Frame

A Language for Rapid Interface Prototyping

Eddy Varela & Taylor Beebe

1 Introduction

Our programming language addresses the repetitive and tedious nature of writing HTML code by providing a language with simple semantics and grammar rules which allows users to build more with less. Creating a new webpage from scratch can be extremely time-consuming, causing most smaller web developers to alter pre-existing themes. Larger companies typically hire designers to draft mockups for various interfaces, but this process is slow, expensive, often a luxury for smaller companies.

The goal of Frame is to give developers the power of a modern programming language when developing in HTML. The repetitive and verbose nature of HTML makes it challenging to focus on design when developing. Modern programming language features such as variables, looping, and list structures would greatly simplify HTML generation. Our current version of Frame implements many of these useful features, such as variable declaration and reuse, comma-separated lists which compile to individually wrapped expressions, and a simple commenting mechanism. We'l look at some of the benefits of these features in the Examples section.

Portability has been another primary goal during the development of Frame. We believe businesses and individuals shouldn't have to worry about how their work will port to various systems and screen dimensions. A universal language like Frame allows fast HTML development across multiple systems and significantly reduces the overhead of developing on multiple platforms.

In the future, Frame will be able to output simple and syntactically correct code for many different web-development languages, though it currently only compiles to HTML. Now that a robust parser has been created, the next goal is to create multiple interpreters which can produce Javascript, Swift, and Java code. This lightweight language could provide a universal way of prototyping regardless of screen dimensions or platform. Our language will allow developers to mock up interfaces with less code and in less time than any other language.

Our initial version of Frame handles different kinds of text, containers, nested containers, navigation bars, variables, and repetitive structures.

2 Design Principles

The fundamental design principle behind Frame is modularity – we want users to build out components which can be later combined. Frame has been created with readability in mind, so there are many different ways to write a program which outputs the same result. For example, primatives in our language can be expressed in a verbose way (such as Frame(HeadText("Hello")) or using shorter notation (such as fr(ht("Hello")).

Frame is designed to be incredibly flexible, meaning we don't do much checking when parsing the input. If the user wants to combine expressions which don't make sense, we see no reason to stop them. As long as the user input compiles to a syntactically correct program, they are free to piece together expressions as they see fit.

3 Examples

Example 1

sample1.fr contains
```
fr(fr(ht("Hello World")));
```

```
> dotnet run sample1.fr
Success! Check frame.html
```

frame.html Contains
```
<HTML Boilerplate code/>
...
<body>
    <div>
        <div>
            <h1>
                Hello World
            </h1>
        </div>
    </div>
</body>
```

Example 2

sample2.fr Contains
```
fr(ht("Hello"), ht("World"));
```

```
>dotnet run sample2.fr
Success! Check frame.html
```

frame.html contains
```
<HTML Boilerplate code/>
...
<body>
    <div>
        <h1>
            Hello
        </h1>
        <h1>
            World
        </h1>
    <div>
</body>
```

Example 3

sample3.fr Contains
```
fr(ht("Hello","World"));
```

```
>dotnet run sample3.fr
Success! Check frame.html
```

frame.html contains
```
<HTML Boilerplate code/>
...
<body>
    <div>
        <h1>
            Hello
        </h1>
        <h1>
            World
        </h1>
    <div>
</body>
```

Example 4

sample4.fr contains
```
fr(
    bt(
        "\"Fizz Buzz\""
    )
);
```

```
> dotnet run sample4.fr
Success! Check frame.html
```

frame.html Contains
```
<HTML Boilerplate code/>
...
<body>
    <div>
        <b>
            "Fizz Buzz"
        </b>
    </div>
</body>
```

Example 5

sample5.fr Contains
```
x = "Fizz";
y = "Buzz";
z = fr(pt(x), pt(y));
z;
```

```
>dotnet run sample5.fr
Success! Check frame.html
```

frame.html contains
```
<HTML Boilerplate code/>
...
<body>
    <div>
        <p>
            Fizz
        </p>
        <p>
            Buzz
        </p>
    </div>
</body>
```

Example 6

sample6.fr Contains
```
//this is a comment
x = ht("Fizz Buzz");
fr(
    x //another comment
);
```

```
>dotnet run sample6.fr
Success! Check frame.html
```

frame.html contains
```
<HTML Boilerplate code/>
...
<body>
    <div>
        <h1>
            Fizz Buzz
        </h1>
    </div>
</body>
```

3.1 Language Concepts

Frame's primitives are all strings since HTML is a markup language. We however, built in some intuitive ways to combine different elements of our language which reduce repetition and allows users to write more structured code when developing interfaces. Some of our combining forms allow users to represent lists of information and composition of elements. For example, Frame(Expr) represents a div container holding an expression inside. HeadText(ListString) represents a list of strings wrapped in h1 tags. Similarly we allow users to declare variables for later use.

4 Formal Syntax

Expression =
| *EmptyList* ::= ' '
| *Digit* ::= 0|1|...|9
| *Number* ::= Number Digit
| *Character* ::= a|...|z|A|...|Z
| *Symbols* ::= ! | @ | # | $ | % | & | * | (|) | _ | + | - | { | }
| *Quote* ::= ""'
| *string* ::= Number Symbols Characters string
| *End* ::= ';'
| *Equals* ::= '='

TextExpression =
| *String* ::= Quote Number Symbols Character String Quote
| *StringList* ::= String, StringList | EmptyList
| *StringTuple* ::= String, String
| *HeadText* ::= ht(TextExpression — ContainerExpression)End
| *ParaText* ::= pt(TextExpression — ContainerExpression)End
| *BoldText* ::= bt(TextExpression — ContainerExpression)End
| *OrderedList* ::= ol(TextExpression — ContainerExpression)End
| *Variable* ::= (TextExpression — ContainerExpression)End
| *VariableDeclaration* ::= Character string Equals Variable End

ContainerExpression =
| *Frame* ::= fr(TextExpression | ContainerExpression)End
| *NavFrame*::= nav(TextExpression | ContainerExpression)End
| *Intput*::= in(StringTuple)End
| *Button* ::= btn(StringTuple)End

Frame does not recognize whitespace, allowing the developer to include as much or little as they like as long as each statement ends in a semicolon. **Example 6** shows this in use.

Users can write comments by typing two consecutive forward-slashes, followed by whatever text they want ignored. **Example 4** shows this.

Users can declare new variables by typing a string beginning with a letter followed by an equal sign and an expression terminated by a semicolon. **Examples 5** & **6** show this.

Repetitive components can be shortened by constructing lists of text components wrapped in a single Frame or TextExpression like in **Examples 2**, **3**, & **5**

Users can compose frames inside frames in order to construct complex interfaces. Eventually, we will allow the user to control the head portion of their HTML file, but for now, we have some boilerplate code to make things work. **Example 1** shows this feature.

5 Semantics of the Language

Here are the types supported in our language as of the publishing of this document.

type $Expr$ =
| $String$ of string
| $ParaText$ of Expr
| $HeadText$ of Expr
| $ListItem$ of Expr
| $BoldText$ of Expr
| $OrderedList$ of Expr
| $LinkText$ of Expr
| $NavFrame$ of Expr
| $Button$ of string * string
| $FofF$ of Expr * Expr
| $Input$ of string * string
| $Frame$ of Expr
| $VariableDeclaration$ of string * Expr
| $Variable$ of Expr

Shorthand Syntax	Abstract Syntax	Type	Meaning
link(Expr)	LinkText("str")	Expr	assoc. link to text
fr(Expr)	Frame(Expr)	Expr	wrapper
pt(Expr)	ParaText(String)	Expr	<h1> wrapper
ht(Expr)	HeadText(String)	Expr	<p> wrapper
bt(Expr)	BoldText(String)	Expr	<b> wrapper
li(Expr)	ListItem(String)	Expr	wrapper
nav(Expr)	NavFrame(String)	Expr	<nav> wrapper
btn(String,String)	Button(String,String)	String*String	button wrapper
ol(String,String,...)	Button(String)	String*Expr	ordered list wrapper
in(String, String)	Input(Type,Label)	String*String	input wrapper
String = Expr	VariableDeclaration(String,Expr)	String*Expr	Variable declaration
String	Variable(Expr)	String	Variable

6 Remaining Work

1. We want to implement ordered and unordered lists to work with an elegant solution. For example, fr(ol("foo", "bar", "baz" ,...)); or fr(ul("ufoo","ubar","ubaz")); compiles to

```
<div>
    <ol>
        <li>foo</li>
        <li>bar</li>
        <li>baz</li>
    </ol>
</div>
<div>
    <ul>
        <li>foo</li>
        <li>bar</li>
        <li>baz</li>
    </ul>
</div>
```

2. We are working on implementing forms. We are building out the software from the bottom up so we have implemented all types of input fields, labels, button, and text. We want to implement validations and locations to where the information is going to post to. Then we want to find an elegant way to layout the parameters for the From expression.

3. We want to begin implementing media types such as pictures, videos, music. We think about how many developers reference images from third party sites and how to build out some feature of the language that accounts for that. We can possibly have a media expression with a parameter telling us the type of media, a parameter for the location on the screen, and finally the reference to the media.