This is work that I’ve done with my advisor, Brad Myers.
Programmers are users too, and programming environments are the primary user interfaces for the millions of professional and end-user programmers' in the world.

Most of a programmer’s time is spent in source code editors. ☛
Let's improve code editors usability, user experience, and utility!

Therefore, it is very important to improve code editors usability, user experience, and utility. But not only that, code editors should be more fun to use!

Now, textual code editors are a little bit boring. ☹
Textual Editors

Syntax highlighting

Status markers

Immediate feedback about errors

The Java editor in Eclipse

For example, here is the Eclipse Java editor. Now aside from being boring, it actually providing many powerful search, analysis, and navigation tools, and several types of useful visual feedback, including syntax coloring, status markers in the column, and even immediate feedback about syntax errors.

While these types of visual tools and feedback are very helpful, one problem with textual editors is that editor designers are very restricted in the types of tools they can provide directly in the code editor, because code is visually represented as rows of characters.
Tools and feedback must either fit within this layout, or be placed out of context, in a separate window.
Monolithic architecture of editor makes customization hard

Furthermore, textual editors like these can be difficult to customize, because they are often designed monolithically. For example, adding features to the Eclipse editor is very difficult, because all of the existing tools depend greatly on the layout of the code.

There are some kinds of editors, called structured editors, which avoid these limitations...
For example, this is the Alice development environment, which offers embedded tools in a number of unique ways. For example, the drop down menu shown here is used to control the value of a boolean expression.

These tools are possible because Alice visually represents code as trees of interactive views, like most traditional graphical user interfaces are presented.
This representation allows for more flexibility in the placement of tools, and the types of visual feedback that can be provided.

Unfortunately, this has traditionally required users...
Interaction techniques can be inflexible and indirect

...to edit code exclusively through drag and drop techniques or through menus, which are much less flexible ways of editing code for users. Modifications that might take a few edits with text, can take dozens of drags and drops.

So: the problem is that ... ☛
## The Problem

<table>
<thead>
<tr>
<th>Flexible interaction techniques for <strong>users</strong></th>
<th>Flexible visual medium for <strong>editor designers</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Text Editors</td>
<td>![Checkmark]</td>
</tr>
<tr>
<td>![Checkmark]</td>
<td>Structured Editors</td>
</tr>
<tr>
<td>![X]</td>
<td>✔</td>
</tr>
</tbody>
</table>

...textual editors provide a flexible interaction technique, but limit editor designers’ ability to embed visual tools and interactions in the code,

...whereas structured editors provide much more flexibility in this regard, but have traditionally offered inflexible ways of editing code.

Our goal in designing Barista was to try to provide both flexible interaction techniques and a flexible visual medium for editor designers to design new tools.
The Problem

<table>
<thead>
<tr>
<th>Flexible interaction techniques for users</th>
<th>Flexible visual medium for editor designers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Text Editors</td>
<td>☑</td>
</tr>
<tr>
<td>Structured Editors</td>
<td>☑</td>
</tr>
<tr>
<td>Barista Editors</td>
<td>☑</td>
</tr>
</tbody>
</table>

...textual editors provide a flexible interaction technique, but limit editor designers’ ability to embed visual tools and interactions in the code, whereas structured editors provide much more flexibility in this regard, but have traditionally offered inflexible ways of editing code.

Our goal in designing Barista was to try to provide both flexible interaction techniques and a flexible visual medium for editor designers to design new tools.
Barista is a new toolkit for creating code editors. It's central contribution is that it provides editor designers flexibility in designing embedded tools and interactions, while providing editor users with several ways of editing code, including text editing, drag and drop editing, and menu-based editing.

The toolkit uses a model-view-controller architecture, which many user interface developers are familiar with from popular user interface toolkits.

The toolkit is implemented in the Citrus programming language, which I presented last fall at UIST.

There are lots of cool things that Barista enables, including ☛
Visual meta data, embedded tools, alternative views of expressions, and pretty printing.

In this talk...
I’m going to cover four topics:

- Three ways to edit code in a Barista editor
- How to implement an editor using the Barista toolkit
- How Barista implements its standard text editing interactions
- And several examples of embedded tools, made possible by Barista.

So let’s start with a look at a Barista editor.
This is a Java Editor that we built using the Barista toolkit, showing a simple Hello World program.

In this editor, we’ve provided three ways for users to edit code...
This is a Java Editor that we built using the Barista toolkit, showing a simple Hello World program.

In this editor, we've provided three ways for users to edit code...
Users can create code by typing it as text. Here you see a user typing out a class named Blah from scratch, and the editor recognizes structures as they are typed.
Users can create code by typing it as text. Here you see a user typing out a class named Blah from scratch, and the editor recognizes structures as they are typed.
Users can edit code by using dragging structures of various types onto the editor. Here you see a user dragging an import statement, a class, a method declaration, and a local variable declaration.
Users can edit code by using dragging structures of various types onto the editor. Here you see a user dragging an import statement, a class, a method declaration, and a local variable declaration.
Users can also edit code through auto-complete menus, which show the legal structures and tokens that can be inserted at various points in the program, like the traditional structured editors that were developed in the 80's.

Now...
Users can also edit code through auto-complete menus, which show the legal structures and tokens that can be inserted at various points in the program, like the traditional structured editors that were developed in the 80's.

Now...
... let’s move our discussion to how an editor designer uses Barista to implemented an editor ☛
Our goals in designing the Barista toolkit were to enable the creation of an editor for any language. To date, we've prototyped editors for Java and Citrus, but others should be easy to create.

To do this, the toolkit provides fundamental data structures and algorithms, that are reusable in any editor.

Furthermore, we wanted to define a consistent API for customizing and extending Barista editors. This is in contrast to conventional plain text editors, which are very monolithic and difficult to customize.
The designer's first step is to specify the target language's structures. These are the basic models in the editor, which represent the various structures in a program. To do this, two things are required.

The first thing is the syntax grammar for the structure. For example, this is the syntax grammar for a Java if-statement on top. It has several tokens, like the if keyword, the parentheses, and the else keyword, and several structures, including the condition and the two statements. Designers must include this to enable textual editing of code.

The second thing is a class definition to represent the structure in the abstract syntax tree. This is Citrus code, which declares a class named IfStatement that extends Structure, and declares fields for each of the tokens and structures of the if statement.

Now there are many situations where the designer can supply a grammar and the class can be generated automatically. However, this isn't always desirable, because parsable grammars don’t always represent the desired structure of the abstract syntax tree. There’s more detail on this in the paper.

In addition to defining the structures in the language, the editor designer...
an **Identifier** is a **Token**

for which *(text matches "[a-zA-Z][a-zA-Z0-9_]*")*

for which *(whitespace matches "\s+")*

**Editor designer must specify regular expressions the text and whitespace of each type of token in the target language.**

For example, here’s how one might specify the Java Identifier token. This says that the token text must match this regular expression, and the whitespace must be one or more whitespace characters.

Once the structures and tokens are defined, the editor designer then specifies structure and token views.
...to represent these structures and tokens in a program. The view outlined in green here is the structure view representing the if statement.

StructureViews are made up of TokenViews and other StructureViews. For example, this if statement view has three StructureViews in it.
One for the condition, and two for the then and else statements.

The if statement also has four TokenViews, including ☛.
the if and else keywords and the two parentheses.
A **TokenView** is a text field, so it may have custom fonts, sizes, decorations, interaction techniques, etc.

```lisp
(defn IdentifierView [model]
  (TokenView model
    (has Identifier model)
    (has FontFace font
      (cond
        (model.type is DECLARATION) style.declarationFont
        (model.type is TYPE) style.typeFont
        (true) style.identifierFont)
    ))
```

All of these are implemented using the Citrus UI toolkit, meaning it can take full advantage of Citrus language features, such as constraints.

Now every TokenView is a little text field, so it may have custom fonts, sizes, decorations, interaction techniques, and so on.

For example, here is code for our JavaIdentifier view, which has a constraint on the font so that the font is determined by the type of the identifier. If its a declaration, it has one font, if its a type name, it has another, and then there’s a default font for other references.

Similarly,
A StructureView’s width and height can be controlled by constraints, and

```
an IfView is a BaselinedView that

has If model = ?

has Real width <- (this rightmostChildsRight)
has Real height <- (this bottommostChildsBottom)
```

This IfView uses Barista’s BaselinedView to layout its structures and tokens based on their whitespace, fonts, and other factors.

StructureView’s width, height can be controlled by constraints. For example, this IfView’s width and height are bounded by the rightmost and bottommost edge of its children.

This IfView also uses Barista’s BaselinedView to layout its structures and tokens based on their whitespace, fonts, and other factors.

Now let’s move on to discuss how Barista uses these views to implement its standard support for text editing.
● Editing code in a Barista editor
● How to implement a Barista editor
● How Barista implements text editing
● Examples of embedded tools
Remember that every TokenView is a little text field.

When the user types in it, if the changes to the token make it the text no longer comply with the token’s type, it invokes an incremental parser, which uses the regular expressions supplied by the editor designer:

In this example, the user types a plus into the token a123, which makes it is no longer a legal Java identifier, invoking the parser.

This parsing process occurs in three simple steps. The first step is:

```java
print(a123);
print(a+123);
```
Text Editing

print(a123);

(1) On a change, unparse the tree into a list

(2) Tokenize the modified text

(3) Parse the new token stream

print(a+123);

...to “unparse” the affected part of the program from an abstract syntax tree into a list of tokens. This is determined by an algorithm that analyzes the structures and the grammar. Any parts of the program that are unaffected are left alone.

Now, what exactly “unaffected” means is a bit subtle, so you can refer to the paper for details.
(1) On a change, unparse the tree into a list

(2) Tokenize the modified text

(3) Parse the new token stream

The next step is to tokenize the modified token and its neighbors. In this example, the `a+123` gets tokenized into three new tokens, `a`, `plus`, and `123`. 

```
print(1+a123);
print(1+a+123);
print(1+a+123);
```
The last step is to parse the new list of tokens into a new abstract syntax tree. While this happens, any existing structures in the program are reused. This is possible because each token and structure points back to the structure that contains it. For example, the print points back to the invocation, so the invocation can be reused. This ensures that a minimal amount of information is lost as the user is typing.

Once this is complete, the new tree then replaces the old tree, and their corresponding views are updated automatically by Barista.

This process happens seamlessly while a user is editing...
...so as you can see here, all of the parsing happens behind the scenes as the user is typing. The user can use the caret like in any other editor, with support for selections and copy and paste, and keyboard navigation. The editor is perfectly happy to accept typos, backspaces, and even undo.

Aside from this basic support for editing code,
...so as you can see here, all of the parsing happens behind the scenes as the user is typing. The user can use the caret like in any other editor, with support for selections and copy and paste, and keyboard navigation. The editor is perfectly happy to accept typos, backspaces, and even undo.

Aside from this basic support for editing code,
...Barista enables a new class of embedded tools, many of which we’ve prototyped in our Java editor in order to demonstrate the power of the toolkit.

Let's discuss a few examples.
Name Resolution Errors

Most editors have some form of feedback about name resolution errors. For example, a method or variable name that can’t be found. Using Barista, this type of feedback was particularly easy to implement, and provides more immediate feedback than other editors. As you can see, as the user changes the name of the factorial function, red underlines appear as the user is typing.

To implement this, just four lines of code were required. Each IdentifierToken has a binding property, which is constrained to the result of the resolve method().

Each IdentifierView has a red line at its bottom, whose transparency constrained so that when the binding chances to nothing, the line is shown automatically. When it changes to something the line is hidden.

Now part of this simplicity comes from the fact that we have constraints, and part of it comes from the fact that we have a nice modularized structure for the editor and data types.

• Each IdentifierToken has a binding property, which is constrained to the resolve method.

• Each IdentifierTokenView has a red line whose visibility is controlled by a constraint.

Each IdentifierTokenView has a red line at its bottom, whose transparency constrained so that when the binding chances to nothing, the line is shown automatically. When it changes to something the line is hidden.

Most editors have some form of feedback about name resolution errors. For example, a method or variable name that can’t be found. Using Barista, this type of feedback was particularly easy to implement, and provides more immediate feedback than other editors. As you can see, as the user changes the name of the factorial function, red underlines appear as the user is typing.

To implement this, just four lines of code were required. Each IdentifierToken has a binding property, which is constrained to the result of the resolve method().

Each IdentifierView has a red line at its bottom, whose transparency constrained so that when the binding chances to nothing, the line is shown automatically. When it changes to something the line is hidden.

Now part of this simplicity comes from the fact that we have constraints, and part of it comes from the fact that we have a nice modularized structure for the editor and data types.
Each MethodDeclaration can declare a block of HTML as metadata, and its view portrays it above the header.

Programmers often use textual comments to store annotations and other meta data about code, but text is a very restricted medium for such information. The view shown here allows users to provide several kinds of rich information about this method, including example code and a diagram.

To implement this, just five lines of code were required. Each MethodDeclaration declares a block of HTML, and the corresponding MethodDeclarationView simply displays it above its method header. There’s also an event that allows users to double-click the info to collapse and expand it.
A special method invocation view was defined for invocations of `Math.sqrt()` and other functions.

Conventional views are used when the text caret is inside the expression.

To improve the readability of code, we implemented alternative views for several standard arithmetic functions, but we use a conventional view when the code is being edited. For example, when the user moves the text caret into the square root, it switches to a conventional view of the code, and when it moves out, it switches back to a pretty printed mode.

To implement this, just 12 lines of code were required. A special method invocation view was defined for method invocations that call `Math.sqrt()`. When the caret is inside the invocation, it reverts to the standard view.
Most code editors allow users to collapse and expand blocks of code. Our Java editor allows users to shrink blocks, so they take up less space, but are still legible.

Implementing this required a single 3 line event handler for a double-click event, to toggle a BlockView’s scale between 100% and 50%.

```java
public class FruitPainter {

    Paints a fruit based on the shape supplied.

    public void paintFruit(Shape shape) {
        int left = shape.minX();
        int top = shape.minY();
        int right = shape.maxX();
        int bottom = shape.maxY();

        if (shape.round()) {
            shape.red() shape.green()
        }

        Image apple = load("apple.png");
        paint(apple, 40);
    }
}
```
Programmers often need to try alternative expressions in code, but often need to go back to previous expressions that they’ve tried. Rather than having to comment them out, our Java editor allows users to store the alternative expressions in a special tool. These expressions are saved along with the code, as XML inside of comments.

To implement required about 15 lines of code: which involved implementing the view of alternatives required, and adding a list of alternatives to the Expression definition.
Limitations

- Requires a mature graphics platform
  - Doesn’t have the same universal portability like vi and emacs
- Requires an editor designer to provide syntax and semantics about a language
  - No more than is required to make a mature Eclipse code editor for a language

Now I’ve talked a lot about the benefits of Barista, but it does have some limitations when compared to other types of code editors.

For example, Barista requires a mature graphics platform, like any standard desktop OS. So you’re not going to be able to log in to a UNIX machine remotely and edit like you would with vi and emacs.

Furthermore, to create a Barista editor, editor designers have to supply a lot of information about the syntax and semantics of a language. In fact, this really isn’t any more than would be required to make a mature Eclipse code editor for a language. Furthermore, the information that does have to be specified is much more modularized than in plain text editors, enabling people to customize editors more easily.
Limitations

- Performance
  - Thousands and thousands of text fields!
  - Largely an engineering challenge, rather than research challenge.

- Serialization
  - Barista editors save code and metadata as XML by default.
  - Can be saved in target language’s syntax, but must be reparsed on every load

Performance is also more of an issue with Barista editors, essentially because there are thousands of little text fields floating around the screen. Managing all of these views, constraints, and other information is not trivial, but it’s largely an engineering issue, and not a fundamental research challenge.

Another limitation is that Barista editors save code as XML by default, in order to store meta information like alternative expressions. Although code can be saved in the target language’s syntax, and store meta information in comments, code must be reparsed every time its loaded.
Limitations

• Non-traditional layouts can make the text caret move unpredictably while typing
  – But it’s still deterministic, and thus learnable

• Small syntax errors on large structures (such as removing a { } can have a profound impact on the view of the code
  – Editor designers should offer interaction techniques for performing structural edits, or prohibit such edits altogether.

There are some interactive limitations as well. For example, because the views are often laid out in non-traditional ways, the text interaction technique can be somewhat unpredictable. However, where the caret moves is still deterministic, it’s just a matter of learning the layout of the tokens. It’s no worse than some of the text caret movements in Microsoft Word.

One final issue is that small syntax errors on large structures (such as removing a curly brace from a class definition) can have a profound impact on the view of the code. So editor designers should offer interaction techniques that perform edits structurally, or prohibit such large structured edits altogether. The nice thing about Barista is that the interaction techniques for each structure can be individually controlled.
In the future, we’d like to develop a fast, feature complete Java editor. The prototype we have now is very pre-alpha, so we don’t feel comfortable releasing it yet.

We’d also like to open source the toolkit and editors that we’ve created.

And obviously, we’d like to use this new platform to invent new kinds of embedded tools, and we’d like the community to do so as well.
This work was supported by the National Science Foundation under NSF grant IIS-0329090 and as part of the EUSES consortium under NSF grant ITR CCR-0324770. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the National Science Foundation. The first author was supported by an NDSEG fellowship.