FUNCTIONAL ABSTRACTION AND COMPILATION IN A DATA FACTORY

by

Allen Chen

A senior thesis submitted in partial fulfillment of
the requirements for the degree of

Bachelor of Science
with Departmental Honors

University of Washington
Department of Computer Science and Engineering
June 2007

Presentation of work given on: May 9th, 2007

Thesis and presentation approved by: _____

Steven Tanimoto, Project Advisor

Date: _____

CONTENTS

Abstract

Functional abstraction is a central concept in many modern programming languages. However, the syntactical nature of these languages may sometimes intimidate and discourage younger users. As an alternative, visual programming languages can provide a way to introduce programming concepts such as functional abstraction to novice programmers. For some, the visual nature of visual programming languages may be more approachable than standard programming languages. We demonstrate an experimental version of the Data Factory, a data flow visual programming language, that supports functional abstraction and the compilation of virtual devices for seamless re-use.

1. Introduction

Most modern programming languages, like Java and C, support abstraction through the use of functions, and for good reason. Not only does this concept reduce duplicated code, it also encourages users to approach and think about problems from a more general standpoint. Whereas an inexperienced programmer may solve a problem as a series of distinct tasks, one who understands the usefulness of functional abstraction may realize that the solution is in fact a repetition of the same task, but with different parameters. The latter thinking process is ideal, as it demonstrates a truer understanding of the task at hand. However, for most, acquiring such a skill proves to be a daunting task.

We introduce an experimental version of the Data Factory visual programming language that is targeted for use by children between the ages of ten to eighteen. This experimental version supports functional abstraction through the compilation of devices created by the user. This not only allows the user to re-use devices they've created to design increasingly complex factories, but also addresses an issue of scaling. Since devices are compiled to a byte-code representation, they behave and perform just like the devices natively included in the program. As a result, showing the data flow of a factory with deeply nested custom devices will not suffer the same types of slow-downs that are experienced when the devices are emulating an instantaneous calculation.

In this paper, we will discuss the usage and implementation details of this version of the Data Factory, as well as the results of some informal user testing.

2. Related Work

2.1 The Data Factory

The Data Factory is a visual programming language based on the "factory model." [5] Users create a program with a specific behavior by building a factory with various devices chained together with conveyers. In a real factory, parts are put onto the conveyer belts, pass through large machines, and come out the other side more "complete", in some sense. Likewise, in the Data Factory, data is put onto conveyers and passes through devices, which then perform a computation on the data and output the result.

One important feature of the Data Factory is that it supports explicit visual data flow. This means that users can watch the data move across the conveyers and observe every step of a factory's computation. This can help in an educational context because the results are less mysterious when the user can see what is happening every step of the way. Explicit visual data flow also helps with troubleshooting by allowing the user to quickly locate and fix problem areas in their factory. Had the data flow not been visible, any single element of the factory could be the culprit, degrading the troubleshooting process to trial-and-error.

Previous work on the Data Factory has been done to support advanced features such as functional abstraction, but did so without compilation [3].

2.2 Programming Environments for Children

Stagecast Creator is an environment that allows children to create games and simulations visually [4]. It is relatively open ended, allowing the user to create a wide variety of programs, but does not explicitly support the kind of data flow model as seen in the Data Factory.

ToonTalk is a visual programming environment that represents many programming constructs as animated objects and characters in the LEGO-style world [2]. However, unlike the Data Factory, where programs are created by connecting a series of devices, programs in ToonTalk are created by teaching "robots" a series of operations.

2.3 Visual Data Flow

LabVIEW is a visual programming language used in industry for data acquisition and instrument control, but also can be used a a teaching tool [1]. LabVIEW is a data flow language, but unlike the Data Factory, it does not allow the state of the data flow to be viewed at every step of execution.

3. An Experimental Version of the Data Factory

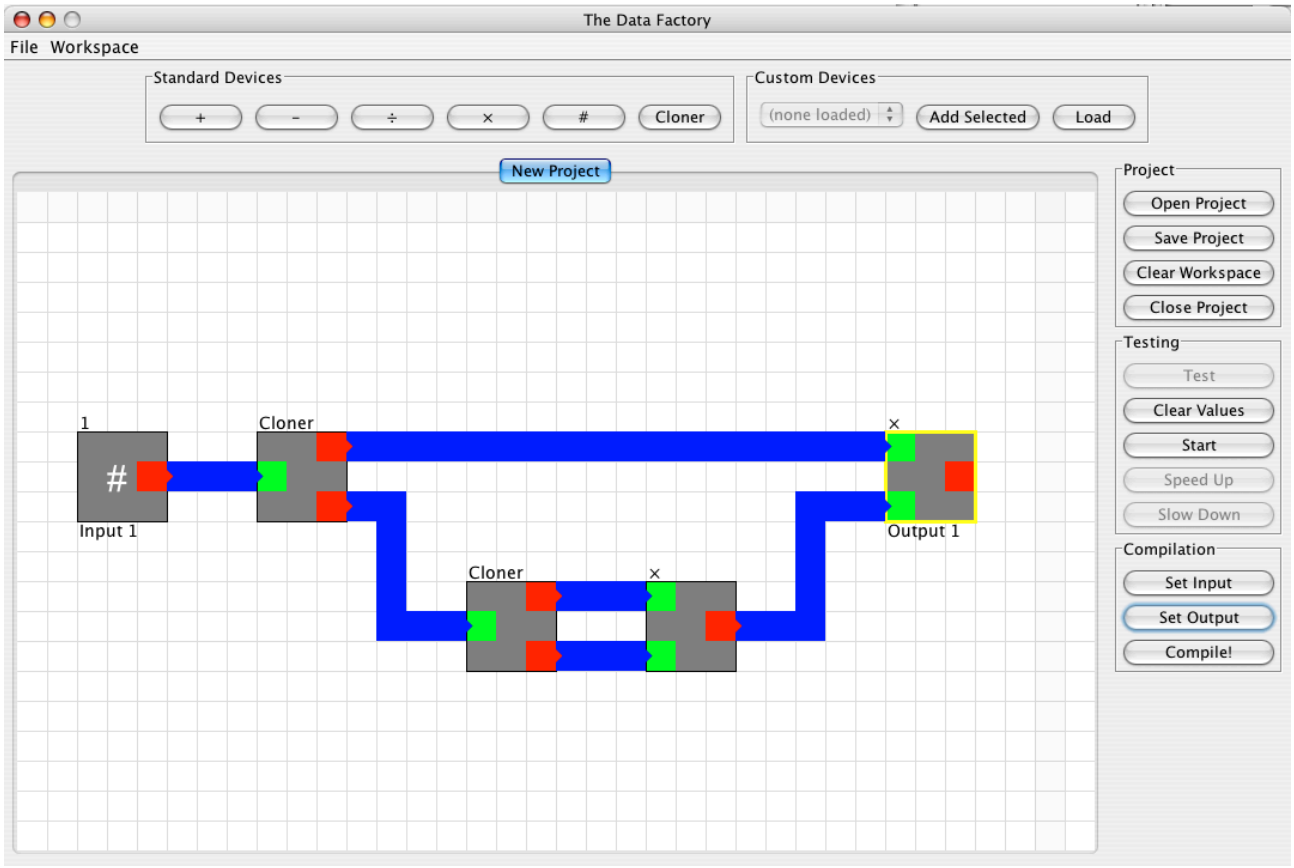


Figure 1: The experimental Data Factory interface. In the workspace (the gridded area) is a factory that takes a single input and outputs the value of the input cubed. Around the workspace are five button palettes that contain all of the program's operations and options.

3.1 Project Goals

The primary goal of this project was to implement a visual programming language that retained the data flow and factory model aspects of the Data Factory while implementing support for functional abstraction and compilation. More specifically, we define functional abstraction as the ability to take a user-created factory and turn it into an atomic device for re-use in another user-created factory. Compilation is what will transform the factory into a single, programming language native unit (in this case, a Java .class file).

Another goal of this project is to keep a simple feature set and target this program towards younger children. By allowing users to manipulate data with basic arithmetic operations, we hoped to create a version of the Data Factory that could be used as an educational tool to reinforce basic mathematical and problem solving skills.

3.2 Structure of the Experimental Data Factory

From this point on, unless otherwise noted, the Data Factory will refer to the experimental version addressed in this paper.

When a user first launches the Data Factory, he/she will see a screen much like the one in Figure 1, except with a blank workspace. The workspace is the gridded area where the user will place devices and connect them with conveyers. Located around the workspace are five button palettes that provide access to all of the program's functionality.

For the sake of clarity, we will refer to an arrangement of devices connected with conveyers as a factory. The area where the user lays out the factory is called the workspace. Devices added from the Standard Devices palette will be called native devices, whereas devices created by the user will be called custom devices. A project refers to the state of a factory, it's layout, and it's designated input and output devices. Projects can be saved and opened by the user to be worked on in multiple sessions.

3.3 Program Palettes

The Data Factory user interface has five button palettes, which are groupings of buttons with functions related to a single type of operation. These are also sometimes known as toolbars or panes.

3.3.1 The Project Palette

The Project Palette contains basic project-related operations. From here, a user can open, save, and close a project. The user can also clear the workspace of any devices and conveyers that have been placed.

3.3.2 The Standard Palette

The Standard Palette allows the user to select which device he/she would like to add to the workspace. In addition to the devices corresponding to the four basic arithmetic operations (addition, subtraction, multiplication, and division), there are two other devices the user can select. The # (pronounced "number") device serves two purposes. First, it can supply a constant value. For example, to multiply a value by two, it is necessary to use a # device to output twos into an input of a multiplication device. Also, # devices can be set as the location where outside input will flow into a factory. When a completed factory is later used as a single, compiled device, values sent into the device's inputs will logically enter into the factory at these # devices. The Cloner device does what you might expect; it takes in a single input and outputs two copies of that value. In a conventional programming context, this operation is not one that seems particularly useful. However, because devices consume

their input values, it is necessary to clone values in order to use them as the input for two different devices.

3.3.3 The Testing Palette

The Testing Palette allows the users to control the playback of the data flow. In additional to being able to start and stop the playback, the user can adjust the speed as well as clear all the data currently on the conveyers and in the devices. When the user hits the Test button, all # devices in the current working factory will output a single value onto their respective output conveyers.

3.3.4 The Compilation Palette

The Compilation Palette allows the user to prepare a factory for compilation. Before actually compiling the factory into a custom device, the user must specify which of the devices will act as the inputs and outputs of the custom device. Users can select # devices and then click Set Input to set that device as the input. All other types of devices can be selected and set as output device using the Set Output button. There can be a maximum of two inputs devices and two output devices.

3.3.5 The Custom Devices Palette

After compiling a factory, the user can add it as a custom device using the Custom Devices Palette. A successful compilation will result in a .device file to be created, which can be selected from the file chooser after clicking the Load button. After the custom device is loaded, it will appear in the dropdown menu. By selecting the desired device from the dropdown and clicking Add Selected, the user can add custom devices to the workspace.

3.4 Building a Factory

To add a device to the workspace, the user selects the desired device from the Standard Devices palette and clicks on the grid to place the device. To connect two devices, the user clicks on the output port of one device (in red) and clicks on input port of another (in green). If necessary, multiple conveyers links can be chained together by clicking on an empty space in the workspace. The link is set when the conveyer goes from transparent to opaque.

To delete a device, the user first selects a device by clicking on it. A yellow border will appear, signifying that the device has been selected. Hitting delete on the keyboard will delete the device, as well as any conveyers it is connected to.

The value output by # devices during testing can be changed by right-clicking on the # sign and selecting a number from the pop-up menu.

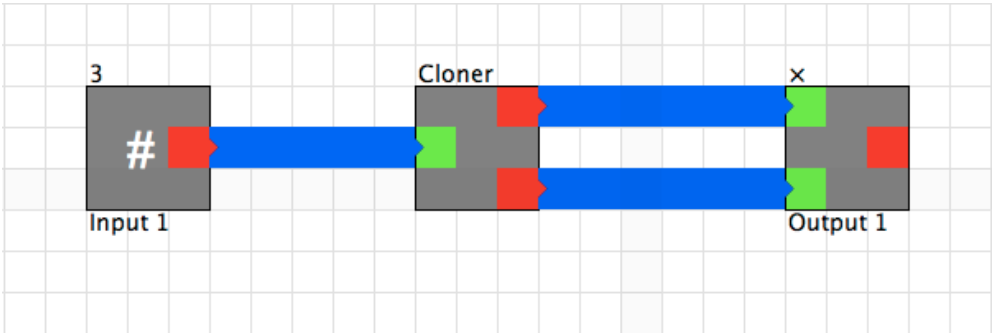


Figure 2: A factory for computing the square of a number

4. Sample Factories

The Data Factory can be used to create factories that perform a wide variety of operations, ranging from very simple to relatively complex. Here, we will discuss several factories that span this range of complexities and the steps necessary to create them.

4.1 Square

As shown in Figure 2, a user can create a factory that takes a single input and outputs its squared value using three devices. The input value will flow in from the # device on the left, enter the Cloner device, which will then send two copies of the input value into a multiplication device to obtain the final squared value.

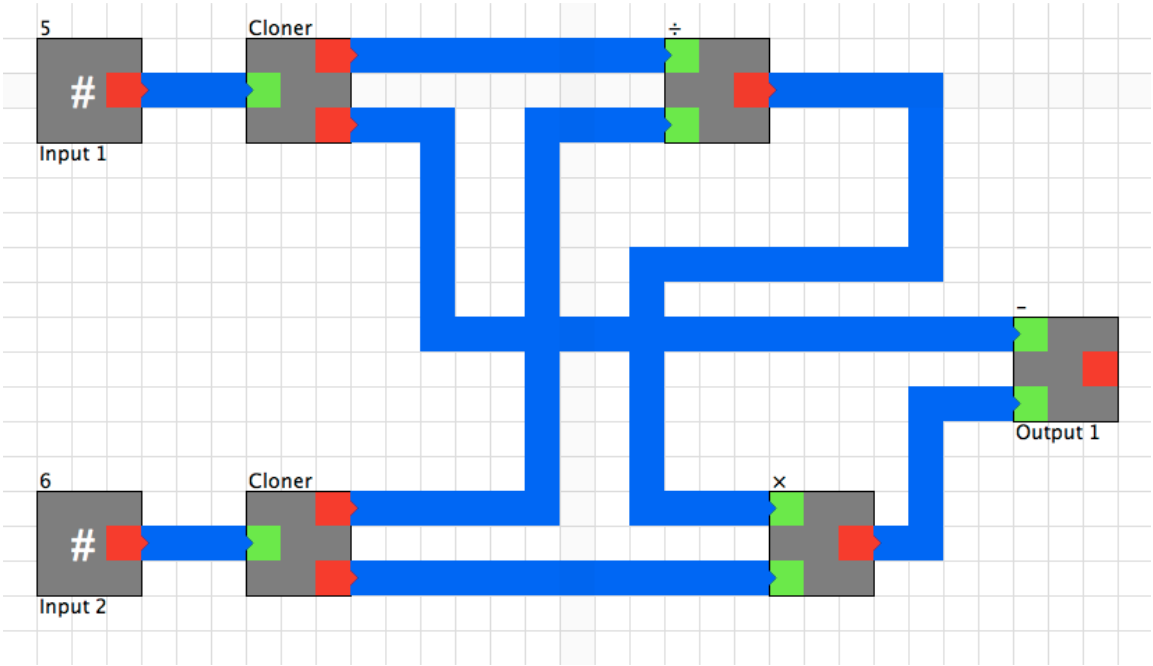


Figure 3: A factory for computing modulus

Notice that the # device has been designated Input 1 and the multiplication device Output 1. As a result, when this factory is compiled and inserted into a new factory as a custom device, it will have one input and one output. Those two ports will be logically "linked" to their respectively designated input and output device (in this case, the # device and the multiplication device).

The 3 above the # device designates the value that the device outputs when the Test button is clicked. For # devices that have been designated as an input device, this value is only output when testing. During actual usage as a custom device, the value that is input into the custom square device's input port will be that value that logically flows from this # device.

4.2 Modulus

To compute modulus, a factory with more devices will be necessary, as shown in Figure 3. Since modulus is an operation that requires two input values, we need to have two # devices that are appropriately designated as input devices. As with the square device, we

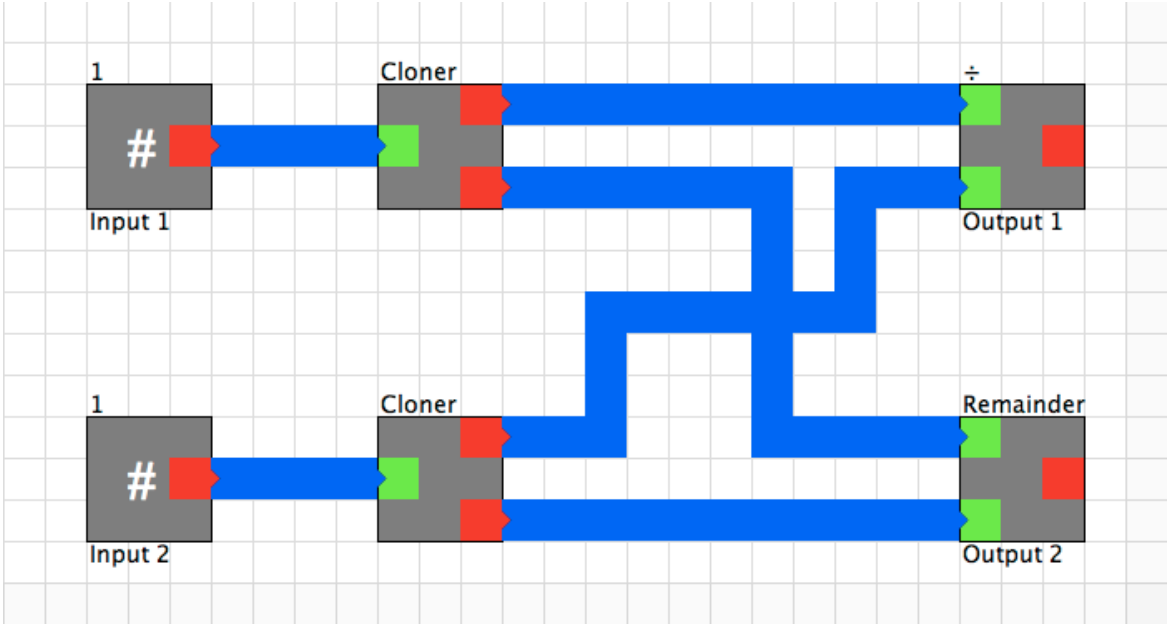


Figure 4: A factory that can be compiled into a device to implement the SuperDivide device interface

designate the device that performs the final computational step as the output port.

Because the length of the conveyers leading into the inputs of each device are of varying length, the input values will often arrive in a staggered manner. Every device in the Data Factory will wait for both of its inputs to be filled before consuming the values and outputting the result.

4.3 Super Divide

Suppose that a student has been given the task of completing a specific factory in the Data Factory. Such an assignment might read like this:

Create a SuperDivide factory that has two inputs and two outputs. The first of these outputs should be the first input divided by the second input, ignoring the remainder (integer division). The other output should be the remainder (modulus) of the division operation.

After familiarizing himself/herself to the Data Factory, the student will realize that the built-in divide device does exactly what is specified to be first output value. However, the Data Factory lacks any native support for computing modulus. The student will need to create their own device to compute modulus, so at this point, the student would begin a new project on a separate tab and eventually arrive at a factory that looks much like the one described in Section 4.2. The student could then save and compile their modulus/remainder factory and add it as a custom device in their original project to complete their SuperDivide device. Their factory will likely look something like the one depicted in Figure 4.

5. Implementation Details

5.1 Language Selection

This version of the Data Factory, like the previous ones, is implemented entirely in Java, specifically Java 1.5. Although this project came out of an entirely new code base, the decision to continue using Java was motivated by platform portability and a familiarity.

Platform portability is an important concern because it allows us to quickly deploy the software regardless of operating system as well as gives users a consistent look-and-feel. The decision to use Java for its familiarity was advantageous in improving development time, although it did present its share of challenges, which will be discussed later on.

5.2 Stages of the Data Factory

The processes that drive the creation of a factory, its compilation, and re-use as a custom device can be described in three stages. First is, of course, the actual construction of the factory in the project workspace. The second stage involves taking that factory and turning it into Java byte-code. Lastly, the Data Factory must read in that byte-code and turn it into a Java class that can be instantiated and used in a factory just like the other native Data Factory devices.

5.3 Device Creation

The workspaces on which factories are laid out are represented by 2D arrays of objects, with each array position corresponding to its respective visible grid square. Devices and conveyers are subclasses of Connectable, which requires that they each also have a 2D array of objects, representing their individual layouts. For example, each device has a three-by-three array, with input and output ports instantiated in the appropriate positions. When a device is placed onto the workspace, this three-by-three array is put onto a three-by-three area on the workspace's array at the desired location.

When one device's output is connected to another's input, the latter device keeps a

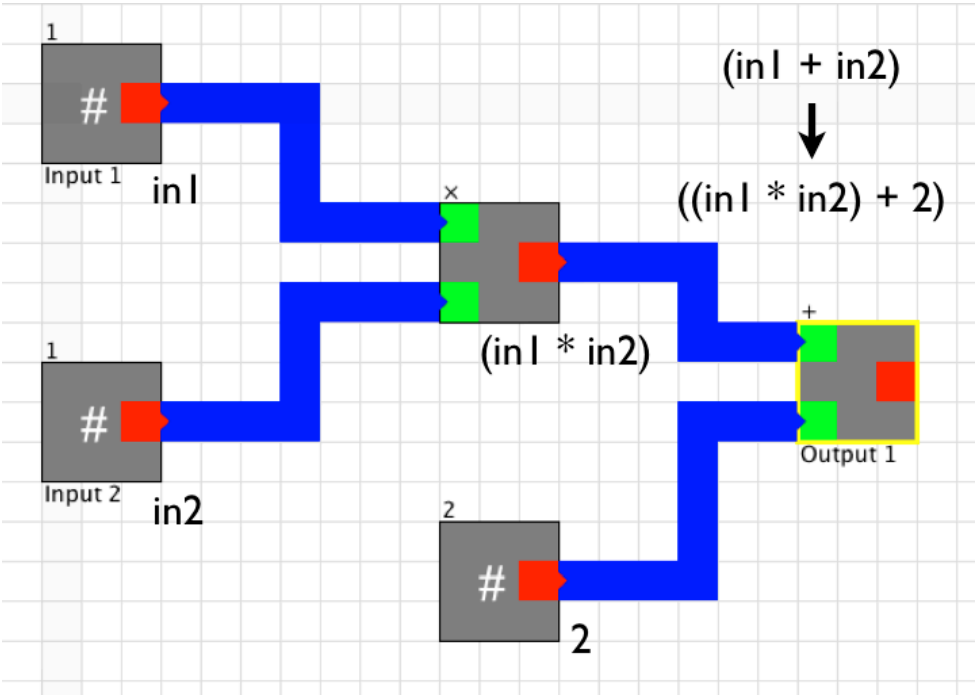


Figure 5: A tree-structured factory being traversed to produce an equivalent Java statement.

reference to the device that is supplying the value for that input port. Thus, in the process of building a factory, we have effectively constructed a binary tree of devices (devices can have at most two inputs), though this may not be visually evident. This fact will have important consequences when it comes time to generate Java code for the factory in preparation for compilation and will be discussed in greater detail in the next section.

5.4 Java Code Generation and Compilation

Though it may have been possible to generate Java byte-code directly from a completed factory, this was neither a practical nor efficient way to approach the problem. By first generating proper Java code, we greatly improve our ease of troubleshooting as well as our use of already available tools, in this case, the javac compiler. Thus, the factory compilation stage is in fact two steps: first, the Java code generation, and then the actual invocation of the Java compiler.

As mentioned in Section 5.3, completed factories will have a logical tree structure. This is the key to allowing us to produce the corresponding Java code. First, we will discuss code generation in the case where a factory is strictly tree-structured. Then, we will discuss how we approach generating code for factories that look more like directed acyclic graphs.

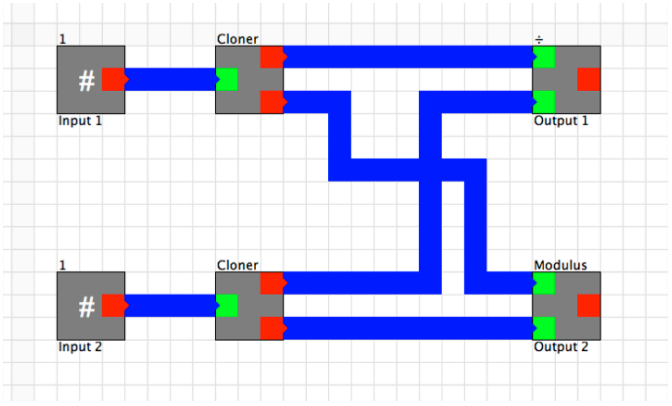
5.4.1 Code Generation of Tree-structured Factories

This version of the Data Factory does not currently support any types of loops or cycles. As a result, many basic factories will look much like trees tilted on their side. The consequence of this is that, for generating Java code, a factory can be treated as an abstract syntax tree. Since each device "knows" of its equivalent Java statement, traversing the tree from root (output) to leaves (inputs) can generate a new, single statement that is representative of the entire factory.

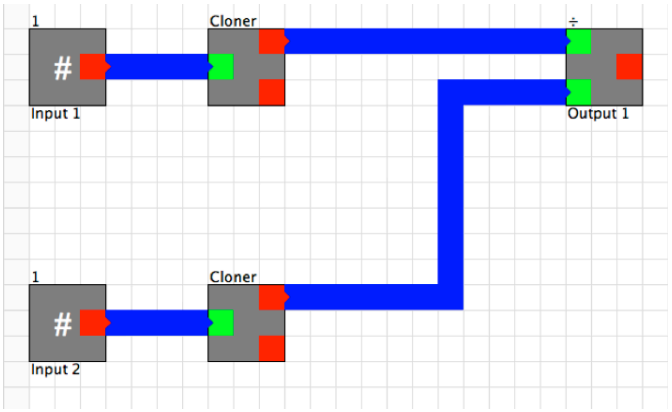
In Figure 5, we have a factory that takes in two inputs, multiplies them, and then adds two. Since the addition device has been designated the single output device, the traversal will begin there. Traversing the tree will result in a final string of ((in1 * in2) + 2), which is the statement that will be put into the generated Java code to represent the behavior of this factory.

5.4.2 Code Generation of Directed Acyclic Factories

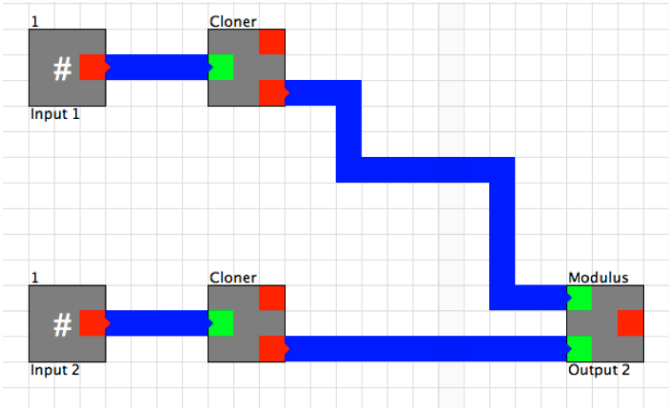
As mentioned in Section 5.4.1, often times, in order to create more complex factories, you cannot adhere to a strict tree structure. In these cases, the code generation process remains largely the same, though several design decisions had to be made for it to work correctly. Perhaps the most important of these decisions was in the design of the devices with two output ports (notice that these devices are what allows non-tree factories to be created). Though in the workspace, they appear as single devices, the input ports of two-output devices are actually linked to two (unseen) single-output devices that provide output values for the two-output device that they implement. What this allows is the output ports of a two-output device to be traversed independently through their respective internal devices, effectively restoring a tree structure.



(a)



(b)



(c)

Figure 6: The factory in (a) is traversed as two separate trees rooted at each output device, as illustrated in (b) and (c). This effectively restores the tree structure necessary for traversal. The traversals will produce two Java statements, which are then used to generate code for the two internal devices necessary to implement the factory in (a).

The design decision described above is also what allows factories with two designated output devices to be compiled. As shown in Figure 6, two traversals are done, one rooted at each of the outputs. This allows two single-output devices to be created, which will then be used to implement the actual two-output factory being compiled. Also seen in Figure 6 is how the output ports of the Cloner device (a two-output device) can be traversed separately to restore the tree structure.

5.5 Class Loading

After factories are compiled, they must be loaded back into the JVM as proper Java classes before instantiating them as a custom device. The class loading in the Data Factory is implemented by extending Java's ClassLoader with a custom class loader. The .class files are read in as a byte-stream and then loaded as a Java Class that can later be instantiated.

However, as a user begins adding custom devices to factories, class dependencies begin to build. This causes a problem when a user tries to open a project with a factory using custom devices. The project will not be able to load properly if the external classes for the custom devices have not been already loaded (using the Custom Devices palette). Due to the way persistence is implemented (using Serializable objects), there is no simple way to first execute an initialization routine to load the dependent classes before the rest of a project. One possible solution is to bundle saved projects as a jar file with all dependent classes copied into it. Only after those classes have been loaded will the project be opened.

6. Results of Informal Testing

Unfortunately, a user study with the targeted audience (late elementary to high school-aged children) was not able to be completed in time for this paper. However, informal testing was conducted with two college sophomore students. Each student participated in a single session lasting approximately twenty to twenty-five minutes. The first ten minutes of each session was spent introducing the Data Factory and going over how to use the program. Basic usage was demonstrated by building a factory for squaring a value, as described in Section 4.1. Once the subjects felt comfortable using the program, they were given the task to complete a factory that takes a single input and outputs the cubed value of that input.

6.1 Subject 1

The first subject was a student that had no previous computer science related coursework. It took this subject eight minutes to complete a properly working cube factory. While the subject immediately understood that the cubed value could be obtained by taking

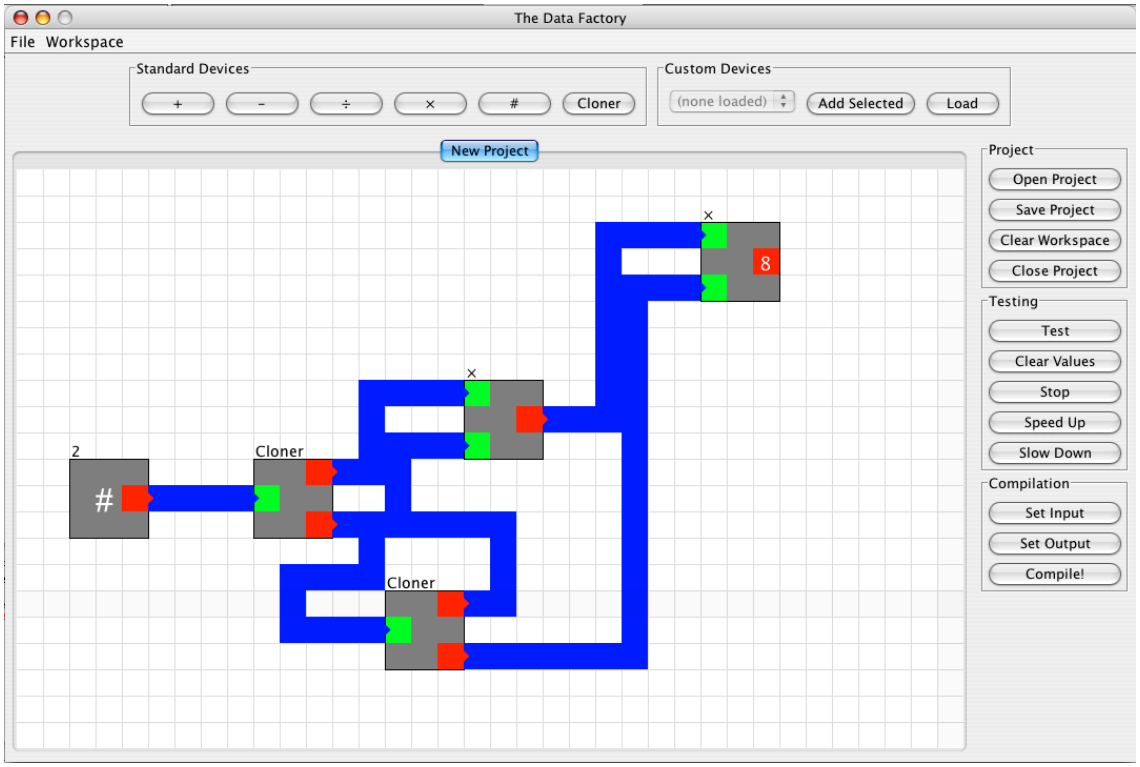


Figure 7: Subject 2's solution to the cube factory exercise.

the square and then multiplying again by the original input, the subject had a difficult time visualizing what the corresponding factory would look like. After getting stuck, the subject decided to go ahead and test the factory he/she had built, though incomplete. Once the values began to flow on the conveyers and through the devices, the steps necessary to complete a properly working factory immediately became evident to the subject.

6.2 Subject 2

The second subject was a student that had previously taken an introductory computer programming course in Java but was not interested in continuing in the series. It took this subject ten minutes to complete the cube factory. On this subject's first attempt, he/she built the square factory, cloned the squared value, and then sent the cloned values into a multiplication device. After testing, the subject realized that this factory raised the input to the fourth power rather than the third and was able to fix the problem to complete the factory. The subject's solution is shown in Figure 7.

6.3 Observations

Perhaps the most striking insight gained from these informal tests was the usefulness of the visual data flow. Neither of the subjects were able to easily visualize the correct

solution in their head and instead resorted to placing devices and conveyers in a manner that "just made sense." However, once the subjects saw the data flowing on the conveyers and where exactly the data was going (or not going), the problems with their factories became immediately evident to them. For many computer scientists who are accustomed to the type of thinking required by building factories, the visual data flow may not provide much more in addition to what they can already simulate in their heads. However, it appears that visualizing the flow of data can quickly accelerate the understanding and troubleshooting process for some subjects.

The informal testing also brought to light some usability deficiencies in the Data Factory, which will be discussed in greater depth in Section 7.2.

7. Future Improvements

7.1 Additional Functionality

While this version of the Data Factory added support for functional abstraction and compilation, some features were sacrificed in order to make those features possible. Perhaps most notably of these missing features is loops. Loops open up a variety of new possibilities for the types of factories that can be created. For example, in this version, there is no way to create a factory that takes two values and outputs the first to the power of the second without a native power device. With loops, creating such a device would be possible.

Another useful feature would be to have recursion. Currently, there is no way to signify a self-invocation in a factory. However, as in a previous non-compiling version of the Data Factory, this could be done with a "self" device that dynamically adjusts its number of inputs and outputs to match the current working factory [3]. Also, implicit in the support of recursion is the support of boolean operations, which would prevent infinite recursion.

Additional data types would also help increase the power of the Data Factory. Currently, only integer values are supported, though fractions and floating-point numbers would be welcome additions. These improvements, along with those discussed previously, would all greatly improve the Data Factory's usefulness as an educational tool by expanding the possibilities for the types of factories that can be created.

7.2 Layout and Usability

Since this version of the Data Factory was considered experimental, some usability conveniences were left out. For example, the moving of devices was not supported, nor was the deletion of an individual conveyer. However, after the informal user tests were conducted, it became clear how necessary these types of features were. One of the goals of visual data flow is to aid users by allowing them to see their factories operate, and then let

them quickly make any necessary changes. Unfortunately, the level of control provided in the program workspace did not make this process particularly simple or streamlined. Allowing the user a finer-grained level of control in manipulating devices and conveyers in the workspace would allow factories to be completed faster, as well reduce the time penalty for making mistakes.

Another usability issue that was brought to light in the course of user testing was the appearance of the conveyers. Since they do not have distinct edges, the conveyers that were side-by-side bled into each other and created a layout that was confusing to the eyes. This is particularly evident in Figure 7. Adding a border to the conveyers would help alleviate this issue.

8. Conclusion

We have described the design and implementation of a mechanism by which functional abstraction with compilation can be supported in a visual programming language. Though a proper user study of this version of the Data Factory could not be completed in time for this paper, it still presents an interesting approach in visualizing functional abstraction, a concept that is so fundamental to modern programming languages. In the future, such a user study would allow this visual programming language to be evaluated as a teaching tool and then appropriately modified to create a interesting and usable method of teaching programming language concepts visually.

9. Acknowledgments

The author would like to thank Professor Steven Tanimoto and Tyler Robison for their guidance and advice, as well as their previous work on the Data Factory.

10. References

[1] Egarievwe, S. U., Ajiboye, A. O., Biswas, G., Okobiah, O. K., Fowler, L. A., Thorne, S. K., and W. E. Collins. Internet Application of LabVIEW in Computer Based Learning. *European Journal of Open and Distance Learning 2000*.

[2] Kahn, K. A Computer Game to Teach Programming. *Proceedings of the National Educational Computing Conference 1999*, pp. 127-135. 1999.

[3] Robison, T. and S. Tanimoto. Transparent Procedural Abstraction. University of Washington, Seattle, WA. 2006.

[4] Smith, D. C. and A. Cypher. Making Programming Easier for Children. In Druin A., ed. *The Design of Children's Technology*, Morgan Kaufmann, San Francisco, 1998, pp. 201-221.

[5] Tanimoto, S. Programming in a Data Factory. *Proceedings of the 2003 IEEE Symposium on Human Centric Computing Languages and Environment*, Auckland, New Zealand, pp. 100-107. 2003.