

Deuce: Direct Manipulation Source Program Editor

Bachelor's Thesis
Advisor: Ravi Chugh
Winter 2017

Grace Lu
The University of Chicago
gracelu@uchicago.edu

Abstract

Commonly used text editors today often require users to manipulate code in a linear fashion. These editors, such as Sublime and vim, provide limited features for users to easily and quickly restructure code. Users therefore manually edit code by adding and deleting text or copying and pasting lines to make changes. The goal of Deuce as a direct manipulation text editor is to provide tools to help programmers improve the readability and maintainability of code with minimal effort, especially as code bases grow in size. Such tools rely on being able to interact with the program beyond simply rewriting or reorganizing text. Deuce introduces new methods of interacting with the text through keyboard and mouse, including selection of text and drag-and-drop of code elements. These direct manipulation text editing features are made possible by the editor keeping track of information beyond just the text that makes up the program. This requires an editor that is structure-aware. Deuce is not a standalone text editor but rather a layer on top of text editors to make certain classes of editing and refactoring tasks easier to perform and less prone to errors. This layer gives users the option to switch back and forth between traditional text editing and interactive program manipulation. Furthermore, this editor is unique in that it focuses on providing features for refactoring functional code, which lacks the rigid formatting of other programming paradigms.

1 Introduction

Sketch-n-Sketch (Resources #1, References [1] and [2]) is a direct manipulation programming system written in Elm that automatically generates code when interacting with the system. This allows a user to generate a program with minimal text editing. Figure 1 depicts the interface of this system. The left side is the text editor or code box; the right side is the output canvas. Users are able to draw shapes including lines, rectangles, ellipses, and polygons in the output canvas. After drawing the shape, Sketch-n-Sketch automatically generates the code that renders these shapes, as seen in the code box. Sketch-n-Sketch supports programs written in a simple lisp-like programming language called Little (#3). Users are able to write Little programs in the code box that will generate shapes when the program is run, or interact with the output canvas to have code be automatically written. The interface provides additional features in the output canvas such as editing the size and color of shapes without needing to manipulate variables through text editing in the code box. Users are able to invoke these features by clicking on selection zones and using sliders on the shapes. Higher-level features include being able to set variables of shapes equal to each other (Make Equal), such as making the x coordinate of the rectangle equal to the x coordinate of the line, or creating a relation between multiple shapes, such as one shape being half the width of another (Relate).

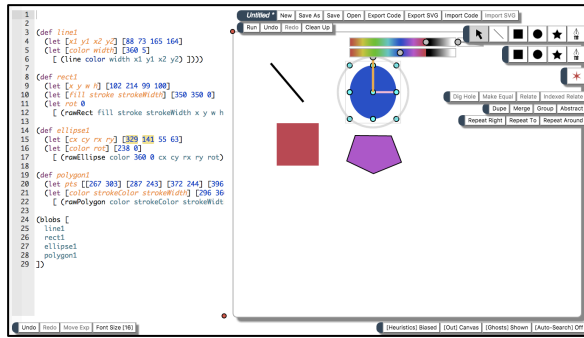


Figure 1. *Sketch-n-Sketch interface. The left side depicts the code box in which users can view and edit the text of Little programs. The right side depicts the output canvas in which users can draw and manipulate shapes, after which the system automatically generates or changes the code that renders the shapes on the left.*

The Sketch-n-Sketch text editor seen on the left side of the interface uses the Ace code editor (#4) written in JavaScript. Ace provides many built-in features such as line numbers and scrolling. This editor provides the foundation for the Deuce interactive layer.

Deuce is a layer built on the text editor portion of Sketch-n-Sketch as an extension of Ace. Similar to the direct manipulation program generator, Deuce aims to minimize time spent text editing to create a more fluid experience when interacting with the program text. This layer allows users to interact with the text through mouse features including selection and dragging in addition to traditional text editing. Therefore, Deuce is not considered a replacement for text editors but rather an extension. With these new interaction elements, users will be able to more easily edit and refactor complex code.

2 Related Work

2.1 DNDRefactoring

Researchers at the University of Illinois at Urbana-Champaign created a tool called Drag-and-Drop Refactoring (DNDRefactoring) in the Eclipse IDE (integrated development environment) to support the restructuring of code through mouse movements [4]. Their work applies specifically to the Java programming language and focuses on common code restructuring tasks seen in object-oriented programming. While the Eclipse editor itself does provide refactoring tools, users often experience difficulty finding, invoking, and configuring the features to work as intended. To overcome the proliferation

of menu buttons and configuration options, the DNDRefactoring editor provides the user with the ability to select drag sources and drop targets to easily interact with the text program. This drag-and-drop feature allows users to perform program transformations with a single movement as opposed to clicking through menu options and selecting configurations. Therefore, users are able to directly interact with variables, expressions, statements, methods, and other elements in a program [4]. DNDRefactoring supports two classes of refactorings—within the Java editor or between the Package Explorer and Outline View. Editor refactorings with drag-and-drop include promoting local variable to field, extracting temp variable, introducing parameter, extracting method, and moving methods or members; refactorings between and within the Package Explorer and Outline View include moving methods, moving types to files or packages, converting types, and extracting classes. To test the performance of the DNDRefactoring tool, the researchers conducted user studies. In these user studies, the researchers timed how long it took participants to perform refactorings with and without the DNDRefactoring tool. Results shows the participants were able to perform almost all common refactorings faster in the DNDRefactoring editor than in the regular Eclipse interface. In fact, refactorings were performed on average 3 times faster, showing drag-and-drop to be a more efficient tool for code restructuring [4].

2.2 Barista

Similarly, Ko and Myers of Carnegie Mellon University created a programming environment called Barista to overcome limitations of plain text editors. Barista provides data structures, algorithms, and interactive techniques to augment traditional text editors [3]. These new tools and interactions will improve usability and utility of code editors to increase programmer productivity. This framework allows users to move back and forth between structured and unstructured versions of the code. Barista uses a model-view-controller architecture in which the model is an abstract syntax tree composed of structures and tokens, the view is a tree of interactive views, and the controller is the event handlers. An abstract syntax tree representation of the code is created internally to be able to maintain the structure of a program and also provides a visual user interface to enable interactive editing features. With the abstract syntax tree, users of

Barista are able to match delimiters and quickly see errors in code such as improper scoping of variables since these are easily identifiable when the program structure is determined. For frontend features, the framework allows for rich annotation metadata to be displayed and provides focused views on specific blocks of code. In addition, the environment supports a drag-type feature to help with movement of code blocks. This feature is implemented with the keyboard, with users holding down a modifier key near statements or delimiters and using the arrow keys to select the target location.

3 Deuce

Deuce (#2) aims to accomplish many of the same goals as DNDRefactoring and Barista in creating a more interactive program editor that provides refactoring features beyond normal plain text editing. These additional features that augment traditional text manipulation aim to make refactoring easier and more reliable.

Similar to Barista, the Deuce project is split into two parts: the frontend user interface and the backend program transformations. The program transformations rely on the user interface to be able to trigger backend changes to the code. Section 4 describes the creation of the Deuce user interface and the features it provides; Section 5 details the program transformations that are made possible by the frontend.

The focus of the completed work has been the user interface discussed in Section 4. This interactive interface is necessary for the program transformations discussed in Section 5 to be implemented.

4 Deuce: User Interface

Since Deuce is an added layer built on top of Ace, the state of Deuce is kept track of in the backend Elm code in a model. This model maintains information about the code box and its contents. With this information, the code box can be displayed as intended.

Code items that Deuce keeps track of include expressions, patterns, and target positions. Target positions are spaces before and after expressions and patterns.

4.1 Item Identification

The first step in creating a direct manipulation text editor is indexing elements within the program. This means attaching identifiers to expressions, patterns, and target positions in the program. This information is needed to be able to select elements and also move them around within a program. Deuce therefore is a structure-aware editor. Plain text editors such as Ace do not keep track of the structure of the program.

Deuce keeps track of expressions, patterns, and target positions when parsing through code. IDs for these items are kept track of in the model. Furthermore, a dependence graph of variables in the code is created. This allows Deuce to keep track of dependencies between parts of the program to determine which transformations are valid and which are out of scope. This is similar to the abstract syntax tree structure Barista creates.

4.2 Item Selection: Initial Design

To enable the user to interact with the program beyond text manipulation, a selection feature was introduced into the Deuce editor. The first iteration involved the user being able to select an item such as an expression, pattern, or target position based on the mouse position of the click, and then the item would be highlighted using Ace editor highlights as seen in Figure 2. Initially, the Ace mouse position was used to identify click positions. Later on, a function was written to calculate the exact row and column position of a mouse click using pixel position for higher accuracy.

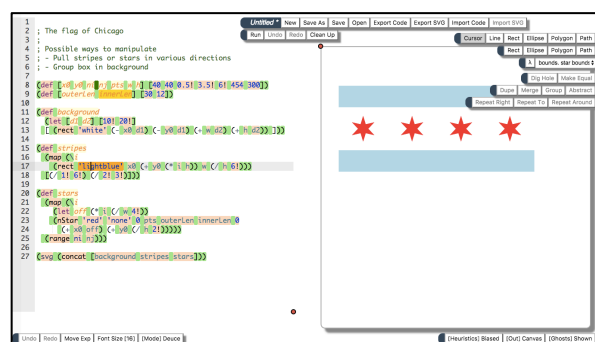


Figure 2. Ace highlights. The code box currently shows highlights of all items—expressions (orange), patterns (yellow), and target positions (green). Items that are selected appear in darker shades of the highlight color.

For the selection of larger expressions such as “def” and “let” expressions, which are composed of smaller expressions, the left parentheses were first used for selection. In the next iteration, the keywords themselves (i.e. “def” and “let”) were used for selection to provide the user with a larger area in which to select the expression. For a list of patterns, the left open bracket allowed for the selection of the list.

Target positions are spaces within expressions and patterns. In addition to allowing for selection of expressions and patterns, target positions are also selectable by clicking anywhere within the row and column position of the space. These are needed when moving pieces of code around the program.

Item selections occur when a user clicks down on the selection area of an item; deselection similarly occurs when the user clicks on an already selected item.

4.3 Bounding Box

As discussed in Section 4.2 about Item Selection, larger expressions such as “def” and “let” were only selectable by the keywords themselves. This greatly limits the selectable area for large, nested expressions. Therefore, instead of only allowing for selection of “def” and “let” for larger expressions and using Ace highlights to indicate selected items, SVGs (Scalable Vector Graphics) are drawn over items. SVG is a vector-based image format. This gives greater control over the look of hovered and selected items in Deuce and also allows for event handlers to be attached to these items. Instead of using the mouse position to determine which elements are being hovered over or selected, SVGs allows Deuce to simply attach functions that are executed when the SVG is hovered or clicked.

For the SVGs to be properly drawn over expressions that span multiple lines, the start and end columns are calculated for each line of the expression by using the unparsed text representation of lines. Using this information, the points of a polygon to be drawn over the expression can be calculated. This results in a SVG polygon that bounds the entire expression as seen in Figure 3.

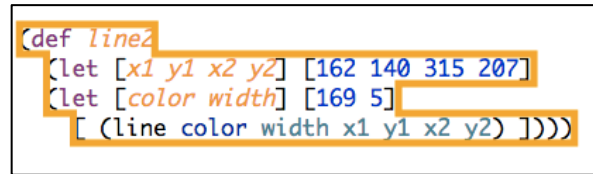


Figure 3. Bounding box for an expression.

4.4 Item Selection: Improved Design

The use of SVGs expanded beyond just “def” and “let” expressions to include all expressions, patterns, and target positions. The use of SVGs replaces the reliance on mouse position to determine selected elements. With SVGs being used for item selection, the use of mouse position for selection can be replaced by event handlers tied to the SVGs. Therefore, all expressions, patterns, and target positions are indicated with SVGs instead of Ace highlights. For expressions and patterns, the SVGs are polygons; for target positions, the SVGs are circles. These circles no longer occupy the entire row and column space position but rather are centered in the middle of the space.

In the Deuce layer, all polygons and circles that indicate expressions, patterns, and target positions are calculated but hidden until hovered or selected. All of these items are calculated recursively so that larger expressions are below smaller expressions, with patterns on top of expressions and target positions on the very top, on top of patterns. This allows the user to be able to “click through” to larger expressions. See the Figures below to better understand this feature.

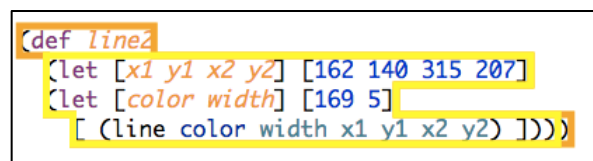


Figure 4. Overlapping expressions. As shown, this function definition includes smaller sub-expressions. Highlighted in orange is the larger expression. Highlighted in yellow is a nested smaller expression. Before the use of SVGs for selection, the orange expression was only selectable by clicking on the word “def” and the yellow expression was only selectable by clicking on the word “let”. Now, the orange expression can be selected by the left parenthesis, word “def”, and space around the target position seen after the word “def” (described in Figures 5-1 and 5-2). In addition, the orange expression can be selected by the right parenthesis, too. Similar-

ly, the yellow expression can be selected by its parentheses, the word “let”, and other spaces that are not occupied by other SVG elements.

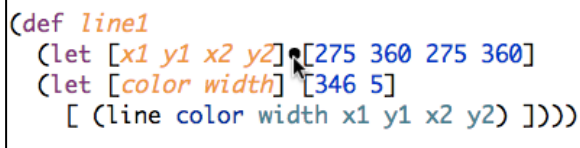


Figure 5-1. Mouse hovering target position circle.

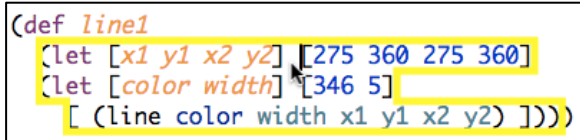


Figure 5-2. Mouse hovering the row-column space position of target position seen in Figure 5-1, but outside of the area of the SVG circle. When the mouse hovers the SVG circle of the target position, the user can select the target position. When the mouse moves out of the target position but stays in the row-column position of the space of the target position, the larger expression shown in yellow can be selected if clicked.

Before the use of SVGs over expressions, patterns, and target positions, hovering and selection of items was handled by determining which item the mouse position corresponded to in terms of row-column position. With SVGs, event handlers can be attached to indicate which items are currently being hovered over (onMouseOver) or clicked on (onMouseClicked).

Since the SVGs are drawn in the Deuce layer, which overlays the Ace layer, this affects basic Ace features such as scrolling. The position of polygons and circles drawn in the Deuce layer therefore need to be manually recalculated when a scroll occurs to correctly position the SVGs over selected elements in the code box.

4.5 Mouse Drag-and-Drop

With identification and selection of items in place, drag-and-drop of elements is possible. Selection allows the user to choose what to be moved where, and identification allows the program to know which items are selected. A drag in the Deuce layer involves a start position initiated with the click of the mouse and holding this click while moving to an end position where the click is released. Throughout this drag motion,

the start and end positions of the mouse are stored in the model so that the items associated with these positions can be identified.

Drag-and-drop is a feature that allows for program transformations to be performed in a simple and smooth motion. This removes the need for multiple selections or clicking through options.

4.6 Deuce Mode Features

Given Deuce is an added layer on top of the Ace text editor, features were added to be able to easily switch between Deuce and Ace mode. When the shift key is pressed, the user is in Deuce mode and is able to see selectable items when hovering over the item; the user is also able to select an item. When the shift key is not pressed, the code box functions as a normal Ace editor, though the selected items will still appear in the program (i.e. an expression selected with the shift key pressed will still appear selected when the shift key is not pressed). Deselection of items can only occur in Deuce mode. In both Deuce and Ace modes, though, the user can press the escape key to remove all selected items. This will reset the Deuce state to not contain any selected elements.

5 Deuce: Program Transformations

With the Deuce user interface in place, program transformations can take advantage of these interaction features to accomplish general tasks. Deuce currently supports drag-and-drop definition reordering described in Section 5.1. The other program transformations discussed in this section are currently being implemented. The program transformations discussed in this section are based on general programming tasks often done in functional languages.

While the user interface is the main focus of the work described in this paper, the following program transformations will be supported by the developed interface features. The figures in this section show how the implemented user interface will allow a user to perform specific refactorings.

5.1 Definition Reordering

Currently, Deuce supports movement of variables through drag-and-drop. Specifically, a user can click down on a pattern and drag it to a target position before an expression or to a target

position before or after a pattern. This allows the user to reorder a definition and also have the associated value be moved through a single drag-and-drop motion as opposed to moving the pattern name and also the value of the pattern in separate edits of the code.

```
(def line2
  (let [x1 y1 x2 y2] [253 146 350 201]
    (let [color width] [117 5]
      [ (line color width x1 y1 x2 y2) ])))
```

Figure 6-1. Selected pattern (orange) and target (black circle) position for drag-and-drop.

```
(def line2
  (let [x1 color y1 x2 y2] [253 117 146 350 20]
    (let [width] [5]
      [ (line color width x1 y1 x2 y2) ])))
```

Figure 6-2. Program transformation after clicking on “color” seen in Figure 6-1 and keeping the mouse down while moving to the target position indicated by the black dot in Figure 6-1 before releasing the mouse. This moved “color” to the target position and also moved the value of color (117) to the associated location.

5.2 Combining and Splitting Definitions

This program transformation allows users to create or remove variables. Combining definitions allows a user to clean up code by creating a reusable definition. For example, Figure 7 shows two “x1” variables defined in separate places. If the user wants to create a single “x1” variable to be used in both places with the same value, one of the “x1” variables could be dragged on top of the other to create the output seen in Figure 8. The value of global “x1” is then the value of the “x1” that was not dragged. Combining definitions can therefore create more readable and reusable code.

```
(def line1
  (let [x1 y1 x2 y2] [69 173 211 265]
    (let [color width] [118 5]
      [ (line color width x1 y1 x2 y2) ])))

(def line2
  (let [x1 y1 x2 y2] [175 144 338 232]
    (let [color width] [305 5]
      [ (line color width x1 y1 x2 y2) ])))
```

Figure 7. Two variables to be combined.

```
(def x1 69)

(def line1
  (let [y1 x2 y2] [173 211 265]
    (let [color width] [118 5]
      [ (line color width x1 y1 x2 y2) ])))

(def line2
  (let [y1 x2 y2] [144 338 232]
    (let [color width] [305 5]
      [ (line color width x1 y1 x2 y2) ])))
```

Figure 8. After dragging the second “x1” onto the first “x1”, the two “x1” variables have been combined into one definition at the top of the program that is used in both places.

Users may also be interested in the reverse action—splitting definitions when values are no longer shared. For example, if the two places “x1” is used do not share the same value anymore, the user could drag the “x1” to the places it would be uniquely defined as shown in Figure 9. This allows users to easily and quickly redefine variable values.

```
(def x1 69)

(def line1
  (let [y1 x2 y2] [173 211 265]
    (let [color width] [118 5]
      [ (line color width x1 y1 x2 y2) ])))

(def line2
  (let [y1 x2 y2] [144 338 232]
    (let [color width] [305 5]
      [ (line color width x1 y1 x2 y2) ])))
```

Figure 9. Variable to be split. If the value of “x1” is no longer shared, the user should drag “x1” to each place it will be uniquely defined. The result of this transformation is shown in Figure 10.

```
(def line1
  (let [x1 y1 x2 y2] [69 173 211 265]
    (let [color width] [118 5]
      [ (line color width x1 y1 x2 y2) ])))

(def line2
  (let [x1 y1 x2 y2] [69 144 338 232]
    (let [color width] [305 5]
      [ (line color width x1 y1 x2 y2) ])))
```

Figure 10. Splitting variables. The split “x1” values have the same value but now have a local scope. Therefore, the user can change the value of one without affecting the other.

5.3 Introducing Variables

Users may want to create new variables within a program to increase readability of code. For example, as seen in Figure 11, one variable is an expression defined in terms of another variable. The code may be made cleaner by extracting this variable out and then using the new variable name in its place as seen in Figure 12.

```
(def line1
  (let [x1 y1 x2] [136 104 227]
    (let [color width] [40 5]
      [ (line color width x1 y1 x2 (- y1 10)) ])))
```

Figure 11. Instance to introduce a new variable.

```
(def line1
  (let [x1 y1 x2] [136 104 227]
    (let num (- y1 10)
      (let [color width] [40 5]
        [ (line color width x1 y1 x2 num) ]))))
```

Figure 12. Introducing a new variable.

5.4 Introducing Lambdas

When similar pieces of code are frequently reused within a program, a lambda function can be introduced to reduce the program size. As shown in Figure 13, the definitions for “line1”, “line2”, and “line3” use the same code with different variable values. Therefore, a function that parameterizes the changing values can be created to simplify the code (Figure 14).

To perform this program transformation, the user drags the similar definitions on top of one another to combine them. In the example shown in Figure 13, the user can drag the “line2” definition on top of the “line1” definition, which would create the “line1” function seen in Figure 14. Then the user would drag the “line3” definition in Figure 13 on top of the newly created “line1” function to create the program seen in Figure 14.

```
(def line1
  (let [x1 y1 x2 y2] [100 117 150 233]
    (let [color width] [0 5]
      [ (line color width x1 y1 x2 y2) ])))

(def line2
  (let [x1 y1 x2 y2] [182 58 242 218]
    (let [color width] [468 5]
      [ (line color width x1 y1 x2 y2) ])))

(def line3
  (let [x1 y1 x2 y2] [255 79 343 267]
    (let [color width] [242 5]
      [ (line color width x1 y1 x2 y2) ])))

(blobs [
  line1
  line2
  line3
])
```

Figure 13. Repetitive code snippets.

```
(def line1 (\(x1 y1 x2 y2 color)
  (let [color width] [color 5]
    [ (line color width x1 y1 x2 y2) ])))

(blobs [
  (line1 100 117 150 233 0)
  (line1 182 58 242 218 468)
  (line1 255 79 343 267 242)
])
```

Figure 14. Introducing a lambda function to remove repetitive code snippets.

5.5 Adding or Removing Function Arguments

Building off of the previous program transformation (Introducing Lambdas), there may be certain parameters to the created function or any other defined function that should be added and others that should be removed. For example, in Figures 14 and 15, the “width” variable is set to be 5 for all calls to the “line1” function. The user may instead want “width” to be a parameter to the “line1” function. To invoke this transformation, the user would drag the 5 inside of the “line1” function definition into the parameter list for the function as depicted in Figure 15. This would then change the function to have a “width” parameter and also add an argument value of 5 to each place the “line1” function is called, as seen in Figure 16.

```

(def line1 (\(x1 y1 x2 y2 color)
  (let [color width] [color 5]
    [ (line color width x1 y1 x2 y2) ])))

(blobs [
  (line1 100 117 150 233 0)
  (line1 182 58 242 218 468)
  (line1 255 79 343 267 242)
])

```

Figure 15. Adding a function argument to a function.

```

(def line1 (\(x1 y1 x2 y2 color width)
  (let [color width] [color width]
    [ (line color width x1 y1 x2 y2) ])))

(blobs [
  (line1 100 117 150 233 0 5)
  (line1 182 58 242 218 468 5)
  (line1 255 79 343 267 242 5)
])

```

Figure 16. Adding a function argument also edits the function at call-sites.

Similarly, function arguments can also be removed. To invoke this transformation, the user should drag the argument to be removed directly on top of the parameter, as seen in Figure 17. The result will then be Figure 14. Note the “width” parameters are also removed at the call-sites.

While all “width” values in Figure 17 have a value of 5, the value that is dragged on top of the parameter variable will be the value that is set in the function. For example, if the second call to “line1” in Figure 17 had a “width” value of 6 while the other “width” values remained the same with values of 5, and this 6 value was dragged on top of “width”, the “width” value in Figure 14 would be 6.

```

(def line1 (\(x1 y1 x2 y2 color width)
  (let [color width] [color width]
    [ (line color width x1 y1 x2 y2) ])))

(blobs [
  (line1 100 117 150 233 0 5)
  (line1 182 58 242 218 468 5)
  (line1 255 79 343 267 242 5)
])

```

Figure 17. Removing a function argument.

5.6 Reordering Function Arguments

To reorder function arguments, users should drag the argument to be moved onto a target position. In the following example in Figure 18, the user wants to move “x2” before “y1” but after “x1”. This results in Figure 19, where the “x2” parameter is after the “x1” parameter, and the corresponding “x2” values in the calls to the “line1” function have also moved to the corresponding places.

```

(def line1 (\(x1 y1 x2 y2 color)
  (let [color width] [color 5]
    [ (line color width x1 y1 x2 y2) ])))

(blobs [
  (line1 68 116 165 237 456)
  (line1 182 76 286 213 32)
])

```

Figure 18. Reordering a function argument.

```

(def line1 (\(x1 x2 y1 y2 color)
  (let [color width] [color 5]
    [ (line color width x1 y1 x2 y2) ])))

(blobs [
  (line1 68 165 116 237 456)
  (line1 182 286 76 213 32)
])

```

Figure 19. Reordered function argument also changes argument ordering at call-sites.

5.7 Button Transformations

In addition to drag-and-drop program transformations, Deuce can support transformations induced by selection of items and button clicks. For example, if the user wants to make two variables equal, one option is to use the drag-and-drop method described in Section 5.2. Another option would be to select the two variables to be set equal and click a “Make Equal” button that becomes available when the selected items satisfy the requirements of using the button. This selection option to perform program transformations can be used in addition or in place of drag-and-drop options. The risk of relying on selection and buttons is the proliferation of menus in the user interface. The addition of menus, though, could provide the user with more options and configurations to produce the exact refactoring outcome intended. Nevertheless, drag-and-drop allows the user to easily perform program transformations with a single motion.

6 Future Work

The user interface features implemented for the Deuce layer lay the foundation for a wide variety of refactorings. These refactorings include, but are not limited to, the program transformations discussed in Section 5 (Deuce: Program Transformations).

On the frontend, the user interface can continually be iterated on. As discussed in Section 4 (Deuce: User Interface), multiple design iterations were implemented and improved on. Additional features that can be added to the user interface include elements to better guide a user through Deuce. This may include arrows to indicate where an item may be moved after it has been selected, a before and after comparison of the code once a transformation has been performed, and a short pause before the actual code change is made so that it is not so sudden. These would allow a new user to better understand the features Deuce provides and also give all users more ideas about how to refactor the program.

On the backend, much of the future work is described in Section 5. These transformations rely on both the user interface and structure-aware components of Deuce. With more program transformations in place, users are able to see the full potential of the direct manipulation source program editor to make editing and refactoring easier.

With more features on both the frontend and backend in place, user studies can be conducted to test the performance of Deuce. Similar to the user studies done by the DNDRefactoring team, users will be asked to perform various program transformations in Deuce and in the regular Ace editor. The tests will measure the time needed in both cases to complete a task and also the users' opinions. These studies will be useful in improving Deuce's user interface and also program transformation features. This feedback will drive future work.

7 Conclusion

Deuce aims to make editing and refactoring of programs easier and less prone to errors. This is done by creating a layer on top of a traditional text editor that provides features that make such tasks easier to invoke and execute. To be able to provide these features, the editor must be struc-

ture-aware, keeping track of elements of the program and their relationships. With an interactive user interface and useful program transformations, Deuce can improve the programming experience by making common tasks simpler and quicker to perform.

8 Acknowledgements

I would like to thank Professor Ravi Chugh, my thesis advisor, for giving me this opportunity to work on the Sketch-n-Sketch project and for providing continuous guidance throughout the entire endeavor. I would also like to thank Professor Shan Lu for her advice and mentorship. I am also grateful for the Liew Family College Research Fellows Research Grant for supporting this project.

Resources

#1 Sketch-n-Sketch project:

<https://ravichugh.github.io/sketch-n-sketch/index.html>

#2 Deuce code (last commit by author):

<https://github.com/ravichugh/sketch-n-sketch/commit/c794383cfe3e60c8490c739a4e6533af7f2fe511>

#3 Little language syntax guide:

<https://github.com/ravichugh/sketch-n-sketch/blob/master/README.md>

#4 Ace Editor:

<https://ace.c9.io/#nav=about>

References

- [1] R. Chugh, B. Hempel, M. Spradlin, and J. Albers, "Programmatic and Direct Manipulation, Together at Last," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2016.
- [2] B. Hempel and R. Chugh, "Semi-Automated SVG Programming via Direct Manipulation," in *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST)*, 2016.
- [3] A.J. Ko and B.A. Myers, "Barista: An Implementation Framework for Enabling New Tools, Interaction Techniques and Views in Code Editors", in *Proc. ACM Conference on Human Factors in Computing Systems*, 2006.
- [4] Y.Y. Lee, N. Chen, R.E. Johnson, "Drag-and-Drop Refactoring: Intuitive and Efficient Program Transformation," in *ICSE*, 2013.