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

Introduction

The Single Presentation Web App Generator (SPWAG) combines HTML, CSS, and JavaScript to generate a web-based slide-show presentation. 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 light-weight as well as compatible across all browsers and platforms.

Lexical Conventions

Identifiers

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

Keywords

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

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

Data Types and Variables

SPWAG contains 3 primitive data types: *integers*, *booleans* and *strings*. All variables are 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.

Integer

Integers can fall under several different integer types: regular integer, pixel, and percentage. A regular integer literal does not require a suffix, but a pixel integer literal should always be followed by a 'px' (e.g. 300px) and 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.

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**.

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. The escape sequence **\”** inserts the **“** character, \\ inserts the **** character, and the escape sequence **\n** inserts a new line into the string.

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

In HSLA:

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

In standard HTML color:

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

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.

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 we define 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

>	greater than operator; can only compare variables of the same type
=	assignment operator
==	equal to
!=	not equal to
+	*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, pixels, and percents; subtraction for integers, pixels, and percents may only occur with two variables of the same data type
*****	multiplication; only works for integers, pixels, and percents; multiplication for integers, pixels, and percents may only occur with two variables of the same data type
/	division; only works for integers, pixels, and percents; division for integers, pixels, and percents may only occur with two variables of the same data type
|	or
&	and

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. ##
```

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.

Function Calls

To call a function, the following syntax must be used: ***`identifier(parameters)`***. The identifier must be previously defined as a function, while ***`parameters`*** 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`**.

Native Functions

SPWAG includes several native functions for two primary purposes:

1. To transition from one slide to another
2. To handle changes and animations in each slide

The **`change-slide(slide)`** function, when called, changes the current visible slide to the one given in the parameter.

The **`hide(string`** ***`component-id`*****`)`** and **`show(string`** ***`component-id`*****`)`** functions hide and reveal the component whose ID is given as the parameter. Components that can be revealed or hidden include images and text fields.

The **`on-click(function)`** function calls the function given in its argument when the user clicks on the component that **`on-click(function)`** is under. This function is required to be a function under a component. Possible functions that can be called include the other native functions of SPWAG or user-defined functions to open a link to a new webpage.

The **`on-press(key, function)`** function has similar functionality to the **`on-click(function)`** function, but rather than performing the function when a click is registered, the **`on-press(key, function)`** binds the function to a keypress.

Lastly, 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`*****`, component`** ***`parent-component`*****`)`**, **`get(string`** ***`attr-name`*****`, component`** ***`parent-component`*****`)`**, **`get(string`** ***`component-id`*****`, string`** ***`slide-name`*****`)`**, and the **`get(string`**

***attr-name*, string *slide-name*)** functions are accessor functions to retrieve specific components and attributes contained within specific components respectively. For example, consider the following code:

```
slide main()
  text-color("blue")

  box("my-box")
    text("Hello world!")
    text-color("red")

    box("other-box")
      text("Yay world!")
      text-color("green")

      box("last-box")
        text-color("yellow")
end
```

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

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

The box component with the id "my-box" can be retrieved using the **get(string *component-id*, string *slide-name*)** function:

```
var my-box = get("my-box", "main") # returns the my-box box component
```

The text color attribute of "my-box" can be retrieved with the **get(string *attr-name*, component *parent-component*)** function:

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

The box component with the id "other-box" can be retrieved with the **get(string *component-id*, component *parent-component*)** function:

```
var other-box = get("other-box", get("my-box", "main")) # returns other-box
```

The text color attribute of "other-box" can be retrieved with the **get(string *component-id*, string *slide-name*, string *attr-name*)** function. Note the following nested **get()** syntax:

```
var other-box-text-color = get(“text-color”, get(“other-box”, get(“my-box”, “main”)))
    # returns “green”
```

Finally, the text color attribute of “last-box” can be retrieved with the **get(string *component-id*, string *slide-name*, string *attr-name*)** function. Note the following nested **get()** syntax:

```
var last-box-text-color = get(“text-color”, get(“last-box”, get(“other-box”,
    get(“my-box”, “main”)))) # returns “yellow
```

The following table summarizes the native functions of SPWAG:

change-slide(string *slide-id*)	changes the currently visible to **slide** to another **slide** which is specified by the ***slide-id***
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
hide(string *component-id*)	hides the component given by the ***component-id***
show(string *component-id*)	shows the component given by the ***component-id*** if the component was previously hidden
random(integer *i*)	randomly generates a random integer between 0 and *i*
get(string *component-id*, component *parent-component*)	returns the component whose ID is ***component-id*** found within the ***parent-component***
get(string *attr-name*, component *parent-component*)	returns the value of the specified attribute ***attr-name*** of the component ***parent-component***
get(string *component-id*, string *slide-name*)	returns the component whose ID is ***component-id*** found as a top level component of the slide ***slide-name***. Note, this component must be the a top level component of the slide (cannot have any other component containing it).
get(string *attr-name*, string *slide-name*)	returns the value of the specified attribute ***attr-name*** of the slide ***slide-name***. Note, this attribute must be an attribute of a slide.

Function Definition

`slide main`	defines the main **`slide`** of the slideshow; this is a required function in all SPWAG programs
`slide some-slide`	defines a custom **`slide`** in the slideshow

`define comp my-comp(string` ***`component-id`*****`,`** ***`my-comp-parameters`*****`) isa spwag-comp(string`** ***`component-id`*****`,`** ***`spwag-comp-parameters`*****`)`**
`...`
`end`
	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`**
	
	note that ***`component-id`*** is the same parameter in both **`my-comp`** and **`spwag-comp`**

`define attr my-attr(`***`parameters`*****`)`**
`...`
`end`
	defines a custom attribute

`define my-funct()`
`...`
`end`
	defines a custom function

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 the first parameter during component creation. Any component created with the same ID as a previous component will overwrite that component (the handle to that component will be lost).

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.`

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:

```
slide my-slide()
  ...
end
```

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`**:

```
slide main()
  ...
end
```

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.

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.

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(pixel` *`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(pixel 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

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(“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 example, to change the background color and text color, one can write:

```
var box-to-change = get("my-box")
box-to-change
  background-color("blue")
  text-color("green")
```

Program Structure

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***

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:

```
slide main()

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

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

```
end
```

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.

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`
`  ...`
`end`
	if-else conditional

`while <condition>`
`  ...`
`end`
	while loop

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

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.

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.

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.

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.

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.

Scoping

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.

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.

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.

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.