WaveScript Rev 3657 User Manual

November 12, 2008

Contents

Chapter 1

Introduction

WaveScript is a functional language for writing highly abstract programs that generate efficient dataflow graphs through a two-stage evaluation model. These graphs are executable in several backends (Scheme, ML, C, Java, TinyOS) that offer different tradeoffs in features, performance, and compile times, as well as parallel and distributed execution. The compiler has several command line entry-points for the different backends.

1.1 A taste of the language

The academic publications on WaveScript will give you a top-down account of its features, reason for being, and so on. In this manual, our goal is instead to introduce you to programming in WaveScript in a bottom-up manner. We'll start from "Hello-world" and proceed from there.

Here's a simple but complete WaveScript program:

```
main = timer(10.0)
```

This creates a timer that fires at 10hz. The *return value* of the `timer` function is a stream of empty-tuples (events carrying no information). The return value of the whole program is, by convention, the stream named "main". The *type* of the above program is `Stream ()`, where `()` designates the empty-tuple.

In our next program, we will print "Hello world" forever.

```
main = iterate x in timer(10.0) {
         emit "Hello world!";
       }
```

The `iterate` keyword provides a special syntax for accessing every element in a stream, running arbitrary code using that element's value, and producing a new stream as output. In the above example, our `iterate` ignores the values in the timer stream ("x"), and produces one string value on the output stream on each invocation (thus, it prints "Hello world!" at 10hz). The type of the program is `Stream String`.

Note, that we may produce two or more elements on the output stream during each invocation. For example, the following would produce two string elements for each input element on the timer stream.

```
main = iterate x in timer(10.0) {
         emit "Hello world!";
         emit "Hello again!";
       }
```

Actually, timers are the only built-in WaveScript stream *sources*. All other built-in stream procedures only transform existing streams.

Iterate also allows persistent, mutable state to be kept between invocations. This is introduced with the sub-keyword `state`. For example, the following produces an infinite stream of integers counting up from zero.

```
main = iterate x in timer(10.0) {
         state { cnt = 0; }
         emit cnt;
         cnt := cnt + 1;
       }
```

Notice that the assignment operator for mutating mutable variables (`:=`) is different than the operator used for declaring new variables (`=`). (WaveScript also has `+=`, `-=` etc for use in mutating variables.) It is also possible for mutable state to be declared *outside* the scope of the `iterate`, but that introduces the possibility of referring to the state from the bodies of two different iterates, which is not allowed.

As a final example, we'll merge two streams operating at different frequencies.

```
s1 = iterate x in timer(3.0) { emit 0; }
s2 = iterate x in timer(4.0) { emit 1; }
main = merge(s1,s2)
```

This will output a sequence of zeros and ones, with four ones for every three zeroes. The `merge` operator combines two streams of the same type, interleaving their elements in real time.

Before we get into the nitty gritty, I will leave you with two high-level points regarding how you should think of WaveScript.

1. **Everything is a value.** Streams and functions are just values (or objects if your prefer). The purpose of a WaveScript program is to return a Stream.

2. **But you need to understand multi-stage evaluation.** There are some restrictions on the code that executes inside `iterate`s, having to do with what features are supported at runtime vs. compile time. This detracts from the simplicity of (1), because, for example, function values may not be stored at run time. We'll return to this issue in Chapter 5.

1.2 Pragmatics: Using WaveScript

This section addresses pragmatic issues concerning invoking the system and running dataflow graphs. If you are new to WaveScript, it may help to skip to chapter 2 to get a sense for the language itself, then come back to this section afterwards.

Installing

See the README for install instructions. The bottom line is that you will need Mzscheme and Petite Chez Scheme to run the WaveScript compiler and the Scheme backend. (Petite for Linux is included in the repository.) You will need Boost, the Gnu Scientific Library (GSL), FFTW, and their respective headers to run compiled C++ code.

Once the system is installed you should be able to execute `ws` on any WaveScript source files (`.ws` files). This will compile the query and execute it directly in the Scheme backend. Other backends (and variations on this one) will be discussed in this section.

Develop Incrementally

WaveScript is a research compiler. It doesn't offer the same level of support that you expect from production-quality compilers. Sometimes the error messages might be difficult to interpret. (However, the type checker messages have been improved immensely and do include code locations [04/08/2007].)

Therefore it helps to build your program bit by bit. Compile often to make sure that the pieces of your program parse and type-check.

wsparse: While entrypoints such as **ws** parse source files internally, if you want the full error message for a parser error (including line and column number!), run `wsparse` directly on the `.ws` file. It will either print the AST (abstract syntax tree) for the file, or will give you an error message that does include line and character numbers.

1.2.1 ws: The Interactive Scheme Backend

Running the command "`ws mysource.ws`" will compile your source file, and execute the resultant dataflow graph directly within Scheme, without generating code for another platform. The dataflow graph is converted to a Scheme program which is compiled to native code and executed on the fly. Compile time is low, because we need not call a separate backend compiler. However, performance is not as good as the other backends we will discuss. Further, the Scheme backend does not support external (real time) data sources, only queries running in virtual time using data stored in files.

One major advantage of the Scheme backend is that it's *interactive*. The query starts up paused, and the user may step through and inspect the elements of the output stream. Thus the user may *pull* on the stream rather than being *pushed* a torrent of output.

Variants of ws

There are several other useful ways to invoke WaveScript with the Scheme backend.

Consider ws.opt for higher performance

The first thing to do to improve the performance of `ws` is make sure that you have the full version of Chez Scheme. This version includes the incremental native code compiler. If you are using the free, Petite Chez Scheme interpreter, your `.ws` query files will be converted to *interpreted* Scheme code, which runs considerably slower. The executable `chez` should be in your path.

To go further, you might consider running `ws.opt` in place of `ws`. First, `ws.opt` has debugging assertions disabled. Second, `ws.opt` is compiled in the highest Chez Scheme optimize level, and also compiles your query in this higher optimize level. However, this optimize level is dangerous in that it omits various checks necessary to maintaining type safety. Thus the process can crash if there's an error, or memory can become corrupted (just like C/C++). Therefore, debug your code using `ws` first.

Run ws.debug occasionally for sanity-checking

Ws.debug is too slow for the normal development cycle, but it is important to occasionally compile your program with ws.debug. Mainly this is important for building confidence that the compiler is behaving properly on your program. If the compiler were a mature system not under active development, this would not be as important.

Specifically, when you run `ws.debug`:

1. It turns on extra ASSERT statements in the code that check data structure invariants to help ensure proper operation.
2. It dumps the whole program at several points in the compilation to files such as "__elaboratedprog.ss". It can be helpful to look at these for debugging.
3. It includes additional type checks of the entire program between passes (regular `ws` type checks several times, but `ws.debug` does more). This helps expose compiler bugs.

4. Finally, it does "grammar checks" on the output of each compiler pass. Each pass includes a BNF grammar that describes its output. `ws.debug` mechanically checks to make sure every intermediate expression conforms to the appropriate grammar. Again, this helps to expose compiler bugs.

For very low compile times: ws.early

If you are running many small programs — stream "queries" — then the latency of compilation may be more important to you than the throughput of the query once it is running. In this case it is reasonable to use `ws.early`.

1.3 Premier Compiler Backends

1.3.1 The WaveScript MLton Backend

The WaveScript Scheme backend (ws) provides a reference implementation suitable to prototyping and debugging applications. The eventual target for most WaveScript programs is to generate an efficient stand-alone binary using one of our other backends. One of these backends is the MLton backend, invoked with **wsmlton**.

MLton is an aggressive, whole-program optimizing compiler for Standard ML. Generating ML code from the WaveScript dataflow graph is straightforward because of the similarities between the languages' type systems. This provides us with an easy to implement single-threaded solution that exhibits surprisingly good performance [**?**], while also ensuring type- and memory-safe execution.

1.3.2 The "Low-Level" C Backend

Invoked with "**wsc2**".

Moving forward, this will serve as the *primary* WaveScript backend. It has three purposes. First, it generates the most efficient code. Second, it serves as a platform for us to build custom garbage collectors (the C++ backend simply uses Boost smart pointers) and experiment with other aspects of the runtime. Third, it generates code with minimal dependencies (not requiring a runtime scheduler). This last point was critical in adapting the backend to target TinyOS (described in the next section).

Currently [2008.08.15], **wsc2** includes several garbage collection modes (simple reference counting, specialized deferred reference counting, boehm), has some support for multithreaded execution, some support for client/server execution over a LAN using ssh, and some support for client/server execution over a network of sensor nodes running JavaME or TinyOS and a base station. (See the following section.)

1.3.3 The NesC/TinyOS 2.0 Backend

Invoked with "**wstiny**".

This backend "inherits" from **wsc2** and modifies it to support TinyOS. As of this writing [2008.02.22], the TinyOS backend is limitted in its supported language features. (See the end of this section for a list of limitations.)

wstiny can be invoked in multiple ways. First, simply invoking **wstiny** on a WaveScript file (in the same way as any other backend) will compile the entire application to run on the mote.

```
[joe@computer] wstiny demo11g_tinyos.ws
```

In the same way that one invokes "`query.exe`" or "`query.mlton.exe`" when using another WaveScript backend, one can run the **wstiny** program with "`query.py`":

```
[joe@computer] ./query.py
```

This is a python script that invokes TOSSIM, the TinyOS simulator. The output of the program will be printed to the terminal—including printed output and return values on the main stream—again, just like any other backend.

Of course, the point of compiling for TinyOS is to run code on a *mote*! This is slightly more involved. After executing **wstiny**, we need to build the binary for the particular mote platform. And then we invoke a listener on the serial port to see the output of the program.

```
[joe@computer] make -f Makefile.tos2 telosb install
[joe@computer] MOTECOM=serial@/dev/ttyUSB0:telosb
[joe@computer] export MOTECOM
[joe@computer] java PrintfClient
```

wstiny has polluted your working directory with several files, including `Makefile.tos2`, `PrintfClient.java`, `query.py` and others. The PrintfClient program can listen to a particular device specified either through the `MOTECOM` environment variable, or through a "`-comm`" flag. Please refer to the the TinyOS documentation for instructions on how to connect motes to your computer.

Executing across mote *and* PC

Alas, it is not very interesting to simply run code on a mote that is attached to a PC's serial port. For a real sensor network application, we need to run WS code on an entire sensor network *and* on the PC connected to the network, seemlessly splitting the application across the two tiers.

Currently, **wstiny** supports a method for manually splitting programs across tiers. This is implemented using a simple naming convention. A "`Node`" namespace contains all the streams that will live inside the network, and everything outside of that namespace lives on the PC. The following program illustrates this method, but it may help to return to it after reading Section 2, and in particular Section 2.4 which explains namespaces.

```
namespace Node {
  s1 = ...
  s2 = ...
}
main = smap((+ 1), Node:s2);
```

Only the "(+ 1)" operation is executed on the PC. The "cut point" that splits `s2` between the mote and PC will cause the compiler to generate code on both sides that accomplishes the communication. To compile this program, and execute it across the PC and a single mote connected to a serial interface, we type the following:

```
[joe@computer] wstiny -split demo11g_tinyos.ws
[joe@computer] make -f Makefile.tos2 telosb install
[joe@computer] ./query.exe
```

First, we compile the program, producing code for both platforms. Second, we install the progam onto the mote. Third, we execute the PC portion of the program, which listens to the serial port, unpacks the messages from the stream `s2`, and executes the rest of the program, producing output to `stdout` as usual. If we had compiled *without* `-split`, then the entire application would run on the mote.

Note, the mote-side communication is accomplished using the TinyOS interface `AMSend`, which by default is wired to a `SerialAMSenderC` component. If we want to execute on more than one mote, we must instead wire this interface to the radio (the `AMSenderC` component). This is accomplished with:

```
[joe@computer] wstiny -split-radio demo11g...
```

Which is just shorthand for:

```
[joe@computer] wstiny -split-with "AMSenderC" ...
```

The resulting NessC code can be used to program several motes. Then the TinyOS-included "BaseStation" application can be used to enable a PC to receive these radio messages on its serial port. This method described in the here: Lesson 4: Mote-PC serial communication.

Note that the semantics of this method is that the messages from *every* mote's `s2` stream are combined to form the `s2` stream seen on the PC. Thus, the `s2` streams will most likely include the a node identifier so that the PC can demultiplex the stream. Please see Section 3.4 for information on integrating legacy TinyOS code with a WaveScript application.

Limitations

- The above `-split` methodology only works for one-hop networks. We must adapt it to use the CTP (Collection Tree Protocol) to work over larger networks.
- Code running on motes may not currently allocate memory dynamically.
- Only string constants, static arrays, numbers, and tuples are currently supported.
- Not all datatypes can be sent across a "cut" stream. Currently, tuples and numbers support marshaling (including nested tuples). Arrays support marshaling but *only* if the array is not contained within a tuple.

Future plans

We are in the process of removing the above limitations and adding support for new features:

- Dynamic allocation and garbage collection.
- Automatic and semi-automatic partitioning of stream programs across PCs and motes.

1.4 Deprecated Compiler Backends

1.4.1 The WaveScript CaML Backend

This backend is no longer supported and will receive no new features! But if you need to use it, let me know.

1.4.2 The C++/XStream Backend

This backend is no longer supported and will receive no new features! But if you need to use it, let me know.

The C++ backend[1] generates code that uses the XStream runtime/scheduling engine. It is invoked with the **wsc** command. In practice, as WaveScript undergoes development, the **wsc** compiler often lags behind **ws** in terms of features and functionality. Again, develop incrementally, refer to `demos/wavescope` for programs that should work with **wsc**.

Refer to the XStream documentation[2] for information on how to configure the XStream engine (number of threads, scheduling policy, etc)).

[2007.03.09] Ideally the output from the XStream-executed query would be the same as the ws's output. Currently, however, there are some basic disparities in how objects are printed to text form. Hopefully, these are straightforward to work around, and should go away.

Note that the C++/XStream backend does not have competitive performance with either the MLton backend or the newer ANSI C backend (**wsc2**).

[1] It uses a C++ compiler not because it generates object-oriented code, but because the XStream runtime engine it links with has been engineered in C++.

[2] [2007.03.09] Currently nonexistent

Chapter 2

Language Basics

Please make liberal use of the demo files located at `demos/wavescope/*.ws` as a reference when learning WaveScript. These demos cover most of the basic functionality in the language.

Wavescript is a functional language with many similarities to ML or Haskell. Functions are values. The language is strongly typed and uses a type-inference algorithm to infer types from variable usages. It provides the usual basic datatypes, including various numeric types, strings, lists, arrays, hash tables as well as the more WaveScript-specific Streams and Sigsegs. Valid *expressions* in WaveScript are written much as in C.

arithmetic	:	`3 + 4 * 10`
function calls	:	`f(x, y)`
blocks	:	{ e_1; e_2; ... e_n }

But there are also many syntactic differences from C. For example, WaveScript doesn't have a sharp division between commands and expressions. Conditionals use a different syntax and are valid in *any* expression position.

```
3 + (if true then 1 else 2)    →   4
```

(Note that the arrow above means "evaluates to", but WaveScript does not currently have an interactive read-eval-print-loop in which you can type such incomplete program fragments.)

Moreover, *blocks* in WaveScript—delimited by curly braces—are just expressions!

```
3 + {1; 2; 3}    →   6
```

This is similar to the `begin/end` blocks found in other functional languages, or the "comma operator" sequencing construct in C++. Only the value of the last expression within the block is returned, the other statements/expressions only bind variables or perform side effects.

Note that this different convention makes semi-colon usage in WaveScript somewhat unintuitive. Within blocks, semi-colons are only required as *separators*; they are permitted, but not required after the last expression in the block. Also, since curly-brace delimited blocks are merely expressions, they sometimes must be followed with a semi-colon, as below:[1]

```
{
  foo;
  if b then {
    bar();
  } else {
    baz();
  };  <-- SEMICOLON REQUIRED
  done();
}
```

[2007.03.09] Be wary that with the current parser, a semi-colon error may appear as a strange parse error in the next or previous line.

2.0.1 Streams

WaveScript is for stream-processing and it would be useless without Streams. Streams in WaveScript are like any other values: bound to variables, passed and returned from functions. The primary means of operating on the data inside streams is the `iterate` syntactic construct.

```
S2 = iterate x in S1  {
  state{ counter = 0 }
  counter := counter + 1;
  emit counter + x;
}
```

The `iterate` construct can be placed in any *expression position*. It produces a new stream by applying the supplied code to every element of its input stream. The above example executes the body of the iterate every time data is received on the stream `S1`, it adds an increasing quantity to each element of `S1`, and the resulting stream is bound to the variable `S2`.

In addition to `iterate` several library procedures (such as `stream_map`) and many primitives (such as `unionList`) operate on `Stream` values. For example, `merge` combines

[1] As a special case, when a function body consists of curly braces, it needn't be followed by a semi-colon.

two streams of the same type (their tuples interleaved in real time), whereas `unionList` also combines streams of the same type, but tags all output tuples with an integer index indicating from which of the input streams it originated.

2.0.2 Defining functions

Named functions can be defined at the top-level, or within a {...} block, as follows:

```
fun f(x,y) {
  z = x+y;
  sqrt(z)
}
```

The function body, following the argument list can be any expression. Above it happens to be a statement block that returns the square-root of `z`.

Unnamed, or anonymous, functions can occur anywhere that an expression might occur. In the following we pass an anonymous "plus three" function as the first input to `stream_map`. Note that the body of the function is a single expression, not a statement block, and is not delimited by curly braces.

```
stream_map(fun(x) x+3, S)
```

2.0.3 Constructing complete programs

A complete program file contains function and variable declarations. As seen in Section 1.1, a special variable declaration binds the name "main" to a stream:

```
main = stream-valued-expression;
```

As a side-note, WaveScript can be called with a "-ret foo" flag to return the stream "foo" instead of the stream "main". This is frequently useful for invoking testing entrypoints as well as the production entrypoint for the program.

Note also that *only* the returned stream is instantiated at runtime. Other streams declared in the program will simply be ignored. They become dead code.

2.0.4 Tips for C/Java Programmers

As a quick cheat sheet, refer to Figure 2.1. This addresses some minor syntactic differences between WS and C/C++ that commonly trip people up.

One important piece of advice is to be careful with mutation (`:=`). Don't use it when it's unnecessary! And if you need to use it, keep the scope of the variables in question as small as possible. If, for example, you introduce mutable top-level (global) variables it may have major deleterious effects. First of all, it opens up the possibility that you will refer to the state from multiple `iterate` bodies, which will make the compiler yell at you. More subtly, it can disable optimizations that act on `iterate`s without persistent state—for example, replicating them in parallel.

C/C++ code	*WS Equivalent*
`int x = 3;`	`x = 3;` ***OR*** `x = (3::Int);` `x::Int = 3;`
`x = 4;`	`x := 4;`
`if (a) b; else c;`	`if a then b else c;`
`typedef char myty;`	`type myty = Char;`
`List< Array<int> >`	`List (Array Int)`

Figure 2.1: Some highlighted syntactic differences between WaveScript and C/C++/Java.

2.1 Datatypes

This section goes over syntax and primitives for manipulating built-in data structures.

2.1.1 Tuples

Tuples, or *product types*, are ordered sets of elements which can each be of different types.

`(1,''hello'')` → A tuple of type `(Int * String)`

Tuples are accessed by a pattern-matching against them and binding variable names to their constituent components. This process is described in section 2.5.

2.1.2 Numbers

WaveScript supports several different types of numbers. Currently, this set includes Int, Int16, Uint16, Int64, Float, Double, Complex, but it will be extended to include 8-bit integers, other unsigned integers, and complex-doubles. WaveScript includes generic numeric arithmetic operations ($+$, $-$, $*$, $/$) that work for any numeric types, but must be used on two numbers of the same type. There are also type-specific numeric operations that are not used frequently:

generic	:	`+ - * / ^`
Int	:	`+_ -_ *_ /_ ^_`
Float	:	`+. -. *. /. ^.`
Complex	:	`+: -: *: /: ^:`
*Int*16	:	`+I16 -I16 *I16 /I16 ^I16`
*Int*64	:	`+I64 -I64 *I64 /I64 ^I64`

Other numeric operations, such as `abs` or `sqrt`, follow the naming convention `absI` for integers, `absF` for floats, `absC` for complex, and `absI16` for 16-bit integers. Eventually, WaveScript will include a type-class facility which will simplify the treatment of numeric operations.

2.1.3 Lists

Lists can be written as constants and support the usual primitive operations.

```
ls = [1, 2, 3];
ls2 = head(ls):::ls;
print(List:length(ls2)); // Prints '4'
print(ls == []);         // Prints 'false'
```

The `:::` operator adds an element to the front of a list. Also use `head`, `tail`, `List:reverse`, `List:append` to operate on lists.

Other operations include, but are not limited to the following. `List:zip` combines two equal-length lists into a list of two-tuples. `List:map(f,ls)` returns a new list generated by applying the function `f` to each element of the input list. `List:filter(pred,ls)` returns only those elements of the list that satisfy `pred`. `List:fold(op,init,ls)` reduces a list by repeatedly applying `op` to pairs of elements, for example, to sum the elements of a list. `List:mapi(f,ls)` is a variant of map that also passes the index of the element to the input function. `List:foreach(f,ls)` applies a function to each element for side effect only, not building a new list. `List:build(len, f)` builds a new list, using a function (on index) to populate each position in the list.

Many of these operators (map, filter, foreach, mapi, fold, build, etc) are used consistently for different container types (lists, arrays, matrices, etc).

2.1.4 Arrays

Arrays, unlike lists, are mutable. Use a hash symbol to build a constant array rather than a list, for example `#[1,2,3]`. Use *Array:make* to allocate new arrays. The *Array:ref* and *Array:set* primitive access arrays elements, but it's shorter to use the syntactic sugar, `arr[i]` to access arrays, and `arr[i] := x;` to modify them. Nested array references work as expected, as do the `+=` style shorthands, e.g. `arr[i][j] += 3`. However, to use these shorthands with an arbitrary, non-variable expression extra parentheses are required: `(f(x))[i]`.

Many array operations are analogous to the list operations (map, fold, etc). See the documentation for additional primitives below.

2.1.5 Extensible Records

Tuples are useful for packing together a small number of data items. But from a software engineering perspective they have a number of limitations. First, they quickly grow confusing as the number of items increases. Second, they require that client code be fully aware of all the data inside the tuple (and thus be fragile and difficult to refactor).

A solution to the first problem is to use named fields. Tuples with named fields are called "records" (similar to C structs). A solution to the second problem is to make those records *extensible*. WaveScript implements the extensible record system documented in the paper "Extensible Records with Scoped Labels" by Dan Leijen.

In WaveScript, records are constructed in a manner similar to tuples, but with the labels on each data field. Data fields can then be extracted using then common "dot syntax".

```
r = (A=3, B=true)  // Record construction
x = r.A            // Extract the field 'A'
```

But records can also be extended with new fields. This uses a "bar" syntax as follows:

```
( r | A=3)        // Extend a record r
(A=3) == (|A=3)   // The bar is optional here
```

When updating records, it is also possible to remove fields:

```
( r | ~A, B=4)    // Remove field A, add field B
```

Removing a field and then replacing it is a common idiom, and has a special syntactic sugar:

```
( r | ~B, B = 4) // Remove field B, add new B
( r | B := 4)    // Shorthand for the same
```

Field labels are required to begin with capital letters in WaveScript. This is to avoid conflict with another use of the dot syntax.

Extensible records are particularly useful in a streaming language like WaveScript. In WaveScript, the only communication between different parts of the program is through streams. Thus we typically need to pack data together to travel on a stream (for example, raw data plus metadata). With extensible records, it is possible to write an application as a series of modular components that take and produce records—consuming some fields, and adding others, while remaining agnostic to the presence of unused fields:

```
// Function foo takes a record that has at
// least X,Y. (But might have other fields.)
fun foo(r) {
  n = f(r.X, r.Y);
  ( r | Z=n )      // add new results field
}
```

This form of processing is very common in stream SQL dialects, but here can be used in a more general purpose language with type-inference.

2.1.6 Additional primitive functions

Appendix A contains a table of all currently supported WaveScript primitives, together with their type signatures.

For more documentation on these primitives, please refer to this file within your working copy:

`src/generic/`
`compiler_components/prim_defs.ss`

You can find the online documentation at:

http://regiment.us/codedoc/html/generic/compiler_components/prim_defs.ss.html

This file contains type-signatures (and minimal documentation) for all built-in wavescript primitives, many of which are not covered in this manual. Within the online documentation linked above you should look at regiment-primitives which is defined in terms of regiment-basic-primitives, regiment-distributed-primitives, wavescript-primitives, meta-only-primitives, higher-order-primitives, and regiment-constants .

Please also examine the library files found in the `lib/` sub-directory. These files, for example ``stdlib.ws'' and ``matrix.ws'', include library routines written in WaveScript. As a general design principle, it is best to implement as much as possible in the language, while keeping the set of built-in primitives relatively small.

For various historical reasons there are several primitives included in the current primitive table that *should not* ultimately be primitive. These will eventually be removed and implemented instead as library routines. Likewise, there are certainly many additional primitives that one might like to see incorporated as the language matures.

2.2 Type Annotations

We have already seen several types in textual form within this manual: `Float`, `(Int * String)` These are valid WaveScript types. WaveScript also has compound types such as `Sigseg (Array Int)`.

Similar to Haskell or ML, the user is permitted, but not generally required to add explicit type annotations in the program. For example, we may annotate a function `f` with its type by writing the following.

```
f :: (Type, Type) -> Type;
fun f(x,y) { ... }
```

Indeed, a type annotation may be attached to any expression with the following syntax:

```
(expression ::  Type)
```

Further, type annotations may be added to variable declarations with:

```
var ::  Type = expression;
```

In general, compound types may be built from other types in several ways.

tuples	:	`(T1 * T2 * T3)`
lists	:	`List T1`
arrays	:	`Array T1`
hashtables	:	`HashTable (T1,T2)`
functions	:	`(T1, T2, T3) -> T4`

Note that parentheses must be used when nesting type constructors as in `List (List T)`.

[2007.03.09] In the future, the user will be able to create their own type definitions and type constructors, including tagged-union or *sum* types.

2.2.1 Reading data with readFile

The one place where type annotations *are* currently mandated is when importing data from a file. This is done with the primitive `readFile`. Readfile is not a stream *source*, but rather is driven by another stream, reading a tuple from the file for every element on its input stream. This makes it more general.

```
main =
  (readFile("foo.txt", "", timer(10.0))
   :: Stream (Int16 * Float))
```

The above complete program reads space-separated values from a text file, one-tuple-per-line. In this case, each tuple contains two values: a 16 bit integer, and a floating point value. The `readFile` primitive may also be used to read data from a file in blocks (Sigsegs), which is generally more efficient. All that is required is to specify a stream of `Sigseg` type, as follows.

```
main =
  (readFile("foo.dat",
            "mode: binary  repeat: 3",
            timer(10.0))
   :: Stream (Sigseg Int16))
```

Also note that the second argument to `readFile` is an option string. This string is parsed at compile time (during meta-program evaluation). The string must contain a (space separated) list of alternating option names and option values. The following are the available options, and their default values.

- **mode:** one of 'text' or 'binary' (default **'text'**)
- **repeats:** a number specifying whether to replay the file's contents when the end of file is reached. Set to a non-negative integer to specify the number of repeats, or to **-1** to repeat indefinitely. (default **0**)
- **rate:** the rate (in tuples per second) to play the back the data from the file. For the emulator, this refers to virtual time, and is used only to maintain relative timing of different data streams. Note that this is orthogonal to windowing; whether data is windowed or not, the rate will be interpreted in the same way. (default **1000**)
- **offset:** The offset, in bytes, at which to start reading from the file. (default **0**)
- **skipbytes:** The number of bytes to skip between reading each tuple from the data file. (default **0**)

- **window:** If the output stream is blocked into Sigsegs (windows), this parameter determines the size of each Sigseg. (default **1**)

2.2.2 Type Aliases

Because types can grow large and complex, it is often helpful to define aliases, or shorthands, similar to C/C++'s `typedef`s.

```
type MyType        = List Int;
type MyType2  t    = Stream (List t);
type MyType3 (x)   = List (x);
type MyType4 (x,y) = List (x * y);

x :: MyType;
x = [3];

s1 :: MyType2 Int;
...
```

2.3 Sigsegs

Sigsegs are a flexible ADT for representing windows of samples on a stream. Please refer to the CIDR'07 publication with the title "The Case for a Signal-Oriented Data Stream Management System" for details. Also check `prim_defs.ss` for the specific names and type signatures of the Sigseg primitives.

2.4 Namespaces

WaveScript, while not having a sophisticated module system, does include a simple system for managing namespaces.

```
namespace Foo {
  x = ...;
  y = ...;
}
var = Foo:x + Foo:y;
fun f() {
  using Foo;
  var = x + y;
}
```

2.5 Patterns

WaveScript allows pattern matching in addition to plain variable- In any variable-binding position it is valid to use a pattern rather than a variable name—this includes the arguments to a function, a local variable binding, or the variable binding within an `iterate` construct. Currently, patterns are used to bind names to the interior parts of tuples. In the future, we will support list patterns, and tagged union patterns.

Let's look at an example. We saw how to bind variables in WaveScript:

```
z = (1,2);
```

This binds `z` to a tuple containing two elements. This is actually a shorthand for the more verbose syntax:

```
let z = (1,2);
```

An unfortunate limitation of the parser is that `'let'` cannot be omitted if we a pattern is used in place of a simple identifier. The following binds the individual components of the tuple by using a pattern in place of the variable `'z'`:

```
let (x,y) = (1,2);
```

Similarly, we may use patterns within a function's argument list. Here's a function that takes a 2-tuple as its second argument:

```
fun foo (x,(y,z)) { ... }
```

2.6 Syntactic Sugar

Syntactic sugars are convenient shorthands that are implemented by the parser and make programming in WaveScript more convenient (at the risk of making reading code more difficult for the uninitiated).

2.6.1 Dot syntax

For convenience, functions can be applied using an alternative "dot syntax". For example rather than taking the first element of a list with "`head(ls)`", we can write "`ls.head`". This generalizes to functions of more than one argument; only the first argument is moved before the dot. For example,

```
List:ref(ls,i)
```

may be written as

```
ls.List:ref(i)
```

This is useful because many functions on data structures take the data structure itself as their first argument. Thus it is concise to write the following:

```
ls.tail.tail.head
```

Note that this syntax requires that the name of the function (not counting namespace modifiers) begin with a lower case letter. This is to avoid ambiguity with record field projection (see Section 2.1.5).

2.6.2 $: "Unary parentheses"

The "$" operator for procedure application is taken from Haskell and sometimes called the "unary parenthesis". Instead of `f(g(x))`, we write "`f $ g $ x`". This is useful if you have a large expression spanning many lines to which you want to apply a function:

```
iterate x in strm {
   (many lines) ...
```

We can apply a function `myfunction` without scrolling down to insert a close parenthesis:

```
myfunction $
iterate x in strm {
   (many lines) ...
```

2.6.3 Stream Projectors

WaveScript also includes a syntax for binding streams of tuples in a way that associates a projector function for each of the tuples' fields. For example:

```
S as (a,b) = someStream;
```

Subsequently, "`S.(a)`" or "`S.(a,b,a)`" can be used to project a new stream where each tuple represents an arrangement of the fields within each tuple in S. If used in conjunction with the type-annotation syntax, note that the "`as`" clause must go first:

```
S as (a,b) ::  Stream(Int * Float) = someStream;
```

Chapter 3

Foreign (C/C++) Interface

The WaveScript compiler provides a facility for calling external (foreign) functions written in C or C++. The primary reasons for this are two-fold.

1. We wish to reuse existing libraries without modification (e.g. Gnu Scientific Library, FFTW, etc).
2. We wish to add new stream sources and sinks — for network communication, disk access and so on — without modifying the WaveScript compiler itself. Also, we frequently want to add support for new hardware datasources (sensors).

There are three WaveScript primitives used to interface with foreign code. The `foreign` primitive registers a single C function with WaveScript. Alternatively, `foreign_source` imports a stream of values from foreign code. It does this by providing a C function that can be called to add a single tuple to the stream. Thus we can call from WaveScript into C and from C into WaveScript. The third primitive is `inline_C`. It allows WaveScript to generate arbitrary C code at compile time which is then linked into the final stream query. We can of course call into the C code we generate from WaveScript (or it can call into us).

3.1 Foreign functions

The basic foreign function primitive is called as follows: "`foreign(`*`function-name, file-list`*`)`". Like any other primitive function, `foreign` can be used anywhere within a WaveScript program. It returns a WaveScript function representing the corresponding C function of the given name. The only restriction is that any call to the `foreign` primitive *must* have a type annotation. The type annotation lets WaveScript type-check the program, and tells the WaveScript compiler how to convert (if necessary) WaveScript values into C-values when the foreign function is called.

The second argument is a list of *dependencies*—files that must be compiled/linked into the query for the foreign function to be available. For example, the following would import a function "foo" from "foo.c".

```
c_foo ::  Int -> Int = foreign("foo", ["foo.c"])
```

Currently C-code can be loaded from source files (`.c`, `.cpp`) or object files (`.o`, `.a`, `.so`). When loading from object files, it's necessary to also include a header (`.h`, `.hpp`). For example:

```
c_bar =
   (foreign("bar", ["bar.h", "bar.a"])
    :: Int -> Int)
```

Of course, you may want to import many functions from the same file or library. WaveScript uses a very simple rule. If a file has already been imported once, repeated imports are suppressed. (This goes for source and object files.) Also, if you try to import multiple files with the same basename (e.g. "bar.o" and "bar.so") the behavior is currently undefined.

3.2 Foreign Sources

A call to register a foreign source has the same form as for a foreign function: "`foreign_source(`*`function-name, file-list`*`)`". However, in this case the *function-name* is the name of the function being *exported*. The call to `foreign_source` will return a stream of incoming values. It must be annotated with a type of the form `Stream` T, where T is a type that supports marshaling from C code.

We call the function exported to C an *entrypoint*. When called from C, it will take a single argument, convert it to the WaveScript representation, and fire off a tuple as one element of the input stream. The return behavior of this entrypoint is determined by the scheduling policy employed by that particular WaveScope backend. For example, it may follow the tuple through a depth-first traversal of the stream graph, returning only when there is no further processing. Or the entrypoint may return immediately, merely enqueuing the tuple for later processing. The entrypoint returns an integer error code, which is zero if the WaveScope process is in a healthy state at the time the call completes. Note that a zero return-code does not guarantee that an error will not be encountered in the time between the call completion and the next invocation of the entrypoint.

Currently, using multiple foreign sources is supported (i.e. multiple entrypoints into WaveScript). However, if

using foreign sources, you cannot also use built-in WaveScript "timer" sources. When driving the system from foreign sources, the entire WaveScript system becomes just a set of functions that can be called from C. The system is dormant until one of these entrypoints is called.

Because the main thread of control belongs to the foreign C code, there is another convention that must be followed. The user must implement *three* functions that WaveScript uses to initialize, start up the system, and handle errors respectively.

```
void wsinit(int argc, char** argv)
void wsmain(int argc, char** argv)
void wserror(const char*)
```

`Wsinit` is called at startup, before any WaveScript code runs (e.g. before `state{}` blocks are initialized, and even before constants are allocated). `Wsmain` is called when the WaveScript dataflow graph is finished initialing and is ready to receive data. `Wsmain` should control all subsequent acquisition of data, and feed data into WaveScript through the registered `foreign_source` functions. `Wserror` is called when WaveScope reaches an error. This function may choose to end the process, or may return control to the WaveScope process. The WaveScope process is thereafter "broken"; any pending or future calls to entrypoints will return a non-zero error code.

3.3 Inline C Code

The function for generating and including C code in the compiler's output is `inline_C`. We want this bso that we can *generate* new/parameterized C code (by pasting strings together) rather than including a static `.c` or `.h` file, and instead of using some other mechanism (such as the C preprocessor) to generate the C code. The function is called as "`inline_C(`*`c-code, init-function`*`)`". Both of its arguments are strings. The first string contains raw C-code (top level declarations). The second argument is either the null string, or is the name of an initialization function to add to the list of initializers called before `wsmain` is called (if present). This method enables us to generate, for example, an arbitrary number of C-functions dependent on an arbitrary number of pieces of global state. Accordingly we also generate initializer functions for the global state, and register them to be called at startup-time.

The return value of the `inline_C` function is a bit odd. It returns an empty stream (a stream that never fires). This stream may be of any type; just as the empty list may serve as a list of any type. This convention is an artifact of the WaveScript metaprogram evaluation. The end result of metaprogram evaluation is a dataflow graph. For the inline C code to be included in the final output of the compiler, it must be included in this dataflow graph. Thus `inline_C` returns a "stream", that must in turn be included in the dataflow graph for the inline C code to be included. You can do this by using the `merge` primitive to combine it with any other Stream (this will not affect that other stream, as `inline_C` never produces any tuples). Alternatively, you can return the output of `inline_C` directly to the "main" stream, as follows:

```
main = inline_C(...)
```

3.4 Foreign code in TinyOS

There are a few differences in how foreign code works in TinyOS. The `foreign_source` function is virtually the same. The `foreign` function is similar, but it should be noted that one can cheat a little by suppying an arbitrary string for the function name. For example, here is a foreign function that toggles an LED, written using NesC's `call` syntax:

```
led0Toggle =
  (foreign("call Leds.led0Toggle", [])
   :: () -> ());
```

Currently [2008.02.22] foreign only works for functions returning `void`. Bug me to fix this.

The major difference lies in `inline_C`, which is replaced by `inline_TOS`. TinyOS simply has more places that one might want to stick code, thus more hooks are exposed:

```
inline_TOS(top, conf1, conf2, mod1, mod2, boot)
```

All arguments are strings. They inject code into different contexts as follows:

1. **top**: Inject code at top-level, not iside a *configuration* or *module* block.
2. **conf1**: Inject code into the `configuration` block produced by the WaveScript compiler.
3. **conf2**: Inject code into the `implementation` section of that `configuration` block.
4. **mod1**: Inject code into the `module` block produced by the WaveScript compiler.
5. **mod2**: Inject code into the `implementation` section of that `module` block.
6. **boot**: Inject code into the `Boot.booted()` event handler.

This mechanism for inlining NesC code can be used for adding support for new timers or data sources. In fact, this is how existing functions like `tos_timer` and `sensor_uint16` are implemented. (See `internal_wstiny.ws` inside the `lib/` directory.)

feature	*ws*	*wsmlton*	*wsc*
`foreign`	yes	yes	yes
`foreign_source`	never	yes	not yet
`inline_C`	not yet	yes	not yet
loads `.c`	yes	yes	yes
loads `.h`	yes	yes	yes
loads `.o`	yes	not yet	yes
loads `.a`	yes	not yet	yes
loads `.so`	yes	no	yes
marshal scalars	yes	yes	yes
marshal arrays	no	yes	not yet
`ptrToArray`	no	yes	not yet
`exclusivePtr`	yes	not yet	yes

Figure 3.1: Feature matrix for foreign interface in different backends

3.5 Other backend's support

The foreign interface works to varying degrees under each backend. Below we discuss the current limitations in each backend. The feature matrix in Figure 3.1 gives an overview.

This has not been updated to address **wsc2**.

Note that even though the Scheme backend is listed as supporting `.a` and `.o` files, the semantics are slightly different than for the C and MLton backends. The Scheme system can only load shared-object files, thus when passed `.o` or `.a` files, it simply invokes a shell command to convert them to shared-object files before loading them.

Including source files also has a slightly different meaning between the Scheme and the other backends. Scheme will ignore header files (it doesn't need them). Then C source files (`.c` or `.cpp`) are compiled by invoking the system's C compiler. On the other hand, in the XStream backend, C source files are simply `#included` into the output C++ query file. In the former case, the source is compiled with no special link options or compiler flags, and in the latter it is compiled under the same environment as the C++ query file itself.

Thus the C source code imported in this way must be fairly robust to the `gcc` configuration that it is called with. If the imported code requires any customization of the build environment whatsoever, it is recommended to compile them yourself and import the object files into WaveScript, rather than importing the source files.

[2007.05.03] Note: Currently the foreign function interface is only supported on Linux platforms. It also has very preliminary support for Mac OS but has a long way to go.

WaveScript	*C*	*explanation*
`Int`	`int`	native ints have a system-dependent length, note that in the Scheme backend WaveScript `Int`s may have less precision than C `int`s
`Float`	`float`	WaveScript floats are single-precision
`Double`	`double`	
`Bool`	`int`	
`String`	`char*`	pointer to null-terminated string
`Char`	`char`	
`Array T`	*T*`*`	pointer to C-style array of elements of type `T`, where `T` must be a scalar type
`Pointer`	`void*`	Type for handling C-pointers. Only good for passing back to C.

Figure 3.2: Mapping between WaveScript and C types. Conversions performed automatically.

3.6 Converting WaveScript and C types

An important feature of the foreign interface is that it defines a set of mappings between WaveScript types and native C types. The compiler then automatically converts, where necessary, the representation of arguments to foreign functions. This allows many C functions to be used without modification, or "wrappers". Figure 3.2 shows the mapping between C types and WaveScript types.

[2007.08.24] Currently wsmlton does not automatically null terminate strings. This needs to be fixed, but in the meantime the user must null terminate them manually.

[2007.05.03] The system will very soon support conversion of Complex and Int16 types. types.

3.7 Importing C-allocated Arrays

A WaveScript array is generally a bit more involved than a C-style array. Namely, it includes a length field, and potentially other metadata. In some backends (`wsc`, `wsmlton`) it is easy to pass WaveScript arrays to C without copying them, because the WS array contains a C-style array within it, and that pointer may be passed directly.

Going the other way is more difficult. If an array has been allocated (via `malloc`) in C, it's not possible to use it directly in WaveScript. It lacks the necessary metadata and lives outside the space managed by the garbage collector. However, WaveScript does offer a way to *unpack* a pointer to C array into a WaveScript array. Simple use the primitive `''ptrToArray''`. But, as with foreign functions, be sure to include a type annotation. (See the table in Figure 3.1 for a list of backends that currently support `ptrToArray`.)

3.8 "Exclusive" Pointers

Unfortunately, `ptrToArray` is not always sufficient for our purposes. When wrapping an external library for use in WaveScript, it is desirable to use memory allocated outside WaveScript, while maintaining a WaveScript-like API. For instance, consider a Matrix library based on the Gnu Scientific Library (GSL), as will described in the next chapter. GSL matrices must be allocated outside of WaveScript. Yet we wish to provide a wrapper to the GSL matrix operations that feels natural within WaveScript. In particular, the user should not need to manually deallocate the storage used for matrices.

For this purpose, WaveScript supports the concept of an *exclusive* pointer. "Exclusive" means that no code outside of WaveScript holds onto the pointer. Thus when WaveScript is done with the pointer the garbage collector may invoke `free` to deallocate the referenced memory. (This is equivalent to calling `free` from C, and will not, for example, successfully deallocate a pointer to a pointer.)

Using exclusive pointers is easy. There is one function `exclusivePtr` that converts a normal `Pointer` type (machine address) into a managed exclusive pointer. By calling this, the user guarantees that that copy of the pointer is the only in existence. Converting to an exclusive pointer should be thought of as "destroying" the pointer—it cannot be used afterwards. To retrieve a normal pointer from the exclusive pointer, use the `getPtr` function. In the future, getting an exclusive pointer will "lock" it, and you'll have to release it to make it viable for garbage collection again. Currently, this mechanism is unimplemented, so you must be careful that you use the `Pointer` value that you get back immediately, and that there is a live copy of the exclusive pointer in existence to keep it from being free'd before you finish with it.

Chapter 4

Standard Libraries

4.1 stdlib.ws

Most of the basic stream operators (window, rewindow, sync, etc) are contained within "stdlib.ws". Thus most programs will start with `include ''stdlib.ws''`. The file is contained within the `lib/` subdirectory of the WaveScript root. The top portion of the file contains type signatures for all the functions exported by the library. Further down, above each definition, should be a brief explanation of its function.

Here is a high level overview of the functionality provided by `stdlib.ws` as of [2007.08.24]:

- Stream operators: snoop on streams, zip streams together, sync them, window, rewindow, and dewindow them. Interleave and deinterleave the elements of streams.
- Sigseg operators: map functions over them, lift some operations like fft to work on sigsegs.
- Basic math functions that are not included as primitive.
- Prints, asserts, and constant definitions (e.g. π).
- Extended list and array operations not included as primitive (e.g. `List:map2`).
- Shorthands for common procedures (e.g. `i2f` for `intToFloat`).
- Curried versions of higher order operators.

4.2 Fourier Transforms

WaveScript uses fftw. See prim_defs.ss for a list of the different fft interfaces provided.

4.3 Matrix Operations

WaveScript uses the Gnu Scientific Library to support matrix operations. There are three files of interest within the `wavescript/lib/` directory. The first, `gsl.ws` declares prototypes for accessing a subset of the low level GSL functions directly. You shouldn't need to use this file directly. The second, `matrix_gsl.ws`, which you should use, provides a wrapper around GSL's matrix functionality that's more in the spirit of WaveScript. (Note that both of these files are generated by the C-preprocessor from the corresponding `.pp` files.)

The third file of interest is `matrix.ws`. This is a native-WaveScript library that implements the basic matrix operations described below using a simple array-of-arrays representation. Except for a few operations implemented only in the GSL-based version, the libraries should be interchangeable. You should choose which to use based on the operations you need, availability of GSL on the target platform, and performance requirements.

4.3.1 Other numeric types

All of the below matrix operations are explained with the example of 32-bit floating point matrices. These operators are contained in the namespace `Matrix:Float` and can may be referred to with fully qualified names (`Matrix:Float:add`) or by first importing the namespace, "`using Matrix:Float;`", followed by only the short name "`get`". Also included in the matrix library are analogous routines in the `Matrix:Complex` and `Matrix:Double` namespaces. [1]

4.3.2 Matrix Interface

Here are the functions contained within the namespace `Matrix:Float`. On the left is a typical function call, with meaningful names for the arguments, and on the right is the WaveScript type of the function. For brevity "`M`" is used to abbreviate the type `Matrix Float`, which in turn is a type alias for whatever the internal representation of a matrix is.

[1] (Currently, the `invert` operation is available only for `Matrix:Double`.)

- create($dim1, dim2$) :: (Int, Int) →M
- eq($m1, m2$) :: (M, M) →Bool
- get(m, i, j) :: (M, Int, Int) →Float
- set(m, i, j, val) :: (M, Int, Int, Float) →()
- dims(m) :: M →(Float * Float)
- row(m, i) :: (M, Int) →Array Float
- col(m, j) :: (M, Int) →Array Float
- toArray(m) :: M →Array Float
- fromArray($arr, dim1$) :: (Array Float, Int) →M
- fromArray2d($arr, dim1$) :: (Array (Array Float)) →M
- fromList2d($arr, dim1$) :: (List (List Float)) →M
- submatrix($m, i, j, dim1, dim2$) :: (M, Int, Int, Int, Int) →Matrix Float

This basic interface includes functions for creating a (zeroed) matrix, accessing and mutating it, and converting to and from one-dimensional (row-major) arrays, or from two-dimensional arrays and lists. A very important note about Array extraction operators such as `toArray`, `row`, and `column` is that they provide *no guarantee* as to whether or not they return an alias to the existing storage in the matrix, or newly allocated storage. We need to leave both these possibilities open because of the diversity of possible backends, platforms, and matrix implementations. Thus the Arrays returned from these operations must be treated as *immutable*.

In addition to the above basic matrix functions, the matrix library namespace also includes common linear algebra matrix operations.

- add($m1, m2$) :: (M, M) →M
- sub($m1, m2$) :: (M, M) →M
- mul_elements($m1, m2$) :: (M, M) →M
- div_elements($m1, m2$) :: (M, M) →M
- scale($m, coef$) :: (M, Float) →M
- add_constant($m1, const$) :: (M, Float) →M
- mul(m) :: (M, M) →M →M
- invert(m) :: M →M

The above operations are purely functional. That is, they do not destroy their arguments; they allocate new matrices to store the results of their computations. Because matrices can be large, this is not always desirable. The matrix library includes destructive, *in place* versions of all the above operations (except `mul` and `invert`): for example, `add_inplace`. These mutate their first argument and return unit, "`()`", rather than returning a new matrix.

Higher order matrix operations

Below are additional *higher order* matrix operations (those that take functions as arguments).

- build :: (Int, Int, (Int, Int) →Float) →M
- foreach :: (Float →(), M) →()
- foreachi :: ((Int, Int, Float) →(), M) →()
- rowmap :: (Array Float →b, M) →Array b
- map :: (Float →Float, M) →M
- map2 :: ((Float,Float) →Float, M, M) →M
- map_inplace :: (Float →Float, M) →()
- map2_inplace :: ((Float,Float) →Float, M, M) →()

The `build`, `foreach`, `foreachi`, and `map` operations follow the same conventions as the standard WaveScript container operators of the same names. For example, `build(`*rows*`, `*cols*`, `*f*`)` builds a matrix of the specified size by applying f at every (i, j) index pair. Additionally, `rowmap` exposes each individual row as an array, and the "inplace" map variants provide destructive updates.

A note on polymorphism

The operations contained in `Matrix:Float` are *monomorphic*. They operate only on matrices of floats. This is due to the fact that these are wrappers around non-polymorphic native C routines. While the pure WaveScript matrix implementation provides these monomorphic interfaces for compatibility, it also provides polymorphic operations directly under the `Matrix` namespace (i.e. `Matrix:add`). There are definite advantages to the polymorphic interface. For example, the `Matrix:map` function can be used to apply a function f ::Float→Complex to build a complex matrix from a float matrix. The monomorphic versions can only map float matrices to float matrices and complex to complex.

Going further: other GSL functionality

The GSL libraries contained a wealth of functionality not currently exposed in WaveScript. It is straightforward to extend `gsl.ws` to export more of this functionality, but there has not been the opportunity or need to do so at the moment. You can see the GSL documentation here:

`http://www.gnu.org/software/gsl/manual/html_node/`

If there's something you need, bug Ryan to add it (`newton@mit.edu`), or take a look at `gsl.ws.pp` and `matrix_gsl.ws.pp` and try to add it yourself.

Chapter 5

WaveScript Evaluation Model

WaveScript is a *metaprogramming* language. For further explanation, I'll refer you to a quote from the *Matrix Reloaded* that was brought to my attention by Yannis Smaragdakis.

> NEO: Programs hacking programs. Why?
> ORACLE: They have their reasons, ...

The *reason* in WaveScript, is that we want to write programs of a different character than we want to *run*. We call this *asymmetric metaprogramming*. We want to *write* abstract, reusable, polymorphic, higher-order programs, but we want to *run* parallelizable dataflow graphs where individual nodes are fast, monomorphic, first-order, C-programs that don't require a heavyweight runtime. In WaveScript your program essentially *generates* the specialized, high-performance stream-processing program that is subsequently compiled and run.

How does this work? WaveScript runs your program through an interpreter at compile time. Your program returns a stream value. WaveScript must then make sure that it can convert that stream-value back to code, and reduce that code to fit the restrictions of the backend. There are restrictions on the backend because we omit heavyweight language features to target tiny embedded devices (e.g. no functions as values / no closures). Hence *asymmetric* metaprogramming.

Your is therefore really two programs: a *meta*-program and an *object*-program. The meta-program runs at compile-time and generates the object-program. Other metaprogramming systems typically have heavyweight quotation mechanisms for explicitly constructing object code, so you know exactly what object-code you get out. However, these mechanisms are also very onerous, and place a burden on the programmer.

WaveScript intentionally blurs the distinction between the stages—uses the same language, the same syntax, semantics, and libraries for both stages—and most of the time you don't need to think about it. But sometimes you do. In particular, you have to be careful not to try to store function values in data structures within object code (inside `iterates`). Recursive functions are also currently disallowed in object code.

5.1 Execution Model

UNFINISHED

After the metaprogram evaluates, what is left is a dataflow graph of stream operators (kernels). Now you may ask: what are the semantics of this dataflow graph's execution? WaveScript implements an *asynchronous dataflow* model. Streams are sequences discrete events, with no implied synchronization. Stream kernels are thus event-handlers. Further, the order of evaluation of kernels within a data flow graph is entirely at the whim of the scheduler, subject to loose fairness constraints. Any kernel with input data available is *ready* and may execute at any time; yet no ready kernel may execute an infinite number of times without all other ready kernels executing.

Therefore computations must be robust to the timing of the communication channels between kernels. After all, the program may be distributed, with some channels going over slow wireless links, and other channels intra-node. How do we accomplish this? Well, we rely on well-engineered libraries to insulate programmers from as much of the pain of asynchronicity as possible. In particular, be careful whenever multiple streams are being merged. If there is a one-to-one correspondence between stream elements, use `zip` library functions if possible.

Appendix A

Primitive table

This is an automatically generated table of primitives and their type signatures. Also see the libraries in `lib/` for additional standard functions.

```
cons   :: ('a, List 'a) -> List 'a
car    :: List 'a -> 'a
cdr    :: List 'a -> List 'a
head   :: List 'a -> 'a
tail   :: List 'a -> List 'a
append :: (List 'a, List 'a) -> List 'a
List:head    :: List 'a -> 'a
List:tail    :: List 'a -> List 'a
List:ref     :: (List 'a, Int) -> 'a
List:append  :: (List 'a, List 'a) -> List 'a
List:make    :: (Int, 'a) -> List 'a
List:length  :: List 'a -> Int
List:reverse :: List 'a -> List 'a
List:toArray :: List 'a -> Array 'a
List:assoc   :: (List ('a * 'b), 'a) -> List ('a * 'b)
List:assoc_update  :: (List ('a * 'b), 'a, 'b) -> List ('a * 'b)
gint   :: Int -> #a
g+     :: (#a, #a) -> #a
g-     :: (#a, #a) -> #a
g*     :: (#a, #a) -> #a
g/     :: (#a, #a) -> #a
g^     :: (#a, #a) -> #a
makeComplex  :: (Float, Float) -> Complex
int16ToInt   :: Int16 -> Int
int16ToInt64 :: Int16 -> Int64
int16ToFloat :: Int16 -> Float
int16ToDouble      :: Int16 -> Double
int16ToComplex     :: Int16 -> Complex
int64ToInt   :: Int64 -> Int
int64ToInt16 :: Int64 -> Int16
int64ToFloat :: Int64 -> Float
int64ToDouble      :: Int64 -> Double
int64ToComplex     :: Int64 -> Complex
intToInt16   :: Int -> Int16
intToInt64   :: Int -> Int64
intToFloat   :: Int -> Float
intToDouble  :: Int -> Double
intToComplex :: Int -> Complex
floatToInt16 :: Float -> Int16
floatToInt64 :: Float -> Int64
floatToInt   :: Float -> Int
floatToDouble      :: Float -> Double
floatToComplex     :: Float -> Complex
doubleToInt16      :: Double -> Int16
doubleToInt64      :: Double -> Int64
doubleToInt  :: Double -> Int
doubleToFloat      :: Double -> Float
doubleToComplex    :: Double -> Complex
complexToInt16     :: Complex -> Int16
complexToInt64     :: Complex -> Int64
complexToInt :: Complex -> Int
complexToDouble    :: Complex -> Double
complexToFloat     :: Complex -> Float
stringToInt  :: String -> Int
stringToFloat      :: String -> Float
stringToDouble     :: String -> Double
stringToComplex    :: String -> Complex
roundF :: Float -> Float
+_     :: (Int, Int) -> Int
-_     :: (Int, Int) -> Int
*_     :: (Int, Int) -> Int
/_     :: (Int, Int) -> Int
^_     :: (Int, Int) -> Int
+I16   :: (Int16, Int16) -> Int16
-I16   :: (Int16, Int16) -> Int16
*I16   :: (Int16, Int16) -> Int16
/I16   :: (Int16, Int16) -> Int16
^I16   :: (Int16, Int16) -> Int16
+I64   :: (Int64, Int64) -> Int64
-I64   :: (Int64, Int64) -> Int64
*I64   :: (Int64, Int64) -> Int64
/I64   :: (Int64, Int64) -> Int64
^I64   :: (Int64, Int64) -> Int64
+.     :: (Float, Float) -> Float
-.     :: (Float, Float) -> Float
*.     :: (Float, Float) -> Float
/.     :: (Float, Float) -> Float
^.     :: (Float, Float) -> Float
+D     :: (Double, Double) -> Double
-D     :: (Double, Double) -> Double
*D     :: (Double, Double) -> Double
/D     :: (Double, Double) -> Double
^D     :: (Double, Double) -> Double
+:     :: (Complex, Complex) -> Complex
-:     :: (Complex, Complex) -> Complex
*:     :: (Complex, Complex) -> Complex
/:     :: (Complex, Complex) -> Complex
^:     :: (Complex, Complex) -> Complex
realpart     :: Complex -> Float
imagpart     :: Complex -> Float
sqrtI  :: Int -> Int
sqrtF  :: Float -> Float
sqrtC  :: Complex -> Complex
moduloI      :: (Int, Int) -> Int
absI16 :: Int16 -> Int16
absI64 :: Int64 -> Int64
absI   :: Int -> Int
absF   :: Float -> Float
absD   :: Double -> Double
absC   :: Complex -> Float
logD   :: Double -> Double
exptI  :: (Int, Int) -> Int
cos    :: Float -> Float
sin    :: Float -> Float
tan    :: Float -> Float
acos   :: Float -> Float
asin   :: Float -> Float
atan   :: Float -> Float
max    :: ('a, 'a) -> 'a
min    :: ('a, 'a) -> 'a
=      :: ('a, 'a) -> Bool
<      :: ('a, 'a) -> Bool
>      :: ('a, 'a) -> Bool
<=     :: ('a, 'a) -> Bool
>=     :: ('a, 'a) -> Bool
not    :: Bool -> Bool
or     :: (Bool, Bool) -> Bool
and    :: (Bool, Bool) -> Bool
foreign      :: (String, List String) -> 'any
foreign_source     :: (String, List String) -> Stream 'any
exclusivePtr :: Pointer 'name -> ExclusivePointer 'name
getPtr :: ExclusivePointer 'name -> Pointer 'name
ptrToArray   :: (Pointer 'name, Int) -> Array 'a
```

```
ptrIsNull    :: Pointer 'name -> Bool
ptrMakeNull  :: () -> Pointer ""
inline_C     :: (String, String) -> Stream 'a
clock  :: () -> Double
realtime     :: () -> Int64
Mutable:ref  :: 'a -> Ref 'a
deref  :: Ref 'a -> 'a
timer  :: (List Annotation, Float) -> Stream ()
prim_window  :: (Stream 'a, Int) -> Stream (Sigseg 'a)
ensBoxAudio  :: Int -> Stream (Sigseg Int16)
ensBoxAudioF :: Int -> Stream (Sigseg Float)
ensBoxAudioAll     :: () -> Stream (Sigseg Int16)
readFile     :: (List Annotation, String, String, Stream 'a) -> Stream 'b
readFile     :: (List Annotation, String, Stream 'a, String, Int, Int, Int, Int, List Symbol) -> Stream 'a
readFile-wsearly   :: (String, Stream 'a, String, Int, Int, Int, Int, List Symbol) -> Stream 'a
fftC   :: Array Complex -> Array Complex
ifftC  :: Array Complex -> Array Complex
fftR2C :: Array Float -> Array Complex
ifftC2R      :: Array Complex -> Array Float
memoized_fftR2C    :: Array Float -> Array Complex
unionList    :: List (Stream 'a) -> Stream (Int * 'a)
_merge :: (List Annotation, Stream 'a, Stream 'a) -> Stream 'a
feedbackloop :: (Stream 't, Stream 't -> Stream 't) -> Stream 't
gnuplot_array      :: Array #a -> ()
gnuplot_array2d    :: Array (#a * #b) -> ()
gnuplot_process    :: (Stream String, Stream String) -> Stream 'any
spawnprocess :: (String, Stream String) -> Stream String
Array:make   :: (Int, 'a) -> Array 'a
Array:makeUNSAFE   :: Int -> Array 'a
Array:ref    :: (Array 'a, Int) -> 'a
Array:length :: Array 'a -> Int
Array:toList :: Array 'a -> List 'a
m_invert     :: Array (Array #a) -> Array (Array #a)
internString :: String -> Symbol
uninternString     :: Symbol -> String
HashTable:make     :: Int -> HashTable 'key 'val
HashTable:contains :: (HashTable 'key 'val, 'key) -> Bool
HashTable:get      :: (HashTable 'key 'val, 'key) -> 'val
HashTable:set      :: (HashTable 'key 'val, 'key, 'val) -> HashTable 'key 'val
HashTable:rem      :: (HashTable 'key 'val, 'key) -> HashTable 'key 'val
iterate      :: (List Annotation, ('in, VQueue 'out) -> VQueue 'out, Stream 'in) -> Stream 'out
show   :: 'a -> String
gnuplot_array_stream     :: Stream (Array #a) -> Stream (Array #a)
gnuplot_sigseg_stream    :: Stream (Sigseg #t) -> Stream (Sigseg #t)
gnuplot_array_stream2d   :: Stream (Array (#a * #b)) -> Stream (Array (#a * #b))
gnuplot_sigseg_stream2d  :: Stream (Sigseg (#a * #b)) -> Stream (Sigseg (#a * #b))
string-append      :: (String, String) -> String
String:length      :: String -> Int
String:explode     :: String -> List Char
String:implode     :: List Char -> String
intToChar    :: Int -> Char
charToInt    :: Char -> Int
toArray      :: Sigseg 'a -> Array 'a
toSigseg     :: (Array 'a, Int64, Timebase) -> Sigseg 'a
joinsegs     :: (Sigseg 'a, Sigseg 'a) -> Sigseg 'a
subseg :: (Sigseg 'a, Int64, Int) -> Sigseg 'a
seg-get      :: (Sigseg 'a, Int) -> 'a
width  :: Sigseg 'a -> Int
start  :: Sigseg 'a -> Int64
end    :: Sigseg 'a -> Int64
timebase     :: Sigseg 'a -> Timebase
Array:set    :: (Array 'a, Int, 'a) -> ()
HashTable:set_BANG :: (HashTable 'key 'val, 'key, 'val) -> ()
HashTable:rem_BANG :: (HashTable 'key 'val, 'key) -> ()
print  :: 'a -> ()
emit   :: (VQueue 'a, 'a) -> ()
break  :: () -> 'a
wserror      :: String -> 'a
inspect      :: 'a -> 'a
GETENV :: String -> String
FILE_EXISTS  :: String -> Bool
SHELL  :: String -> String
SETCPU :: (Int, Stream 't) -> Stream 't
SETCPUDEEP   :: (Int, Stream 't) -> Stream 't
map    :: ('a -> 'b, List 'a) -> List 'b
fold   :: (('acc, 'b) -> 'acc, 'acc, List 'b) -> 'acc
List:map     :: ('a -> 'b, List 'a) -> List 'b
List:zip     :: (List 'a, List 'b) -> List ('a * 'b)
List:fold    :: (('acc, 'b) -> 'acc, 'acc, List 'b) -> 'acc
Array:map    :: ('a -> 'b, Array 'a) -> Array 'b
Array:fold   :: (('acc, 'b) -> 'acc, 'acc, Array 'b) -> 'acc
Array:andmap :: ('a -> Bool, Array 'a) -> Bool
Array:build  :: (Int, Int -> 'a) -> Array 'a
List:build   :: (Int, Int -> 'a) -> List 'a
IS_SIM :: Bool
nullseg      :: Sigseg 'a
Array:null   :: Array 'a
nulltimebase :: Timebase
```