Wind: A Flow-based Programming Language

Joshua Weinstein

2018

Contents

0.1 Preface

This book is intended as an overview of the Wind programming language. It discusses the paradigm of flow-based programming, and the principles of it. The book discusses the advantages and disadvantages of flow-based programming, while then proceeding onto the syntax and usage of the Wind language. The book also, in detail, describes the C implementation of the language. The state management, the instructions, the translation of source code, and execution are all covered.

The Wind language is not a fully fledged, general purpose programming language. It is a language with a fundamental set of computation tools and components. Wind was developed with the following goals in mind.

Goals

1. An extremely light-weight language that is highly portable.
2. A programming language which does not use dynamic memory allocation.
3. A fluid, highly dynamically typed runtime.
4. A system that allows efficient transfers of immutable data.

The most unique element of Wind is that it is a "bare bones" language. It has no abstract syntax tree, no tokenizer, parser, or standard library. It reads and executes instructions directly from source code, which transition and alter several internal buffers. From a high level overview, these buffers pass around data to one another allowing the state of the data in the program to change. This choice of design permits the language to focus on manipulating and changing data, instead of managing and allocating resources.

Overall, the hope is for this book to serve as an inspiration or reference for computing and programming in a flow-based manner.

0.2 Dedication

This book is dedicated to my mother, Vida, who has guided me and raised me to be an incredible human being I am today. I would not be who I am today without her support.

0.3 About the Author

Joshua Weinstein Grew up in the sunny, green fields of Los Angeles, California. He went to the University of California Berkeley, where he studied Computer Science and Fine Art. He now works as a software engineer, specializing in distributed cloud systems and high performance search. In his spare time, he is an active contributor to the open source community. He is passionate about the creation of new programming languages and advancing the array of new programming paradigms.

Email: jweinst1@berkeley.edu

Website: `https://github.com/jweinst1`

Chapter 1

Flow-based Programming

Flow-based programming is a paradigm of programming that deals with the flow of data over one or more destinations. These destinations could in theory be any sort of structure. For this book, and the implementation of *Wind*, they are only considered buffers; fixed sized arrays controlled with pointers and indices.

This chapter will focus on an abstract perspective of flow-based programming. It will describe the fundamental concepts, such as nodes, data, and flows. It will also discuss the arrangements of how data flows can be constructed, and give some respective examples.

This chapter serves as an introduction to flow-based programming, and a preparation for the actual Wind language.

1.1 Data

In flow-based programming, data is considered any of the values or information that are computed or processed. Numbers, integers, strings, characters, lists, are all examples of data. Nearly all programming languages type the data that they contain and compute. At the raw level, data to a computer is simply sequences of 1's and 0's. Programming languages provide much more human readable forms of data. Integers or floats to hold numbers, strings to hold texts, and numerous data structures like a *list* to hold an ordered collection of data.

An important distinction in how data is typically assigned types is whether a type is mutable or immutable. An *immutable* type is one that cannot be altered or changed after it is created. In order to alter or change it, you must create a new copy of that immutable type from existing instances of the type. For example, an integer, or any number, is an immutable type. Numbers are created from operations performed on other numbers.

$$a = 1$$
$$b = a + 1$$

In the above example, a is a variable with the value of 1, while b is a variable with the value of the sum of a and 1. Neither of them can be changed. They might be able to be recreated under the same variable name, but this is not altering the existing value. Next, let's look at an example in C that deals with mutable and immutable data.

```
1 int a = 1;
2 int* b = malloc(sizeof(int));
3 *b = a;
```

Here, b is an integer allocated dynamically on the program's heap, while a is a stack allocated variable. The main difference between them is that, b is a *mutable* data value that can be changed freely after it has been created. Yet a is immutable. The last statement, in the third, line sets the value of b to a. This means b can be considered a mutable type.

However, a and b are not both integers. b is a *pointer* to an integer, specifically a block of heap memory the size of one integer. In comparison to dynamically typed languages, b can be thought of as a one element list. In flow-based programming, the same data types have both mutable and immutable characteristics. Which characteristics of data determine it's behavior is determined by the location and the *movement* of the data.

1.1.1 Data Movement

Generally, moving data in a program means the ownership of values and data are being passed around. Some languages define specific criteria for the movement of objects and memory, like C++. *Move semantics* usually refers to the process of moving dynamically allocated objects from one owner to another. Here is a simple example.

```
1 int* a = malloc(sizeof(int) * 2);
2 a[0] = 1;
3 a[1] = 2;
4 int* b = a;
```

The memory containing enough space for two integers, pointed to by a, is passed to b. After line four, both a and b point to or "hold" the same value. Accessing through either one yields the same behavior. Changing one of the integers of a changes it in b as well.

```
printf("a is %d %d\n", a[0], a[1]);
printf("b is %d %d\n", b[0], b[1]);
b[0] = 5;
printf("a is %d %d\n", a[0], a[1]);
```

Compiling this code will display the following in the terminal:

```
1 2
1 2
5 2
```

Moving the resource always involves sharing or copying a *pointer* or *reference* to some value. Movement never writes the value itself again in a new location. Thus, immutable data cannot be moved in the context of usual move semantics.

In flow-based programming, data is copied from one node or location to another. A reference to data is never moved from owner to owner. During that copy, the data is most often transformed in some way, such that the data written to a new location is not the same as the data in the old location. The following example illustrates how a flow-based program would transform and copy data.

Example: Two node transport

```
1
2 void enterNode(int* node, int arg) {
3       *node = arg;
4 }
5
6 void transferToNode(int* startNode, int* endNode, int modify) {
7       *endNode = *startNode + modify;
8       *startNode = 0;
9 }
```

Two functions above are shown, *enterNode* and *transferToNode*. Both of these functions demonstrate the acquisition and transfer of data between two nodes. It's important to note this is a very basic example. The nodes themselves here do not contain any information of how to "modify" data that they acquire and transfer. The *modify* parameter is simply used as an example of how the next node a data value is transferred to modifies each data point it receives. Take note again that no objects, pointers, or references enter the nodes, only literal data values.

1.2 Nodes and Locations

Nodes, destinations, and locations determine the path data will travel in a flow-based program. Nodes which receive, transform, and send data can take on an incredible amount of different forms. The form of a node is usually determined by the functionality or role it plays. While the structure of a node is quite variable, it must always abide by the following rules:

Rules of Flow Nodes:

1. *A node must possess some amount of readable and writable memory.*
2. *A node must have a valid memory address that can refer to itself.*
3. *A node must live longer than the data it handles.*

The first rule is the least ambiguous. All nodes must be containers, and be able to some non-zero amount of data. Specifically, nodes must own either an array or a pointer to a block of memory. If a node has to transport data along some path, it must be able to hold data temporarily at certain steps. For example, an `enum` value would not be a valid node, as even though it evaluates to an integer, it is not a container, nor does it possess the ability to own a block of memory.

Rule two is quite simple, a node must exist or be allocated in addressable memory. A *register* is not addressable, therefore a register cannot be a valid node. Constants defined with preprocessor statements like `#define` do not have a memory address and therefore cannot be used as nodes. Compile-time constants and values can be used as nodes if they are permitted for modification during runtime.

The last rule ensures that nodes will be able to support, transform and move data from the start of the program to the end of it. The key idea behind flow-based programming is that, nodes are the supporting component of data flow. Nodes need live and be reusable throughout data flowing through the program. Specifically, nodes should not be created dynamically as data is intended to be transported. When data enters a flow, all the nodes should already be initialized and ready to pass along data.

1.2.1 Node Memory

As explained earlier, nodes must possess or own modifiable memory. Nodes can store the incoming data they acquire in a variety of ways. An important factor to consider in how a node uses memory is what kind of data the node deals with. Further, if the data is all the same type, or if the node accepts multiple data types in the same pass.

First, let's look at a node which only holds a single value, and how it can be initialized, and deinitialized:

```
1  #include <string.h>
2
3  typedef struct
4  {
5          void* data;
6          size_t size;
7          int occupied;
8  } SingleNode;
9
10 void SingleNode_init(SingleNode* node, size_t size) {
11          node->size = size;
12          node->data = malloc(size);
13          node->occupied = 0;
14 }
15
16 void SingleNode_deinit(SingleNode* node) {
17          free(node->data);
18          memset(node, 0, sizeof(SingleNode));
19 }
```

Let's break down the above code. First, we see the `struct` describing the attributes of the node. It has a `void` pointer to a block of memory, a field indicating the size, and a field acting as boolean to check if the node currently possess a data value. This implementation allows any C type to be placed and written into the memory of the node. However, in order to access the data we

must cast the data pointer as some C type. While this isn't problematic, it urges the need for the types of data to be tracked externally to the node.

In this setup, initializing a node requires that the node itself, the scalar type, already be declared and stored somewhere. The pointer to the node body is passed in, initializing it with it's memory and size. Then, once the node is done being used in the program, or needs to be resized, it can be de-initialized, by freeing the memory and clearing the data in the node's body. We want to permit the node itself to be stored in a centralized location, rather than allocating the node itself with a call to `malloc()` each time it is initialized.

A downside to this approach is that it decouples what is essentially the metadata of the node, and the data of the node. It requires two steps to have a fully functional node. Next, let's look at nodes that combine both it's data and information in a single scalar body.

```c
#include <stdlib.h>

typedef struct
{
    size_t capacity;
    size_t len;
    unsigned char data[];
} BlockNode;

BlockNode* BlockNode_new(size_t size) {
       BlockNode* node = malloc(sizeof(BlockNode) + size);
       node->capacity = size;
       node->len = 0;
       return node;
}
```

Note: The above uses a flexible array parameter, a special array available since C99. Flexible array parameters must always be the last members of a struct, and cannot be directly used with the sizeof operator.

BlockNode is a node that uses a dummy pointer, also called a flexible array, to refer to a variably sized body of memory it carries at the end of itself. Calling `sizeof()` on BlockNode will not work directly, as every newly constructed node of this type exists with a differently sized block of memory. Therefore the size of a BlockNode is given by the sum of it's `capacity` property, and the sizeof operator. Likewise, the `len` property indicates the amount of space occupied in the node's memory body segment. This makes the BlockNode slightly more useful than the SingleNode previously discussed, as we can store differently sized data in the same node.

The *len* property also makes it fairly easy to write data into the node, as well as expand the size of the node if and when needed.

```c
// Wrapped in () to prevent macro bloat errors
#define BLOCKNODE_SIZEOF(node) \
             (sizeof(BlockNode) + node->capacity)

```

```
5  #define BLOCKNODE_EXPAND(node, extraSpace) \
6                do { \
7                node->capacity += extraSpace; \
8                size_t newSize = BLOCKNODE_SIZEOF(node); \
9                node = realloc(node, newSize);\
10               } while(0)
```

Note: In C, the function realloc() takes as arguments, an existing pointer to a dynamically allocated chunk of memory, and the new desired size for that memory. The function attempts to resize the memory chunk by simply extending it first, if the downstream addresses are unoccupied. However, the function can return a pointer with a different address than what was passed in, the event it needs to allocate and copy over the memory to a new address. Thus, a macro ideally should be used when dealing with a memory-embedded struct, as this alleviates the need to return expanded pointers to nodes.

We implemented two macros, `BLOCKNODE_SIZEOF` and `BLOCKNODE_EXPAND`. Each of them assists in expanding a BlockNode, allowing more new data to be written to it, if and when needed. However, in order for the node to acquire new data, it must increment it's *len* field and check if it currently has enough capacity. This behavior can be demonstrated with the following:

```
1  #include <string.h>
2
3  BlockNode* BlockNode_write(BlockNode* node, void* data,size_t n)
       {
4          if(n > (node->capacity - node->len)) {
5                  BLOCKNODE_EXPAND(node, (n + 10));
6          }
7          memcpy(node->data, data, n);
8          node->len += n;
9          return node;
10 }
```

Prior to the node acquiring new data, the capacity of the node needs to be checked. The remaining space for a BlockNode, or any real block of memory is given by:

$$space = capacity - length$$

Where *capacity* is the total allocated space of the memory segment, and *length* is the currently occupied boundary of the memory segment. The amount of space available is used to determine the procedure to write incoming data to the node. The cases below describe the course of action, given that n is a positive integer representing the size of the incoming data:

$$\begin{cases} write & n \leq space \\ expand \rightarrow write & n > space \end{cases}$$

Figure 1.1: Different actions taken for a BlockNode to acquire new data.

It's important to understand that although this type of node, a *BlockNode*, has to grow and expand to accumulate more data in it's own memory, nodes do not need to be implemented to grow dynamically. Nodes can use a fixed amount of memory, one that cannot be changed at runtime. These fixed-size nodes benefit from never having to check their space or expand themselves.

The type of nodes the *Wind* language uses are an even further optimization, nodes which only use static memory. In C, declaring an array with the `static` keyword tells the compiler to store that array in the resulting program's data segment. This means that the array can be available throughout the entire runtime, and uses absolute form of addressing. It does not need to wait for neither stack nor heap memory to be allocated for a node to be initialized. This implementation in Wind ill be discussed in a lter chapter.

1.2.2 Node Connections

Up until now, we have mainly described the types of nodes, their pros and cons, as well as their key behavior. Next, the manner in which nodes connect to each other, and the structure of these connections will be discussed. Despite the different ways one can connect nodes to form a *flow*, the main guideline is to not treat or handle nodes as objects. This implies that the nodes themselves and the states they might have should drive the movement of data. The transportation of data should not rely on using external data structures.

The first, most naive form of a node connection is a single direction link. This allows a one way from of travel from one node to another.

```
struct Node {
        size_t cap;
        size_t len;
        struct Node* next;
        unsigned char data[];
};
```

The *next* property, in this arrangement, is always `NULL` or a pointer to another node. This node connection functions very similar to a linked list. In this implementation though, more arbitrary data can be stored in the node, while a linked list is most often only storing individual values in the head of the nodes. Connections like these can similar be changed to produce doubly-connected nodes, or tree like node arrangements. However, nodes do not absolutely need to be linked by their addresses. Given that a population of nodes is constant or near constant over time, certain nodes can be programmed to pass data to other specific nodes.

Nodes can specialize themselves in a flow-based program by assuming specific roles. The roles allow nodes to form a more consistent architecture. This also makes a node arrangement adhere more closely to the lifetime rule of nodes. The longer and more constant a path of nodes exists, the less chance of errors or data leaks.

1.3 Flows

The final topic to introduce in flow-based programming are flows themselves. A flow is the path and direction data follows as it traverses from node to node in a flow-based program. In a previous section, node *connections* were discussed. The difference between a flow and a connection is that, a connection between nodes describes a localized, singular transition, while a flow describes the pathway of data through a program from input to output. To make this distinction, we can describe the simplest possible flow, a flow with no nodes at all. Let's take a look at it visually:

$$I \longrightarrow O$$

Figure 1.2: A flow with zero nodes.

where I is the *input* of the program, and O is the *output* of the program. In this flow, there are no nodes present to manipulate or act whatsoever on the incoming data. Thus, data is simply moved from input to output. There is no acquisitions or transfer of data, it's simply written right back out. From a code perspective, this would occur when command line arguments to a program are just written directly back to `stdout` or `stderr`. This is illustrated in the following program:

```
1 #include <stdio.h>
2
3 int main(int argc, const char** argv) {
4         for(int i=1 ; i<argc ; i++) {
5                 puts(argv[i]);
6          }
7   return 0;
8 }
```

This program loops over the arguments the program is executed with, and prints them as strings to `stdout`. The arguments are never even written to any dynamic or static memory within the program. Flow-based programs with no nodes have no real effect on any arguments or data they consume.

1.3.1 Single Node Flows

Flows with a single node have more control over their data then zero-node flows, yet are restricted to essentially one operation. A flow with a single node only has one "stop", before it reaches it's destination, the output of the program. Here is a diagram of a single node flow:

$$I \longrightarrow N_1 \longrightarrow O$$

Figure 1.3: A flow with one node.

Here, N_1 is the node in the flow. It is enumerated with 1 because in addition to being the only node it's also the first in the flow. In this arrangement, the

flow has a single point in which manipulation of data occurs. However, there is no rule in a flow that the total amount of data coming into the program must be the same as the amount coming out. One possible role of a single node flow is to filter unwanted data, and only allow data that meets some criteria to reach the output of the program. This can be shown as:

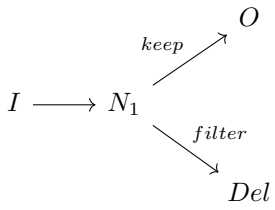


Figure 1.4: A flow with one node that filters.

Del is not a node, but rather a terminal destination that signifies the data is deleted, and will not reach the output of the program. This diagram accurately conveys the path in which data could take in a filtering situation. Yet, there are several operations in which the amount of data from input to output stays the same. For example, a single node flow can map each data value it's node acquires. It could also reduce all the data that passes through it's node to a single value. Overall, a singular node flow supports only one operation that can be performed on data.

1.3.2 Two Node Flows

Flows with two nodes in them offer a tremendous increase in utility and functionality over flows with a single node. First, having two nodes at the basic level allows for more than one operation to be performed on any incoming data. Such as the operations discussed previously, data can me mapped in one node and filtered in another, or mapped in both. The first and most direct advantage of a dual node flow is an additional operation.

With two nodes, the flow can also be arranged in a fashion where the first node is a filter, and data that does not pass the filter is passed to the second node, where it is mapped to the output. Data that does pass the filter test is simply carried to the output directly. This is called a *fork* node, and is illustrated below:

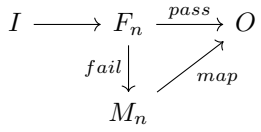


Figure 1.5: A dual flow node with a conditional mapping arrangement.

In the above diagram, F_n is a filtering node, and M_n is a mapping node.

When data arrives at F_n from the input, it is tested on the node's filter. If it passes, it goes straight to the output, O. If it does not pass, it is transferred to M_n, the mapping node. From there, data is mapped directly to the output. The data that travels through this flow forms a *one way path*. There is, one, and only one direction the data can travel. The data cannot backtrack, it has no way of moving back toward the input.

A dual node flow can also be arranged to act on data recursively. This setup is very similar to the filter-map approach. Except, data is continuously checked against the filter, then mapped if it does not fit the filter's criteria. Data in such a flow can only progress to the output if it initially or eventually passes the filter.

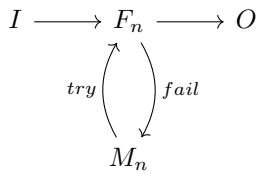


Figure 1.6: A dual flow node with a recursive loop.

In this flow, there is a recursive *loop*, where once data fails the filter, it is mapped, and then tested in the filter node again. This loop continues until the data passes the filter. The main concept here is that, the value of the data is different every time it passes through the filtering node. This procedure mimics the computational concept of recursion. The data is checked continuously against a base case, while if failing it, is incremented in some way.

Overall, using two or more nodes in a flow greatly increases the flow-based program's utility. The exact flow implementation that the Wind language uses will be discussed in a later chapter.

Chapter 2

Setup, Syntax, and Types

In this chapter, the building and setup of Wind from source code will be discussed. We will also go over the basic syntax and data types present in the wind language. The Wind language also features several configurable options prior to compilation. The chapter will explain how to change and use those options for different scenarios.

From a high level perspective, Wind is compiled from a repository of C files and header files. This compilation is done via a *Makefile*, which produces an executable. That executable is runnable via a terminal, and can accept command line arguments. Currently, Wind code can be run interactively, in a *REPL*. Wind can also run code from strings, and with different options, produce debug and performance information.

2.1 Download and Building

The Wind source code is stored, managed, and maintained in a git repository on GitHub. If you want to download or visit the repository, you can go to the following link:

```
https://github.com/jweinst1/Wind
```

In order to download the repository to your local computer, you need to *clone* it. As long as you are on a UNIX operating system, you can run the following command in your terminal:

```
$ git clone https://github.com/jweinst1/Wind.git
```

Note:The "$" as in most conventions is not part of the command, it symbolizes the current user of the terminal.

Next, go through and perform each of the following steps to build Wind:

```
$ cd Wind
$ make build
```

These commands will create a *bin* directory which contains an executable named *Wind*. This executable is the program that actually translates and executes Wind source code. The Wind repository does not come with any installers, but you can move the executable to any folder you wish. Later in this chapter, the different command line arguments for the executable will be discussed. However, for now, the "-h" flag will print out the health menu:

```
$ ./bin/Wind -h
The Wind Programming Language Help Guide
          save command:
               The save command allows you to save the current active data
               to a file path. This file is written in the Wind binary format.
               Using this allows you to save and load an infinite amount of states
               that Wind can operate and run from.
               __example__: push 3 5 -> map ** 4 | + 4 -> save "nums"
               Saved at: nums
               ----------------------
               A file named 'nums.bwind' will appear on disk.
          load command:
               The load command allows you to load binary wind files into the
               current active data.
               This command permits Wind to load data from an infinite
               number of sources,
               and expand it's processing capabilities to an immense degree.
               __example__: push 3 5 -> map ** 4 | + 4 -> save "nums" ->
                           clr -> load "nums" -> out
               [ 85 629 ]
          reduce command:
               The reduce command fuses data through a flow of operations.
               It can be used with operations like +.
               __example__: push 3 3 3 3 3 -> reduce + -> out
               [ 15 ]
          filter command:
               The filer command restricts data through a flow of operations.
               It can be used with operations like < or > or !.
               __example__: push 5 4 -> map + 3 | * 3 -> out -> filter > 22 -> out
               [ 24 21 ]
               [ 24 ]
           map command:
               The map command transforms data through a flow of operations.
               It can be used with operations like +, - and more.
               __example__: push 5 5 -> map + 3 | * 3 -> out
               [ 24 24 ]
           push command:
               The push command appends data to the end of the active data.
               __example__: push 5 6 7 -> out
```

```
   [ 5 6 7 ]
out command:
    The out command prints the entire active data to stdout.
clr command:
    The clr command resets the active data.
```

2.1.1 Compiling Options

Upon running the build commands for Wind, you might see some messages in your terminal that look like this:

```
gcc -c -Wall -Iinclude
-DWindData_BUF_SIZE=50000
-DWindData_LOAD_SIZE=10000
-DWindComp_BUF_SIZE=2000 -c
-o lib/flow/WindState.o src/flow/WindState.c
```

The above message contains several definitions, which are values passed into macros of the C preprocessor. In the general sense, this allows values to be defined inside source code externally upon compilation, versus manually editing source code. In this case, all of the names appearing after $-D$ are such flags. These flags control the amount of memory used in the buffer nodes of the language internally.

As discussed in the beginning of this book, a unique feature of Wind is that it does not use dynamically allocated memory, also called "heap" memory. It relies on static buffers that have memory within the data segment of the executable. This allows Wind to be compiled in a scalable fashion, and it's resource be throttled or increased as needed. Additionally, this feature permits Wind to easily run on embedded systems. Although the specific C-level implementation of Wind has not been discussed yet, the following configuration options can be found in the `Makefile` of the repo:

Wind Compiler Configurations

- **WindData_LOAD_SIZE**: *This option controls the size of load buffer. In Wind, the load buffer accounts for the maximum amount of size of the arguments passed to each command. For now, commands are similar to function calls, such as $f(x, y)$. The size of this buffer is generally small, but it can be customized depending on the size of argument lists needed to be processed.*

- **WindData_BUF_SIZE**: *This option controls the size of the active and inactive buffers. These buffers hold the data in Wind that can actively be extended, cleared, and manipulated with various operations. The size of this option should always be larger than the load or comp buffer options.*

- **WindComp_BUF_SIZE**: *This option controls the size of the computation buffer. In the Wind flow, this buffer is used to temporarily store and*

modify data. Think of it as a larger version of a register. This config option must always be less than the load or active and inactive buffer size options.

The settings of these configurations vary depend on the sizes of data you will be handling while running Wind code and whether or not you are building Wind for an embedded system. A general relationship between the different options though can be be described as:

$$C_o << L_o < B_o$$

where C_o is the comp size option, L_o is the load size option, and B_o is the active and inactive buffer size option. The size of the comp option should always be *far* smaller than the load option.

2.2 Command Line Arguments

The Wind executable that comes from building the language has an array of different options it can be run with for different uses. Command line options in this case are also referred to as "flags". These flags can be used alone or with additional arguments, depending on their purpose. For the following examples, the Wind code used will be: $push\ 5 \longrightarrow out$. Although specific commands in the Wind language have not been explained, we can interpret this code as: *push the integer 5 onto the active buffer and write it to stdout.*

2.2.1 Running Strings of Code

In order to run a string of Wind source code, you can choose an option that allows you to specify a string of code as one of the command line arguments. The first of this, is the straight *compile* option, written as "-c":

```
$ ./bin/Wind -c "push 5 -> out"
[ 5 ]
```

Upon executing this command, the list of the active buffer is written to stdout. This is the active buffer inside the Wind runtime. The -c option can be used at any time you need to simply run a string of code. However, let's say if you want some more detailed information about the internal state of Wind, like the command, the state of the translator, and more. There is another option, $-d$:

```
$ ./bin/Wind -d "push 5 -> out"
_____Wind___Debug_______
..........State.........
Has Error: false
Mode: Command
Command: null
```

```
.........Data.........
Load Buffer: -> [ ]
Active Buffer: -> [ 5 ]
Inactive Buffer: -> [ ]

.........Computation.........
Comp Buffer: -> [ ]
------------------------
```

Using the $-d$ option runs the string of source code *and* prints out debug information about the state of the Wind executor and translator. The *state* tab above details the mode of translation, the current command found in the source code, and whether or not an error has been detected. This option is useful for visualizing the current data active in different buffers inside Wind. The computation buffer is separated because it always holds temporary values that are in transit between the flow. The exact structures of these buffers will be discussed in a later chapter.

Another option that can be used is the $-t$ option. This option runs the string of source code and also times the run. It prints out the time in seconds it took to fully translate and execute the string of code. This is a great option to check performance of the code your running.

```
$ ./bin/Wind -t "push 5 -> out"
[ 5 ]
Time: 0.000081
```

2.2.2 The Wind REPL

Aside from command line arguments, Wind also supports a read, eval and print loop (REPL). This REPL can be activated by executing the Wind executable with no arguments. The REPL prompts the terminal for input, allowing many lines of code to be run continuously. This allows an interactive feel into the flow based nature of the Wind language, and is great for testing out new ideas. Below is an example of the REPL being use:

```
$ ./bin/Wind
Wind - Version (0.0.1)
The Wind Programming Language REPL
To exit, simply enter 'exit'. For help, run with '-h' flag.
wind> push 5 6 4 3
wind> out
[ 5 6 4 3 ]
wind> map + 5 | -2 | ** 2 -> out
Error: Cannot map argument of type: 'Number'
wind> map + 5 | - 2 -> out
[ 8 9 7 6 ]
```

```
wind> exit
```

The error, in this example was attempting to map a negative number and not the *minus* symbol. Symbols and mapping will be discussed in a later chapter.

2.3 Syntax

The Wind language uses a unique syntax that emphasizes a continuous, stream-like grammar. Wind is designed to be read in a piece by piece, intermittent basis. Such that, Wind programs can be simply described as concatenation of commands, their arguments, and the arrow, $\longrightarrow$ tokens. This permits Wind source code to be easily constructed by other processes and programs. Additionally, having stream-like syntax allows code to be called in smaller increments on environments that have tight memory constraints, like embedded systems.

2.3.1 Grammar

The abstract grammar of Wind can be described as follows:

$$
\begin{aligned}
code &:= C_1 \rightarrow \dots C_n \\
command &:= name\ \ a_1 \dots a_n \\
name &:= [a - zA - Z]+ \\
argument &:= number \mid boolean \mid none \mid string \\
boolean &:= True \mid False \\
number &:= [0 - 9]+
\end{aligned}
$$

Figure 2.1: The syntactic grammar of the Wind language

However, the grammar of Wind does not have strict, enclosed rules. For example, even the code:

```
push  4 5 6 ->
```

will run without any error, because Wind is designed to be read in a continuous fashion, and be readable as a stream. This means that, like the REPL, the state of code can be preserved between each run. In fact, running the above string of code in the repl, then just the string "out", will write the previously formed active buffer in the first string to stdout. Wind also supports saving it's current internal state to a binary file and reloading that file at later time. The *save* and *load* commands will be discussed later.

2.4 Types

The Wind language uses a simple, efficient type system, that is entirely immutable. The typing system allows types to easily be copied, and written over, such as if transition of one data type to another is needed. In comparison to other programming language terminology, all of Wind's types are *primitive*, there are no constructed or initialized types that can be user defined.

The most simplistic data type in Wind is the *None* type. It has no real value, and is meant to symbolize the absence of a value. However, Wind's types also abide by trueness, where types correspond to either `False` or `True` bool types depending on their representation. Since *None* essentially implies an absent or "off" state, in operations that result in a bool output, None will be considered as `False`:

```
$ ./bin/Wind -c "push None -> map ! -> out"
[ False ]
```

2.4.1 Bools

The Bool type in Wind represents the boolean states of true and false. Those values are displayed as `True` and `False` respectively. Bool types are used extensively for filtering in Wind. For now though, we can refer to the logical *not* operation performed in previous examples. This is one operator that when used with the *map* command, turns other data types into their respective logical NOT booleans:

```
$ ./bin/Wind -c "push None True False 0 3 4 7 3 0 0  -> map ! -> out"
[ False False True False False False False False False False ]
```

Currently in Wind, *numbers* do not abide by a trueness rule, and thus since a number is not a boolean, it always evaluates to the value False. However, Bools can be evaluated to numbers in the cases of number oriented symbols being used in a command like `map`, see the example below:

```
$ ./bin/Wind -c "push 1 2 -> map + True -> out"
[ 2 3 ]
```

2.4.2 Numbers

Numbers in Wind represent numerical values. They can represent both floating pointer numbers and integers. Internally, numbers in Wind are double precision floating point numbers that are displayed as an integer if their value corresponds to one. Numbers are an important type in Wind, used for many commands such as reduction, filtering, mapping and more.

Numbers can be pushed onto the active buffer like so:

```
$ ./bin/Wind -c "push 1 2 0.0005 -> out"
[ 1 2 0.001 ]
```

Wind only displays up the third decimal place for floating point numbers, but the precision, binary value is stored internally. This makes some number operations like addition or division a little faster as real conversion between integer and floating point formats is not necessary.

Numbers in Wind are considered *safe* numbers. Numbers cannot exhibit unsafe behavior, such as being divided by zero, or overflowing. For example, if a number such as in the case of a mapping were to be divided by zero, the zero would just be treated as one, and thus a safe division. See below:

```
$ ./bin/Wind -c "push 3 4 5 -> map / 0 -> out"
[ 3 4 5 ]
```

No unsafe side effects arise from an unsafe number operation.

2.4.3 Strings

In Wind, strings currently are used for specifying file paths, and printing as apart of the active or inactive buffer. Strings cannot be concatenated yet, that will come in a future version of Wind. They can be pushed onto the stream, or used to store the internal state of the active buffer in a file. The specific commands of *load* and *save* that will be discussed in the next chapter, however the utility of strings can be seen in the sample below

```
jweinstein-MBP-71683:wind jweinstein$ ./bin/Wind
Wind - Version (0.0.1)
The Wind Programming Language REPL
To exit, simply enter 'exit'. For help, run with '-h' flag.
wind> push 4
wind> push 3 4 True
wind> out
[ 4 3 4 True ]
wind> save "data"
Saved at: data.bwind
wind> clr
wind> out
[ ]
wind> load "data" -> out
Loaded data from: data.bwind
[ 4 3 4 True ]
wind> exit
```

Strings can also be pushed onto the active buffer, and written to stdout.

2.4.4 Symbols

Unlike in other programming languages, Wind considers operators a type of their own. This, in part, allows them to be moved around freely for executing

commands, just as numbers and strings or other types are. Instead of operators being a key word or syntax token, operators are encapsulated as symbols, immutable data that represents a single value. Much like the *None* type, except symbols correspond to specific roles and operations.

Wind's symbol types consist of the following:

Symbol Types:

1. **Assign:** $=$
2. **Not:** !
3. **Delete:** *Del*
4. **Plus:** $+$
5. **Minus:** $-$
6. **Multiply:** $*$
7. **Divide:** /
8. **Power:** $**$
9. **Greater Than:** $>$
10. **Less Than:** $<$
11. **Separator:** |

The above symbols offer a wide variety of computational tools that can be used in Wind. The *Separator* symbol, |, is a special symbol that allows multiple operations inside commands like *map* or *filter* to be included in the same command. It is similar to the idea of a pipe command on operating systems like UNIX.

Chapter 3

Commands

In Wind, all computation is initiated and encapsulated inside commands. Wind's stream-like syntax eliminates the use of traditional grammars where a program is generally composed of statements, each which have their different forms. Commands are far more analogous to function calls, they have a name and the arguments the command is invoked with. Each command offers specific functionality, and can differ in it's results and side effects greatly depending on the arguments supplied to them.

One key concept behind commands is they are executed as they are read from source code, in an interpreted fashion. There is no true compilation of entire documents of source code in Wind. Source code is treated as units of instructions, commands. These are read and executed sequentially. An error occurring does not prevent previous code from properly running:

```
wind> push 3 -> out -> push 4 -> out -> c
[ 3 ]
[ 3 4 ]
Error: Expected command symbol, found 'c'
```

This chapter will go over each command, and example use cases. It will also discuss the limitations of each command, and the recommended approach for desired manipulations of data.

3.1 push

The most fundamental and often used command in Wind is *push*. The push command has shown up in previous examples in this text, and has the role of appending and writing new data to the end of the active buffer. In Wind, is it one of the two possible ways to add new data into Wind's buffers, the other being the *load* command.

The command can be used with none, one, or more arguments. Using the push command with no arguments has no effect, as exemplified below:

```
wind> push 1 2 3 -> out
[ 1 2 3 ]
wind> push -> out
[ 1 2 3 ]
```

Attempting to push an incorrectly specified type results in an error. In fact, this is true of any command used in Wind.

```
wind> push r
Error: Expected argument or value, found 'r'
```

3.2 out

The *out* command is used to write the current contents of the active buffer to the standard output. It does not use or take any arguments. Internally, it prints each of the data item's string representations. These offer a conversion between the binary representation of Wind data and a human readable version. Every type in Wind has it's own unique string representation. Below are several examples using the *out* command:

```
wind> out
[ 3 4 ]
wind> push True False None
wind> out
[ 3 4 True False None ]
wind> push "hello" -> out
[ 3 4 True False None "hello" ]
wind> push + - * / -> out
[ 3 4 True False None "hello" + - * / ]
```

3.3 clr

Perhaps the simplest command in Wind is the *clr* command, also called "clear". It erases the contents of the active buffer and sets the length of it to zero, which allows all of it's space to be reused again.

```
wind> push 3 4 5 -> out -> clr -> out
[ 3 4 5 ]
[ ]
```

The *clr* command, similar to other commands in Wind, does not error if excess arguments are used. This can be seen below:

```
wind> push 3 4 -> clr 4 5 3 -> out
[ ]
```

Part of the reasoning behind this is, if Wind intends to be a high performance language, and work on embedded systems, the flow of computation and data processing should only stop fi absolutely necessary. Errors that would only be caused by the programmer not obeying a specific rule are not essential to producing intended behavior. Wind changes the idea of safety in a programming language to more pertain to the safety of the runtime, and the platform it runs on, versus keeping track of what the programmer is doing.

3.4 map

The map command in Wind maps operations and their arguments to data. It transforms data to change values according to different procedures, such as addition, or logical NOT. Mapping in Wind involves creating and writing a new copy of that data with a transformed value. The exact mechanism for continuous immutable states will be discussed in the next chapter, but for now we can think of mapping as a function:

$$map(x, s, a_1, \ldots, a_n) \longrightarrow y$$

where x is the target data, s is the symbol, $a_1 \ldots a_n$ is the list of arguments, and y is the result data. The above relationship describes the process of a map command for a single data value. Mapping multiple operations and arguments to multiple data values is different, as the question of order arises. In Wind, one can think of the execution of a *map* command as the application of a set of tuples against a set of data:

$$\begin{aligned} S_d &= \{d_1, \ldots, d_n\} \\ S_s &= \{(s_1, a_1, \ldots, a_n), \ldots, (s_k, a_1, \ldots, a_n)\} \\ S_r &= S_d \times S_s \end{aligned}$$

Here, S_d is the initial data set, S_s is the set of tuples of symbols and their arguments, and S_r is the resulting data. The map command execution is really just the multiplication of sets and their members, which can be further simplified to:

$$map : \mathbb{D} \times \mathbb{S} \longrightarrow \mathbb{R}$$

3.4.1 Usage

To use the *map* command, it can be, as explained previously, paired with tuples of symbols and arguments, to relevant data types:

```
wind> push 1 2 3 4
wind> map + 5 | * 3 -> out
[ 18 21 24 27 ]
```

The *map* command can also be used with several other symbols, such as the logical NOT, as well as the exponent symbol:

```
wind> push 3 10 10 10
wind> map ** 2 -> out
[ 9 100 100 100 ]
wind> map * 3 | / 3 6 -> out
[ 1.500 16.667 16.667 16.667 ]
```

3.5 filter

The *filter* command in Wind works similarly to the map command theoretically. Yet, instead of transforming data it performs a test on data, to check whether or not it should retained in the active buffer, or filtered out. The filtering test can best be though of as a boolean function:

$$F_t : b^k \longrightarrow \{0, 1\}$$

where b is the input, and k is the *ary* of the function. This means that if the domain of the function is 0, then the result of the function will be constant. However, as that input changes, the arity increases, which makes the output more variable and less certain. Therefore, we can deduce a filtered set of data is:

$$S_f = \{x | x \in S_d \iff F_t(x) = 1\}$$

In Wind, when data fails a filter test, it is removed for the active buffer's data. The relationship between how the active and inactive buffer's filter data internally will be discussed later.

3.5.1 Usage

In Wind, the *filter* command can be used with two symbols, $>$ and $<$, each representing lesser than or greater than logical operations for numbers. Some examples are as follows:

```
wind> push 3 4 5 8 54 20 5 -> out
[ 3 4 5 8 54 20 5 ]
wind> filter < 5 | > 30 -> out
[ ]
wind> push 3 6 8 5
wind> filter > 3 -> out
[ 6 8 5 ]
```

As shown above, the | symbol can be used to chain multiple filter symbols and their requirements. Since the filter command is meant to divide groups of data that either pass the filter or do not pass the filter, the equals and not equals

operations not supported. More operations such as greater than or equals will be added to Wind in the future.

3.6 reduce

The *reduce* command in Wind allows data to be reduced through a symbol's applied operation. Reduction uses data already present in the active buffer, and reduces the first data value in the active buffer against the trailing data. A common example of reduction anywhere is to sum a list of numbers. Wind supports a sum operation:

```
wind> push 1 3 4 6 7 -> out
[ 1 3 4 6 7 ]
wind> reduce + -> out
[ 21 ]
wind> push 20 20 20 -> out
[ 21 20 20 20 ]
wind> reduce *
Error: Cannot run reduce operation with instruction type: 'Multiply'
wind> reduce - -> out
Error: Cannot run reduce operation with instruction type: 'Minus'
wind> reduce + -> out
[ 81 ]
```

If the reduce command is attempted with a symbol it does not support, an error is raised.

3.7 save

The *save* command in Wind allows the internal active buffer contents to be saved to a file that can be reloaded and used in the Wind runtime later on. All of the Wind buffers, which functions as node's in a flow of data, all store data in a special binary format. This format allows a non-segmented chunk of data to be treated as an ordered collection of data values. This method of encapsulating data makes it highly available for performance tight situations as well as embedded systems.

Since Wind has no system of variables or storing values as variables in an environment, keeping the data in binary sequences allows the save and load commands to leverage storing the output of processed data. Additionally, these commands allow other programs to synthesize data in Wind's binary specification. The load and save commands create a way for other programs to leverage the use of Wind.

The *save* command can be used as follows:

```
wind> push 5 6 7 8 True -> out
[ 5 6 7 8 True ]
wind> save "/sample"
Error: File path '/sample' cannot be written to.
wind> save "../foo"
Saved at: ../foo.bwind
```

If a path cannot be written to for some reason, an error is displayed.

3.8 load

The *load* command in Wind allows `.bwind` files to be read and loaded back as the active buffer in Wind's runtime. It allows full restoration of state from a previously stored binary wind file. Coupled with the *save* command, the load command allows the manipulation, filtration, and alteration of many different data sets.

The load command is used similarly to the save command, it loads a .bwind file at a specific path:

```
wind> load "../foo"
Loaded data from: ../foo.bwind
wind> out
[ 5 6 7 8 True ]
wind> push 3 -> out
[ 5 6 7 8 True 3 ]
```

similarly to the save command, if a path cannot be accessed, it will raise an error.

Chapter 4

Implementation

The Wind programming language is run with an executable that is compiled from a C source code project. Wind is written in C to make the language highly performant and compile to a small executable size. One of Wind's main goals is for the language to run on embedded systems. However, another intention of Wind is to create a language which can also serve higher level abstractions (mapping, filtering, reducing), without extensive runtime and low level utilities.

Additionally, the implementation of Wind in C also ensures that no dynamic memory allocation is used. Only stack and data-segment (`static` declared in C/C++) memory are used during computation and runtime. The main buffers of Wind's data flow live in the data segment of the Wind executable. Many embedded systems have issues using heap allocation for memory, thus Wind tackles this issue efficiently.

The implementation of Wind is broken up into three main parts: *translation*, *execution* and *computation*. Wind processes source code very differently from other languages as it does not have a traditional pipe line. The language does not use abstract syntax trees, nested-syntax, byte code, or terminal grammar components. It is read as a series of commands and arguments, as explained in previous chapters. Therefore the phases between source code and resulting computation are quite unique.

Translation is the component in which source code is matched against syntax patterns to transition the internal state of Wind, and loads arguments or data to appropriate buffers. Execution is the phase where the command found from translation is invoked with the current data on the load buffer. Lastly, computation is the component of Wind that manipulates the specific binary data formats to evaluate expressions and binary operations.

This chapter will detail the C-level implementation of Wind and the concepts behind the design.

4.1 Translation

In most programming languages, the source code is first tokenized, then arranged into an abstract syntax tree. This process is commonly referred to as parsing. Parsing traditionally involves looking through characters in source code by a grammar which is ordered by priority. This is useful for languages with a nested structure. Such as functional languages normally require parsing many different elements, such as function parameters, function bodies, variables, names and more. All of which only occur under certain parent elements.

Wind has no nested structure or syntax. It is a truly *linear* language. This means instead of being concerned with the state of the syntactic structure, we are only concerned with the source code itself. For checking a direct *match* for a sequence of characters within an input string, naive comparison or a form of regular expression is used. In Wind however, both of these approaches would be far too slow.

Naive matching of a collection of substrings against a target string has a high complexity. Regular expressions can indeed be fast, yet normally require heavy duty utilities. Regular expressions are usually compiled as non-deterministic state machines prior to being able to match or search strings. This is tricky to due in a small scale environment. More so, the syntactic elements of Wind are not overly complex. The true power of regex is revealed in the matching and searching of more intricate structures.

Wind uses a technique called *static jump prefixing*, which uses nested jump tables to match prefixes of strings. These jump tables are created when the C-compiler detects a switch statement with 5 or more cases. Most modern C-compilers, including gcc, support this. Nested jump tables allow a syntactic element in source code to be translated to an instruction in $O(N_c)$, where N_c is the number of characters in the element of the source code. The *static* word in the title refers to the jump tables being formed at compile time, meaning they cannot be made or changed during the runtime of the executable. Dynamic jump tables are possible to implement, but would require the tedious task of compiling and running assembly code on the fly. Either way, the algorithm will work the same. To get a better understanding, a jump table is best described as a function, mapping some input to some output to a location within a set.

Theorem 1 (Jump Tables)**.** *Let $j(x)$ be a piecewise function who's domain is any non-negative integer x. Assume that j maps to some position y, where $j(x) \longrightarrow y$. Assume that y is either a valid position, or is the null position,* 0*.*

Next, let's say our function j has 2 possible positions, and the null position. The following is a valid piecewise representation:

$$\begin{cases} 0 & \longleftarrow x \neq 45 \wedge x \neq 64 \\ 1 & \longleftarrow x = 45 \\ 2 & \longleftarrow x = 64 \end{cases}$$

If we assume the input of $j(x)$ are 8-bit signed integers or characters, we can

see that such a function only gives a valid output if the character matches it's piece wise mapping. In a computational scenario, the zero case of the piecewise function corresponds to a `default` or error scenario of a jump table. We can compare the piecewise function with the following switch statement.

```
switch(x) {
      case '$': return 1;
      case '*': return 2;
      default:
            fprintf(stderr, "Syntax Error");
            return;
}
```

In the above switch statement, characters are used in place of integer literals to convey the idea of characters from the source code being used as an index to match against. However, so far we have only explored the use of a jump table to match a *length one* element. For multi length elements, such as keyword or command names, we have to use nested jump tables. One can approximate the idea of a nested jump table via nested piecewise functions.

The following is an example taken from the Wind source code to match the *clr* command:

```
1  int WindRun_command(const char** code)
2  {
3          while(**code)
4          {
5                  switch(**code)
6                  {
7                  case ' ':
8                  case '\n':
9                  case '\t':
10                 case '\v':
11                         *code += 1; //white space
12                         break;
13                 case WindRun_COMMENT_SYM:
14                         *code += 1;
15                         *code = _move_ptr_end_cmnt(*code);
16                         break;
17                 case 'c':
18                         switch((*code)[1])
19                         {
20                         case 'l':
21                                 switch((*code)[2])
22                                 {
23                                 case 'r':
24                                         // exec out
25                                         *code += 3;
26                                         WindState_set_cmd(
                                               WindCommand_clr);
27                                         goto TRANS_TO_LOAD;
28                                 default:
29                                         WindState_write_err("
                                               Expected command
                                               symbol, found 'cl%c'
                                               ", *code[2]);
30                                         return 0;
```

```
31                          }
32                          break;
33                  default:
34                          WindState_write_err("Expected
                                command symbol, found 'c%c'"
                                , *code[1]);
35                          return 0;
36                  }
37                  break;
```

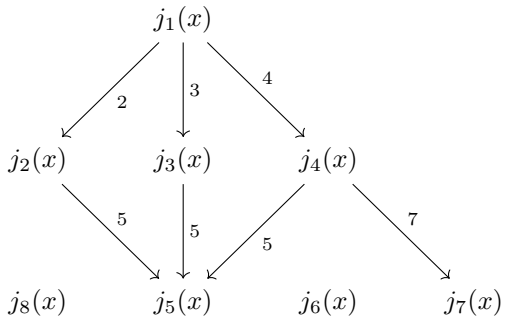
Starting from the top, let's break down the code. This function accepts a `const char*` to source code, which it moves as the translation finds more elements, piece by piece. The function's while loop will execute until the source code reaches the null terminating character. When the function encounters certain characters, it may call specialized translation functions to match specific elements, such as an integer or quote-bounded string.

This function ignores whitespace, but considers it a length one syntactic element of any white space character, and advances the source code by one.

If a character is encountered the matches 'c', then the look-ahead character is then checked against a nested switch statement. The main concept here is that, *both* switch statements are jump tables, and nested tables are only reachable through their outer tables. This allows advancement in state towards matching an element in $O(1)$ complexity, one for each character matched. If we want to model this mathematically, let's consider the following:

Theorem 2 (Jump Prefixing)**.** *Let $j(x) \longrightarrow T$ be a function with a domain of any non-negative integer, x. Let S_t be a set of functions. The range of $j(x)$ are the members of set S_t. Every function member of S_t subsequently has a range of more sets of functions. This continues until the range of a function results in a termination symbol.*

We can further explore this idea in the following diagram:



The above is just an example, yet illustrates the nested nature of such switch statements.

4.2 Execution and Computation

Once the Wind runtime has translated source code into relevant instructions, Wind proceeds to form it's state information and compute data. For Wind, there are three distinct phases of execution, each which change the context in which incoming code is used. Violations of these execution states will lead to errors being raised. Wind is well suited for running in a REPL-like fashion. It reads code, executes the code, and prints out a response or simply issues a computation to be made.

In this section we will discuss the different components that structure Wind's execution model. The computation buffer, and movement of data during computation will also be explained.

4.2.1 States

Wind operates in a *mode-like* manner by focusing it's internal state on what next action it should anticipate to do. The State of Wind and the functionality behind switching states is contained within the `WindState` module and related header file. Within the c file and it's header, you can find many functions related to changing the state, reading the state, and printing the state (for debugging or string functions). The actual data type that contains Wind's state is an enum:

```
typedef enum
{
        WindMode_command,
        WindMode_load,
        WindMode_exec
} WindMode;
```

The value of the current state is stored as this enum type in a `static` variable inside WindState.c. That state variable is meant to be accessible and consistent throughout the program, as it governs all other functionality in Wind's runtime.

Wind Modes

- *command*: Accepts source code input and scans for a command name. One a command is found, transitions to looking for arguments.
- *load*: This mode looks and scans for data values and arguments to load onto the load buffer. Once a $->$ arrow symbol is found, the mode transitions to exec.
- *exec*: This mode uses the computation buffer and related functions to compute the command name and the arguments on the load buffer. This mode also invokes the use of the inactive buffer for the trampoline effect.

When the *exec* mode is complete, the state simply transitions back to *command*

Error States

Wind has a special intermediary state to handle errors. It also uses a `static` boolean state to indicate if an error has been detected or not. During any of the three main modes, an error can be encountered. Normally in the command and load modes, syntax errors are most common. Errors encountered during the exec mode can be more intricate, such as using an invalid type for a specific command's argument.

The error state is accessed through the following functions:

```
void WindState_write_err(const char* fmt, ...);
int WindState_has_err(void);
void WindState_print_err(void);
```

Between every mode transition, Wind checks to see if the error state has been set to true. If it has, then it prints the error, sets the error state back to false, and terminates the runtime. Terminating the runtime does not mean Wind exits, it just means Wind will stop translating and executing the current string of source code and wait for new input.

The only time Wind exits upon an error is if there is no more available static memory in any of the buffers. To fix this, address the compiler flags in the makefile.

4.2.2 Computing Commands

Once Wind has translated source code and loaded appropriate arguments onto it's load buffer, it can commence executing the command. All command names in source code are mapped to an enum, similar to that of Wind's modes. Each mode is also equipped with it's own string representation. Part of the reason most state-type variables in Wind are implemented as enums are so they can be easily handled in `switch` statements.

When a commend is due for execution, it first starts at the following function, `WindRun_exec`

```
1  int WindRun_exec(const char** code)
2  {
3          switch(WindState_get_cmd())
4          {
5          case WindCommand_null:
6                  break;
7          case WindCommand_out:
8                  (void) WindExec_out();
9                  break;
10         case WindCommand_push:
11                 (void) WindExec_push();
12                 break;
13         case WindCommand_clr:
14                 WindExec_clr();
15                 break;
16         case WindCommand_map:
17                 WindExec_map();
18                 break;
```

```
19          case WindCommand_filter:
20                  WindExec_filter();
21                  break;
22          case WindCommand_reduce:
23                  WindReduce_reduce();
24                  break;
25          case WindCommand_save:
26                  WindExec_save();
27                  break;
28          case WindCommand_load:
29                  (void) WindExec_load();
30                  break;
31          }
32          WindData_load_reset(); // Resets load buf.
33          WindState_set_cmd(WindCommand_null);
34          WindState_set_mode(WindMode_command);
35          return 1;
36  }
```

From the above code, we can infer that the command enum type is dispatched to it's respective handling function. After that handling function completes, three calls take place, the same for any command that's executed. First, the load buffer is reset. This means the *current pointer* of the load buffer is set back to the beginning, permitting the entire size of the buffer set at compile time to be used for writing new data. Whenever this happens for a buffer, memory is never freed or allocated. The current pointer of a buffer acts as the "end" marker, allowing old data to be overwritten.

Next, the current command is set back to null, and the mode of Wind is set back to command. This allows Wind to restart the process, looking for a new command and arguments to translate.

Now, lets explore what happens to one of the specific handling functions for commands, such as the one fo the *map* command:

The Map command handler:

```
1   int WindExec_map(void)
2   {
3           if(!WindData_load_len()) return 1;
4           unsigned char* loadStart;
5           const unsigned char* loadStop;
6
7           unsigned char* activeStart = WindData_active_start();
8           const unsigned char* activeStop = WindData_active_ptr();
9
10          WindData_inactive_reset(); // resets inactive for
                writing new data.
11          while(activeStart != activeStop)
12          {
13                  loadStart = WindData_load_start();
14                  loadStop = WindData_load_ptr();
15                  // loads into the comp buf.
16                  activeStart += WindComp_write_typed(activeStart)
                        ;
17                  // map proceeds.
18                  if(!WindComp_map(loadStart, loadStop)) return 0;
```

```
19                  // If inactive buffer is full, this will exit
                        program.
20                  // Only happens in map if using size increasing
                        ops.
21                  WindData_inactive_write(WindComp_begin(),
                        WindComp_get_len());
22          }
23
24          WindData_active_switch();
25          return 1;
26  }
```

`WindData` as a prefix refers to the file and object file responsible for the having the functionality that controls Wind's buffers at the lowest level. While functions in *WindExec.c* do manipulate and trigger computation, the functions of *WindData* are actually doing the raw, binary work. First, let's walk through the individual steps of this function, and explain the logic:

Map Execution Routine:

1. Check if the load buffer is empty. Return if no arguments to map command.

2. Initialize pointers to the start of the data in the active buffer and the end of the data in active buffer.

3. Reset the inactive buffer. When data finishes it's flow transition from the active to computation to inactive buffer, it has all the capacity of the inactive buffer available.

4. Begin a loop over the active data. This intends for all the mapping arguments to be applied to every member of the active buffer once.