The Sway Reference Manual

John C. Lusth

February 7, 2010
# Contents

1 Starting Out ................................................. 9
   1.1 Installing Sway ........................................... 9
   1.2 Running Sway ........................................... 9

2 Primitives .................................................... 11
   2.1 Integers .................................................. 11
   2.2 Real Numbers .......................................... 12
   2.3 Strings .................................................. 13
   2.4 Symbols ................................................ 13

3 Combining Primitives ......................................... 15
   3.1 Numeric operators ....................................... 15
   3.2 About whitespace ...................................... 17
   3.3 Comparing things ...................................... 17
   3.4 Combining comparisons .................................. 18

4 Precedence and Associativity ................................ 21
   4.1 Precedence ............................................... 21
   4.2 Associativity ........................................... 22

5 Variables and Environments .................................. 23
   5.1 Variables ............................................... 23
   5.2 Environments .......................................... 25
   5.3 Variable naming ....................................... 27

6 Assignment ................................................... 29
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>Precedence and Associativity of Assignment</td>
<td>31</td>
</tr>
<tr>
<td>6.2</td>
<td>ADVANCED TOPIC: How it works</td>
<td>31</td>
</tr>
<tr>
<td>6.3</td>
<td>ADVANCED TOPIC: Assigning to Thunks</td>
<td>32</td>
</tr>
<tr>
<td>7</td>
<td>Functions</td>
<td>35</td>
</tr>
<tr>
<td>7.1</td>
<td>Encapsulating a series of operations</td>
<td>35</td>
</tr>
<tr>
<td>7.2</td>
<td>Passing arguments</td>
<td>37</td>
</tr>
<tr>
<td>7.3</td>
<td>Creating functions on the fly</td>
<td>38</td>
</tr>
<tr>
<td>7.4</td>
<td>Using objects</td>
<td>39</td>
</tr>
<tr>
<td>7.5</td>
<td>Functions versus operator</td>
<td>41</td>
</tr>
<tr>
<td>8</td>
<td>Using Files</td>
<td>43</td>
</tr>
<tr>
<td>8.1</td>
<td>Your first program</td>
<td>43</td>
</tr>
<tr>
<td>8.2</td>
<td>Vim and Sway</td>
<td>43</td>
</tr>
<tr>
<td>8.3</td>
<td>A Neat Macro</td>
<td>44</td>
</tr>
<tr>
<td>8.4</td>
<td>Writing Sway Programs</td>
<td>44</td>
</tr>
<tr>
<td>8.5</td>
<td>Order of definitions</td>
<td>44</td>
</tr>
<tr>
<td>8.6</td>
<td>Including code</td>
<td>45</td>
</tr>
<tr>
<td>8.7</td>
<td>Multiple includes</td>
<td>45</td>
</tr>
<tr>
<td>8.8</td>
<td>Includes are considered definitions</td>
<td>46</td>
</tr>
<tr>
<td>8.9</td>
<td>Encapsulated includes</td>
<td>46</td>
</tr>
<tr>
<td>9</td>
<td>More about Functions</td>
<td>49</td>
</tr>
<tr>
<td>9.1</td>
<td>Predefined Functions</td>
<td>49</td>
</tr>
<tr>
<td>9.2</td>
<td>Function syntax</td>
<td>50</td>
</tr>
<tr>
<td>9.3</td>
<td>Function Objects</td>
<td>50</td>
</tr>
<tr>
<td>9.4</td>
<td>Calling Functions</td>
<td>51</td>
</tr>
<tr>
<td>9.5</td>
<td>Scope</td>
<td>52</td>
</tr>
<tr>
<td>9.6</td>
<td>ADVANCED TOPIC: Environments</td>
<td>54</td>
</tr>
<tr>
<td>9.7</td>
<td>Returning from functions</td>
<td>55</td>
</tr>
<tr>
<td>10</td>
<td>Debugging</td>
<td>57</td>
</tr>
</tbody>
</table>
14.1 Reading ......................................................... 87
14.2 Writing ......................................................... 88
14.3 Pretty printing ............................................... 89
14.4 Formatting ..................................................... 89
14.5 Testing for end of file ....................................... 90
14.6 Pushing back a character ................................... 90

15 Arrays and Lists ................................................ 91
15.1 Arrays .......................................................... 91
   15.1.1 Getting array elements .................................. 91
   15.1.2 Setting array elements .................................. 92
   15.1.3 Allocating empty arrays ................................. 93
15.2 Lists ........................................................... 93
15.3 Mixing arrays and lists ...................................... 94
15.4 Shallow versus deep copies .................................. 94
15.5 Changing the tail of a list ................................... 98
15.6 Inserting into the middle of a list ............................ 98
15.7 Objects ........................................................ 99

16 Objects .......................................................... 101
16.1 Adding Methods ............................................... 103
16.2 Objects and Types ............................................. 103
16.3 A formal view of object-orientation .......................... 104
16.4 Using the objects library .................................... 105

17 Inheritance ....................................................... 107
17.1 Single inheritance .............................................. 107
   17.1.1 Extension inheritance ..................................... 107
   17.1.2 Reification inheritance .................................... 108
   17.1.3 Variation inheritance ..................................... 109
17.2 Multiple inheritance ......................................... 109
17.3 Retrieving overridden functions ........................................ 110

18 Exceptions .............................................................. 111
18.1 Handling exceptions .................................................. 111
18.2 Throwing exceptions .................................................. 112
18.3 Try catch blocks ........................................................ 112
18.4 Returns are exceptions .............................................. 112

19 Lazy Evaluation .......................................................... 115
19.1 Infinite lists ............................................................. 115
19.2 Memoization ............................................................ 117
19.3 Simulating pass-by-reference ........................................ 117

20 Variadic Functions ....................................................... 119
20.1 Calling a variadic function recursively ............................. 119
20.2 Variadic functions with lazy evaluation ......................... 120

21 Reflection .................................................................... 121
21.1 The type function ....................................................... 121
21.2 The is function .......................................................... 121
21.3 Decomposition of functions ......................................... 122
21.4 Decomposition of arbitrary code .................................. 123

22 Returning Local Functions ........................................... 125
22.1 Why returning local functions works ............................ 125
22.2 Using local functions in place of objects ....................... 126

23 Sway Libraries ............................................................ 129
Chapter 1

Starting Out

A word of warning: if you require fancy graphically oriented development environments, you will be sorely disappointed in the Sway Programming System. Sway is programming at its simplest: a prompt at which you type in expressions which the Sway interpreter evaluates. Of course, you can create Sway programs using your favorite text editor (mine is vim). Sway can be used as a scripting language, as well.

1.1 Installing Sway

Procedures for downloading and installing Sway can be found at http://sway.cs.ua.edu.

1.2 Running Sway

In a terminal window, simply type the command:

```
sway
```

and press the <Enter> key. You should be rewarded with the Sway prompt.

```
sway>
```

At this point, you are ready to proceed to next chapter.
Chapter 2

Primitives

Sway works by figuring out the meaning or value of some code. This is true for the tiniest pieces of code to the largest programs. The process of finding out the meaning of code is known as *evaluation*.

The things whose values are the things themselves are known as *primitives*. The primitives of Sway can be categorized by the following types: *integers, real numbers, strings, and symbols*.

Sway (or more correctly, the Sway interpreter) responds to primitives by giving the type of the value plus the value itself.

Here are examples of each of the types:

```
sway> 3;
INTEGER: 3

sway> -4.9;
REAL: -4.9

sway> "hello";
STRING: "hello"

sway> :age;
SYMBOL: :age
```

Let’s examine the four types in more detail.

2.1 Integers

Integers are numbers without any fractional parts. Examples of integers are:

```
sway> 3;
INTEGER: 3

sway> -5;
INTEGER: -5

sway> 0;
INTEGER: 0
```
Integers must begin with a digit or a minus sign. The initial minus sign must immediately be followed by a digit.

2.2 Real Numbers

Reals are numbers that do have a fractional part (even if that fractional part is zero!). Examples of real numbers are:

```sway
sway> 3.2;
REAL: 3.20000000000

sway> 4.0;
REAL: 4.00000000000

sway> 5.;
REAL: 5.00000000000

sway> 0.3;
REAL: 0.30000000000

sway> .3;
REAL: 0.30000000000

sway> 3.0e-4;
REAL: 0.00030000000

sway> 3.0e4;
REAL: 30000.0000000

sway> .000000987654321
REAL: 9.8765432e-07
```

Real numbers must start with a digit or a minus sign or a decimal point. An initial minus sign must immediately be followed by a digit or a decimal point. An initial decimal point must immediately be followed by a digit. Sway accepts real numbers in scientific notation. For example, 3.0 \( \times 10^{-11} \) would be entered as 3.0e-11. The ‘e’ stands for exponent and the 10 is understood, so e-11 means multiply whatever precedes the e by \( 10^{-11} \).

The Sway interpreter only displays between six and eleven significant digits, depending on the size of the number, so 1.234567898765432 would display as:

```sway
sway> 1.234567898765432;
REAL: 1.2345678988
```

Don’t be fooled. Although the result looks like 1.2345678988, it is not. The following code tests if the two numbers are equal:

```sway
sway> 1.234567898765432 == 1.2345678988;
SYMBOL: :false
```

Numbers greater than \( 10^6 \) and less than \( 10^{-6} \) are displayed in scientific notation.
2.3 Strings

To display more (or fewer) significant digits, the \texttt{fmt} function can be used. Here’s how to display 15 digits after the decimal point:

\begin{verbatim}
sway> fmt("%.15f",1.234567898765432);
STRING: 1.234567898765432
\end{verbatim}

The \texttt{fmt} function uses the same format codes as the \texttt{printf} function in the C programming language.

Don’t try and display more than 16 significant digits; the least significant digits will not be accurate.

2.3 Strings

Strings are sequences of characters delineated by double quotation marks:

\begin{verbatim}
sway> "hello, world!";
STRING: "hello, world!"

sway> "x\nx";
STRING: "x\nx"

sway> "\"z\""
STRING: "\"z\"

sway> "";
STRING: ""
\end{verbatim}

Characters in a string can be escaped with the backslash character, which changes the meaning of some characters. For example, the character \texttt{n}, in a string refers to the letter \texttt{n} while the character sequence \texttt{\n} refers to the \texttt{newline} character. A backslash also changes the meaning of the letter \texttt{t}, converting it into a tab character. When other characters are escaped, their meanings are not changed. Thus \texttt{\z} is equivalent to the letter \texttt{z}. To include a backslash character in a string, escaped it with a backslash, \texttt{\\}. Note that Sway, when asked the value of strings that contain newline and tab characters, displays them as escaped characters. When newline and tab characters in a string are printed in a program, however, they are displayed as actual newline and tab characters, respectively. Double quotes can be embedded in a string by escaping them with backslashes. A string with no characters between the double quotes is known as an empty string.

Unlike some languages, there is no character type in Sway. A single character \texttt{a}, for example, is entered as the string \texttt{"a"}.

2.4 Symbols

Sway symbols are collections of certain characters beginning with a colon:

\begin{verbatim}
sway> :abc;
SYMBOL: :abc

sway> :a+b;
SYMBOL: :a+b
\end{verbatim}

When symbols are printed in a program, the colon is omitted from the output. Symbols may contain any of the printable characters except parentheses, braces, semicolons, and commas. The colon marks the beginning
of the symbol while whitespace (newlines, tabs, and spaces) and the prohibited characters signal the end of the symbol. As we shall see later, symbols are useful for setting up symbolic constants and for passing messages.

There are a few special symbols in Sway, two of which are :true and :false. These symbols are known as the boolean values and are used to guide the flow of a program. The term boolean is derived from the last name of George Boole, who, in his 1854 paper *An Investigation of the Laws of Thought, on which are founded the Mathematical Theories of Logic and Probabilities*, laid one of the cornerstones of the modern digital computer. The so-called boolean logic or boolean algebra is concerned with the rules of combining truth values (i.e., true or false). As we will see, knowledge of such rules will be important for making Sway programs behave properly. In particular, boolean expressions will be used to control conditionals and loops.

Another special symbol is :null. This symbol is used to indicate the end of lists and arrays; it also indicates an ‘not yet created’ object.
Chapter 3

Combining Primitives

Like the primitives themselves, combinations of primitives are also expressions. For example, suppose you have forgotten your times table and aren’t quite sure whether 8 times 7 is 54 or 56. We can ask Sway, presenting the interpreter with the expression:

```
sway> 8 * 7;
INTEGER: 56
```

As before, the semicolon signals the end of the expression. The multiplication sign * is known as an operator, as it operates on the 8 and the 7, producing an equivalent primitive value. The 8 and the 7 are known as operands. It seems that the actual names of various operands are not being taught anymore, so for nostalgia’s sake, here they are. The operand to the left of the multiplication sign (in this case the 8) is known as the multiplicand. The operand to the right (in this case the 7) is known as the multiplier. The result is known as the product.

The operands of the other basic operators have special names too. For addition, the left operand is known as the augend and the right operand is known as the addend. The result is known as the sum. For subtraction, the left operand is the minuend, the right the subtrahend, and the result as the difference. Finally for division (and I think this is still taught), the left operand is the dividend, the right operand is the divisor, and the result is the quotient.

In general, operators are separated from their operands by spaces, tabs, or newlines, collectively known as whitespace. It would be an error to enter the expression `8*7` as the times operator does not have the requisite whitespace surrounding it.

In fact, Sway always takes in an expression and displays an equivalent primitive expression (e.g., integer or real). All Sway operators are binary, meaning they operate on exactly two operands. We first look at the numeric operators.

3.1 Numeric operators

If it makes sense to add two things together, you can probably do it in Sway using the + operator. For example:

```
sway> 2 + 3;
```

1Computer Scientists, when they have to write their annual reports, often refer to the things they are reporting on as darkspace. It’s always good to have a lot of darkspace!
INTEGER: 5

sway> 1.9 + 3.1;
REAL: 5.00000000000

sway> "hello" + "world"
STRING: helloworld

One can see that if one adds two integers, the result is an integer. If one does the same with two reals, the result is a real. The same is true for adding strings to strings. Things get more interesting when you add things having different types. Adding an integer and a real (in any order) always yields a real.

sway> 2 + 3.3;
REAL: 5.30000000000

sway> 3.3 + 2;
REAL: 5.30000000000

Adding an string to an integer (with an augend integer) yields an integer. The addend string is interpreted as an integer, if possible:

sway> 2 + "3";
INTEGER: 5

sway> 2 + "3.3";
INTEGER: 5

sway> 2 + "hello";
INTEGER: 2

Adding an integer to a string (with an augend string) yields a string. The addend integer is interpreted as string:

sway> "hello" + 2;
INTEGER: "hello2"

sway> "3" + 2;
STRING: "32"

In general, when adding two things (with the exception of integers and reals), the thing on the left wins, with respect to the type of the result.

Subtraction, multiplication, and division of numbers follow the same rules as addition. However, these operators, as defined, do not work for strings.\(^2\)

Of special note is the division operator with respect to integer operands. Consider evaluating the following expression:

\[^2\text{This does not mean you can never subtract strings. Later you will learn how to override Sway’s minus operator so that you can subtract strange things to your heart’s content.}\]
If one asked the Sway interpreter to perform this task, the result would be 7, as expected. However, if asked
the interpreter to evaluate the expression...

the result would be 2, not 2.8. Consistent with the rule above, integer division always returns an integer and
if there is a fractional part, it is discarded. Note that all six of the following expressions evaluate to 2.8:

The last three illustrate the use of a conversion function. In the above cases, the integers are converted to
reals before the division occurs.

The complement to integer division is the modulus operator %. While the result of integer division is the
quotient, the result of the modulus operator is the remainder. Thus

evaluates to 4 since 4 is left over when 5 is divided into 14. To check if this is true, one can ask the interpreter
to evaluate:

This complicated expression asks the question ‘is it true that the quotient times the divisor plus the remainder
is equal to the original dividend?’ The Sway interpreter will respond that, indeed, it is true. The reason
for the parentheses is due to a lack of precedence among the Sway arithmetic operators and is explained in
more detail in the next chapter.

3.2 About whitespace

Whitespace (tab, newline, and space) in a computer program has two purposes. The first is to make
expressions in the language pleasing to the eye. The second is whitespace often signals the end of a primitive.
Unlike many programming languages, Sway uses whitespace to signal the end of operators. Thus, 8 * 7 is
interpreted as two numbers separated by the * operator while 8*7 (no spaces) will be considered a ‘bad
number’. We will learn about variables in a bit.

3.3 Comparing things

Remember the boolean primitives, :true and :false? We can use the boolean comparison operators to
generate such values. For example, we can ask if 3 is less than 4:
CHAPTER 3. COMBINING PRIMITIVES

sway> 3 < 4;
SYMBOL: :true

The interpreter's response says that, indeed, 3 is less than 4. If it were not, the interpreter would respond with :false. Besides < (less than), there are other boolean comparison operators: <= (less than or equal to), > (greater than), => (greater than or equal to), == (equal to), and != (not equal to).

Besides integers, we can compare reals with reals and strings with strings using the less-than-or-greater-than-like operators. In general, it is illegal to compare integers or reals with strings.

Any Sway type can be compared with any other type with the == and != comparison operators. If an integer is compared with a real with these operators, the integer is converted into a real before the comparison is made. In other cases, comparing different types with == will yield a value of :false. Conversely, comparing different types with != will yield :true (the exception, as above, being integers compared with reals). If the types match, == will yield true only if the values match as well. The operator != behaves accordingly.

3.4 Combining comparisons

We can combine comparisons with the boolean logical connectives && (AND) and || (OR).

sway> 3 < 4 && 4 < 5;
SYMBOL: :true

sway> 3 < 4 || 4 < 5;
SYMBOL: :true

sway> 3 < 4 && 5 < 4;
SYMBOL: :false

sway> 3 < 4 || 5 < 4;
SYMBOL: :true

The first interaction asks if both the expression 3 < 4 and the expression 4 < 5 are true. Since both are, the interpreter responds with :true. The second interaction asks if at least one of the expressions is true. Again, the interpreter responds with :true. The difference between && and || is illustrated in the last two interactions. Since only one expression is true (the latter expression being false) only the || operator yields a true value.

There is one more boolean logic operation, called not. It simply reverses the value of the expression to which it is attached. The not operator can only be called as a function (since it is not a binary operator). Since you do not yet know about functions, I'll show you what it looks like but won't yet explain its actions.

sway> not(3 < 4 && 4 < 5);
SYMBOL: :false

sway> not(3 < 4 || 4 < 5);
SYMBOL: :false

sway> not(3 < 4 && 5 < 4);
SYMBOL: :true

sway> not(3 < 4 || 5 < 4);
SYMBOL: :false
Note that we attached *not* to each of the previous expressions involving the logical connectives. Note also that the response of the interpreter is reversed from before in each case.
Chapter 4

Precedence and Associativity

4.1 Precedence

Precedence (partially) describes the order in which operators, in an expression involving different operators, are evaluated. In Sway, the expression

\[ 3 + 4 < 10 - 2 \]

evaluates to true. In particular, 3 + 4 and 10 - 2 are evaluated before the <, yielding 7 < 8, which is indeed true. This implies that + and - have higher precedence than <. If < had higher precedence, then 4 < 10 would be evaluated first, yielding 3 + true - 2, which is nonsensical.

Note that precedence is only a partial ordering. We cannot tell, for example whether 3 + 4 is evaluated before the 10 - 2, or vice versa. Upon close examination, we see that it does not matter which is performed first as long as both are performed before the expression involving < is evaluated. It is common to assume that the left operand is evaluated before the right operand. For the boolean connectives && (AND) and || (OR), this is indeed true. But for other operators, such an assumption can lead you into trouble. You will learn why later. For now, remember never, never, never depend on the order in which operands are evaluated!

The lowest precedence operator in Sway is the assignment operator which is described later. Next come the boolean connectives && and ||. At the next higher level are the boolean comparatives,<, <=, >, >=, ==, and !=. After that come the arithmetic operators +, -, *, /, ^, and % . Finally, at the highest level of precedence is the selection, or dot, operator (the dot operator is a period or full-stop). Higher precedence operations are performed before lower precedence operations. Functions which are called with operator syntax have the same precedence level as the mathematical operators.

Unlike the algebraic precedence you learned in grammar school, multiplication and division are at the same precedence level as addition and subtraction in Sway. Thus, in combinations of these operators, operations to the left are performed before operations to the right. To force multiplication and division to be performed prior to addition and subtraction, parentheses must be used. For example:

```
sway> 5 + 2 * 3 // 5 plus 2 is 7, 7 times 3 is 21
INTEGER: 21
```

but:

```
sway> 5 + (2 * 3) // 2 * 3 is 6, 5 + 6 is 11
```
4.2 Associativity

Associativity describes how multiple expressions connected by operators at the same precedence level are evaluated. All the operators, with the exception of the assignment operator, are left associative. For example the expression $5 - 4 - 3 - 2 - 1$ is equivalent to $(((5 - 4) - 3) - 2) - 1)$. For a left-associative structure, the equivalent, fully parenthesized, structure has open parentheses piling up on the left. If the minus operator was right associative, the equivalent expression would be $(5 - (4 - (3 - (2 - 1))))$, with the close parentheses piling up on the right. For a commutative operator, it does not matter whether it is left associative or right associative. Subtraction, however, is not commutative, so associativity does matter. For the given expression, the left associative evaluation is $-5$. If minus were right associative, the evaluation would be $3$. The only operator that is right associate is the assignment operator. You will see why in the assignment chapter.
Chapter 5

Variables and Environments

Suppose you found an envelope lying on the street and on the front of the envelope was printed the name `numberOfDog'sTeeth`. Suppose further that you opened the envelope and inside was a piece of paper with the number 42 written upon it. What might you conclude from such an encounter? Now suppose you kept walking and found another envelope labeled `meaningOfLifeUniverseEverything` and, again, upon opening it you found a slip of paper with the number 42 on it. Further down the road, you find two more envelopes, entitled `numberOfDotsOnPairOfDice` and `StatuteOfLibertyArmLength`, both of which contain the number 42.

Finally, you find one last envelope labeled `sixTimesNine` and inside you, yet again, find the number 42. At this point, you’re probably thinking ‘somebody has an odd affection for the number 42’ but then the times table that is stuck somewhere in the dim recesses of your brain begins yelling at you saying ‘54! It’s 54!’.

After this goes on for an embarrassingly long time, you realize that 6 * 9 is not 42, but 54. So you cross out the 42 in the last envelope and write 54 instead and put the envelope back where you found it.

This strange little story, believe it or not, has profound implications for writing programs that both humans and computers can understand. For programming languages, the envelope is a metaphor for something called a variable, which can be thought of as a label for a place in memory where a primitive value can reside. In many programming languages, one can change the value at that memory location, much like replacing the contents of an envelope.

1 Languages that do not allow changes to a variable are called functional languages. Sway is an ‘impure’ functional language since it is mostly functional but allows for variable modification.

2 Another fundamental concept is analogy and if you understand the purpose of the envelope story after reading this section, you’re well on your way to being a computer scientist!

5.1 Variables

Most likely, you’ve encountered the term variable before. Consider the slope-intercept form of an algebraic equation of a particular line:

\[ y = 2x - 3 \]

You probably can tell from this equation that the slope of this line is 2 and that it intercepts the y-axis at -3. But what role do the letters \( y \) and \( x \) actually play? The names \( x \) and \( y \) are placeholders and stand for the \( x\)- and \( y\)-coordinates of any conceivable point on that line. Without placeholders, the line would have to be described by listing every point on the line. Since there are an infinite number of points, clearly an exhaustive list is not feasible. As you learned in your algebra class, the common name for a place holder for a specific value is the term variable.
One can generalize the above line resulting in an equation that describes every line.³

\[ y = mx + b \]

Here, the variable \( m \) stands for the slope and \( b \) stands for the \( y \)-intercept. Clearly, this equation was not dreamed up by a computer scientist since a cardinal rule is to choose good names for variables, such as \( s \) for slope and \( i \) for intercept. But alas, for historical reasons, we are stuck with \( m \) and \( b \).

The term variable is also used in most programming languages, including Sway, and the term has roughly the equivalent meaning. The difference is programming languages use the envelope metaphor while algebraic meaning of variable is an equivalence to a value.⁴ The difference is purely philosophical and not worth going into at this time. Suppose you found three envelopes, marked \( m \), \( x \), and \( b \), and inside those three envelopes you found the numbers 6, 9, and -12 respectively. If you were asked to make a \( y \) envelope, what number should you put inside? If the number 42 in the sixTimesNine envelope in the previous story did not bother you (e.g., your internal times table was nowhere to be found), perhaps you might need a little help in completing your task. We can have Sway calculate this number with the following dialog:

```
sway> var m = 6;
INTEGER: 6

sway> var x = 9;
INTEGER: 9

sway> var b = -12;
INTEGER: -12

sway> var y = m * x + b;
INTEGER: 42

sway> y;
INTEGER: 42
```

One creates variables in Sway with the \textit{var} keyword.⁵

The creation of a variable is known as a \textit{declaration} or \textit{definition}. In Sway, the token after the keyword \textit{var} is the variable being defined and the value after the equals sign is the initial value of the variable. The last definition shows that the initial value can be an expression which refers to other variables. The Sway interpreter, when asked to compute the value of an expression containing variables, goes to those envelopes (so to speak) and retrieves the values stored there. Note also that Sway requires the use of the multiplication sign to multiply the slope \( m \) by the \( x \) value. In the algebraic equation, the multiplication sign is elided, but is required here.

Here are some more examples of variable creation using the \textit{var} keyword:

³The third great fundamental concept in computer science is generalization. In particular, computer scientists are always trying to make things more abstract and more general (but not overly so). The reason is that software/systems/models exhibiting the proper levels of abstraction and generalization are much much easier to understand and modify. This is especially useful when you are required to make a last second change to the software/system/model.

⁴Even the envelope metaphor can be confusing since it implies that two variables having the same value must each have a copy of that value. Otherwise, how can one value be in two envelopes at the same time? For simple primitives, copying is the norm. For more complex objects, the cost of copying would be prohibitive. The solution is to storing the \textit{address} of the object, instead of the object itself, in the envelope. Two variables can now ‘hold’ the same object since the address is copied.

⁵A keyword is a token that cannot be used as the name of a variable. In actually, \textit{var} and the related term \textit{function} are not keywords but variables that can be given new values. Compared to most languages, Sway has surprisingly few keywords. In fact, there is just one: \textit{else}. The \textit{else} keyword will be discussed in Chapter 11.
This interaction defines a variable named `dots` whose value is bound to the number 42. The response of the interpreter to a variable declaration is to display the value bound to the variable.

```sway
sway> var bones = 206;
INTEGER: 206

sway> dots;
INTEGER: 42

sway> bones;
INTEGER: 206

sway> var CLXIV = bones - dots;
INTEGER: 164
```

After the declaration of variables, the variable and its value can be used interchangeably. Thus, one use of variables is to set up constants that will be used over and over again. For example, it is an easy matter to set up an equivalence between the variable `PI` and the real number 3.14159.

```sway
var PI = 3.14159;
var radius = 10;
var area = PI * radius * radius;
var circumference = 2 * PI * radius;
```

Notice how the expressions used to compute the values of the variables `area` and `circumference` are more readable than if 3.14159 was used instead of `PI`. In fact, that is one of the main uses of variables, to make code more readable. The second is if the value of `PI` should change (e.g., a more accurate value of `PI` is desired), we would only need to change the definition of `PI` (this assumes, of course, we can store those definitions for later retrieval and do not need to type them into the interpreter again).

### 5.2 Environments

The variables and their values are stored in a structure called an *environment*. You can think of an environment as a shoebox full of envelopes. When we define a variable, we create a new envelope with the name of the variable on the outside of the envelope and its value on the inside. We then place the new envelope at the front of the shoebox. When we need the value of a variable, we start looking in the shoebox, from front to back, looking for the right envelope. When we find the properly labeled envelope, we retrieve the value inside the envelope.

The environment is said to hold variables and their *bindings* (i.e., the values in the envelopes). A variable is said to be *bound* with respect to an environment if it has a binding in that environment. When asked to evaluate a variable, Sway simply looks up the variable in the environment and retrieves its value. Sometimes, the value bound to a variable is another environment. In fact, there is a variable named `this` that is that holds the location of the current environment (the `this` variable is predefined so you do not need to define it yourself). If you wish to see what bindings are in effect, you can ask the interpreter by issuing the command

---

6 The believed value of `PI` has changed throughout the centuries and not always to be more accurate (see [http://en.wikipedia.org/wiki/History_of_Pi](http://en.wikipedia.org/wiki/History_of_Pi))
If we do so after defining all the variables above, we get the following response from the interpreter:

```
sway> pp(this);
<OBJECT 1651>:
    context: <OBJECT 323>
    dynamicContext: null
    callDepth: 0
    constructor: null
    this: <OBJECT 1651>
    dots: 42
    bones: 206
    CLXIV: 164
    PI: 3.141590
    radius: 10
    area: 314.159000
    circumference: 62.831800
OBJECT: <OBJECT 1651>
```

Issuing the `pp` command is, in programming language parlance, an example of ‘calling a function’ and ‘passing an argument’. We will learn more about functions later, but the main idea is that a function performs some useful task or set of tasks. In this case, the task is to show the current set of variable-value pairs.

In addition to the bindings of the variables we previously defined, we see some additional bindings for `constructor`, `dynamicContext`, `callDepth`, `context`, and `this`. These variables are predefined by Sway for every environment. In this particular case, the variable `context` is bound to the environment that has the Sway built-in functions defined. As you will later see, sometimes we will try to look up a variable in the current environment. If we don’t find it, we will look up the value of the `context` variable instead and continue our search using that environment instead. We will discuss the other predefined variables further at a later time.

If you were to ask the interpreter to display the bindings of `context`, you would get a long list:

```
sway> pp(context);
<OBJECT 137>:
    ...
    array?: <function array?(a)>
    string?: <function string?(a)>
    real?: <function real?(a)>
    integer?: <function integer?(a)>
    ...
    +: <function +(a,b)>
    -: <function -(a,b)>
    *: <function *(a,b)>
    /: <function /(a,b)>
    ...
    if: <function if(a,$b,$c)>
    while: <function while($a,$b)>
    ...
    ||: <function ||>
    &&: <function &&>
    ...
```
5.3. VARIABLE NAMING

Note that even though the variables *context* and *this* are bound to environments, the *pp* function says they are bound to *objects*. In Sway, unlike other languages, objects and environments are the same thing. In fact, the terms object and environment should be considered equivalent. You will learn more about objects in a later chapter.

After displaying the bindings, the interpreter displays the result of evaluating the *pp* command, which is always the thing passed to the *pp* function. The fact that something was displayed other than the returned value is known as a *side-effect*. The side-effect is not part of the value of the expression. It is these very side-effects that make it dangerous to depend on the order of operand evaluation.

The ability to look at the current set of bindings is a useful tool for figuring out why a program is not behaving as you think it should. The process of fixing such problems is known as *debugging*. Debugging is discussed further in chapter 10.

5.3 Variable naming

Unlike many languages, Sway is quite liberal in regards to legal variable names and to what entities variables can be bound. Consider the following variable declarations:

```
sway> var times = *;
BUILTIN: <function *>

sway> 6 times 7;
INTEGER: 42

sway> var dots+bones = 5;
INTEGER: 5

sway> dots+bones;  
INTEGER: 5

sway> dots + bones; 
INTEGER: 248
```

The first declaration defines a variable named *times* and gives it the same value as the operator *. Note that the response of the interpreter tells us that times has been bound to the function originally bound to *.\(^7\) At this point, both the variables *times* and * are bound to the same function and can be used interchangeably, as the next interaction shows. From this example, we can see that operators are regular variables which happen to be bound to appropriate built-in functions. As stated earlier, this allows the Sway programmer great latitude in creating new operators. From the next interactions, we can see that whitespace is very important in delineating variable names: *dots+bones* is a single variable name while *dots + bones* is a combination of two distinct variables named *dots* and *bones*: in particular their values are combined with the built-in function bound to the variable *.

Variables are the next layer in a programming languages, resting on the primitive expressions and combinations of expressions (which are expressions themselves). In fact, variables can be thought of as an abstraction

\(^7\)In Sway, all operators are *functions*. Conversely, all functions that take two arguments are operators. More on this later.
of the primitives and collections of primitives. As an analogy, consider your name. Your name is not you, but it is a convenient (and abstract) way of referring to you. In the same way, variables can be considered as the names of things. A variable isn’t the thing itself, but a convenient way to referring to the thing.

While Sway lets you name variables in wild ways, you should temper your creativity if it gets out of hand. For example, rather than use the variable \textit{m} for the slope, we could use the name \textit{slope} instead:

\begin{verbatim}
var slope = 6;
\end{verbatim}

We could have also used a different name:

\begin{verbatim}
var !@#$ = 6;
\end{verbatim}

The name \texttt{!@#$} is a perfectly good variable name from Sway’s point of view. It is a particularly poor name from the point of making your Sway programs readable by you and others. It is important that your variable names reflect their purpose. In the example above, which is the better name: \textit{m}, \textit{slope}, or \texttt{!@#$}?
Chapter 6

Assignment

Once a variable has been declared, it is possible to change its binding using the assignment operator. Consider the following interaction with the interpreter:

```sway
sway> var BLACK = 1; //initialization!
INTEGER: 1

sway> var BROWN = 2;
INTEGER: 2

sway> var GREEN = 3;
INTEGER: 3

sway> var eyeColor = BLACK;
INTEGER: 1

sway> eyeColor;
INTEGER: 1

sway> eyeColor = GREEN; //assignment!
INTEGER: 3

sway> eyeColor == BROWN; //equality?
SYMBOL: :false

sway> eyeColor == GREEN;
SYMBOL: :true
```

The operator/variable = (equals sign) is bound to the assignment function. The assignment function, however, is not a true function, like those bound to + and *. Recall that + and the like evaluate the things on either side (recall that those things on either side are generically known as operands) before combining them. For =, the left operand is not evaluated: (if it were, the assignment `eyeColor = GREEN` would attempt to change the meaning of 1 to be 3). In general, an operator which does not evaluate all its arguments is known as a special form.¹

¹Unlike most languages, Sway allows user defined special forms.
Now we can see a variable definition has two steps. In the first step, the variable is created and in the second step, a value is assigned to that variable.

The last two expressions given to the interpreter in the previous interaction refer to the \(==\) (equality) operator. The function bound to \(==\) returns true if its operands refer to the same thing and false otherwise. In the interaction above, the variables BLACK, GREEN, and BROWN are not meant to change from their initial values. We denote variables whose values aren't supposed to change by naming the variable using (mostly) capital letters (this convention is borrowed from earlier programming languages). The use of caps emphasizes the constant nature of the (not too) variable.

In the above interaction with the interpreter, we use the integers 1, 2, and 3 to represent the colors black, brown, and green. By abstracting 1, 2, and 3 and giving them meaningful names (i.e., BLACK, BROWN, and GREEN) we find it easy to read code that assigns and tests eye color. We do this because it is difficult to remember which integer is assigned to which color. Without the variables BLACK, BROWN, and GREEN, we have to keep little notes somewhere to remind ourselves what’s what. Here is an equivalent interaction with the interpreter without the use of the variables BLACK, GREEN, and BROWN.

```
sway> var eyeColor = 1;
INTEGER: 1

sway> eyeColor;
INTEGER: 1

sway> eyeColor = 3;
INTEGER: 3

sway> eyeColor == 2;
SYMBOL: :false

sway> eyeColor == 3;
SYMBOL: :true
```

In this interaction, the meaning of the test `eyeColor == 3` is not so obvious.

Another approach to constants is to use symbols. Since Sway has symbols as primitives, we can dispense with the constant-like variables and the integers altogether:

```
sway> var eyeColor = :black;
SYMBOL: :black

sway> eyeColor;
SYMBOL: :black

sway> eyeColor = :green;
SYMBOL: :green

sway> eyeColor == :brown;
SYMBOL: :false

sway> eyeColor == :green;
SYMBOL: :true

sway> println("eye color is ",eyeColor);
```
6.1. PRECEDENCE AND ASSOCIATIVITY OF ASSIGNMENT

Assignment has the lowest precedence among the binary operators. It is also right associative. The right associativity allows for statements like

\[ a = b = c = d = 0; \]

which conveniently assigns a zero to four variables at once and, because of the right associative nature of the operator, is equivalent to:

\[ (a = (b = (c = (d = 0)))); \]

The resulting value of an assignment operation is the value assigned, so the assignment \( d \equiv 0 \) returns 0, which is in turn, assigned to \( c \) and so on.

6.2 ADVANCED TOPIC: How it works

Sway takes a novel approach to assignment. Every time a value is retrieved, its address is (usually) saved in an internal register. The assignment operator takes advantage of this fact by:

1. looking up the value of the left-hand-side
2. checking for a valid address in the internal register
3. writing the value of the right hand side to the valid address

For example, the assignment

\[ x = 3; \]

proceeds by looking up the value of \( x \). The current value of \( x \) is ignored, but the internal register now contains the location of \( x \). The value of 3 is then copied to the address and \( x \) has a new value.

By this method, one can write to array and list elements as well as objects:

\[
\begin{align*}
\text{a}[y] &= z; \\
\text{head(tail(items))} &= :\text{green}; \\
\text{a} . \text{f}() . \text{b} &= "hello";
\end{align*}
\]

There are some limitations to assignment. For example, you cannot use the assignment operator to change the tail of a list. For example, this attempt will fail:
tail(items) = list(a,b,c);

because the tail of a list does not have not a normal address in Sway. To change the tail of the list, one uses the \textit{tail=} operator, instead:

\textit{items tail= list(a,b,c);}

or

\textit{tail=(items, list(a,b,c));}

if you prefer function call syntax.

There is a \textit{head=} operator for changing the head of the list, but it is bound to the normal assignment function, so you can use either.

\section*{6.3 ADVANCED TOPIC: Assigning to Thunks}

If the code in a thunk has a regular Sway address, you can use the assignment operator to update the value at that address. This is how a \textit{forEach} function would work:

\begin{verbatim}
function forEach($target, items, $body)
{
    while (items != null)
    {
        $target = head(items);
        force($body);
        items = tail(items);
    }
}
\end{verbatim}

Here, the thunk $\$target is repeatedly updated with the (new) head of the list. Once the update is made with the assignment operator, the body of the \textit{forEach} is forced and the process repeated. The \textit{forEach} function can be called with a simple variable, as in

\begin{verbatim}
var i;
forEach(i, range(0,4))
{
    println("i is ",i);
}
\end{verbatim}

or with an array or list location, as in

\begin{verbatim}
var i = array(1,2,3);
forEach(i[2], range(0,4))
{
    println("i[2] is ",i[2]);
}
\end{verbatim}
or with an object member, as in

```javascript
function bundle(a,b) { this; }
var i = bundle(1,2);
forEach(i . a,range(0,4))
{
    println("i . a is ",i . a);
}
```
Chapter 7

Functions

Recall, the series of expressions we evaluated to find the $y$-value of a point on the line

$$y = 5x - 3$$

given an $x$-value:

```sway
sway> var m = 5;
INTEGER: 5

sway> var x = 9;
INTEGER: 9

sway> var b = -3;
INTEGER: -3

sway> var y = m * x + b;
INTEGER: 42

sway> y;
INTEGER: 42
```

Now, suppose we wished to find the $y$-value corresponding to a different $x$-value or, worse yet, for a different $x$-value on a different line. All the work we did would have to be repeated. A function is a way to encapsulate all these operations so we can repeat them with a minimum of effort.

7.1 Encapsulating a series of operations

First, we will define a not-too-useful function that calculates $y$ give a slope of 5, a $y$-intercept of -3, and an $x$-value of 9 (exactly as above). We do this by wrapping a function around the sequence of operations above. The return value of a function is the value of the last thing evaluated.

```sway
function y()
{
    var m = 5;
    var x = 9;
    var b = -3;
    var y = m * x + b;
    return y;
}
```
m * x + b; //this quantity is returned
}

There are a few things to note. The keyword function indicates that a function definition is occurring. The name of this particular function is \( y \). The stuff between the curly braces is the code that will be evaluated (or executed) when the function is called. This code is not evaluated until then.

You can copy and paste this function into the Sway interpreter. If you do, you’ll see something like:

```
sway> function y()
more> {  
more> var m = 5;
more> var x = 9;
more> var b = -3;
more> m * x + b; //this quantity is returned
more> }
FUNCTION: <function y()>
```

Notice that the interpreter prompt changes to more when the input is incomplete. In the case of the function definition above, that occurs when the curly close brace is entered.\(^1\)

Once the function is defined, we can find the value of \( y \) repeatedly:

```
sway> y();
INTEGER: 42

sway> y();
INTEGER: 42
```

The parentheses after the \( y \) indicate that we wish to call the \( y \) function and get its value.

The \( y \) function, as written, is not too useful in that we cannot use it to compute similar things, such as the \( y \)-value for a different value of \( x \). But before we improve our function, let’s modify it so that it displays the current environment.\(^2\) This may help you to understand what happens in a function call. while the body of the function is executing:

```
function y()
{
    var m = 5;
    var x = 9;
    var b = -3;
    pp(this);
    m * x + b; //this quantity is returned
}
```

---

\(^1\)Typing a long construct into the interpreter is tedious. In a later chapter, we will learn how to store our code in a file and have the interpreter execute the code in that file. If we need to make changes to our code, we simply edit the file. That way, we do not need to type in the modified code into the interpreter from scratch.

\(^2\)Sway lets you redefine variables and functions.
When we call the new version of \( y \), we see its current environment, which has bindings for \( b, x, \) and \( m \).

```sway
sway> y();
<OBJECT 2566>:
  context: <OBJECT 749>
dynamicContext: <OBJECT 749>
callDepth: 1
constructor: <function y()>
this: <OBJECT 2566>
b: -3
x: 9
m: 5
INTEGER: 42
```

The variables \( b, x, \) and \( m \) are known as local variables since they are not directly visible outside the neighborhood of the function body.

### 7.2 Passing arguments

A hallmark of a good function is that it lets you compute more than one thing. We can modify our function to take in the value of \( x \) in which we are interested. In this way, we can compute more than one value of \( y \).

We do this by passing in an argument, in this case, the value of \( x \).

```sway
function y(x)
{
  var slope = 5;
  var intercept = -3;
  return slope * x + intercept;
}
```

We give names to the values being passed in by placing variable names between the function definition parentheses. In this case, we chose \( x \) as the name. Notice that since we are passing in \( x \), we no longer need (or want) the definition of \( x \), so we delete it. Now we can compute \( y \) for an infinite number of \( x \)'s:

```sway
sway> y(9);
INTEGER: 42
sway> y(0);
INTEGER: -3
sway> y(-2);
INTEGER: -13
```

What if we wish to compute a \( y \)-value for a given \( x \) for a different line? One approach would be to pass in the slope and intercept as well as \( x \):

```sway
function y(x,slope,intercept)
{
  slope * x + intercept;
```
CHAPTER 7. FUNCTIONS

If we wish to calculate using a different line, we just pass in the new slope and intercept along with our value of \( x \). This certainly works as intended, but is not the best way. One problem is that we keep on having to type in the slope and intercept even if we are computing \( y \)-values on the same line. Anytime you find yourself doing the same tedious thing over and over, be assured that someone has thought of a way to avoid that particular tedium. So assuming that is true, how do we customize our function so that we only have to enter the slope and intercept once per particular line? We will explore three different ways for doing this. In reading further, it is not important if you understand all that is going on. What is important is that you know other approaches exist and understand the pros and cons of each approach:

7.3 Creating functions on the fly

Since creating functions is hard work (lots of typing) and Computer Scientists avoid hard work like the plague, somebody early on got the idea of writing a function that itself creates functions! Brilliant! We can do this for our line problem. We will tell our creative function to create a \( y \) function for a particular slope and intercept! While we are at it, let’s change the variable names \( m \) and \( b \) to slope and intercept, respectively:

```plaintext
function makeLine(slope, intercept)
{
    function y(x)
    {
        slope * x + intercept;
    }
    y; // the value of y is returned
}
```

The `makeLine` function creates a local \( y \) function and then returns it. This next version is equivalent:

```plaintext
function makeLine(slope, intercept)
{
    function y(x)
    {
        slope * x + intercept;
    }
}
```

Since the last thing `makeLine` does is to define the \( y \) function, the \( y \) function is returned by a call to `makeLine`

So our creative function simply defines a \( y \) function and then returns it. Now we can create a bunch of different lines:

```plaintext
sway> var a = makeLine(5,-3);
```
Notice how lines \( a \) and \( b \) remember the slope and intercept supplied when they were created.\(^3\) While this is decidedly cool, the problem is many languages (C and Java included) do not allow you to define functions that create other functions. Fortunately, Sway does allow this.

### 7.4 Using objects

Another approach to our line problem is to use something called an object. In Sway, an object is simply an environment and we have seen those before. So there is nothing new here except in how to use objects to achieve our goal. Here, we define a function that creates and returns a line object. A function that creates and returns an object is known as a constructor.

```sway
function line(slope, intercept)
{
  this;
}
```

The `this` variable always points to the current environment, which in this case includes the bindings of the formal parameters `slope` and `intercept`. By returning `this`, we return the environment of `line`, and we can look up the values of `slope` and `intercept` at our leisure. To prove that `slope` and `intercept` exist, we can use the built-in pretty printing function, `pp`:

```sway
sway> m = line(5,-3);
OBJECT: <OBJECT 231>

sway> pp(m);
<OBJECT 231>:
  context : <object 145>
  dynamicContext: <object 145>
  constructor: <function line(slope,intercept)>
  this: <object 231>
  intercept: -3
  slope : 5
  OBJECT: <OBJECT 231>
```

We access the variables in an object with the `.\(^\prime\)` (dot) operator:

```sway
sway> m . slope;
3
```

\(^3\)The local function \( y \) has, as its context, the local environment of the `makeLine` function. This environment holds the bindings for `slope` and `intercept`.\)
Now we modify our \( y \) function to take in a line object as well as \( x \) and use the dot operator to extract the line’s slope and intercept:

```javascript
function y(line,x)
{
    line . slope * x + line . intercept;
}
```

In this scenario, we create different lines, then pass each line to our new \( y \) function:

```javascript
sway> var m = line(5,-3);
OBJECT: <object 231>

sway> var n = line(6,2);
OBJECT: <object 256>

sway> y(m,9);
INTEGER: 42

sway> y(n,9);
INTEGER: 56
```

The problem with this approach is we have separated line objects from finding \( y \) values, yet these two concepts are closely related. As an example, suppose we have parabola objects as well as line objects. Our \( y \) function would fail miserably for parabola objects even though the concept of \((x,y)\) points on a parabola is just as valid as points on a line.\(^4\)

In the object-oriented world, we solve this problem by bundling the object and functions that work specifically on that object together. In our case, we make the \( y \) function part of the \texttt{line} object:

```javascript
function line(slope,intercept)
{
    function y(x)
    {
        slope * x + intercept;
    }
    this;
}
```

This is very similar to the functions-on-the-fly approach, but we return \texttt{this} instead of the function bound to \( y \). Now we call the \( y \) function via the line object.

\(^4\)Here is a concrete example of trying to generalize, so that our function works for all objects for which the concept of the function is valid.
saway> var m = line(5,-3);
OBJECT: <object 231>

saway> var n = line(6,2);
OBJECT: <object 256>

saway> m . y(9);
INTEGER: 42

saway> n . y(9);
INTEGER: 56

Should we have a parabola object, it would have its own y function with a different implementation. We would call it just the same, however:

saway> var p = parabola(2,0,0);
OBJECT: <object 453>

saway> p . y(7);
INTEGER: 49

This approach is supported in object oriented languages such as Java. The earlier approach (where the function was separated from the object) is supported in procedural languages such as C.

7.5 Functions versus operator

All operators are functions and can be called using operator and function call syntax. For example, the following expressions both sum the values of a and b:

```javascript
var sum = a + b;
var sum = +(a,b);
```

Conversely, any function of two arguments can be called using operator syntax. Sometimes using operator syntax makes your code more clear. Let’s make a function that increments a variable by a given amount, similar to the C, C++, and Java operator of the same name:

```javascript
function +=($v,amount)
{
    $v = force($v) + amount;
}
```

Don’t worry about how the code works; just note that the += function has two formal parameters ($v and amount) and thus takes two arguments. We can call += to increment a variable using function call syntax:

```javascript
var x = 2;
+=x,1;
inspect(x);
```

or we can use operator syntax:
var x = 2;
x += 1;
inspect(x);

In both cases, the output of the code fragments is the same:

x is 3

Functions that are called using operator syntax have the same precedence level as the mathematical operators and are left associative.
Chapter 8

Using Files

After a while, it gets rather tedious to cut and paste into the Sway interpreter. A more efficient method is to store your program in a text file and then load the file.

I use vim as my text editor. Vim is an editor that was written by programmers for programmers (emacs is another such editor) and serious Computer Scientists and Programmers should learn vim (or emacs).

8.1 Your first program

Create a text file named hello.s. The name really doesn’t matter and doesn’t have to end in .s (the .s is a convention to remind us this file contains Sway source code). Place in the file:

    println("hello, world!");

Save your work and exit the text editor. Now execute the following command at the system prompt (not the Sway interpreter prompt!):

    sway hello.s

You should see the phrase:

    hello, world!

displayed on your console. Here’s a trace using Linux:

    lusth@warka:~$ sway hello.s
    hello, world!
    lusth@warka:~$

The lusth@warka: $ is my system prompt.

8.2 Vim and Sway

To use vim effectively, run the following series of commands, in the order given:
cd wget sway.cs.ua.edu/.exrc mkdir .vim cd .vim wget sway.cs.ua.edu/filetype.vim mkdir syntax cd syntax wget sway.cs.ua.edu/sway.vim

This configures vim to understand Sway syntax and to color various primitives and keywords in a pleasing manner.

### 8.3 A Neat Macro

One of the more useful things you can do is set up a vim macro. Edit (or create) the file .exrc in your home directory and find these lines:

```vim
map @ :!sway %\~M
map # :!sway %
set ai sm sw=4
```

If you were unable to download the files in the previous section, just enter the lines above in the .exrc file.

The first line makes the 'key, when pressed, run the Sway interpreter on the file you are currently editing (save your work first before tapping the @ key). The \~M part of the macro is not a two character sequence (\~ followed by M), but a single character made by typing <Ctrl>-v followed by <Ctrl>-m. It’s just when you type <Ctrl>-v<Ctrl>-m, it will display as \~M. TRhe second line defines a similar macro that pauses to let you enter command-line arguments to your sway program. The third line sets some useful parameters: autoindent and showmatch. The expression sw=4 sets the indentation to four spaces.

### 8.4 Writing Sway Programs

A typical Sway program is composed of two sections. The first section is composed of variable and function definitions. The next section is composed of statements, which are Sway expressions, each of which is (usually) followed by a semicolon.

The hello.s file above was a program with no definitions and a single statement. A Sway program composed only of definitions will usually run with no output to the screen. Such programs are usually written for the express purpose of being included into other programs.

Typically, one of the function definitions is a function named main (by convention); this function takes no arguments. The last line of the program is a call to main. Here is a rewrite of hello.s using that convention.

```sway
function main()
{
    println("hello, world!");
}

main();
```

This version’s output is exactly the same as the previous version.

### 8.5 Order of definitions

A function or variable must be defined before it is used. This program will generate an error:

```sway
var x = y * y; //undefined variable error
```
var y = 3;

since \( x \) can’t be given a value until \( y \) is defined. This program is legal, however:

```sway
function x()
{
    y() * y();
}

function y()
{
    3;
}

x();
```

because even though the body of function \( x \) refers to function \( y \), function \( y \) is defined by the time function \( x \) is called (the last statement of the program).

### 8.6 Including code

One can include one Sway module into another by use of the `include` function:

```sway
include("moduleX.s");
```

where `moduleX.s` is the name of the file containing the Sway definitions you wish to include. Including a module imports all the top level definitions in that module. The statements beyond the definitions in the included module are ignored and are not evaluated.

If `moduleX.s` has includes, those modules will be included as well.

Include statements are used include the standard Sway libraries.

### 8.7 Multiple includes

Sometimes, in a complex implementation, a module can get included more than once. Sway prevents this by creating a special variable for each included module. The name of the variable begins with two underscores. The underscores are followed by the name of the included module which, in turn, is followed by the string ",included". For example, if the module name is `nothing`, then the variable created is `_nothing_included_`:

```sway
include("nothing");

sway>_nothing_included_
SYMBOL: :true
```

If a module is included again, at the same level, Sway checks to see if the special variable exists. If it does, then the module is not included a second time.

It is important to note that this guard against multiple includes only works within a scope level, but not across levels. In the following example, the `nothing` module is included twice, once at the outermost scope level and once at the nested level when \( f \) is called:
include("nothing");

function f(x)
{
    include("nothing");
    ...
}

f(1);

8.8 Includes are considered definitions

The call to the include function is the only piece of executable code that can appear before function and variable definitions. Calls to include are considered definitions and thus can appear in the definition area of a Sway module. It is legal, however, to call the include function in an expression, as in:

    last = include("nothing");

Of course, such calls to include are not part of the definition area. In the following example, only the top level definitions from modules B.s and C.s are included in A.s.

file A.s

    include("B.s");
    ...

file B.s

    include("C.s");
    var x;
    x = include("D.s");

The definitions from module D.s are not included into A.s since only the definition area of B.s is included in A.s. The call to include D.s is in the expression area and is ignored when B.s is included. In contrast, the definitions of of C.s are included into A.s.

8.9 Encapsulated includes

Sometimes it is useful to encapsulate an include into an object to prevent namespace collisions. Suppose the module nothing has a function named going. With a regular include, nothing's version of going would be overridden by the local version. By nesting the include in an object-creating scope, the two definitions can be kept separate:

    var nothing =
    {
        include("nothing");
        //turn this scope into an object
        this;
    }
function going(somewhere)
{
    ...
}

going(:home);   //local going
nothing . going(:on); //included going
Chapter 9

More about Functions

We have already seen some examples of functions, some user-defined and some built-in. For example, we have used the built-in functions, such as *. In reality, * is not a function but a variable that is bound to the function that multiplies two numbers together, but it is tedious to say ‘the function bound to *’ so we say the more concise (but technically incorrect) phrase ‘the * function’.

9.1 Predefined Functions

Sway has many predefined functions. You can see the list of built-in functions by executing the command

```sway
sway -p
```

at the system prompt (don’t confuse the system prompt with the Sway interpreter prompt). More detail on the built-in functions is given in the last chapter. No one, however, can anticipate all possible tasks that someone might want to perform, so most programming languages allow the user to define new functions. Sway is no exception and provides for the creation of new and novel functions. Of course, to be useful, these functions should be able to call built-in functions as well as other programmer created functions.

For example, a function that determines whether a given number is odd or even is not built into Sway but can be quite useful in certain situations. Here is a definition of a function named `even?` which returns true if the given number is even, false otherwise:

```sway
sway> function even?(x) { return x % 2 == 0; }
FUNCTION: <function even?(x)>

sway> even?;
FUNCTION: <function even?(x)>

sway> even?(4);
SYMBOL: :true

sway> even?(5);
SYMBOL: :false

sway> even?(4 + 5);
SYMBOL: :false
```
We could talk for days about what’s going on in these interactions with the interpreter. First, let’s talk about
the syntax of a function definition. Later, we’ll talk about the purpose of a function definition. Finally, will
talk about the mechanics of a function definition and a function call.

9.2 Function syntax

Recall that the words of a programming language include its primitives, keywords and variables. A function
definition corresponds to a sentence in the language in that it is built up from the words of the language. And
like human languages, the sentences must follow a certain form. This specification of the form of a sentence
is known as its syntax. Computer Scientists often use a special way of describing syntax of a programming
language called the Backus-Naur form (or BNF). Here is a high-level description of the syntax of a Sway
function definition using BNF:

```
functionDefinition : 'function' variable '(' optionalParameterList ')' block

optionalParameterList : <math>\Epsilon</math> | parameterList

parameterList : variable | variable ',' parameterList

block : '{' definitionSequence statementSequence '}'
```

The first BNF rule says that a function definition begins with the keyword function (parts of the rule
that appear verbatim appear within single quotes), followed by a variable, followed by an open parenthesis,
followed by something called an optionalParameterList, followed by a close parenthesis, followed by something
called a block. By reading the remaining rules, we see that one defines a function by entering the keyword
function followed by the name of the function, followed by a parenthesized list of formal parameters (possibly
empty, as indicated by ε), followed by the body of the function. The body is a brace-enclosed list of definitions,
then statements (the block that follows the parameter list is often called the function body).

The parameter list is composed of zero or more variable names, separated by commas. Parameters are local
variables that will be bound to the values given in the call to the function. In the particular case of even?,
the variable x will be bound to the number whose evenness is to be determined. It is customary to call x a
formal parameter of the function even?. In function calls, the values to be bound to the formal parameters
are called arguments.

9.3 Function Objects

Let’s look at the body of even?. The % operator is bound to the remainder or modulus function. The ==
operator is bound to the equality function and determines whether the value of the left operand expression is
equal to the value of the right operand expression, yielding true or false as appropriate. The value produced
by == is then immediately returned as the value of the function.

When given a function definition like that above, Sway performs a couple of tasks. The first is to create
the internal form of the function, known as a function object, which holds the function’s name, parameter
list, and body, plus the current environment. The second task is to add the function name and the function
object to the current environment as a variable-value binding. Thus the name of the function is simply a
variable that happens to be bound to a function object. As noted before, we often say ‘the function even?’
even though we really mean ’the function bound to the variable even?’.

The value of a function definition is the function object, which has type FUNCTION (indicating a user of
9.4 CALLING FUNCTIONS

Sway has defined the function and a printable value of \texttt{function definitionName formalParameters} where \textit{definitionName} is the name of the function when created and \textit{formalParameters} is the parenthesized list of parameters. Although the print value of the function object only lists the original name and the parameters, the actual object contains the body and context in which it was created (known as the defining environment) as well.

We can see the actual components of a function object by passing the function object to the built-in \texttt{ppObject} function. The \texttt{pp} in \texttt{ppObject} stands for \textit{pretty printing} which means to display in a pleasing format.

\begin{verbatim}
sway> ppObject(even?);
<FUNCTION 2421>:
context: <OBJECT 749>
prior: :null
filter: :null
parameters: (x)
code: { return x % 2 == 0; }
name: even?
FUNCTION: <function even?(x)>
\end{verbatim}

In addition to some other fields we will learn about later, we see that the \texttt{parameters}, \texttt{code}, and \texttt{name} fields look as expected. The only unexpected item is that the code contains a \texttt{return} where it did not before. You will learn more about returns in a later chapter. For now, we’ll just say that returns allow you to return a value from somewhere other than the last expression in the function body.

While we are on the subject of pretty printing, we can also look at \texttt{even?} with the \texttt{pp} function:

\begin{verbatim}
sway> pp(even?)
function even?(x)
{
    return x % 2 == 0;
}
FUNCTION: <function even?(x)>
\end{verbatim}

We see that the \texttt{pp} function, when given a function to pretty print, reproduces the function definition.

9.4 Calling Functions

Once a function is created, it is used by \textit{calling} the function with \textit{arguments}. A function is called by supplying the name of the function followed by a parenthesized, comma separated, list of expressions. The arguments are the values of those expressions and are to be bound to the formal parameters. In general, if there are \textit{n} formal parameters, there should be \textit{n} arguments.\footnote{For \textit{variadic} functions, the number of arguments may be more than the number of formal parameters. For more information on variadic functions, see Chapter 20.} Furthermore, the value of the first argument is bound to the first formal parameter, the second argument is bound to the second formal parameter, and so on. Moreover, all the arguments are usually evaluated before being bound to any of the parameters.\footnote{It is possible to delay evaluation of arguments. See Chapter 19 on \textit{lazy evaluation} for more details.}

Once the evaluated arguments are bound to the parameters, then the body of the function is evaluated. Most times, the expressions in the body of the function will reference the parameters. If so, how does the interpreter find the values of those parameters? That question is answered in the next section.
9.5 Scope

The formal parameters of a function can be thought of as variable definitions that are only in effect when the body of the function is being evaluated. That is, those variables are only visible in the body and no where else. For that reason, parameters are considered to be local variable definitions, since they only have local effect (the function body). Any direct reference to those particular variables outside the body of the function is not allowed.\footnote{One can sometimes get to these variables indirectly, by using objects. See Chapter 15.} Consider the following interaction with the interpreter:

```
sway> function square(a) { return a * a; }
FUNCTION: <function square(a)>

sway> square(4);
INTEGER: 16

sway> a;
EVALUATION ERROR: :undefinedVariable
stdin,line 3: variable a is undefined
```

Scope refers to where a variable is visible.

In the above example, the scope of variable \textit{a} is restricted to the body of the function \textit{square}. Any reference to \textit{a} other than in the context of \textit{square} is invalid. Now consider a slightly different interaction with the interpreter:

```
sway> var a = 10;
INTEGER: 10

sway> var b = 1;
INTEGER: 1

sway> function almostSquare(a) { return a * a + b; }
FUNCTION: <function almostSquare(a)>

sway> almostSquare(4);
INTEGER: 17
```

In this dialog, two variable definitions, \textit{a} and \textit{b}, precede the definition of \textit{almostSquare}. In addition, the variable serving as the formal parameter of \textit{almostSquare} has the same name as the first variable defined in the dialog. Moreover, the body of \textit{almostSquare} refers to both variables \textit{a} and \textit{b}. Variable \textit{a} is defined twice (once as a regular variable and once as a formal parameter) while variable \textit{b} is referenced but is not a formal parameter. Although it seems confusing at first, the Sway interpreter has no difficulty in figuring out what’s what. From the responses of the interpreter, the \textit{b} in the body must refer to the variable that was defined with an initial value of 1 (since there it is the only \textit{b}). The \textit{a} in the function body must refer to the formal parameter whose value was set to 4 by the call to the function (given the output of the interpreter).

When a formal parameter (or any definition local to a function) has the same name as another variable that is also in scope, the formal parameter is said to \textit{shadow} the other. The term shadowed refers to the fact that the other variable is in the shadow of the formal parameter and cannot be seen. A variable is said to be in scope if it is bound in the current environment or in the environment bound to the context variable in the current environment. We can see this clearly in Sway by looking at bindings. Consider this dialog:
9.5. SCOPE

```sway
sway> var a = 10;
INTEGER: 10

sway> var b = 1;
INTEGER: 1

sway> function almostSquare(a) { pp(this); a * a + b; }
FUNCTION: <function almostSquare(a)>

Note that in the body of this version of almostSquare, we pretty print the current environment.

sway> almostSquare(4);
<Object 2569>:
    context: <Object 749>
    dynamicContext: <Object 749>
    callDepth: 1
    constructor: <function almostSquare(a)>
    this: <Object 2569>
        a: 4

INTEGER: 17
```

It is in this current environment that the value of `a` is retrieved in calculating the return value. We see that, indeed, `a` has a value of 4. But where is the value of `b` found?

When the value of a variable is needed, Sway looks in the current environment (`this`). If the variable is not found there, Sway looks in `context`. If the variable is not in `context`, it looks in the `context` of `context`, and so on. Let’s modify almostSquare to illustrate:

```sway
sway> function almostSquare(a) { pp(this); pp(context); a * a + b; }
FUNCTION: <function almostSquare(a)>

This latest version pretty prints both its current environment and its context:

sway> almostSquare(4);
<Object 2603>:
    context: <Object 749>
    dynamicContext: <Object 749>
    callDepth: 1
    constructor: <function almostSquare(a)>
    this: <Object 2569>
        a: 4

<Object 749>:
    context: <Object 18>
    dynamicContext: :null
    callDepth: 0
    constructor: :null
    this: <Object 749>
    almostSquare: <function almostSquare(a)>
    b: 1
    a: 10
SwayEnv: ["SSH_AGENT_PID=5076","SHELL=/bin/bash","TERM=x...
Here we see the bindings of \(a\), \(b\), and \(almostSquare\), as expected.

This shows two things:

1. that within the function body, formal parameters are found in the current environment
2. that the context of the environment active in a function call is bound to the defining environment of the function.

When a reference to \(a\) is made, a search is made of the current environment. Within the function body, the value of 4 is immediately found. When the value of \(b\) is required, it is not found in the current environment. The interpreter then searches the current environment’s context (in this case <OBJECT 1616>). This object does have a binding for \(b\).

Since \(a\) has a value of 4 and \(b\) has a value of 1, the value of 17 is returned by the function. Finally, the last interaction with the interpreter illustrates the fact that the initial binding of \(a\) was unaffected by the function call.

In general, a variable found in the current environment is considered local. A variable that is in scope but is not in the current environment is considered non-local. Alternatively, local variables are considered to reside in the local scope while non-local variables are said to reside in the non-local scope. Local and non-local scope are sometimes referred to as \textit{inner} and \textit{outer} scope, respectively. If a non-local variable resides in the outermost scope, it is considered a global variable and the environment holding the bindings of global variables is called the global environment. In Sway, the initial environment is the global environment. Contrary to the definition above, Sway’s global environment does have an outer scope; this outer scope holds the bindings of the built-in functions. The built-in environment cannot be modified so perhaps a better definition of the global environment is the outermost environment which can be modified by the programmer.

### 9.6 ADVANCED TOPIC: Environments

How did the environment <OBJECT 1845>, the environment under which the function body was executed, come into being? The evaluation of the expression \(almostSquare(4)\) triggers a number of actions. The first is the creation of a new environment that will hold the formal parameters of the function to be called (in this case, the single parameter \(a\)) bound to the values of the corresponding arguments in the call (in this case, the single argument has a value of 4). This new environment has its context variable bound to context variable found in the function object associated with the function being called (this new environment is also populated with other pre-defined variables such as constructor). The body of the function being called is then executed under this new environment. The process of creating a new environment and linking its context to another environment via its context variable is called extending an environment. To summarize, when a function call is performed, the following actions are performed:

1. the arguments to the function call are evaluated under the current environment
2. the function object associated with the function to be called is retrieved from the current environment
3. the formal parameters are retrieved from the function object
4. the defining environment is retrieved from the function object
5. a new environment is extended from the defining environment
6. the new environment is populated by binding the formal parameters to the evaluated arguments
7. the function body is retrieved from the function object
8. the function body is evaluated under the newly extended environment
9. the result of this evaluation is returned as the result of the function call

There are two important concepts about function calls: the static chain and the dynamic chain. The static chain we have already seen, the current environment, the context of the current environment, the context of the context, and so on. It is this chain that is searched for the value of a referenced variable. The dynamic chain, on the other hand, is the current environment, the environment of the calling function, the environment of the caller of the calling function, and so on. Unlike most languages, the dynamic chain is available for programmers to search and manipulate. For the moment, that kind of manipulation is beyond us, so we will postpone that topic to a later date.

9.7 Returning from functions

The return value of a function is the value of the last expression evaluated when executing the function body. The return function is used to make an expression anywhere in the function be the last expression evaluated.

```sway
function test(x,y)
{
    if (y == 0) {
        return(0);
    }
    println("good value for y!");
    return(x / y);
}
```

In the example, if \( y \) is zero, then there is an immediate return from the function and no other expressions in the function body are evaluated. If not, a message is printed and the a quotient is returned. The final return is not really needed; just having \( x/y \) as the final expression would work as well.

To make Sway, which is a functional language, look like C and Java, a call to the return function has an alternate syntax, the parentheses enclosing the single argument can be ommitted:

```sway
function test(x,y)
{
    if (y == 0) { return 0; }
    println("good value for y!");
    x / y;
}
```
Chapter 10

Debugging

At this point in time, you should be able to write some sophisticated Sway programs. In addition, I’m sure you are incorporating some sophisticated bugs, as well. Such is the life of a programmer!

There are two types of errors typically encountered in programming, syntax errors, in which the source code has problems and cannot be compiled and executed, and semantic errors, in which, the code runs (after a fashion) and then terminates too early or produces the wrong result.

Human languages have syntax and semantic errors. Here are some examples:

\[
\text{The box mxyskd the water.}
\]

This sentence is not valid English since token ’mxyskd’ is not a word. This is a syntactic error.

\[
\text{The box drank water the}
\]

This sentence is also not valid English since the article ’the’ is out of place and there is no punctuation to indicate the end of the sentence. These are also syntactic errors.

\[
\text{The box drank the water.}
\]

This sentence is syntactically correct, but doesn’t make sense semantically: boxes usually don’t drink.

Programming languages have errors similar to these and we will explore how detect and fix them in the subsequent sections.

As you learn these techniques, remember, the number one rule of debugging is to catch errors as early as possible. That is, find the smallest improvement to your code that gets you a step closer to the final version, then implement, test, and debug that step. Then repeat. This technique is known as stepwise-refinement.

Woe to the programmer that attempts to write a large project all at once and then begin testing!

10.1 Syntax Errors

Syntax errors found by the Sway interpreter always look something like this:

\[
\text{SYNTAX ERROR: :syntaxError}
\]
A file name and line number is given by the error report. Usually this is exactly where the problem is, but sometimes this is only where the problem was was detected; the actual error may have occurred earlier in the file.

Many of the non-obvious errors involve putting a semicolon in the wrong place or omitting one where it is necessary. Remember, semicolons do not follow named function definitions and certain function calls like if’s and whiles. In the next chapter, you will learn which function calls need to be followed by semicolons and which do not.

If you have a strange syntax error that you cannot track down, try commenting out pieces of the code until the error goes away. The error is likely within the code just commented out. Sway has three kinds of comments. The first is the end-of-line comment. Anything on the line after a double slash (two consecutive forward slashes) is ignored.

```sway
//removing the next function
//function id(x) { x; }
```

The second is the triple slash. This causes the rest of the file to be ignored and is quite useful for debugging syntax.

```sway
///removing the rest of the file
function id(x) { x; }
```

To use the triple slash, start with the /// at the very top of the file so that the whole file is ignored. This should remove the syntax error. Then keep moving the /// downward into the file until the syntax error appears. It is likely that the newly uncovered section of code contains the error.

The final kind of comment is the block comment. Anything appearing between a slash-star and a star-slash is ignored:

```sway
/* removing this function
function id(x) { x; }
*/
```

Note that a block comment cannot contain another block comment.

## 10.2 Semantic Errors

Syntax errors are found when the Sway interpreter reads your code, while semantic errors are found while the interpreter is evaluating code. Semantic error reports look like:

```
EVALUATION ERROR: :mathError
file prog1.s,line 3: division: cannot divide by zero
```
If you run your program with the -t option, more information about the error is provided.

```
EVALUATION ERROR: :mathError
file prog1.s,line 3: division: cannot divide by zero
```

```
CALL TRACE:
initial call:
  prog1.s,line 5: inspect(f(3));
in <function f(x)>...
  prog1.s,line 3: x / 0;
```

The file name and line number are given; like syntax errors, this is where the semantic error was detected. Following the description of the error comes a call trace. The call trace is read from top to bottom. In the example, the problem started on line 5 when the inspect function was called. This call triggered a call to the function f where the actual divide-by-zero was attempted on line 3. The call trace is extremely useful for tracking down problems.

Semantic errors usually occur when an important variable ends up having a different value than you expected. Therefore, it is important to see the values of your variables when debugging your code.

### 10.3 Print statements

You should get into the habit of being able to 'visualize' the state of your program. For example, suppose you wish to see the value of a variable named x at a number of different points in your program. The simplest visualization is a print statement of the form:

```
println("x is ",x);
```

I must comment, as a teacher, how rarely students make use of this simple tool. You should liberally use print statements to debug your semantic errors.

It gets rather tedious to add print statements of this sort, however, since you are only going to delete those lines once you solve the problem. What's needed is a faster way to add print statements. One such faster way is the inspect function. For example, the following two function calls are equivalent:

```
println("x is ",x);
inspect(x);
```

The output of both calls is exactly the same. If the value of x is 5, then the output of both calls is:

```
x is 5
```

Now the amount of time saved by using inspect isn't much in this case (a savings of typing eight characters fewer), but the savings grow when the expression to be inspected gets complicated. Consider these calls:

```
println("alpha . beta(gama,delta) is ",alpha . beta(gama,delta));
inspect(alpha . beta(gama,delta));
```
Again, the output of each are identical, but the savings are considerable when using `inspect`.

Finally, if even `inspect` is too much to write, define a variable such as `vv`, for view variable:

```
var vv = inspect;
```

and use it instead of `inspect`:

```
vv(x);
```

### 10.4 Tracing a function call

Sometimes it is useful to trace the execution of a function call, line by line. Suppose you wish to call a function named `f` and trace each line of the function as it executes. Simply wrap the call with statements that set the filter component of the function.

```
f . filter = trace*;
y = f(x);
f . filter = :null;
```

or more simply, using the wrapper function `trace::`

```
y = trace(f(x));
```

Setting the filter to `trace*` (which `trace` does) turns on tracing of the called function; setting the filter to `null` turns off tracing. You will need to include the `debug` library at the top of your file. For example, this program:

```
include("debug");

function f(x)
{
    var a = x + 1;
    var b = x - 1;
    a * b;
}

var result = trace(f(5));
inspect(result);
```

yields the following output:

```
prog.s,line 5: var a = x + 1;
prog.s,line 6: var b = x - 1;
prog.s,line 7: a * b;
result is 24
```
If you set the filter to `trace*` or, equivalently, use the `trace` function, each line of traced function’s body will be displayed as it is executed. If you set a function’s filter to `step*` or, equivalently, use the `step` function instead, your program will pause after printing out the traced statement. If you press the Enter key, the program will move on to the next line in the function.

If you type any character other than whitespace and hit return, you will be thrown into a miniature Sway interpreter. Here you can do nearly anything you can do in the interactive Sway interpreter. The only difference is that each definition or expression to be executed must be entered on a single line.

### 10.5 Stepping through functions

As noted in the last section, stepping through functions is similar to tracing, expect your program will pause before each statement is executed. If you simply press Enter, the current statement is executed, the next statement is displayed (if it exists), and the program pauses again.

However, if you type anything else during the pause, your input is evaluated as a Sway expression and the result displayed. This evaluation is performed under the environment in force during the evaluation of the function body. In other words, you can examine (and modify) the variables in scope during the function call. This mini-interpreter keeps on running until you stop entering expressions to evaluate. When you stop entering expressions, the program advances to the next line in the function call.

Setting up stepping through a function is similar to tracing:

```sway
\color{CodeGreen}
\begin{codesize}
\begin{verbatim}
f . filter = step*;
y = f(x);
f . filter = :null;
\end{verbatim}
\end{codesize}
```

or more simply, using the wrapper function `step`:

```
y = step(f(x));
```

In the interaction (as above, but using `step`), the value `a` is examined and `b` is modified:

```
prog.s,line 5: var a = x + 1;>
prog.s,line 6: var b = x - 1;> a;
INTEGER: 6
>
prog.s,line 6: var b = x - 1;>
prog.s,line 7: a * b;> b = 10;
INTEGER: 10
>
 prog.s,line 7: a * b;>
result is 60
```

Note that the line presented at any given point has not yet been executed. Thus, we had to wait until the variable `a` was declared until we could examine it. As with tracing, you must include the `debug` library to use stepping.
The main difference between the mini-interpreter and the Sway interpreter is that the expression passed to the mini-interpreter must be entered all on one line.

You can exit the mini-interpreter by entering a Ctrl-d, or by entering a blank line.

**10.6 Breakpoints**

Sometimes, you know exactly where in a function you wish to pause and examine variables. Rather than use `step`, you can call the mini-interpreter directly by using the `sway` function.

```sway
include("debug");

function f(x)
{
    var a = x + 1;
    var b = x - 1;
    println("breakpoint!");
    sway();  //call the mini-interpreter
    println("done.");
    a * b;
}

var result = f(5);

inspect(result);
```

Running this program yields:

```
breakpoint!
sway> a;
INTEGER: 6
sway> b = 10;
INTEGER: 10
sway> done.
result is 60
```

Like before, input to the mini-interpreter must be entered on a single line.

The `sway` function is built-in; thus, you do not need to include `debug`.

**10.7 Assertions**

There are two kinds of errors you must deal with. The first is the user of your program has supplied erroneous input. These kinds of errors are known as *external errors*. 'Bulletproofing' your code means adding the logic to deal external errors.

The second kind of errors arise from errors in the code itself. These are known as *internal errors*. A good Computer scientist anticipates that these errors will happen no matter how good a programmer he or she is and will use assertions to find these errors early. For example, suppose you 'know' that the input to a function is always a non-negative integer. You can use an assertion to detect variations from this constraint:
function f(x)
{
    assert(x >= 0);
    ...
}

During the development of code, the assertions make sure you haven’t called such functions inappropriately.

When you are finished with your code, you may comment out your assertions.
Chapter 11

Conditionals

Conditionals implement decision points in a computer program. Suppose you have a program that performs some task on an image. You may well have a point in the program where you do one thing if the image is a JPEG and quite another thing if the image is a GIF file. Likely, at this point, your program will include a conditional expression to make this decision.

Before learning about conditionals, it is important to learn about logical expressions. Such expressions are the core of conditionals and loops.\(^1\)

11.1 Logical expressions

A logical expression evaluates to a truth value, in essence true or false. For example, the expression \(x > 0\) resolves to true if \(x\) is positive and false if \(x\) is negative or zero. In Sway, truth is represented by the symbol \(\text{:true}\) and falsehood by the symbol \(\text{:false}\). Together, these two symbols are known as boolean values.

One can assign truth values to variables:

\[
\text{var } z = c > 0;
\]

Here, the variable \(z\) has a value of \(\text{:true}\) if \(c\) is positive; otherwise it has a value of \(\text{:false}\).

11.2 Logical operators

Sway has the following logical operators.

\[
== \quad \text{equal to}
\]
\[
!= \quad \text{not equal to}
\]
\[
>= \quad \text{greater than or equal to}
\]
\[
> \quad \text{greater than}
\]
\[
< \quad \text{less than}
\]
\[
<= \quad \text{less than or equal to}
\]
\[
&& \quad \text{and}
\]
\[
|| \quad \text{or}
\]

The first five operators are used for comparing two things, while the last two operators are the glue that joins up simpler logical expressions into more complex ones.

---

\(^1\) We will learn about loops in the next chapter.
11.3 Short circuiting

When evaluating a logical expression, Sway evaluates the expression from left to right and stops evaluating as soon as it finds out that the expression is definitely true or definitely false. For example, when encountering the expression:

\[ x \neq 0 \land y / x > 2 \]

if \( x \) has a value of 0, the subexpression on the left side of the \( \land \) connective resolves to false. At this point, there is no way for the entire expression to be true (since both the left hand side and the right hand side must be true for an \( \land \) expression to be true), so the right hand side of the expression is not evaluated. Note that this expression protects against a divide-by-zero error.

There are two basic constructs for selecting among alternatives in Sway. They are the if-then-else statement and the switch (or case) statement.

11.4 If expressions

Sway's if expressions are used to conditionally execute code, depending on the truth value of what is known as the test expression. One version of if has a block of code following the test expression:

Here is an example:

```sway
if (name == "John")
{
    println("What a great name you have!");
}
```

In this version, if the test expression is true (i.e., the string "John" is bound to the variable name), then the following block is executed (i.e., the compliment is printed). If the test expression is false, the following block is not executed.

Unlike most programming languages, Sway's if is a function. In the above example, the two arguments are the test expression and the following block. The following block is known as a proximal block.

Here is another form of if:

```sway
if (major == "Computer Science")
{
    println("Smart choice!");
}
else
{
    println("Ever think about changing your major?");
}
```

In this version, if takes three arguments: the test expression and the two following blocks.\(^2\) As before, the first block is evaluated if the test expression is true. If the test expression is false, however, the second block is evaluated instead.

\(^2\)The else is there for looks only and is known as 'syntactic sugar'.
Be aware that any proximal arguments must be blocks. Although the following is legal in some languages, a syntax error will be generated in Sway:

```swift
if (name == "John")
    println("What a great name you have!"); //not a block!
```

Since wrapping a single expression in a block can be considered tedious, you may dispense with the proximal blocks and use regular function call syntax for `if`:

```swift
if (a < b, min = a, min = b); //find the minimum of two numbers
```

The above is equivalent to:

```swift
if (a < b)
    { min = a; }
else
    { min = b; }
```

Note that if you move expressions inside as regular arguments, you will need a semicolon after the `if`'s close parenthesis, as with any other function call.

The function `if` has a return value, as does any other function in Sway. Indeed, you can use the return value to simplify the above expression:

```swift
min = if (a < b) { a; } else { b; };
```

or

```swift
min = if (a < b,a,b);
```

If you use the return value of an if, you must end the expression with a semicolon, regardless of whether you use proximal blocks or not.

Remember, if you use an `if` by itself and the `if` has proximal blocks, do not follow the expression with a semicolon. Otherwise a semicolon is needed at the end of the entire expression containing the `if`.

### 11.5 if chains

You can chain `if expressions` together, as in:

```swift
if (bases == 4) { println("HOME RUN!!"); }
else if (bases == 3) { println("Triple!!"); }
else if (bases == 2) { println("double!"); }
else if (bases == 1) { println("single"); }
else { println("out"); }
```
As before, the value of the block that is eventually evaluated is the return value of the entire construct. If no block is evaluated, the return value is `false.

What is the difference between chained `if`s and a sequence of unchained `if`s? For example,

```sway
define home_run(b) {
  if (bases == 4) { println("HOME RUN!!!"); } 
  if (bases == 3) { println("Triple!!"); } 
  if (bases == 2) { println("double!"); } 
  if (bases == 1) { println("single"); } 
  else { println("out"); }
}
```

The answer is left as an exercise for the reader.

### 11.6 Switch expressions

The above if-chain can be rewritten using a `switch` expression:

```sway
var x = 0;

switch (x) case(4) println(x,"HOME RUN!!!"); case(3) println("Triple!!"); case(2) println("double!"); case(1) println("single"); else println("out");
```

If the switch expression, in this case `x`, matches any of the case expressions, in this case 4, 3, 2, and 1, then the block following the matching case is evaluated.

In languages like C, the switch statement is more efficient than an equivalent if-chain. In Sway, not so much. In fact, the switch is equivalently (un)-efficient, but often more pleasing to the eye. Unlike C, which requires the switch and case expressions to be integral and known at compile time, Sway’s switch is much more flexible, in that both kinds of expressions can be any valid Sway expression, including variables.

To use `switch`, you will need to include the `basics` library. To use the "basics" library, add the function call:

```sway
include("basics");
```

at the top of your source code file.

As with `if`, switch is a function with proximal block arguments. By itself, a call to switch with a proximal block does not have a terminating semicolon. As with `if`, switch is a function with proximal block arguments. By itself, a call to switch with a proximal block does not have a terminating semicolon. In all other cases, a semicolon will be needed.
Chapter 12

Loops

Sometimes, a programmer wishes to execute a section of code multiple times. This kind of task is commonly performed using a loop. The most basic loop structure in Sway is the while loop, whose format is...

```
whileloop: 'while' '(' 'expr' ')' 'block'
```

where expr is any Sway expression that resolves to the symbols :true or :false.

A while loop tests its condition before the body of the loop is executed. If the initial test fails, the body is not executed at all. For example:

```sway
var i = 10;
while (i < 10)
{
    print(i);
    i = i + 1;
}
```

never prints out anything since the test immediately fails. In this example, however:

```sway
var i = 0;
while (i < 10)
{
    print(i);
    i = i + 1;
}
```

the loop prints out the digits 0 through 9:

```
0123456789
```

A while loop executes its body (the block portion of the loop) repeatedly (as long as the condition remains true).

To write an infinite loop, use :true as the test expression:
while (:true)
{
var i = getInput();
process(i);
}

12.1 Loops are function calls

As with if, all Sway loops are function calls, usually with proximal block arguments. If desired, small proximal blocks can be moved inside the parentheses as regular arguments. Also, the semicolon rule for if applies to loops as well: A loop, by itself, having a proximal block argument is not followed by semicolon. In all other cases, a semicolon is necessary. If you were to use the return value of a loop (for while, it is :true if the test expression was initially true and :false if the test expression was initially false), you would need to terminate the entire expression with a semicolon.

Here is an example:

```
incremented = while (i % 10 != 0, i = i + 1);
```

This loop increments i until it is evenly divisible by 10.

12.2 Other loops

Because loops are so easy to write in Sway, only the while loop is built-in. Other loops can be found in the basics library. To use the basics library, add the function call:

```
include("basics");
```

at the top of your source code file.

12.2.1 The until loop

The until loop is similar to the while loop, but the body of the loop is executed as long as the test condition is false (or, in other words, until the test expression is true). For example:

```
var i = 0;
until (i >= 10) //note the test
{
  print(i);
  i = i + 1;
}
```

prints out the digits:

```
0123456789
```

Note that the until is exactly the same as a while loop except that the test expression is logically inverted.
12.2. OTHER LOOPS

12.2.2 The do-until loop

The do-until loop is similar to the until loop, but the body of the loop is executed once before the test is made. For example:

```javascript
var i = 10;
do-until (i >= 10)
{
    print(i);
i = i + 1;
}
```

prints out the value of \( i \) once:

10

The do-until loop is not used very often, though sometimes it is useful for processing interactive input. You can always write a do-until as an until loop. Consider this loop:

```javascript
// compute the gcd of n and d
do-until (r == 0)
{
    r = n / d;
n = d;
d = r;
}
```

It can be rewritten as a until loop thusly...

```javascript
// compute the gcd of n and d
r = start;
until (r == 0)
{
    r = n / d;
n = d;
d = r;
}
```

Note that the variable \( r \), being initialized to \( \text{start} \), ensures that the body of the loop is entered at least once.

Usually, the do-until is easier to read and understand.

12.2.3 The for loop

Another loop found in many programming languages is the for loop. The for loop is composed of four major parts:

1. the initialization step
2. the test
3. the body
4. the update

Note that these components correspond exactly to a *while* loop:

```plaintext
i = 0; // initialize
while (i < MAX) // test
{
    inspect(i);
    i = i + 1; // update (in body)
}
```

with the update usually being the last statement in the loop body. The corresponding *for* loop is:

```plaintext
for (i = 0, i < MAX, i = i + 1) // init, test, update
{
    inspect(i); // body
}
```

See how the update has been moved out of the body.

For loops are commonly used to sweep through each element of an array:

```plaintext
for (i = 0, i < length(items), i = i + 1)
{
    a[i] = b[i] * c[i];
}
```

The *for* loop is a counting loop. With most, but not all, counting loops, the loop variable, in this case *i*, is initialized to zero and the loop test, in this case *i < length(items)*, is configured to succeed as long as the loop variable is strictly less than maximum of the count. If the loop body is to be executed *n* times, then the loop would look something like:

```plaintext
for (i = 0, i < n, i = i + 1)
{
    ...
}
```

This convention stems from the fact that arrays in Sway (detailed in a later chapter) use 'zero-based indexing'. That is to say, the first element of an array is located at index zero, the second at index one and so on.

You should always use this convention unless there is good reason not to.\(^1\)

\(^1\) Just trust me on this one.
12.2.4 The \textit{for-each} loop

The \textit{for-each} loop is also useful for sweeping through arrays and lists. It does so without explicitly counting, however:

\begin{verbatim}
for-each (s,items)
  {
    inspect(s);
  }
\end{verbatim}

In this loop, each element in \textit{items} is bound to the variable \textit{s}, in turn. The loop body executes as many times as there are elements in the array or list. Note how much shorter the \textit{for-each} loop is compared to the equivalent \textit{for} loop:

\begin{verbatim}
for (i = 0; i < length(item); i = i + 1)
  {
    var s = items[i];
    inspect(s);
  }
\end{verbatim}

which explicitly counts.

The \textit{for-each} loop can also be used as a base for a loop that explicitly counts as well if one uses the \textit{range} function. The \textit{range} function, at its simplest, produces a list of numbers from zero to one less than its given argument. Thus, in general, any \textit{for} loop of the form:

\begin{verbatim}
for (i = 0, i < n, i = i + 1)
  {
    ...
  }
\end{verbatim}

can be replaced by the following for-each loop:

\begin{verbatim}
for-each(i,range(n))
  {
    ...
  }
\end{verbatim}

12.3 Leaving loops

There are numerous ways to leave a loop, besides having the loop condition or test failing (or succeeding in the case of the \textit{until} loop). It is possible to return from a function even if in the middle of a loop. This technique is often used when searching an array:

\begin{verbatim}
function find(x,items)
  {
    var s;
\end{verbatim}
for-each(s,items)
{
    if (x == s)
    {
        return :true;
    }
}
:false;
}

Note that if the item isn’t found, the loop eventually terminates and failure is noted by returning :false.

You can also leave a loop without leaving the function completely by calling break:

```plaintext
count = 0;
while (count < 10)
{
    {if (count > 5) { break(); }}
    inspect(count);
    count = count + 1;
}
println(:done);
```

Running this code yields:

```
i is 0
i is 1
i is 2
i is 3
i is 4
i is 5
done
```

Similar to break is continue. When a continue is encountered, the remainder of the loop body is skipped.

```plaintext
count = 0;
while (count < 10)
{
    count = count + 1;
    if (count < 5) { continue(); }
    inspect(count);
}
println(:done);
```

This loop prints the numbers from 5 to 10 inclusive. Notice that the update was performed before the continue. This is a requirement, otherwise the loop may loop for ever. You will need to include the basics library to use break.

Finally, you can leave a loop by leaving the program via the exit function:
while (1) {
    var input = getInput();
    if (reallyBad(input)) {
        exit(-1);       // non-zero exit code
    }
}

By convention, Sway programs that exit with code of zero means processing went normally. A non-zero exit code indicates that some sort of fatal error occurred.
Chapter 13

Recursion

In the previous chapter, we learned about looping using the functions \textit{while}, \textit{until}, \textit{do-while}, \textit{do-until}, \textit{for}, and \textit{for-each}. These functions implement what is known as \textit{iterative} looping.

\textit{Recursion} is another way of looping; often, recursive loops are easier to write and understand, as compared to the iterative loops such as \textit{while}s and \textit{for}s. Many mathematical functions are easy to implement recursively, so we will start there. Recall that the factorial of a number \( n \) is:

\[
\text{factorial}(n) = n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1
\]

Consider writing a function which computes the factorial of a positive integer. For example, if the function were passed the value of 4, it should return the value of 24 since 4! is 4*3*2*1 or 24. One approach would be to use an iterative loop to cycle through all the multipliers, saving the partial product at each iteration. Getting the code to work correctly does take a bit of fiddling to get all the important bits right. A working implementation might look like:

```javascript
function factorial(n) {
    var i;
    var product = 1;
    for (i = 1, i <= n, i = i + 1) {
        product = product * i;
    }
    product;
}
```

This implementation raises many questions, since it goes against the general convention for structuring a counting loop:

- Why was the product initialized to one instead of zero?
- Why was the loop initialization chosen to be \( i = 1 \) instead of \( i = 0 \)?
- Why was the loop test \( i <= n \) instead of \( i < n \)?

With some pondering, these questions can be answered because of the mathematics of multiplication.
There is a radically different approach that can be taken, an extremely simple, elegant, and powerful approach, called recursion. To apply recursion to solve a problem, it must be possible to state the solution of a problem in ever simpler terms. For factorial, the factorial of a number can be stated in terms of a simpler factorial.

\[
0! = 1 \\
(n)! = n \times (n - 1)!!
\]

This second formulation states that the factorial of zero is one and that the factorial of any other (positive) number is obtained by multiplying the that number by the factorial of one less than that number. After some study, you should be able to see that both the first formulation (with the ellipses ...) and this new formulation are equivalent. The second form is particularly well suited for implementation as a computer program:

```javascript
function factorial(n)
{
    if (n == 0)
        { 1; }
    else
        {
            n * factorial(n - 1);
        }
}
```
or:

```javascript
function factorial(n)
{
    if (n == 0, 1, n * factorial(n - 1));
}
```

using if's pure function call syntax.

Note how these two versions of factorial precisely implement the second formulation. Convince yourself that the function really works by tracing the function call:

```
factorial(3)
```

Decomposing the call, we find that:

```
factorial(3) is 3 * factorial(2)
```

---

1. Mathematicians, being an inclusive bunch, like to invite zero to the factorial party.
2. The first formulation gets the basic idea of factorial across but is not very precise. For example, how would you computer the factorial of three using the first formulation?
since \( n \), having a value of 3, is not equal to 0, and so the second block of the if is evaluated. We can replace \texttt{factorial(2)} by \( 2 \times \texttt{factorial(1)} \), yielding:

\[
\texttt{factorial(3)} \text{ is } 3 \times 2 \times \texttt{factorial(1)}
\]

since \( n \), now having a value of 2, is still not zero. Continuing along this vein, we can replace \texttt{factorial(1)} by \( 1 \times \texttt{factorial(0)} \), yielding:

\[
\texttt{factorial(3)} \text{ is } 3 \times 2 \times 1 \times \texttt{factorial(0)}
\]

Now in this call to \texttt{factorial}, \( n \) does have a value of zero, so we can replace \texttt{factorial(0)} with its immediate return value of zero:

\[
\texttt{factorial(3)} \text{ is } 3 \times 2 \times 1 \times 1
\]

Thus, \texttt{factorial(3)} has a value of six:

\[
\texttt{sway> factorial(3)};
\]

\[
\texttt{INTEGER: 6}
\]

as expected.

### 13.1 The parts of a recursive function

Recursive approaches rely on the fact that it is usually simpler to solve a smaller problem than a larger one. In the factorial problem, trying to find the factorial of \( n - 1 \) is a little bit simpler than finding the factorial of \( n \). If finding the factorial of \( n - 1 \) is still too hard to solve easily, then find the factorial of \( n - 2 \) and so on until we find a case where the solution is easy. With regards to factorial, this is when \( n \) is equal to zero. The easy-to-solve code (and the values that gets you there) is known as the base case. The find-the-solution-for-a-simpler-problem code (and the values that get you there) is known as the recursive case. The recursive case usually contains a call to the very function being executed. This call is known as a recursive call.

Most well-formed recursive functions are composed of at least one base case and at least one recursive case.

### 13.2 The greatest common divisor

Consider finding the greatest common divisor (gcd) of two numbers. One approach is use repeated division. The first division divides the two numbers in question, saving the remainder. Now make the divisor the dividend and the remainder the divisor. Repeat this process until the remainder is zero. At that point, the current divisor is the gcd.

Let’s turn it into a function, first iteratively and then recursively.

```sway
#include("basics")

// compute the gcd of two positive integers iteratively

function gcd(first,second) {  
  // ...

}


```
{  
var remainder;

do-until (remainder == 0)  
{  
remainder = first % second;
first = second;
second = remainder;
}

first;  //first holds the last divisor
}
```

The do-until loop insures that the loop body is executed before the remainder is tested. Let’s rewrite the function recursively. First we identify the recurrence case(s) and then the base case(s). In this case, the recurrence and base cases are a little different than in the factorial example. We recur 3 if the remainder is not zero. We immediately return the answer if the remainder is zero. Now that the recurrence and base cases have been identified, we can write the recursive version without too much trouble.

```
// compute the gcd of two positive integers

function gcd(first,second)  
{  
if (first % second == 0)  
{  
second;
}
else  
{  
gcd(second,first % second);
}
}
```

or:

```
function gcd(first,second)  
{  
if (first % second == 0,second,gcd(second,first % second));
}
```

or, to remove the repeated computation of the remainder:

```
function gcd(first,second)  
{  
var remainder = first % second;
if (remainder == 0,second,gcd(second,remainder));
}
```

Again, the recursive version is more compact.

---

3The word is recur, NOT recurse!
Look at how the recursive version turns second into first by passing second as the first argument in the recursive call. By the same token, remainder becomes second by nature of being the second argument in the recursive call. To convince yourself that the routine really works, modify gcd to inspect the arguments:

```javascript
function gcd(first,second)
{
    var remainder = first % second;
    inspect(array(first,second,remainder));
    if (remainder == 0,second,gcd(second,remainder));
}
```

You haven’t learned about arrays or the built-in array function yet, but think of it as a way to group multiple things together. Inspecting the array is a trick to get multiple values inspected with a single call to inspect.

```console
sway> gcd(66,42);
array(first,second,remainder) is [66,42,24]
array(first,second,remainder) is [42,24,18]
array(first,second,remainder) is [24,18,6]
array(first,second,remainder) is [18,6,0]
INTEGER: 6
```

Note, how the first remainder, 24, keeps shifting to the left. In the first recursive call, the remainder becomes second, so the 24 shifts one spot to the left. On the second recursive call, the current second, which is 24, becomes first, so the 24 shifts once again to the left.

### 13.3 The Fibonacci sequence

A third example of recursion is the computation of the $n^{th}$ Fibonacci number. The Fibonacci series looks like this:

| n  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | ...
|----|---|---|---|---|---|---|---|---|---|----|
| Fib(n) | 0 | 1 | 1 | 2 | 3 | 5 | 8 | 13 | 21 | ...

and is found in nature again and again. In general, a Fibonacci number is equal to the sum of the previous two Fibonacci numbers. The exceptions are the zeroth and the first Fibonacci numbers which are equal to 0 and 1 respectively. Voila! The recurrence case and the two base cases have jumped right out at us! Here, then is a recursive implementation of a function which computes the nth Fibonacci number.

```javascript
// compute the nth Fibonacci number
// n must be non-negative!

function fibonacci(n)
{
    if (n == 0) return 0;
    if (n == 1) return 1;
    fibonacci(n-1) + fibonacci(n-2);
}
```

---

4 Pineapples, the golden ratio, chambered nautilus, etc.
Our implementation is straightforward and elegant. Unfortunately, it’s horribly inefficient. Unlike our recursive versions of \textit{factorial} and \textit{gcd}, which recurred about as many times as the iterative versions looped, our Fibonacci version will recur many, many more times than an iterative version of Fibonacci when computing larger Fibonacci numbers. Here’s why.

Consider the call to \texttt{fib(6)}. Tracing all the recursive calls to \texttt{fib}, we get:

\[
\text{fib(6) is fib(5) + fib(4)}
\]

Replacing \texttt{fib(5)} with \texttt{fib(4) + fib(3)}, we get:

\[
\text{fib(6) is fib(4) + fib(3) + fib(4)}
\]

We can already see a problem, we will compute \texttt{fib(4)} twice, once from the original call to \texttt{fib(6)} and again when we try to find \texttt{fib(5)}. If we write down all the recursive calls to \texttt{fib(6)} with each recursive call indented from the previous, we get a structure that looks like this:

\[
\begin{aligned}
\text{fib(6)} \\
& \quad \text{fib(5)} \\
& \quad \quad \text{fib(4)} \\
& \quad \quad \quad \text{fib(3)} \\
& \quad \quad \quad \quad \text{fib(2)} \\
& \quad \quad \quad \quad \quad \text{fib(1)} \\
& \quad \quad \quad \quad \quad \quad \text{fib(0)} \\
& \quad \quad \quad \quad \text{fib(1)} \\
& \quad \quad \quad \text{fib(2)} \\
& \quad \quad \quad \quad \text{fib(1)} \\
& \quad \quad \quad \quad \text{fib(0)} \\
& \quad \quad \text{fib(3)} \\
& \quad \quad \quad \text{fib(2)} \\
& \quad \quad \quad \quad \text{fib(1)} \\
& \quad \quad \quad \quad \text{fib(0)} \\
& \quad \quad \text{fib(1)} \\
& \quad \text{fib(4)} \\
& \quad \quad \text{fib(3)} \\
& \quad \quad \quad \text{fib(2)} \\
& \quad \quad \quad \quad \text{fib(1)} \\
& \quad \quad \quad \quad \text{fib(0)} \\
& \quad \quad \quad \quad \text{fib(1)} \\
& \quad \quad \quad \text{fib(2)} \\
& \quad \quad \quad \quad \text{fib(1)} \\
& \quad \quad \quad \quad \text{fib(0)}
\end{aligned}
\]

We would expect, based on how the Fibonacci sequence is generated, to take about six ‘steps’ to calculate \texttt{fib(6)}. Instead, ultimately there were 13 calls to either \texttt{fib(1)} or \texttt{fib(0)}\textsuperscript{5}. There was a tremendous amount of duplicated, and therefore wasted effort. So let’s try to compute Fibonacci numbers using an iterative loop:

\[
\begin{aligned}
\text{function fib(n)} \\
\end{aligned}
\]

\textsuperscript{5}13 is a Fibonacci number. Curious!
In the loop body, we see that \( \text{fib} \) is much like \( \text{gcd} \); the second number becomes the first number and some combination of the first and second number becomes the second number. In the case of \( \text{gcd} \), the combination was the remainder and, in the case of \( \text{fib} \), the combination is sum.

As with factorial, hitting on the right way to proceed iteratively is not exactly straightforward, while the recursive version practically wrote itself. Noting the similarity of \( \text{fib} \) and \( \text{gcd} \) suggest a way to have both recursion and efficiency at the same time.

### 13.4 More on recursive loops

To transform an iterative loop into a recursive loop, one first identifies those variables that are changing in the loop body; these will become formal parameters in the recursive function. For example, the \( \text{fib} \) loop above has three (not two!) variables that are being changed during each iteration of the loop: \( \text{first} \), \( \text{second} \), and \( i \). So, we start out our recursive function as thus:

```javascript
function loop(first, second, i)
{
    ... 
}
```

The loop test becomes an \texttt{if} test in the body of the \texttt{loop} function:

```javascript
function loop(first, second, i)
{
    if (i < n)
    {
        ...
    }
    else
    {
        ...
    }
}
```

The \texttt{if-true} block becomes the recursive call. The arguments to the recursive call encode the updates to the loop variables. The \texttt{if-false} block becomes the value the loop attempted to calculate:
function loop(first, second, i) {
    if (i < n) {
        loop(second, first + second, i + 1);
    } else {
        first;
    }
}

Using function call syntax for if shortens up the function:

function loop(first, second, i) {
    if (i < n) loop(second, first + second, i + 1), first);
}

Next, we embed the loop function into our function containing the original loop. That way, any non-local variables referenced in the test or body of the original loop will be visible to the loop function:

function fib(n) {
    function loop(first, second, i) {
        if (i < n) loop(second, first + second, i + 1), first);
    }
    ...
}

Finally, we call the loop function with the initial values of the loop variables:

function fib(n) {
    function loop(first, second, i) {
        if (i < n) loop(second, first + second, i + 1), first);
    }
    loop(0, 1, 0);
}

For more practice, let’s convert the iterative version of factorial into a recurise function using this method. We’ll see we’ll end up with a different recursive function than before.

function factorial(n) {
    var i;
    var product = 1;
for (i = 1, i <= n, i = i + 1)
    {
        product = product * i;
    }
product;
}

We start, as before, by working on the loop function. In this case, only two variables are changing in the loop: product and i.

function loop(product,i)
    {
        ...
    }

Next, we write the if expression:

function loop(product,i)
    {
        if (i <= n,loop(product * i,i + 1),product);
    }

Next, we embed the loop function and call it:

function factorial(n)
    {
        function loop(product,i)
            {
                if (i <= n,loop(product * i,i + 1),product);
            }
        loop(1,1);
    }

The moral of this story is that any iterative loop can be rewritten as a recursion and any recursion can be rewritten as an iterative loop. Moreover, in good languages, there is no reason to prefer one way over the other, either in terms of the time it takes or the space used in execution. Use a recursion if that makes the implementation more clear, otherwise, use an iterative loop.

Someday, Sway will be a good language.
Chapter 14

Input and Output

Sway uses a port system for input and output. When Sway starts up, the current input port defaults to stdin (the keyboard) and the current output port defaults to stdout (the screen).

To change these ports, one first creates new port and then sets the port. For example, to read from a file (say "data") instead of the keyboard, first create a file port:

```swift
var p = open("data", :read);  // p points to a port
var oldInput = setPort(p);
...  // read stuff from the file data
setPort(oldInput);  // restore the old input port
```

Once the port is set, all input will come from the new port. The `setPort` function, in addition to setting the port, returns the old port so that it eventually can be restored.

To change the output port, the procedure is similar, except the symbol :write is used instead.

```swift
var p = open("data", :write);  // p points to a port
var oldOutput = setPort(p);
...  // write stuff to the file data
setPort(oldOutput);  // restore the old output port
```

Opening a file in :write mode overwrites the file; to append content to an existing file, use the :append symbol instead.

Sway only allows a limited number of ports to be open at any given time. If you no longer need a port, close it with the built-in function close, which takes a port as its sole argument:

```swift
close(p);
```

14.1 Reading

Sway supplies built-in functions for reading characters, integers, reals, strings, and whitespace delimited tokens:

```swift
s = readChar();
```
Both the `readChar` and the `readToken` functions return strings. Sway uses the same rules as the C programming language for what characters constitute an integer and a real. None of these functions take an argument; they use the current input port.

To read a symbol, use the `symbol` function in conjunction with the `readToken` function:

```sway
s = symbol(readToken());
```

To read a line of text, use the built-in `readLine` function:

```sway
l = readLine();
```

The `readLine` function reads up to, and including, the next newline character, but the newline is not part of the returned string.

The `pause` function always reads from `stdin`, regardless of the current input port. It reads (and discards) a line of text (up to and including the newline). Its purpose is to pause execution of a program for debugging purposes.

### 14.2 Writing

Most output functions write to the current output port.

The simplest output function is `display`. It takes a single argument, which can be any Sway object:

```sway
display("Hello, world!\n");
```

The character sequence followed by `n` indicate that a newline is to be displayed.

More useful than `display` are the functions `print` and `println` in that they take any number of arguments:

```sway
print("(f(x) is ",f(x),"\n");
println("(f(x) is ",f(x));
```

The `println` function is just like `print`, except it outputs a newline after the displaying the last argument. Thus, the two calls above produce the same output.

When a string is printed, the quote marks are not displayed. Likewise, when a symbol is printed, the colon is not displayed.

The `inspect` function is discussed in more detail in Chapter 10, but, in short, prints out the unevaluated version of its argument followed by its evaluated argument:
14.3 Pretty printing

There are three built-in pretty printing functions: `pp`, `ppFlat`, and `ppObject`. They print reasonable representations of all Sway objects to the current output port. The difference between `pp` and `ppFlat` is that `ppFlat` coalesces all contiguous white space (including tabs and newlines) into a single space, thus printing the representation on a single line.

The `ppObject` function is used for printing Sway functions in object form. Given the function:

```sway
function f(x)
{
    var y = x + 1;
    y * y;
}
```

the expression `println(f)` outputs:

```plaintext
<function f(x)>
```

The expression `pp(f)` outputs:

```sway
function f(x)
{
    var y = x + 1;
    y * y;
}
```

The expression `ppObject(f)` outputs:

```plaintext
<FUNCTION 2626>:
context: <OBJECT 807>
prior: :null
filter: :null
parameters: (x)
code: { var y = x + 1; y * y }
name: f
```

14.4 Formatting

The `fmt` function can be used to format numbers and strings if the default formatting is not acceptable. It uses the C programming language formatting scheme, taking a formatting specification as a string, and the item to be formatted. The function returns a string.

For example,
A format specification begins with a percent sign and is usually followed by a number representing the width (in number of characters) of the resulting string. If the width is positive, the item is right justified in the resulting string. If the width is negative, the item is left justified. After any width specification is a character specifying the type of the item to be formatted: d for an integer, f for a real number, and s for a string.

The format specification is quite a bit more sophisticated than shown here. You can read more on a Linux system by typing the command `man 3 printf` at the system prompt.

### 14.5 Testing for end of file

The `eof?` function can be used to test whether the last read was successful or not. The function is NOT used to test if the next read would be successful. Here is a typical use of `eof?` in tokenizing a file:

```rust
t = readToken();
while (eof?() == :false)
{
    store(t);
    t = readToken();
}
```

### 14.6 Pushing back a character

Sometimes, it is necessary to read one character too many from the input. This happens in cases like advancing past whitespace in the input. Here is a typical whitespace-clearing loop:

```rust
ch = readChar();
while (space?(ch))
{
    ch = readChar();
}
```

//last character read wasn't whitespace
//so push it back to be read again later

```rust
pushBack(ch);
```

The `pushBack` function takes a string as its sole argument, but only pushes back the first character of the string; subsequent characters in the string are ignored.
Chapter 15

Arrays and Lists

There are three main data structures built into Sway: arrays, lists, and objects. A data structure is simply a collection of bits of information\(^1\) that are somehow glued together into a single whole. Each of these bits can be be accessed individually. Usually, the bits are somehow related, as well, so the data structure is a convenient way to keep all these related bits nicely packaged together.

15.1 Arrays

An array is a data structure with the property that each bit of information can be accessed just as quickly as any of the others. To create an array, one uses the array function:

\[
\text{var } a = \text{array}(42, 64.0, "hello", :world);
\]

This call creates an array that packages together four items or elements. The variable \(a\) is created and set to point to the package.

15.1.1 Getting array elements

To access individual elements, one uses square bracket notation or dot notation. For example, here’s how to access the first item in an array using each of the notations:

\[
\text{sway}> a[0];
\]

INTEGER: 42

\[
\text{sway}> a . 0
\]

INTEGER: 42

Note that array indexing (the number between the brackets or after the dot is known as an index) is zero-based. That is, the first element is at index 0, the second at index 1, and so on. Usually, the dot notation is not used for arrays, because of the high precedence of the dot. For example:

\[
a[\text{row} * \text{ncols} + \text{col}]
\]

is not the same as:

\(^1\) Bits in the informal sense, not zeros and ones.
a . row * ncols + col

With the later, the element at the row index is retrieved and then that element is multiplied by ncols and then the resulting product is added to col. In other words, the following two expressions are equivalent:

\[
\text{a . row} \times \text{ncols} + \text{col} \\
(\text{a . row}) \times \text{ncols} + \text{col}
\]

To make the latter the same as the former, one would parenthesize the index calculation:

\[
\text{a . (row} \times \text{cols} + \text{col)}
\]

With bracket notation, this parenthesization is performed automatically.

One can also retrieve all of the elements of an array except the first element, at the tail end of an array. The tail end of an array is the array without the first, or head, element.

\[
\text{sway> tail(a);}
\text{ARRAY: [64.000000000,"hello",:world]}
\]

\[
\text{sway> tail(tail(a));}
\text{ARRAY: ["hello",:world]}
\]

If you keep taking the tail of an array, eventually you end up with the symbol :null:

\[
\text{sway> tail(tail(tail(tail(a))));}
\text{SYMBOL: :null;}
\]

There is a head function which gets the first element of an array. These two expressions are equivalent:

\[
\text{head(a)} \\
\text{a[0]}
\]

### 15.1.2 Setting array elements

One can change the elements in an array as well, via the assignment operator:

\[
\text{sway> a[0] = :forty-two;}
\text{SYMBOL: :forty-two}
\]

\[
\text{sway> a[0];}
\text{SYMBOL: :forty-two}
\]
15.1.3 Allocating empty arrays

It is possible to allocate an array without specifying the actual elements:

```sway
define b = allocate(10);
```

This creates an array of ten elements (with indices ranging from zero to nine).

```sway
sway> b[0];
SYMBOL: :null;
```

Each of the elements is initialized to :null.

15.2 Lists

A list is a data structure that has the property that it can be lengthened and shortened by inserting or removing elements at the front, in the middle, or at the end of the list. The trade-off for this flexibility is that elements further down in the list take longer to access than those elements earlier in the list. Contrast this behavior with arrays where each element can be accessed or updated in the same amount of time.

Creating and accessing elements in a list is very similar to arrays:

```sway
define a = list(42,64.0,"hello",:world);
sway> a[2];
STRING: "hello"
sway> a[3] = 13;
INTEGER: 13
sway> a . 3;
INTEGER: 13
```

To make a list longer, one can easily add an element to the front of an existing list:

```sway
sway> var b = "apple" join a;
LIST: ("apple",42,64.000000000,"hello",:world)
```

```sway
sway> a
LIST: (42,64.000000000,"hello",:world)
```

Note that joining an element to a list leaves the original list unmodified.

You cannot use `join` to add an element to the end of a list. Instead, you must `concatenate` the original list with a new list containing the element to appended. The plus operator is used for list concatenation:

```sway
sway> var c = a + list("apple");
ARRAY: [42,64.000000000,"hello",:world,"apple"]
```
Note that you get an array as a result of concatenation. As with joining, the original list is unchanged.

15.3 Mixing arrays and lists

You can concatenate lists together with arrays. If you do so, an array results:

```sway
sway> array(1,2,3) + list(4,5,6);
ARRAY: [1,2,3,4,5,6]
```

If you join an element to an array, you get this mad-scientist amalgamation of list and array:

```sway
sway> var d = "zero" join array("one","two","three");
LIST: ("zero" # ["one","two","three"])
```

While the result looks rather strange, you can access each each element as if it were a pure list or pure array:

```sway
sway> d[0];
STRING: "zero"

sway> d[1];
STRING: "one"
```

15.4 Shallow versus deep copies

Consider the following interaction:

```sway
sway> var a = list(13,21,34);
sway> var b = 0 join a;
sway> a;
LIST: (13,21,34)
sway> b;
LIST: (0,13,21,34)
```

From the previous discussion, we know that the value of variable \( a \) is unchanged by the \texttt{join} operation that was used to create variable \( b \)'s value. Look what happens when we change \( b \)'s first element:

```sway
sway> b[0] = "zero";
sway> b;
LIST: ("zero",13,21,34)
```
As we expected, we see that \texttt{b}'s value changes while \texttt{a}'s does not. What happens when we change the second element of \texttt{b}?

\begin{verbatim}
sway> b[1] = :apple;
sway> b;
LIST: ("zero", :apple, 21, 34)
sway> a;
LIST: (:apple, 21, 34)
\end{verbatim}

Surprisingly, our change to \texttt{b} was reflected in \texttt{a} as well! To understand why, let’s look at a picture of \texttt{a} just after it is created:

The above image is known as a \textit{box-and-pointer} diagram. The boxes denote \textit{objects} that hold two pieces of information: how to get the next object (the horizontal arrow) and how to get to a value (the vertical arrow). The chain of objects is known as the \textit{backbone} of the list. The variable \texttt{a} points to the first object. When we wish to display the list, we walk from object to object, following the horizontal arrows, and displaying the value associated with each object in turn. The symbol \texttt{null} is used to indicate that there are no more following objects.

When we create \texttt{b}, this becomes the state of the world:

We now see that the \textit{join} operator creates a single object that glues a value onto an existing list. We also see why list \texttt{a} appears to be unchanged by the creation of \texttt{b} and by the changing of the first element of \texttt{b}, but is changed when the second element of \texttt{b} is changed:
The reason for this behavior is efficiency. The construction of list \( b \) took very little time since only one object was created. Consider using \( \text{join} \) to glue a value onto a list of one million elements. Without this efficiency trick, adding the new value might take a very long time indeed.

Suppose we wish to make a list \( b \) that is completely independent of list \( a \). Here is a function named \( \text{join}^* \) that attempts that very feat:

\[
\text{function } \text{join}^*(\text{value,items})
\{
    \text{if (items == :null)}
    \{
        \text{value join :null;}
    \}
    \text{else}
    \{
        \text{value join (head(items) join* tail(items));}
    \}
\}
\]

This function walks along the list, rejoining up elements with new objects. When we are done, \( b \) will have its own backbone separate from \( a \’s. \)

The problem is that the copy made by \( \text{join}^* \) is a \textit{shallow copy}. That is, the incoming list was copied, but the elements were not. In the case of a list of symbols, the elements do not need to be copied, but that is not always the case. Consider a list that has a list as one of its elements:

\[
\text{var m = list(2,3,4);}
\]

\textit{sway> var p = list(1,M,5);}
\textit{LIST: (1,(2,3,4),5)}

At this point, the second element of list \( p \) is list \( m \):
Now, let’s add the number 0 to the front of list p using `join*`.

```javascript
sway> var q = 0 join* p;       // note, join*, not join
LIST: (0,1,(2,3,4),5)
```

Now we change the second element of third element of q:

```javascript
q[2][1] = 333;
```

generating this situation:

From the illustration, we can see that the change, made through q, will modified both m and the display of p.

To make p, q, and m all independent, one needs to make a deep copy, where the elements of the list are copied and the elements of the elements are copied, and so on.

Deep copies are so difficult to do correctly, almost no modern language has deep copying built in.
15.5 Changing the tail of a list

One can use the `tail=` function to change the tail of a list:

```javascript
var a = list(1,2,3);

a tail= list(222,333,444);

sway> a;
LIST: (1,222,333,444);
```

This exchange essentially removes all but the first element of list `a`, replacing those elements with the elements 222, 333, and 444.

You can use `tail=` on any kind of list, even those produced by `tail`:

```javascript
var b = list(1,2,3);

tail(b) tail= list(333);

sway> b;
LIST: (1,2,333);
```

Here, the first two elements of list `b` are preserved.

15.6 Inserting into the middle of a list

One uses a combination of `join`, `tail`, and `tail=` to insert into the middle of a list. Here’s how to insert an item after the first element:

```javascript
var c = list(1,3,4);

c tail= (2 join tail(c));

sway> c;
LIST: (1,2,3,4);
```

The parentheses are needed to insure that the tail of `c` is not set to the number 3 and the result joined with the tail of `c`. This method of insertion changes the original list.

It is also possible to make a general-purpose insertion function. The following implementation inserts an element into a list of items at the given index (using zero-based counting):

```javascript
function insert(element,items,index)
{
    if (index == 0)
    {
        element join items; //put the element at the front
    }
```
else {
    items[0] join insert(element, tail(items), index - 1);
}
}

Here is the function in action:

```javascript
var d = list(1,3,4);
var e = insert(2,d,1); //insert 2 at index 1 of list d

sway> d;
LIST: (1,3,4);

sway> e;
LIST: (1,2,3,4);
```

Note that this `insert` function does a combination of a shallow copy of the elements preceding the inserted element and creates a sharing of the elements after the inserted element. Thus `d` is unchanged by this operation (but some changes to `e` can affect `d`).

### 15.7 Objects

Objects, in the simplest view, are like lists and arrays except elements are accessed or updated by name rather than by index. Objects are discussed in the next chapter.
Chapter 16

Objects

In the Sway world, an object is a simple a collection of related variables. You’ve already been exposed to objects, although you may not have realized it. When you created a variable, you modified the environment, which is an object. When you defined a function, you created an object. To view an object, we use the predefined function `pp` or `ppObject`.¹ Evaluating the code:

```sway
function square(x)
  {
    x * x;
  }

ppObject(square);
```

yields something similar to the following output:

```
<FUNCTION 3059>:
  context: <OBJECT 677>
  prior: :null
  filter: :null
  parameters: (x)
  code: { x * x; }
  name: square
```

We see that the `square` function is made up of six fields.² These fields are: `context`, `prior`, `filter`, `parameters`, `code`, and `name`.

Usually, an object lets you look at its individual components. For example:

```sway
println("square's formal parameters are: ", square . parameters);
```

yields:

```
square's formal parameters are: (x)
```

¹The `pp` in the function names stands for pretty print which means to print out something so it is 'pretty looking'.
²Some people use the term component or instance variable instead of field. Also, if you try this, you may see different numbers than 3059 and 677.
We use the function ‘.’ (usually called the *dot operator*) to extract the fields of an object.

It is easy to create your own objects. First you must make a *constructor*. A constructor is just a function that returns the predefined variable *this*. Suppose you want a constructor to create an object with fields *name* and *age*. Here is one possibility:

```javascript
function person()
{
    var name;
    var age;
    this;
}
```

We can create an object simply by calling the constructor:

```javascript
var p = person();
```

The variable *p* now points to a *person* object and we can use *p* and the dot operator to set the fields of the *person* object:

```javascript
p . name = "Boris";
p . age = 33;
inspect(p . name);
```

Evaluating this code yields the following output:

```javascript
p . name is Boris
```

It is often convenient to give initial values to the fields of an object. Here is another version of *person* that allows us to do just that when we create the object:

```javascript
function person(name,age) { this; }
var p = person("Boris",33);
inspect(p . name);
```

The output is the same:

```javascript
p . name is Boris
```

In general, if a field is to be initialized when the object is constructed, make that field a formal parameter. If not, make the field a locally declared variable.
16.1 Adding Methods

Objects can have methods as well. Here's a version of the person constructor that has a \textit{birthday} method.

\begin{verbatim}
function person(name, age)
{
    function birthday()
    {
        println("Happy Birthday, ", name, "!");
        age = age + 1;
    }
    this;
}

var p = person("Boris", 33);
p . birthday();
inspect(p . age);
\end{verbatim}

The output of this code is:

\begin{verbatim}
Happy Birthday, Boris!
p . age is 34
\end{verbatim}

In summary, one turns a function into a constructor by making

\texttt{this;}

or

\texttt{return this;}

the last line of a function. The local variables, including formal parameters, become the fields of the function while any local functions serve as methods.

16.2 Objects and Types

If you were to ask an object, "What are you?", it would respond, "I am an object!". The \texttt{type} function is used to ask such questions:

\begin{verbatim}
var p = person("betty", 19);
inspect(type(p));
\end{verbatim}

yields:

\begin{verbatim}
type(p) is :OBJECT
\end{verbatim}

\footnote{A method is just another name for a local function.}
While this is useful at times, often we would like to know what kind of object the object is. For example, we might like to know whether or not \( p \) is a person object. One way to do this is to ask the constructor of the object if it is the person function. Luckily, all objects carry along a pointer to the function that constructed them:

```sway
var p = person("betty",19);
inspect(p . constructor . name);
```

yields:

```bash
p . constructor . name is person
```

So, to ask if \( p \) is a person, we would use the following expression:

```sway
if (type(p) == :OBJECT && p . constructor . name == :person) ...
```

Since this construct is rather wordy, there is a simple function, named `is`, that you can use instead:

```sway
if (p is :person) ...
```

The `is` function works for non-objects too. All of the following expressions are true:

```bash
3 is :INTEGER
3.4 is :REAL
"hello" is :STRING
:blue is :SYMBOL
list(1,2,3) is :LIST
array("a","b","c") is :ARRAY
person("betty",19) is :OBJECT
person("betty",19) is :person
this is :OBJECT
```

### 16.3 A formal view of object-orientation

Sway is a fully-featured object-oriented language. What does that mean exactly? Well, to begin with, a programming language is considered object-oriented if it has these three features:

1. encapsulation
2. inheritance
3. polymorphism

Encapsulation in this sense means that a programmer can bundle *data* and *methods* into a single entity. We’ve seen that a Sway function can have local variables and local functions. So, if we consider local variables (including the formal parameters) as data and local functions as methods, we see that Sway can encapsulate in the object-oriented sense.
Inheritance is the ability to use the data and methods of one kind of object by another as if they were defined in the other object to begin with.

Polymorphism means that an object that inherits appears to be both kinds of object, the kind of object it is itself and the kind of object from which it inherits. We will learn more about inheritance and polymorphism in a later chapter.

16.4 Using the objects library

If you include the objects library, you can add toString methods to your objects. Then when you display the objects, the toString method is called to help generate the output. Here is an example:

```javascript
function person(name, age)
{
  function birthday()
  {
    println("Happy Birthday, ", name, "!");
    age = age + 1;
  }
  function toString()
  {
    name + "(age " + age + ")";
  }
  this;
}

var p = person("boris", 33);

Given the above definition of person, printing the value of p:

    println(p);

yields:

    boris (age 33)
Chapter 17

Inheritance

Inheritance, in a practical sense, is primarily about code reuse. Sway uses a simple, but powerful technique to implement inheritance: concatenation. When one object inherits from another, Sway simply joins the two objects together into a single object, with the fields and methods of the descendant object taking precedence over the fields and methods of the ancestor objects.

The following terminology is often used when discussing inheritance: object, constructor, class, ancestor, and descendant. You have already been exposed to the terms object and constructor. The term class means a description of the features an object will have when created. Sway, class is just another word for constructor. In more complicated languages, class is yet another entity separate from constructor. Ancestor refers to the class from which a descendant class inherits. Typically, inheritance bestows upon the descendant features found in the ancestor.

Commonly, a descendant inherits from one ancestor. This is known as single inheritance. Less commonly, a descendent inherits from more than one ancestor. This is known as multiple inheritance.

17.1 Single inheritance

Sway allows for three of the most common types of inheritance: extension, reification, and variation. Each of these types is discussed below.

17.1.1 Extension inheritance

The most basic extension inheritance can be illustrated by an example. Suppose I have two constructors, both of which have the same local method.\(^1\).

```sui
function matchAnyCharacter()
{
    function same?(ch) { :true; }
    function matches?(str,next) { same?(str[0]) && next . match(tail(str)); }
    function toString() { "."; }
    this;
}

function matchSpecificCharacter(char)
{
    function same?(ch) { ch == char; }
    function matches?(str,next) { same?(str[0]) && next . match(tail(str)); }
```
I wrote these functions when implementing a regular expression library for Sway. Note that the function `matches?` is the same in both. Using inheritance, we can factor out the common code so that it appears only once:

```sly
function matchCharacter() //ancestor
{
    function matches?(str,next) { same?(str[0]) && next . match(tail(str)); }  
    this;
}

function matchAnyCharacter() //descendant
{
    function same?(ch) { :true; }  
    function toString() { "."; }  
    extends(matchCharacter());  
    this;
}

function matchSpecificCharacter(char) //descendant
{
    function same?(ch) { ch == char; }  
    function toString() { char; }  
    extends(matchCharacter());  
    this;
}
```

The function `extends` appends the given object passed to the current object. Since objects are environments and environments can be viewed as tables, the `extends` function can be thought of as pasting two tables together.\(^2\)

The descendant classes are said to extend the ancestor class because they provide additional methods. In other words, they extend the behavior of the ancestor.

### 17.1.2 Reification inheritance

Reification inheritance occurs when a method of an ancestor class refers to a method that does not (yet) exist. It is the duty of the descendant class to reify the missing method.\(^3\) Note that if one creates an ancestor object and calls the referring method off that object, an undefined variable error will occur when the missing method is called. However, if one creates a descendant object and calls the referring method, then the reified method is found.

Consider the `matchCharacter` class of the previous example. The `matches?` function refers to a `same?` method, but the class does not provide it:

```sly
function matchCharacter()
```

---

\(^2\)It's a little more complicated than that, but not much more. Only the locals of the object passed to `extends` are pasted. Moreover, the context pointers of any local methods are set to the context of the object doing the extension. For this reason, inheritance in Sway is relatively expensive in some circumstances.

\(^3\)Reify is just a fancy word meaning to make real.
Because the descendant classes `matchAnyCharacter` and `matchSpecificCharacter` provide implementations of `same?`, they are said to exhibit *reification* inheritance.

### 17.1.3 Variation inheritance

One can think of the methods provided by an ancestor as the default methods for a descendant. Sometimes, a descendant may wish a particular method to do something different than as specified by the ancestor. In such cases, the descendant provides an alternative version, *overriding* the method of the ancestor. In Sway, an overriding method always takes precedence. A descendant that overrides an ancestor is said to exhibit *variation* inheritance.

Suppose, in our example, we move one of the `same?` methods up to the ancestor. In this case, we choose the `same?` method from the `matchAnyCharacter` class. Our example now exhibits variation inheritance:

```swoy
function matchCharacter()
{
    function matches?(str,next) { same?(str[0]) & next.match(tail(str)); }
    function same?(ch) { :true; }
    this;
}

function matchAnyCharacter()
{
    function toString() { "."; }
    extends(matchCharacter());
    this;
}

function matchSpecificCharacter(char)
{
    function same?(ch) { ch == char; }
    function toString() { char; }
    extends(matchCharacter());
    this;
}
```

Since the default method is exactly right for the `matchAnyCharacter` class, there is no need for that class to provide an alternative. However, the default method is incorrect for the `matchSpecificCharacter` class. Thus, that class must provide an alternative. Note that our example no longer exhibits reification inheritance.

### 17.2 Multiple inheritance

Rarely, it is convenient for an object to inherit from multiple ancestors. This is easily accomplished in Sway by making multiple calls to the `extends` function.

```swoy
function ancestor1() { ... this; }
```
function ancestor2() { ... this; }

function descendant() {
    ...
    extends(ancestor1());
    extends(ancestor2());
    this;
}

Since `extends` performs an append, if both ancestors have a method of the same name, the method from the earlier extension is used since it is ‘closer’ to the original object.

### 17.3 Retrieving overridden functions

Although an overriding function takes precedence over the overridden function, the overridden function can still be called. In every instance of variation inheritance, the overriding function holds a pointer to the overridden function. The variable holding this pointer is named `prior`. Here is an example:

```plaintext
function alpha() {
    function me() { println("I'm an alpha object"); }
    this;
}

function beta() {
    function me() { println("I'm a beta object"); }
    extends(alpha());
    this;
}

beta() . me();
beta() . me . prior();
```

This produces the output:

```
I'm a beta object
I'm an alpha object
```
Chapter 18

Exceptions

Exceptions occur when the interpreter detects an error in the code. These errors could be syntax errors or semantic errors. You’ve seen the aftermath of exceptions before: undefined variable exceptions, divide by zero exceptions, and so on.

When an exception is generated, it is said to be raised or thrown. Unless the exception is caught, it terminates processing of the current function call, then propagates upward, terminating the processing of the function that made the call, and so on and so on.

Unless the exception is caught (see the next section), the exception eventually terminates the Sway interpreter if executing a file. If Sway is running interactively, the processing terminates up to, but not including, the prompt-read-eval loop (the one with the sway> and more> prompts).

18.1 Handling exceptions

It is possible to stop the propagation of an exception by using the catch function. The sole argument to catch is an expression that may generate an exception. The catch function evaluates its argument and, if no exception was raised, returns the value of the evaluated expression. If an exception was raised, catch converts the exception into an error object.

Here is a typical use of catch:

```sway
var x = catch(process(a,b));

if (x is :ERROR && x . type == :mathError)
  {
    x = 0;
  }
else if (x is :ERROR)
  {
    throw(x);
  }
...
```

Note that if you catch an exception but it turns out you don’t wish to handle it, you can rethrow it with the throw function.
18.2 Throwing exceptions

You can generate your own exceptions with a different use of the throw function. This use of throw requires two arguments. These arguments can be anything, but, by convention, the first argument is a symbol and the second is a string giving more explanation concerning the error. For example, suppose you wish raise an exception if a variable is supposed to be even and it isn’t:

```plaintext
if (x % 2 == 1)
{
    throw(:oddValue,"x is " + x + ", it should be even.");
}
```

18.3 Try catch blocks

If you include the basics library, you can use the try function, which simplifies the catching and processing of exceptions. Here is a version of the typical usage shown previously:

```plaintext
var error;
...

try (error)
{
    x = process(a,b);
}
else if (error . type == :mathError)
{
    x = 0;
}
else
{
    throw(error);
}
...
```

The try function requires that a variable be passed as the first argument; it will set this variable to the caught exception, should an exception occur.

18.4 Returns are exceptions

The return function generates a special exception of type :
return, so the expression

```plaintext
return x + y;
```

is equivalent to:

```plaintext
throw(:return,x + y);
```
Since a return is ultimately an exception, encountering a return causes an immediate end to the processing of the body of a function.

This exception, however, is caught by the part of the interpreter that sets up the function call and is not allowed to propagate further.
Chapter 19

Lazy Evaluation

Sway provides for programmer-controlled lazy evaluation. With lazy evaluation, the evaluation of function call arguments, which usually happens before the actual call is made, can be delayed.

To delay an argument, one names the corresponding formal parameter so that it begins with a dollar sign. To perform the evaluation, one uses the force function. Here is a lazy version of the identity function:

```sway
function identity($x) { force($x); }
```

The fact that lazy evaluation is used gives us an opportunity to investigate the argument to the function before it is evaluated:

```sway
function identity($x)
{
    pp($x);
    force($x);
}
```

Consider this interaction:

```sway
sway> var a = 3;
sway> identity(2 + a);
<THUNK 11069>:
    context: <OBJECT 1047>
    code: 2 + a
    INTEGER: 5
```

From this result, one can see that the unevaluated argument is stored in something called a thunk. A thunk is a simple object that contains some code to be evaluated under some context. In this case, the context is the calling environment, exactly the same context the expression 2 + a would have been evaluated if evaluation of the argument had not been delayed.

The uses of lazy evaluation are many; one important use is described in the next section.

19.1 Infinite lists

Lazy evaluation, one can create the illusion of infinite lists. We start by overloading the join operator to delay both its arguments:
function has head

function join($head,$tail)
{
    array($head,$tail);
}

Likewise, the head and tail functions are overloaded to force the head and tail thunks created by join:

function head(cell)
{
    force(cell . 0);
}

function tail(cell)
{
    force(cell . 1);
}

Since join now does not evaluate its arguments, it can be used to join items to lists that don’t yet exist. Here is a classic use of the new join to create a seemingly infinite list of ones:

var ones = 1 join ones;

To view our new list, we need to write a special function to convert the first \( n \) elements of the list to a string representation:

function infilister(s,n)
{
    if (n == 0,"...","" + car(s) + "," + infilister(cdr(s),n - 1);
}

Now we can look at our list:

inspect(infilister(ones,10));

yields:

infilister(ones,10) is "1,1,1,1,1,1,1,1,1,1,..."

Let’s write another useful function, one that adds one to every element of an infinite list:

function inc-inf(s)
{
    1 + car(s) join inc-inf(cdr(s));
}
Now, we can create an infinite list of integers:

```javascript
var integers = 1 join inc-inf(integers);
inspect(infilister(integers,10));
```

yields:

```
infilister(integers,10) is "1,2,3,4,5,6,7,8,9,10,..."
```

### 19.2 Memoization

Using the strategy in the previous section, every time the \( n \)th element is requested from an infinite stream, all the computations needed to generate the \( n \)th element are repeated. To prevent this from happening, one can arrange for `head` and `tail` to store the forced value back into the thunk:

```javascript
function head(cell) {
    var result = force(cell . 0);
    cell . 0 . code = result;
    result;
}

function cell(cell) {
    var result = force(cell . 1);
    cell . 1 . code = result;
    result;
}
```

The next time the thunk is forced, the primitive value stored in the thunk is quickly retrieved and returned.

### 19.3 Simulating pass-by-reference

Sway, like Java, C, Python, and Scheme, binds the value of arguments to formal parameters when a function call is made. Thus, changing the value of a formal parameter generally has no effect on the variables used as arguments in the function call.

Sway has a notable exception to the general rule: thunks. If a delayed formal parameter is bound to a thunk with the unevaluated expression in the thunk being an assignable object, then assigning directly to the formal parameter has the effect of assigning to the assignable expression in the thunk. This exception is used in the implementation of `+=` in the basics library:

```javascript
var x = 3;

function +=($v,amount) {
    $v = force($v) + amount;
}
```
The above code produces:

```
x is 3
x is 4
```

As another example, consider `swap`:

```swift
function swap($x,$y)
{
    var temp = force($x);
    $x = force($y);
    $y = temp;
}
```

Even if direct assignment to a thunk were disallowed, one could still simulate pass-by-reference by taking advantage of the fact that environments and thunks are objects. Here is a somewhat longer version of `+=`:

```swift
function +=($v,amount)
{
    $v . context . ($v . code) = force($v) + amount;
}
```

Even though Sway has the ability to reach back and modify the calling environment, it is still purely pass-by-value.
Chapter 20

Variadic Functions

Generally, the number of arguments in a function call must match the number of formal parameters in the function definition. In such cases, once a function is defined, the number of arguments to that function is fixed. It is possible, however, to define functions that a variable number of arguments. This is done by using a special formal parameter named @ which must be the last parameter in the function definition’s parameter list. Functions whose parameter lists end with @ are known as variadic functions.

For variadic functions, the parameters preceding the @ parameter are known as fixed parameters. In a call to a variadic function, the number of arguments must be equal to or greater than the number of fixed parameters. Any arguments beyond those bound to the fixed parameters are gathered up into a list and bound to the @ parameter. If there are no extra arguments, @ is bound to null.

20.1 Calling a variadic function recursively

The built-in function apply is used to call a function on a list or array of arguments. Thus,

\[
\text{a + b}
\]

can be written as:

\[
\text{apply(+,list(a,b))}
\]

or:

\[
+ \text{apply list(a,b)}
\]

using infix notation.

The apply function becomes especially useful in implementing a recursive variadic function. Here is a function that sums up all its arguments. Since there is one fixed parameter in the definition, the function must be called with at least one argument:

```plaintext
function plus(x, @)
{
    if (@ = :null, x, apply(plus, list(x + @ . 0) + tail(@)));
}
```
20.2 Variadic functions with lazy evaluation

If the last formal parameter of a function is $\$\$\$ instead of $\$, then any arguments beyond those bound to fixed parameters are thunkized and bound up into a list. This list is bound to $\$. 
Chapter 21

Reflection

A language is reflexive if it can examine itself.

21.1 The type function

Any Sway entity can ask, “What type of thing are you?” using the type function.

```sway
sway> type(0);
SYMBOL: INTEGER
sway> type(3.14);
SYMBOL: REAL
sway> type("hello");
SYMBOL: STRING
sway> type(+);
SYMBOL: PRIMITIVE
sway> type(list(1,2,3));
SYMBOL: LIST
sway> type(function (x) { x; });
SYMBOL: FUNCTION
sway> type(function (x) { x; } . parameters);
SYMBOL: LIST
sway> type(function (x) { x; } . parameters . 0);
SYMBOL: VARIABLE
sway> type(function (x) { x; } . code);
SYMBOL: SIMPLE_BLOCK
```

21.2 The is function

Related to the type function is the is function:

```sway
sway> 0 is :INTEGER;
SYMBOL: true
sway> + is :PRIMITIVE;
SYMBOL: true
```

The is function is especially useful for dealing with objects:
function node(value, next) { this; }

var n = node(3, :null);

sway> n is :OBJECT;
SYMBOL: true
sway> n is :node;
SYMBOL: true

Note that the name of the constructing function is consider the class name of the constructed object.

The is function handles inheritance as well:

```javascript
function alpha() { this; }
function beta() { extends(alpha()); this; }

var a = alpha();
var b = beta();

sway> a is :alpha;
SYMBOL: true
sway> a is :beta;
SYMBOL: false
sway> b is :beta;
SYMBOL: true
sway> b is :alpha;
SYMBOL: true
```

### 21.3 Decomposition of functions

Any user-defined function can be decomposed by extracting the name, parameter list, and body of the function:

```javascript
function f(x, y)
{
    var z = x + 1;
    z * y;
}

sway> f . name;
VARIABLE: f
sway> f . parameters;
LIST: (x, y)
sway> f . code;
SIMPLE_BLOCK: {
    var z = x + 1;
    z * y;
}
```
21.4 Decomposition of arbitrary code

Sway source code can be decomposed into smaller pieces using the following strategy:

- thunkize the source code of interest
- extract the code from the thunk
- traverse the code using head and tail
Chapter 22

Returning Local Functions

If you return a local function from a function, the local function will remember the values of the outer function. Here’s an example:

```swift
function augment(x)
{
    function dispatch(y)
    {
        return x + y;
    }
}
```

Since the definition of `dispatch` is the last action of the `augment` function, the `dispatch` function is returned. When you pass a value to the `dispatch` function, it will remember the value of `x` passed to the `augment` function and compute the sum correctly:

```swift
var a = augment(10);
var b = augment(5);

sway> a(3);
INTEGER: 13;
sway> b(3);
INTEGER: 8;
sway> a(32);
INTEGER: 42;
sway> b(16);
INTEGER: 21
```

We see that both `a` and `b` remember the particular value of `x` used in their creation.

### 22.1 Why returning local functions works

If we look at function `a` (from above) as an object:

```swift
sway> ppObject(a);
```

Suppose you have written a function that takes two arguments. Suppose further that only one of those arguments is ever going to change. There is a technique, called currying that allows you to bui
22.2 Using local functions in place of objects

Here is a stack constructor, typical of the way objects are created in Sway:

```sway
function stack()
{
    var store = :null;
    var empty? = :true;
    function push(item)
    {
        store = item join store;
        empty? = :false;
        item;
    }
    function pop()
    {
        var item = head(store);
        store = tail(store);
        empty? = store == :null;
        item;
    }
    this;
}
```

Here is a version using local functions instead of environments:

```sway
function stack()
{
    var store = :null;
    var empty? = :true;
    function push(item)
    {
        store = item join store;
        empty? = :false;
        item;
    }
    function pop()
    {
        var item = head(store);
        store = tail(store);
        empty? = store == :null;
        item;
    }
    function dispatch(msg,@)
    {
        if (msg == :push) { apply(push,@); }
        else if (msg == :pop) { apply(pop,@); }
        else if (msg == :empty?) { empty?; }
        else { throw(:stackError,"bad stack message: " + msg); }
    }
}
```
Note that the dispatch functions receives a message and applies the appropriate function to the remaining arguments. Since the definition of dispatch is the last action of the stack function, the function bound to dispatch is returned.

Here’s the new version of stack in use:

```sway
var s = stack();

s(:push,1);
s(:push,"hello");
s(:push,:world);

sway> s(:pop);
SYMBOL: :world
sway> s(:pop);
STRING: "hello"
sway> s(:empty?);
SYMBOL: :false
sway> s(:pop);
INTEGER: 1
sway> s(:empty?);
SYMBOL: :true
```
Chapter 23

Sway Libraries
23.1 Draw

The draw library uses two linux applications: flydraw and imagemagick. A draw-based program uses flydraw to render the image and the display program of the imagemagick suite to display the image.

Here is a simple program that draws a blue dot in the middle of a rectangular field:

```plaintext
include("draw");
openImage("dot",100,100);
fcircle(50,50,4,:blue);
closeImage();
showImage();
```

The first line of the program causes the draw library to be loaded. The second line opens an image file named dots.gif (the default image type is gif). The third line draws a small filled circle (colored blue) in the middle of the image. The fourth line closes the image file while the (optional) last line displays the image.

The next section, in man page format, describes the public interface of the draw library.

The Sway Draw Library

```
NAME
   draw - a Sway drawing library

SYNOPSIS
   include("draw");

DESCRIPTION
   draw is a library for Sway and is based upon the flydraw and
   imagemagick packages. There are functions for most of the
   flydraw primitives. There is also a generic function (named
   flydraw) for emitting flydraw directives that do not have
   equivalent functions in the draw library.

TOP-LEVEL FUNCTIONS
   There are three top-level functions. The first two are required, the
   third function is optional.

   function openImage(baseName,w,h)
      Start an image with base name baseName with width
      w and height h. Type of image is set using the setImageType
      function. Default image type is gif.

   function closeImage()
      Writes the image data out to the file name constructed by
```
appending the baseName with the image type.

function showImage()
    Display the image (image must be output or closed first).
The default display function is display (from the imagemagick suite of programs).

function loadImage(location)
    Load an image located at the given location, resizing it to fit in the current image size. If the location is a valid URL beginning with http://, the image is retrieved over the network. If the location is not a url, it must be a relative or absolute path to the local image.

DRAWING FUNCTIONS
In the following lines, [color] may be either an optional color symbol, or a string of 3 integers between 0 and 255, separated by commas, for the values of red,green,blue. Example color symbols are :black, :red,:blue, etc. The following function descriptions are taken from the flydraw man page.

function arc(x,y,w,h,a1,a2,[color])
    Arc segment of an ellipse of width w and height h centered at (x,y), from angle a1 to angle a2.

function arrow(x1,y1,x2,y2,[color])
    Arrow (x1,y1)-->(x2,y2), where l is the length (in pixels) of arrowhead. Arrow head length is set by the function setArrowheadLength.

function darrow(x1,y1,x2,y2,[color])
    (Synonym: dasharrow dashedarrow) Dashed arrow (x1,y1)-->(x2,y2), where l is the length (in pixels) of arrowhead. Arrow head length is set by the function setArrowheadLength.

function circle(x,y,d,[color])
    Circle of center (x,y) and diameter d (in pixels).

function fcircle(x,y,d,[color])
    (Synonym: ball disk filledcircle) Filled circle of center (x,y) and diameter d (in pixels).

function copy(x,y,x1,y1,x2,y2,filename)
    (Synonym: insert) Insert the region from (x1,y1) to (x2,y2) (in pixels) of filename to (x,y). If x1=y1=x2=y2=-1, the whole filename is copied.

function copyresized(x1,y1,x2,y2,dx1,dy1,dx2,dy2,filename)
    Insert the region from (x1,y1) to (x2,y2) (in pixels) of filename, possibly resized, to the region of (dx1,dy1) to (dx2,dy2). If x1=y1=x2=y2=-1, the whole filename is copied and resized.

function ellipse(x,y,w,h,[color])
Ellipse with center \((x,y)\), width \(w\) and height \(h\).

```sway
function fellipse(x,y,w,h,[color])
(Synonym: filledellipse) Filled ellipse with center \((x,y)\), width \(w\) and height \(h\).
```

```sway
function fill(x,y,[color])
(Synonym: flood floodfill) Flood fill the region containing \((x,y)\) with the same original color, by color.
```

```sway
function filltoborder(x,y,boundingColor,[color])
Flood fill by color the region containing \((x,y)\) and bounded by boundingColor.
```

```sway
function line(x1,y1,x2,y2,[color])
(Synonym: seg segment) Line segment \((x1,y1)\)---\((x2,y2)\).
```

```sway
function dline(x1,y1,x2,y2,[color])
(Synonym: dashedline dashline) Dashed line segment \((x1,y1)\)---\((x2,y2)\).
```

```sway
function dlines([color],x1,y1,x2,y2,x3,y3...)
(Synonym: dashedlines) dashlines n dashed line segments \((x1,y1)\)---\((x2,y2)\)---\((x3,y3)\)...
```

```sway
function hline(x,y,[color])
(Synonym: horizontalline) Horizontal line through \((x,y)\).
```

```sway
function dhline(x,y,[color])
(Synonym: dashedhorizontalline dashhorizontalline hdline horizontaldashedline) Dashed horizontal line through \((x,y)\).
```

```sway
function vline(x,y,[color])
(Synonym: verticalline) Vertical line through \((x,y)\).
```

```sway
function dvline(x,y,[color])
(Synonym: dashedverticaline dashverticalline vdline verticaldashedline) Dashed vertical line through \((x,y)\).
```

```sway
function new(x,y)
Set a new image of size \(x,y\).
```

```sway
function output(fileName)
Output the current image to fileName.
```

```sway
function pixel(x,y,[color])
Color the pixel at \((x,y)\).
```

```sway
function poly([color],x1,y1,x2,y2,x3,y3...)
(Synonym: polygon) Polygon \((x1,y1)-(x2,y2)-(x3,y3)\)...
```

```sway
function fpoly([color],x1,y1,x2,y2,x3,y3...)
(Synonym: filledpoly filledpolygon fpolygon) Filled polygon \((x1,y1)-(x2,y2)-(x3,y3)\)...
```
function rays([color], x0, y0, x1, y1, x2, y2...)  
Line segments (x0, y0)--(x1, y1), (x0, y0)--(x2, y2), ...

function rect(x1, y1, x2, y2, [color])  
(Synonym: rectangle) Rectangle with corners (x1, y1) and (x2, y2).

function frect(x1, y1, x2, y2, [color])  
(Synonym: filledrectangle) Filled rectangle with corners (x1, y1) and (x2, y2).

function square(x, y, s, [color])  
Square with sides s (in pixels) and first corner at (x, y).

function fsquare(x, y, s, [color])  
(Synonym: filledsquare) Filled square with sides s (in pixels) and first corner at (x, y).

function text(x, y, string, [color])  
Write the string at (x, y). Text size can be :small, :medium, :large or :giant (see setTextSize). Text can be :vertical or :horizontal (see setTextDirection).

function transparent(color)  
Makes color a transparent color.

function triangle(x1, y1, x2, y2, x3, y3, [color])  
Triangle with vertices (x1, y1), (x2, y2), (x3, y3).

function ftriangle(x1, y1, x2, y2, x3, y3, [color])  
(Synonym: filledtriangle) Filled triangle with vertices (x1, y1), (x2, y2), (x3, y3).

function xrange(x1, x2)  
(Synonym: rangex) Set the horizontal drawing range to [x1, x2]. Defaults to [0, xszize-1].

function yrange(y1, y2)  
(Synonym: rangey) Set the horizontal drawing range to [y1, y2]. Defaults to [ysize-1, 0].

CONFIGURATION FUNCTIONS
The following set functions return the previous value of the attribute.

function setFillColor(color)  
function getFillColor()

function setLineColor(color)  
function getLineColor()

function setTextColor(color)  
function getTextColor()
function getTextColor()

function setTextSize(size)
    size can be :small, :medium, :large, or :giant
    default is :large

function getTextSize()

function setTextDirection(dir)
    dir can be :horizontal or :vertical
    default is :horizontal

function getTextDirection()

function setImageType(type)
    type can be :gif, :jpg, :png
    default is :gif

function getImageType()

function setLineWidth(w)

function getLineWidth()

function setArrowTipSize(w)

function getArrowTipSize()

function setDisplay(s)
    default is display (imagemagick)

function getDisplay()

DIRECT COMMUNICATION FUNCTIONS
To emit a flydraw command directly, use the following function:

function flydraw(command)
Example: flydraw("circle 10,20,5,blue");

AUTHOR
Written by John C. Lusth, October, 2009.