	R	Boolean NOT (low precedence)
x *and* y	L	Boolean AND (low precedence)
x *xor* y	L	Boolean XOR (low precedence)
x *or* y	L	Boolean OR (low precedence)

7.6.2 Mathematical Operators

Plus Operator (+) The plus operator is a binary operator that can perform addition on doubles. On strings and lists, the plus operator acts as concatenation. If the identifier on one side of the plus expression is a string the type on the other side is automatically converted to a string and they are concatenated. The sum of an integer and a double is a double.

Example:

```
a = 45 + 45.1 #a is 90.1
a = hello +  world # a is hello world
a = 45 + hello # a is 45hello
```

Minus Operator (−) The minus operator is a binary operator that can perform subtraction on doubles. On sets the minus operator performs the set difference operation.

Example:

```
a = 45.1  45 # a is .1
a = [1,2,3,4,5]  [1,2,3] # a is [4,5]
```

Multiplication Operator (∗) The multiplication operator is a binary operator that can perform multiplication on numbers (doubles, binary, hex). On strings and lists, this performs a repetition of elements.

Example:

```
a = 45*2 # a = 90
"hello" * 3 # "hellohellohello"
```

Division Operator (/) The division operator is a binary operator that can perform division on numbers. Division is similar to division in C: when dividing integers, the answer will also be an integer. To return a double, you must add a 0 to one of the operands.

Example:

```
a = 45/2 # a is 22
b = 45.0/2 # b is 22.5
```

Modulus Operator (mod**, %)** This operator is a binary operator that has 2 symbols, mod and %, which perform the same function. It performs the modulus function on doubles, using the mathematical style of Ruby, allowing for non-integer values.

Example:

```
a = 3.1%3.0 # a is .1
```

Exponentiation Operator (**) This operator is a binary operator that raises the left hand to the right hand power, on numbers.

Example:

```
a = 2 ** 3 # a is 2 to the 3rd power, 8
```

7.6.3 Bitwise Operators

Bitwise complement (~) This operator, the tilde on an US keyboard, performs two's complement on numbers, hexadecimals, and bytes.

Example:

```
a = ~0xFFFFFF
```

Bitwise and (/\\) This operator, a combination of the forward slash followed by a backslash, is a binary operator that performs bitwise and on all numeric types. On sets, this performs the set intersection operation.

Example:

```
a = 0xFFFFF /\ 0xAAAAA
a = [1,2,3,4] /\ [3,4,5,6]
```

Bitwise xor (^) This operator, the carrot found as Shift+6 on an US keyboard, is a binary operator that performs bitwise xor on all numeric types. On sets, this performs the set symmetric difference operation.

Example:

```
a = 0xFFFFF ^ 0xAAAAA
a = {1,2,3,4} ^ {3,4,5,6}
```

Bitwise or (\\/) This operator, the combination of a backslash followed by a forward slash, is a binary operator that performs bitwise or on all numeric types. On sets, this performs the set union operations.

Example:

```
a = 0xFFFFF \/ 0xAAAAA
a = [1,2,3,4] \/ {3,4,5,6}
```

Bitwise shift-left (<<) This operator shift-lefts doubles, hexadecimals, and bytes, discarding the bits shifted out and shifting in zeros on the right. The operand on the right determines how many bits are shifted.

Example:

```
a = 0xFFFFFF << 1
```

Bitwise shift-right (>>): This operator shift right doubles, hexadecimals, and bytes the sign bit is shifted in on the left, thus preserving the sign of the operand. Further on, while shifting right, the empty spaces will be filled up with a copy of the most significant bit (MSB). The operand on the right determines how many bits are shifted.

Example:

```
a = 0xFFFFFF >> 1
```

7.6.4 Logical Operators

Boolean not (!) This operator performs the logical complement on numbers, hexadecimals, bytes, strings, and booleans.

Example:

```
a = ~0xFFFFFF
```

Boolean and (&&, *and*) And will evaluate to true only when both the left operand and the right operand are true. There is a difference in precedence between the *and* keyword and the && symbol; *and* has a lower precedence.

Example:

```
a = true
b = true
a && b #evaluates to true
```

Boolean xor (*xor*) Xor will to true only when one and only one of the operands on either side of the expression is true.

Example:

```
a = true
b = false
a xor b #evaluates to true
```

Boolean or (*or*, ||) Or evaluates to true when at least one of the operands on either side of the expression is true. There is a difference in precedence between the keyword *or* and the || symbol; *or* has a lower precedence.

Example:

```
a = true
b = false
a or b #evaluates to true
```

Equals (== , *is*) Equals returns true when the value of the left operand is equal to the value of the right operand. For numbers, it is mathematical equality, for strings, if all the characters are the same. There is a difference in precedence between the keyword is and the == symbol: *is* has a lower precedence.

Example:

```
a = 1
b = 1
a == b # evaluates to true
```

Not equals (*is not*, ! =) Not equals returns when the value of the right operand is not equal to the value of the left operand. For numbers, it is mathematical inequality, for strings, if all the characters are not the same. There is a difference in precedence between the keyword *is not* and the ! = symbol.

Not (*not*, !) : Not *null* is true, not *false* is true, and the complement of everything else is false.

Example:

```
a = 1
!a # evaluates to false
b = null
not b # evaluates to true
```

7.6.5 Assignment Operators

Equals (=) The equals operator is a assignment operator that sets the value of the identifier on the left side to the value of the identifier on the right side. When an indexing operator is used, this can set elements with a list or a hash.

Example:

```
a = 4 # a is 4
b = 5 # b is 5
a = b # a is now 5
```

Exponentiation Equals (∗∗ =) The exponentiation equals is an assignment operator that sets the value of the identifier on the left side to the value of the identifier on the left side raised to the value of the identifier on the right.

Example:

```
a **= b # equivalent to a = a**b
```

Divide Equals (/ =) The divides equals is an assignment operator that sets the value of the identifier on the left side to the value of the identifier on the left side divided by the value of the identifier on the right.

Example:

```
a /= b # equivalent to a = a/b
```

Modulus Equals (% =) The modulus equals is an assignment operator that sets the value of the identifier on the left side to the value of the identifier on the left side modulus the value of the identifier on the right.

Example:

```
a %= b # equivalent to a = a%b
```

Plus Equals (+ =) The plus equals is an assignment operator that sets the value of the identifier on the left side to the value of the identifier on the left side plus the value of the identifier on the right.

Example:

```
a += b # equivalent to a = a+b
```

Minus Equals (− =) The minus equals is an assignment operator that sets the value of the identifier on the left side to the value of the identifier on the left side minus the value of the identifier on the right.

Example:

```
a -= b # equivalent to a = a-b
```

Bitwise left shift Equals (<<=) The bitwise left shift equals is an assignment operator that sets the value of the identifier on the left side to the value of the operator on the right side left shifted.

Example:

```
a <<= b # equivalent to a = <<b
```

Bitwise right shift Equals (>>=) The bitwise right shift equals is an assignment operator that sets the value of the identifier on the left side to the value of the operator on the right side right shifted.

Example:

```
a >>= b # equivalent to a = >>b
```

And equals (&& =) The 'and-equals' is an assignment operator that sets the value of the identifier on the left side to the value of the identifier on the left anded with the the value of the identifier on the right.

Example:

```
a &&= b # equivalent to a = a&&b
```

Or equals ($||=$) The or equals is an assignment operator that sets the value of the identifier on the left side to the value of the identifier on the left or'ed with the the value of the identifier on the right.

Example:

```
a ||= b # equivalent to a = a||b
```

Bitwise or equals ($\backslash/=$) The bitwise or equals is an assignment operator that sets the value of the identifier on the left side to the value of the identifier on the left bitwise or'ed with the the value of the identifier on the right.

Example:

```
a \/= b # equivalent to a = a\/b
```

Bitwise and equals ($/\backslash=$) The bitwise and equals is an assignment operator that sets the value of the identifier on the left side to the value of the identifier on the left bitwise and'ed with the the value of the identifier on the right.

Example:

```
a /\= b # equivalent to a = a/\b
```

Bitwise xor equals ($\hat{}=$) The bitwise xor equals is an assignment operator that sets the value of the identifier on the left side to the value of the identifier on the left bitwise xor'ed with the the value of the identifier on the right.

Example:

```
a ^= b # equivalent to a = a^b
```

7.6.6 Relational Operators

Less than ($<$) The less than sign is a relational operator that returns true if the left operand is smaller than the right operand, for numbers, and if the left operand is lexicographically before the right operand for strings. Otherwise, the expression returns false. On sets, this checks if the left operand is a true subset of the right operand: that is, $set < other$ means $set <= other$ and $set! = other$.

Less than or equal to ($<=$) The less than or equal to sign is a relational operator that returns true if the left operand is smaller than or equal to the right operand for numbers, or if the left operand is lexicographically before or equal to the right operand for strings. Otherwise, the expression returns false. On sets, this checks if every element in the left operand is in the right operand.

Greater than (>) The greater than sign is a relational operator that returns true if the left operand is greater than the right operand, for numbers, and if the left operand is lexicographically after the right operand for strings. Otherwise, the expression returns false. On sets, this checks if the left operand is a true superset of the right operand: that is, $set > other$ means $set >= other$ and $set! = other$.

Greater than or equal to (>=) The greater than or equal to sign is a relational operator that returns true if the left operand is greater than or equal to the right operand for numbers, or if the left operand is lexicographically after or equal to the right operand for strings. Otherwise, the expression returns false. On sets, this checks if every element in the right operand is in the left operand.

7.6.7 Indexing

For the index of list, 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 a null value. Indexing starts from 0. Index values are truncated to integers. To assign to an element in the list, the list identifier followed by a square bracketed index is followed by an assignment statement. The list can be extended by passing a positive number larger than the size as the bracketed index in the assignment. The list will be padded with null values.

Example:

```
a = [1,2,3,4]
a[2] = 1 # a is now [1,2,1,4]
a[-1] # returns 4
# a[-5] is an error
```

For hashes, indexing is done by the key. If the key exists in the hash, the hash returns the corresponding value. Otherwise, the hash returns *null*.

Example:

```
b = {"hello" => 1, "world" => 2}
b["hello"] => 1
b[0] # returns null

b["goodbye"] = 3

# b is now {"hello" => 1, "goodbye" => 3, "world" => 2}.
# Notice hashes do not have a guaranteed ordering.
```

Sets do not have an index, because there is no guaranteed position that each element in the set has. You can still iterate through a set, however, and can save each element in a list, at which point you can access the elements by their indices.

8 Scoping

8.1 Variable Scoping

Variables declared inside of a node are not accessible to the code within the function definitions. We are using lexical scoping, so the code within a sub-node is not accessible to the body of the primary node, and the code within the node is not accessible to the main.

Example:

```
node N()
        x = 3
        y = 2

        node F()
                x = 5 #a different x variable
                return x #returns 5
        end

        return x /* returns 3, if the return was missing
                                the return would be the value of y */
end
```

8.2 Node Scoping

To access a node in another file the import declaration must be used. To access sub nodes of a node, the dot operator is used. Nodes must be declared before they are accessed or called.

Example:

```
import "sort.td"
# imports nodes from the sort file,
# contained in the file sort.td
# May include mergesort, quicksort, and selection
# sort nodes.

import "sort.td"
/* The above imports all public nodes and their public
        sub-nodes from the file sort.td */
```

9 Type Conversions

9.1 Numeric and String Conversions

Converting an int to a double is valid.

Example:

```
a = 45.1
b = 45
b = a + b # Valid assignment; b = 90.1
```

Converting an int or double to a string is valid.
Example:

```
a = 45.1
b =
b = a + b # b is a string containing 45.1
```

9.2 Boolean Conversions

Converting a boolean to a string is valid.
Example:

```
a = true
b =
b = a + b #b is a string reading true
```

Converting a string to a boolean is only valid if the string literal is 'true' or 'false' (case insensitive).
Example:

```
a = true
b = false
b = a # b is now true
```

10 Statements and Expressions

10.1 Statements

The two types of statements are Node Statements and Main Statements.

10.1.1 Node Statements

A node is the basic unit of the language. Nodes can contain expressions and nodes. A node is declared using the node keyword followed by comma separated parameters in parentheses. The node code ends with an 'end' keyword, the grammar determines which 'end' keyword within the node is the proper closing *end* keyword. If no return is explicitly declared, the last expression before the closing end is considered the return value.
Example:

```
node Node1(a, b, c)
        if a < 2
                return x = 5
        else
```

```
        end #this end closes the if else expression

        if b > 2
                return y = 1 # declares a variable and returns it
        else

        end

        node Inner_node(d) # declaring an inner node
                return d+1
        end #ending an inner node

        z = inner_node c /*c is being passed a parameter to inner_node
        /* if the returns are not specified , z is the return value , implicitly .
           The last non-node line 's value is the default return value. */
end # this is the end that closes node1
```

10.1.2 Main Statements

Main statements can be Loop Statements, Assignment Statements, or Expressions.

Loop Statements There are for-loops, while-loops, do-while-loops, Until-loops, and generic loops.

The *break* keyword is used to break out of the current loop. *continue* is used to proceed to the next iteration. Every loop is ended by the keyword *end*.

Generic Loops: Generic loops are done as follows:

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

For-loops: For loops are used to loop through all the items of a Set, List, Hash, or Range.

Example:

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

While-Loops: While loops are used to loop as long as condition holds. Example:

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

Until-Loops: Until-loops loop until a condition is met:

```
Until condition
        # do stuff
end
```

10.2 Expressions

Expressions are statements that return a value. There are conditional expressions, pipeline expressions, and the expressions containing the operators in Section 7.6.

10.2.1 Pipeline Expressions

A pipeline is an expression that contains a node id followed by space separated parameters followed by the pipeline (|) symbol, followed by the any number of node ids separated by pipelines. Also, you can have a pipeline of one node, consisting of a single node that is followed by space separated parameters. As a matter of good style and to avoid any errors, just follow the simple rule:

NO NODE CODE IN THE PIPELINE!

By this we mean that you do not put conditionals and expressions in the pipeline, only nodes, literals, ids for basic types. For example, lists may not go in the pipeline. Also note that a value piped from one node to another is a copy of the return value of the preceding node so as to preserve thread safety.

Example:

```
Node1 1 2 3 | Node2 | Node3 /* This means the return of node1
taking parameters 1,2, and 3 will be given as parameters to node2,
the results of node2 will be given as parameters to node3 */

a = 5
Node1 a 2 3 | Node2 # good

a = 5
Node1 a+1 2 3 | Node2 # Error NO NODE CODE IN THE PIPELINE!
Node1 if a < 6 | Node2 else | node3 # Error NO NODE CODE IN THE PIPELINE!
```

10.2.2 Conditional Expressions

Conditional expressions are used to evaluated different statements and return a value based on some condition(s). The three conditional expression types are *cond*, *if*, and *unless*. Conditionals return the last line executed unless the *return* keyword is specified.

If-expressions If-expressions evaluate a certain portion of code if a certain condition is met; if the condition is not met, it will evaluate the other. If expressions must have an else portion.

Example:

```
if condition
	# do stuff
else
	# do some other stuff
end
```

Unless-expressions Unless-expressions will evaluate a block of code unless conditions is met:

```
unless condition
	# do stuff
end
```

Cond-expressions Cond-expressions evaluate blocks of code given their respective conditions hold. If a condition is met, its block of code is executed and the other conditions are skipped.

Example:

```
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
end
```

11 Imports and System Functions

11.1 Import Syntax

Imports allow you to access nodes from other files or libraries. You can import every node in a file this way:

```
import filename
```

11.2 System Functions

The system functions (nodes) are print, read, and write. Additionally, math and random nodes are provided, as well as a set node.

11.2.1 Using Ruby Classes and Require Syntax

Require allows you to access nodes from Ruby files, allowing access to far more system functions than a single developer could ever create him or herself. Ruby classes are loaded, and then used within Tandem by using the pipeline syntax to instantiate, call, and manipulate objects. Ruby library files can be loaded by using

```
require 'filename'
```

11.2.2 Print:

The print node takes a value and prints it to standard out, without printing a newline. Example:

```
Print 1 # this prints 1
Print 'gamma' # this prints the string 'gamma'
List | QuickSort | Print # prints a sorted list
```

11.2.3 Read

Read reads in a file and tokenizes it by line by default. The only parameter is the file name.

Example:

```
book = File.read 'Starship Troopers.txt'  # reads in the text

for line in book
        #do something with the information
end
```

11.2.4 Write

Write takes a file name (does not have to be existent yet) for the first parameter, 'w' (indicating write) or 'a' (indicating append) as the second element, and an array of lines to add as the last element.

Example:

```
book = File.read 'Starship Troopers.txt' # read it in
book2 = File.read 'War of the Worlds.txt'# read the other in

File.write 'Starship Troopers Backup.txt'  'w' book # back up Starship Troopers
File.write 'War of the Worlds Backup.txt'  'w'  book2  # back up Starship Troope
File.write 'War of the Worlds.txt'  'a'  book   # append Starship Troopers to Wa
```

11.2.5 Math Functions

There are many math System functions included (refer to the Ruby documentation): (Math.abs, Math.sqrt, Math.log, Math.pi, Math.inf, Math.e, Math.sin, Math.cos, Math.tan, Math.asin, Math.acos, Math.atan, Math.sinh, Math.cosh, Math.tanh, Math.gamma, Math.floor, Math.ceil).

11.2.6 Random Functions

A Random node is provided that returns a random integer within some provided range, can return a double between 0 and 1, or can pick an element from a sequence (list, hash, set, string, or range) at random.

```
prng = Random.new 1234 # provide a seed value to the rng
a = prng.rand # a is now a random float
b = rand(1..10) # b is now a random integer between 1 and 10, inclusively
```

11.2.7 Set

A set is declared as follows: an identifier is followed by a single equals sign followed by the *Set* node , followed by any number of unique literals separated by spaces. The set is not guaranteed to be ordered.

Example:

```
a = Set 1 2 3 4 5
```

12 Future Language Extensions

In the future, extensions will be added to Tandem to provide first-class functions, forking nondeterministically, add exceptions, complex numbers, comprehensions, and add option types.

12.1 First-class functions

This feature would allow users to pass nodes as parameters and to create anonymous functions. Users would be able to use higher-order functions to create functions that return other functions, and these functions could possibly close over the variables in scope at the time to allow for closures. This would make recursive functions more powerful.

12.2 Fork

Fork will work exactly like cond, except all of the conditions that evaluate to true will result in a new thread being spawned and the corresponding pipeline being run. This will allow for nondeterministic code. While there are always thread safety issues, all data passed between nodes is immutable, and therefore should not cause invalid reference errors, or be altered unexpectedly.

12.3 Exceptions

This feature would add an exceptions system, as well as provide for functionality with the *try*, *catch*, and *finally* as done in Java. Users could create exception nodes, throw and catch exceptions, and then guarantee code to be run after the exception is handled. Also, a *with* keyword would be provided for, as in Python, that would automatically close up files after reading or writing so users would not have to do all reading and writing within a *try/catch* statement.

12.4 Comprehensions

This feature would add list, set, and hash comprehensions. A microsyntax would be created to allow users to pick elements that satisfy some condition, such as in the following:

```
squares = [x**2 for x in 0..9]

# equivalent to the following
squares = []
for x in 0..9
        squares += x**2
end

# squares is now [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

12.5 Option and Empty Types

This would allow for *none* and *some*, to indicate whether an element was empty, or if there was some element there. This would allow for the disambiguation between the usage of *null* to represent undefined values and to represent an empty value.

12.6 Unicode Support

This would allow for users to use Unicode characters in *id* names, node names, and elsewhere. Unicode support would not only be useful for international users, who may not be as used to reading code as native English speakers are.

13 Appendix

13.1 Language Grammar

Listing 12: TanG.g. The lexer and parser. Creates an AST.

```
//TanG Grammar  Patrick De La Garza - Language Guru


grammar TanG;

options{
language=Java;
output=AST;
ASTLabelType=CommonTree;
}

@lexer::members{
 public List<String> errors = new ArrayList<String>();
}



//Start: rewritten so that start Token is not null
tanG      :         prog ->^(ROOTNODE[",,,,"] prog?);

//Describes the program layout
prog      :         (NEWLINE* ((i ((NEWLINE+ EOF)?|(NEWLINE+
     m (NEWLINE+ EOF)?)))? | (m)));

//Import Statements
i         :         ((td_imp^ filename)|td_require^ STRING) (
    NEWLINE+ iprime)?;

 iprime :           ((td_imp^ filename)|td_require^ STRING) (
     NEWLINE+ i)?;

//Main body: this is composed of any number of valid
    statements
m         :         (statementNL (NEWLINE+ statementNL)*)->^(
    MAIN[",,,"] statementNL+);

//This production is used to rewrite statements so that
    we minimize changes to the original code generator
statementNL
          :         statement->statement NEWLINE["\n"];
```

```
//This is the list of valid statement types, starting with a node definition
statement
        :       td_node^ NODEID LPAREN params RPAREN NEWLINE+ (m NEWLINE+)? td_end
        |       expression
        |       loopType
        |       td_return orExpression
        |       td_assert orExpression
        |       td_break (orExpression)?
        |       td_continue;

//valid node parameters
params  :       (ID(COMMA ID)*)?;

//All of the loop types
loopType        :       td_for ID td_in iterable NEWLINE+ (m NEWLINE+)? td_end
        |       td_while orExpression NEWLINE+ (m NEWLINE+) td_end
        |       td_loop NEWLINE+ (m NEWLINE+) td_end
        |       td_until orExpression NEWLINE+ (m NEWLINE+)? td_end;

//Things that can be iterated through
iterable        :       rangeExpr;

//Expressions, these consist of condition statements and expressions
expression
        :       condType | orExpression;

//conditionals
condType        :       td_if orExpression NEWLINE+ (m NEWLINE+)? td_else NEWLINE+ (m NEWLINE+)? td_end
        |       td_unless orExpression NEWLINE+ (m NEWLINE+)? td_end
        |       td_cond^ NEWLINE+ (cstatement NEWLINE+)* td_end;

//Cases for cond statements
cstatement
        :       orExpression^ NEWLINE+ (m NEWLINE+)? td_end;
```

```
//ExpressionTypes
orExpression
        :       xorExpr (td_or^ xorExpr)*;

xorExpr :       andExpr (td_xor^ andExpr)*;

andExpr :       notExpr (td_and^ notExpr)*;

notExpr :       (td_not^)*  memExpr;

memExpr :       idTestExpr (td_memtest^ idTestExpr)?;

idTestExpr
        :       modExpr (td_idtest^ modExpr)?;


modExpr :       assignment (td_mod^ assignment)*;

assignment
        :       assignable (ASSN^ assignment)|rangeExpr;

assignable
        :       (assnAttr^ (LBRACK assnAttr RBRACK)*);

assnAttr:       (ID (DOT^ ID)*);

rangeExpr
        :       boolOrExpr (RANGE^ boolOrExpr)?;

boolOrExpr
        :       boolAndExpr (BOOLOR^ boolAndExpr)*;

boolAndExpr
        :       eqTestExpr (BOOLAND^ eqTestExpr)*;

eqTestExpr
        :       magCompExpr (EQTEST^ magCompExpr)?;

magCompExpr
        :       bitOrExpr (MAGCOMP^ bitOrExpr)?;

bitOrExpr
        :       bitXorExpr (BITOR^ bitXorExpr)*;

bitXorExpr
        :       bitAndExpr (BITXOR^ bitAndExpr)*;
```

```
bitAndExpr
        :       bitShiftExpr (BITAND^ bitShiftExpr)*;

bitShiftExpr
        :       addSubExpr (BITSHIFT^ addSubExpr)*;

addSubExpr
        :       multExpr (ADDSUB^ multExpr)*;

multExpr:       unariesExpr ((MULT^| STAR^) unariesExpr)
    *;


unariesExpr
        :       (ADDSUB^)* bitNotExpr;

bitNotExpr
        :       (BITNOT^)* expExpression;
expExpression
        :       pipelineExpr (EXP^ expExpression)?;


pipelineExpr
        :       atom|(pipeline -> ^(PIPEROOT[",,"]
            pipeline))
        ;

pipeline:       ((pipestart (indexable)* (pipe^ pipenode)
    *));


pipe    :       PIPE;

pipestart
        :       attrStart^ (LBRACK (pipestart|pipeatom2)
            RBRACK)*;//(ID|NODEID) (DOT^ (NODEID|ID|FUNCID
            ))*;

pipenode
        :       ((NODEID) (DOT^ (NODEID|ID|FUNCID))*)|(ID
            (DOT^ (ID|NODEID|FUNCID))+);


indexable
```

```
        :       (nonAtomAttr^ (LBRACK indexable RBRACK)+)
            |pipeattributable;

attrStart
        :       (ID|NODEID) (DOT^ (ID|NODEID|FUNCID))*;

nonAtomAttr
        :       ID (DOT^ ID)*;


pipeattributable
        :       (ID (DOT^ ID)+)|pipeatom;

//atom
atom    :       INT|FLOAT|HEX|BYTE|STRING|paren|list|
    hashSet|td_truefalse|td_null|filename;
pipeatom:       ID|INT|FLOAT|HEX|BYTE|STRING|paren|
    hashSet|td_truefalse|td_null|filename;
pipeatom2
        :       INT|FLOAT|HEX|BYTE|STRING| paren|hashSet|
            td_truefalse|td_null|filename;

paren   :       LPAREN! orExpression RPAREN!; //->^(
    PARENTOKEN[",,,,,,,"] orExpression);

list    :       list2 ->^(LISTTOKEN[",,,,,"] list2);

list2   :       LBRACK (orExpression (COMMA orExpression)
    *)? RBRACK;

hashSet :       hashSet2->^(HASHTOKEN[",,,,,,"] hashSet2)
    ;

hashSet2
        :       LBRACE (orExpression (hashInsides))?
            RBRACE;
hashInsides
        :       FATCOMMA orExpression (COMMA orExpression
            FATCOMMA orExpression)*;


//Keywords
td_from :       FROM;
td_imp  :       IMPORT
                ;
filename        :       FILENAME;
```

```
td_node        :        NODE;
td_end         :        END;
td_return      :        RETURN;
td_assert      :        ASSERT;
td_break       :        BREAK;
td_continue    :        CONTINUE;
td_for         :        FOR;
td_in          :        IN;
td_while       :        WHILE;
td_do          :        DO;
td_loop        :        LOOP;
td_until       :        UNTIL;
td_if          :        IF;
td_else        :        ELSE;
td_unless      :        UNLESS;
td_cond        :        COND;
td_fork        :        FORK;
td_or     :        OR;
td_xor    :        XOR;
td_and    :        AND;
td_not    :        NOT;
td_memtest
          :        NOT? IN;
td_idtest
          :        IS (NOT)?;
td_mod    :        MOD;
td_truefalse
          :        TF;
td_none   :        NONE;
td_null   :        NULL;
td_some   :        SOME;
td_require
          :        REQUIRE;

//Lexer/Tokens

//Operators
PARENTOKEN
          :        ' , , , , , , , ';
HASHTOKEN
          :        ' , , , , , , ';
LISTTOKEN
          :        ' , , , , , ';
ROOTNODE:          ' , , , , ';
MAIN      :        ' , , , ';
PIPEROOT
```

```
        :       ',';
FUNCID  :       ('a'..'z'|'A'..'Z') ('a'..'z'|'A'..'Z'|'0'..'9'|'_')*'?';
COMMENT
    :   ('#' |'//') ~('\n'|'\r')*    {skip();}
;

FROM
        :       'from'
        ;
FILENAME        :       (('"')('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'0'..'9'|'_')* '.td' ('"'))
        |       (('\'')('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'0'..'9'|'_')* '.td' ('\''));
IMPORT
        :       'import'
        ;
REQUIRE :       'require';
NODE
        :       'node'|'public node'
        ;
TOKEN   :       'private';
END
        :       'end'
        ;
RETURN
        :       'return'
        ;
ASSERT  :       'assert';
CONTINUE        :       'continue';
BREAK   :       'break';
FOR     :       'for';
IN      :       'in';
WHILE   :       'while';
DO      :       'do';
LOOP    :       'loop';
IF      :       'if';
ELSE    :       'else';
UNTIL   :       'until';
UNLESS  :       'unless';
COND    :       'cond';
FORK    :       'fork';
OR      :       'or';
XOR     :       'xor';
AND     :       'and';
NOT     :       'not';
```

```
IS        :       'is';
MOD       :       'mod';
TF        :       'true'|'false';
NULL      :       'null';
SOME      :       'some';
NONE      :       'none';
WITH      :       'with';
TRY       :       'try';
CATCH     :       'catch';
FINALLY   :       'finally';
RANGE     :       '..';
FATCOMMA
          :       '=>';
EQTEST    :       '=='|'!=';
ASSN      :       '='|'+='|'-='|'*='|'/='|'%='|'**='|'>>='|
    '<<='|'^='
          |       '/\\='|'\\/='|'&&='|'||=';
BOOLOR    :       '||';
BOOLAND   :       '&&';
MAGCOMP   :       '>'|'<'|'>='|'<=';
BITOR     :       '\\/';
BITXOR    :       '^';
BITAND    :       '/\\';
BITSHIFT          :       '>>'|'<<';
ADDSUB    :       '+'|'-';
EXP       :       '**';
STAR      :       '*';
MULT      :       '/'|'%';
BITNOT    :       '!'|'~';
PIPE      :       '|';
DOT
          :       '.'
          ;
LPAREN
          :       '('
          ;
COMMA
          :       ','
          ;
RPAREN
          :       ')'
          ;

LBRACK    :       '[';
RBRACK    :       ']';

```

```
LBRACE  :        '{';

RBRACE  :        '}';
//other stuff
INT :   '0'..'9'+
    ;

FLOAT
    :   ('0'..'9')+  (
        {input.LA(2) != '.'}? => ('.' ('0'..'9')+
           EXPONENT? {$type = FLOAT;})
        |({$type = INT;})

    )
    |   ('0'..'9')+ EXPONENT
    ;



NEWLINE :        ('\r'? '\n')+
                 ;


WS  :   ( ' '
        | '\t'
        ) {skip();};



HEX     :        '0x' (HEX_DIGIT)+;

BYTE    :        '0b' ('1'|'0')+;

STRING
    :  '"' ( ESC_SEQ | ~('\\'|'"') )* '"'
    ;

EXPONENT : ('e'|'E') ('+'|'-')? ('0'..'9')+ ;

PUBPRIV :        'public'|'private';

NODEID  :        ('A'..'Z')('a'..'z'|'A'..'Z'|'0'..'9'|'_'
   )*;

ID  :   ('a'..'z'|'_') ('a'..'z'|'A'..'Z'|'0'..'9'|'_')*
```

```
        ;



fragment
HEX_DIGIT : ('0'..'9'|'a'..'f'|'A'..'F') ;

fragment
ESC_SEQ
        :    '\\' ('b'|'t'|'n'|'f'|'r'|'\"'|'\''|'\\');



INVALID
  :   . {
            errors.add("Invalid character: '" + $text + "' on
                line: " +
                 getLine() + ", index: " +
                     getCharPositionInLine()); };
```

Part IV

Project Plan

14 Development Process

14.1 Planning

Our team met early on at least once a week to discuss features to add to the compiler. After submitting the language reference manual, we began to develop an understanding of what was feasible and what was not feasible to include in the grammar. With this information in mind, we assigned parts and began the development process.

14.2 Development

We originally envisioned an agile development process, where we would be able to add a feature, and test as soon as it was developed (perhaps because we were using test driven development). We would perform this in one to two week cycles and then add a new set of features. However, this did not come to pass. Instead, we spent a great deal of time getting the grammar to work, which prevented us from writing a functional tree walker early on because the tree walker is so heavily dependent on the structure of the tree and the tokens that are produced. This meant that our development process very quickly converged on changing the grammar to make it easier to traverse the AST, which would require us to change the tree walker all over again.

14.3 Testing

Our runtime and testing environments were as decoupled as possible from the compiler itself, so they were not affected by changes in the syntax tree. Testing was distributed among all the team members, with each person responsible for testing their own code and also writing general unit tests to test the compiler as a whole.

15 Roles and Responsibilities of Team Members

15.1 Varun

Varun was the project manager. He focused on the buildfile.xml to compile the grammar, wrote TandemTest to test the parser, helped Donald create the runtime environment, and led the decisions on what could finally be added or removed from the language. He and generally kept people on task. He managed dependencies for the project, and helped all the team members set up and keep their runtime environments working, including helping the others with git, helping with the tandem executable, and managing the repository on Github. Finally, he assembled the tutorial, language resource manual, and project report after receiving other members' parts.

15.2 Patrick

Patrick fulfilled the role of the Language Guru. He monitored the direction and identity of the language. Patrick developed the grammar and tested most of its productions. He oversaw tests and made sure that the target code was semantically equivalent to the original Tandem code. Patrick spent a significant portion of his time tending to the grammar, which underwent changes throughout the evolution of the project. Due to time constraints, not all of the features of the language were able to be implemented and changes were made to the grammar in order to allow for some quick fixes and patches. Patrick enacted these changes while trying to preserve those aspects of the language that the group had determined to be most important.

15.3 Jeneé

Jeneé assisted in the creation of test cases for the program during the first half of the semester. Then after the grammar was completed, Jeneé took on the role of code generation-translating all the code from Tandem to Ruby. Using a skeleton of switch-cases, Jeneé filled in the translations to Ruby, which included a lot of manipulation of the Abstract Syntax Tree that we received as input from the parser and the tokens. The code generator-TreeWalker.java will output the ruby file with the same name as the input file, but with the rb extension. If the Tandem file had invalid syntax, then the Ruby translation will be incorrect. If the Tandem syntax was correct, the Ruby file will run.

15.4 Donald

Donald wrote scripts that checked dependencies and executed the parts of the tool chain. The scripts also made sure that the tandem files intended to be converted actually existed. Furthermore, he also wrote a script that resolved the dependencies between tandem files. The final compiler executable was the creation of Donald. He wrote out the basic skeletons of TandemTree and TreeWalker, which were then filled out by systems architect, Jeneé. Also, he wrote a geometry library and test programs that used the geometry functions to check for the proper value of those functions.

16 Implementation Style Sheet

The team had a very loose style guide. Members were to make sure that interfaces were agreed upon, that commit messages were meaningful, and that code was commented when the logic was not readily apparent. For the most part, this methodology worked out well; team members worked on their own files and kept to their own styles. However, early on, when members made a small change to a file, they would also change the spacing on the file. This inflated the commit in git and made measuring lines written by each team member harder to judge.

17 Project Timeline

Date	Milestone
2/2/12	First major meeting. Discussed language idea, and settled on finite state machine language.
2/22/12	Handed in white paper.
3/1/12	Started work on grammar.
3/21/12	Finished Language Resource Manual and Language Tutorial.
4/8/12	First git commit. Started work on ANTLR grammar.
4/11/12	Basic grammar stabilized. build.xml file started.
4/19/12	Unit tests automation complete.
4/20/12	TreeWalker started.
4/25/12	Lexer and parser working. build.xml file working.
5/3/12	TanG.gunit tests started.
5/4/12	Ivy added to eliminate dependencies. Bash file added to run compilation from start to finish.
5/6/12	Compiler completed. Project report completed.

18 Project Log

```
# Project Log

## 5/6/12

Finished project report. Wrote script to manage
    dependencies without falling into an infinite loop.
    Continued testing and debugging tree walker, and made
    slight changes in the grammar to facilitate this. ./
    tandem must be called with absolute path to file.

## 5/5/12

Fixed bugs in the pipeline. Updated language tutorial and
    LRM to reflect grammar changes. Continued debugging
    the pipeline in TreeWalker. Got rid of private nodes
    and appended nodes with a main function that is called
    in the pipeline. Added Ruby system classes. Only
    allow chained assignments to be assigned to variables.
    Added language evolution, development and runtime
```

```
    environment sections to project report, as well as
    lessons learned and on test methodology.

## 5/4/12

Added support for Ivy, which downloads dependencies and
    puts them in the lib folder to be used by code.
    Removed old dependencies. Allowed files to be run with
     JRuby. Finished basic pipeline support for nodes.
    Fixed traversing the tree. Finished Bash script to
    start compilation process and then call Ruby on the
    resulting Ruby file. Started putting project report
    together. Patrick and Donald finished writing their
    parts for the project report.

## 5/3/12

Now, only ID's can be indexable or attributable. Literal
    lists cannot be pipeline parameters.  Nodes are now
    required to have capital names.

Added support for importing Ruby code. Can only call
    methods that are in the form of Class.method or Class.
    method?, but Ruby methods that end in other
    punctuation will not work. This takes care of the
    system libraries.


## 5/2/12

Removed chain comparisons in order to be able to move on.
     Changed code generator to use nested classes instead
    of modules, and began to add public and private
    keywords to nodes. Removed multiline comments because
    we couldn't get them to work and recognize that the
    code was failing.

## 4/26/12

Started on the tree grammar for the parser. Need to
    rewrite 3 or so rules to do the transformations for
    magnitude comparisons, equality testing, and the
    pipeline. Started work on the code generator, where a
    Ruby file is created within the Java code.

## 4/25/12
```

Worked on unit tests. Added target for TreeWalker, fixed whitespace bugs in grammar, prevented users from adding lists, sets, or hashes to the pipeline. Worked on understanding how to write the Tree grammar, and improved on visualizing the nodes that were created in the AST.

4/19/12

Got unit tests automation working, so running `ant test` will run the entire compilation process. Added assignments, literals, and corrected associativites on the grammar. Also fixed comments in the grammar.

4/18/12

Created a test file that can parse a file and report whether it parses correctly or not so we can run unit tests. Converted arithmetic, logical, and bitwise expressions with correct precedences.

4/8/12

Debated about whether we should have true division or integer division. Made project structure and added them to Github. Got git up and running on everyone's computer, and taught everybody the basic commands to push, pull, commit, and checkout code. Started writing the ANTLR grammar, and discovered ANTLRWorks, an IDE to work on the ANTLR file, display the DFA for the grammar, and check the grammar.

3/29/12

Went over the comments that Prof. Aho and Shuai sent us about the LRM, realizing that some of their complaints were just located elsewhere in the LRM. Debated about the tool we should use to write the grammar, and finally settled on ANTLR. Realized that we would need to rewrite the grammar as LL(*), so started working on the transformations neededed from LR(1) to LL(*).

3/20/12

Continued working on the tutorial and finished the grammar. Added the grammar to the LRM, and asked others for sample programs to add to the tutorial. Started talking about what to write the grammar in. Realized that yacc's support for outputting to Java is experimental at best, so looked at ANTLR, JCup, JFlex. Also realized that the dynamic typing we would need prevent us from writing Java code. Evaluated Python, Ruby, and Groovy code, and decided to choose Ruby as our code target.

3/19/12

Added keywords like while and for. Fixed the pipeline to allow for multiple pipelines and literals in the pipeline. Added operator, associativity, and operation meaning table. These include arithmetic, boolean, logical, and bitwise operators.

3/1/12

Began working on the grammar. Decided that nodes can only call imported nodes, and that you cannot create nodes within other nodes. Nodes are static, and are compiled down to Java classes, and any code in the main section of the code is the main node, located in the main function of the Java class. Created import, node, main body, whitespace, and basic pipeline production in the grammar.

2/26/12

Met to start working on the syntactic features of the language. Decided that we should have pipelines, like in Unix, to have function calls. Planned out the aspects of the grammar that we would need, expressions we wanted, and conditionals and loops that we might need.

2/15/12

Discussed features of the language. Decided that the language should be used for hardware, network state programming, game theory programs, neural networks, networking programs, and simulations. The programming language is platform independent, nodal, threaded,

```
    dynamic, compiled, functional, modular, and massively
    parallel. Assigned parts of the white paper to the
    others, and asked them to be done by 2/20 so we could
    have some time to review the language. Decided to call
     the language Tandem.

## 2/2/12

Discussed ideas on what sort of language to create.
    Talked about functional language ideas, languages for
    parallel programming, languages to do finite state
    machines, languages like Python and Ruby (general-
    purpose languages), and other ideas. Decided to create
     a finite state machine language.
```

Part V
Language Evolution

19 Sticking to the Language Proposal

Due to time constraints, design decisions, and limitations of the tools used for compiler construction, Tandem underwent many changes during implementation. In order to preserve the aspects that make Tandem useful and unique functionality, modularity, object-orientation, dynamic typing, and architecture independence we changed some of the features in the original version of the language and made amendments throughout the design process.

Before completion of the original language reference manual (LRM), the language had 'def' functions that could only be used within the scope of a given node. It was decided that allowing for nested node statements would better fit the nodal paradigm. We also developed our current Unix-pipeline-inspired pipe syntax to replace individual node connections; originally, each node had to be connected to another with a special operator. The pipeline syntax is conducive to producing readable code and allows for our state-machine-like logic.

20 Compiler Tools

Our compiler depended heavily on ANTLR, Apache Ant, Apache Ivy, and Java to run successfully. ANTLR is a tool that creates lexers and parsers based upon an input grammar file. We used Ant to download Ivy, which in turn managed our dependencies. Ivy downloaded the necessary dependencies to a local folder, and checked that the versions matched the code's requirements. Our dependencies included ANTLR, gUnit (used to test the grammar), JUnit (used to test Tandem files to see if they parsed), and JRuby. We then compiled the grammar using Ant, compiled the corresponding lexer and parser, compiled the tree walker, and ran the main driver class to create the Ruby file, which *tandem* ran by calling JRuby on the file.

21 Maintaining Consistency

The original language and LRM are different from those included in this report. Almost all of the changes to the original LRM and language were made in order to have our language fit the constraints of our compiler tools and libraries - despite these changes, our language did not undergo any drastic modifications.

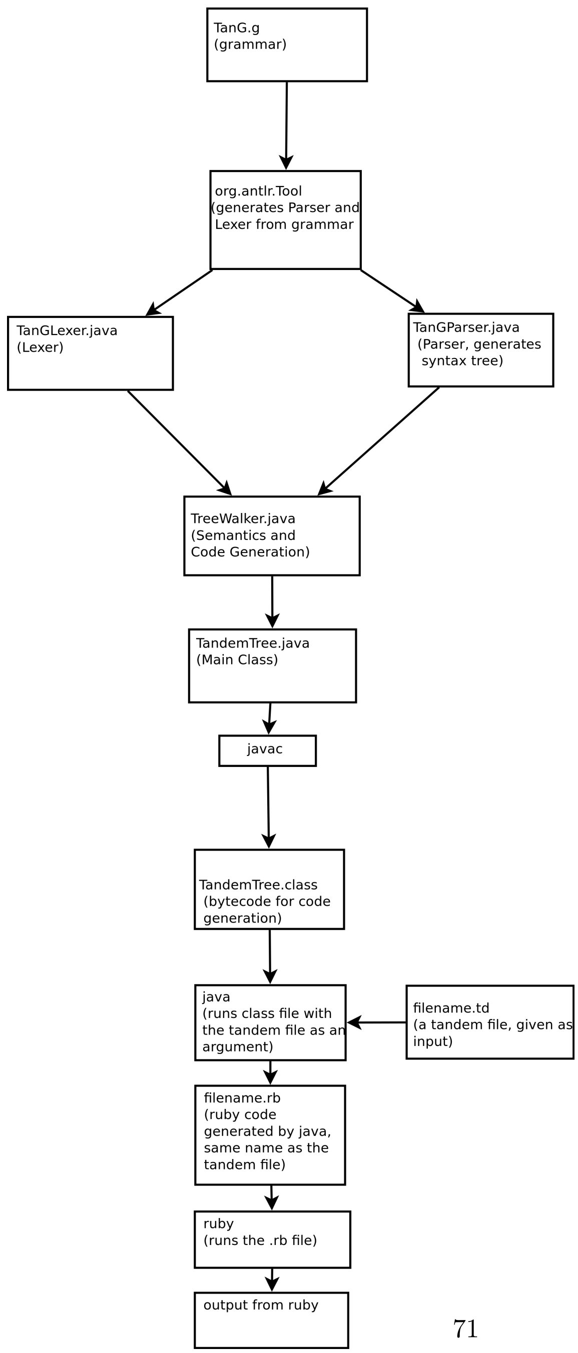
We used ANTLR to generate our lexer and parser; thus, we had to change our grammar to an LL(*) grammar. This consisted of rewriting each production in extended Backus-Naur Form, or EBNF. Due to the fact that the lexer produced by ANTLR skips whitespace tokens unless you specify their position in the grammar explicitly, there was an ambiguity in the pipeline syntax caused by an inability to tell list literals from indexed variables in the pipeline (i.e. $A\ [1, 2, 3]\ a[1]|\ B$). In order to fix this, we changed the grammar to not allow list literals in the pipeline. Set literals are now replaced by calling the special Set node on the list of items. ANTLR also has trouble emitting error tokens whenever you have an unclosed multiline comment. For this reason, we no longer allow multiline comments in the code. Do-while statements were deemed to be unnecessary and created too many problems in the grammar, so we removed them also. All of the above changes were made in the LRM as well.

Because of time constraints, we were still unable to implement the fork conditional. We decided to have our code compile down into JRuby code so that we could preserve architecture independence by running our code in the JVM. In order to implement the System Library as specified in the new LRM, we changed import statements so that you can either import an entire Tandem file or use the keyword require to import a Ruby file directly. Also due to time constraints and the fact that Ruby does not allow you to make classes private, we were not able to implement the public and private keywords semantically - we do reserve them however. All nodes are made public instead.

We kept project logs and stored all changes as commits on Github in order to keep track of what changes needed to be made to the LRM.

Part VI
Translator Architecture

22 Architecture Block Diagram



23 Module Interfaces

The org.antlr.Tool serves as the interface between the grammar (TanG.g) and the parser and lexer, it automatically generates those files. The TandemTree uses the Parser and Lexer to generate a syntax tree, which is traversed by TreeWalker. The driver class is TandemTree, running java on the bytecode with a .td as an argument will generate ruby code. The ruby code is executed by the ruby program, either the pre-installed ruby of jruby. The entire compile process is automated by bash script tandem, which runs java via an Ant build file, build.xml, on TandemTree with the passed .td as an argument and then runs the ruby file generated by the java.

24 Module Authorship

- TanG.g - written by Patrick De La Garza
- TanGLexer and TanGParser - Automatically generated by org.antlr.Tool
- TreeWalker - Written by Jeneé Benjamin and Donald Pomeroy
- TandemTree.java - Skeleton was written by Donald Pomeroy, the majority of the code was written by Jeneé Benjamin
- The ruby code - Automatically generated by Java
- tandem Written by Donald Pomeroy
- TandemTest.java Written by Varun Ravishankar
- build.xml Written by Varun Ravishankar

Part VII
Development and Run-Time Environment

25 Software Development Environment

The software development environment included ANTLR, Java, Ruby, Ant, and Bash scripting including AWK, bc, the basic calculator. ANTLR was used to generate the lexer and parser. A grammar file (.g) was run through ANTLR to produce the java lexer and parcer .java files. These .java files were compiled into classes used to generate a tree walker. The tree walker produced ruby code, writing a new ruby file (.rb) with the name of the tandem (.td) file. The Ant file compiles the grammar, producing the java files, TanGLexer and TanGParser, then it compiles the java classes we wrote, which use the TanGLexer and parser to generate and traverse a tree. The code to compile and run a .td file was written in bash, which checks the file extensions and the existence of the file, and the dependencies with functions from AWK, grep, and bc. To help debug during the development, the tools ANTLRworks and GraphViz were used. ANTLRworks is an IDE for ANLTLR grammars, and supports checking the grammar, refactoring, showing the resulting DFA, and more. GraphViz, on the other hand, allows you to display an abstract syntax tree for debugging when walking the tree. For text editing, gedit and sublime text were used to develop the code. For version control we used git and github.com.

26 Ant buildfile

This file acted as our makefile. It downloaded dependencies, compiled the grammar and source files, ran the tree walker, and cleaned the Tandem directory.

Listing 13: build.xml. Ant file used to compile grammar, tree walker, test files. Downloads dependencies and runs tests.

```
<!-- build.xml -->
<!-- Written by Varun Ravishankar -->

<project xmlns:ivy="antlib:org.apache.ivy.ant" name="Tandem" default="compile" basedir=".">
  <description>
    build file for Tandem programming language.
  </description>

  <property name="project.name" value="Tandem" />

  <!-- program version -->
  <property name="version" value="0.1" />

  <!-- set global properties for this build -->
  <property name="src" location="src"/>
  <property name="build" location="bin"/>
  <property name="dist"  location="dist"/>
  <property name="test" location="test"/>
  <property name="tutorial" location="tutorial"/>
  <property name="lib" location="lib"/>
  <property name="grammar" value="TanG.g"/>
  <property name ="treeGrammar" value="TanG_TG.g"/>
  <property name="grammar1" value="WateredDownTanG.g"/>
  <property name="gunit_test" value="TanG.gunit" />
  <property name="test.class.name" value="TandemTest" />
```

```
<property name="jruby_jar" value="jruby-complete.jar" />

<!-- Properties required to download Ivy. -->
<property name="ivy.install.version" value="2.3.0-rc1" />
<condition property="ivy.home" value="${env.IVY_HOME}">
  <isset property="env.IVY_HOME" />
</condition>
<property name="ivy.jar.dir" value="${lib}/ivy" />
<property name="ivy.jar.file" value="${ivy.jar.dir}/ivy.jar" />

<property name="ivy.lib.dir" value="${lib}" />

<path id="lib.path.id">
  <fileset dir="${ivy.lib.dir}" includes="*.jar"/>
</path>


<!-- Path required to run the tests. Uses built-in JUnit,
 if one is installed, or the provided JUnit otherwise. -->
<path id="test.classpath">
 <fileset dir="${lib}">
    <include name="*.jar"/>
  </fileset>
  <fileset dir="${lib}/test">
    <include name="**/*.jar"/>
  </fileset>
  <pathelement location="${java.class.path}" />
</path>

<!-- Path required to build the grammar. Uses downloaded ANTLR,
 or the installed ANTLR otherwise. -->
<path id="build.classpath">
  <fileset dir="${lib}">
    <include name="*.jar"/>
  </fileset>
  <fileset dir="${lib}/build">
    <include name="**/*.jar"/>
  </fileset>
  <pathelement location="${java.class.path}" />
</path>
```

```
<!-- Path required to build the grammar. Uses downloaded ANTLR,
 or the installed ANTLR otherwise. -->
<path id="runtime.classpath">
 <fileset dir="${lib}">
    <include name="*.jar"/>
  </fileset>
  <fileset dir="${lib}/runtime">
    <include name="**/*.jar"/>
  </fileset>
  <pathelement location="${java.class.path}" />
</path>


<!-- An ant macro which invokes ANTLR3
      This is just a parameterizable wrapper to simplify the invocation of ANTLR3.
      The default values can be overriden by assigning a value to an attribute
      when using the macro.
      Example with ANTLR3 outputdirectory modified:
      <antlr3 grammar.name="TanG.g" outputdirectory="${src}/${package}"/>
-->

<taskdef resource="org/apache/tools/ant/antlr/antlib.xml"
 classpath="${lib}/antlr3-task/ant-antlr3.jar" />


<macrodef name="antlr3">
  <attribute name="grammar.name"/>
  <attribute name="outputdirectory" default="${lib}/grammar"/>
  <attribute name="libdirectory" default="${lib}"/>
  <attribute name="multithreaded" default="true"/>
  <attribute name="verbose" default="true"/>
  <attribute name="report" default="false"/>
  <attribute name="debug" default="false"/>
  <sequential>
    <ant-antlr3 xmlns:antlr="org/apache/tools/ant/antlr"
      target="@{grammar.name}"
      outputdirectory="@{outputdirectory}"
      libdirectory="@{libdirectory}"
      multithreaded="@{multithreaded}"
```

```
          verbose="@{verbose}"
          report="@{report}"
          debug="@{debug}">
          <classpath>
            <pathelement location="${lib}/antlr3-task/ant-antlr3.jar"/>
            <path refid="build.classpath"/>
          </classpath>
          <jvmarg value="-Xmx512M"/>
        </ant-antlr3>
      </sequential>
    </macrodef>

    <target name="download-ivy" unless="offline">
      <mkdir dir="${ivy.jar.dir}"/>
      <!-- download Ivy from web site so that it can be used even without any special installation -->
      <get src="http://repo2.maven.org/maven2/org/apache/ivy/ivy/${ivy.install.version}/ivy-${ivy.install.version}.jar"
        dest="${ivy.jar.file}" usetimestamp="true"/>
    </target>

    <target name="install-ivy" depends="download-ivy">
      <!-- try to load ivy here from ivy home, in case the user has not already dropped
           it into ant's lib dir (note that the latter copy will always take precedence).
        We will not fail as long as local lib dir exists (it may be empty) and
        ivy is in at least one of ant's lib dir or the local lib dir. -->
      <path id="ivy.lib.path">
        <fileset dir="${ivy.jar.dir}" includes="*.jar"/>
      </path>

      <taskdef resource="org/apache/ivy/ant/antlib.xml"
        uri="antlib:org.apache.ivy.ant" classpathref="ivy.lib.path"/>
     </target>

     <target name="resolve" depends="install-ivy" description="Resolve the dependencies">
        <ivy:retrieve pattern="lib/[conf]/[artifact](-[classifier]).[ext]"/>
     </target>
```

```

   <target name="init" depends="resolve">
    <!-- Create the time stamp -->
    <tstamp/>
    <!-- Create the build directory structure used by compile -->
    <mkdir dir="${build}"/>
    <mkdir dir="${dist}"/>
    <mkdir dir="${lib}"/>
    <mkdir dir="${lib}/grammar"/>
  </target>

  <target name="antlr_classpath">
    <whichresource property="antlr.in.classpath" class="org.antlr.Tool"/>
    <fail message="ANTLR3 not found via CLASSPATH ${java.class.path}">
      <condition>
        <not>
          <isset property="antlr.in.classpath"/>
        </not>
      </condition>
    </fail>
  </target>

  <target name="grammar" depends="init" description="compile the grammar" >
    <!-- Compile the grammar from ${src} into ${build} -->
    <antlr3 grammar.name="${src}/${grammar}" outputdirectory="${lib}/grammar"/>
    <!-- <antlr3 grammar.name="${src}/${treeGrammar}" outputdirectory="${lib}/grammar"/> -->
  </target>


  <target name="compile" depends="grammar" description="compile the source " >
    <!-- Compile the java code from ${src} into ${build} -->
    <!-- includeantruntime="false" -->
    <javac destdir="${build}" includeantruntime="false" verbose="false">
      <classpath>
        <path refid="build.classpath"/>
      </classpath>
```

```
      <src path="${lib}/grammar" />
    </javac>
    <javac destdir="${build}" includeantruntime="false" verbose="false">
      <!-- <classpath path="${build}" /> -->
      <classpath>
        <path refid="test.classpath"/>
      </classpath>
      <src path="${src}" />
      <exclude name="**/Test.java"/>
      <!-- <exclude name="**/TreeWalker.java"/> -->
    </javac>
  </target>

  <target name="dist" depends="compile"
     description="generate the distribution" >
    <!-- Create the distribution directory -->
    <mkdir dir="${dist}/lib"/>

    <!-- Put everything in ${build} into the Tandem-${DSTAMP}.jar file -->
    <jar jarfile="${dist}/lib/Tandem-${DSTAMP}.jar" basedir="${build}"/>
  </target>

  <target name="parse_test" depends="compile">
    <junit showoutput="no" fork="yes" haltonfailure="no">
      <test name="${test.class.name}" />
      <formatter type="plain" usefile="false" />
      <classpath>
        <pathelement path="${build}"/>
          <path refid="test.classpath"/>
      </classpath>
    </junit>
  </target>

  <target name="gunit_test" depends="compile">
    <copy file="${src}/${gunit_test}" tofile="${build}/${gunit_test}"/>
    <java classname="org.antlr.gunit.Interp">
      <arg value="${build}/${gunit_test}"/>
      <classpath>
        <pathelement path="${build}"/>
          <path refid="runtime.classpath"/>
          <path refid="test.classpath"/>
      </classpath>
```

```
    </java>
  </target>


  <target name="test" depends="gunit_test, parse_test">

  </target>

  <target name="walk" depends="compile">
    <java classname="TandemTree">
      <arg value="${file}"/>
      <classpath>
        <pathelement path="${build}"/>
          <path refid="runtime.classpath"/>
      </classpath>
    </java>
  </target>

  <target name="rubyexec" depends="init">
    <java jar="${lib}/runtime/${jruby_jar}" fork="true">
      <arg value="${file}"/>
      <classpath>
        <pathelement path="${build}"/>
          <path refid="runtime.classpath"/>
      </classpath>
    </java>
  </target>

  <target name="rubytest" depends="compile">
    <java jar="${lib}/runtime/${jruby_jar}" fork="true">
      <arg value="test/tutorial/geometry_question1_test.rb"/>
      <classpath>
        <pathelement path="${build}"/>
          <path refid="runtime.classpath"/>
      </classpath>
    </java>
    <java jar="${lib}/runtime/${jruby_jar}" fork="true">
      <arg value="test/tutorial/geometry_question2_test.rb"/>
      <classpath>
        <pathelement path="${build}"/>
        <path refid="runtime.classpath"/>
      </classpath>
    </java>
    <java jar="${lib}/runtime/${jruby_jar}" fork="true">
```

```
      <arg value="test/tutorial/geometry_question2_test.rb"/>
      <classpath>
        <pathelement path="${build}"/>
        <path refid="runtime.classpath"/>
      </classpath>
    </java>
    <java jar="${lib}/runtime/${jruby_jar}" fork="true">
      <arg value="test/tutorial/geometry_question2_test.rb"/>
      <classpath>
        <pathelement path="${build}"/>
        <path refid="runtime.classpath"/>
      </classpath>
    </java>
  </target>

  <target name="clean" description="clean up" >
    <!-- Delete the ${build} and ${dist} directory trees -->
    <delete dir="${build}"/>
    <delete dir="${dist}"/>
    <delete dir="${lib}/grammar"/>
  </target>

    <!-- =================================
          target: clean-ivy
         ================================= -->
  <target name="clean-ivy" depends="clean-downloads" description="--> clean the ivy installation">
    <delete dir="${ivy.jar.dir}"/>
  </target>

  <target name="clean-downloads" description="--> clean the downloaded jars">
    <delete dir="${ivy.lib.dir}/build"/>
    <delete dir="${ivy.lib.dir}/runtime"/>
    <delete dir="${ivy.lib.dir}/test"/>
  </target>

    <!-- =================================
          target: clean-cache
         ================================= -->
  <target name="clean-cache" depends="install-ivy"
      description="--> clean the ivy cache">
    <ivy:cleancache />
```

```
    </target>
</project>
```

27 Tandem Executable

This is a Bash script that calls the Ant file, compiles the Tandem file, and runs the corresponding Ruby output. It also checks for dependencies.

Listing 14: tandem. Calls Ant file and runs Ruby output.

```
#!/usr/bin/env bash

# Written by Donald Pomeroy

EXPECTED_ARGS=1
E_BADARGS=65

#Check if a file is passed, if not fail

if [ $# -ne $EXPECTED_ARGS ]
then
  echo "Usage: `basename $0` {arg}"
  echo "COMPILATION FAILED"
  exit $E_BADARGS
fi

FILE=$1

#Check if the filename is valid .td

if [[ "$FILE" == *.td ]]
then
  echo "valid filename"
else
     echo "COMPILATION FAILED - invalid filename"
     exit
fi

if [ -f "$FILE" ]
then
    echo "the file exists"
else
    echo "COMPILATION FAILED - the file does not exist"
    exit
fi

#check java
#check ant 1.8
#check ruby 1.9.2

```

```
if type -p java; then
    echo "found java executable in PATH"
    _java=java
elif [[ -n "$JAVA_HOME" ]] && [[ -x "$JAVA_HOME/bin/java" ]]; then
    echo "found java executable in JAVA_HOME"
    _java="$JAVA_HOME/bin/java"
else
    echo "COMPILATION FAILED - no java"
    exit
fi

if [[ "$_java" ]]; then
    version=$("$_java" -version 2>&1 | awk -F '"' '/version/ {print $2}')
    #echo version "$version"
    if [[ "$version" > "1.5" ]]; then
        echo "version is more than 1.5"
    else
        echo "COMPILATION FAILED - version is less than 1.5"
        exit
    fi
fi



if type -p ant; then
    echo found ant executable in PATH
    _ant=ant
elif [[ -x "usr/bin/env ant" ]]; then
    echo "found ant executable in ENV_ANT"
    _ant="usr/bin/env ant"
else
    echo "COMPILATION FAILED - no ant"
    exit
fi

if [[ "$_ant" ]]; then
    version=$("$_ant" -v 2>&1 | awk -F ' ' '/ / {print $4}')
    echo version "$version"
    if [[ "$version" > "1.8.0" ]]; then
        echo "version is more than 1.8"
    else
```

```
        echo "COMPILATION FAILED - version is less than 1.8"
        exit
    fi
fi

#Make JRuby use Ruby 1.9

JRUBY_OPTS=--1.9


if type -p ruby; then
    echo "found ruby executable in PATH"
    _ruby=ruby
elif [[ -x "usr/bin/env ruby" ]];  then
    echo "found ruby executable in ENV"
    _ruby="usr/bin/env ruby"
else
    echo "COMPILATION FAILED - no ruby"
    exit
fi

use_jruby=0

if [[ "$_ruby" ]]; then
    version=$("$_ruby" -v 2>&1 |grep -e "1\.9\.[2-9]" )
   # echo $?
    if [[ "$?" > "0" ]]; then
    use_jruby=1
    else
    echo "Passed Ruby Check"
    fi
fi


DIR=$(pwd)

#$(echo $IN | tr ";" "\n")

a=$(ruby src/dependency.rb "$FILE" | tr " " "\n")

for x in $a
do
        ant walk -Dfile="$x" -q
done

```

```
echo "compiling ..."

ant walk -Dfile="$FILE" -q

ruby_file=${FILE%.*}

#run a different version of ruby based on ant rubyexec

if [[ "$_use_jruby" > 0 ]]; then
        java -jar lib/runtime/jruby-complete.jar --1.9 "
            $ruby_file"
        else
        ruby "$ruby_file".rb
fi
```

28 Run-time Environment

The run-time of the compiler requires that tandem executable be in the same directory as build.xml. Tandem can run on any system that has bash installed. The Bash script calls Java, which generates the Ruby code, and Ruby, which runs the code. Ruby can be jRuby or the usual ruby compiler. ANTLR is required to run the compiler, as the Java requires the libraries to traverse the tree. To fulfill dependencies on ANTLR, StringTemplate, JUnit, gUnit, and JRuby, the Ant file will call Ivy to check for and download the missing dependencies. The bash script will check for ant, java, and ruby. Compilation will halt and fail if the computer running it is missing Bash, Java, ANTLR, Ant, or Ruby. Also when compiling a file, if the file is not in the same directory as the tandem script, the directory must be specified.

Part VIII
Test Plan

29 Test methodology

Our tests were crucial to promoting an environment where we could make changes without being very worried that our code would not work. This enabled high turnover rates in the grammar, and after instituting a rule that only working code would be added to the repo, ensured that the repo was always in a working state. Our tests included unit tests for the grammar and parser, integration testing for the tree walker, functional testing for the produced Ruby code, and system testing for the compiler end-to-end.

29.1 Lexer and Parser Testing

We used the gUnit unit-test framework for ANTLR grammars in order to test that the lexer and parser rules accepted the proper inputs. This tool allowed for test input and output pairings for each individual production. Inputs consisted of character streams that the lexer and parser were supposed to interpret; outputs were either OK or FAIL depending on whether the input text was expected to be accepted or not. Unit tests were done for most productions and lexer tokens. gUnit achieves this by interpreting the gUnit script and running the tests using Java reflection to invoke methods from the grammars parser objects. Both valid and malicious code was used for these tests.

Within the ant buildfile, we automated the gUnit tests with the target:

```
$ ant gunit_test
gunit_test:
      [copy] Copying 1 file to /Users/Varun/Documents/
         Computer Science/COMS W4115 Programming languages
```

```
    and translators/Tandem/bin
  [java]
    ------------------------------------------------------------------------------------------

  [java] executing testsuite for grammar:TanG with 69
    tests
  [java]
    ------------------------------------------------------------------------------------------

  [java] 0 failures found:
  [java] Tests run: 69, Failures: 0
BUILD SUCCESSFUL
```

29.2 JUnit Testing

We used JUnit to test the parser. The ant file, after compiling the grammar and TandemTree, attempts to lex and parse the Tandem file. If this produces lexer errors, the errors are printed out to the screen. TandemTree then attempts to parse the file, and prints a failure to the screen if there is a parser error. Otherwise, the parser suceeds and exits.

JUnit lets you assert whether a method variable is equal to some expected value. By checking for a Boolean value that was set by the parser, we can test if the parser produces errors. Automated testing was crucial to being able to make quick changes to the grammar to eliminate or add productions, as well as creating rewrites in the tree.

```
$ ant parse_test
parse_test:
    [junit] Testsuite: TandemTest
    [junit] Tests run: 9, Failures: 0, Errors: 0, Time
       elapsed: 0.329 sec
    [junit] ------------- Standard Error
       -----------------
    [junit] /Users/Varun/Documents/Computer Science/COMS
       W4115 Programming languages and translators/Tandem
       /test/failure/crazycharacter.td line 1:0 no viable
        alternative at input '@@'
    [junit] Number of syntax errors in /Users/Varun/
       Documents/Computer Science/COMS W4115 Programming
       languages and translators/Tandem/test/failure/
       crazycharacter.td: 1
    [junit]
    [junit] /Users/Varun/Documents/Computer Science/COMS
       W4115 Programming languages and translators/Tandem
       /test/failure/expression.td line 23:5 mismatched
       input 'n1' expecting NODEID
```

```
[junit] Number of syntax errors in /Users/Varun/
    Documents/Computer Science/COMS W4115 Programming
    languages and translators/Tandem/test/failure/
    expression.td: 1
[junit]
[junit] /Users/Varun/Documents/Computer Science/COMS
    W4115 Programming languages and translators/Tandem
    /test/failure/node.td line 1:5 mismatched input '
    node1' expecting NODEID
[junit] /Users/Varun/Documents/Computer Science/COMS
    W4115 Programming languages and translators/Tandem
    /test/failure/node.td line 18:6 mismatched input '
    node2' expecting NODEID
[junit] Number of syntax errors in /Users/Varun/
    Documents/Computer Science/COMS W4115 Programming
    languages and translators/Tandem/test/failure/node
    .td: 2
[junit]
[junit] /Users/Varun/Documents/Computer Science/COMS
    W4115 Programming languages and translators/Tandem
    /test/failure/simpleStatementTest.td line 1:0 no
    viable alternative at input '@@'
[junit] Number of syntax errors in /Users/Varun/
    Documents/Computer Science/COMS W4115 Programming
    languages and translators/Tandem/test/failure/
    simpleStatementTest.td: 1
[junit]
[junit] /Users/Varun/Documents/Computer Science/COMS
    W4115 Programming languages and translators/Tandem
    /test/failure/unclosed_comment.td line 1:0 no
    viable alternative at input '/'
[junit] Number of syntax errors in /Users/Varun/
    Documents/Computer Science/COMS W4115 Programming
    languages and translators/Tandem/test/failure/
    unclosed_comment.td: 1
[junit]
[junit] /Users/Varun/Documents/Computer Science/COMS
    W4115 Programming languages and translators/Tandem
    /test/failure/ватereddownimportfail.td line 1:0 no
     viable alternative at input 'from'
[junit] Number of syntax errors in /Users/Varun/
    Documents/Computer Science/COMS W4115 Programming
    languages and translators/Tandem/test/failure/
    watereddownimportfail.td: 1
[junit]
[junit] ------------- ----------------
```

```
[junit]
[junit] Testcase: test_whitespace took 0.064 sec
[junit] Testcase: test_comments took 0.001 sec
[junit] Testcase: test_expression took 0.052 sec
[junit] Testcase: test_math_expression took 0.008 sec
[junit] Testcase: test_bitwise_expression took 0.007 sec
[junit] Testcase: test_logic_expression took 0.004 sec
[junit] Testcase: test_statement took 0.033 sec
[junit] Testcase: test_failures took 0.017 sec
[junit] Testcase: test_tutorial took 0.1 sec
```

29.3 Test/unit Functional Tests

These tests checked for valid Ruby code. They take converted Tandem files as inputs and check that the output matches an expected value. If not, we know that the tree walker is converting files incorrectly.

Listing 15: geometry.td

```
#geometry_operations.td - returns numerical values
#for the properties of common shapes such as area and volume
#Donald Pomeroy

node Hypotenuse_length(leg1, leg2)
        temp1 = leg1**2 + leg1**2
        return temp1**(0.5)
end

node Circle_area(radius)
        return (PI * (radius**2))
end

node Circle_perimeter(radius)
        return (2*PI*radius)
end

node Square_perimeter(side_len)
        return (side_len*4)
end

node Rectangle_area(len,width)
        return (len*width)
```

```
end

node Rectangle_perimeter(len,width)
        return ((2*len)+(2*width))
end

node Square_area(side_len)
        return side_len * side_len
end

node Cylinder_volume(radius, height)
        return (Circle_area radius)*height
end

node Cube_Volume(side_len)
        return side_len ** 3
end

node Cone_Volume(radius, height)
        return (1.0/3.0)*(Circle_area radius)*height
end

node Cube_surface_area(side_len)
        return (Square_area side_len)*6
end

node Cylinder_surface_area(radius,height)
        return 2*(PI*(radius**2)) + (2*PI*r*h)
end

node Sphere_surface_area(radius)
        return 4*(Circle_area radius)
end

node Sphere_volume(radius)
        return (4.0/3.0)*(Circle_area radius)*(radius)
end

node Cone_surface_area(height,radius)
        PI * radius * (Hypotenuse_length height radius) *
            (Circle_area radius)
end

node Rectangular_prism_volume(length,width,height)
        return length*width*height
end
```

```

node Rectangular_prism_surface_area(length,width,height)
        return (2*(length*width)) + (2*(length*height)) +(2*(height*width))
end

node Trapezoid_area (base1,base2,height)
        return (1.0/2.0)*(base1 + base2)*height
end

node Trapezoidal_prism_volume(base1,base2,baseHeight, height)
        return height * (Trapezoid_area base1 base2 baseHeight)
end

node Triangle_area(base, height)
        return (1.0/2.0)*(base*height)
end

node Regular_pentagon_area(side_length)
        return (side_length**2)*1.7
end

node Regular_hexagon_area(side_length)
        return (side_length**2)*2.6
end

node Regular_octagon(side_length)
        return (side_length**2)*4.84
end

node Regular_icosahedron_volume(side_length)
        return (side_length**3)*2.18
end

node Regular_icosahedron_surface_area(side_length)
        return 8.66*(side_length**2)
end

node Torus_volume(minorRadius,majorRadius)
        return 2*PI*majorRadius * (Circle_area minorRadius)
end

node Torus_surface_area(minorRadius,majorRadius)
```

```
        return (2*PI*minorRadius)*(2*PI*majorRadius)
end

node Regular_tetrahedron_volume(side_length)
        return (2**(1.0/2.0))*(1.0/12.0)*(side_length**3)
end

node Regular_tetrahedron_surface_area(side_length)
        return (3**(1.0/2.0))*(side_length**2)
end
```

Listing 16: geometryquestion1.td

```
import "geometry.td"

#"You have an 10 by 10 rubiks cube You paint the outside.
    How many cubes have paint on them?"
#Donald Pomeroy

node Solve_question1(side_len)
        return (Cube_Volume side_len) - ((side_len-2)**3)
end

Solve_question1 10 | Print
```

Listing 17: geometryquestion1test.rb

```
require_relative 'geometry_question1'
require "test/unit"

#Donald Pomeroy

class TestGeometryQuestion1 < Test::Unit::TestCase

  def test_simple
    assert_equal(488, Solve_question1.new().main(10))
  end

end
```

Listing 18: geometryquestion2.td

```
import "geometry.td"
#Donald Pomeroy
#Ben draws a circle inside a square piece of paper whose
    area is 400 square inches.
#He makes sure than the circle touches each side of the
    square.
```

```
#What is the area of the circle?

node Solve_question2(area)
        radius = (area**(0.5))/2
        return Circle_area radius
end

Solve_question2 400 | Print
```

Listing 19: geometryquestion2test.rb

```
require_relative 'geometry_question2'
require "test/unit"
#Donald Pomeroy
class TestGeometryQuestion1 < Test::Unit::TestCase

  def test_simple

    assert_in_delta(314.16,Solve_question2.new().main
        (400),.1)
  end

end
```

Listing 20: geometryquestion3.td

```
import "geometry.td"
#Donald Pomeroy
#A length of wire is cut into several smaller pieces.
#Each of the smaller pieces are bent into squares.
#Each square has a side that measures 2 centimeters.
#The total area of the smaller squares is 92 square
    centimeters.
#What was the original length of wire?

node Solve_question3(total_area, side)
        num_squares = total_area / (Square_area side)
        return num_squares * (Square_perimeter side)
end

Solve_question3 92 2 | Print
```

Listing 21: geometryquestion3test.rb

```
require_relative 'geometry_question3'
require "test/unit"
#Donald Pomeroy
class TestGeometryQuestion3 < Test::Unit::TestCase
```

```

  def test_simple
    assert_equal(184, Solve_question3.new().main(92,2))
  end

end
```

Listing 22: geometryquestion4.td

```
import "geometry.td"

#Donald Pomeroy
#A piece of square paper has a perimeter of 32
    centimeters.
#Nicky's dog, Rocky, tore off 1/4 of the paper.
#What is the area of the remaining paper?

node Solve_question4(perimeter, loss)

        area =  Square_area (perimeter / 4.0)
        return area - (loss * area)
end

Solve_question4 32 (1.0/4.0) | Print
```

Listing 23: geometryquestion4test.rb

```
require_relative 'geometry_question4'
require "test/unit"
#Donald Pomeroy
class TestGeometryQuestion4 < Test::Unit::TestCase

  def test_simple
    assert_equal(48, Solve_question4.new().main(32,.25))
  end

end
```

29.4 Testing Code Generation

We spent the majority of the time before the midterm working on the grammar. Since we could not seriously start on the code generation before the grammar was complete, we spent the time building test programs. When we got to the code generation, my team members wrote test functions as I wrote the translation code from Tandem to Ruby in TandemTree and TreeWalker. So, we had test functions to test the different token of the grammar and to test functionality of our code, like the pipeling, expressions, and node definition and declaration.

We created programs testing the different operators addition, subtraction, multiplication, etc. and other operations that required us to translate the Tandem operators to Ruby operators- power 10, bitor, and a few more. We also had several td files to test importing files, and especially the node.

30 Test programs

Listing 24: TanG.gunit. Unit tests for grammar.

```
//Patrick De La Garza - Language Guru
gunit TanG;




//Test Parser

//Test tanG production
tanG  :
//This should fail
<<
node node1(a, b)
  cond
    a>b
      x=a+b
      node2 x
    end
    b>a
      y=b-a
      node3 b
    end
    a==b
      node4 a b

      x=y=z=1

    end
  end
  node node2(greaterA)
    print "I am at node2"
    greaterA-4
  end
  node node3(greaterB)
    answer =greaterB-8
    node4 answer greaterB | node2
```

```
      a|b|c|d|e|f
    end
    node node4(myA, myB)
      a = myA *5
      b = myB -5
      node5 a b
      node node5(g,h)
        sum = "I am at node 5"
        print sum
      end
    end
  end
>>FAIL //expect failure
<<
node Node1(a, b)
  cond
    a>b
      x=a+b
      Node2 x
    end
    b>a
      y=b-a
      A b c| D e| F
      Node3 b
    end
    a==b
      Node4 a b

      x=y=z=1

    end
  end
  node Node2(greaterA)
    print "I am at node2"
    greaterA-4
  end
  node Node3(greaterB)
    answer =greaterB-8
    Node4 answer greaterB | Node2
    A|B|C|D|E|F
  end
  node Node4(myA, myB)
    a = myA *5
    b = myB -5
    Node5 a b
    node Node5(g,h)
```

```
        sum = "I am at node 5"
        Print sum
      end
    end
end
>> FAIL //expect failure

//Should Succeed
<<
import "File.td"
import "File2.td"

import "StandardLib.td"

node Node1(a, b)
  cond
    a>b
      x=a+b
      Node2 x
    end
    b>a
      y=b-a
      A b c| D| F
      Node3 b
    end
    a==b
      Node4 a b

      x=y=z=1

    end
  end
  node Node2(greaterA)
    Print "I am at node2"
    greaterA-4
  end
  node Node3(greaterB)
    answer =greaterB-8
    Node4 answer greaterB | Node2
    A|B|C|D|E|F
  end
  node Node4(myA, myB)
    a = myA *5
    b = myB -5
    Node5 a b
    node Node5(g,h)
```

```
      sum = "I am at node 5"
      Print sum
    end
  end
end>>OK


//Test imports
i :
<<import "success.td"
import "mobetter.td">> OK

<<require "test">>OK

<<import "success.td"
require "rubyfile"
import "test.td">>OK

<<import require "game">>FAIL

<<
  import "success.td"
>>OK


//Should fail
<<import fail.td>>FAIL



//Main Body Test
m :

//cond failures
<<cond
  a is b
    a is c
      1+1
    end
  end
end>>FAIL

//assert test
<<assert a is b>>OK

<<assert assert>>FAIL
```

```
<<break true>>OK

<<assert loop
  break
end>>FAIL

//should succeed
<<a=b>>OK
//Should fail, no NodeCode again!
<<2|A>>FAIL

//Should fail, no NodeCode again!
<<2 A>>FAIL

<<2 a b>>FAIL

<<3|4|5>>FAIL

<<a|b|b>>FAIL

<<"Car"|"A"|C>>FAIL

//Should succeed; proper pipeline
<<A a b|B|C>>OK

//should fail: no NodeCode in the pipeline
<<A|B + 2|C>>FAIL


//This should fail
<<node node1(a, b)
  cond
    a>b
      x=a+b
      node2 x
    end
    b>a
      y=b-a
      node3 b
    end
    a==b
      node4 a b

      x=y=z=1

```

```
      end
    end
    node node2(greaterA)
      print "I␣am␣at␣node2"
      greaterA -4
    end
    node node3(greaterB)
      answer =greaterB -8
      node4 answer greaterB | node2
      a|b|c|d|e|f
    end
    node node4(myA, myB)
      a = myA *5
      b = myB -5
      node5 a b
      node node5(g,h)
        sum = "I␣am␣at␣node␣5"
        print sum
      end
    end
end
>>FAIL //expect failure
<<node Node1(a, b)
    cond
      a>b
        x=a+b
        Node2 x
      end
      b>a
        y=b-a
        A b c| D e| F
        Node3 b
      end
      a==b
        Node4 a b

        x=y=z=1

      end
    end
    node Node2(greaterA)
      print "I␣am␣at␣node2"
      greaterA -4
    end
    node Node3(greaterB)
      answer =greaterB -8
```

```
    Node4 answer greaterB | Node2
    A|B|C|D|E|F
  end
  node Node4(myA, myB)
    a = myA *5
    b = myB -5
    Node5 a b
    node Node5(g,h)
      sum = "I am at node 5"
      Print sum
    end
  end
end
>> FAIL //expect failure

//Should Succeed
<<node Node1(a, b)
  cond
    a>b
      x=a+b
      Node2 x
    end
    b>a
      y=b-a
      A b c| D| F
      Node3 b
    end
    a==b
      Node4 a b

      x=y=z=1

    end
  end
  node Node2(greaterA)
    Print "I am at node2"
    greaterA-4
  end
  node Node3(greaterB)
    answer =greaterB-8
    Node4 answer greaterB | Node2
    A|B|C|D|E|F
  end
  node Node4(myA, myB)
    a = myA *5
    b = myB -5
```

```
    Node5 a b
    node Node5(g,h)
      sum = "I am at node 5"
      Print sum
    end
  end
end>>OK

//should fail
<<1[2]>>FAIL




//Expression Unit Tests

expression:

//Should fail, no NodeCode in the PIPELINE!!
<<A|B|(C+2)>>FAIL

//Should fail, no NodeCode again!
<<A|2>>FAIL

//should succeed: note that it is actually two pipelines (pipe has higher precedence than +)
<<A|B+C|D>>OK


//LoopTests
loopType:

//for-loop
<<for item in list
  stuff
end>>OK

<<for item in list in biggerList
  makeithappen
end>>FAIL

//while-loop
<<while x>2
  x=x+1
end>>OK

```

```
<<while return true
  beelzebub
end>>FAIL

//loop
<<loop
  break x
end>>OK

<<loop true
  nope
end>>FAIL

//until
<<until pigsFly
  !hellFreezeOver
end>>OK

<<until true false
  asdf
end>>FAIL



//condTypes
condType:
//if-statements

<<if x is y
  x is z
else
  x is a
end>>OK

<<if x is y
  skipElse
end>>FAIL

//unless

<<unless player is charlieParker
  not listen
end>>OK

<<unless if true
  wait
```

```
end>>FAIL



//cond-statements

<<cond
  a>b
    jazz
  end
end>>OK

<<cond
  a is b
    1+1
  end
  a is c
    1+2
  end
end>>OK



//SPECIAL ATOMS

//list
list:
<<[a is b, c, d, [1,2]]>>OK
<<[a,b,[c,d]>>FAIL

//hash
hashSet:
//hash success
<<{"jam" => 0b0110, "jar" => 0xAFFC2}>>OK
//hash fail
<<{=>}>>FAIL
<<{a,b,c}>>FAIL
<<{a<=c=>d<=e}>>OK


//Lexer Tests

//Comment Tests
COMMENT :

//success
```

```
<<#this is a comment>>OK

//success
<<//This is also a comment>>OK

<</This is not a comments>>FAIL

//success
<<//#//#//#?>>OK


//Float stuff
FLOAT:
//success
<<1.1e+99>>OK

//fail due to improper exponent
<<1.1e-9.9>>FAIL

//float stuff
<<.0978>FAIL

//float stuff
<<.0E-0>>FAIL

//success
<<99e-99>>OK


//Test hex
HEX:
<<0x09aAFff>>OK

<<0x>>FAIL

//Test Byte
BYTE:
<<0b010010110>>OK
<<0b>>FAIL
<<0b012>>FAIL

//TEST string
STRING:
<<"WithotEscape codes babe">>OK
<<"Dog\nDog on new line">>OK
<<"DOG>>FAIL
```

```

//TEST_FUNCID
FUNCID:
<<Abcdefg?>>OK
<<gu?ess?>>FAIL
<<empty?>>OK
```

Listing 25: TandemTest.java. Unit tests for TandemTree. Checks if parser works on Tandem file.

```java
// TandemTest.java
// Written by Varun Ravishankar

import org.antlr.runtime.ANTLRStringStream;
import org.antlr.runtime.CommonTokenStream;
import org.antlr.runtime.RecognitionException;
import org.antlr.runtime.TokenStream;
import org.antlr.runtime.tree.CommonTree;
import java.io.*;
import org.antlr.runtime.*;
import org.junit.Test;
import org.junit.BeforeClass;
import static org.junit.Assert.*;


public class TandemTest
{
    private static File currentDir = new File(".");
    private static String currentDirName;
    private static String testPath;
    private final static String whitespace = "misc/whitespace/";
    private final static String comments = "misc/comments/";
    private final static String expression = "expression/";
    private final static String mathexp = "expression/math";
    private final static String bitwiseexp = "expression/bitwise";
    private final static String logicexp = "expression/logic";
    private final static String statement = "statement/";
    private final static String failure = "failure/";
    private final static String tutorial = "tutorial/";

    public static void main(String args[])
    {
        try
        {
            currentDirName = currentDir.getCanonicalPath();
        }
```

```
        catch(Exception e)
        {
            e.printStackTrace();
        }

        testPath = currentDirName + "/test/";
    }

    @BeforeClass
    public static void oneTimeSetUp()
    {
        try
        {
            currentDirName = currentDir.getCanonicalPath();
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }

        testPath = currentDirName + "/test/";
    }

    public static boolean parseFile(String filename)
    {
        boolean lexing_success = false;
        boolean parsing_success = false;

        try
        {
            // System.setErr(null);
            CharStream input = new ANTLRFileStream(filename);
            TanGLexer lexer = new TanGLexer(input);

            TokenStream ts = new CommonTokenStream(lexer);

            int errorsCount = lexer.getNumberOfSyntaxErrors();
            // ts.toString();
            if(errorsCount == 0)
            {
                lexing_success = true;
            }
```

```
            else
            {
                lexing_success = false;
                System.err.println("Number of lexer errors in " + filename + ": " + errorsCount + "\n");
                // return lexing_success;
            }


            TanGParser parse = new TanGParser(ts);
            parse.tanG();

            errorsCount = parse.getNumberOfSyntaxErrors();

            if(errorsCount == 0)
            {
                parsing_success = true;
            }
            else
            {
                parsing_success = false;
                System.err.println("Number of syntax errors in " + filename + ": " + errorsCount + "\n");
                // return parsing_success;
            }
        }
        catch(Exception t)
        {
            // System.out.println("Exception: "+t);
            // t.printStackTrace();
            parsing_success = false;
            return parsing_success;
        }

        return lexing_success && parsing_success;

    }

    public static File[] listTDFiles(File file)
    {
        File[] files = file.listFiles(new FilenameFilter() {
            @Override
```

```
            public boolean accept(File dir, String name)
            {
                if(name.toLowerCase().endsWith(".td"))
                {
                    return true;
                }
                else
                {
                    return false;
                }
            }
        });

        return files;
    }

    public static void run_success(File file)
    {
        File[] files = listTDFiles(file);

        for(File f : files)
        {
            if(f != null)
            {
                //System.out.println(f.getAbsolutePath());
                assertTrue("Failed␣to␣parse␣" + f.getName(), parseFile(f.getAbsolutePath()) );
            }
        }
    }

    public static void run_failure(File file)
    {
        File[] files = listTDFiles(file);

        for(File f : files)
        {
            if(f != null)
            {
                //System.out.println(f.getAbsolutePath());
                assertFalse("Should␣not␣have␣parsed␣" + f.getName(), parseFile(f.getAbsolutePath()) );
            }
```

```java
        }
    }

    @Test
    public void test_whitespace()
    {
        // System.out.println(testPath + whitespace);
        File file = new File(testPath + whitespace);
        run_success(file);
    }

    @Test
    public void test_comments()
    {
        File file = new File(testPath + comments);
        run_success(file);
    }

    @Test
    public void test_expression()
    {
        File file = new File(testPath + expression);
        run_success(file);
    }

    @Test
    public void test_math_expression()
    {
        File file = new File(testPath + mathexp);
        run_success(file);
    }

    @Test
    public void test_bitwise_expression()
    {
        File file = new File(testPath + bitwiseexp);
        run_success(file);
    }

    @Test
    public void test_logic_expression()
    {
        File file = new File(testPath + bitwiseexp);
        run_success(file);
    }

```

```
    @Test
    public void test_statement()
    {
        File file = new File(testPath + statement);
        run_success(file);
    }

    @Test
    public void test_failures()
    {
        File file = new File(testPath + failure);
        run_failure(file);
    }

    @Test
    public void test_tutorial()
    {
        File file = new File(testPath + tutorial);
        run_success(file);
    }
}
```

We also wrote programs that were included in the tutorial that were used for unit testing.

Listing 26: hello.td

```
Println "Hello, World!"
```

Listing 27: alt-hello.td

```
node Hello()
	Println "Hello, World!"
end

Hello
```

Listing 28: hello2.td

```
node Hello2()
	node MyWords(text)
		return text
	end

	MyWords "Hello, World!" | Println
end

Hello2
```

Listing 29: n.td

```
node N()
		node MyAge()
			age = 21
		end

	MyAge | Println
	# prints the output of MyAge
end
```

Listing 30: dsliterals.td

```
require "set"
node DSLiterals()
	node MakeList()
		a = [1,3,5,7,9,9]
		element = a[0] # element is equal to 1
	end

	node MakeSet()
		a = Set.new 0 2 4 6 8
```

```
			anotherElement = a[3] # anotherElement is equal to 6
	end

	finished = true
end

DSLiterals
```

Listing 31: sillymath.td

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

Listing 32: FirstFibonacci.td

```
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
```

Listing 33: Fibonacci.td

```
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
```

Listing 34: pipeline.td

```
import "Fibonacci.td"

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

    node G(x)
            x
    end

    cond
            a > 0
                    F a | Recursive | Println
            true
                    G a | Iterative | Println
            end
    end
end
```

Listing 35: fourbitshiftregister.td

```
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 | Println
```

Part IX
Conclusions

31 Lessons Learned as a Team

The need to effectively communicate the exact features of the language before attempting to implement it is a major lesson. In terms of writing tandem files, certain syntactic features we not standardized until later in the design process, such as the capitalization of nodes, the presence of do while loops, and the private keyword. Also, we should have been aware of what features we were capable of producing. In terms of designing a language, we should start with a minimal numbers of features and expand, rather than start with an extensive language and remove features we find hard to implement. Moreover, we found that waiting for the grammar to be finished before moving on to traversing the tree and semantic analysis delayed the development process. In terms of discovering bugs in the language, comprehensive unit testing helped us refine the compiler design. Testing for each of the productions of the grammar was the preferred method.

32 Lessons Learned by Team Members

32.1 Varun

The most important lesson we learned was that it is utterly necessary to define interfaces between classes so that team members can work on code in parallel. Our grammar took more than a month to finally complete, which prevented our other members from creating a working tree walker because token names and subtrees kept changing form. If we had created interfaces, this would have been less of a problem. We could have rewritten the syntax tree to keep the interface constant, and we would have had a working tree walker much sooner.

I also learned that it is important to teach tools to the entire team, and to not expect team members to pick up tools like git on the go. In one scenario, a team member deleted all the repository's file inadvertently, and we had to jump through hoops to get our files back.

32.2 Patrick

I learned the importance of clear and succinct communication of project expectations. Up until the very end of the project, we still asked each other questions about what the language should be allowed to do; we extensively discussed the language fundamentals and details early on, but it became quite apparent throughout the project that each team member had slightly different expectations for the language. Unit testing helped with this to a great extent. When creating tests that were expected to fail or succeed, small differences in the way the language was perceived to work became apparent. In this way, unit testing acted as both a validation tool and a test for consistency among group members. I believe that we could have implemented some of the future language extensions and saved a lot of time if we had better standards and consistency of direction.

32.3 Jeneé

This being my first course with rigorous teamwork, I learned quite a few things about working on a semester-long project with my peers. Setting up a consistent schedule is key and setting goals for meetings is also important- so we could pace ourselves in moving forward with the project. My team spent more meetings than necessary planning features that we wanted to do, rather than starting on a kernel for the basic features. Because of our late start, and weekly-biweekly irregular meetings, I think we were behind a bit in the grand scheme of things. I learned from this to start early and to do small parts at a time and only move on when we have the previous parts functioning. As the system architect, I would tell future teams to do a rewrite for the grammar so that it is easier to translate the code into the target language. Decide on a project topic and features as soon as possible. Use ANTRL- its a wonderful tool and it was very helpful in producing the AST and Parser. In terms of academically learning, I learned how to use a number of utilities over the course of the semester. I learned ANTLR, quite a bit of Ruby, the Graphviz Application and Node Generator code (or displaying the AST tree visually, which made it clearer to traverse the tree). I also became more proficient in git and vim. I learned about JRuby, build files, and a tiny bit of bash. Lastly, I learned that spending more than 15 hours straight with my team is mindboggling. At the same time though, I realized how brilliant they are and I learned a lot with them. So, I would recommend choosing partners who you know that you can learn a few new things from, but just friends.

32.4 Donald

The most perplexing part of the program was determining what to incorporate in the language. We spent too much time trying to figure out what features could be implemented, we often wavered on what features were necessary to include in the language. We started with too many features and thus we had to drop them or use more restrictive cases. In the future the design should start with a few simple features and then expand to incorporate more complex ones, rather than vice versa. On the more practical end of things, I learned how to use git more effectively. Furthermore, I learned to write more complex bash scripts and ruby code. I think the time management could have been done more effectively, most of the development was done towards the end of the project. This is probably related to the issues we had in deciding upon how to implement features effectively earlier in the project.

33 Advice for Future Teams

First off, start with a small, concise, and definite description of the language. This will allow the team to complete a basic version of the compiler. More complex features should not be included in the initial design. Changes to the language's style and syntax should be consistent and acknowledged by each team member during the design process. Also, stages further on in the development process should not necessarily depend on the completion of stages earlier in the process. For example, a team should develop semantic analysis and grammar at the same time. Also, the importance of testing cannot be underestimated, as it will catch many errors in the compiler.

34 Suggestions for the Instructor

Future editions of this course would be far more interesting if they discussed some special topics. It would have been fantastic to learn about garbage collection. Much of that would have required some operating systems knowledge, but learning to create a simple virtual machine would have taught us much about making modernday programming languages that do not require manual memory management. It would also have been nice to learn about parallel compilers, especially as compilers can increasingly take advantage of multiple cores and distributed machines to increase efficiency. Finally, a lecture briefly covering type theory would have been wonderful; we briefly covered type inference in class and type systems, but a lecture on types would have been nicer than yet another lecture on lambda calculus.

Part X
Appendix

A Source Code

Listing 36: TanG.g. The lexer and parser. Creates an AST.

```
//TanG Grammar  Patrick De La Garza - Language Guru


grammar TanG;

options{
language=Java;
output=AST;
ASTLabelType=CommonTree;
}

@lexer::members{
 public List<String> errors = new ArrayList<String>();
}



//Start: rewritten so that start Token is not null
tanG     :       prog ->^(ROOTNODE[",,,,"] prog?);

//Describes the program layout
prog     :       (NEWLINE* ((i ((NEWLINE+ EOF)?|(NEWLINE+ m (NEWLINE+ EOF)?)))? | (m)));

//Import Statements
i        :       ((td_imp^ filename)|td_require^ STRING) (NEWLINE+ iprime)?;

 iprime :        ((td_imp^ filename)|td_require^ STRING) (NEWLINE+ i)?;

//Main body: this is composed of any number of valid statements
m        :       (statementNL (NEWLINE+ statementNL)*)->^(MAIN[",,,"] statementNL+);

```

```
//This production is used to rewrite statements so that we minimize changes to the original code generator
statementNL
        :       statement->statement NEWLINE["\n"];

//This is the list of valid statement types, starting with a node definition
statement
        :       td_node^ NODEID LPAREN params RPAREN NEWLINE+ (m NEWLINE+)? td_end
        |       expression
        |       loopType
        |       td_return orExpression
        |       td_assert orExpression
        |       td_break (orExpression)?
        |       td_continue;

//valid node parameters
params  :       (ID(COMMA ID)*)?;

//All of the loop types
loopType        :       td_for ID td_in iterable NEWLINE+ (m NEWLINE+)? td_end
        |       td_while orExpression NEWLINE+ (m NEWLINE+) td_end
        |       td_loop NEWLINE+ (m NEWLINE+) td_end
        |       td_until orExpression NEWLINE+ (m NEWLINE+)? td_end;

//Things that can be iterated through
iterable        :       rangeExpr;

//Expressions, these consist of condition statements and expressions
expression
        :       condType | orExpression;

//conditionals
condType        :       td_if orExpression NEWLINE+ (m NEWLINE+)? td_else NEWLINE+ (m NEWLINE+)? td_end
        |       td_unless orExpression NEWLINE+ (m NEWLINE+)? td_end
        |       td_cond^ NEWLINE+ (cstatement NEWLINE+)* td_end;

//Cases for cond statements
```

```
cstatement
        :       orExpression^ NEWLINE+ (m NEWLINE+)?
            td_end;

//ExpressionTypes
orExpression
        :       xorExpr (td_or^ xorExpr)*;

xorExpr :       andExpr (td_xor^ andExpr)*;

andExpr :       notExpr (td_and^ notExpr)*;

notExpr :       (td_not^)*  memExpr;

memExpr :       idTestExpr (td_memtest^ idTestExpr)?;

idTestExpr
        :       modExpr (td_idtest^ modExpr)?;


modExpr :       assignment (td_mod^ assignment)*;

assignment
        :       assignable (ASSN^ assignment)|rangeExpr;

assignable
        :       (assnAttr^ (LBRACK assnAttr RBRACK)*);

assnAttr:       (ID (DOT^ ID)*);

rangeExpr
        :       boolOrExpr (RANGE^ boolOrExpr)?;

boolOrExpr
        :       boolAndExpr (BOOLOR^ boolAndExpr)*;

boolAndExpr
        :       eqTestExpr (BOOLAND^ eqTestExpr)*;

eqTestExpr
        :       magCompExpr (EQTEST^ magCompExpr)?;

magCompExpr
        :       bitOrExpr (MAGCOMP^ bitOrExpr)?;

bitOrExpr
```

```
        :       bitXorExpr (BITOR^ bitXorExpr)*;

bitXorExpr
        :       bitAndExpr (BITXOR^ bitAndExpr)*;

bitAndExpr
        :       bitShiftExpr (BITAND^ bitShiftExpr)*;

bitShiftExpr
        :       addSubExpr (BITSHIFT^ addSubExpr)*;

addSubExpr
        :       multExpr (ADDSUB^ multExpr)*;

multExpr:       unariesExpr ((MULT^| STAR^) unariesExpr)
    *;


unariesExpr
        :       (ADDSUB^)* bitNotExpr;

bitNotExpr
        :       (BITNOT^)* expExpression;
expExpression
        :       pipelineExpr (EXP^ expExpression)?;


pipelineExpr
        :       atom|(pipeline -> ^(PIPEROOT[",,"]
            pipeline))
        ;

pipeline:       ((pipestart (indexable)* (pipe^ pipenode)
    *));


pipe    :       PIPE;

pipestart
        :       attrStart^ (LBRACK (pipestart|pipeatom2)
            RBRACK)*;//(ID|NODEID) (DOT^ (NODEID|ID|FUNCID
            ))*;

pipenode
        :       ((NODEID) (DOT^ (NODEID|ID|FUNCID))*)|(ID
            (DOT^ (ID|NODEID|FUNCID))+);
```

```
indexable
        :       (nonAtomAttr^ (LBRACK indexable RBRACK)+)
            |pipeattributable;

attrStart
        :       (ID|NODEID) (DOT^ (ID|NODEID|FUNCID))*;

nonAtomAttr
        :       ID (DOT^ ID)*;


pipeattributable
        :       (ID (DOT^ ID)+)|pipeatom;

//atom
atom    :       INT|FLOAT|HEX|BYTE|STRING|paren|list|
    hashSet|td_truefalse|td_null|filename;
pipeatom:       ID|INT|FLOAT|HEX|BYTE|STRING|paren|
    hashSet|td_truefalse|td_null|filename;
pipeatom2
        :       INT|FLOAT|HEX|BYTE|STRING| paren|hashSet|
            td_truefalse|td_null|filename;

paren   :       LPAREN! orExpression RPAREN!; //->^(
    PARENTOKEN[",,,,,,,,"] orExpression);

list    :       list2 ->^(LISTTOKEN[",,,,,"] list2);

list2   :       LBRACK (orExpression (COMMA orExpression)
    *)? RBRACK;

hashSet :       hashSet2->^(HASHTOKEN[",,,,,,"] hashSet2)
    ;

hashSet2
        :       LBRACE (orExpression (hashInsides))?
            RBRACE;
hashInsides
        :       FATCOMMA orExpression (COMMA orExpression
            FATCOMMA orExpression)*;


//Keywords
td_from :       FROM;
```

```
td_imp    :           IMPORT
                      ;
filename              :           FILENAME;
td_node               :           NODE;
td_end                :           END;
td_return             :           RETURN;
td_assert             :           ASSERT;
td_break              :           BREAK;
td_continue           :           CONTINUE;
td_for                :           FOR;
td_in                 :           IN;
td_while              :           WHILE;
td_do                 :           DO;
td_loop               :           LOOP;
td_until              :           UNTIL;
td_if                 :           IF;
td_else               :           ELSE;
td_unless             :           UNLESS;
td_cond               :           COND;
td_fork               :           FORK;
td_or     :           OR;
td_xor    :           XOR;
td_and    :           AND;
td_not    :           NOT;
td_memtest
          :           NOT? IN;
td_idtest
          :           IS (NOT)?;
td_mod    :           MOD;
td_truefalse
          :           TF;
td_none   :           NONE;
td_null   :           NULL;
td_some   :           SOME;
td_require
          :           REQUIRE;

//Lexer/Tokens

//Operators
PARENTOKEN
          :           ',,,,,,,,';
HASHTOKEN
          :           ',,,,,,';
LISTTOKEN
          :           ',,,,,';
```

```
ROOTNODE:       ',,,,';
MAIN    :       ',,,';
PIPEROOT
        :       ',,';
FUNCID  :       ('a'..'z'|'A'..'Z') ('a'..'z'|'A'..'Z'|'0'..'9'|'_')*'?';
COMMENT
    :   ('#' |'//') ~('\n'|'\r')*    {skip();}
;

FROM
        :       'from'
        ;
FILENAME        :       (('"')('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'0'..'9'|'_')* '.td' ('"'))
        |       (('\'')('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'0'..'9'|'_')* '.td' ('\''));
IMPORT
        :       'import'
        ;
REQUIRE :       'require';
NODE
        :       'node'|'public_node'
        ;
TOKEN   :       'private';
END
        :       'end'
        ;
RETURN
        :       'return'
        ;
ASSERT  :       'assert';
CONTINUE        :       'continue';
BREAK   :       'break';
FOR     :       'for';
IN      :       'in';
WHILE   :       'while';
DO      :       'do';
LOOP    :       'loop';
IF      :       'if';
ELSE    :       'else';
UNTIL   :       'until';
UNLESS  :       'unless';
COND    :       'cond';
FORK    :       'fork';
OR      :       'or';
```

```
XOR         :       'xor';
AND         :       'and';
NOT         :       'not';
IS          :       'is';
MOD         :       'mod';
TF          :       'true'|'false';
NULL        :       'null';
SOME        :       'some';
NONE        :       'none';
WITH        :       'with';
TRY         :       'try';
CATCH       :       'catch';
FINALLY     :       'finally';
RANGE       :       '..';
FATCOMMA
            :       '=>';
EQTEST      :       '=='|'!=';
ASSN        :       '='|'+='|'-='|'*='|'/='|'%='|'**='|'>>='|
    '<<='|'^='
            |       '/\\='|'\\/='|'&&='|'||=';
BOOLOR      :       '||';
BOOLAND     :       '&&';
MAGCOMP     :       '>'|'<'|'>='|'<=';
BITOR       :       '\\/';
BITXOR      :       '^';
BITAND      :       '/\\';
BITSHIFT            :       '>>'|'<<';
ADDSUB      :       '+'|'-';
EXP         :       '**';
STAR        :       '*';
MULT        :       '/'|'%';
BITNOT      :       '!'|'~';
PIPE        :       '|';
DOT
            :       '.'
            ;
LPAREN
            :       '('
            ;
COMMA
            :       ','
            ;
RPAREN
            :       ')'
            ;
```

```
LBRACK   :         '[';
RBRACK   :         ']';

LBRACE   :         '{';

RBRACE   :         '}';
//other stuff
INT  :   '0'..'9'+
     ;

FLOAT
     :   ('0'..'9')+    (
         {input.LA(2) != '.'}? => ('.' ('0'..'9')+
            EXPONENT? {$type = FLOAT;})
         |({$type = INT;})

     )
     |   ('0'..'9')+ EXPONENT
     ;



NEWLINE :         ('\r'? '\n')+
                  ;


WS   :   ( ' '
         | '\t'
         ) {skip();};



HEX       :         '0x' (HEX_DIGIT)+;

BYTE      :         '0b' ('1'|'0')+;

STRING
     :  '"' ( ESC_SEQ | ~('\\'|'"') )* '"'
     ;

EXPONENT : ('e'|'E') ('+'|'-')? ('0'..'9')+ ;

PUBPRIV :         'public'|'private';
```

```
NODEID  :       ('A'..'Z')('a'..'z'|'A'..'Z'|'0'..'9'|'_'
    )*;

ID  :   ('a'..'z'|'_') ('a'..'z'|'A'..'Z'|'0'..'9'|'_')*
    ;



fragment
HEX_DIGIT : ('0'..'9'|'a'..'f'|'A'..'F') ;

fragment
ESC_SEQ
    :   '\\' ('b'|'t'|'n'|'f'|'r'|'\"'|'\''|'\\');



INVALID
  :  . {
          errors.add("Invalid character: '" + $text + "' on
              line: " +
              getLine() + ", index: " +
                  getCharPositionInLine()); };
```

Listing 37: TreeWalker.java. Takes an AST and creates the Ruby code.

```java
import org.antlr.runtime.ANTLRStringStream;
import org.antlr.runtime.CommonTokenStream;
import org.antlr.runtime.RecognitionException;
import org.antlr.runtime.TokenStream;
import org.antlr.runtime.tree.CommonTree;
import java.io.*;
import org.antlr.runtime.*;
import java.util.*;


public class TreeWalker {
  LinkedList<CommonTree> printedAlready = new LinkedList<CommonTree>();
  HashSet<String> nodes = new HashSet<String>();

  public void walkTree(CommonTree t, String filename) {
    try {
      BufferedWriter out = new BufferedWriter(new FileWriter(filename
          + ".rb"));
      if (!(t.getType() == 0)) {

        walk((CommonTree) t, out);
      }
      // traverse all the child nodes of the root if root was empty
      else {
        walk((CommonTree) t, out);
      }
      out.close();

    } catch (IOException e) {
    }
  }

  public void walk(CommonTree t, BufferedWriter out) {
    try {

      if (t != null) {
        {
          // every unary operator needs to be preceded by a open
          // parenthesis and ended with a closed parenthesis
```

```java
if (printedAlready.contains((CommonTree) t)) {

} else
  switch (t.getType()) {
  case TanGParser.ADDSUB:
    // if the operation is binary, read the two children and
    // output that to the ruby code
    printedAlready.add(t);
    if (t.getChildCount() > 1) {
      out.write("(");
      walk((CommonTree) t.getChild(0), out);
      out.write(t.getText() + " ");
      walk((CommonTree) t.getChild(1), out);
      out.write(")");
    }
    // if the operation is a unary minus, surround the
    // right-hand side with parentheses
    // this is to differenciate between unary operators and
    // operations done within assignment operator
    else {
      if (t.getText().equals("- ")) {
        out.write("(");
        out.write(t.getText());
        walk((CommonTree) t.getChild(0), out);
        out.write(")");
      } else {
        walk((CommonTree) t.getChild(0), out);
      }
    }

    break;
    // binary operations like this simply prints out the 1st child,
    // the operation and the 2nd child
  case TanGParser.AND:
    printedAlready.add(t);
    out.write("(");
    walk((CommonTree) t.getChild(0), out);
    out.write(t.getText() + " ");
    walk((CommonTree) t.getChild(1), out);
    out.write(")");

```

```
                break;
                // expressions like these do not require translation and can
                // simply to outputed to the ruby file
            case TanGParser.ASSERT:
                printedAlready.add(t);
                out.write(t.getText() + " ");

                break;
            case TanGParser.ASSN:
                printedAlready.add(t);
                out.write("(");
                walk((CommonTree) t.getChild(0), out);
                out.write(t.getText() + " ");
                walk((CommonTree) t.getChild(1), out);
                out.write(")");

                break;
                // this operator and a few of the following operators are
                // different in ruby so a translation was necessary
            case TanGParser.BITAND:
                printedAlready.add(t);
                out.write("(");
                walk((CommonTree) t.getChild(0), out);
                out.write("& ");
                walk((CommonTree) t.getChild(1), out);
                out.write(")");

                break;
            case TanGParser.BITNOT:
                printedAlready.add(t);
                out.write("(");
                out.write(t.getText() + " ");
                walk((CommonTree) t.getChild(0), out);
                out.write(")");

                break;
            case TanGParser.BITOR:
                printedAlready.add(t);
                out.write("(");
                walk((CommonTree) t.getChild(0), out);
                out.write("| ");
                walk((CommonTree) t.getChild(1), out);
                out.write(")");
```

```java

            break;
        case TanGParser.BITSHIFT:
            printedAlready.add(t);
            out.write("(");
            walk((CommonTree) t.getChild(0), out);
            out.write(t.getText() + " ");
            walk((CommonTree) t.getChild(1), out);
            out.write(")");

            break;
        case TanGParser.BITXOR:
            printedAlready.add(t);
            out.write("(");
            walk((CommonTree) t.getChild(0), out);
            out.write("^ ");
            walk((CommonTree) t.getChild(1), out);
            out.write(")");

            break;
        case TanGParser.BOOLAND:
            printedAlready.add(t);
            out.write("(");
            walk((CommonTree) t.getChild(0), out);
            out.write(t.getText() + " ");
            walk((CommonTree) t.getChild(1), out);
            out.write(")");

            break;
        case TanGParser.BOOLOR:
            printedAlready.add(t);
            out.write("(");
            walk((CommonTree) t.getChild(0), out);
            out.write(t.getText() + " ");
            walk((CommonTree) t.getChild(1), out);
            out.write(")");

            break;
        case TanGParser.BREAK:
            printedAlready.add(t);
            out.write(t.getText() + " ");
            break;
        case TanGParser.BYTE:
            printedAlready.add(t);
            out.write(t.getText() + " ");

```

```java
                break;
            case TanGParser.COMMA:
                printedAlready.add(t);
                out.write(t.getText() + " ");

                break;
            case TanGParser.COMMENT:
                printedAlready.add(t);
                out.write(t.getText());
                break;
            case TanGParser.COND:
                printedAlready.add(t);
                // we start at the second child node and skip every other
                // one to skip the newlines
                out.write("case ");
                out.newLine();
                for (int j = 1; j < t.getChildCount(); j = j + 2) {
                    // for all the conditions, except the last, begin it
                    // with the keyword "when"
                    // begin the last condition with else
                    if (j < t.getChildCount() - 3) {
                        out.write("when ");
                        walk((CommonTree) t.getChild(j), out);
                        int k = 0;
                        while (!(((t.getChild(j).getChild(k)).getType()) == (TanGParser.NEWLINE))) {
                            k++;
                        }
                        while (k < t.getChild(j).getChildCount() - 1) {
                            walk((CommonTree) (t.getChild(j).getChild(k)),
                                out);
                            k++;
                        }
                    } else if (j == t.getChildCount() - 3) {
                        out.write("else ");
                        walk((CommonTree) t.getChild(j), out);
                        int k = 0;
                        while (!(((t.getChild(j).getChild(k)).getType()) == (TanGParser.NEWLINE))) {
```

```
                k++;
            }
            while (k < t.getChild(j).getChildCount() - 1) {
                walk((CommonTree) (t.getChild(j).getChild(k)),
                    out);
                k++;
            }
        } else {
            walk((CommonTree) t.getChild(j), out);
        }
    }

    break;
case TanGParser.CONTINUE:
    printedAlready.add(t);
    out.write("next ");

    break;
case TanGParser.DO:
    printedAlready.add(t);
    out.write(t.getText() + " ");
    break;
case TanGParser.DOT:
    printedAlready.add(t);
    out.write("(");
    walk((CommonTree) t.getChild(0), out);
    out.write(t.getText());
    walk((CommonTree) t.getChild(1), out);
    out.write(")");

    break;
case TanGParser.ELSE:
    printedAlready.add(t);
    out.write(t.getText() + " ");

    break;
case TanGParser.END:
    printedAlready.add(t);
    out.write(t.getText() + " ");
    out.newLine();

    break;
case TanGParser.EOF:
    break;
```

```
            case TanGParser.EQTEST:
              printedAlready.add(t);
              out.write("(");
              walk((CommonTree) t.getChild(0), out);
              out.write(t.getText() + "␣");
              walk((CommonTree) t.getChild(1), out);
              out.write(")");

              break;
            case TanGParser.ESC_SEQ:
              printedAlready.add(t);
              out.write(t.getText() + "␣");

              break;
            case TanGParser.EXP:
              printedAlready.add(t);
              out.write("(");
              walk((CommonTree) t.getChild(0), out);
              out.write(t.getText() + "␣");
              walk((CommonTree) t.getChild(1), out);

              out.write(")");
              break;
            case TanGParser.EXPONENT:
              printedAlready.add(t);
              // the power 10 operator in Tandem is simply e. It needs to
              // be transformed to ruby code.
              out.write("(");
              walk((CommonTree) t.getChild(0), out);
              out.write("*␣10␣**␣");
              walk((CommonTree) t.getChild(1), out);
              out.write(")");

              break;
            case TanGParser.FATCOMMA:
              printedAlready.add(t);
              out.write(t.getText() + "␣");
              break;
            case TanGParser.FILENAME:
              printedAlready.add(t);
              out.write("\"" + t.getText().substring(1,
                  t.getText().length() - 4)
                  + "\"␣");
              break;
            case TanGParser.FLOAT:
```

```
            printedAlready.add(t);
            out.write(t.getText() + " ");
            break;
          case TanGParser.FOR:
            printedAlready.add(t);
            out.write(t.getText() + " ");
            break;
          case TanGParser.FORK:
            printedAlready.add(t);
            out.write(t.getText() + " ");
            break;
          case TanGParser.FROM:
            printedAlready.add(t);
            out.write(t.getText() + " ");
            break;
          case TanGParser.FUNCID:
            printedAlready.add(t);
            out.write("td_" + t.getText() + " ");

            break;
          case TanGParser.HASHTOKEN:
            printedAlready.add(t);
            for (int i = 0; i < t.getChildCount(); i++)
              walk((CommonTree) t.getChild(i), out);
            break;
          case TanGParser.HEX:
            printedAlready.add(t);
            out.write(t.getText() + " ");
            break;
          case TanGParser.HEX_DIGIT:
            printedAlready.add(t);
            out.write(t.getText() + " ");
            break;
          case TanGParser.ID:
            printedAlready.add(t);
            if ((t.getParent()).getType() == TanGParser.DOT
                && t.getChildIndex() != 0) {
              String param = "";
              int w = (t.getParent().getParent()).getChildCount();
              int i = 0;

              while (t.getParent().getParent().getChild(i) != t
                  .getParent() && i < w) {
```

```
        i++;
      }
      i++;

      while (t.getParent().getParent().getChild
          (i) != null
          && t.getParent().getParent().getChild
              (i)
          .getType() != TanGParser.NEWLINE
          && i < w) {
        if (printedAlready.contains((CommonTree
            ) t.getParent().getParent().
            getChild(i))==false) {
          if ((t.getParent().getParent().
              getChild(i))
              .getType() == TanGParser.ID
              || (t.getParent().getParent()
                  .getChild(i).getType()) ==
                      TanGParser.FUNCID) {
            param = param
                + "td_"
                + t.getParent().getParent()
                .getChild(i).getText()
                + ", ";
          }else{
            param = param
                + t.getParent().getParent()
                .getChild(i).getText()
                + ", ";}



          printedAlready.add((CommonTree) t.
              getParent().getParent().getChild(i
              ));
        }else{

          printedAlready.remove(t.getParent().
              getParent().getChild(i));

        }
        i++;
      }

      if (param.length() > 0) {
```

```
                out.write(t.getText()
                    + "("
                    + param.substring(0, param.length()
                        - 2)
                    + ")");
            } else {
                out.write(t.getText() + "(" + param + "
                    )");
            }
        } else {
            if (t.getParent().getType() == TanGParser
                .DOT
                && t.getChildIndex() == 0) {
                out.write("td_" + t.getText());
            } else {
                out.write("td_" + t.getText() + " ");
            }

        }
        int q=0;
        while(t.getChild(q)!= null){
            walk((CommonTree) t.getChild(q), out);
            q++;
        }

        break;
    case TanGParser.IF:
        printedAlready.add(t);
        out.write(t.getText() + " ");
        break;
    case TanGParser.IMPORT:
        printedAlready.add(t);
        out.write("require_relative ");
        walk((CommonTree) t.getChild(0), out);
        int d=1;
        while(t.getChild(d)!= null){
            walk((CommonTree) t.getChild(d), out);
            d++;
        }
        break;
    case TanGParser.IN:
        printedAlready.add(t);
        out.write(t.getText() + " ");
        break;
    case TanGParser.INT:
        printedAlready.add(t);
```

```
            out.write(t.getText() + "␣");
            break;

        case TanGParser.IS:
            printedAlready.add(t);
            out.write("(");
            walk((CommonTree) t.getChild(0), out);
            out.write("==␣");
            walk((CommonTree) t.getChild(1), out);
            out.write(")");
            break;
        case TanGParser.LBRACE:
            printedAlready.add(t);
            out.write(t.getText());
            break;
        case TanGParser.LBRACK:
            printedAlready.add(t);
            out.write(t.getText());
            break;
        case TanGParser.LISTTOKEN:
            printedAlready.add(t);
            for (int i = 0; i < t.getChildCount(); i++)
              walk((CommonTree) t.getChild(i), out);
            break;
        case TanGParser.LOOP:
            printedAlready.add(t);
            out.write("while␣true␣");
            break;
        case TanGParser.LPAREN:
            printedAlready.add(t);
            out.write(t.getText());
            break;
        case TanGParser.MAGCOMP:
            printedAlready.add(t);
            out.write("(");
            walk((CommonTree) t.getChild(0), out);
            out.write(t.getText() + "␣");
            walk((CommonTree) t.getChild(1), out);
            out.write(")");
            break;
        case TanGParser.MAIN:
            printedAlready.add(t);
            for (int i = 0; i < t.getChildCount(); i++)
              walk((CommonTree) t.getChild(i), out);
            break;
        case TanGParser.MOD:
```

```
        printedAlready.add(t);
        out.write("(");
        walk((CommonTree) t.getChild(0), out);
        out.write(t.getText());
        walk((CommonTree) t.getChild(1), out);
        out.write(")");
        break;
      case TanGParser.MULT:
        printedAlready.add(t);
        out.write("(");
        walk((CommonTree) t.getChild(0), out);
        out.write(t.getText() + " ");
        walk((CommonTree) t.getChild(1), out);
        out.write(")");

        break;
      case TanGParser.NEWLINE:
        printedAlready.add(t);
        out.write("\n");
        break;
      case TanGParser.NODE:
        printedAlready.add(t);
        LinkedList<CommonTree> list = new LinkedList<CommonTree>();

        // every node will be converted to a class with the name of
        // the node as the class name
        if (t.getText().equals("public node")) {
          out.write("class ");
          out.write(t.getChild(0).getText());
          nodes.add(t.getChild(0).getText());
        }
        // if the class is private, add private after writing the
        // constructor of the class
        else {
          out.write("class ");
          out.write(t.getChild(0).getText());
          nodes.add(t.getChild(0).getText());
          out.newLine();
        }
        out.newLine();
        // then each class will have a main method with the node
        // definition code
```

```
      out.write("def main");
      for (int i = 1; i < t.getChildCount(); i++) {
        if (t.getChild(i).getType() == TanGParser.MAIN) {
          for (int k = 0; k < t.getChild(i).getChildCount(); k++) {
            if (t.getChild(i).getChild(k).getType() == TanGParser.NODE) {
              list.addLast(((CommonTree) t.getChild(i)
                  .getChild(k)));
            } else {
              walk((CommonTree) t.getChild(i).getChild(k),
                  out);
            }
          }
        } else {
          walk((CommonTree) t.getChild(i), out);
        }
      }

      while (list.isEmpty() == false) {
        walk((CommonTree) list.getFirst(), out);
        list.remove();
      }
      out.newLine();
      out.write("end\n");

      out.newLine();

      break;
    case TanGParser.NODEID:
      printedAlready.add(t);
      doCheck(t, out);

      break;
    case TanGParser.NOT:
      printedAlready.add(t);
      out.write(t.getText() + " ");
      walk((CommonTree) t.getChild(0), out);

      break;
    case TanGParser.NONE:
      printedAlready.add(t);
```

```
          out.write(t.getText() + " ");
          break;
        case TanGParser.NULL:
          printedAlready.add(t);
          out.write("nil ");
          break;
        case TanGParser.OR:
          printedAlready.add(t);
          out.write("(");
          walk((CommonTree) t.getChild(0), out);
          out.write(t.getText() + " ");
          walk((CommonTree) t.getChild(1), out);
          out.write(")");
          break;
        case TanGParser.PARENTOKEN:
          printedAlready.add(t);
          for (int i = 0; i < t.getChildCount(); i++)
            walk((CommonTree) t.getChild(i), out);
          break;
        case TanGParser.PIPE:
          printedAlready.add(t);
          LinkedList<CommonTree> paramOps = new LinkedList<CommonTree>();

          String params = "";
          CommonTree first = (CommonTree) t.getChild(0);
          LinkedList<CommonTree> list2 = new LinkedList<CommonTree>();
          for (int i = 0; i < t.getChildCount(); i++) {
            // if child is a node, but not the last node, push it
            if ((t.getChild(i).getType() == TanGParser.NODEID && i != t
                .getChildCount() - 1)) {
              list2.push((CommonTree) t.getChild(i));

            }else if (t.getChild(i).getType() == TanGParser.ID) {
              paramOps.add((CommonTree) t.getChild(i));
              //params = params + "td_" + t.getChild(i) + ",";

```

```
}else if (t.getChild(i).getType() != TanGParser.NODEID

    ){

  paramOps.add((CommonTree) t.getChild(i));
}
// if next token is a pipe, push it
else if (t.getChild(i).getType() != TanGParser.NODEID && (t.getChild(i).getType() != TanGParser.ID)) {
  list2.push((CommonTree) t.getChild(i));

}
// if next token is an id, it is a parameter so it is
// not pushed
// when we walk the node that has the parameters (the
// first node), we will print them
else if (i == t.getChildCount() - 1){
  // walk the tree if the child is the last node in
  // the chain
  walk((CommonTree) t.getChild(i), out);
  while (list2.isEmpty() == false) {
    out.write("(");

    if ((list2.peek()) == first) {
      walk((CommonTree) list2.pop(), out);
      if(paramOps.size()>0){
        out.write("(");
        while(paramOps.isEmpty()==false){
          walk((CommonTree)paramOps.pop(),out);
          if(!(paramOps.isEmpty()))
            out.write(", ");

        }

        out.write(")");}
    } else {
      walk((CommonTree) list2.pop(), out);
```

```
                }
                out.write(")");
              }
            }
            else {
              paramOps.add((CommonTree) t.getChild(i));

            }
          }
          break;
        case TanGParser.PIPEROOT:
          printedAlready.add(t);
          for (int i = 0; i < t.getChildCount(); i++)
            walk((CommonTree) t.getChild(i), out);
          break;

        case TanGParser.PUBPRIV:
          printedAlready.add(t);
          out.write(t.getText() + " ");
          break;
        case TanGParser.RANGE:
          printedAlready.add(t);
          walk((CommonTree) t.getChild(0), out);
          out.write(t.getText());
          walk((CommonTree) t.getChild(1), out);

          break;
        case TanGParser.RBRACE:
          printedAlready.add(t);
          out.write(t.getText());
          break;
        case TanGParser.RBRACK:
          printedAlready.add(t);
          out.write(t.getText());
          break;
        case TanGParser.REQUIRE:
          printedAlready.add(t);
          out.write("require_relative " + t.getChild(0));
          int e=1;
          while(t.getChild(e)!= null){
            walk((CommonTree) t.getChild(e), out);
            e++;
          }
          break;
```

```
case TanGParser.RETURN:
  printedAlready.add(t);
  out.write(t.getText() + " ");
  break;
case TanGParser.RPAREN:
  printedAlready.add(t);
  out.write(t.getText());
  break;
case TanGParser.ROOTNODE:
  printedAlready.add(t);
  for (int i = 0; i < t.getChildCount(); i++)
    walk((CommonTree) t.getChild(i), out);
  break;
case TanGParser.SOME:
  printedAlready.add(t);
  out.write(t.getText() + " ");
  break;
case TanGParser.STAR:
  printedAlready.add(t);
  out.write("(");
  walk((CommonTree) t.getChild(0), out);
  out.write(t.getText() + " ");
  walk((CommonTree) t.getChild(1), out);
  out.write(")");
  break;
case TanGParser.STRING:
  printedAlready.add(t);
  out.write(t.getText() + " ");
  break;
case TanGParser.TF:
  printedAlready.add(t);
  out.write(t.getText() + " ");
  break;
case TanGParser.UNLESS:
  printedAlready.add(t);
  out.write(t.getText() + " ");
  break;
case TanGParser.UNTIL:
  printedAlready.add(t);
  out.write(t.getText() + " ");
  break;
case TanGParser.WHILE:
  printedAlready.add(t);
  out.write(t.getText() + " ");
  break;
case TanGParser.WS:
```

```
            printedAlready.add(t);
            for (int i = 0; i < t.getChildCount(); i++)
              walk((CommonTree) t.getChild(i), out);
            break;
          case TanGParser.XOR:
            printedAlready.add(t);
            out.write("(");
            walk((CommonTree) t.getChild(0), out);
            out.write("^ ");
            walk((CommonTree) t.getChild(1), out);
            out.write(")");
            break;
          case 0:

            for (int i = 0; i < t.getChildCount(); i++)
              walk((CommonTree) t.getChild(i), out);
            break;
          }

      }
    }} catch (IOException e) {
    }
}
private void doCheck(CommonTree t, BufferedWriter out) {
  try {


    if (t.getParent().getType() != 0
        && t.getParent().getType() != TanGParser.PIPE
        && t.getParent().getType() != TanGParser.DOT) {
      LinkedList<CommonTree> pList = new LinkedList<CommonTree>();
      int w = (t.getParent()).getChildCount();
      int i = 0;
      while (!(t.getParent().getChild(i).toStringTree().equals(t
          .toStringTree())) && i < w) {
        i++;
      }
      i++;
      while (t.getParent().getChild(i) != null
          && !(t.getParent().getChild(i).getText().contains("\n"))
          && i < w) {
```

```
      if (t.getParent().getChild(i).getType() == TanGParser.ID
          || t.getParent().getChild(i).getType() == TanGParser.FUNCID) {

        pList.addLast((CommonTree)t.getParent().getChild(i));
      } else {

        pList.addLast((CommonTree)t.getParent().getChild(i));
      }

      i++;
    }

    if (t.getText().equals("E")) {
      out.write("Math::E ");
    } else if (t.getText().equals("PI")) {

      out.write("Math::PI ");
    } else if (t.getText().equals("Print")) {
      if (pList.size() > 0) {
        out.write("Kernel.print(");
        while(!(pList.isEmpty())){
          walk((CommonTree)pList.pop(), out);
          if(!(pList.isEmpty())){
            out.write(", ");
          }
        }
        out.write(")");
      } else {
        out.write("print()");
      }
    } else if (t.getText().equals("Println")) {
      if (pList.size() > 0) {
        out.write("Kernel.puts(");
        while(!(pList.isEmpty())){
          walk((CommonTree)pList.pop(), out);
          if(!(pList.isEmpty())){
            out.write(", ");
          }
        }
        out.write(")");


```

```
        } else {
          out.write("puts()");
        }
      }
      // this set checks if NodeID a system function or not. if not,
      // .main is added

      else  if (pList.size() > 0) {
        out.write(t.getText() + ".new().main(");
        while(!(pList.isEmpty())){
          walk((CommonTree)pList.pop(), out);
          if(!(pList.isEmpty())){
            out.write(", ");
          }
        }
        out.write(")");

      } else {
        out.write(t.getText() +  ".new().main()");

      }

    }
    else if (t.getText().equals("E"))
    {

      out.write("Math::E ");

    } else if (t.getText().equals("PI")) {

      out.write("Math::PI ");
    }
    else if (t.getText().equals("Print")) {

      out.write("Kernel.print");

    } else if (t.getText().equals("Println")) {

      out.write("Kernel.puts");

    }
    else if (t.getParent().getType() == TanGParser.DOT) {
      out.write(t.getText());

```

```
        }
        else {
          out.write(t.getText() + ".new().main");
        }

      } catch (IOException e) {
      }
    }
}
```

Listing 38: TandemTree.java. Main driver class. Calls lexer, parser, and tree walker.

```java
import org.antlr.runtime.ANTLRStringStream;
import org.antlr.runtime.CommonTokenStream;
import org.antlr.runtime.RecognitionException;
import org.antlr.runtime.TokenStream;
import org.antlr.runtime.tree.CommonTree;
import org.antlr.runtime.tree.*;
import java.io.*;
import org.antlr.runtime.*;
import org.antlr.stringtemplate.*;

//Donald Pomeroy

public class TandemTree{
  public void printTree(CommonTree t, int indent) {
    if ( t != null ) {
      StringBuffer sb = new StringBuffer(indent);
      for ( int i = 0; i < indent; i++ )
        sb = sb.append("   ");
      for ( int i = 0; i < t.getChildCount(); i++ ) {
        System.out.println(sb.toString() + t.getChild(i).
            toString());
        printTree((CommonTree)t.getChild(i), indent+1);
      }
    }
  }

  public static void main(String args[]){
    try{

      TanGLexer lex = new TanGLexer(new ANTLRInputStream(
          new FileInputStream(args[0])));
      Token token;
      TokenStream ts = new CommonTokenStream(lex);
      lex.reset();
      TanGParser parse = new TanGParser(ts);
      TanGParser.tanG_return result = parse.tanG();
      CommonTree t = (CommonTree)result.getTree();
      TreeWalker walk = new TreeWalker();
      walk.walkTree(t, args[0].substring(0, args[0].
          length()-3));
      TandemTree Tr = new TandemTree();
    //  Tr.printTree(t, 2);
      DOTTreeGenerator gen = new DOTTreeGenerator();
```

```
            StringTemplate st = gen.toDOT(t);
            try {
                BufferedWriter out = new BufferedWriter(new FileWriter("graph.txt"));
                out.write(st.toString());
                out.close();
            } catch (IOException e) {}
        } catch(Exception e) {
            e.printStackTrace();
            System.err.println("exception: "+e);
        }
    }
}
```

Listing 39: build.xml. Ant file used to compile grammar, tree walker, test files. Downloads dependencies and runs tests.

```
<!-- build.xml -->
<!-- Written by Varun Ravishankar -->

<project xmlns:ivy="antlib:org.apache.ivy.ant" name="Tandem" default="compile" basedir=".">
  <description>
    build file for Tandem programming language.
  </description>

  <property name="project.name" value="Tandem" />

  <!-- program version -->
  <property name="version" value="0.1" />

  <!-- set global properties for this build -->
  <property name="src" location="src"/>
  <property name="build" location="bin"/>
  <property name="dist"  location="dist"/>
  <property name="test" location="test"/>
  <property name="tutorial" location="tutorial"/>
  <property name="lib" location="lib"/>
  <property name="grammar" value="TanG.g"/>
  <property name ="treeGrammar" value="TanG_TG.g"/>
  <property name="grammar1" value="WateredDownTanG.g"/>
  <property name="gunit_test" value="TanG.gunit" />
  <property name="test.class.name" value="TandemTest" />
  <property name="jruby_jar" value="jruby-complete.jar" />

  <!-- Properties required to download Ivy. -->
  <property name="ivy.install.version" value="2.3.0-rc1" />
  <condition property="ivy.home" value="${env.IVY_HOME}">
    <isset property="env.IVY_HOME" />
  </condition>
  <property name="ivy.jar.dir" value="${lib}/ivy" />
  <property name="ivy.jar.file" value="${ivy.jar.dir}/ivy.jar" />

  <property name="ivy.lib.dir" value="${lib}" />

  <path id="lib.path.id">
    <fileset dir="${ivy.lib.dir}" includes="*.jar"/>
```

```
  </path>


  <!-- Path required to run the tests. Uses built-in JUnit,
   if one is installed, or the provided JUnit otherwise. -->
  <path id="test.classpath">
    <fileset dir="${lib}">
      <include name="*.jar"/>
    </fileset>
    <fileset dir="${lib}/test">
      <include name="**/*.jar"/>
    </fileset>
    <pathelement location="${java.class.path}" />
  </path>

  <!-- Path required to build the grammar. Uses downloaded ANTLR,
   or the installed ANTLR otherwise. -->
  <path id="build.classpath">
    <fileset dir="${lib}">
      <include name="*.jar"/>
    </fileset>
    <fileset dir="${lib}/build">
      <include name="**/*.jar"/>
    </fileset>
    <pathelement location="${java.class.path}" />
  </path>

  <!-- Path required to build the grammar. Uses downloaded ANTLR,
   or the installed ANTLR otherwise. -->
  <path id="runtime.classpath">
    <fileset dir="${lib}">
      <include name="*.jar"/>
    </fileset>
    <fileset dir="${lib}/runtime">
      <include name="**/*.jar"/>
    </fileset>
    <pathelement location="${java.class.path}" />
  </path>


  <!-- An ant macro which invokes ANTLR3
```

```
        This is just a parameterizable wrapper to simplify the invocation of ANTLR3.
        The default values can be overriden by assigning a value to an attribute
        when using the macro.
        Example with ANTLR3 outputdirectory modified:
        <antlr3 grammar.name="TanG.g" outputdirectory="${src}/${package}"/>
  -->

<taskdef resource="org/apache/tools/ant/antlr/antlib.xml"
  classpath="${lib}/antlr3-task/ant-antlr3.jar" />


  <macrodef name="antlr3">
    <attribute name="grammar.name"/>
    <attribute name="outputdirectory" default="${lib}/grammar"/>
    <attribute name="libdirectory" default="${lib}"/>
    <attribute name="multithreaded" default="true"/>
    <attribute name="verbose" default="true"/>
    <attribute name="report" default="false"/>
    <attribute name="debug" default="false"/>
    <sequential>
      <ant-antlr3 xmlns:antlr="org/apache/tools/ant/antlr"
        target="@{grammar.name}"
        outputdirectory="@{outputdirectory}"
        libdirectory="@{libdirectory}"
        multithreaded="@{multithreaded}"
        verbose="@{verbose}"
        report="@{report}"
        debug="@{debug}">
        <classpath>
          <pathelement location="${lib}/antlr3-task/ant-antlr3.jar"/>
          <path refid="build.classpath"/>
        </classpath>
        <jvmarg value="-Xmx512M"/>
      </ant-antlr3>
    </sequential>
  </macrodef>

  <target name="download-ivy" unless="offline">
    <mkdir dir="${ivy.jar.dir}"/>
```

```
    <!-- download Ivy from web site so that it can be used even without any special installation -->
    <get src="http://repo2.maven.org/maven2/org/apache/ivy/ivy/${ivy.install.version}/ivy-${ivy.install.version}.jar"
      dest="${ivy.jar.file}" usetimestamp="true"/>
  </target>

  <target name="install-ivy" depends="download-ivy">
    <!-- try to load ivy here from ivy home, in case the user has not already dropped
          it into ant's lib dir (note that the latter copy will always take precedence).
        We will not fail as long as local lib dir exists (it may be empty) and
        ivy is in at least one of ant's lib dir or the local lib dir. -->
    <path id="ivy.lib.path">
      <fileset dir="${ivy.jar.dir}" includes="*.jar"/>
    </path>

    <taskdef resource="org/apache/ivy/ant/antlib.xml"
      uri="antlib:org.apache.ivy.ant" classpathref="ivy.lib.path"/>
   </target>

   <target name="resolve" depends="install-ivy" description="Resolve the dependencies">
     <ivy:retrieve pattern="lib/[conf]/[artifact](-[classifier]).[ext]"/>
   </target>

   <target name="init" depends="resolve">
    <!-- Create the time stamp -->
    <tstamp/>
    <!-- Create the build directory structure used by compile -->
    <mkdir dir="${build}"/>
    <mkdir dir="${dist}"/>
    <mkdir dir="${lib}"/>
    <mkdir dir="${lib}/grammar"/>
  </target>

  <target name="antlr_classpath">
    <whichresource property="antlr.in.classpath" class="org.antlr.Tool"/>
```

```
    <fail message="ANTLR3 not found via CLASSPATH ${java.class.path}">
      <condition>
        <not>
          <isset property="antlr.in.classpath"/>
        </not>
      </condition>
    </fail>
  </target>

  <target name="grammar" depends="init" description="compile the grammar" >
    <!-- Compile the grammar from ${src} into ${build} -->
    <antlr3 grammar.name="${src}/${grammar}" outputdirectory="${lib}/grammar"/>
    <!-- <antlr3 grammar.name="${src}/${treeGrammar}" outputdirectory="${lib}/grammar"/> -->
  </target>


  <target name="compile" depends="grammar" description="compile the source " >
    <!-- Compile the java code from ${src} into ${build} -->
    <!-- includeantruntime="false" -->
    <javac destdir="${build}" includeantruntime="false" verbose="false">
      <classpath>
        <path refid="build.classpath"/>
      </classpath>
      <src path="${lib}/grammar" />
    </javac>
    <javac destdir="${build}" includeantruntime="false" verbose="false">
      <!-- <classpath path="${build}" /> -->
      <classpath>
        <path refid="test.classpath"/>
      </classpath>
      <src path="${src}" />
      <exclude name="**/Test.java"/>
      <!-- <exclude name="**/TreeWalker.java"/> -->
    </javac>
  </target>

  <target name="dist" depends="compile"
```

```
      description="generate the distribution" >
    <!-- Create the distribution directory -->
    <mkdir dir="${dist}/lib"/>

    <!-- Put everything in ${build} into the Tandem-${DSTAMP}.jar file -->
    <jar jarfile="${dist}/lib/Tandem-${DSTAMP}.jar" basedir="${build}"/>
  </target>

  <target name="parse_test" depends="compile">
    <junit showoutput="no" fork="yes" haltonfailure="no">
      <test name="${test.class.name}" />
      <formatter type="plain" usefile="false" />
      <classpath>
        <pathelement path="${build}"/>
          <path refid="test.classpath"/>
      </classpath>
    </junit>
  </target>

  <target name="gunit_test" depends="compile">
    <copy file="${src}/${gunit_test}" tofile="${build}/${gunit_test}"/>
    <java classname="org.antlr.gunit.Interp">
      <arg value="${build}/${gunit_test}"/>
      <classpath>
        <pathelement path="${build}"/>
          <path refid="runtime.classpath"/>
          <path refid="test.classpath"/>
      </classpath>
    </java>
  </target>


  <target name="test" depends="gunit_test, parse_test">

  </target>

  <target name="walk" depends="compile">
    <java classname="TandemTree">
      <arg value="${file}"/>
      <classpath>
        <pathelement path="${build}"/>
          <path refid="runtime.classpath"/>
      </classpath>
```

```
    </java>
  </target>

  <target name="rubyexec" depends="init">
    <java jar="${lib}/runtime/${jruby_jar}" fork="true">
      <arg value="${file}"/>
      <classpath>
        <pathelement path="${build}"/>
          <path refid="runtime.classpath"/>
      </classpath>
    </java>
  </target>

  <target name="rubytest" depends="compile">
    <java jar="${lib}/runtime/${jruby_jar}" fork="true">
      <arg value="test/tutorial/geometry_question1_test.rb"/>
      <classpath>
        <pathelement path="${build}"/>
          <path refid="runtime.classpath"/>
      </classpath>
    </java>
    <java jar="${lib}/runtime/${jruby_jar}" fork="true">
      <arg value="test/tutorial/geometry_question2_test.rb"/>
      <classpath>
        <pathelement path="${build}"/>
        <path refid="runtime.classpath"/>
      </classpath>
    </java>
    <java jar="${lib}/runtime/${jruby_jar}" fork="true">
      <arg value="test/tutorial/geometry_question2_test.rb"/>
      <classpath>
        <pathelement path="${build}"/>
        <path refid="runtime.classpath"/>
      </classpath>
    </java>
    <java jar="${lib}/runtime/${jruby_jar}" fork="true">
      <arg value="test/tutorial/geometry_question2_test.rb"/>
      <classpath>
        <pathelement path="${build}"/>
        <path refid="runtime.classpath"/>
      </classpath>
    </java>
```

```
  </target>

  <target name="clean" description="clean up" >
    <!-- Delete the ${build} and ${dist} directory trees -->
    <delete dir="${build}"/>
    <delete dir="${dist}"/>
    <delete dir="${lib}/grammar"/>
  </target>

    <!-- =================================
          target: clean-ivy
         ================================= -->
  <target name="clean-ivy" depends="clean-downloads" description="--> clean the ivy installation">
    <delete dir="${ivy.jar.dir}"/>
  </target>

  <target name="clean-downloads" description="--> clean the downloaded jars">
    <delete dir="${ivy.lib.dir}/build"/>
    <delete dir="${ivy.lib.dir}/runtime"/>
    <delete dir="${ivy.lib.dir}/test"/>
  </target>

    <!-- =================================
          target: clean-cache
         ================================= -->
  <target name="clean-cache" depends="install-ivy"
      description="--> clean the ivy cache">
    <ivy:cleancache />
  </target>
</project>
```

Listing 40: ivy.xml. Contains dependency listing for Tandem.

```
<!-- ivy.xml -->
<!-- Written by Varun Ravishankar -->

<ivy-module version="2.0">
    <info organisation="edu.columbia.comsw4115-spring2012
        .tandem" module="tandem-deps"/>
    <configurations>
      <conf name="build"  description="build dependancies
          "/>
      <conf name="runtime" description="runtime 
          dependancies"/>
      <conf name="test" extends="build, runtime"
          description="junit dependances"/>
  </configurations>
    <dependencies>
    <dependency org="org.antlr" name="antlr" rev="3.4"
        conf="build->default"/>
    <dependency org="org.antlr" name="ST4" rev="4.0.4"/>
    <dependency org="org.antlr" name="gunit" rev="3.4"
        conf="build->default"/>
        <dependency org="junit" name="junit" rev="4.10"
            conf="test->default"/>
        <dependency org="org.jruby" name="jruby-complete"
             rev="1.6.7.1" conf="runtime->default"/>
    </dependencies>
</ivy-module>
```

Listing 41: tandem. Calls Ant file and runs Ruby output.

```
#!/usr/bin/env bash

# Written by Donald Pomeroy

EXPECTED_ARGS=1
E_BADARGS=65

#Check if a file is passed, if not fail

if [ $# -ne $EXPECTED_ARGS ]
then
  echo "Usage: `basename $0` {arg}"
  echo "COMPILATION FAILED"
  exit $E_BADARGS
fi

FILE=$1

#Check if the filename is valid .td

if [[ "$FILE" == *.td ]]
then
  echo "valid filename"
else
     echo "COMPILATION FAILED - invalid filename"
     exit
fi

if [ -f "$FILE" ]
then
    echo "the file exists"
else
    echo "COMPILATION FAILED - the file does not exist"
    exit
fi

#check java
#check ant 1.8
#check ruby 1.9.2

if type -p java; then
    echo "found java executable in PATH"
    _java=java
```

```
elif [[ -n "$JAVA_HOME" ]] && [[ -x "$JAVA_HOME/bin/java" ]]; then
    echo "found java executable in JAVA_HOME"
    _java="$JAVA_HOME/bin/java"
else
    echo "COMPILATION FAILED - no java"
    exit
fi

if [[ "$_java" ]]; then
    version=$("$_java" -version 2>&1 | awk -F '"' '/version/ {print $2}')
    #echo version "$version"
    if [[ "$version" > "1.5" ]]; then
        echo "version is more than 1.5"
    else
        echo "COMPILATION FAILED - version is less than 1.5"
        exit
    fi
fi



if type -p ant; then
    echo found ant executable in PATH
    _ant=ant
elif [[ -x "usr/bin/env ant" ]]; then
    echo "found ant executable in ENV_ANT"
    _ant="usr/bin/env ant"
else
    echo "COMPILATION FAILED - no ant"
    exit
fi

if [[ "$_ant" ]]; then
    version=$("$_ant" -v 2>&1 | awk -F ' ' '/ / {print $4}')
    echo version "$version"
    if [[ "$version" > "1.8.0" ]]; then
        echo "version is more than 1.8"
    else
        echo "COMPILATION FAILED - version is less than 1.8"
        exit
    fi
```

```
fi

#Make JRuby use Ruby 1.9

JRUBY_OPTS=--1.9


if type -p ruby; then
    echo "found ruby executable in PATH"
    _ruby=ruby
elif [[ -x "usr/bin/env ruby" ]];  then
    echo "found ruby executable in ENV"
    _ruby="usr/bin/env ruby"
else
    echo "COMPILATION FAILED - no ruby"
    exit
fi

use_jruby=0

if [[ "$_ruby" ]]; then
    version=$("$_ruby" -v 2>&1 |grep -e "1\.9\.[2-9]" )
   # echo $?
    if [[ "$?" > "0" ]]; then
    use_jruby=1
    else
    echo "Passed Ruby Check"
    fi
fi


DIR=$(pwd)

#$(echo $IN | tr ";" "\n")

a=$(ruby src/dependency.rb "$FILE" | tr " " "\n")

for x in $a
do
        ant walk -Dfile="$x" -q
done


echo "compiling ..."

ant walk -Dfile="$FILE" -q
```

```
ruby_file=${FILE%.*}

#run a different version of ruby based on ant rubyexec

if [[ "$_use_jruby" > 0 ]]; then
        java -jar lib/runtime/jruby-complete.jar --1.9 "$ruby_file"
        else
        ruby "$ruby_file".rb
fi
```

Listing 42: tandem. Calls Ant file and runs Ruby output.

```
#! /usr/bin/env ruby
#Donald Pomeroy
require "set"
require 'pathname'


dependency_q = []
dependency_set =  Set.new
file = ARGV[0]
dir = ARGV[1]
#puts "flag 1 " + file
#puts "flag 2 " + dir
dependency_set << file


split_arr = ARGV[0].split(%r{\/})
split_arr.pop

path_to_dir=""

split_arr.each do|n|
    path_to_dir += n+"/"
end

#puts "flag 5" + path_to_dir

Dir.chdir(path_to_dir)

def dep_helper(d_q,d_set,  d_file)
          dependency_array = File.open(d_file, "r").grep(/import\s+\".*.td\"/)
␣␣␣␣␣␣␣␣x␣=␣[]
␣␣␣␣␣␣␣␣dependency_array.each␣do␣|y|
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣y␣=␣y.chomp
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣t␣=␣y.split(%r{\s+})[1]
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣x␣<<␣t[1..(t.size-2)]
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣d_q.unshift␣t[1..(t.size-2)]␣if␣not␣d_set.include?(t[1..(t.size-2)])
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣d_set␣<<␣t[1..(t.size-2)]
␣␣␣␣␣␣␣␣end
end


dep_helper(dependency_q,␣dependency_set,file)
```

```

while(dependency_q.size != 0) do
        #puts Dir.getwd
        puts  dependency_q.last
        path = Pathname.new(dependency_q.last)
        #puts path
        #puts " flag 4 "+ File.absolute_path(dependency_q.last)
       # puts (Dir.getwd + "/"+dependency_q.last)
        #puts File.dirname( dependency_q.last)
        dep_helper(dependency_q, dependency_set, dependency_q.pop)
end



#get first file , add to q and add to set , call dep_help on the front of q until is empty , if d_set doesn't contain add to list
```

Listing 43: TanG.gunit. Unit tests for grammar.

```
//Patrick De La Garza - Language Guru
gunit TanG;




//Test Parser

//Test tanG production
tanG :
//This should fail
<<
node node1(a, b)
  cond
    a>b
      x=a+b
      node2 x
    end
    b>a
      y=b-a
      node3 b
    end
    a==b
      node4 a b

      x=y=z=1

    end
  end
  node node2(greaterA)
    print "I am at node2"
    greaterA-4
  end
  node node3(greaterB)
    answer =greaterB-8
    node4 answer greaterB | node2
    a|b|c|d|e|f
  end
  node node4(myA, myB)
    a = myA *5
    b = myB -5
    node5 a b
    node node5(g,h)
      sum = "I am at node 5"
```

```
        print sum
      end
    end
end
>>FAIL //expect failure
<<
node Node1(a, b)
    cond
      a>b
        x=a+b
        Node2 x
      end
      b>a
        y=b-a
        A b c| D e| F
        Node3 b
      end
      a==b
        Node4 a b

        x=y=z=1

      end
    end
    node Node2(greaterA)
      print "I am at node2"
      greaterA-4
    end
    node Node3(greaterB)
      answer =greaterB-8
      Node4 answer greaterB | Node2
      A|B|C|D|E|F
    end
    node Node4(myA, myB)
      a = myA *5
      b = myB -5
      Node5 a b
      node Node5(g,h)
        sum = "I am at node 5"
        Print sum
      end
    end
end
>> FAIL //expect failure

//Should Succeed
```

```
<<
import "File.td"
import "File2.td"

import "StandardLib.td"

node Node1(a, b)
  cond
    a>b
      x=a+b
      Node2 x
    end
    b>a
      y=b-a
      A b c| D| F
      Node3 b
    end
    a==b
      Node4 a b

      x=y=z=1

    end
  end
  node Node2(greaterA)
    Print "I␣am␣at␣node2"
    greaterA-4
  end
  node Node3(greaterB)
    answer =greaterB-8
    Node4 answer greaterB | Node2
    A|B|C|D|E|F
  end
  node Node4(myA, myB)
    a = myA *5
    b = myB -5
    Node5 a b
    node Node5(g,h)
      sum = "I␣am␣at␣node␣5"
      Print sum
    end
  end
end>>OK


//Test imports
```

```
i :
<<import "success.td"
import "mobetter.td">> OK

<<require "test">>OK

<<import "success.td"
require "rubyfile"
import "test.td">>OK

<<import require "game">>FAIL

<<
  import "success.td"
>>OK


//Should fail
<<import fail.td>>FAIL



//Main Body Test
m :

//cond failures
<<cond
  a is b
    a is c
      1+1
    end
  end
end>>FAIL

//assert test
<<assert a is b>>OK

<<assert assert>>FAIL

<<break true>>OK

<<assert loop
  break
end>>FAIL

//should succeed
```

```
<<a=b>>OK
//Should fail, no NodeCode again!
<<2|A>>FAIL

//Should fail, no NodeCode again!
<<2 A>>FAIL

<<2 a b>>FAIL

<<3|4|5>>FAIL

<<a|b|b>>FAIL

<<"Car"|"A"|C>>FAIL

//Should succeed; proper pipeline
<<A a b|B|C>>OK

//should fail: no NodeCode in the pipeline
<<A|B + 2|C>>FAIL


//This should fail
<<node node1(a, b)
  cond
    a>b
      x=a+b
      node2 x
    end
    b>a
      y=b-a
      node3 b
    end
    a==b
      node4 a b

      x=y=z=1

    end
  end
  node node2(greaterA)
    print "I am at node2"
    greaterA-4
  end
  node node3(greaterB)
    answer =greaterB-8
```

```
    node4 answer greaterB | node2
    a|b|c|d|e|f
  end
  node node4(myA, myB)
    a = myA *5
    b = myB -5
    node5 a b
    node node5(g,h)
      sum = "I␣am␣at␣node␣5"
      print sum
    end
  end
end
>>FAIL //expect failure
<<node Node1(a, b)
  cond
    a>b
      x=a+b
      Node2 x
    end
    b>a
      y=b-a
      A b c| D e| F
      Node3 b
    end
    a==b
      Node4 a b

      x=y=z=1

    end
  end
  node Node2(greaterA)
    print "I␣am␣at␣node2"
    greaterA -4
  end
  node Node3(greaterB)
    answer =greaterB -8
    Node4 answer greaterB | Node2
    A|B|C|D|E|F
  end
  node Node4(myA, myB)
    a = myA *5
    b = myB -5
    Node5 a b
    node Node5(g,h)
```

```
        sum = "I am at node 5"
        Print sum
      end
    end
end
>> FAIL //expect failure

//Should Succeed
<<node Node1(a, b)
  cond
    a>b
      x=a+b
      Node2 x
    end
    b>a
      y=b-a
      A b c| D| F
      Node3 b
    end
    a==b
      Node4 a b

      x=y=z=1

    end
  end
  node Node2(greaterA)
    Print "I am at node2"
    greaterA-4
  end
  node Node3(greaterB)
    answer =greaterB-8
    Node4 answer greaterB | Node2
    A|B|C|D|E|F
  end
  node Node4(myA, myB)
    a = myA *5
    b = myB -5
    Node5 a b
    node Node5(g,h)
      sum = "I am at node 5"
      Print sum
    end
  end
end>>OK

```

```
//should fail
<<1[2]>>FAIL




//Expression Unit Tests

expression:

//Should fail, no NodeCode in the PIPELINE!!
<<A|B|(C+2)>>FAIL

//Should fail, no NodeCode again!
<<A|2>>FAIL

//should succeed: note that it is actually two pipelines (pipe has higher precedence than +)
<<A|B+C|D>>OK


//LoopTests
loopType:

//for-loop
<<for item in list
  stuff
end>>OK

<<for item in list in biggerList
  makeithappen
end>>FAIL

//while-loop
<<while x>2
  x=x+1
end>>OK

<<while return true
  beelzebub
end>>FAIL

//loop
<<loop
  break x
end>>OK
```

```

<<loop true
  nope
end>>FAIL

//until
<<until pigsFly
  !hellFreezeOver
end>>OK

<<until true false
  asdf
end>>FAIL



//condTypes
condType:
//if-statements

<<if x is y
  x is z
else
  x is a
end>>OK

<<if x is y
  skipElse
end>>FAIL

//unless

<<unless player is charlieParker
  not listen
end>>OK

<<unless if true
  wait
end>>FAIL



//cond-statements

<<cond
  a>b
```

```
    jazz
  end
end>>OK

<<cond
  a is b
    1+1
  end
  a is c
    1+2
  end
end>>OK



//SPECIAL ATOMS

//list
list:
<<[a is b, c, d, [1,2]]>>OK
<<[a,b,[c,d]>>FAIL

//hash
hashSet:
//hash success
<<{"jam" => 0b0110, "jar" => 0xAFFC2}>>OK
//hash fail
<<{=>}>>FAIL
<<{a,b,c}>>FAIL
<<{a<=c=>d<=e}>>OK


//Lexer Tests

//Comment Tests
COMMENT :

//success
<<#this is a comment>>OK

//success
<<//This is also a comment>>OK

<</This is not a comments>>FAIL

//success
```

```
<<//#//#//#?>>OK


//Float stuff
FLOAT:
//success
<<1.1e+99>>OK

//fail due to improper exponent
<<1.1e-9.9>>FAIL

//float stuff
<<.0978>FAIL

//float stuff
<<.0E–0>>FAIL

//success
<<99e-99>>OK


//Test hex
HEX:
<<0x09aAFff>>OK

<<0x>>FAIL

//Test Byte
BYTE:
<<0b010010110>>OK
<<0b>>FAIL
<<0b012>>FAIL

//TEST string
STRING:
<<"WithotEscape codes babe">>OK
<<"Dog\nDog on new line">>OK
<<"DOG>>FAIL

//TEST FUNCID
FUNCID:
<<Abcdefg?>>OK
<<gu?ess?>>FAIL
<<empty?>>OK
```

Listing 44: TandemTest.java. Unit tests for TandemTree. Checks if parser works on Tandem file.

```java
// TandemTest.java
// Written by Varun Ravishankar

import org.antlr.runtime.ANTLRStringStream;
import org.antlr.runtime.CommonTokenStream;
import org.antlr.runtime.RecognitionException;
import org.antlr.runtime.TokenStream;
import org.antlr.runtime.tree.CommonTree;
import java.io.*;
import org.antlr.runtime.*;
import org.junit.Test;
import org.junit.BeforeClass;
import static org.junit.Assert.*;


public class TandemTest
{
    private static File currentDir = new File(".");
    private static String currentDirName;
    private static String testPath;
    private final static String whitespace = "misc/whitespace/";
    private final static String comments = "misc/comments/";
    private final static String expression = "expression/";
    private final static String mathexp = "expression/math";
    private final static String bitwiseexp = "expression/bitwise";
    private final static String logicexp = "expression/logic";
    private final static String statement = "statement/";
    private final static String failure = "failure/";
    private final static String tutorial = "tutorial/";

    public static void main(String args[])
    {
        try
        {
            currentDirName = currentDir.getCanonicalPath();
        }
```

```
        catch(Exception e)
        {
            e.printStackTrace();
        }

        testPath = currentDirName + "/test/";
    }

    @BeforeClass
    public static void oneTimeSetUp()
    {
        try
        {
            currentDirName = currentDir.getCanonicalPath
                ();
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }

        testPath = currentDirName + "/test/";
    }

    public static boolean parseFile(String filename)
    {
        boolean lexing_success = false;
        boolean parsing_success = false;

        try
        {
            // System.setErr(null);
            CharStream input = new ANTLRFileStream(
                filename);
            TanGLexer lexer = new TanGLexer(input);

            TokenStream ts = new CommonTokenStream(lexer)
                ;

            int errorsCount = lexer.
                getNumberOfSyntaxErrors();
            // ts.toString();
            if(errorsCount == 0)
            {
                lexing_success = true;
            }
```

```
            else
            {
                lexing_success = false;
                System.err.println("Number of lexer errors in " + filename + ": " + errorsCount + "\n");
                // return lexing_success;
            }


            TanGParser parse = new TanGParser(ts);
            parse.tanG();

            errorsCount = parse.getNumberOfSyntaxErrors();

            if(errorsCount == 0)
            {
                parsing_success = true;
            }
            else
            {
                parsing_success = false;
                System.err.println("Number of syntax errors in " + filename + ": " + errorsCount + "\n");
                // return parsing_success;
            }
        }
        catch(Exception t)
        {
            // System.out.println("Exception: "+t);
            // t.printStackTrace();
            parsing_success = false;
            return parsing_success;
        }

        return lexing_success && parsing_success;

    }

    public static File[] listTDFiles(File file)
    {
        File[] files = file.listFiles(new FilenameFilter() {
            @Override
```

```java
            public boolean accept(File dir, String name)
            {
                if(name.toLowerCase().endsWith(".td"))
                {
                    return true;
                }
                else
                {
                    return false;
                }
            }
        });

        return files;
    }

    public static void run_success(File file)
    {
        File[] files = listTDFiles(file);

        for(File f : files)
        {
            if(f != null)
            {
                //System.out.println(f.getAbsolutePath());
                assertTrue("Failed to parse " + f.getName(), parseFile(f.getAbsolutePath()) );
            }
        }
    }

    public static void run_failure(File file)
    {
        File[] files = listTDFiles(file);

        for(File f : files)
        {
            if(f != null)
            {
                //System.out.println(f.getAbsolutePath());
                assertFalse("Should not have parsed " + f.getName(), parseFile(f.getAbsolutePath()) );
            }
```

```java
        }
    }

    @Test
    public void test_whitespace()
    {
        // System.out.println(testPath + whitespace);
        File file = new File(testPath + whitespace);
        run_success(file);
    }

    @Test
    public void test_comments()
    {
        File file = new File(testPath + comments);
        run_success(file);
    }

    @Test
    public void test_expression()
    {
        File file = new File(testPath + expression);
        run_success(file);
    }

    @Test
    public void test_math_expression()
    {
        File file = new File(testPath + mathexp);
        run_success(file);
    }

    @Test
    public void test_bitwise_expression()
    {
        File file = new File(testPath + bitwiseexp);
        run_success(file);
    }

    @Test
    public void test_logic_expression()
    {
        File file = new File(testPath + bitwiseexp);
        run_success(file);
    }

```

```java
    @Test
    public void test_statement()
    {
        File file = new File(testPath + statement);
        run_success(file);
    }

    @Test
    public void test_failures()
    {
        File file = new File(testPath + failure);
        run_failure(file);
    }

    @Test
    public void test_tutorial()
    {
        File file = new File(testPath + tutorial);
        run_success(file);
    }
}
```

B Credits

- Abdullah Al-Syed (aza2105), for assisting with the white paper, tutorial, and language resource manual.
- Shuai Sun
- Professor Alfred Aho