Simple Presentation Web App Generator

Code Name: SPWAG

Members

Lauren Zou (ljz2112)
Aftab Khan (ajk2194)
Richard Chiou (rc2758)
Yunhe (John) Wang (yw2439)
Aditya Majumdar (am3713)

Table of Contents

1 Introduction

1.1 Project Overview

The Single Presentation Web App Generator (SPWAG) programming language combines HTML, CSS, and JavaScript to generate a web-based slide-show presentation. The primary element of SPWAG is the slide; a SPWAG program produces a presentation containing one or many slides. Slides can be defined with components (e.g. text, images, and diagrams) and attributes (e.g. text color, width, and length). Code written in SPWAG generates a single HTML file with Javascript and CSS representing the components and attributes defined by the user.

1.2 Motivation

SPWAG aims to allow the user to create fast and efficient slide-show presentations without having to worry about complicated and bloated graphical user interfaces. Since SPWAG's output is a single HTML file, it is lightweight as well as compatible across all browsers, platforms, and operating system.

SPWAG aims to combine HTML, CSS, and JavaScript into a single and simple language. In SPWAG, the developer does not need to worry about which aspects of his or her webapp needs to be coded in HTML, CSS, or JavaScript. The developer can spend more time focusing on the content of the website. SPWAG will handle cross-browser compatibility, graceful degradation, and producing a visually attractive simple page web application.

1.3 Design Goals

SPWAG is an extremely clean, concise language. Through the use of spaces, tabs, and newlines, SPWAG's syntax does not require semicolons, reducing overall clutter. Moreover, the syntax for defining new components and attributes is simply the identifier of the new class and parameters enclosed with parentheses, making it incredibly succint and readable. Thus, SPWAG enabkes users to easily modify their programs to create the slides that comprise their presentations.

Moreover, SPWAG aims to be modular. Aside from primitives and functions, all classes in SPWAG consist of only three basic types: slides, components, and attributes. The slide is the basic page containing all other components; a SPWAG presentation can be viewed as a set of slides. All other objects contained in the slides are components. While the base component in SPWAG is the box, users are able to define custom components that can transform boxes into other objects such as detailed images and clickable buttons. Lastly, attributes refer to the customized aspects of the components; examples include color and size.

2 Language Tutorial

The entry point for all SPWAG programs is the statement define slide main() (equivalent to public static void main (String args[]) in Java or int main(int argc, char** argv) in C), which generates the first slide in the presentation to be created. Typically, a SPWAG program first defines the slides that will appear in the final presentation, then custom components, attributes, and lastly regular functions. In each slide, the user can create components (e.g. boxes) and assign attributes to them using SPWAG's built-in functions. All top-level variable declarations and function definitions in a SPWAG program are considered to be global, which allows them to be defined in any order the user wishes. Note that all distinct components in a SPWAG presentation must have an identifier, which is defined using the id() function.

The code for SPWAG's "Hello World" program, which produces a single slide with text "Hello World" and an image underneath, is presented below. The "Hello World" slide contains the most basic components in SPWAG, boxes. The first box displays the text "Hello world!" using the native text() function. The box also calls four native functions that describe the position of the box in the slide, the font size used, and the overall width of the box relative to the slide. The second box calls the image function to display an image of a cat located in a nearby directory.

The following is the code after pre-processing:

```
define slide main()
{
  box() {
    id("hello-world-text")
    text("Hello world!")
    padding-top(40)
    padding-bottom(20)
    font-size(40)
    width(100%)
  }
  box() {
    id("hello-world-image")
    image("../resources/cat.jpg")
  }
}
```

To add more slides to a SPWAG presentation, simply define another slide. All slides in the presentation, except the very last one, should call the **next(string *next-slide-id*)** function so that the presentation will advance on-click (or however the user defines it). Calling the similar **prev(string *next-slide-id*)** function enables the presentation to return the preceding one; note that slide main cannot call this function for obvious reasons.

3 Language Manual

3.1 Lexical Conventions

3.1.1 Identifiers

An identifier is a sequence of letters, digits, and the hyphen '**-**' ; the first character must be alphabetic. An identifier is not case-sensitive.

3.1.2 Keywords

The following is a list of reserved words that may not be used as identifiers:

attr	**else**	**isa**	**slide**	**while**
comp	**false**	**null**	**true**	
define	**if**	**return**	**var**	

3.1.3 Data Types and Variables

SPWAG contains 3 primitive data types: *integers*, *booleans* and *strings*. Variables may be declared using the keyword **var**, and all variables, if not assigned, have default value **null**. A variable declaration is distinct from its definition in that a variable is assigned a value in its definition. It is important to note that while each variable must have a unique identifier, a variable may not have the same identifier as a slide (see the Scoping section).

3.1.3.1 Integer

Integers can fall under one of two different integer types: regular integers and percentages. A regular integer literal does not require a suffix, but a percentage integer literal should always be followed by a '%' (e.g. 100%). An *integer* is declared and defined as follows:

```
var identifier = value
```

Here, ***identifier*** is to be replaced by a valid identifier, and ***value*** by a valid integer literal. The storage of integers is machine dependent, but for the most part can be assumed to be in a 32-bit signed integer representation.

3.1.3.2 Boolean

A boolean is either **true** or **false**. For conversion purposes, anything that has the value 0 or **null** is considered **false**, and everything else is considered **true**. Booleans are declared and defined as follows:

```
var identifier = value
```

Here, ***identifier*** is to be replaced by a valid identifier, and ***value*** by either **true** or **false**.

3.1.3.3 Strings

A string is a sequence of 0 or more characters (including digits, symbols, etc.). Note that a character is fundamentally an integer interpreted by its ASCII equivalent if possible. String literals are surrounded by quotes (“”) and may span more than one line. A string variable is declared and defined as follows:

```
var identifier = “string literal”
```

Here, ***identifier*** is to be replaced by a valid identifier, and ***string literal*** by a sequence of 0 or more characters.

Strings can be used to mean many things in SPWAG. Some uses include IDs for components. Every component should have an ID so that functions can reference each component using its ID. The ID for each component must be unique across its specific scope context (see the Scoping section).

Strings can also represent colors. SPWAG accepts four different color formats: RGBA, HSLA, hex with an alpha at the end, and standard HTML color names (e.g. “red”, “blue”, “yellow”, “green”). For example, to make the text color hot pink, one could write any one of the following:

In RGBA:

```
box()
  id(“a-hot-pink-box”)
  text(“I am some text!”)
  text-color(“rgba(255,105,180,100)”)
```

In HSLA:

```
box()
  id(“a-hot-pink-box”)
  text(“I am some text!”)
  text-color(“hsla(330,100,71,100)”)
```

In hex with an alpha at the end:

```
box()
  id(“a-hot-pink-box”)
  text(“I am some text!”)
  text-color(“#FF69B4FF”)
```

In standard HTML color:

```
box()
  id(“a-hot-pink-box”)
  text(“I am some text!”)
  text-color(“hotpink”)
```

3.1.3.4 Other types

Note that there are several other data types that may be assigned to variables. There is the value **null**, slide names (which is of type slide), and bound components. By bound we mean the component’s call has finished, and these can be referred to using bracket syntax. For example, slide1[“acomp”] will get the immediate inner component of slide1 that has the id “acomp”.

3.1.4 Conversions

It is illegal to assign a variable of a primitive data type a primitive data type of another type. Implicit conversions are made, however, in other places in the code. Integers valued at 0 or **null** if interpreted as a boolean is **false**, all other integer values are interpreted as **true**. Strings valued at **null** are interpreted as false, all other strings are interpreted as **true**. If an integer is concatenated with a string using the + operator, it is first converted to its string representation. If a boolean is concatenated with a string using the + operator, it is first converted to “true” or “false”. All other uses of a data type where another one is expected is illegal.

3.1.5 Operators

In general, SPWAG supports the same operators typically used in other languages. The assignment operator is binary and right-associative, and its function is just to pass a legal value to the left operand. The assignment must be consistent, which is defined to be the left and right operands having the same type.

In SPWAG, arithmetic operators are all binary and left-associative. Once again, the left and right operands must be of the same type (SPWAG supports integers, pixels, and percent types). The **+** operator has a special case in which a string can be in the right hand side expression, and concatenating that with any other data type will always produce a string of that combination.

Comparative operators are also binary and left-associate, and can only compare variables of the same type. The comparisons produce boolean values of true or false.

Finally, the logical operators of |**|** and &**&** are binary and left-associative, must have boolean-producing expressions on either side of the operator, and return a boolean value as well.

<	less than operator; can only compare variables of the same type, strings, ints, percents
>	greater than operator; can only compare variables of the same type, string, ints, percents
=	assignment operator
==	equal to

!	not
+	*string* concatenation, integer addition; addition for integers, pixels, and percents may only occur with two variables of the same data type; a *string* can be concatenated to any data type and the result will be a *string*
-	subtraction; only works for integers and percents; subtraction for integers and percents may only occur with two variables of the same data type
*****	multiplication; only works for integers and percents;
/	division; only works for integers and percents; you can divide percent with integer, but not vice versa
||	or
&&	and

3.1.6 Comments

Comments can be placed anywhere in the program. A single-line comment begins with a **#** character. Multi-lined comments are placed between two **##** character sequences. Comments do not nest and are ignored by the compiler.

```
# This is a single line comment.

## This is a multi-lined comment.
   We can use this to comment out code,
   write a poem or a story or some documentation. ##
```

3.2 Functions

SPWAG's basic unit of execution is a *function*. Functions can represent actions to be performed (e.g. on-click()), web page *components* (which are analogous to HTML elements), *slides*, and *attributes* (which are analogous to but not limited to CSS styles that can be applied to the components). *Components* and *attributes* are mutually exclusive special subtypes of functions. Functions are all global in scope, and no functions are permitted to be anonymous. All functions must be uniquely named in order to be identified, and duplicate names will overwrite any previously declared functions.

3.2.1 Function Calls

To call a function, the following syntax must be used: ***identifier*(*parameters*)**. The identifier must be previously defined as a function, while ***parameter*s** is a comma delimited list of zero or more expressions whose values will be assigned to the arguments of the function. The number of parameters in the function call must be identical to that of the function definition.

When the function is called, control shifts to the first line of the function and continues through its body until a return statement is evaluated or the end of the function body is encountered. Control then returns to the point immediately after where the function call was made. If a return statement was evaluated, the value of the function call will be that of the expression in the return statement. Otherwise, the function call will evaluate to **null**. Only function calls may have return statements.

Note that slides cannot be called as functions. Instead, a slide function is “called” the first time the slide’s name appears in the control flow. This could be the slide name by itself, as an argument to next(), or even when trying to fetch a component via slide-name[“id”][“id”] ie bracket syntax. Of course, the main slide is the first function called, so that is where control flow starts.

3.2.2 Native Functions

SPWAG includes three native functions which serve to interact with various elements during SPWAG runtime, as well as functions for performing basic mathematical functions. Fundamentally, these functions correspond to the following actions:

- get() - element and attribute retrieval
- set() - element and attribute assignment
- random(integer) - random number generation

The native **random(integer)** function generates an integer between 0 and the integer passed in the parameter inclusive. For example, random(100) will return a random integer x such that $0 \leq x \leq 100$. Potential uses for the **random()** function include generating transitions to any slide in the presentation and creating unpredictable changes to the layout of the current slide by altering the position of current components.

The **get(string *component-id*, string *attr-name*)**, and ***slide-name*[string *child-component-id*]** functions are accessor functions to retrieve specific attributes and components, respectively, that are contained within specific slides or potentially other elements.

- ***slide-name*[string *child-component-id*] -** retrieves the specified component from a slide
- **get(string *component-id*, string *attr-name*) -** retrieves the specified attribute value from the specified component

In combination, these two allow for the accessing of any SPWAG elements and their attributes. For example, consider the following code:

```
define slide main()
 text-color(“blue”)

 box()
  id(“my-box”)
  text(“Hello world!”)
  text-color(“red”)

  box()
   id(“other-box”)
   text(“Yay world!”)
   text-color(“green”)
```

```
    box()
      id("last-box")
      text("Hi world!")
      text-color("yellow")
```

The text color attribute of the **main()** slide can be retrieved with the **get(string *component-id*, string *attr-name*)** function:

```
var main-text-color = get(main, "text-color") # returns the string "blue"
```

The box component with the id "my-box" can be retrieved using the ***slide-name*[string *child-component-id*]** function:

```
var my-box = main["my-box"] # returns the my-box box component
```

In combination, these functions allow the text color attribute of "my-box" to be retrieved:

```
var my-box-text-color = get(main["my-box"], "text-color") ##returns the string "red" ##
```

The ***slide-name*[string *child-component-id*]** function brackets can also be concatenated in order to access nested components. Thus:

```
var other-box = main["my-box"]["other-box"] # returns other-box
```

Finally, once again, in combination:

```
var other-box-text-color = get(main["my-box"]["other-box"], "text-color")    # returns "green"
```

```
var last-box-text-color = get(main["my-box"]["other-box"]["last-box"], "text-color") # returns
yellow
```

Similarly, the **set(string *component-id*, string *attr-name*, string *attr-value*)** function serves as an accessor to retrieve specific attributes that are contained within specific slides or elements. Unlike with the get command, set can only be used to set attribute values.

- **set(string *component-id*, string *attr-name*, string *attr-value*) -** sets the specified attribute value to the specified attribute of the specified component. Can be used with values assigned to vars

In continuation of the above example, this would allow for the following:

```
var last-box-text-color = “green”
set(main[“my-box”][“other-box”][“last-box”], “text-color”, last-box-text-color) ## sets the
“text-color” of the “last-box” as “green”
```

The following table summarizes the native functions of SPWAG:

random(integer *i*)	randomly generates a random integer between 0 and *i*
***slide-name*[string *child-component-id*]**	retrieve the specified component from the specified slide
get(string *component-id*, string *attr-name*)	retrieve the specified attribute value from the specified component
set(string *component-id*, string *attr-name*, string *attr-value*)	sets the specified string value to the specified attribute of the specified component

3.3 Function Definition

In SPWAG, defining custom functions utilizes a standardized syntactic structure beginning with the **define** keyword. The function definition prototype is presented below:

```
define function-type identifier(parameter-list) ...
```

Here, ***identifier*** is to be replaced by a valid identifier, and ***function-type*** is a valid selection of one of the fundamental function types in SPWAG (ie: function, slide, component, attributes). This typing scheme allows each function to perform a certain task, such as specifying a custom attribute set, adding a slide, writing a reusable function, and dictating the properties of an element integer literal. Each of these options is outlined in greater detail in the table below. Additionally, the ***parameter-list*** provides the means by which to pass in values to a user’s custom function, allowing for dynamic insertion of objects during compile-time, as well as during program execution (executed at runtime via Javascript). Any values specified in this list (consisting of ***type*** and ***name***) may be accessed as local variables by any of the function’s child function calls.

Note that the preprocessor adds braces, Allman style, to all definitions, and that is what the compiler actually works on. In general, any block needs to be surrounded by curly braces before input to the compiler, and the curly braces need to be on its own lines.

define slide main()	defines the main **slide** of the slideshow; this is a required function in all SPWAG programs
define slide my-slide()	defines a custom **slide** in the slideshow, where ***my-slide*** is the custom identifier of the slide

define comp my-comp(*my-comp-parameters*) isa spwag-comp(string *component-id*, *spwag-comp-parameters*) **...**	defines a custom component; the **spwag-comp** must be either a custom component already defined somewhere in the code or a native SPWAG component such as **box()** ***my-comp-parameters*** are a list of comma-delimited parameters specific to *my-comp* ***spwag-comp-parameters*** are a list of comma-delimited parameters specific to **spwag-comp**
define attr my-attr(*parameters*) **...**	defines a custom attribute
define func my-funct() **...**	defines a custom function

3.4 Components

A component is a special subtype of a function, which represents a single visual/graphical element of a SPWAG presentation. Custom components may be defined as combinations of other functions, with the exception of **slide** (see the Function Definition section for syntax), and may have any number of attributes applied to them. After it is created, each component may be referenced by a unique **string** ID, passed in as an attribute during component creation. Any component created with the same ID as a previous component within the same scope will overwrite that component (the handle to that component will be lost).

Component functions are called via their function names. They are not "bound" to their outer component or slide until their function call ends. In addition, when a component is called, it may be modified right after the component call in an indented block of text. These statements are executed as if they are in the function of the component being called.

3.4.1 Native Components

The **box(string *id*)** forms the single *native component* and core graphical unit found in the SPWAG language. Using Attributes (including custom attributes), a box may be used to represent a text field, an image, or other content the user might like to include when creating multimedia SPWAG presentations.

box(string *id*)	creates a single graphical unit of content in a SPWAG presentation.

3.5 Slides

Slides make up the basic structure of the program and the resulting presentation. They are created using the following syntax, using the keyword **slide**, in the global scope of the program:

```
define slide my-slide()
  ...
```

In addition, there is a special **slide** called **main()**. The program begins execution through starting the main **slide**. The main **slide** is also by default the first slide in the resulting presentation, although it is possible to change this by specifying a **prev() slide**. It is defined just like any other **slide**:

```
define slide main()
  ...
```

Note that slides can only be created in the global scope of the program, as the resulting presentation is essentially made out of slides. In particular, a slide cannot be called from any other Function. In addition, all slides must have a unique name, that is, a slide's name identifies the slide. This means, for example, if my-slide is created as above, there cannot be any other slide named my-slide in the whole program.

The body of a slide is just like the body of any other component—it can contain function calls, component calls, or attribute calls. Calling a function executes that function, calling a component creates the corresponding component as part of the slide, and calling an attribute changes the corresponding attribute of the slide. Technically, slides can make any of the Native Attribute calls listed under Native Attributes, but only certain attributes actually make a difference to the slide. See Native Attributes for details.

3.6 Attributes

Attributes describe the properties of the component upon which they are specified. Attributes may be thought of as analogous, but not limited, to CSS styles, allowing the user to specify traits such as color, border, etc. Custom attributes may be specified by the user (the declaration syntax of which is described in the Function Declaration section). A custom attribute is a collection of other attributes. Custom attributes are used to implement reusable modular styling. However, custom attributes may not call components at any point in the program (Thus, any function eventually called by an attribute cannot call components).

Note that attributes cascade. This means that, for example, if there is a component inside another component, the program first looks at the Attributes applied to the inner component in order to format the inner *component*, and if any of those Attributes are **null**, it looks at the corresponding Attribute as applied to the outer component in order to format the inner component.

3.6.1 Native Attributes

Fundamentally, there is a fixed set of native attributes which may be applied to any component or slide, which is described in the following table. All custom attributes must eventually terminate in a selection of these native attributes.

position-x(integer *x*)	*Component*: Specifies the absolute horizontal position of the component on the slide where 0px and 0% is the left-most side of the slide; can be a pixel or a percentage *Slide*: Does not do anything
position-y(integer *y*)	*Component*: Specifies the absolute vertical position of the component on the slide where 0px and 0% is the top-most side of the slide; can be a pixel or a percentage *Slide*: Does not do anything
margin-top(integer *margin*)	*Component*: Specifies the amount of outer spacing at the top of the component *Slide*: Does not do anything
margin-bottom(integer *margin*)	*Component*: Specifies the amount of outer spacing at the bottom of the component *Slide*: Does not do anything
margin-left(integer *margin*)	*Component*: Specifies the amount of outer spacing at the left of the component *Slide*: Does not do anything
margin-right(integer *margin*)	*Component*: Specifies the amount of outer spacing at the right of the component *Slide*: Does not do anything
padding-top(integer *padding*)	*Component*: Specifies the amount of spacing inside the top boundary of the component *Slide*: Specifies the amount of spacing inside the top boundary of the slide
padding-bottom(integer *padding*)	*Component*: Specifies the amount of spacing inside the bottom boundary of the component *Slide*: Specifies the amount of spacing inside the bottom boundary of the slide
padding-left(integer *padding*)	*Component*: Specifies the amount of spacing inside the left boundary of the component

	Slide: Specifies the amount of spacing inside the left boundary of the slide
padding-right(integer *padding*)	*Component*: Specifies the amount of spacing inside the right boundary of the component *Slide*: Specifies the amount of spacing inside the right boundary of the slide
text-color(string *color*)	*Component*: Specifies the color of the text in the component (accepts RGBA, HSLA, hex) *Slide*: Specifies the default color of the text in the slide (accepts RGBA, HSLA, hex)
background-color(string *color*)	*Component*: Specifies the background color of the component (accepts RGBA, HSLA, hex) *Slide*: Specifies the background color of the slide (accepts RGBA, HSLA, hex)
font(string *font-family*)	*Component*: Specifies font family of any text in the component *Slide*: Specifies the default font family of any text in the slide
font-size(integer *size*)	*Component*: Specifies font size of any text in the component (accepts the font size as a pixel) *Slide*: Specifies the default font size of any text in the slide (accepts the font size as a pixel)
font-decoration(string *decoration*)	*Component*: Specifies font attribute (accepts “bold”, “italic”, “underline” values) of any text in the component *Slide*: Specifies default font attribute (accepts “bold”, “italic”, “underline” values) of any text in the slide
border(integer *size*)	*Component***:** Specifies a border around the component with a specified stroke width *Slide***:** Specifies a border around the slide with a specified stroke width
border-color(string *color*)	*Component***:** Specifies the color of a component’s border, if it exists

	*Slide***:** Defines the color of a slide's border, if it exists
width(integer *width*)	*Component***:** Specifies the width of the component; can be pixel or percentage *Slide*: Does not do anything
height(integer *height*)	*Component***:** Specifies the height of the component; can be pixel or percentage *Slide*: Does not do anything
next(string *next-slide-id*)	*Component*: Does not do anything *Slide*: Specifies the slide that goes after the current slide
prev(string *prev-slide-id*)	*Component***:** Does not do anything *Slide*: Specifies the slide that comes prior to the current slide
image(string *url*)	*Component***:** Specifies an image to be displayed based on a passed-in link to the image file *Slide*: Specifies a background for the slide image to be displayed based on a passed-in link to the image file
text(string *text*)	*Component***:** Specifies the text to be displayed based on a string that is passed in *Slide*: Does not do anything
display(bool boolean)	*Component*: Specifies whether the component is shown (display: true) or hidden (display: false) *Slide*: Does not do anything
on-click(function *f*)	executes a function *f* when the user clicks on the component that this function is under
on-press(string *key*, function *f*)	binds an **on-click** function to a key press

3.6.2 Setting Component Attributes

Attributes for a component are set upon the initialization of the component. For example, the following code declares a text box with a pink background, red text, and a border:

```
box()
  id("my-box")
  text("Text goes here.")
  background-color("pink")
  text-color("red")
  border(1px)
```

This box can be changed later in the program. In order to change the box, one must first get the box using the **get(string *component-id*)** function and then list the new attributes under the box. For more information on this, please see the "Native Functions" section above, describing element and attribute retrieval and setting.

3.7 Program Structure

3.7.1 Statements and Control Flow

Programs consist of blocks of statements. A statement is a complete line of code. Statements include valid expressions, variable assignments, control structures (i.e. if and while), or function definitions. Assignment statements are used to assign a value to an identifier (or variable), and have the following syntax: ***identifier = expression***

3.7.1.1 Code Blocks and White Space

A code block is a set of tabbed statements under a header statement (e.g. a function declaration). The **end** keyword ends a block of code, started by a **define** statement (defining a function), or an **if**, **else**, or **while** block.

Since SPWAG does not utilize brackets to surround blocks of statements, white-space indentations delimit the structure of the program. Each indentation of white space indicates a different block in the program. For example, in the following code:

```
define slide main()
  box()
    id("ss-text")
    text("Welcome to my slide show!")
    font("Consolas")
    font-size(20px)
    font-decoration(bold)

  box()
    id("ss-img")
    image("cats.gif")
    border(5px)
    border-color("#222")
```

There are three blocks: the block under **slide main()**, the block under **box("ss-text"),** and the block under **box("ss-img")**. Each block modifies the contents of the statement that it lies under. So

box("ss-text") and **box("ss-img")** are components in the main slide, the font attributes describe what the font of the text looks like, and the border attributes describe what kind of border the image has.

The unit of white-space does not matter in SPWAG. A unit of white-space can be two spaces, four spaces, a tab, etc. The only unit of white-space that is accepted is a single (1) space character. All empty lines are ignored in SPWAG.

3.7.1.2 Control Structures

SPWAG supports two types of keywords that allows for algorithmic, conditional execution of particular blocks of code: the **while** keyword (continually executes a block of statements while a particular condition is **true**), and the **if** keyword (execute a certain section of code *only if* a particular test evaluates to **true**). An end keyword will return control to outside the conditional block. Optionally, a block with an if header statement may be immediately succeeded by a block with an else header statement, which must be terminated by an end statement.

if <condition>
  ...
else
  ...
	if-else conditional

while <condition>
  ...
	while loop

3.7.1.3 Precedence

Expressions and any operators associated with them (discussed in the next section), are applied in the following order, listed from highest to lowest precedence).

()	parenthesized expressions
function(parameters) get-attr(attribute)	function calls, referencing
*** /**	multiplication, division
+ -	addition, subtraction
== != < >	comparison
!	logical negation
&&	and
||	or

3.7.2 Basic Expressions

The following are all considered to be direct expressions:

- Identifiers, which refer to variables or functions

- Primitive types (i.e. integers, boolean, strings)
- Components
- Attributes
- Slides

The sole unary expression in SPWAG is **!**, i.e. the negation operator. This operator, used with a single boolean constant or a boolean variable, gives a boolean type with value opposite that of the operand. Unary expressions are grouped right-to-left.

SPWAG implements four standard arithmetic operators: **+**, **-**, *, and **/**. These operators are defined for any two integers, producing the arithmetic sum, difference, product, and quotient (all integers) of these numbers, respectively. While **+** and **-** are commutative, * and **/** are not.

String concatenation is performed using the **+** operator as follows: ***string + string***, where each string is a string literal or a string variable. The expression evaluates to a single string, the concatenation of the two string operands.

Parenthesized expressions have type and value are identical to those of the expression within the parentheses.

3.7.3 Logical Expressions

There are four types of logical expressions: direct boolean expressions, unary boolean expressions, comparison expressions, and binary boolean expressions, all of which evaluate to boolean values.

3.7.3.1 Unary Boolean Expressions

Unary boolean expressions involve boolean expressions that are immediately preceded by the negation operator (**!**). These expressions evaluate to the boolean value opposite that of the expression directly following the negation operator. The **!** operator has higher precedence than the other boolean operators (**&**, **|**), but lower precedence than the equality and comparison operators.

3.7.3.2 Binary Boolean Expressions

Binary boolean expressions have the following form: ***operand operator operand***. Operands are boolean expressions, while the operator is either the **&** or **|** operator. Operations involving the **&** (and) operator return **true** if both operands have the value **true**, and **false** otherwise. The **|** (or) operator returns **false** if both have the value **false**, and **true** otherwise. The **&** operator has higher precedence than the **|** operator, and the equality comparison has higher precedence than both the **&** and **|** operators.

3.7.3.3 Comparison Expressions

Comparison expressions share the same form as binary boolean expressions. Operands can be of any primitive type, an identifier whose value is a primitive type, or an expression which evaluates to a primitive type, while operators may be one of **==**, **!=**, **<**, or **>**. The operands must have the same type.

The **==** and **!=** operators are valid for all primitive types. The **==** operator functions as an XNOR comparison, returning **true** if both operands have the same value and **false** otherwise. The **!=** operator functions as an XOR comparison, returning **true** if the operands have different values and false otherwise. (For strings, character-wise comparisons are performed.)

The **<** and **>** operators are valid for integers, but not strings and booleans. For integers, the strictly less than operator (**<**) returns **true** if the first operand is less than the second, and **false** otherwise. The strictly greater operator (**>**) returns true if the first operand is greater than the second, and false otherwise.

3.7.4 Scoping

3.7.4.1 Global Scope

Global scope is equivalent to file scope. That is, when the keyword **import** is used to combine code in several different files, the code in those different files are treated as if they are in one file. All Functions, then, are "global" in scope. That means all functions are visible from any other Function.

This visibility does not mean, however, that all functions may be called from other functions - other rules, not associated with scoping, apply (see Function specific headings above). In particular, specific instances of slide or of a component must be retrieved using the native function **get()**.

In addition, all variables declared outside any function are global in scope. However, unlike functions, the scope of a global variable begins from when it's declared. In particular, since an **import** statement replaces the statement with all the code from the file to be imported, any variables declared in the file to be imported will only be visible starting from the **import** statement.

Finally, there is the unique scope of an id passed in when creating a *component*. This id must be unique to the *component* or *slide* in which its corresponding *component* is contained. See **get()** in the Native Functions section for more details on how *components* that are created can be uniquely accessed.

3.7.4.2 Block Scope

The other scope considered is local scope within a "block", or block scope. Because all functions are global, it makes sense to consider only variables in the local scope. All variables in a block are visible starting from when they are declared in the block to the end of the same block. In particular, a function formal parameter is visible throughout the whole block that makes up the function. Finally, when defining inheritance, formal parameters for the component being defined is immediately visible to the call to the component being inherited from after the **isa** keyword.

3.7.5 Inheritance

Only components may have inheritance, and in fact, all custom components must inherit from another component. See the Function Definition section for details on how to use the **isa** keyword to define custom components.

Component inheritance can be thought of in this way. Assume **comp1** inherits from **comp2**. Then, the syntax for defining **comp1** would be as follows, where ***id*** can be any valid identifier, and (…) refers to a list of parameters:

```
define comp comp1(id, ...) isa comp2(id, ...)
  ...
end
```

When **comp1** is called (that is, created), **comp2** is first called (that is, created) using the parameters passed in. Next, the body of **comp1** is evaluated on the instance of **comp2** created. This changed component is then referred to as an instance of **comp1**, uniquely identified by the passed in ID. Note that in the above example, ID is separated from all other parameters because it is the first parameter passed into **comp1**, and must be directly passed into **comp2** as the first parameter. This is the string passed in that is the unique identifier of the component when it's created.

Finally, note that because any components created by **comp2** will necessarily be created when comp1 is called, unique ID's assigned to components created by **comp2** cannot be repeated by any components created by comp1. The native component **box()** does not create any components in its definition.

3.8 Execution

SPWAG compiles to a single HTML file (.html file extension), containing all necessary HTML, CSS, and JavaScript code. As such, SPWAG output HTML files may be executed by any modern web browser, and, as such, are platform agnostic and HTML5/CSS3 standards compliant.

4 Project Plan

Our group met at least once per week to discuss the design of the language, current progress, and upcoming goals. In particular, the Language Reference Manual (LRM) underwent numerous changes to further streamline the syntax, and thus the user experience, of SPWAG.

The development process for our language can be divided into three phases. During the first phase, the entire group worked on the scanner, parser, and abstract syntax tree (AST) using the specifications listed in the LRM. During the second phase of development, group members were assigned individual tasks; the IR specification and IR generator, the semantic analyzer and the semantic abstract syntax tree (SAST), and the compiler were primarily coded during this period. After these files were written, group members moved on to writing test drivers and examining their code. In the final phase of development, the preprocessor and final interpreter (spwag.ml) were written, allowing for programs written in SPWAG to be compiled, tested, and outputted to the desired HTML/JS/CSS page.

4.1 Directory Structure and Programming Styles

The primary components of the compiler (detailed under Part 5: Architectural Design), along with the Makefile to build the compiler, were stored in the main directory (plt-spwag). This home directory contains a Makefile that builds the compiler, a bash script (testcases.sh) that generates the output of any or all of the programs in the testcases subdirectory, and three subdirectories. The test subdirectory stores all files used to test the components of the compiler such as the AST, the compiler, the IR, and the preprocessor. The testcases subdirectory contains all examples of SPWAG code used to demonstrate the capabilities of the language; examples include a "Hello World" slide, the actual presentation used at our final project presentation (index.html), and fiver, a program named after the eponymous game that demonstrates the algorithms, operations, and change of control flow in SPWAG. Lastly, the config folder stores the default CSS and JS used to generate the standard main slide in a SPWAG program.

SPWAG structurally resembles the Python programming language. In SPWAG, tabs represent 4 spaces and signal new blocks of code. Instead of snake case or camel case, SPWAG uses hyphens in its ids (for example, hello-world is the convention in SPWAG as opposed to helloWorld or hello_world).

SPWAG compiles to one HTML file called index.html. To make sure that our testing environment is organized, we have created a directory structure so that each test case has its own folder with at least an output folder inside. The compiled HTML file will be saved under output/index.html and the original SPWAG source file can be found as test.spwag under the test case's folder.

4.2 Project Timeline

Date	Task to Accomplish
September 25	Project proposal complete
October 28	Language reference manual complete
November 29	Scanner complete
December 6	Parser and AST complete
December 11	Basic "Hello World" program
December 16	Translator complete
December 18	Testing and debugging
December 20	Final report and presentation

4.3 Roles and Responsibilities

Group Member	Contributions
Lauren Zou	conceived of idea of SPWAG scanner.mll parser.mly ir.ml compile.ml spwag.ml ir and compile test cases

	spwag test cases / testcases.sh javascript and css configuration files
Aftab Khan	preprocessor.c scanner.mll parser.mly spwag.ml ir.ml irgenerator.ml ir and preprocessor test cases
Richard Chiou	project leader scanner.mll parser.mly ast.ml sast.ml semantic_analyzer.ml and testing
Yunhe (John) Wang	scanner.mll parser.mly linecounter.ml ast.ml ir.ml irgenerator.ml sastinjector.ml spwag.ml ast testing
Aditya Majumdar	ast.ml scanner.mll parser.mly semantic_analyzer.ml ast/sast testing

4.4 Development Environment

Our proposal, LRM, and write-up were done collaboratively in Google Docs. Initially, we started coding the Scanner, Parser, and AST using Google Docs to finalize the basic setup of these important files, but as the code base quickly grew, we moved to Git/GitHub for version control.

Nearly all of the code is written in OCaml, an exception being the preprocessor (written in C). No special IDEs were needed; code was just written in our favorite text editors (vim, Sublime Text, Notepad++, etc). As mentioned before, SPWAG compiles into one html file, which is rendered easily by all common web browsers and thus straightforward to test.

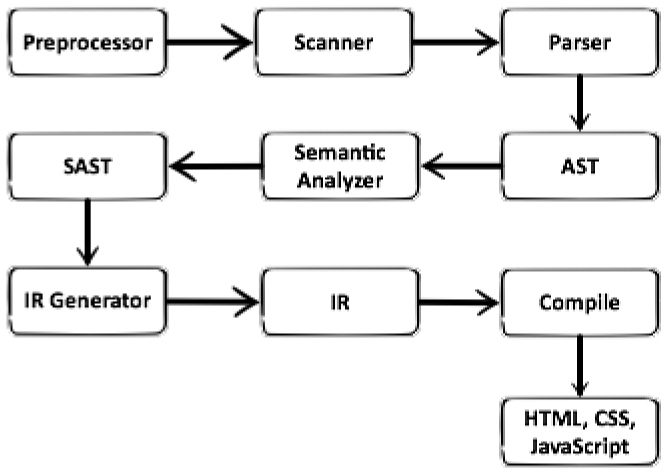
4.5 Project Log

Date	Milestone

September 18	Project idea proposed
September 20	Project idea, purpose, and functionality further developed
September 24	Project proposal completed
October 9	GitHub project created and collaborators added
October 14	Initial meeting with TA (John Sizemore) to evaluate proposal feasibility, recommendations
October 25	Started on language reference manual (LRM)
October 27	Language reference manual completed
November 14	Started implementing scanner, parser, and AST
November 24	Completed scanner and made more improvements on parser and AST
December 5	Parser and AST almost completed, in the process of debugging
December 10	Outlined final project write-up Structure of SAST discussed, implementation started Wrote up skeleton of semantic analyzer (in conjunction with SAST)
December 11	Completed two test cases (hello-world and random-colors) SAST improved, accommodates parser correctly Semantic Analyzer improved to check simplest type instance conformity
December 12	Redesigned element and attribute addressing / retrieval Improved AST, Parser, Scanner and subsequent updates to SAST Completed majority of spwag.ml
December 13	Wrote toString testing methods to test functionality of and finalize AST Restructured some code segments between SAST and semantic analyzer SAST compiles fully
December 14	Completed “one-fish-two-fish” test case Added JavaScript functionality for switching between slides Continued debugging semantic analyzer
December 15	Finalized intermediate representation data structure Continued adding content to semantic analyzer + debugging
December 16	Func_Definitions analysis added to semantic analyzer (including some basic testing) More statements/expressions further defined in semantic analyzer
December 18	Created test cases for irgenerator.ml and compile.ml Progress on IR generator Compile prints out some CSS
December 19	Compile prints out CSS and HTML Wrote Preprocessor, linked to spwag.ml

	Rewriting parts of semantic analyzer
December 20	Completed semantic analyzer Finished Compile file Wrote a test case to display our final presentation using SPWAG code

5 Architectural Design



5.1 Component Interface Interaction

5.1.1 Preprocessor (preprocessor.c: written by Aftab Khan)

The preprocessor scans the SPWAG source file from Standard In and converts it into a more concretely structured form suitable for scanning, parsing, and determining scoping and code hierarchy. Functionally, the preprocessor handles SPWAG's tab-sensitive syntax, converting it into a form delimited by curly braces (no longer tab-sensitive) according to parent/child structural context and control flow.

5.1.2 Scanner (scanner.mll: all group members contributed)

The scanner processes the source file that passed through the preprocessor and transform it into a series of lexicals tokens for the parser to interpretate. The scanner also detects for the presence of invalid characters considered to be invalid in SPWAG.

5.1.3 Parser (parser.mly: all group members contributed)

The parser takes the tokens generated by the scanner and checks for syntax errors, determining whether they can be used based on the context-free-grammar represented by SPWAG. The parser produces an abstract syntax tree after processing these tokens.

5.1.4 AST (ast.ml: written by Richard Chiou, Yunhe(John) Wang, and Aditya Majumdar)

The abstract syntax tree, representing the syntactic structure of the source code, defines the primary relationships between the tokens used in SPWAG. The AST is used for the semantic analyzer, the compiler, and spwag.ml.

5.1.5 Semantic Analyzer (semantic_analyzer.ml, written by Richard Chiou with contribution from Aditya Majumdar)

The semantic analyzer traverses the nodes of the abstract syntax tree, checking each node to ensure that the source code is written properly. The semantic analyzer serves many purposes: aside from assigning proper types to expressions and evaluating statements and blocks properly, the semantic analyzer must manage a symbol table, keeping track of the identifiers and functions defined across the multiple scopes of the SPWAG source code. Additionally, the semantic analyzer checks to see if the function calls and function definitions in the source code satisfy the appropriate structures defined in the LRM. All AST nodes passed through the semantic analyzer are converted to nodes in the type-checked semantic abstract syntax tree (SAST).

5.1.6 SAST (sast.ml: written by Richard Chiou and Yunhe(John) Wang)

The SAST is very similar to the AST, with most of its nodes structurally similar to those of the AST. However, the SAST contains types that are assigned to the expressions evaluated by the semantic analyzer. The SAST is used by the IR generator in the creation of the intermediate representation of the SPWAG source code.

5.1.7 IR generator (irgenerator.ml: Written by Yunhe(John) Wang with contribution from Aftab Khan)

The IR generator takes in the AST and SAST structural forms of the SPWAG source in order to develop a concrete hierarchical structure for the compiled output, conducive to a straightforward conversion to HTML with appropriately defined CSS (see IR.ml for more information). Additionally, the IR Generator is responsible for resolving static function references in the code (resolved into static CSS and HTML references), such that they can be determined at compile-time, as opposed to dynamic functions, which are evaluated during runtime via javascript.

5.1.8 IR (ir.ml: Written by Yunhe(John) Wang and Lauren Zou)

IR stands for “immediate representation,” and serves as the mid-way data structure that exists between the SAST and the compiled HTML, CSS, and JavaScript. The IR is a hierarchical record that represents each element, each of its respective child elements, and its applicable style attributes. Much of the IR is very much a one-to-one correspondence to the compiled CSS and HTML, since they both possess a roughly identical hierarchical structure. The dynamic function calls and function definitions from the SAST are the only function calls and function definitions left in the IR. These dynamic functions are to be interpreted by the compiler and compiled into JavaScript.

5.1.9 Compiler (compile.ml: Written by Lauren Zou)
The compiler takes the IR and translates it into HTML, CSS, and JavaScript. The compiler is the last step in the process and produces one HTML file with the CSS and JavaScript embedded at the end. The compiler is passed a list of slide components with element components embedded in lists within them as well as a list of dynamic function definitions. The compiler iterates through the list of slides and their respective elements (with their respective elements and so on) three times; the first time to abstract out the CSS, the second time to parse out the HTML, and the third time to abstract out the on-click and the on-press functions into the JavaScript. In order to assist with cross-browser compatibility and ease of object manipulation, jQuery is the preferred library in the compiler. Any getters and setters translated into JavaScript use jQuery syntax. The function list is then also translated into JavaScript so that other methods can call them.

5.1.10 Spwag.ml (Written by Aftab Khan, Yunhe(John) Wang, and Lauren Zou)
Spwag.ml serves as the top-level program, providing the executable entry point to the SPWAG compiler after it has been compiled, and distributing tasks to the individual compiler modules according to execution flags specified by the user.

6 Test Plan
Throughout the process of writing the compiler, we used various testing methods to test our programs. For testing the scanner, parser, and AST, we had raw SPWAG source code to feed into the programs. For IR generator and compile, we created data structures of the IR and of the SAST to feed into their respective programs. These code snippets can be found in the test folder of our project. Besides the small test cases that we used to debug our code, we created four test cases for the SPWAG language. Below are the four test cases created for this language. These test cases demonstrate the various features of SPWAG. To execute these test cases, we wrote a bash script called testcases.sh which ran all the test cases or ran a specific test case if given a specific test case s a command line argument.

6.1 Test Cases

6.1.1 hello-world
hello-world is the most basic test case. As seen in part 2, this test case includes just one slide with a text box and an image. There are no custom components, custom attributes, or functions. The SPWAG output is a one-slide presentation, which means that the next and prev key bindings and general functionality are disabled.

6.1.2 one-fish-two-fish
one-fish-two-fish is a test case similar to the hello-world test case. In this test case, there are multiple slides, which means that the next and prev key bindings and hover clicks are enabled for all slides that have a next, prev, or both. This test case also uses custom components. The two different custom components in this test case are poem-text-box and fish-box. Poem-text-box is a component that adds padding and centers the text in a box. Fish-box is an image of a fish with a custom background

color so that the fish can change colors. The significance of this test case is to test custom components without worrying about interference from functions.

6.1.3 presentation

presentation is a test case that combines the use of custom components and attributes. presentation uses several different functions to define global colors and different custom components to define a simple and flexible template for the entire slideshow. This presentation also features a changebackground() function which changes the background of slides and components when invoked by an on-click(). This presentation was used to demonstrate the final project to Professor Edwards, highlighting SPWAG's powerful ability to generate elegant, dynamic presentations without overhead or additional software.

6.1.4 fiver

7 Lessons Learned

7.1 Lauren Zou

During this project, I learned valuable skills on how to work with a team. At first, I was hesitant to work with multiple people since I already had an idea of what kind of compiler I wanted to write in mind and I wanted to write the compiler own my own time. I was surprised at how accepting my group was to take on my idea for writing a compiler that compiled to HTML, CSS, and JavaScript, but I was even more surprised at how fast my simple idea of writing small compiler that created a one page web application turned into a monstrosity that was supposed to handle practically every edge case. I was a little overwhelmed by the language designs that my group came up with, but at the same time, I was happy that others were able to build off of and improve upon my ideas.

I anticipated that it would be a challenge to meet up on a regular basis, since I had a really busy schedule. Looking back at the semester, I wish that we had actually kept to the meetings that we scheduled at the beginning of the year. At the beginning of the year, I thought we had a lot of time and entertained the idea that we could complete the entire project during reading week. By reading week, I regretted not having completed a majority of the compiler ahead of time.

I also struggled with the steep learning curve for OCaml. I was really confused by all the different formats and the different ways of compiling various OCaml files. I didn't really have a strong grasp on the connection between the scanner, parser, abstract syntax tree, and the main executable. It wasn't until I was halfway done writing compile.ml that I started appreciating OCaml's laconic syntax. I think that I wrote compile.ml in a semi-object-oriented way since I was so used to programming in Java. There were many times during this assignment when I wished that I could have programmed the entire compiler in Java.

My advice for future teams is to start early and really try to learn OCaml's weird quirks before attempting the compiler. I wrote a lot of small programs before I felt comfortable with writing compiler files in OCaml. There aren't a lot of OCaml examples on the internet, so a lot of learning

OCaml was trial-and-error. I think that it’s crucial to be comfortable with OCaml first before writing the compiler.

7.2 Aftab Khan

During the development process of this project, I learned not only a great deal about programming languages, programming techniques, and development methodologies and approaches, but also gained much experience contributing to collaborative, practical, large-scale development projects according to a strict timelines.

From the very outset, this project was an exercise in familiarizing oneself with the unfamiliar. Not only did it require learning a great deal about Computer Science theory, it required concurrently learning an entirely new programming language and it’s associated functional paradigm in an extremely short period of time, which, having had most of my previous programming experience in procedural languages such as C or Java, was extremely unfamiliar to me. As a result, I was forced to quickly develop a working understanding of functional programming, which allowed me to consider methods of problem solving that would previously have been inaccessible or undesirable. Additionally, while OCaml proved to be extremely difficult to learn, it was an effort rewarded by promising and exciting results, adding another area of experience to my professional toolbelt. Finally, as the only group member that programmed portions of the SPWAG compiler not only in OCaml, but in C as well, it was fascinating to me to see how such different languages (in terms of both approach, methodology, and implementation) were able to link cleanly into a single executable. I’d heard of different languages being linked into executables concurrently in large-scale development projects, but having had minimal experience in such implementations, this provided me an opportunity to experiment with linking and the compilation process first-hand.

This project also served as a major learning experience in terms of working collaboratively in teams in order to coplete project goals. Attempting to reconcile our differences and come to an agreement on a fixed set of functionalities and objectives for our language proved to be not only one of the most challenging parts of this project, but was also among the most enjoyable and enriching. Through my many conversations with my teammates, it would always be an interesting to see what new issues or design considerations they had conceived, as they were almost always ones I had either overlooked, or were interpreted in different ways. This strength also proved to be a weakness, however, as excessive time was spent debating minutiae of the project, rather than beginning our implementation process, and letting the optimal design solutions take shape organically in the process.

Ultimately, it is clear to me that if anything would have been useful to facilitating the completion of this project, it would have been having more time in order to work. Because of each individual’s prior commitments, schoolwork, etc, combined with the rapid pace of this course, It was often difficult to coordinate with group members to find enough mutually agreeable meeting times, resulting in many potential lost hours of work. Additionally, it would have been extremely useful to have a prior working knowledge of OCaml before beginning implementing the code. Much time was spent trying to decode and debug the workings of this foreign language, in some cases, preventing measurable progress on the actual project itself from being made.

7.3 Yunhe (John) Wang

It was very interesting to work on a large scale project with a team, and certainly one of the first times I've had such an experience for a class project. Overall, working in a team is fun, as outside the classroom setting, communication, cooperation, and working off of each other's strengths is crucial. After all, the most awe-inspiring things are mostly completed by the work of many.

That said, there were things we've done correctly, other things we need to improve upon. I felt we did a good job with matching people to what they were most familiar with. For example, Lauren was better on the front end (ie compiler) since she was more intimately familiar with html/css/js than the rest of us, Richard had more experience debugging and so worked on the semantic analyzer, I am the most detail oriented, and so ended up writing the bulk of irgenerator etc.

On the other hand, I felt like we could have budgeted our time on this project better. I remember our first few meetings spent on arguing about what to include and what's more user friendly, and what's more stylish (There's a reason SPWAG is swag with a P). But, in hindsight, we should have asked ourselves more often, "Is implementing this feature as such going to work?" We had to make so many changes to our design decisions while implementing because we realized that those just won't work. That certainly harmed the remaining time we had for testing.

Overall, if anything, this was worth it for the experience and the lessons learned.

7.4 Richard Chiou

As the leader of the project, I gained invaluable experience from our SPWAG project. First, I added a new programming language (OCAML, to a certain extent SPWAG) to my repertoir of languages. I remember from the first homework assignment being absolutely confused by the novelties of functional programming, but as the semester went on and the time I spent on the project increased, it all started to make sense. Despite its very vague syntax error notification, OCAML is really good with its strong type checking. Moreover, hours of testing and debugging OCAML programs has made me really familiar with OCAML syntax. Personally, I feel that's the only way for newcomers to functional programming to learn: through trial and experience.

Aside from functional programming, I finally got the experience of working with a group on a large-scale project. I personally thought we did a good job of having regular meetings (at least three hours once a week, sometimes even twice) during much of the semester, but I will attest that we could have been much more efficient. As one can imagine, there was a lot of arguments over design philosophies and issues, which is fine, but it needs to be supplemented with written code. Without actually writing the files, there's no real way to come to final conclusions about how our AST should be structured, and at some point, design decisions need to be fixed. Unfortunately, we recognized some potential pitalls far too late, which made some parts of the compiler much longer and more complex than it needed to be. If I could repeat the process, I would initiate the development phase much more quickly, and have everyone start on individual assignments much sooner instead of having the entire group decide on every facet of SPWAG. At this level, I think it's fine for one person to exercise judgment on his or her specific part of the code.

That being said, the team was clearly enthusiastic about the project, and that's super important. If you truly enjoy what you work on, you will be motivated, and you will feel absolutely great when you finish. How many other groups have a language that can be used for both teaching and playing games? Even better, how many people coded their final presentation in their actual programming language, and had it work? For our group, both answers are yes.

7.5 Aditya Majumdar

First, some of my thoughts regarding team communication and organization - for the first two months, weekly meetings are an absolute must. The majority of our group optimistically believed that 2 or 3 straight weeks of solid work at the end of the semester would be enough to pull off this project, but that is absolutely the worst mindset to take toward this project, and this nonchalant attitude prevented us from being able to complete the project because we just could not meet early enough to make decisions. When you're still trying to nail down the details of your language and write the Scanner/Parser/AST, all of the group members need to be available - without everyone present, the team will be unable to effectively communicate their ideas and finalize those important decisions. I'm not sure how easily we could've avoided this because we literally had about 1 hour a week on average when all 5 of us could meet, and we failed to come to decisions within those time periods. Our downfall was that we were still changing details in the last week or two, meaning splitting up the code was a challenge, and this just made us extremely inefficient overall.

On the bright side, this was a good project to learn a new programming paradigm, begin to uncover the wonders of OCaml, and also get a good intro to the types of decisions you have to consider and make when designing something as large and complicated as a programming language. I highly recommend that future students force their teams to meet as often as possible, as early as possible during the semester no matter how much it seems like you can put this off until later - the truth is, you just can't. Read up on OCaml tutorials as much as possible, because you will see some crazy errors and constructs that you won't understand for a while, but as you see them over and over again, will lead you to similar solutions in most cases. Furthermore, if you've never tried pair programming before, this is a good project to try it with (if you have enough team members, that is). Working with someone else on the same piece of code at the same time is great, especially in OCaml, because both people have approximately the same familiarity with the language, so you can both contribute your ideas of how to solve a specific problem, as well as catch small errors and bugs as you work together.

8 Appendix

- Attach a complete code listing of your translator with each module signed by its author
- Do *not* include any ANTLR-generated files, only the .g sources.

8.1 Main SPWAG Language Compiler Files

8.1.1 preprocessor.c

```
/* Author: Aftab Khan */
```

```
/*
 * Reads SPWAG source files (without curly braces "{}") from Standard In
 * and inserts curly braces as appropriate. Preprocessed SPWAG code is
 * returned to calling OCaml function as a String.
 */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <regex.h>
#include <sys/types.h>
#include <caml/alloc.h>
//#include <caml/memory.h>
#include <caml/mlvalues.h>

#define BUF_SIZE 1024

 void addNTabs(int n, char *destination){
        int i;
        for (i = 0; i < n; ++i){
                strcat(destination, " ");
        }
 }


 int isSpecialSyntax(char *lineString, int tabLength){
        regex_t regex;
        int result;
        //result = regcomp(&regex, "(define )|((if|while)[' ']*\()", REG_EXTENDED);
        //result = regcomp(&regex, "(define )|(if|while) |(if|while)\(", REG_EXTENDED);
        result = regcomp(&regex, "^(       | )*((define )|(if |while )|(if\\(|while\\())", REG_EXTENDED);
                if (result) {fprintf(stderr, "Could not compile regex\n"); exit(1);}

        char *trimmedString = lineString; //+ tabLength;
        //printf("trimmedstring: %s", trimmedString);

        result = regexec(&regex, trimmedString, 0, NULL, 0);
        //if ((NULL != strchr(trimmedString, '(')) && result){
        if (result && (strlen(trimmedString) > 1)){
                //printf("unusual box instance syntax\n");
                return 1;
/*      } else if (result == REG_NOMATCH){
     return 0; */
        } else {
     //fprintf(stderr, "Regex match failed");
                return 0;
     //exit(1);
	}
}


// corresponds to OCaml function type: unit -> String
CAMLprim value caml_preprocess(value unit)
{
	char buffer[BUF_SIZE];
	char lineBuffer[BUF_SIZE];

	// Valid Tab Characters: Tab (\t), Space
	char *tabCharSet = "\t ";

	size_t contentSize = 1; // includes NULL

	char *content = malloc(sizeof(char) * BUF_SIZE);
	if(content == NULL){
		perror("Failed to allocate content");
		exit(1);
	}

	content[0] = '\0'; // make null-terminated

	int oldTabLength = -1;
	int tabLength = 0;
	int specialSyntaxFlag = 0;

	while(fgets(lineBuffer, BUF_SIZE, stdin)){

		buffer[0] = '\0';

		oldTabLength = tabLength;
		tabLength = strspn(lineBuffer, tabCharSet);

		if(tabLength > oldTabLength){
			if (specialSyntaxFlag){
				// Special component instance syntax
				content[strlen(content) - 1] = '\0';
				strcat(buffer, "{\n");
				specialSyntaxFlag = 0;
			} else {
				addNTabs(oldTabLength, buffer);
				strcat(buffer, "{\n");
			}
		} else if (tabLength < oldTabLength){
			addNTabs(tabLength, buffer);
			strcat(buffer, "}\n");
		}
		strcat(buffer, lineBuffer);

		// updates special syntax flag
		specialSyntaxFlag = isSpecialSyntax(lineBuffer, tabLength);

		char *old = content;
		contentSize += strlen(buffer);
		content = realloc(content, contentSize);
		if(content == NULL){
			perror("Failed to reallocate content");
			free(old);
			exit(2);
		}
		strcat(content, buffer);

	}

	// Prints the final curly brace, if the file contained any lines
	if (oldTabLength > -1) strcpy(buffer, "\n}\n\0");
	char *old = content;
	contentSize += strlen(buffer);
	content = realloc(content, contentSize);
	if(content == NULL){
		perror("Failed to reallocate content");
		free(old);
		exit(2);
	}
	strcat(content, buffer);

	if(ferror(stdin)){
		free(content);
		perror("Error reading from stdin");
		exit(3);
	}

	//printf("Result\n\n%s\n", content);
	return caml_copy_string(content);
}
```

8.1.2 Scanner.mll

```
(* Authors: Lauren Zou, Aftab Khan, Yunhe(John) Wang *)

{
  open Parser
  open Linecounter
}

(* Add interpretation for string literals *)
```

```
rule token =
    parse [' ' '\r'  '\t' ]                    { token lexbuf }
    | "##"                                     { multi_line_comment lexbuf }
    | "#"[^'#']([^'\n']*)"\n"                  { incr Linecounter.linecount; NEWLINE }
    | '+'                                      { PLUS }
    | '-'                                      { MINUS }
    | '*'                                      { TIMES }
    | '/'                                      { DIVIDE }
    | '='                                      { ASSIGN }
    | "=="                                     { EQUALS }
    | "!="                                     { NOTEQUALS }
    | '<'                                      { LESSTHAN }
    | '>'                                      { GREATERTHAN }
    | '!'                                      { NOT }
    | "||"                                     { OR }
    | "&&"                                     { AND }
    | '('                                      { LPAREN }
    | ')'                                      { RPAREN }
    | '{'                                      { LBRACE }
    | '}'                                      { RBRACE }
    | '['                                      { LBRACK }
    | ']'                                      { RBRACK }
    | ','                                      { COMMA }
    | '\n'                                     { incr Linecounter.linecount; NEWLINE }
    | "attr"                                   { ATTR }
    | "comp"                                   { COMP }
    | "func"                                   { FUNC }
    | "define"                                 { DEF }
    | "else"                                   { ELSE }
    | "end"                                    { END }
    | "false"                                  { FALSE }
    | "if"                                     { IF }
    | "import"                                 { IMPORT }
    | "isa"                                    { ISA }
    | "null"                                   { NULL }
    | "return"                                 { RETURN }
    | "slide"                                  { SLIDE }
    | "true"                                   { TRUE }
    | "var"                                    { VAR }
    | "while"                                  { WHILE }
    | ['a'-'z']['a'-'z' '0'-'9' '-']*  as idstr  { ID(idstr) }
    | '"'[^ '"']*'"' as str                    { STRING(String.sub str 1 ((String.length str)-2)) }
    | ['0'-'9']+ as lit                        { LITERAL(int_of_string lit) }
    | ['0'-'9']+'%' as lit                     { PERCENT(int_of_string (String.sub lit 0 ((String.length lit)-1))) }
    | eof                                      { EOF }
    | _ as char                                { raise (Failure("illegal character " ^ Char.escaped char)) }
and multi_line_comment = parse
```

```
   "##" { token lexbuf } (* End-of-comment *)
   | _ { multi_line_comment lexbuf } (* Eat everything else *)
```

8.1.3 parser.mly

```
/* Yunhe (John) Wang, Lauren Zou, Aftab Khan */

%{
   open Ast
   open Linecounter
   let parse_error msg = Printf.eprintf "%s at around line %d \n" msg !linecount
%}

%token LPAREN RPAREN PLUS MINUS TIMES DIVIDE ASSIGN EOF EQUALS NOTEQUALS
LESSTHAN GREATERTHAN NOT OR AND COMMA NEWLINE
%token ATTR COMP FUNC DEF ELSE END IF IMPORT ISA NULL RETURN SLIDE VAR WHILE LBRACE
RBRACE LBRACK RBRACK
%token <int> LITERAL PERCENT
%token <string> ID STRING
%token TRUE FALSE

%nonassoc NOELSE
%nonassoc ELSE
%left COMMA
%right ASSIGN
%left OR
%left AND
%left EQUALS NOTEQUALS
%left LESSTHAN GREATERTHAN
%left PLUS MINUS
%left TIMES DIVIDE
%right NOT
%left LBRACK RBRACK
%left LPAREN RPAREN

%start program
%type <Ast.program> program

%%

program: /* global vars, functions */
      /* nothing */           { [], [] }
   | program NEWLINE         { $1 }
   | program VAR ID NEWLINE   {(Identifier($3) :: fst $1), snd $1 }
   | program func_decl       { fst $1, ($2 :: snd $1) }

func_decl:
      DEF SLIDE ID LPAREN RPAREN NEWLINE LBRACE stmt_list RBRACE NEWLINE
      {{
        t = Slide;
        name = Identifier($3);
        formals = [];
        inheritance = None;
        paractuals = [];
        body = List.rev $8
      }}
    | DEF COMP ID LPAREN formals_opt RPAREN ISA ID LPAREN actuals_opt RPAREN
      NEWLINE LBRACE stmt_list RBRACE NEWLINE
      {{
        t = Comp;
        name = Identifier($3);
        formals = $5;
        inheritance = Some(Identifier($8));
        paractuals = $10;
        body = List.rev $14
      }}
    | DEF ATTR ID LPAREN formals_opt RPAREN NEWLINE LBRACE stmt_list RBRACE NEWLINE
      {{
        t = Attr;
        name = Identifier($3);
        formals = $5;
        inheritance = None;
        paractuals = [];
        body = List.rev $9
      }}
    | DEF FUNC ID LPAREN formals_opt RPAREN NEWLINE LBRACE stmt_list RBRACE NEWLINE
      {{
        t = Func;
        name = Identifier($3);
        formals = $5;
        inheritance = None;
        paractuals = [];
        body = List.rev $9
      }}

formals_opt:
      /* nothing */              { [] }
    | formal_list                { List.rev $1 }

formal_list:
      ID                         { [Identifier($1)] }
    | formal_list COMMA ID          { Identifier($3) :: $1 }

actuals_opt:
      /* nothing */              { [] }
    | actuals_list               { List.rev $1 }
```

```
actuals_list:
    expr                        { [$1] }
  | actuals_list COMMA expr      { $3 :: $1 }

ids_list:
    LBRACK expr RBRACK           { [$2] }
  | ids_list LBRACK expr RBRACK   { $3 :: $1 }

mods_opt:
    /* nothing */                { Block([]) }
  | LBRACE stmt_list RBRACE       { Block(List.rev $2) }

stmt_list:
    NEWLINE                      { [] }
  | stmt_list NEWLINE            { $1 }
  | stmt_list stmt               { $2 :: $1 }

stmt:
    expr NEWLINE                                { Expr($1) }
  | RETURN expr NEWLINE                         { Return($2) }
  | LBRACE stmt_list RBRACE NEWLINE             { Block(List.rev $2) }
  | IF expr NEWLINE stmt %prec NOELSE           { If($2, $4, Block([])) }
  | IF expr NEWLINE stmt ELSE NEWLINE stmt       { If($2, $4, $7) }
  | WHILE expr NEWLINE stmt                     { While($2, $4) }
  | VAR ID NEWLINE                              { Declaration(Identifier($2))}
  | VAR ID ASSIGN expr NEWLINE                  { Decassign(Identifier($2), $4) }

expr:
    NOT expr                     { Notop($2) }
  | expr PLUS expr        { Binop($1, Plus, $3) }
  | expr MINUS expr       { Binop($1, Minus, $3) }
  | expr TIMES expr       { Binop($1, Times, $3) }
  | expr DIVIDE expr      { Binop($1, Divide, $3) }
  | expr EQUALS expr      { Binop($1, Equals, $3) }
  | expr NOTEQUALS expr    { Binop($1, Notequals, $3) }
  | expr LESSTHAN expr     { Binop($1, Lessthan, $3) }
  | expr GREATERTHAN expr  { Binop($1, Greaterthan, $3) }
  | expr OR expr          { Binop($1, Or, $3) }
  | expr AND expr         { Binop($1, And, $3) }
  | ID ASSIGN expr        { Assign(Identifier($1), $3) }
  | LITERAL               { Litint($1) }
  | STRING                { Litstr($1) }
  | PERCENT               { Litper($1) }
  | ID                 { Variable(Identifier($1)) }
  | TRUE                  { Litbool(true) }
  | FALSE                 { Litbool(false) }
  | NULL                  { Litnull }
  | ID ids_list           { Component(Identifier($1), List.rev $2) }
  | ID LPAREN actuals_opt RPAREN mods_opt
    {Call({
      cname = Identifier($1);
      actuals = $3;
      mods = $5;
    })}
  | LPAREN expr RPAREN   { $2 }
```

8.1.4 Ast.ml

```
(* Authors: Richard Chiou, Yunhe (John) Wang, and Aditya Majumdar *)

(* Identifiers *)
type identifier = Identifier of string

(* Operators *)
type operator = Plus | Minus | Times | Divide | Equals | Notequals | Lessthan | Greaterthan | Or | And

(* Function types *)
type func_type = Slide | Comp | Attr | Func

(* Handles calls of functions and components *)
type func_call = {
  cname : identifier; (* Name of the function *)
  actuals : expr list; (* Evaluated actual parameters *)
  mods : stmt; (* Additional statements, which could be a block *)
}

(* Expressions *)
and expr =
    Binop of expr * operator * expr (* a + b *)
  | Notop of expr (* !a only applies to booleans *)
  | Litint of int (* 42 *)
  | Litper of int (* 42% *)
  | Litstr of string (* “foo” *)
  | Litbool of bool (* true *)
        | Litnull (* null *)
  | Assign of identifier * expr (* foo - 42 *)
  | Variable of identifier (* although this is named Variable, can also be the name of a slide/function *)
  | Component of identifier * expr list (* identifier["child"]["child"] etc. to fetch component *)
  | Call of func_call (* Calling a function, unique in that it can contain statements*)

(* Calls and executes function. Follows a control flow detailed in the LRM/Revisions doc *)
and stmt = (* Statements ; WIP *)
    Block of stmt list (* { ... } *)
  | Expr of expr (* foo = bar + 3; *)
  | Return of expr (* return 42; *)
  | If of expr * stmt * stmt (* if (foo == 42) stmt1 else stmt2 end *)
  | While of expr * stmt (* while (i<10) \n  i = i + 1 \n end \n *)
```

```
    | Declaration of identifier (* Declaring a variable *)
    | Decassign of identifier * expr (* Declaring a variable and then assigning it something*)

(* Function definition that makes up the basic structure of the program*)
type func_definition = { (* Handles declarations of functions, components, attributes, slides *)
    t: func_type; (* e.g. slide, component, attribute, func *)
    name : identifier; (* Name of the function *)
    formals : identifier list; (* Name of the formal parameters *)
    inheritance : identifier option; (* Name of any parent components, ie box, or null *)
    paractuals: expr list; (* This represents the actuals passed to the parent *)
    body : stmt list; (* Conditional, Return Statements, Function Declarations/Calls, etc. *)
}

(* The program itself *)
type program = identifier list * func_definition list (* global vars, funcs*)

(* The following are needed to output the ast for testing *)
let string_of_identifier = function
    Identifier(s) -> s

let rec string_of_call call =
    (string_of_identifier call.cname) ^ "(" ^
    String.concat ", " (List.map string_of_expr call.actuals) ^ ")"
    ^ (match call.mods with
    Block([]) -> ""
    | Block(stmts) -> "{\n" ^ String.concat "" (List.map string_of_stmt stmts) ^ "}"
    | something -> "ERROR!!!!!!" )
and string_of_expr = function
      Notop(e) -> "!(" ^ (string_of_expr e) ^ ")"
    | Binop(e1, o, e2) ->
        "(" ^ string_of_expr e1 ^ " " ^
        (match o with
      Plus -> "+" | Minus -> "-" | Times -> "*" | Divide -> "/"
        | Equals -> "==" | Notequals -> "!="
        | Lessthan -> "<" | Greaterthan -> ">"
        | Or -> "||" | And -> "&&" ) ^ " " ^
        string_of_expr e2 ^ ")"
    | Litint(l) -> string_of_int l
    | Litper(l) -> (string_of_int l) ^ "%"
    | Litstr(s) -> "\"" ^ s ^ "\""
    | Litbool(b) -> if b then "true" else "false"
    | Litnull -> "null"
            | Assign(v, e) -> (string_of_identifier v) ^ " = " ^ string_of_expr e
    | Variable(v) -> (string_of_identifier v)
    | Component(v, e) ->  (string_of_identifier v) ^
        String.concat "" (List.map (fun ex -> "[" ^ (string_of_expr ex) ^ "]") e)
    | Call(f) -> string_of_call f
and string_of_stmt = function
```

```
     Block(stmts) -> "{\n" ^ String.concat "" (List.map string_of_stmt stmts) ^ "}\n"
   | Expr(expr) -> string_of_expr expr ^ "\n";
   | Return(expr) -> "return " ^ string_of_expr expr ^ "\n";
   | If(e, s, Block([])) -> "if " ^ string_of_expr e ^ "\n" ^ string_of_stmt s
   | If(e, s1, s2) ->  "if " ^ string_of_expr e ^ "\n" ^
                   string_of_stmt s1 ^ "else\n" ^ string_of_stmt s2
   | While(e, s) -> "while " ^ string_of_expr e ^ "\n " ^ string_of_stmt s
   | Declaration(i) -> "var " ^ (string_of_identifier i) ^ "\n"
   | Decassign(i, e) -> "var " ^ (string_of_identifier i) ^ "= " ^ (string_of_expr e) ^ "\n"

let string_of_vdecl = function
   Identifier(s) -> "var " ^ s ^ "\n"

let string_of_function_type = function
     Slide -> "define slide"
   | Comp -> "define comp"
   | Attr -> "define attr"
   | Func -> "define func"

let string_of_inheritance i p =
   " isa " ^ (string_of_identifier (match i with Some(id) -> id | None -> Identifier("ERROR!!"))) ^
   "(" ^ String.concat ", " (List.map string_of_expr p) ^ ")"

let string_of_fdecl fdecl =
   (string_of_function_type fdecl.t) ^ " " ^
   (string_of_identifier fdecl.name) ^
   "(" ^ String.concat ", " (List.map string_of_identifier fdecl.formals) ^ ")" ^
   (match fdecl.t with
      Comp -> (string_of_inheritance fdecl.inheritance fdecl.paractuals)
      | _ -> ""
   ) ^
   "\n{\n" ^
   String.concat "" (List.map string_of_stmt fdecl.body) ^
   "}\n"

let string_of_program (vars, funcs) =
   String.concat "" (List.map string_of_vdecl vars) ^ "\n" ^
   String.concat "\n" (List.map string_of_fdecl funcs)
```

8.1.5 semantic_analyzer.ml

```
(*  Author: Richard Chiou

   Contributor: Aditya Majumdar

*)


open Ast
```

```
module StringMap = Map.Make(String)


type symbol_table = {
   parent : symbol_table option;
   functions : Ast.func_definition list;
   variables : Sast.identifier list;
}


type translation_environment = {
   scope : symbol_table;
}


(* We need the types from SAST because of an oversight *)
type t = Int | Per | Str | Bool | Slidetype | Comptype | Attrtype | Functype | Varidentifier | Null


(* See if t1 and t2 have the same types *)
let types_equal t1 t2 = match t1,t2 with _, _ -> if (t1 = t2)
   then true
                  else false


let identifier_of_string = function
   s -> Identifier(s)


let string_of_identifier = function
   Identifier(s) -> s


let string_of_expr = function
   s -> Litstr(s)
```

```
let string_of_type_t = function
      Sast.Int -> "Int"
    | Sast.Bool -> "Bool"
    | Sast.Str -> "Str"
    | Sast.Per -> "Per"
              | Sast.Slidetype -> "Slidetype"
              | Sast.Comptype -> "Comptype"
              | Sast.Attrtype -> "Attrtype"
              | Sast.Functype -> "Functype"
              | Sast.Varidentifier -> "Varidentifier"
              | Sast.Null -> ""

let string_of_func_type = function
                Slide -> "Slide"
              | Comp -> "Comp"
              | Attr -> "Attr"
              | Func -> "Func"

let type_to_t = function
                Slide -> Sast.Slidetype
              | Comp -> Sast.Comptype
              | Attr -> Sast.Attrtype
              | Func -> Sast.Functype

(* Convert AST identifier to SAST identifier *)
let identify env = function
    Ast.Identifier(v) -> Sast.Identifier(v)
    (*let vdecl = find_variable env.scope Ast.Identifier in (* Locate a variable by name *)
              Sast.Identifier(vdecl), Sast.Varidentifier*)
```

```
let string_of_expr = function
          | _ -> "(not implemented ... yet)"

(* Operations: Plus | Minus | Times | Divide | Equals | Notequals | Lessthan | Greaterthan | Or | And *)
let string_of_binop = function
           Plus -> "Plus"
          | Minus -> "Minus"
          | Times -> "Times"
          | Divide -> "Divide"
          | Equals -> "Equals"
          | Notequals -> "Notequals"
          | Lessthan -> "Lessthan"
          | Greaterthan -> "Greaterthan"
          | And -> "And"
          | Or -> "Or"

(* This find_variable function is adapted from the slides *)
let rec find_variable scope name =
   try
                    List.find (fun (s) -> s = name) scope.variables
   with Not_found ->
      match scope.parent with
      Some(parent) -> find_variable parent name
   | _ -> raise Not_found

(* This find_function function goes up to global scope before searching *)
let rec find_function scope name = (* name is an identifier *)
          let rec global scope = match scope.parent with     (* All functions are global *)
```

```
                    | None -> scope
                    | Some(parent) -> (global parent)
          in
          try
                    List.find (fun {t=_; name=s; formals=_; inheritance=_; paractuals=_; body=_} -> s = name)
(global scope).functions
          with Not_found -> (* Not found, print error message *)
                    (*let build_string tmpString nextString = tmpString^" \n"^nextString in
                    let func_names_string = List.fold_left build_string("") (List.map (fun {t=_, name=s,
formals=_, inheritance=_, paractuals=_, body=_} -> n ) (getGlobalScope scope).functions) in
                    let num_funcs = List.length (getGlobalScope scope).functions in*)
                    raise(Failure("Function not found in global scope"))


(*  Evaluate func call: Evaluate identifier to be valid (not slide), evaluate actuals are valid expressions,
evaluate mods are statements  *)


(* check to see if valid function call *)
let functioncall env call =
          (*let actuallist = List.map (expr env) call.actuals
          and id = identify env call.cname in*)
          let funct = find_function env call.cname in


          (*let parentscope = {
                    parent = env.parent;
                    functions = env.functions;
                    variables = env.variables;
   } in*)


          let checked_func_call = {
                    cname = funct.name;
```

```
                        actuals = call.actuals;
                        mods = call.mods;
      } in
            checked_func_call

(* Convert function definitions from Ast to Sast *)
(*let func_def_converstion env (fdef:Ast.func_definition) =
                        {Sast.t=fdef.t;
                        name= identify env fdef.name;
                        formals = List.rev (List.fold_left (fun l (Ast.Identifier(s)) -> Sast.Identifier(s)::l ) []
fdef.formals);
                        inheritance = (match fdef.inheritance with None -> None | Some(Ast.Identifier(s)) ->
Some(Sast.Identifier(s)));
                        paractuals = List.rev (List.fold_left (fun l e -> (expr env e)::l ) [] fdef.paractuals);
                        body = List.rev (List.fold_left (fun l s -> (convert_stmt s)::l ) [] fdef.body);}*)

(* Check if valid expression*)
let rec expr env = function

    (* Simple evaluation of primitives *)
    Ast.Litint(v) -> Sast.Litint(v), Sast.Int
  | Ast.Litper(v) -> Sast.Litper(v), Sast.Per
  | Ast.Litstr(v) -> Sast.Litstr(v), Sast.Str
  | Ast.Litbool(v) -> Sast.Litbool(v), Sast.Bool
  | Ast.Litnull -> Sast.Litnull, Sast.Null

  | Ast.Binop (expr1, op, expr2) ->  (* evaluate operators *)
    (
            let e1 = expr env expr1 and
                        e2 = expr env expr2 in
```

```
let _, t1 = e1 (* Get the type of each child *)
and _, t2 = e2 in

     match t1, op, t2 with
      | (Sast.Bool), (And | Or), (Sast.Bool) ->  (* And/or operators *)
             Sast.Binop(e1, op, e2), Sast.Bool (* Boolean *)

 | (Sast.Int), (Lessthan | Greaterthan), (Sast.Int) ->  (* > , < *)
                    Sast.Binop(e1, op, e2), Sast.Bool

      | (Sast.Int), (Plus | Minus | Times | Divide), (Sast.Int) ->  (* Arithmetic on ints *)
                    Sast.Binop(e1, op, e2), Sast.Int

      | (Sast.Per), (Plus | Minus | Times | Divide), (Sast.Per) ->  (* Arithmetic on percents *)
                    Sast.Binop(e1, op, e2), Sast.Per

 | _, (Equals | Notequals), _  ->   (* Compare Anything *)
                    Sast.Binop(e1, op, e2), Sast.Bool

      | (Sast.Str), Plus, (Sast.Str | Sast.Int | Sast.Bool) ->  (* String Concatenation *)
                    Sast.Binop(e1, op, e2), Sast.Str

     (* Otherwise Invalid *)
      | a, op, b -> raise(Failure("Binop "^ (string_of_binop op) ^" has invalid operands "))
     )

| Ast.Notop(v) -> (* check if negate = ! and e1 is a boolean *)
 (
            let e1 = expr env v in
            let _, t1 = e1 in (* Get the type of e1 *)
            match t1 with
              | Sast.Bool -> Sast.Notop(e1), Sast.Bool
              | _ -> raise(Failure("Not operator requires bool operand"))
            )

  | Ast.Assign(lhs, rhs) ->
    let e1 = expr env rhs                         (* check if valid expression *)
            and id = identify env lhs in          (* check if identifier *)
    (* let _, t1 = e1 (* type of rhs *) in *)
            (*if (types_equal t1 t2) then *)                              (* the types need to match? *)
                    Sast.Assign(id, e1), Sast.Varidentifier
            (*else
                    raise(Failure(string_of_type_t t1^" expression does not match identifier "^string_of_type_t
t2))*)

  | Ast.Variable(v) ->                                     (* If identify function works, this will work *)
            let id = identify env v in
            (*let _, t1 = id (* type of rhs *) in*)
            Sast.Variable(id), Sast.Varidentifier

            (* let string_of_identifier = function
    Identifier(s) -> s *)
  | Ast.Call(funccall: Ast.func_call) -> (
                    let fc = functioncall env funccall in                                     (* Evaluate if this is a
valid func_call *)
                    let funct = find_function env funccall.cname in                   (* Check to see if said function
exists *)
                    (* We need to now check that the arguments are valid *)
                    (* Do we even need to get types from identifiers somehow? *)
                    let formallist = List.map (identify env) funct.formals in
                    (*let formalstoidentifiers = List.map (string_of_identifier) formallist in
                    let identifierstoexprs = List.map (Litstr)*)
                    (*let formalTypes = List.map fst(formallist) in*)
                    (* Get the list of actuals and their types *)
                    let actualstostrings = List.map (string_of_expr) funccall.actuals in
                    let stringstoids = List.map (identifier_of_string) actualstostrings in
                    let actuallist = List.map (identify env) stringstoids in
                    (*let actualtypes = List.map fst(actuallist) in*)


                    (* compare the formals and actuals *)
                    let rec checktypes list1 list2 = match list1, list2 with
                    | [], [] -> true
                    | [], _ -> raise(Failure(" there should be no parameters "))
                    | _ , [] -> raise(Failure(" there are missing parameters "))
                    | _ , _ -> try
                      ( types_equal (List.hd(list1)) (List.hd(list2)) ) && types_equal (List.tl(list1)) (List.tl(list2))
                            with Failure("hd") -> raise(Failure(" mismatched types "))
                    in
                    if (checktypes formallist actuallist) then
                    let rec convert_stmt = function
                            Ast.Block(stmts) -> Sast.Block(List.rev (List.fold_left (fun l s -> (convert_stmt s)::l )
[] stmts))
                      | Ast.Expr(e) -> Sast.Expr(expr env e)
                      | Ast.Return(e) -> Sast.Return(expr env e)
                      | Ast.If(e,s1,s2) -> Sast.If(expr env e, convert_stmt s1, convert_stmt s2)
              | Ast.While(e,s) -> Sast.While(expr env e, convert_stmt s)
              | Ast.Declaration(Identifier(s)) -> Sast.Declaration(Sast.Identifier(s))
              | Ast.Decassign(Identifier(s), e) -> Sast.Decassign(Sast.Identifier(s), expr env e)
```

```
                    in
                    let func_call_conversion env (fc: Ast.func_call) =
                      {Sast.cname = identify env fc.cname;
                              Sast.actuals = List.rev(List.fold_left (fun l e -> (expr env e)::l ) [] fc.actuals);
                              Sast.mods = convert_stmt fc.mods}
                    in (Sast.Call (func_call_conversion env funccall)), type_to_t funct.t
        else
                    raise(Failure("Arguments for function do not match those given in the definition"))
              )


  (* Component of identifier: identifier has to be slide or variable (component or slide) *)
  | Ast.Component (v, exprlist) ->


              let checkedexprs = List.map (expr env) exprlist in
              let id = identify env v in                                  (* How are we going to get the
type of the identifier? *)
              let funct = find_function env v in
              let typ = funct.t in
              let strings = List.map(string_of_expr) exprlist in
              let ids = List.map(identifier_of_string) strings in


              if ((types_equal typ Comp) || (types_equal typ Slide)) then
                    (* We need to do recursion. Here's the general idea:
                    Base step: currentobject = id
                    Step 1: currentobject = id[exprlist[0]]
                    Step 2: remove exprlist[0]
                    Step 3: Go to step 1, break from loop when exprlist has been parsed through *)

                    let rec compfind v exprs comptype = match exprs with
               | [] -> Sast.Component(id, checkedexprs), Sast.Comptype
```

```
                    | x ->
                                    let innercomp = List.hd exprs
                                    and newlist = List.tl exprs in
                                    let compfunct = find_function env innercomp in
                                    let comptyp = compfunct.t in
                                    if ((types_equal comptyp Comp) || (types_equal comptyp Slide)) then
                                          compfind innercomp newlist comptyp
                                    else
                                          raise(Failure("Can only take component of slide or other
components"))
                                    in compfind v ids typ


                    else
                          raise(Failure("Can only take component of slide or other components"))


  (*Below code is old


  | Ast.Assign(id, e1) -> (* General idea is to make sure the arguments are valid; code may not work though
*)
    let vdecl = try
      find_variable env.scope id
    with Not_found ->
      raise (Error("undeclared identifier " ^ id))
    in
    let expreval = try
      find_expr env.scope
      find_variable env.scope e1
    with Not_found ->
      raise (Error("undeclared expression " ^ e1))
    in
```

```
   let (_, id_type) = vdecl in (* get the variable's type *)

   let (_, expr_type ) = expreval in

   Sast.Assign(id, e1), id_type, expr_type

*)


(* Fold over an intermediate scope/statement list tuple. *)

let rec stmts (env, stmtlist) stmt =

         let newscope = {

           parent =  Some env;

           functions = [];

           variables = [];

         } in

         match stmt with


         | Ast.Expr(e) -> env, Sast.Expr(expr env e)::stmtlist

   (*| Ast.Return(e1) -> Sast.Return(expr env e1) (* I have not written checking for e1 yet *)*)

   | Ast.Block(b) -> env, Sast.Block(snd(List.fold_left stmts(newscope,[]) b))::stmtlist

         | Ast.Return(e) ->                    (* Only funcs can call return, so no error-checking should be fine *)

                  let e1 = expr env e in

                  (*let _, t1 = e1 in

                  if (typeEq t1 returntype )

                    then*) env, Sast.Return(e1)::stmtlist

                  (*else raise(Failure("Return type of function body doesn't match: found"^(string_of_datatype
t1)^" but expected "^(string_of_datatype returntype)))*)


(*       | Ast.Block(s1) ->  (* This block code is modified from Edwards' slides and does not work *)

                  let newscope = { S.parent = Some(env.scope); S.variables = []; S.functions = [] } in

       (* New environment: same, but with new symbol tables *)

         let env' = { env with scope = newscope } in

         let s1 = List.map (fun s -> stmt newenv s) s1 in
```

```
        newscope.S.variables <- List.rev newscope.S.variables; (* side-effect *)
        Sast.Block(newscope, s1)
    | Ast.If(e, s1, s2) ->  (
      let e1 = expr env e in
                    let _, t1 = e1 in   (* Get the type of e1 *)
                    if (types_equal t1 Sast.Bool) then
                            let newscope, thenblock = (stmts(
                            Sast.If (e1, stmt env s1, stmt env s2) (* Check then, else *)
                    else
                            raise(Failure(string_of_type_t t1^"type must be bool"))
          )*)

    | Ast.If(e, s1, s2) ->
                    let e1 = expr env e in
                    let _, t1 = e1 in
                    if (types_equal t1 Sast.Bool) then
                            let nextscope, thenblock = (stmts(newscope, []) s1) in
                            let _, elseblock = (stmts (newscope, []) s2) in
                            try
                                    nextscope, Sast.If( e1, List.hd(thenblock), List.hd(elseblock) )::stmtlist
                            with Failure("hd") -> raise(Failure("If failed"))
                    else raise(Failure("predicate of If should be bool but found "^(string_of_type_t t1)))

(*        | Ast.While(e, s1) ->  (
                    let e1 = expr env e in
                    let _, t1 = e1 in   (* Get the type of e1 *)
                    if (types_equal t1 Sast.Bool) then
                            Sast.While (e1, stmt env s1) (* Check body *)
                    else
                    raise(Failure(string_of_type_t t1^"type must be bool"))
        )
*)
        | Ast.While(e, s) ->
                let e1 = expr env e in
                let _, t1 = e1 in
                if (types_equal t1 Sast.Bool) then
                    let nextscope, loopbody = (stmts(newscope, []) s) in
                    try
                        nextscope, Sast.While(e1, List.hd(loopbody))::stmtlist
                    with Failure("hd") -> raise(Failure("While failed"))
                else raise(Failure("predicate of If should be bool but found "^(string_of_type_t t1)))

        | Ast.Declaration(v) ->
                let id = identify env v in
                if (find_variable env id = id) then (* If declaration exists, don't allow duplicate *)
                    raise(Failure("Variable already exists"))    (* We could use an identifier to string
function *)
                (* else we have to add the variable declaration to the symbol table*)
                else
                    ignore (id::(env.variables));
                    env, Sast.Declaration(id)::stmtlist

        | Ast.Decassign(v, e) ->
                let id = identify env v in
                if (find_variable env id = id) then (* If declaration exists, don't allow duplicate *)
                    raise(Failure("Variable already exists"))    (* We could use an identifier to string
function *)
                else
                    ignore (id::(env.variables));
```

```
                                let e = expr env e in
                                env, Sast.Decassign(id, e)::stmtlist
                        (*let _, t2 = e in
                        if (types_equal t1 t2) then             (* do variable types need to match? *)
                                                                        (* we have to add the variable declaration to the
symbol table; I'll write this later *)
                                Sast.Decassign(v, e), t1 (* Declaring a variable and then assigning it something*)
                        else
                                raise(Failure(string_of_type_t t1^" expression does not match identifier
"^string_of_type_t t2))     *)


(* check to see if func_definition is valid, and return function that is evaluated *)
let check_function env func_definition = match func_definition.body with
        [] -> raise(Failure("Empty functions are invalid")) (* Empty functions not allowed *)
  | x ->
     (*let return_type = func_definition.t in Which type the function returns: this is unused because design
issues *)
     (* Why are we even doing this return stuff *)
                        (* let returnidentifier = {
     t = func_definition.t;
     body = func_definition.body; (* "return" ?*)
     name = func_definition.name;
     formals = func_definition.formals;
     paractuals = func_definition.paractuals;
     inheritance = func_definition.inheritance;
} in *)


        let parentscope = {
                parent = env.parent;
                functions = env.functions;
```

```
                        variables = (List.map (identify env) func_definition.formals)@(env.variables);
    } in
            let checked_func_definition = {
                        t = func_definition.t;
        name = func_definition.name;
        formals = func_definition.formals;
                        inheritance = func_definition.inheritance;   (* how are we going to deal with inheritance? *)
                        (*inheritance = (match func_definition.inheritance with None -> None |
Some(Ast.Identifier(s)) -> Some(Sast.Identifier(s))); *)
                paractuals = func_definition.paractuals;
        body = fst(x, (List.fold_left stmts(parentscope, []) func_definition.body ));
     } in
            checked_func_definition


(* Add a list of func_definitions to the symbol table *)
let add_func_definitions env funcdefs =
            let addfunc scope funcdef =
              { parent = scope.parent;
                        functions = funcdef::scope.functions;
                        variables = scope.variables }
            in
            let global = List.fold_left addfunc(env) funcdefs (* new scope containing all func_definitions *)
            in
            let subscope = {
                        parent =  Some global;
                        functions = [];
                        variables = [];
            }
            in (List.map (check_function subscope) funcdefs), global  (* newFdecls, newScope *)
```

```
(* Add a list of identifiers to the symbol table *)
let add_identifiers env identifiers =
        let addid scope id =
          { parent = scope.parent;
                functions = scope.functions;
                variables = id::scope.variables }
        in
        let global = List.fold_left addid(env) identifiers (* new scope containing all func_definitions *)
        in
        let subscope = {
                parent =  Some global;
                functions = [];
                variables = [];
        }
        in let check_identifier env identifier = (* Check to see if identifier is valid: namely, does it exist in the
symbol table already? *)
        if (types_equal (find_variable env identifier) identifier) then
                raise(Failure("Existing variable declaration"))
        else
                identifier
        in
        let ids = (List.map (check_identifier subscope) identifiers) in
        (List.map (identify subscope) ids), global  (* newFdecls, newScope *)


(* Run program with input: Ast.Program, Symbol_Table and output: Sast.Program *)
(* type program = identifier list * func_definition list (* global vars, funcs*) *)
(* Add the identifiers (variables) and function definitions to global scope *)
let evalprogram program globalTable =
   let identifiers, funcdefs = program in     (*Get the identifier list and the func_definition list*)
        let ids = add_identifiers globalTable identifiers
        and funcs = add_func_definitions globalTable funcdefs in
        let output = ids, funcs in
        output
```

8.1.6 sast.ml

```
(* Author: Richard Chiou, Aditya Majumdar *)

open Ast (* We need to import the operators and other type definitions over, the semantic_analyzer is really
confused otherwise. *)

(* All objects must be one of these types *)

type t = Int | Per | Str | Bool | Slidetype | Comptype | Attrtype | Functype | Varidentifier | Null

(* Identifiers *)
type identifier = Identifier of string

(* Handles calls of functions and components *)
type func_call = {
  cname : identifier; (* Name of the function *)
  actuals : expr list; (* Evaluated actual parameters *)
  mods : stmt; (* Additional statements, which could be a block *)
}

(* Expressions *)
and expr_detail =
  | Binop of expr * operator * expr (* a + b *)
  | Notop of expr (* !a only applies to booleans *)
  | Litint of int (* 42 *)
  | Litper of int (* 42% *)
  | Litstr of string (* “foo” *)
  | Litbool of bool (* true *)
        | Litnull (* null *)
  | Assign of identifier * expr (* foo - 42 *)
  | Variable of identifier (* although this is named Variable, can also be the name of a slide/function *)
  | Component of identifier * expr list (* identifier["child"]["child"] etc. to fetch component *)
  | Call of func_call (* Calling a function, unique in that it can contain statements*)

and expr = expr_detail * t

(* Calls and executes function. Follows a control flow detailed in the LRM/Revisions doc *)
and stmt = (* Statements ; WIP *)
  | Block of stmt list (* { ... } *)
  | Expr of expr (* foo = bar + 3; *)
```

```
    | Return of expr (* return 42; *)
    | If of expr * stmt * stmt (* if (foo == 42) stmt1 else stmt2 end *)
    | While of expr * stmt (* while (i<10) \n  i = i + 1 \n end \n *)
    | Declaration of identifier (* Declaring a variable *)
    | Decassign of identifier * expr (* Declaring a variable and then assigning it something*)

(* Function definition that makes up the basic structure of the program*)
type func_definition = { (* Handles declarations of functions, components, attributes, slides *)
    t: func_type; (* e.g. slide, component, attribute, func *)
    name : identifier; (* Name of the function *)
    formals : identifier list; (* Name of the formal parameters *)
    inheritance : identifier option; (* Name of any parent components, ie box, or null *)
    paractuals: expr list; (* This represents the actuals passed to the parent *)
    body : stmt list; (* Conditional, Return Statements, Function Declarations/Calls, etc. *)
}

(* The program itself *)
type program = identifier list * func_definition list (* global vars, funcs*)
```

8.1.7 irgenerator.ml

```
(* Author: Yunhe (John) Wang Contributor: Aftab Khan *)
(* Reference the Ast only for function types and operators, all else is in sast/ir *)

open Sast
open Ir
open Slide
open Element

(* Lookup table
 * Cur_path starts with the slide's name, followed by the parent elements
 * That means if cur_path has only one string, it's a slide
 * Slides are added immediately to slides_out
 * cur_element stores the current element if it's id is not given
 * If its id is given, cur_element is None, and you must look it up in slides_out *)
type lookup_table = {
    funcs_in                : func_definition StringMap.t; (* Map function names to functions *)
            vars_in                 : literal StringMap.t;        (* Global Variables *)
            slides_out              : Slide.slide StringMap.t; (* Output Map of slides *)
            cur_slide               : string; (* Name of the current slide being evaluated *)
            cur_element             : Element.element option; (* Temporary storage for current element, before
it is bound *)
}

(* John says: I agree, identifier is annoying, oh well*)
let id_to_str = function Identifier(s) -> s

(* Converts funcs (Not attrs/comps/slides) to js definitions *)
let funcs_to_js funcs =
          List.fold_left (fun jss (func:Sast.func_definition) -> match func.t with
                                                  Ast.Func ->
{Ir.name=func.name;formals=func.formals;body=func.body}:: jss
                                                  | _ -> jss
                                                  ) [] funcs

(* Given local_outer and loclook returned by a statement eval, this merges them for a single local variable
map for outer scope
 * Returns the new loclook *)
let merge_locals local_outer (local_inner, lookup) =
          (StringMap.fold (fun k d a -> if StringMap.mem k a
                    then StringMap.add k d a
                    else a) local_inner local_outer, lookup)

(* Creates a blank slide given it's Identifier *)
let create_blank_slide i =
          let fill_css =
                    {Slide.padding_top="";padding_bottom="";padding_left="";padding_right = "";
                    text_color= "";background_color = "";font = "";font_size = "";font_decoration = "";
                    border = "";border_color = "";}
          in
          {Slide.id=(id_to_str
i);next="";prev="";image="";style=fill_css;onclick=None;onpress=None;elements=StringMap.empty}

(* Creates a blank element *)
let create_blank_element =
          let fill_css =
                    {Element.display=true; position_x = ""; position_y = ""; margin_top = "";
          margin_bottom = ""; margin_left = ""; margin_right = ""; padding_top = "";
          padding_bottom = ""; padding_left = ""; padding_right = ""; text_color = "";
          background_color = ""; font = ""; font_size = ""; font_decoration = "";
          border = ""; border_color = ""; width = ""; height = "";}
          in
          {Element.id="";image="";text="";style=fill_css;onclick=None;elements=StringMap.empty;}

(* Binds css to an element
 * @param attribute the css attribute, as a string
 * @param value the value to bind it to, Ir.literal
 * @param elementp the element to bind css to
 * @return the updated element
 *)
let ir_bind_css_element attribute value (elementp:Element.element) = (match (attribute, value) with
          ("display", Ir.Litbool(b)) -> {elementp with style= {elementp.style with display=b}}
   | ("position-x", Ir.Litint(i)) -> {elementp with style= {elementp.style with position_x=(string_of_int i)}}
   | ("position-x", Ir.Litper(i)) -> {elementp with style= {elementp.style with position_x=(string_of_int i)^"%"}}
   | ("position-y", Ir.Litint(i)) -> {elementp with style= {elementp.style with position_y=(string_of_int i)}}
   | ("position-y", Ir.Litper(i)) -> {elementp with style= {elementp.style with position_y=(string_of_int i)^"%"}}
   | ("margin-top", Ir.Litint(i)) -> {elementp with style= {elementp.style with margin_top=(string_of_int i)}}
```

```
    | ("margin-top", Ir.Litper(i)) -> {elementp with style= {elementp.style with margin_top=(string_of_int i)^"%"}}
    | ("margin-bottom", Ir.Litint(i)) -> {elementp with style= {elementp.style with margin_bottom=(string_of_int
i)}}
    | ("margin-bottom", Ir.Litper(i)) -> {elementp with style= {elementp.style with margin_bottom=(string_of_int
i)^"%"}}
    | ("margin-left", Ir.Litint(i)) -> {elementp with style= {elementp.style with margin_left=(string_of_int i)}}
    | ("margin-left", Ir.Litper(i)) -> {elementp with style= {elementp.style with margin_left=(string_of_int i)^"%"}}
    | ("margin-right", Ir.Litint(i)) -> {elementp with style= {elementp.style with margin_right=(string_of_int i)}}
    | ("margin-right", Ir.Litper(i)) -> {elementp with style= {elementp.style with margin_right=(string_of_int
i)^"%"}}
    | ("padding-top", Ir.Litint(i)) -> {elementp with style= {elementp.style with padding_top=(string_of_int i)}}
    | ("padding-top", Ir.Litper(i)) -> {elementp with style= {elementp.style with padding_top=(string_of_int
i)^"%"}}
    | ("padding-bottom", Ir.Litint(i)) -> {elementp with style= {elementp.style with
padding_bottom=(string_of_int i)}}
    | ("padding-bottom", Ir.Litper(i)) -> {elementp with style= {elementp.style with
padding_bottom=(string_of_int i)^"%"}}
    | ("padding-left", Ir.Litint(i)) -> {elementp with style= {elementp.style with padding_left=(string_of_int i)}}
    | ("padding-left", Ir.Litper(i)) -> {elementp with style= {elementp.style with padding_left=(string_of_int
i)^"%"}}
    | ("padding-right", Ir.Litint(i)) -> {elementp with style= {elementp.style with padding_right=(string_of_int i)}}
    | ("padding-right", Ir.Litper(i)) -> {elementp with style= {elementp.style with padding_right=(string_of_int
i)^"%"}}
    | ("text-color", Ir.Litstr(s)) -> {elementp with style= {elementp.style with text_color=s}}
    | ("background-color", Ir.Litstr(s)) -> {elementp with style= {elementp.style with background_color=s}}
    | ("font", Ir.Litstr(s)) -> {elementp with style= {elementp.style with font=s}}
    | ("font-size", Ir.Litint(i)) -> {elementp with style= {elementp.style with font_size=(string_of_int i)}}
    | ("font-decoration", Ir.Litstr(s)) -> {elementp with style= {elementp.style with font_decoration=s}}
    | ("border", Ir.Litint(i)) -> {elementp with style= {elementp.style with border=(string_of_int i)}}
    | ("border-color", Ir.Litstr(s)) -> {elementp with style= {elementp.style with border_color=s}}
    | ("width", Ir.Litint(i)) -> {elementp with style= {elementp.style with width=(string_of_int i)}}
    | ("width", Ir.Litper(i)) -> {elementp with style= {elementp.style with width=(string_of_int i)^"%"}}
    | ("height", Ir.Litint(i)) -> {elementp with style= {elementp.style with height=(string_of_int i)}}
    | ("height", Ir.Litper(i)) -> {elementp with style= {elementp.style with height=(string_of_int i)^"%"}}
    | ("id", Ir.Litstr(s)) -> {elementp with id=s}
    | ("image", Ir.Litstr(s)) -> {elementp with image=s}
    | ("text", Ir.Litstr(s)) -> {elementp with text=s}
            | (_,_) -> raise (Failure ("The following built-in attribute is not used correctly on component: " ^
attribute))
            )

(* Binds css to a slide
 * @param attribute the css attribute, as a string
 * @param value the value to bind it to, Ir.literal
 * @param slidep the Slide.slide
 * @return the updated slide
 *)
let ir_bind_css_slide attribute value (slidep:Slide.slide) = (match (attribute, value) with
            ("padding-top", Ir.Litint(i)) -> {slidep with style= {slidep.style with padding_top=(string_of_int i)}}
    | ("padding-top", Ir.Litper(i)) -> {slidep with style= {slidep.style with padding_top=(string_of_int i)^"%"}}
    | ("padding-bottom", Ir.Litint(i)) -> {slidep with style= {slidep.style with padding_bottom=(string_of_int i)}}
    | ("padding-bottom", Ir.Litper(i)) -> {slidep with style= {slidep.style with padding_bottom=(string_of_int
i)^"%"}}
    | ("padding-left", Ir.Litint(i)) -> {slidep with style= {slidep.style with padding_left=(string_of_int i)}}
    | ("padding-left", Ir.Litper(i)) -> {slidep with style= {slidep.style with padding_left=(string_of_int i)^"%"}}
    | ("padding-right", Ir.Litint(i)) -> {slidep with style= {slidep.style with padding_right=(string_of_int i)}}
    | ("padding-right", Ir.Litper(i)) -> {slidep with style= {slidep.style with padding_right=(string_of_int i)^"%"}}
    | ("text-color", Ir.Litstr(s)) -> {slidep with style= {slidep.style with text_color=s}}
    | ("background-color", Ir.Litstr(s)) -> {slidep with style= {slidep.style with background_color=s}}
    | ("font", Ir.Litstr(s)) -> {slidep with style= {slidep.style with font=s}}
    | ("font-size", Ir.Litint(i)) -> {slidep with style= {slidep.style with font_size=(string_of_int i)}}
    | ("font-decoration", Ir.Litstr(s)) -> {slidep with style= {slidep.style with font_decoration=s}}
    | ("border", Ir.Litint(i)) -> {slidep with style= {slidep.style with border=(string_of_int i)}}
    | ("border-color", Ir.Litstr(s)) -> {slidep with style= {slidep.style with border_color=s}}
    | ("next", Ir.Litslide(Identifier(s))) -> {slidep with next=s}
    | ("prev", Ir.Litslide(Identifier(s))) -> {slidep with prev=s}
    | ("image", Ir.Litstr(s)) -> {slidep with image=s}
            | (_,_) -> raise (Failure ("The following built-in attribute is not used correctly on slide: " ^ attribute))
            )

(* Gets the css attribute of a particular component/slide that must have already been bound
 * @param attribute the attribute, as a string
 * @param path the Ir.Litcomp or Ir.Litslide
 * @param lookup the lookup table
 * @return the attribute, as an Ir.literal
 *)
let ir_get_css attribute path lookup = (match path with
            Ir.Litslide(Identifier(s)) ->
                    let the_slide = StringMap.find s lookup.slides_out in
                    let is_null s_in = ((compare s_in "") = 0) in
                    let is_per s_in = ((String.get s_in ((String.length s_in)-1)) = '%') in
                    let per_to_int s_in = int_of_string (String.sub s_in 0 ((String.length s_in)-1)) in
                    let int_or_per the_attr =
                            if is_null the_attr then Ir.Litnull
                            else if is_per the_attr then Ir.Litper(per_to_int the_attr)
                            else Ir.Litint(int_of_string the_attr)
                    in
                    (match attribute with
                            "padding-top" -> int_or_per the_slide.style.padding_top
                      | "padding-bottom" -> int_or_per the_slide.style.padding_bottom
                      | "padding-left" -> int_or_per the_slide.style.padding_left
                      | "padding-right" -> int_or_per the_slide.style.padding_right
                      | "text-color" -> Ir.Litstr(the_slide.style.text_color)
                      | "background-color" -> Ir.Litstr(the_slide.style.background_color)
                      | "font" -> Ir.Litstr(the_slide.style.font)
                      | "font-size" ->  int_or_per the_slide.style.font_size
```

```
                    | "font-decoration" ->  Ir.Litstr(the_slide.style.font_decoration)
                    | "border" -> int_or_per the_slide.style.border
                    | "border-color" -> Ir.Litstr(the_slide.style.border_color)
                    | "next" -> Ir.Litslide(Identifier(the_slide.next))
                    | "prev" -> Ir.Litslide(Identifier(the_slide.prev))
                    | "image" -> Ir.Litstr(the_slide.image)
                         | _ -> raise (Failure ("The following built-in attribute does not exist for slide used in
get(): " ^ attribute))
                 )
        | Ir.Litcomp(Identifier(s), slist) ->
                 let the_slide = StringMap.find s lookup.slides_out in
                 let the_comp =
                         try StringMap.find (List.hd slist) the_slide.elements
                         with Not_found -> raise (Failure ("Cannot find the following component: " ^ s ^ "->" ^
(List.hd slist)))
                 in
                 (* e is the element, p is the full path for error printing, last param is the partial path *)
                 (* returns the element represented by the path *)
                 let rec get_comp (e : Element.element) p = function
                         [] -> e
                         | hd::tl ->
                                 try get_comp (StringMap.find hd e.elements) p tl
                                 with Not_found -> raise (Failure ("Cannot find the following component: " ^
String.concat "->" p))
                 in
                 let the_comp = get_comp the_comp (s::slist) (List.tl slist) in
                 let is_null s_in = ((compare s_in "") = 0) in
                 let is_per s_in = ((String.get s_in ((String.length s_in)-1)) = '%') in
                 let per_to_int s_in = int_of_string (String.sub s_in 0 ((String.length s_in)-1)) in
                 let int_or_per the_attr =
                         if is_null the_attr then Ir.Litnull
                         else if is_per the_attr then Ir.Litper(per_to_int the_attr)
                         else Ir.Litint(int_of_string the_attr)
                 in
                 (match attribute with
                         "display" -> Ir.Litbool(the_comp.style.display)
                         | "position-x" -> int_or_per the_comp.style.position_x
                    | "position-y" -> int_or_per the_comp.style.position_y
                    | "margin-top" -> int_or_per the_comp.style.margin_top
                    | "margin-bottom" -> int_or_per the_comp.style.margin_bottom
                    | "margin-left" -> int_or_per the_comp.style.margin_left
                    | "margin-right" -> int_or_per the_comp.style.margin_right
                         | "padding-top" -> int_or_per the_comp.style.padding_top
                    | "padding-bottom" -> int_or_per the_comp.style.padding_bottom
                    | "padding-left" -> int_or_per the_comp.style.padding_left
                    | "padding-right" -> int_or_per the_comp.style.padding_right
                    | "text-color" -> Ir.Litstr(the_comp.style.text_color)
                    | "background-color" -> Ir.Litstr(the_comp.style.background_color)
```

```
                        | "font" -> Ir.Litstr(the_comp.style.font)
                        | "font-size" ->  int_or_per the_comp.style.font_size
                        | "font-decoration" ->  Ir.Litstr(the_comp.style.font_decoration)
                        | "border" -> int_or_per the_comp.style.border
                        | "border-color" -> Ir.Litstr(the_comp.style.border_color)
                        | "width" -> int_or_per the_comp.style.width
                        | "height" -> int_or_per the_comp.style.height
                        | "id" -> Ir.Litstr(the_comp.id)
                        | "image" -> Ir.Litstr(the_comp.image)
                        | "text" -> Ir.Litstr(the_comp.text)
                                | _ -> raise (Failure ("The following built-in attribute does not exist for comp used in
get(): " ^ attribute))
                        )
        | _ -> raise (Failure ("The first parameter of get() must be a slide or component."))
        )

(* sets the css attribute of a particular component/slide that must have already been bound
 * @param attribute the attribute, as a string
 * @param avalue as a Ir.literal
 * @param path the Ir.Litcomp or Ir.Litslide
 * @param lookup the lookup table
 * @return the updated lookup table
 *)
let ir_set_css attribute avalue path lookup = (match path with
        Ir.Litslide(Identifier(s)) ->
                let the_slide = ir_bind_css_slide attribute avalue (StringMap.find s lookup.slides_out) in
                {lookup with slides_out=(StringMap.add s the_slide lookup.slides_out)}
        | Ir.Litcomp(Identifier(s), child::[]) ->
                let the_slide = StringMap.find s lookup.slides_out in
                let the_comp =
                        try (StringMap.find child the_slide.elements)
                        with Not_found -> raise (Failure ("Cannot find the following component: " ^ s ^ "->" ^
child))
                in
                let the_comp = ir_bind_css_element attribute avalue the_comp in
                (the_slide.elements <- (StringMap.add the_comp.id the_comp the_slide.elements); lookup;)
        | Ir.Litcomp(Identifier(s), slist) ->
                let the_slide = StringMap.find s lookup.slides_out in
                let the_comp =
                        try StringMap.find (List.hd slist) the_slide.elements
                        with Not_found -> raise (Failure ("Cannot find the following component: " ^ s ^ "->" ^
(List.hd slist)))
                in
                (* e is the element, p is the full path for error printing, last param is the partial path *)
                (* returns the element represented by the path *)
                let rec get_comp (e : Element.element) p = function
                        [] -> e
                        | hd::tl ->
```

```
                                        try get_comp (StringMap.find hd e.elements) p tl
                                        with Not_found -> raise (Failure ("Cannot find the following component: " ^
String.concat "->" p))
                    in
                    let parent_comp = get_comp the_comp (s::slist) (List.tl (List.rev (List.tl (List.rev slist)))) in
                    let child_comp =
                              try StringMap.find (List.hd slist) the_slide.elements
                              with Not_found -> raise (Failure ("Cannot find the following component: " ^ s ^ "->" ^
(List.hd slist)))
                    in
                    let child_comp = ir_bind_css_element attribute avalue child_comp in
                    (parent_comp.elements <- (StringMap.add child_comp.id child_comp
child_comp.elements); lookup;)
          | _ -> raise (Failure ("The first parameter of set() must be a slide or component."))
          )

exception ReturnException of literal * lookup_table

(* Main function that performs IR generation *)
let generate (vars, funcs) =

          (* Create lookup table
           * @param vars is the variables from Sast.program
           * @param funcs is the function definitions from Sast.program
           * @return the lookup table as specified *)
          let create_lookup vars funcs =
                    let fill_funcs lookup (func : Sast.func_definition) =
                              if (StringMap.mem (id_to_str func.name) lookup.funcs_in)
                              then raise (Failure ("There are two definitions for function name " ^ (id_to_str
func.name)))
                              else {lookup with funcs_in = StringMap.add (id_to_str func.name) func
lookup.funcs_in}
                    in
                    let fill_vars lookup id = {lookup with vars_in=StringMap.add (id_to_str id) Litnull
lookup.vars_in}
                    in
                    (List.fold_left fill_vars (List.fold_left fill_funcs

({funcs_in=StringMap.empty;

vars_in=StringMap.empty;

slides_out=StringMap.empty;
                                                                                          cur_slide="";
                                                                                          cur_element=None;})
funcs) vars)
          in
```

```
        (* This calls a function to generate ir
         * @param fdef is the function definition (Sast.func_definition)
         * @param actuals are the actual parameter list, of literals
         * @param lookup is the lookup table
         * @return the updated lookup table *)
        let rec call (fdef:Sast.func_definition) actuals lookupparam =

                (* Evaluates expressions for static code, binds elements, etc.
                 * @param loclook (locals, lookup)
                 * @param expression the expression to evaluate
                 * @return (Ir.literal, (locals, lookup)) *)
                let rec eval loclook = function
                        Sast.Binop(e1, op, e2) ->
                                let r1, loclook = eval loclook (fst e1) in
                                let r2, loclook = eval loclook (fst e2) in
                                let evaluate_eq b = function Ast.Equals -> b | _ -> (not b) in
                                (match r1, op, r2 with

                                        Ir.Litint(i1), Ast.Plus, Ir.Litint(i2) -> (Ir.Litint(i1 + i2), loclook)
                                        | Ir.Litint(a), Ast.Plus, Ir.Litstr(b) -> (Ir.Litstr((string_of_int a) ^ b),
loclook)
                                        | Ir.Litstr(a), Ast.Plus, Ir.Litint(b) -> (Ir.Litstr(a ^ (string_of_int b)),
loclook)
                                        | Ir.Litper(i1), Ast.Plus, Ir.Litper(i2) -> (Ir.Litper(i1 + i2), loclook)
                                        | Ir.Litper(a), Ast.Plus, Ir.Litstr(b) -> (Ir.Litstr((string_of_int a) ^ "%"
^ b), loclook)
                                        | Ir.Litstr(a), Ast.Plus, Ir.Litper(b) -> (Ir.Litstr(a ^ (string_of_int b) ^
"%"), loclook)
                                        | Ir.Litstr(a), Ast.Plus, Ir.Litstr(b) -> (Ir.Litstr(a ^ b), loclook)
                                        | Ir.Litstr(a), Ast.Plus, Ir.Litbool(b) -> (Ir.Litstr(a ^ (string_of_bool
b)), loclook)
                                        | Ir.Litbool(a), Ast.Plus, Ir.Litstr(b) -> (Ir.Litstr((string_of_bool a) ^
b), loclook)
                                        | Ir.Litstr(a), Ast.Plus, Ir.Litslide(Identifier(b)) -> (Ir.Litstr(a ^ b),
loclook)
                                        | Ir.Litslide(Identifier(a)), Ast.Plus, Ir.Litstr(b) -> (Ir.Litstr(a ^ b),
loclook)
                                        | Ir.Litnull, Ast.Plus, Ir.Litstr(b) -> (Ir.Litstr(b), loclook)
                                        | Ir.Litstr(a), Ast.Plus, Ir.Litnull -> (Ir.Litstr(a), loclook)

                                        | Ir.Litint(i1), Ast.Minus, Ir.Litint(i2) -> (Ir.Litint(i1 - i2), loclook)
                                        | Ir.Litper(i1), Ast.Minus, Ir.Litper(i2) -> (Ir.Litper(i1 - i2), loclook)

                                        | Ir.Litint(i1), Ast.Times, Ir.Litint(i2) -> (Ir.Litint(i1 * i2), loclook)
                                        | Ir.Litint(i1), Ast.Times, Ir.Litper(i2) -> (Ir.Litper(i1 * i2), loclook)
                                        | Ir.Litper(i1), Ast.Times, Ir.Litint(i2) -> (Ir.Litper(i1 * i2), loclook)

                                        | Ir.Litint(i1), Ast.Divide, Ir.Litint(i2) -> (Ir.Litint(i1 / i2), loclook)
                                        | Ir.Litper(i1), Ast.Divide, Ir.Litint(i2) -> (Ir.Litper(i1 / i2), loclook)

                                        | Ir.Litint(i1), (Ast.Equals | Ast.Notequals), Ir.Litint(i2) ->
(Ir.Litbool(evaluate_eq (i1 = i2) op), loclook)
                                        | Ir.Litper(i1), (Ast.Equals | Ast.Notequals), Ir.Litper(i2) ->
(Ir.Litbool(evaluate_eq (i1 = i2) op), loclook)
                                        | Ir.Litstr(i1), (Ast.Equals | Ast.Notequals), Ir.Litstr(i2) ->
(Ir.Litbool(evaluate_eq ((String.compare i1 i2) = 0) op), loclook)
                                        | Ir.Litbool(i1), (Ast.Equals | Ast.Notequals), Ir.Litbool(i2) ->
(Ir.Litbool(evaluate_eq (i1 = i2) op), loclook)
                                        | Ir.Litcomp(Identifier(i1), slist1), (Ast.Equals | Ast.Notequals),
Ir.Litcomp(Identifier(i2), slist2) ->
                                                (Ir.Litbool(evaluate_eq ((String.compare i1 i2) = 0 &&
                                                (List.fold_left2 (fun t s1 s2 ->
                                                        if t then ((String.compare s1 s2) = 0)
                                                        else false) true slist1 slist2)) op
                                                ), loclook)
                                        | Ir.Litslide(Identifier(s1)), (Ast.Equals | Ast.Notequals),
Ir.Litslide(Identifier(s2)) ->  (Ir.Litbool(evaluate_eq ((String.compare s1 s2) = 0) op), loclook)
                                        | Ir.Litnull, (Ast.Equals | Ast.Notequals), Ir.Litnull ->
(Ir.Litbool(evaluate_eq true op), loclook)
                                        | _, (Ast.Equals | Ast.Notequals), _ -> (Ir.Litbool(evaluate_eq false
op), loclook)

                                        | Ir.Litint(i1), Ast.Lessthan, Ir.Litint(i2) -> (Ir.Litbool(i1 < i2), loclook)
                                        | Ir.Litint(i1), Ast.Greaterthan, Ir.Litint(i2) -> (Ir.Litbool(i1 > i2),
loclook)
                                        | Ir.Litper(i1), Ast.Lessthan, Ir.Litper(i2) -> (Ir.Litbool(i1 < i2),
loclook)
                                        | Ir.Litper(i1), Ast.Greaterthan, Ir.Litper(i2) -> (Ir.Litbool(i1 > i2),
loclook)
                                        | Ir.Litstr(s1), Ast.Lessthan, Ir.Litstr(s2) ->
(Ir.Litbool((String.compare s1 s2) < 0), loclook)
                                        | Ir.Litstr(s1), Ast.Greaterthan, Ir.Litstr(s2) ->
(Ir.Litbool((String.compare s1 s2) > 0), loclook)

                                        | Ir.Litbool(s1), Ast.Or, Ir.Litbool(s2) -> (Ir.Litbool(s1 || s2), loclook)
                                        | Ir.Litbool(s1), Ast.And, Ir.Litbool(s2) -> (Ir.Litbool(s1 && s2),
loclook)

                                        | a, op, b -> raise(Failure("Semantic Analyzer should print out error
for this operation"))
                                )
                        | Sast.Notop(e) ->
                                let r, loclook = eval loclook (fst e) in
                                (match r with
                                        Ir.Litbool(b) -> (Ir.Litbool(not b), loclook)
                                        | _ -> raise (Failure ("This cannot be notted")))
```

```
                    | Sast.Litint(i) -> (Ir.Litint(i), loclook)
                    | Sast.Litper(i) -> (Ir.Litper(i), loclook)
                    | Sast.Litstr(s) -> (Ir.Litstr(s), loclook)
                    | Sast.Litbool(b) -> (Ir.Litbool(b), loclook)
                    | Sast.Litnull -> (Ir.Litnull, loclook)
                    | Sast.Assign(Identifier(i), (e,_)) ->
                        let r, (locals, lookup) = eval loclook e in
                        if StringMap.mem i locals
                            then r, (StringMap.add i r locals, lookup)
                        else if StringMap.mem i lookup.vars_in
                            then r, (locals, {lookup with vars_in=StringMap.add i r
lookup.vars_in})
                        else raise (Failure ("Undeclared identifier " ^ i))
                    | Sast.Variable(Identifier(i)) ->
                        let (locals, lookup) = loclook in
                        if ((StringMap.mem i lookup.funcs_in)
                            && (match (StringMap.find i lookup.funcs_in).t with Ast.Slide ->
true | _ -> false))
                            then if (not (StringMap.mem i lookup.slides_out))
                                then try (Ir.Litslide(Identifier(i)), (locals,
                                    {(call (StringMap.find i lookup.funcs_in) [] lookup)
with cur_slide=lookup.cur_slide}))
                                    with ReturnException(r, l) ->
(Ir.Litslide(Identifier(i)), (fst loclook, {l with cur_slide=lookup.cur_slide}))
                                else (Ir.Litslide(Identifier(i)), (locals, lookup))
                        else if StringMap.mem i (fst loclook)
                            then (StringMap.find i (fst loclook), loclook)
                        else if StringMap.mem i (snd loclook).vars_in
                            then (StringMap.find i (snd loclook).vars_in, loclook)
                        else raise (Failure ("Undeclared identifier " ^ i))
                    | Sast.Component(i, elist) ->
                        let (the_slide, loclook) = (eval loclook (Sast.Variable(i))) in
                        let loclook = (match the_slide with
                            Ir.Litslide(_) -> loclook
                            | _ -> raise (Failure ("The following slide has not been created,
used on a Component literal: " ^ (id_to_str i)))
                            )
                        in
                        let (rlist, loclookp) =
                                List.fold_left
                                (fun (actuals, loclook) actual -> let (r, loclook) = eval
loclook (fst actual) in (r :: actuals, loclook))
                                ([], loclook) (List.rev elist)
                        in
                        let slist = List.fold_left
                            (fun l r -> (match r with
                                Ir.Litstr(s) -> (s :: l)
                                | _ -> raise (Failure ("Only strings allowed in brackets, for
" ^ (id_to_str i)) )) )
                                        [] (List.rev rlist)
                              in
                              (Ir.Litcomp(i, slist), loclookp)
                    | Sast.Call(f) ->
                              let process_built_in_attr built_in_name =
                                        let (actual, loclook) = eval loclook (fst (List.hd f.actuals)) in
                                        let (locals, lookup) = loclook in
                                        (match lookup.cur_element with
                                                            None ->
                                                                      let the_slide = StringMap.find
lookup.cur_slide lookup.slides_out in
                                                                      (actual, (locals,
                                                                      {lookup with slides_out = StringMap.add
lookup.cur_slide
                                                                      (ir_bind_css_slide built_in_name actual
the_slide) lookup.slides_out}))
                                                            | Some(x) -> (actual, (locals,
                                                                      {lookup with cur_element =
Some(ir_bind_css_element built_in_name actual x)}))
                                        )
                              in
                              (match f.cname with
                                        Identifier("display") -> process_built_in_attr "display"
                                        | Identifier("position-x") -> process_built_in_attr "position-x"
                                        | Identifier("position-y") -> process_built_in_attr "position-y"
                                        | Identifier("margin-top") -> process_built_in_attr "margin-top"
                                        | Identifier("margin-bottom") -> process_built_in_attr
"margin-bottom"
                                        | Identifier("margin-left") -> process_built_in_attr "margin-left"
                                        | Identifier("margin-right") -> process_built_in_attr "margin-right"
                                        | Identifier("padding-top") -> process_built_in_attr "padding-top"
                                        | Identifier("padding-bottom") -> process_built_in_attr
"padding-bottom"
                                        | Identifier("padding-left") -> process_built_in_attr "padding-left"
                                        | Identifier("padding-right") -> process_built_in_attr "padding-right"
                                        | Identifier("text-color") -> process_built_in_attr "text-color"
                                        | Identifier("background-color") -> process_built_in_attr
"background-color"
                                        | Identifier("font") -> process_built_in_attr "font"
                                        | Identifier("font-size") -> process_built_in_attr "font-size"
                                        | Identifier("font-decoration") -> process_built_in_attr
"font-decoration"
                                        | Identifier("border") -> process_built_in_attr "border"
                                        | Identifier("border-color") -> process_built_in_attr "border-color"
                                        | Identifier("width") -> process_built_in_attr "width"
                                        | Identifier("height") -> process_built_in_attr "height"
                                        | Identifier("id") -> process_built_in_attr "id"
```

```
                                            | Identifier("image") -> process_built_in_attr "image"
                                            | Identifier("text") -> process_built_in_attr "text"
                                            | Identifier("next") -> process_built_in_attr "next"
                                            | Identifier("prev") -> process_built_in_attr "prev"
                                            | Identifier("random") ->
                                                    let (actual, loclook) = eval loclook (fst (List.hd f.actuals))
in
                                                    let get_rand integer = Random.self_init(); Random.int
integer in
                                                    (match actual with
                                                            Ir.Litint(i) -> (Ir.Litint(get_rand i), loclook)
                                                            | _ -> raise (Failure ("You must pass in an integer
to random()")))
                                            | Identifier("on-click") ->
                                                    let f_to_call = (match (List.hd f.actuals) with
                                                            (Sast.Call(f),_) -> f
                                                            | _ -> raise (Failure ("You must pass in an
function to call  to on-click()")))
                                                    in
                                                    let (pactuals, loclook) =
                                                            List.fold_left
                                                            (fun (actuals, loclook) actual -> let (r, loclook) =
eval loclook (fst actual) in (r :: actuals, loclook))
                                                            ([], loclook) (List.rev f_to_call.actuals)
                                                    in
                                                    let (locals, lookup) = loclook in
                                                    (match lookup.cur_element with
                                                                    None ->
                                                                            let the_slide = StringMap.find
lookup.cur_slide lookup.slides_out in
                                                                            (Litnull, (locals,
                                                                            {lookup with slides_out =
StringMap.add lookup.cur_slide
                                                                            ({the_slide with onclick =
Some({Ir.cname=f_to_call.cname; actuals=pactuals;})}) lookup.slides_out}))
                                                                    | Some(x) -> (Litnull, (locals,
                                                                            {lookup with cur_element =
Some({x with onclick = Some({Ir.cname=f_to_call.cname; actuals=pactuals;})})})))
                                                    )
                                            | Identifier("on-press") ->
                                                    let f_to_call = (match (List.nth f.actuals 1) with
                                                            (Sast.Call(f),_) -> f
                                                            | _ -> raise (Failure ("You must pass in a function
to call as 2nd parameter to on-press()")))
                                                    in
                                                    let (key_to_press, loclook) = eval loclook (fst (List.hd
f.actuals)) in
                                                    let key_to_press = (match key_to_press with
                                                            Ir.Litstr(s) -> s
                                                            | _ ->  raise (Failure ("You must pass in a string
as key as 1st parameter to on-press()")))
                                                    in
                                                    let (pactuals, loclook) =
                                                            List.fold_left
                                                            (fun (actuals, loclook) actual -> let (r, loclook) =
eval loclook (fst actual) in (r :: actuals, loclook))
                                                            ([], loclook) (List.rev f_to_call.actuals)
                                                    in
                                                    let (locals, lookup) = loclook in
                                                    let the_slide = StringMap.find lookup.cur_slide
lookup.slides_out in
                                                            (Ir.Litnull, (locals,
                                                            {lookup with slides_out = StringMap.add
lookup.cur_slide
                                                            ({the_slide with onpress =
Some(key_to_press,{Ir.cname=f_to_call.cname; actuals=pactuals;})}) lookup.slides_out}))
                                            | Identifier("get") ->
                                                    let (param1, loclook) = eval loclook (fst (List.hd f.actuals))
in
                                                    let (param2, loclook) = eval loclook (fst (List.nth f.actuals
1)) in
                                                    (match param2 with
                                                            Ir.Litstr(s) ->  (ir_get_css s param1 (snd loclook),
loclook)
                                                            | _ ->  raise (Failure ("You must pass in a string
as attribute name as 2nd parameter to get()"))
                                                    )
                                            | Identifier("set") ->
                                                    let (param1, loclook) = eval loclook (fst (List.hd f.actuals))
in
                                                    let (param2, loclook) = eval loclook (fst (List.nth f.actuals
1)) in
                                                    let param2 =
                                                            (match param2 with
                                                                    Ir.Litstr(s) -> s
                                                                    | _ ->  raise (Failure ("You must pass in a
string as attribute name as 2nd parameter to set()"))
                                                            )
                                                    in
                                                    let (param3, loclook) = eval loclook (fst (List.nth f.actuals
2)) in
                                                    (param3, (fst loclook, ir_set_css param2 param3 param1
(snd loclook)))
                                            | Identifier(_) ->
                                    (* The rest of these lines are for non-built-in functions *)
                                    let fdecl =
                                            (match f.cname with
                                                    (* Except the special component box *)
                                                    Identifier("box") ->

{Sast.t=Ast.Comp;name=Sast.Identifier("box");formals=[];

inheritance=Some(Sast.Identifier("box"));paractuals=[];body=[];}
                                                    (* All others, you must find the function *)
                                                    | Identifier(_) ->
                                                    let not_slide (check:Sast.func_definition) = (match
check.t with
                                                            Ast.Slide -> raise (Failure ("Cannot call a slide
like a regular function: " ^ (id_to_str f.cname)))
                                                            | _ -> check)
                                                    in
                                            try not_slide (StringMap.find (id_to_str f.cname) (snd
loclook).funcs_in)
                                            with Not_found -> raise (Failure ("undefined function " ^ (id_to_str
f.cname)))
                                            )
                                    in
                                    let (actuals, loclook) =
                                                    List.fold_left
                                                    (fun (actuals, loclook) actual -> let (r, loclook) = eval
loclook (fst actual) in (r :: actuals, loclook))
                                                    ([], loclook) (List.rev f.actuals)
                                    in
                                    let (locals, lookup) = loclook in
                                    let r, returned_lookup =
                                            try (Ir.Litnull, call fdecl actuals lookup)
                                            with ReturnException(r, lookup) -> (r, lookup)
                                    in
                                    (match fdecl.t with
                                            Ast.Slide -> raise (Failure ("This shouldn't be displayed."))
                                            | Ast.Attr -> (Ir.Litnull, (locals, returned_lookup))
                                            | Ast.Func -> (r, (locals, returned_lookup))
                                            | Ast.Comp ->
                                                    let (locals, returned_lookup) = exec (locals,
returned_lookup) f.mods in
                                                    (match lookup.cur_element with
                                                            (* Bind to slide *)
                                                            None ->
                                                                    let the_slide = (StringMap.find
lookup.cur_slide returned_lookup.slides_out) in
                                                                    let the_element = (match
returned_lookup.cur_element with
                                                                        None -> raise (Failure ("Element
binding error"))
```

```
                                                                          | Some(x) -> x)
                                                                  in
                                                                  if (StringMap.mem the_element.id
the_slide.elements)
                                                                  then raise (Failure ("The following
element already exists in slide->comp: " ^ the_slide.id ^ "->" ^ the_element.id))
                                                                  else
                                                                  (the_slide.elements <- (StringMap.add
the_element.id the_element the_slide.elements));
                                                                  (Ir.Litnull, (locals, {returned_lookup with
cur_element=None}))
                                                          (* Bind to another element *)
                                                          | Some(par_element) ->
                                                                  let the_element = (match
returned_lookup.cur_element with
                                                                          None -> raise (Failure ("Element
binding error"))
                                                                          | Some(x) -> x)
                                                                  in
                                                                  if (StringMap.mem the_element.id
par_element.elements)
                                                                  then raise (Failure ("The following
element already exists in definition " ^ (id_to_str fdef.name) ^ ": " ^ the_element.id))
                                                                  else
                                                                  (par_element.elements <- (StringMap.add
the_element.id the_element par_element.elements));
                                                                  (Ir.Litnull, (locals, {returned_lookup with
cur_element=Some(par_element)}))
                                                  )
                                  )
                                  )

                  (* Actually executes statements
                   * This part doesn't use the type info, more useful for javascript compiling later on
                   * @param loclook (locals, lookup)
                   * @param statement the statement the execute
                   * @return (locals, lookup) *)
                  and exec loclook = function
                          Sast.Block(stmts) -> merge_locals (fst loclook) (List.fold_left exec loclook stmts)
                          | Sast.Expr(e) -> let _, loclook = eval loclook (fst e) in loclook
                          | Sast.If(e, s1, s2) ->
                                  let v, loclook = eval loclook (fst e) in
                                  (match v with
                                          Ir.Litbool(false) -> merge_locals (fst loclook) (exec loclook s2)
                                          | Ir.Litint(0) -> merge_locals (fst loclook) (exec loclook s2)
                                          | Ir.Litper(0) -> merge_locals (fst loclook) (exec loclook s2)
                                          | Ir.Litstr("") -> merge_locals (fst loclook) (exec loclook s2)
                                          | Ir.Litnull -> merge_locals (fst loclook) (exec loclook s2)
```

```
                                        | _ -> merge_locals (fst loclook) (exec loclook s1))
                    | Sast.While(e, s) ->
                              let rec loop loclook =
                                        let v, loclook = eval loclook (fst e) in
                                        (match v with
                                                  Ir.Litbool(false) -> loclook
                                                  | Ir.Litint(0) -> loclook
                                                  | Ir.Litper(0) -> loclook
                                                  | Ir.Litstr("") -> loclook
                                                  | Ir.Litnull -> loclook
                                                  | _ -> loop (exec loclook s))
                              in merge_locals (fst loclook) (loop loclook)
                    | Sast.Declaration(Identifier(s)) ->
                              if StringMap.mem s (fst loclook)
                              then raise (Failure("The following variable, already declared: " ^ s))
                              else (StringMap.add s Litnull (fst loclook), snd loclook)
                    | Sast.Decassign(Identifier(s), e) ->
                              let r, locklook = eval loclook (fst e) in
                              if StringMap.mem s (fst loclook)
                              then raise (Failure ("The following variable, already declared: " ^ s))
                              else (StringMap.add s r (fst loclook), snd loclook)
                    | Sast.Return((e, _)) ->
                              let r, (locals, lookup) = eval loclook e in
                              raise (ReturnException(r, lookup))
          in

          (* This section takes care of scoping issues before calling exec on statements *)
          (* Assign the locals from the actual parameters *)
          let locals =
          try List.fold_left2
                              (fun locals formal actual -> StringMap.add (id_to_str formal) actual locals)
                              StringMap.empty fdef.formals actuals
          with Invalid_argument(_) ->
                              raise (Failure ("Wrong number of arguments passed to " ^ (id_to_str
fdef.name)))
          in
          (* Immediately bind if it's a slide, update current slide as well *)
          (* Create the element if it's a comp and its parent is "box" *)
          (* Call its parent if it's a comp and its parent is not "box" *)
          (* Finally, do nothing if it's any other type of function*)
          let lookup = function
                    Ast.Slide -> (locals, {lookupparam with
                              slides_out= StringMap.add (id_to_str fdef.name) (create_blank_slide
fdef.name) lookupparam.slides_out;
                              cur_slide = (id_to_str fdef.name);})
                    | Ast.Comp -> (match fdef.inheritance with
                              Some(Sast.Identifier("box")) -> (locals, {lookupparam with cur_element =
Some(create_blank_element)})
```

```
                                        | Some(Sast.Identifier(s)) ->
                                                  let parent =
                                                            try StringMap.find s lookupparam.funcs_in
                                                            with Not_found -> raise (Failure ("The following component
is not defined: " ^ s))
                                                  in
                                                  let isComp = function Ast.Comp -> true | _ -> false in
                                                  if isComp parent.t
                                                  then
                                                            let loclook = (locals, lookupparam) in
                                                            let (paractuals, loclook) =
                                                                      List.fold_left
                                                                      (fun (actuals, loclook) actual -> let (r, loclook) =
eval loclook (fst actual) in (r :: actuals, loclook))
                                                                      ([], loclook) (List.rev fdef.paractuals)
                                                            in
                                                            try (fst loclook, (call parent paractuals (snd loclook)))
                                                            with ReturnException(r, lookup) -> (fst loclook, lookup)
                                                  else
                                                            raise (Failure ("A component can only inherit from a
component for " ^ (id_to_str fdef.name)))
                                        | None -> raise (Failure ("The following component needs to inherit from a
component: " ^ (id_to_str fdef.name)))
                                        )
                              | _ -> (locals, lookupparam)
                    in
                    (* Now recursively execute every statement with the updated locals *)
                    snd (List.fold_left exec (lookup fdef.t) fdef.body)
          in

          (* Here is where all the functions get called to produce the final output *)
          let lookup = create_lookup vars funcs in
          let pre_ir =
                    try (call (StringMap.find "main" lookup.funcs_in) [] lookup)
                    with Not_found -> raise (Failure ("There must exist a main() slide"))
          in
          (StringMap.fold (fun k d l -> d :: l) (pre_ir.slides_out) [], vars, funcs_to_js funcs)
```

8.1.8 ir.ml

```
(* Authors: Yunhe (John) Wang, Lauren Zou, Aftab Khan *)

(*
   ir.mli is the immediate represenation between the sast.ml and the actual compiled code, which is an
HTML file with embedded CSS and JavaScript.

   All functions that generate static code are resolved into IR components, while all dynamic code are
represented by the SAST for javascript generation.
```

```
   An empty string "" corresponds to a null value. For nonstring, option is used.
*)
open Sast
module StringMap = Map.Make(String)

(* All expressions must be evaluated before passing to javascript *)
type literal =
  | Litint of int (* 42 *)
  | Litper of int (* 42% *)
  | Litstr of string (* “foo” *)
  | Litbool of bool (* true *)
  | Litcomp of Sast.identifier * string list (* identifier["child"]["child"] etc. to fetch component *)
  | Litslide of Sast.identifier (* identifier that is the name of a slide *)
  | Litnull

(* This is a call to whatever function is called onclick or onpress*)
type js_call = {
          cname : Sast.identifier;   (* Name of function passed to javascript, can only be of func type *)
          actuals: literal list; (* The actual parameters of the function passed, can only be literals *)
}

(* This is the template for all possible js function definitions *)
type js_definition = {
          name : Sast.identifier;     (* Name of the function *)
          formals : Sast.identifier list; (* Formal parameters *)
          body : Sast.stmt list;       (* Body of javascript definition *)
}

(* This css is either located with an element, or applies to a class of elements (comp definition) *)
module Element = struct
type css = {
   display : bool;

   position_x : string;
   position_y : string;

   margin_top : string;
   margin_bottom : string;
   margin_left : string;
   margin_right : string;

   padding_top : string;
   padding_bottom : string;
   padding_left : string;
   padding_right : string;

   text_color : string;
   background_color : string;
```

```
    font : string;
    font_size : string;
    font_decoration : string;

    border : string;
    border_color : string;

    width : string;
    height : string;
}

(* Elements with a slide or element *)
type element = {
    id : string;                                (* Unique id of a component WITHIN its outer component, concatenate
with hyphens to obtain css id*)
    image : string;                             (* Image inside the element (optional) *)
    text : string;                              (* Text inside the element (optional) *)
    style : css;                                (* CSS as applied to this particular element with this id *)
    onclick : js_call option;                   (* Name of javascript function to apply on click, empty string means
none *)
    mutable elements : element StringMap.t;     (* Map of element id (string) -> element *)
}
end

module Slide = struct
(* Possible CSS that can apply to slides *)
type slide_css = {
    padding_top : string;
    padding_bottom : string;
    padding_left : string;
    padding_right : string;

    text_color: string;
    background_color : string;

    font : string;
    font_size : string;
    font_decoration : string;

    border : string;
    border_color : string;
}

(* This is a slide*)
type slide = {
    id : string;                                (* Id of the slide = name of the slide function*)
    next : string;                              (* Id of the next slide = name of the slide function that is next *)
```

```
   prev : string;                              (* Id of the previous slide = name of the slide function that is prev
*)
   image : string;                             (* URL of any background image *)
   style : slide_css;                          (* CSS as applied to the slide in general *)
   onclick : js_call option;                   (* Name of javascript function to apply on click *)
   onpress : (string * js_call) option;             (* Key to press, name of javascript function to apply on
press *)
   mutable elements : Element.element StringMap.t;      (* Map of element id (string) -> element *)
}
end

(* Slide list is the list of slides, with its child elements, with their child elements, etc. *)
(* identifier list is a list of the global variables, these start out null at javascript run time *)
(* js definition list is a list of all the functions (not attr/comp/slide) to evaluate javascript *)
type program = Slide.slide list * Sast.identifier list * js_definition list
```

8.1.9 Compile.ml

```
(* Author: Lauren Zou *)

open Ast
open Sast
open Ir
open Ir.Element
open Ir.Slide
module StringMap = Map.Make(String)

(* Inserts the appropriate number of tabs *)
let rec tab number =
   if number == 0 then ""
   else "    " ^ tab (number - 1)

(* Returns a pixel or a percent based on the string*)
let string_pixel_percent integer =
   if String.length integer == 0 then ""
   else
      if (String.get integer ((String.length integer)-1)) == '%' then integer
      else integer ^ "px"
;;

(* Translates a CSS property given the property and its value *)
let string_of_css_property property value =
   if String.length value > 0 then
      tab 3 ^ property ^ ": " ^ value ^ ";\n"
   else ""
;;

(* Translates a CSS display property *)
let string_of_css_display_property display = match display with
```

```
    | true -> ""
    | false -> tab 3 ^ "display: hidden;\n"
;;

(* Translates a CSS font decoration property *)
let string_of_css_font_decoration decoration = match decoration with
    | "" -> ""
    | "italic" -> tab 3 ^ "font-style: italic;\n"
    | "bold" -> tab 3 ^ "font-weight: bold;\n"
    | "underline" -> tab 3 ^ "text-decoration: underline;\n"
    | _ -> ""
;;

(* Translates a CSS position property *)
let string_of_css_position_property left top =
    if String.length top > 0 || String.length left > 0 then
        tab 3 ^ "position: absolute;\n"
    else ""
;;
(* Translates a CSS border property *)
let string_of_css_border_property width color =
    if String.length width > 0 then
        tab 3 ^ "border: " ^ width ^ " " ^ color ^ " solid;\n"
    else ""

(* Translates the CSS of an element given the CSS style and id *)
let string_of_element_css (style:Element.css) id =
    tab 2 ^ "#" ^ id ^ " {\n" ^

    string_of_css_display_property style.display ^

    string_of_css_position_property style.position_x style.position_y ^
    string_of_css_property "left" (string_pixel_percent style.position_x) ^
    string_of_css_property "top" (string_pixel_percent style.position_y) ^

    string_of_css_property "margin-top" (string_pixel_percent style.margin_top) ^
    string_of_css_property "margin-bottom" (string_pixel_percent style.margin_bottom) ^
    string_of_css_property "margin-left" (string_pixel_percent style.margin_left) ^
    string_of_css_property "margin-right" (string_pixel_percent style.margin_right) ^

    string_of_css_property "padding-top" (string_pixel_percent style.padding_top) ^
    string_of_css_property "padding-bottom" (string_pixel_percent style.padding_bottom) ^
    string_of_css_property "padding-left" (string_pixel_percent style.padding_left) ^
    string_of_css_property "padding-right" (string_pixel_percent style.padding_right) ^

    string_of_css_property "color" style.text_color ^
    string_of_css_property "background-color" style.background_color ^
    string_of_css_property "font-family" style.font ^
    string_of_css_property "font-size" (string_pixel_percent style.font_size) ^
    string_of_css_font_decoration style.font_decoration ^

    string_of_css_border_property (string_pixel_percent style.border) style.border_color ^

    string_of_css_property "width" (string_pixel_percent style.width) ^
    string_of_css_property "height" (string_pixel_percent style.height) ^

    tab 2 ^ "}\n"
;;

(* Translates the CSS of a slide given the CSS style *)
let string_of_slide_css (style:Slide.slide_css) classname =
    tab 2 ^ "#" ^ classname ^ " {\n" ^

    string_of_css_property "padding-top" (string_pixel_percent style.padding_top) ^
    string_of_css_property "padding-bottom" (string_pixel_percent style.padding_bottom) ^
    string_of_css_property "padding-left" (string_pixel_percent style.padding_left) ^
    string_of_css_property "padding-right" (string_pixel_percent style.padding_right) ^

    string_of_css_property "color" style.text_color ^
    string_of_css_property "background-color" style.background_color ^

    string_of_css_property "font-family" style.font ^
    string_of_css_property "font-size" (string_pixel_percent style.font_size) ^
    string_of_css_font_decoration style.font_decoration ^

    string_of_css_property "border-width" (string_pixel_percent style.border) ^
    string_of_css_property "border-color" style.border_color ^

    tab 2 ^ "}\n"
;;

(* Retrieves the styles from an element and its children elements *)
let rec get_css_from_element (element, element_id) =
    "\n" ^ string_of_element_css element.Element.style element_id ^

    (* Get the CSS from the elements of this element *)
    String.concat "\n" (List.map get_css_from_element (StringMap.fold (fun id element l -> (element,
element_id ^ "-" ^ id)::l) element.Element.elements []))
;;

(* Retrieves the styles from a slide and its children elements *)
let get_css_from_slide slide =
    (* Get the CSS from the slide *)
    "\n" ^ string_of_slide_css slide.Slide.style slide.Slide.id ^
    (* Get the CSS from the elements of this slide *)
    String.concat "\n" (List.map get_css_from_element (StringMap.fold (fun id element l -> (element,
slide.Slide.id ^ "-" ^ id)::l) slide.Slide.elements []))
;;

(* Translates a string of text to text *)
let string_of_text text tab_level =
    if String.length text > 0 then
        tab (tab_level + 1) ^ text ^ "\n"
    else ""
;;

(* Translates an image url to an <img> tag *)
let string_of_image image tab_level =
    if String.length image > 0 then
        tab (tab_level + 1) ^ "<img src=\"" ^ image ^ "\" />\n"
    else ""
;;

(* Translates next and prev slides into <a> tags *)
let string_of_next_prev next_prev slide =
    if String.length slide > 0 then
        tab 2 ^ "<a class=\"" ^ next_prev ^ "\" href=\"#" ^ slide ^ "\"></a>\n"
    else ""
;;

(* Retrieves the HTML from an element and its children elements *)
let rec get_html_from_element (element, element_id, tab_level) =
    tab tab_level ^ "<div id=\"" ^ element_id ^ "\" class=\"box\">\n" ^

    string_of_text element.text tab_level ^
    string_of_image element.image tab_level ^

    (* Get the HTML from the elements of this element *)
    String.concat "\n\n" (List.map get_html_from_element (StringMap.fold (fun id element l -> (element,
element_id ^ "-" ^ id, tab_level + 1)::l) element.Element.elements [])) ^

    tab tab_level ^ "</div>\n"
;;

(* Retrieves the HTML from a slide and its children elements *)
let get_html_from_slide slide =
    tab 1 ^ "<div id=\"" ^ slide.Slide.id ^ "\" class=\"slide\">\n" ^

    (* Get the HTML from each element of this slide *)
    String.concat "\n\n" (List.map get_html_from_element (StringMap.fold (fun id element l -> (element,
slide.Slide.id ^ "-" ^id, 2)::l) slide.Slide.elements [])) ^
```

```
    (* Next and prev *)
    string_of_next_prev "prev" slide.Slide.prev ^
    string_of_next_prev "next" slide.Slide.next ^

    tab 1 ^ "</div>"
;;

(* Translates an identifier into a string *)
let string_of_identifier = function
    Identifier(s) -> s
;;

(* Translates literals into strings *)
let string_of_literal literal = match literal with
    | Litint integer -> string_of_int integer
    | Litper integer -> string_of_int integer ^ "%"
    | Litstr str -> "\"" ^ str ^ "\""
    | Litbool boolean -> string_of_bool boolean
    | Litcomp (parent, children) ->
        "" ^ string_of_identifier parent ^ "-" ^
        String.concat "-" (List.map (fun child -> child) children)
    | Litslide identifier -> string_of_identifier identifier
    | Litnull -> "null"
;;

(* Translates an operator into a string *)
let string_of_operator operator = match operator with
    | Plus -> "+"
    | Minus -> "-"
    | Times -> "*"
    | Divide -> "/"
    | Equals -> "=="
    | Notequals -> "!="
    | Lessthan -> "<"
    | Greaterthan -> ">"
    | Or -> "||"
    | And -> "&&"
;;

let string_of_set arg1 arg2 = match arg1 with
    | "\"text\"" -> ".html(" ^ arg2 ^ ");\n"
    | "\"image\"" -> ".html(<img src=\"" ^ arg2 ^ "\" />);\n"
    | _ -> ".css(" ^ arg1 ^ ", " ^ arg2 ^ ");\n"
;;

let string_of_get arg1 = match arg1 with
    | "\"text\"" -> ".html();\n"
    | "\"image\"" -> ".children('img');\n"
    | _ -> ".css(" ^ arg1 ^ ");\n"
;;

(* Translates an expression into a string *)
let rec string_of_expression (expression_detail, expression_type) = match expression_detail with
    | Binop (expr1, operator, expr2) ->
        "(" ^ string_of_expression expr1 ^ " " ^ string_of_operator operator ^ " " ^ string_of_expression expr2 ^
")"
    | Notop expr -> "!(" ^ string_of_expression expr ^ ")"
    | Litint integer -> string_of_int integer
    | Litper integer -> string_of_int integer ^ "%"
    | Litbool boolean -> string_of_bool boolean
    | Litstr str -> "\"" ^ str ^ "\""
    | Litnull -> "null"
    | Assign (identifier, expr) ->
        string_of_identifier identifier ^ " = " ^ string_of_expression expr
    | Variable identifier -> string_of_identifier identifier
    | Component (parent, children) ->
        "'#" ^ string_of_identifier parent ^ "'" ^
        String.concat "" (List.map (fun child -> "+ '-' + " ^ string_of_expression child) children)
    | Call func_call -> match func_call.cname with
        | Identifier("set") -> "$(" ^ string_of_expression (List.nth func_call.actuals 0) ^ ")" ^ (string_of_set
(string_of_expression (List.nth func_call.actuals 1)) (string_of_expression (List.nth func_call.actuals 2)) )
        | Identifier("get") -> "$(" ^ string_of_expression (List.nth func_call.actuals 0) ^ ")" ^ (string_of_get
(string_of_expression (List.nth func_call.actuals 1)))
        | _ -> string_of_identifier func_call.cname ^ "(" ^ String.concat ", " (List.map (fun expr ->
string_of_expression expr) func_call.actuals) ^ ");"
;;

(* Translates a statement into a string *)
let rec string_of_statement statement tab_level = match statement with
    | Block stmt ->
        String.concat "\n" (List.map (fun statement -> string_of_statement statement tab_level) stmt)
    | Expr expr ->
        tab tab_level ^ string_of_expression expr ^ ";"
    | Return expr ->
        tab tab_level ^ "return " ^ string_of_expression expr ^ ";"
    | If (expr, stmt1, stmt2) ->
        tab tab_level ^ "if (" ^ string_of_expression expr ^ ") {\n" ^ (string_of_statement stmt1 (tab_level + 1)) ^
"\n" ^
        tab tab_level ^ "} else {\n" ^ (string_of_statement stmt2 (tab_level + 1)) ^ "\n" ^
        tab tab_level ^ "}\n"
    | While (expr, stmt) ->
        tab tab_level ^ "while (" ^ string_of_expression expr ^ ") {\n" ^ (string_of_statement stmt (tab_level + 1))
^ "\n" ^
        tab tab_level ^ "}\n"
    | Declaration identifier ->
        tab tab_level ^ "var " ^ string_of_identifier identifier ^ ";"
    | Decassign (identifier, expr) ->
        tab tab_level ^ "var " ^ string_of_identifier identifier ^ " = " ^ string_of_expression expr ^ ";"
;;

(* Translates the script into JavaScript *)
let get_javascript script =
    (* Function definition *)
    tab 2 ^ "function " ^ string_of_identifier script.name ^ "(" ^

    (* Formals *)
    String.concat ", " (List.map (fun identifier -> string_of_identifier identifier) script.formals) ^ ") {\n"^

    (* Statements *)
    String.concat "\n" (List.map (fun statement -> string_of_statement statement 3) script.body) ^ "\n" ^

    tab 2 ^ "}"
;;

(* Translates onclick functionality *)
let string_of_onclick js_call id = match js_call with
    | None -> ""
    | Some(onclick) ->
        tab 2 ^ "$('#" ^ id ^ "').click(function() {\n" ^ string_of_identifier onclick.cname ^ "(" ^
        String.concat ", " (List.map string_of_literal onclick.actuals) ^
        ")\n});\n"
;;

(* Translates onpress functionality *)
let string_of_onpress js_call id = match js_call with
    | None -> ""
    | Some(onpress) ->
        tab 2 ^ "$('#" ^ id ^ "').keypress(function(e) {\n" ^
        tab 3 ^ "if (e.keycode == '" ^ fst onpress ^ "') {\n" ^
        tab 4 ^ string_of_identifier (snd onpress).cname ^ "(" ^
        String.concat ", " (List.map string_of_literal (snd onpress).actuals) ^
        ");\n" ^
        tab 3 ^ "}\n" ^
        tab 2 ^ "});\n"
;;

(* Gets the onclick of an element *)
let rec get_element_onclick (element, element_id) =
    string_of_onclick element.Element.onclick element_id ^

    (* Get onclick of children elements *)
    String.concat "\n\n" (List.map get_element_onclick (StringMap.fold (fun id element l -> (element,
element_id ^ "-" ^ id)::l) element.Element.elements []))
;;
```

```
(* Gets the onclick and onpress of a slide *)
let get_slide_onclick_onpress slide =
   string_of_onclick slide.Slide.onclick slide.Slide.id ^
   string_of_onpress slide.Slide.onpress slide.Slide.id ^

   (* Get onclick of children elements *)
   String.concat "\n\n" (List.map get_element_onclick (StringMap.fold (fun id element l -> (element,
slide.Slide.id ^ "-" ^id)::l) slide.Slide.elements []))
;;

let compile (slides, identifiers, scripts) =
   "<!DOCTYPE html>\n\n"^
   "<html>\n\n"^
   "<head>\n" ^

   (* Title *)
   tab 1 ^ "<title>SPWAG</title>" ^

   (* If we have time, we should abstract out the config paths *)
   tab 1 ^ "<link rel=\"stylesheet\" type=\"text/less\" href=\"../../../config/config.css\">\n" ^

   tab 1 ^ "<style type=\"text/css\">\n" ^

   (* Abstract out all of the CSS *)
   String.concat "\n" (List.map get_css_from_slide slides) ^ "\n" ^

   tab 1 ^ "</style>\n" ^

   "</head>\n\n" ^
   "<body>\n" ^

   (* HTML components such as slides and elements *)
   String.concat "\n" (List.map get_html_from_slide slides) ^ "\n\n" ^

   tab 1 ^ "<script type=\"text/javascript\"
src=\"http://ajax.googleapis.com/ajax/libs/jquery/1.10.2/jquery.min.js\"></script>\n" ^
   tab 1 ^ "<script type=\"text/javascript\" src=\"../../../config/config.js\"></script>\n\n" ^

   tab 1 ^ "<script>\n" ^

   (* Handle onclicks and onpresses *)
   String.concat "\n" (List.map get_slide_onclick_onpress slides) ^ "\n" ^

   (* Javascript functions *)
   String.concat "\n" (List.map get_javascript scripts) ^ "\n" ^

   tab 1 ^ "</script>\n\n" ^
```

```
  "</body>\n\n" ^
  "</html>\n"
```

8.1.10 spwag.ml

```
(* Author: Aftab Khan, Lauren Zou Contributor: Yunhe (John) Wang *)

type action = Ast | Irgenerator | Preprocessor | Compile

external preprocess: unit -> string = "caml_preprocess"

let _ =
  let action =
    if Array.length Sys.argv > 1 then
      List.assoc Sys.argv.(1) [
        ("-a", Ast);
        ("-i", Irgenerator);
                                  ("-p", Preprocessor);
        ("-c", Compile)
      ]
    else Compile
       in
       (match action with
          Ast ->
                        let lexbuf = Lexing.from_channel stdin in
                        let program = Parser.program Scanner.token lexbuf in
                        let listing = Ast.string_of_program program
                        in print_string listing
          | Irgenerator ->
                        let lexbuf = Lexing.from_channel stdin in
                        let program = Parser.program Scanner.token lexbuf in
                        let sast = Sastinjector.inject program in
                        let ir = Irgenerator.generate sast in
                        let output = Compile.compile ir
                        in print_string output
                | Preprocessor ->
                        let processed_code = preprocess() in
                        let lexbuf = Lexing.from_string processed_code in
                        let program = Parser.program Scanner.token lexbuf in
                        let sast = Sastinjector.inject program in
                        let ir = Irgenerator.generate sast in
                        let output = Compile.compile ir
                        in print_string output
          | Compile -> print_string "Heh heh heh" (*
                        let processed_code = preprocess() in
                        let lexbuf = Lexing.from_string processed_code in
                        let program = Parser.program Scanner.token lexbuf in
```

```
                                        let sast = Semantic_analyzer.evalprogram program in
                                        let ir = Irgenerator.generate sast in
                                        let output = Compile.compile ir
                                        in print_string output *)
            )
```

8.2 Test Cases

8.2.1 testcases.sh

```
#!/bin/bash

if [ $# -eq 0 ]
   then
   echo ""
   echo "TEST CASES"
   echo "----------"

   echo "Making the SPWAG language..."
   make

   testcases[0]='hello-world'
   testcases[1]='one-fish-two-fish'
   testcases[2]='presentation'
   testcases[3]='fiver'

   for testcase in "${testcases[@]}"
   do
      input='testcases/'$testcase'/test.spwag'
      output='testcases/'$testcase'/output/index.html'

      echo ""
      echo "Testing "$testcase"..."
      ./spwag -i < $input > $output
   done

   echo ""
   exit
fi

make
input='testcases/'$1'/test.spwag'
output='testcases/'$1'/output/index.html'

echo ""
echo "Testing "$1"..."
./spwag -i < $input > $output
```

8.2.2 fiver/test.spwag

8.2.3 hello-world/test.spwag

```
define slide main()
{
    box() {
        id("hello-world-text")
        text("Hello world!")
        padding-top(40)
        padding-bottom(20)
        font-size(40)
        width(100%)
    }
    box() {
        id("hello-world-image")
        image("../resources/cat.jpg")
    }
}
```

8.2.4 one-fish-two-fish/test.spwag

```
define slide main()
{
    box() {
        id("title-background")
        image("../resources/titlebg.png")
        position-x(0)
        position-y(0)
    }

    box() {
        id("title-text")
        text("One Fish Two Fish")
        padding-top(40)
        padding-bottom(20)
        font-size(40)
        width(100%)
    }

    box() {
        id("subtitle-text")
        text("by Dr. Seuss")
    }

    next(slide1)
}

define slide slide1()
```

```
{
    poem-text-box("one fish") {
        id("text")
    }
    fish-box("#EEE") {
        id("image")
    }

    prev(main)
    next(slide2)
}

define slide slide2()
{
    poem-text-box("two fish") {
        id("text")
    }

    fish-box("#1DC917") {
        id("image1")
        width(350)
        height(350)
    }

    fish-box("#1DC917") {
        id("image2")
        width(350)
        height(350)
    }

    prev(slide1)
    next(slide3)
}

define slide slide3()
{
    poem-text-box("red fish") {
        id("text")
    }
    fish-box("#E72E28") {
        id("image")
    }

    prev(slide2)
    next(slide4)
}

define slide slide4()
```

```
{
    poem-text-box("blue fish") {
        id("text")
    }
    fish-box("#70C1F8") {
        id("image")
    }

    prev(slide3)
    next(main)
}


define comp poem-text-box(mytext) isa box()
{
    text(mytext)
    padding-top(40)
    padding-bottom(20)
    width(100%)
}

define comp fish-box(color) isa box()
{
    image("../resources/fish.png")
    background-color(color)
}
```

8.2.5 presentation/test.spwag

```
define slide main()
{
    background-color(bgcolor())

    title("SPWAG") {
        id("spwag")
        position-x(0)
        position-y(100)
        font-size(180)
        width(900)
    }

    box() {
        id("block")
        position-x(0)
        position-y(350)
        background-color("#BD5532")
        width(900)
        height(250)
    }
    box() {
        id("subtitle")
        position-x(0)
        position-y(320)
        font("'Josefin Slab', serif")
        font-size(40)
        font-decoration("bold")
        text("SIMPLE PRESENTATION WEB APP GENERATOR")
        text-color("#BD5532")
        width(900)
    }

    box() {
        id("authors")
        position-x(0)
        position-y(430)
        font-size(30)
        text("Lauren Zou, Aftab Khan, Richard Chiou<br />Yunhe (John) Wang, Aditya Majumdar")
        text-color(fgcolor())
        width(900)
    }

    next(overview)
}

define slide overview()
{
    background-color(bgcolor())

    title("What is SPWAG?") {
        id("title")
    }
    title-underline() {
        id("title-underline")
    }

    content() {
        id("content")

        content-text("SPWAG is a simple language that streamlines presentation creation, saving you time
while producing beautiful (and functional) slideshows.")

##          box() {
            id("spwag-box")
            padding-top(30)
            padding-bottom(10)
            background-color(cyan())
            text("SPWAG")
            text-color(bgcolor())
            width(120)
        } ##
    }

    prev(main)
    next(motivation)
}

define slide motivation()
{
    background-color(bgcolor())

    title("Why SPWAG?") {
        id("title")
    }
    title-underline() {
        id("title-underline")
    }

    content() {
        id("content")

        content-text("Cross-Compatibility, Distributability, Interactivity"){
            id("content4")
            padding-top(30)
            font-size(36)
        }

        content-text("SPWAG presentations are accessible on any web-connected Desktop or Mobile
platform, regardless of operating system"){
            id("content3")
            padding-top(52)
            font-size(26)
            text-color("#CCC")
        }

        content-text("Compiled SPWAG may be integrated with existing web content, increasing content
visibility"){
            id("content2")
            padding-top(30)
            font-size(26)
        }

        content-text("SPWAG allows for interactive, real-time content manipulation"){
            id("content1")
            padding-top(30)
            font-size(26)
        }
    }

    prev(overview)
    next(implementation)
}

define slide implementation()
{
    background-color(bgcolor())

    title("How does SPWAG work?") {
        id("title")
    }
    title-underline() {
        id("title-underline")
    }

    content() {
        id("content")

        flow-box("Preprocessor") {
            id("preprocessor")
            position-x(0)
            position-y(204)
        }

        arrow("vert_up") {
            id("arrow0")
            padding-top(10)
            position-x(35)
            position-y(33)
        }

        flow-box("Scanner") {
            id("scanner")
            position-x(0)
            position-y(0)
        }

        arrow("horz") {
            id("arrow1")
            padding-top(10)
            position-x(97)
            position-y(0)
        }
```

```
flow-box("Parser") {
    id("parser")
    position-x(155)
    position-y(0)
}

arrow("vert") {
    id("arrow2")
    position-x(170)
    position-y(44)
}

flow-box("AST") {
    id("ast")
    position-x(155)
    position-y(102)
    background-color(accentcolor())
}

arrow("horz") {
    id("arrow3")
    padding-top(10)
    position-x(213)
    position-y(102)
}

flow-box("Semantic Analyzer") {
    id("semantic-analyzer")
    position-x(271)
    position-y(102)
}

arrow("vert") {
    id("arrow4")
    position-x(290)
    position-y(146)
}

flow-box("SAST") {
    id("sast")
    position-x(271)
    position-y(204)
    background-color(accentcolor())
}

arrow("horz") {
    id("arrow5")
    padding-top(10)
    position-x(341)
    position-y(204)
}

flow-box("IR Generator") {
    id("ir-generator")
    position-x(399)
    position-y(204)
}

arrow("horz") {
    id("arrow6")
    padding-top(10)
    position-x(541)
    position-y(204)
}

flow-box("IR") {
    id("ir")
    position-x(599)
    position-y(204)
    background-color(accentcolor())
}

arrow("horz") {
    id("arrow7")
    padding-top(10)
    position-x(638)
    position-y(204)
}

flow-box("Compile") {
    id("compile")
    position-x(696)
    position-y(204)
}

arrow("vert") {
    id("arrow8")
    padding-top(10)
    position-x(730)
    position-y(238)
}

flow-box("HTML, CSS, JavaScript") {
    id("html-css-javascript")
    position-x(678)
    position-y(306)
```

```
        }
    }

    prev(motivation)
    # next(workflow)
    next(demo)
}

## define slide workflow()
{
    background-color(bgcolor())

    title("Roles & Responsibilities") {
        id("title")
    }
    title-underline() {
        id("title-underline")
    }

    content() {
        id("content")

        content-text("Who did what? Put a chart here!")
    }

    prev(implementation)
    next(demo)
} ##

define slide demo()
{
    background-color(bgcolor())

    title("SPWAG Demo") {
        id("demo")
        position-x(0)
        position-y(120)
        font-size(180)
        width(900)
        on-click(changebackground())
    }

    prev(implementation)
    # prev(workflow)
    next(demo-explanation)
}

define slide demo-explanation()
```

```
{
    background-color(bgcolor())

    title("Demo") {
        id("title")
    }
    title-underline() {
        id("title-underline")
    }

    content() {
        id("content")
        image("../resources/demo.png")
    }

    prev(demo)
    next(lessons-learned)
}

define func changebackground()
{
    set(demo, "background-color", "#BD5532")
    set(implementation["content"]["html-css-javascript"], "background-color", "deeppink")
}

define slide lessons-learned()
{
    background-color(bgcolor())

    title("Lessons Learned") {
        id("title")
    }
    title-underline() {
        id("title-underline")
    }

    content() {
        id("content")

        content-text("Design decisions kept changing. Need a stable design plan from the beginning.") {
            id("content1")
            padding-top(30)
            font-size(26)
        }
        content-text("Could not find a mutually convenient weekly meeting time for entire group. Difficulty
splitting up tasks.") {
            id("content2")
            padding-top(30)
            font-size(26)
            text-color("#CCC")
        }
        content-text("Group work vs. individual work. Need to be more efficient with group meeting time.") {
            id("content3")
            padding-top(30)
            font-size(26)
        }
        content-text("Aspects of language were too ambitions and took more time than anticipated.") {
            id("content4")
            font-size(26)
            text-color("#CCC")
        }
    }

    prev(demo)
}

define comp title(text) isa box()
{
    position-x(70)
    position-y(40)
    text(text)
    text-color(accentcolor())
    font("'Josefin Slab', serif")
    font-size(60)
    font-decoration("bold")
}

define comp title-underline() isa box()
{
    position-x(70)
    position-y(90)
    background-color("#BD5532")
    width(760)
    height(5)
}

define comp content() isa box()
{
    position-x(50)
    position-y(150)
    width(800)
    height(420)
}

define comp content-text(text) isa box()
{
    text(text)
    text-color(fgcolor())
    font-size(30)
}

define comp flow-box(text) isa box()
{
    padding-top(10)
    padding-bottom(10)
    padding-left(10)
    padding-right(10)
    background-color(cyan())
    text(text)
    text-color(bgcolor())
    font-decoration("bold")
}

define comp arrow(direction) isa box()
{
    image("../resources/arrow_" + direction + ".png")
}

# DEFINE SOME COLORS
define func bgcolor()
{
    return "#2d2d2d" # Charcoal Gray
}

define func fgcolor()
{
    return "#EEE" # Light Gray
}

define func accentcolor()
{
    return "#E1B866" # Soft Goldenrod Yellow
}

define func cyan()
{
    return "#73C8A9"
}
```

8.2.6 Fiver

```
var box-size
var dim
```

```
define slide main()
{
        # Some global variables
        box-size = 100
        dim = 5
   background-color("#333")
        box() {
                id("upper")
                height(25)
                width(dim * box-size + 50)
        background-color("#333")
        }

        # Container box
        var r = 0
        while (r < dim)
        {
                box() {
                        var c = 0
                        id("row"+r)
                        height(box-size)
                        width(dim * box-size + 50)
                        while (c < dim)
                        {
                                square(r, c)
                                c = c + 1
                        }
                }
                r = r + 1
        }

        # Numbering box
        box() {
                id("numbers")
                width(dim * box-size + 50)
                text("Red: 0 White: " + dim * dim)
                text-color("white")
        }
}

define comp square(x, y) isa box()
{
   id("square" + x + y)
   background-color("white")
   width(box-size)
   height(box-size)

        on-click(changeneighbors(x, y, dim))
}

define func changeneighbors(x, y, ndim)
{
        # Update
        changecolor(x, y)
        if (x > 0)
                changecolor(x-1, y)
        if (x < ndim - 1)
                changecolor(x+1, y)
        if (y > 0)
                changecolor(x, y-1)
        if (y < ndim + 1)
                changecolor(x, y+1)

        # Detect
        var row = 0
        var num = 0
        while row < ndim
        {
                var col = 0
                while col < ndim
                {
                        var rowid = "row"+row
                        var squareid = "square"+row+col
                        var current = get(main[rowid][squareid], "background-color")

                        if (current == "rgb(255, 255, 255)")
                                num = num + 1
                        col = col + 1
                }
                row = row + 1
        }
        set(main["numbers"], "text", "Red: " + (ndim * ndim - num) + " White: " + num)

}

define func changecolor(x, y)
{
        var rowid = "row"+x
        var squareid = "square"+x+y
        var current = get(main[rowid][squareid], "background-color")

        if (current == "rgb(255, 255, 255)")
                set(main[rowid][squareid], "background-color", "red")
        else
                set(main[rowid][squareid], "background-color", "white")
}
```

8.3 Config CSS and JS

8.3.1 config.css

```
@import url(http://fonts.googleapis.com/css?family=Josefin+Slab:400,700);
@import url(http://fonts.googleapis.com/css?family=Ubuntu:400,700,400italic,700italic);

* {
  margin: 0px;
  padding: 0px;
  border: 0px;
  box-sizing: border-box;
}

html, body {
  padding: 10px 0px;
  background: #EEE;
  font: 20px 'Ubuntu', sans-serif;
  width: 100%;
  height: 100%;
}

div.slide {
  margin: 0px auto;
  padding: 0px 50px;
  position: relative;
  background: white;
  text-align: center;
  box-shadow: 0px 0px 10px rgba(0, 0, 0, 0.1);
  width: 900px;
  height: 600px;
  transition: all 0.1s linear 0s;
}

div.box {
  display: inline-block;
  max-width: 100%;
  max-height: 100%;
}

div.box img {
  display: block;
  max-width: 100%;
  max-height: 100%;
}

a.prev, a.next {
    display: block;
    position: absolute;
    top: 0px;
    width: 50px;
    height: 100%;
    transition: background 0.1s linear 0s;
}

a.prev {
    left: 0px;
}

a.next {
    right: 0px;
}

a.prev:hover, a.next:hover {
    background: rgba(0, 0, 0, 0.02);
}
```

8.3.2 config.js

```
$(document).ready(function() {
    // Display current slide on load
    slideId = getCurrentSlide();
    displaySlide(slideId);

    // Detect hash change
    $(window).bind('hashchange', function(e) {
        slideId = getCurrentSlide();
        displaySlide(slideId);
    });

    // Handle keypresses
    $(window).keydown(function(e) {
        switch(e.keyCode) {
            case 39: next(); break; // right arrow
            case 37: prev(); break; // left arrow
            case 32: next(); break; // spacebar
        }

        function prev() {
            var currentSlide = $('.slide:visible');
            if (currentSlide.children('a.prev').length == 1) {
                window.location.hash = currentSlide.children('a.prev').attr('href');
            }
        }

        function next() {
            var currentSlide = $('.slide:visible');
            if (currentSlide.children('a.next').length == 1) {
                window.location.hash = currentSlide.children('a.next').attr('href');
            }
        }

        return false;
    });
});

function getCurrentSlide() {
    var hash = window.location.hash;
    return hash === ''? 'main' : hash.substring(1);
}

function displaySlide(slideId) {
    $('.slide:visible:not(#' + slideId + ')').hide();
    $('#' + slideId).show();
}
```