FASPL:

The Fast Adaptive Spatial Programming Language

A Project Paper

Presented for the

Master of Science

Degree

The University of Mississippi

David Albert Thigpen

August 2011

To the Graduate Council:

I am submitting herewith a project paper written by David Albert Thigpen entitled “FASPL: The Fast Adaptive Spatial Programming Language.” I have examined the final copy of this project paper for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

_____

Dr. Philip J. Rhodes, Associate
Professor of Computer Science

STATEMENT OF PERMISSION TO USE

In presenting this master's project in partial fulfillment of the requirements for a Master's degree at The University of Mississippi, I agree that the Library shall make it available to borrowers under rules of the Library. Brief quotations from this master's project are allowable without special permission, provided that accurate acknowledgment of the source is made.

Permission for extensive quotation from or reproduction of this master's project may be granted by my major professor or in his absence, by the Head of Interlibrary Services when, in the opinion of either, the proposed use of the material is for scholarly purposes. Any copying or use of the material in this master's project for financial gain shall not be allowed without my written permission.

Signature_____

Date_____

DEDICATION

This work is dedicated to my parents.

They've given a lot so that my brothers and I could succeed,

and have always pushed me to excel in all of my work.

ACKNOWLEDGMENTS

I would like to thank all of the people who have in some way influenced the direction of parts of this document. Specifically, I would like to thank Dr. Philip Rhodes for giving me several dozen hours of his time in the forming of the syntax of the language and the testing of the compiler and its output.

I would also like to acknowledge the fact that I benefited greatly from my compiler construction class under R. Stephen Rice and reading the textbook for that class [Louden 1997]. The shell of the code from my c minus compiler in compiler construction was reusable in my work on building the compiler for FASPL.

I also would like to acknowledge the flow of the order in which things are covered in the programming manual part of this paper is largely based on the ordering of the C programming manual [Kernighan 1988], which is the model of how to write a good programming manual.

ABSTRACT

FASPL – the Fast Adaptive Spatial Programming Language – is a scripting language built to help users create tailor-made metadata for datasets in the Granite database system. It combines this with general scripting abilities to form a suite of tools for automatic creation of Granite DataSource hierarchies.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER I

INTRODUCTION

FASPL, the Fast Adaptive Spatial Programming Language, is primarily an interpreted language that is built to allow users to manipulate spatial data that is modeled in the Granite scientific database system.

Granite is a scientific database system written in Java. It provides scientists with a way to organize multidimensional data, and access that data in an efficient manner [Rhodes 2004]. It can be used by scientists to access data remotely over the grid [Rhodes and Ramakrishnan 2005]. It can be used by scientists to access data remotely within a cluster [Yan 2006].

As such, it has applications in any field of science where researchers are working with multidimensional scientific data and might need to be able to access that data remotely. One example of a dataset that can be stored and accessed via Granite is the visual woman dataset from the National Library of Medicine's visual human project [Rhodes, Tang et al. 2005].

Within Granite, datasets like the visual woman dataset or any other datasets used by scientists are stored in binary files on a hard disk. The way Granite knows how to go about reading the data from the binary file is through the use of XFDL files, which are XML files that describe a *physical datasource*. A physical datasource is a datasource directly associated with a file on disk or remote machine. The XFDL file provides Granite with the dimensionality and bounds of the data, the number of attributes at each

point in the data and the location of the data.

In addition to the physical datasources that XFDL files describe, there are also *composite datasources* that are the result of operations between physical datasources or other composite datasources. Composite datasources are described by XML files called XDDL files that describe the operations being performed on physical datasources, which are represented in XDDL files by the location of an XFDL file.

The problem with this is that as the number of operations grows the complexity of the resulting XDDL file would also grow, so that the number of operations that a researcher might want to run on a piece of data might be limited by the researcher's ability to comprehend the program that the researcher wrote in the XDDL file. FASPL provides a certain level of abstraction for researchers from the XDDL file and gives researchers an abbreviated way of running the operations provided within XDDL files.

In this way, FASPL provides users with the ability to nest a large number of join operations using relatively simple statements that don't force the user to wade through thousands of lines of XML to figure out what is happening, because FASPL generates the XDDL file for the user. It also allows users the flexibility to reuse code through manipulating its six main data types - ints, doubles, strings, inputs, outputs and datasets – to control the labels used for fieldnames and the integers used for bounds. As a result, one function can be specified for a particular combination of operations and be reused for a significantly large series of nested operations with varying fieldnames and bounds. This is what makes FASPL a valuable tool for researchers working with spatial data that is stored within the Granite scientific database system.

CHAPTER II

LANGUAGE, COMPILER AND IMPLEMENTATION CHOICES

There are a lot of decisions that have to be made in terms of how to accomplish a given task and what is the best way to implement a given solution. In many cases, the time spent on evaluating the options and planning the implementation take more time than actually building the solution.

II.1. Language Choices

While FASPL was chosen as a solution to the problem of giving researchers a better way to perform blocked and attribute joins, there were other options that could have been pursued to achieve the same goal.

Researchers could have chosen to use a templating language to generate the XDDL file. A templating language is a language where the programming language is embedded within the output [Lerdorf 2006]. PHP is a good example of a templating language.

```
<p>
This is a paragraph about <? if(isset($title))?
$title:”nothing” ?>
</p>
```

Figure 1

Figure one shows an example of code in PHP being used to embed code within a

statement. Users of templating languages like PHP have to be aware of the format of the HTML output when writing the code. If there are any changes to the HTML format that is being accepted by browsers, programmers that build their applications in PHP have to know this and make the necessary changes to their applications in order for them to still run. In FASPL, users are not required to know the underlying format of the XDDL file.

Had the use of a templating language been pursued for the case of XDDL files, programmers would be forced to track any changes in the underlying format of XDDL files in order for their program to still run. By having a layer of abstraction between the user and the XDDL file format, the only change that has to be made when there is a change in the XDDL file format is with the code generator within the compiler for FASPL. In this case, the researcher's FASPL code is never broken. All the researchers have to do is run their program through the newest version of FASPL to update their XDDL file to the new format. This puts the weight of keeping the compiler code up to date on the people managing the format of the XDDL file accepted by Granite.

Another option that could have been pursued instead of writing a new programming language is creating a library of functions that users could use to generate a XDDL file. In many ways, a programming language is made up of a library of functions within the compiler and is in that sense equivalent. However, we decided that the possibility of creating a special purpose syntax that might be easier to read for users would be worthwhile.

The reasons for choosing to build a programming language are that the language could be extended for circumstances that are specific to Granite and spatial data without

deprecating older code; the language could say more in fewer lines, simplifying coding; and the language could use forms familiar to programmers to make programming easier. To accomplish all of this, the compiler would use a process similar to most other languages.

II.2. Compiler Implementation

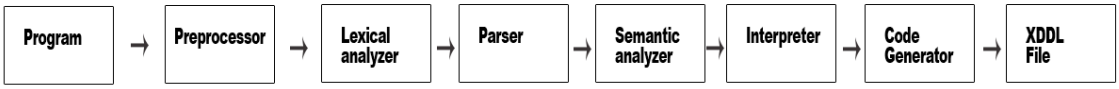

Figure 2

Figure two shows the compile steps used by the FASPL compiler. The program name is fed into the compiler at the command line. The compiler checks to see if the file exists and has the .faspl extension. The preprocessor then reads through the file looking for include statements and appending all of the included files. The preprocessor then passes the input to the lexical analyzer that proceeds to create tokens that are passed on to the parser. As a file is being parsed, a parse tree is created by the parser. A specification for the lexical analyzer and a yacc parser can be found in Appendix I and Appendix II respectively. Most of the implementation logic has been taken out of the two specifications for brevity.

```
reserved word  function
L:reserved word     dataset
    L:identifier         main
R:reserved word     variable
```

```
    L:identifier         A
    reserved word   variable
    L:identifier         B
    reserved word   variable
    L:identifier         C
    special symbol       =
    L:identifier         A
    R:string        head.float.xfdl
    special symbol       =
    L:identifier         B
    R:string        head.RGBfloat.xfdl
    special symbol       =
    L:identifier         C
    R:special symbol     [+]
         L:special symbol     dataset []{}
              L:identifier         A
              R:special symbol     []{}
                   L:reserved word      variable
                        string          density
         R:special symbol     dataset []{}
              L:identifier         B
              R:special symbol     []{}
                   L:string        red
                        reserved word   variable
    reserved word   return
    L:identifier         C
```

Figure 3

Figure three shows an example of a parse tree generated by the parser for a program. Once the parse tree has been generated by the parser, it is then passed onto the semantic analyzer, which goes through the program to make sure that there aren't any semantic errors, such as assigning a string to an int. As the semantic analyzer goes through the program, it is creating a symbol table and adding elements for each scope and then popping them off as it runs through the program.

After the semantic analyzer is done, there are only elements in the zero scope that are still in the symbol table. These elements are all functions, and their elements in the

table have pointers to the point in the parse tree that their instructions are located. This symbol table is then used by the interpreter find the instructions that it needs to run starting with the instructions for the main function. As the interpreter is running, it produces a result tree that is a tree of all of the dataset operations that are run to produce the return value for main, which is a dataset. By this point, all of the operations between ints, doubles and strings have been processed and the results are embedded in the tree.

After the interpreter has finished running all of the instructions, the results tree is passed onto a code generator, which does a post order traversal of the results tree and generates the XDDL file that matches the operations performed on the datasets in the program. The XDDL file is stored at the file location that the researcher specified on the command line or in the program.

CHAPTER III

LANGUAGE DESCRIPTION

For the most part, FASPL joins a long line of languages that have based most of their syntax off of the C programming language and other derivative languages like C++ and Java.

As such, each function has the operations that happen within the function defined by the operations that take place between the opening and closing braces, and variables are declared in the same manner as data types in the C programming language with the exception of the fact that global variables cannot be declared.

Table 1

Data types supported by FASPL	
Data type	Description
int	Integers that are used in arithmetic
double	Floating point numbers that can be used in arithmetic
string	A collection of characters that occupy a defined space in memory. From the compiler side, they are always null terminated.
dataset	A defined data hierarchy that represents a physical dataset or a series of operations involving physical datasets.
input	A data type that represents an input stream.
output	A data type that represents an output stream.

These variables and functions within FASPL can be any one of six data types referenced in table one. FASPL has the int and double, which are familiar to most C programmers. FASPL has the string, which is familiar to most Java programmers.

FASPL has the input and output data types, which are based off of the C++ data types used in file io. The one data type that sets FASPL apart is the dataset, which plays a major role in the output of FASPL.

III.1. Spatial Operators

As a result, all programs in FASPL are required to have a main function of type dataset, which will serve as the starting point for all applications written in FASPL. The dataset is used in a series of spatial operations that allow programmers to combine and manipulate datasets. In much the same way that ints, doubles and strings are initially defined, the leaves of a dataset tree are defined through the use of an assignment operator that assigns a string to the dataset variable. The string is a reference to the location of the XFDL file associated with the dataset. Within the XDDL file, the physical files that are associated with a dataset are called physical datasources. The datasets that are result of a series of operations are called composite datasources.

Table 2

Operations on datasets supported by FASPL	
Operation	Description
a[+]b	a is attribute joined with b
a{+}b	a is block joined with b

The operations supported by FASPL for use with datasets are the *attribute join*

and the *block join*, which each have unique way of manipulating spatial data. Table two shows the format used to describe these operations within FASPL. Conceptually, the spatial operators can be hard to understand.

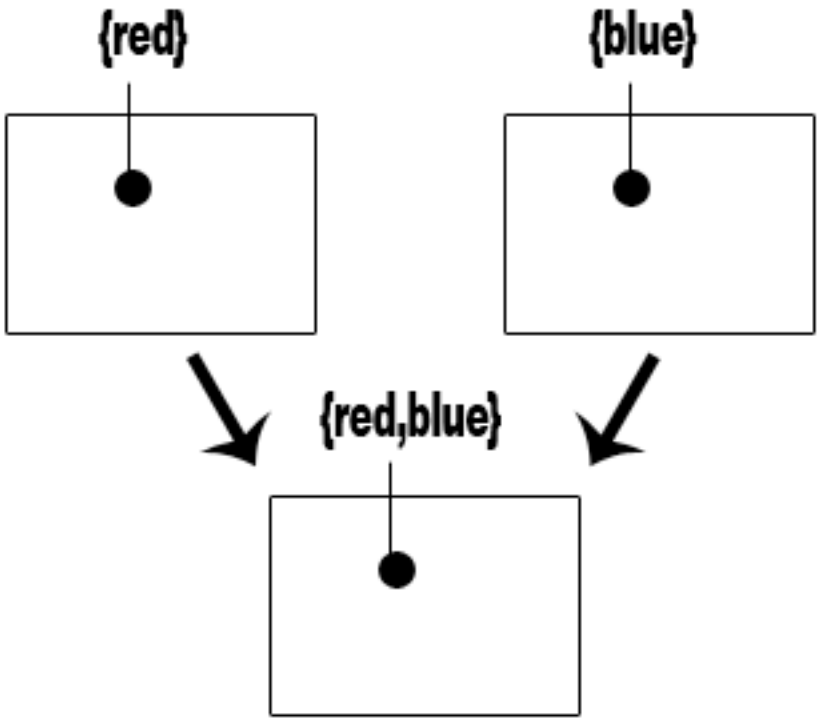

Figure 4

Figure 4 helps visualize what is actually happening when an attribute join is run on a two different datasets. The operation takes data from each of the datasets and combines the specified attributes in each of the datasets for each defined point in the datasets to form a new dataset. In this case, the Attribute join described in figure 4 is

taking *red* from the datasource on the left and *blue* from the datacource on the right to create a datasource that had both *red* and *blue* described for all points where their was data for each attribute.

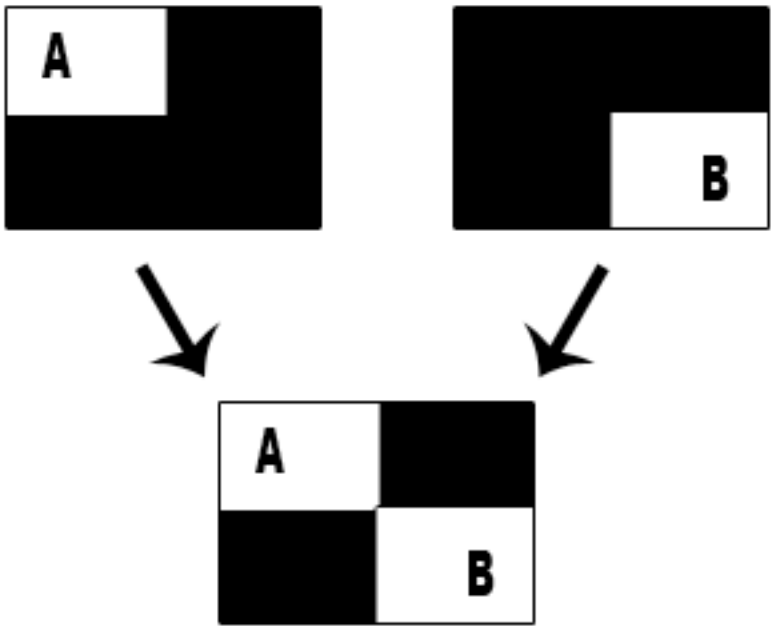

Figure 5

Figure 5 helps visualize what is actually happening when a block join is run on a two different datasets. The operation takes data defined within the specified bounds of each source, which are currently defined in different bounds and combines the spaces such that the two datasources are accessible from one datasources with a new set of bounds.

Within the XDDL files used by Granite, the attributes like the ones stated in the attribute join are called *mappings*, and the bounds like the ones stated in the block join are called *bounds* within Granite. Both the mappings and the bounds are listed under a component element that references the name of the datasource that the mappings and bounds are coming from. Attribute joins require that attributes are defined for one or both elements. Block joins require that attributes are defined for only one elements. Defining bounds is optional for attribute and block joins. Within FASPL, these mappings and bounds can be referenced using a statement that is called a *compound dataset*, which allows users to define just attributes, just bounds or attributes and bounds.

```
E["red", void]{0:127, 0:127, 0:127}
```

Figure 6

In the compound dataset in figure six, the strings in the square brackets represent attributes that are being extracted from the dataset E. The void that is in the attribute list is an empty position that will be filled by the dataset that E will be involved in an operation with. Each of the attributes can be a string expression or void. In any operation, at the time that the compiler gets to a given operation all positions below the highest position defined in the compound datasets must be defined, otherwise the code will not work in Granite. The numbered pairs that are in the curly brackets are the upper and lower bounds for the dataset E for each dimension of the dataset. Each number in the pair can be an int expression. Attributes and bounds can be specified explicitly via the

compound dataset or implicitly by stating just the variable in an operation and letting the attributes defined by prior operations be used.

III.2. General Syntax

In addition to the operations defined for use with datasets there are also operations that are meant to be used with the other data types in FASPL.

Table 3

Operations that are only supported for use with ints, doubles and strings by FASPL	
Operation	**Definition of operation**
n+m	n plus m

Table three has a list of the operations that are supported by the int, double and string data types. For ints and doubles, the plus operator is meant as an addition operator, and, for strings it is meant as a concatenation operator similar to Java

Table 4

Operations that are only supported for use with ints and doubles by FASPL	
Operation	**Definition of operation**
n-m	n minus m
n*m	n times m
n/m	n divided by m

In addition to the plus operator, ints and doubles have support for subtraction, multiplication and division in the form described in table four.

Table 5

Operations that are only supported for use with ints in FASPL	
Operation	**Definition of operation**
n%m	n mod m

The int data type also has support for the mod operator in the form described in table five. All of the operators that are supported by the int, double and string data types are meant to be in furtherance of the goal of allowing for the manipulation of attributes and bounds within compound datasets and for the facilitation of iteration and selection statements. FASPL has if, while, do while and for statements that are all in the same form as the C programming language.

Table 6

Comparison operators supported by FASPL for use with ints, doubles and strings		
Statement	**Meaning**	**Resulting value**
n==m	n is equal to m	1 if true, 0 if false
n!=m	n is not equal to m	1 if true, 0 if false
n<m	n is less than m	1 if true, 0 if false
n<=m	n is less than or equal to m	1 if true, 0 if false
n>m	n is greater than to m	1 if true, 0 if false
n>=m	n is greater than or equal to m	1 if true, 0 if false
!n	not n, where if n is equal to zero the result is true, otherwise the result if false.	1 if true, 0 if false
n	If n is not equal to zero, it is true, otherwise it is false.	n

Table six is a listing of all of the comparison operators that are supported by FASPL for use in iteration and selection statements. The only difference between the operators and the operators C counterparts is the fact that strings can be compared using the same operators that a programmer would use to compare ints and doubles in C.

FASPL also has support for file io through the use of the input and output data

types. The location of the file that is going to be used for input or output is specified through the use of an assignment operator that assigns a string to the input or output variable.

Table 7

Operations supported for use with streams in FASPL	
Operation	**Definition of operation**
a<<b	Send the string b to the output stream pointed to by a
a>>b	Send the input from the input stream a to the string b

The left and right shift operators are used to send data to or from an output or input stream respectively in much the same manner as the left and right shift operators are used in C++. Like C++, FASPL allows programmers to have an endlessly nested series of left or right shift operators. Table seven has a formal definition of how the operators work. In the case of the left shift operator the variable b referenced in table seven could just as well have been a string expression. The right shift operator requires that the stream be shifting data into a string variable.

Table 8

Streams supported for use with streams in FASPL	
Stream	**Definition of operation**
stdin	Standard input stream
stdout	Standard output stream
xmlout	The output stream for the XDDL file

As part of the input and output data type, there are some global variables that the

programmer gets for free. The programmer gets access to the standard input stream, the standard output stream and the XML stream that is used to output the XDDL file. These streams can be manipulated in much the same way that all other input and output streams can be manipulated.

Table 9

Escape Sequence Characters
\a Bell (beep)
\b Backspace
\f Formfeed
\n Newline
\r Return
\t Tab
\\ Backslash
\' Single quote
\" Double quote

With access to the standard input and output streams, it becomes useful to be able to format strings or have access to the special characters. Table nine has a listing of the special characters that are supported by FASPL.

In addition to the operations and statements that can be run within FASPL, the language also has support for multiline and single line comments. Multiline comments take place between the /* and the */ character combinations. Single line comments are begun with a // and end at the new line character. The interpreter will not see the comment.

CHAPTER IV

TUTORIAL

This chapter goes over how to use FASPL to perform attribute and block joins and how to take full advantage of the unique features that FASPL offers with regard to dataset manipulation.

IV.1. Generating XDDL files and Manipulating Datasets

```
/* dataset_fun.faspl */
dataset main()
{
    return "test1.xfdl";
}
```

Figure 7

The program in figure seven returns a physical datasource and performs no operations on datasets. It is an example of assigning a string to a dataset and creating a physical datasource.

```
./faspl test/dataset_fun.faspl
```

Figure 8

The command used to run the example program in figure seven is in figure eight.

All programs that are compiled have to have a .faspl extension. If the user would like to specify a location for the XDDL output file, it can be specified after the name of the program being run. If a file name is not specified, then the XDDL file is output to a.xddl in the directory from which the compiler is executed.

```
test/dataset_fun.faspl:a.xddl
```

Figure 9

The output from the program in figure seven to standard out is in figure nine. The compiler will always output the name of the program being run first followed by the name of the output file specified at the command line or the name of the default output file a.xddl. Any output from the program to standard out will come after that line.

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE DataSourceList PUBLIC '-//SDB//DTD//EN'
'ddl.dtd'>
<DataSourceList>
        <DataSource dsName='op0' type='physical'
fileName='test1.xfdl'>
        </DataSource>
        <PublicDS dsName='op0' />
</DataSourceList>
```

Figure 10

The output to the destination file a.xddl is in figure 10. The output from this program displays the physical datasource op0, which was the return value of the program.

In terms of good programming practices and generating an XDDL file that

Granite can use, it is a good idea to have a physical dataset associated with the leaves of the operations. This will be shown in the following examples.

```
/* dataset_fun.faspl */
dataset main()
{
    dataset a;
    return a;
}
```

Figure 11

The program in figure 11 returns a null datasource and performs no operations on datasets.

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE DataSourceList PUBLIC '-//SDB//DTD//EN'
'ddl.dtd'>
<DataSourceList>
        <PublicDS dsName='op0' />
</DataSourceList>
```

Figure 12

The output to the destination file a.xddl from the program in figure 11 is in figure 12. Notice how the PublicDS with dsName op0 listed in the file is undefined. As a result, this output would not be usable in the granite system, since all of the elements referenced in an operation have to be defined.

```
/* dataset_fun.faspl */
dataset main()
{
    dataset a;
    dataset b;
    dataset c;
    a=b[+]c;
    return a;
}
```

Figure 13

The program in figure 13 returns an attribute join operation between two null datasets.

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE DataSourceList PUBLIC '-//SDB//DTD//EN'
'ddl.dtd'>
<DataSourceList>
        <DataSource dsName='op0' type='joined'>
                <Component dsName='op1'>
                </Component>
                <Component dsName='op2' >
                </Component>
        </DataSource>
        <PublicDS dsName='op0' />
</DataSourceList>
```

Figure 14

The output to the destination file a.xddl from the program in figure 13 is in figure 14. Notice how the dsNames op1 and op2 are mentioned in the component elements within the joined datasource (a type of datasource that is another name for an attribute join), but are undefined in the rest of the file. As a result, this output would not be usable

in the granite system, since all of the elements referenced in an operation have to be defined.

```
/* dataset_fun.faspl */
dataset main()
{
    dataset a;
    dataset b;
    dataset c;
    b="test1.xfdl";
    c="test2.xfdl";
    a=b[+]c;
    return a;
}
```

Figure 15

The program in figure 15 assigns a physical dataset to b and a physical dataset to c. It takes these two variables, attribute joins them and assigns them to the variable a, which is the return value of main.

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE DataSourceList PUBLIC '-//SDB//DTD//EN'
'ddl.dtd'>
<DataSourceList>
        <DataSource dsName='op1' type='physical'
fileName='test1.xfdl'>
        </DataSource>
        <DataSource dsName='op2' type='physical'
fileName='test2.xfdl'>
        </DataSource>
        <DataSource dsName='op0' type='joined'>
                <Component dsName='op1'>
                </Component>
                <Component dsName='op2' >
                </Component>
        </DataSource>
```

```
        <PublicDS dsName='op0' />
</DataSourceList>
```

Figure 16

The output to the destination file a.xddl from the program in figure 15 is in figure 16. The output from this program displays the physical datasets op1 and op2, which were from the assignment of the strings to the variables b and c respectively. The variables op1 and op2 are mentioned again in the joined datasource op0.

This operation doesn't list the mappings and the bounds for dsNames op1 and op2 that will be used in the operations. The mappings and bounds need to be referenced using a compound dataset.

```
/* attempt.faspl*/
dataset main(void)
{
    dataset A;
    dataset B;
    dataset C;
    dataset D;
    dataset E;
    dataset F;
    A="head.float.xfdl";
    B="head.RGBfloat.xfdl";
    C="head.RGBByte.xfdl";

    D= A[void,"density"] [+] B["red",void];

    E= A[ void, void, "density"]{0:127, 0:127, 0:127} [+]
B[void, "green", void]{0:127, 0:127, 0:127} [+] C["red",
void, void]{0:127, 0:127, 0:127};

    F= D["red", "density"] [+] E[void, void, "green"];

    return F;
```

```
}
```

Figure 17

The program in figure 17 produces output that is readable by Granite.

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE DataSourceList PUBLIC '-//SDB//DTD//EN'
'ddl.dtd'>
<DataSourceList>
        <DataSource dsName='op2' type='physical'
fileName='head.float.xfdl'>
        </DataSource>
        <DataSource dsName='op3' type='physical'
fileName='head.RGBfloat.xfdl'>
        </DataSource>
        <DataSource dsName='op1' type='joined'>
                <Component dsName='op2'>
                        <Map position='1'
fieldName='density' />
                </Component>
                <Component dsName='op3' >
                        <Map position='0'
fieldName='red' />
                </Component>
        </DataSource>
        <DataSource dsName='op5' type='physical'
fileName='head.float.xfdl'>
        </DataSource>
        <DataSource dsName='op7' type='physical'
fileName='head.RGBfloat.xfdl'>
        </DataSource>
        <DataSource dsName='op8' type='physical'
fileName='head.RGBByte.xfdl'>
        </DataSource>
        <DataSource dsName='op6' type='joined'>
<Component dsName='op7'>
                        <Map position='1' fieldName='green'
/>
                        <Bounds lower='0' upper='127' />
                        <Bounds lower='0' upper='127' />
```

```
                        <Bounds lower='0' upper='127' />
                </Component>
                <Component dsName='op8' >
                        <Map position='0'
fieldName='red' />
                        <Bounds lower='0' upper='127' />
                        <Bounds lower='0' upper='127' />
                        <Bounds lower='0' upper='127' />
                </Component>
        </DataSource>
        <DataSource dsName='op4' type='joined'>
                <Component dsName='op5'>
                        <Map position='2'
fieldName='density' />
                        <Bounds lower='0' upper='127' />
                        <Bounds lower='0' upper='127' />
                        <Bounds lower='0' upper='127' />
                </Component>
               <Component dsName='op6' >
                        <Map position='1' fieldName='green'
/>
                        <Map position='0'
fieldName='red' />
                        <Bounds lower='0' upper='127' />
                        <Bounds lower='0' upper='127' />
                        <Bounds lower='0' upper='127' />
                </Component>
        </DataSource>
        <DataSource dsName='op0' type='joined'>
                <Component dsName='op1'>
                        <Map position='0'
fieldName='red' />
                        <Map position='1'
fieldName='density' />
                        <Bounds lower='0' upper='127' />
                        <Bounds lower='0' upper='127' />
                        <Bounds lower='0' upper='127' />
                </Component>
                <Component dsName='op4' >
                        <Map position='2' fieldName='green'
/>
                        <Bounds lower='0' upper='127' />
                        <Bounds lower='0' upper='127' />
                        <Bounds lower='0' upper='127' />
                </Component>
```

```
         </DataSource>
        <PublicDS dsName='op0' />
</DataSourceList>
```

Figure 18

The XDDL output from the program in figure 17 is in figure 18. The output really isn't as verbose as it could be. It is possible to produce output that implicitly stores all of the attributes and bounds for a given physical dataset provided that the XFDL file referenced in the physical datasource assignment is accessible to the interpreter at the path specified in the string relative to the directory from where the interpreter is called.

```
/* dataset_fun.faspl */
dataset main()
{
    dataset A;
    dataset B;
    dataset C;
    dataset D;
    dataset E;
    dataset F;
    dataset G;
    A="head.float.xfdl";
    B="head.RGBfloat.xfdl";
    C="head.RGBByte.xfdl";
    D= A[void,"density"] [+] B["red",void];
    E= A[void,"density"] [+] C["red",void];
    G= D[void, void]{0:64,0:64,0:64} {+}
E["red","density"]{65:127,65:127,65:127};
    return G;
}
```

Figure 19

Up until now, all of the operations in the example programs involving datasets

have used attribute joins. The program in figure 19 shows an example of a program that performs a block join and generates an XDDL file that can be used by Granite.

```
?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE DataSourceList PUBLIC '-//SDB//DTD//EN'
'ddl.dtd'>
<DataSourceList>
        <DataSource dsName='op2' type='physical'
fileName='head.float.xfdl'>
        </DataSource>
        <DataSource dsName='op3' type='physical'
fileName='head.RGBfloat.xfdl'>
        </DataSource>
        <DataSource dsName='op1' type='joined'>
                <Component dsName='op2'>
                        <Map position='1'
fieldName='density' />
                </Component>
                <Component dsName='op3' >
                        <Map position='0'
fieldName='red' />
                </Component>
        </DataSource>
        <DataSource dsName='op5' type='physical'
fileName='head.float.xfdl'>
        </DataSource>
        <DataSource dsName='op6' type='physical'
fileName='head.RGBByte.xfdl'>
        </DataSource>
<DataSource dsName='op4' type='joined'>
                <Component dsName='op5'>
                        <Map position='1'
fieldName='density' />
                </Component>
                <Component dsName='op6' >
                        <Map position='0'
fieldName='red' />
                </Component>
        </DataSource>
        <DataSource dsName='op0' type='blocked'>
                <Component dsName='op1'>
```

```
                        <Bounds lower='0' upper='64' />
                        <Bounds lower='0' upper='64' />
                        <Bounds lower='0' upper='64' />
                </Component>
                <Component dsName='op4' >
                        <Map position='0'
fieldName='red' />
                        <Map position='1'
fieldName='density' />
                        <Bounds lower='65' upper='127' />
                        <Bounds lower='65' upper='127' />
                        <Bounds lower='65' upper='127' />
                </Component>
        </DataSource>
<PublicDS dsName='op0' />
</DataSourceList>
```

Figure 20

The XDDL output from the program in figure 19 is in figure 20. It is important to note that the output uses the type *blocked* to denote block joins in the XDDL file and the type *joined* to denote attribute joins in the XDDL file. From looking through the output, it is possible to trace the order of operation and see that the XDDL file output from the program does the operations in the order specified in the program.

IV.2. Automating Dataset Manipulation

FASPL provides programmers with several tools to help automate several of the operations that programmers might want to use to manipulate datasets. It is useful to look at some ways that these tools can be used to make the lives of researchers easier.

```
dataset main()
```

```
{
    input file;
    string dataset_filename;
    dataset a;
    dataset b;

    a="head.RGBfloat.xfdl";
    file = "dataset_list.dat";
    file >> dataset_filename;
    while(dataset_filename != "")
    {
         stdout << dataset_filename << "\n";
         b=dataset_filename;
         a= a["red","green", "blue"] {+}
b[void,void,void];
         file >> dataset_filename;
    }
    return a;
}
```

Figure 21

Figure 21 shows an example program that reads in a list of XFDL files from a file called dataset_list.dat, assigns each of the strings to a dataset to create a physical datasource and runs block joins on all of the datasets to create a composite datasource that is a combination of all of the files. This might be something that is done in the case of having blocks of a much larger datasource stored across multiple files that could be on different computers or hard disks. This is a very common situation when it comes to datasets in physics.

```
/* dataset_tile_for.faspl */
#include "test/string.faspl"

dataset main()
{
```

```
    string dataset_prefix;
    string dataset_maximum;
    string dataset_filename;
    int i;
    int max;
    dataset a;
    dataset b;

    a="head.RGBfloat.xfdl";
    stdout<<"What is the prefix for the datasets that you
have?";
    stdin >> dataset_prefix;
    stdout<<"How many datasets do you have?";
    stdin >> dataset_maximum;
    max = atoi(dataset_maximum);
    for(i=0;i<max; i=i+1)
    {
         dataset_filename = dataset_prefix+i+".xfdl";
         stdout << dataset_filename << "\n";
         b=dataset_filename;
         a= a["red","green", "blue"] {+}
b[void,void,void];
    }
    return a;
}
```

Figure 22

Figure 22 is an example of taking in input from standard in, converting the numeric input to integer using the function atoi and iterating through a series of block joins that have file names based on the earlier input. This is another way of handling the problem of joining data that is stored across several files on a system into one datasource. The function atoi referenced in the program is listed in the file string.faspl that was included at the top of the program. This is an example of the fact that while there aren't any libraries that come built into the language, it is possible to write libraries and include them anywhere in the program.

```
int atoi(string word)
{
     int i;
     int j;
     int length;
     int total;
     int decimal;
     int negative;

     negative = 1;
     for(i=0; word[i]!=""; i=i+1)
     {
     }
     length=i;
     total=0;

     for(i=0;i<length;i=i+1)
     {
          decimal=0;
          if(word[i]=="0")
          {
               decimal=0;
          }
          else if(word[i]=="1")
          {
               decimal=1;
          }
          else if(word[i]=="2")
          {
               decimal=2;
          }
          else if(word[i]=="3")
          {
               decimal=3;
          }
          else if(word[i]=="4")
          {
               decimal=4;
          }
          else if(word[i]=="5")
          {
               decimal=5;
```

```
          }
          else if(word[i]=="6")
          {
               decimal=6;
          }
          else if(word[i]=="7")
          {
               decimal=7;
          }
          else if(word[i]=="8")
          {
               decimal=8;
          }
          else if(word[i]=="9")
          {
               decimal=9;
          }
          else if(word[i]=="-")
          {
               negative=-1;
          }
          else
          {
               return 0;
          }
          for(j=0;j<length-i+1;j=j+1)
          {
               decimal=decimal*10;
          }
          total= total+decimal;
     }
     total = total * negative;
     return total;
}
```

Figure 23

Figure 23 is the format of the atoi function used in the program in figure 22. The atoi function is a good example of string manipulation in FASPL and the use of selection statements.

CHAPTER V

CONCLUSION

With all of the tools enumerated in the tutorial chapter, programmers should now have the ability to quickly and efficiently write a FASPL program that the FASPL compiler can use to generate a complex XDDL file that is capable of having hundred of thousands of nested join operations.

Even with FASPL's current set of tools, there are still some things that FASPL does not do that could be future work. The first thing is adding an execl functionality similar to C, so that it will be possible to interface with other database systems. After that, it might be good to add structures, the ability to listen to ports, multithreading and to create a module with apache, so users can create web content. All of that is for further down the line, and will to be approached with caution in order to prevent making the language too complicated for users to learn.

APPENDIX I

LEX SPECIFICATION

```
P               [.]
D               [0-9]
L               [a-zA-Z_]
H               [a-fA-F0-9]
E               [Ee][+-]?{D}+
FS              (f|F|l|L)
IS              (u|U|l|L)*

%{
#include <stdio.h>
#include <string.h>
#include "header.h"

void comment();
void line_comment();
void count();
int lineno;

%}


%%

"//"            { line_comment();  }
"/*"            { comment();  }


"return"        { return RETURN; }
"void"              { return VOID; }
"int"               { return INT; }
"double"        { return DOUBLE; }
"dataset"       { return DATASET; }
"string"            { return STRING; }
"input"             { return INPUT; }
"output"        { return OUTPUT; }

({D}*|{D}*{P}{D}*)  { return NUMBER; }
{L}({L}|{D})*       { return check_type(); }
```

```
\"(\\.|[^\\"])*\"        { return STRING_LITERAL; }

">>"           { return READ; }
"<<"           { return WRITE; }
"[+]"              { return ATTRIBUTE_JOIN; }
"{+}"              { return BLOCK_JOIN; }
"&&"           { return AND; }
"||"           { return OR; }
"<"            { return LT; }
"<="           { return LTE; }
">"            { return GT; }
">="           { return GTE; }
"=="           { return IE; }
"!="           { return NE; }
"!"            { return NOT; }
":"            { return ':'; }
";"            { return ';'; }
"{"            { return '{'; }
"}"            { return '}'; }
","            { return ','; }
"="            { return '='; }
"+"            { return '+'; }
"-"            { return '-'; }
"*"            { return '*'; }
"/"            { return '/'; }
"%"            { return '%'; }
"("            { return '('; }
")"            { return ')'; }
"["            { return '['; }
"]"            { return ']'; }

[\n\r]             { lineno++; }
[ \t\v\f]          {  ; }
.              { return yytext[0];}
<<EOF>>        {return EOF;}

%%

int yywrap()
{
     return 1;
}


void comment()
```

```
{
     char c;
     char c1;

     while(1){
          c = input();
          while (c != '*' && c != 0)
          {

               if(c=='\n' || c=='\r'){
                    lineno++;
               }
               c = input();
          }

          if ((c1 = input()) == '/' || c == 0)
          {
               break;
          }
          else
          {
               unput(c1);
          }
     }

}

void line_comment()
{
     char c, c1;

     while ((c = input()) != '\n' && c != '\r' && c != 0)
     {
     }
     lineno++;
}


int check_type ()
{
     int i;
     for (i=0;i<MAXRESERVED;i++)
          if (!strcmp(yytext,reservedWords[i].str))
               return reservedWords[i].tok;
     return IDENTIFIER;
```

```
}
```

APPENDIX II

YACC SPECIFICATION

```
%token BUFLEN MAXTOKENLEN ENDFILE ERROR RETURN VOID DATASET
%token INT IDENTIFIER NUMBER STRING_LITERAL ATTRIBUTE_JOIN
%token BLOCK_JOIN PAR AND OR LT LTE GT GTE IE NE NOT
%token COMMENT COMPOUND_DATASET DATSET_INFO IF WHILE FOR
%token DO ELSE STRING DOUBLE COORDINATE READ WRITE INPUT
%token OUTPUT


%start program
%%

program
     : declaration_list
     ;

declaration_list
     : declaration declaration_list
     | declaration
     ;

declaration
     : DATASET id '(' parameter_list ')' '{'
compound_statement '}'
     | DATASET id '(' ')' '{' compound_statement '}'
     | INT id '(' parameter_list ')' '{' compound_statement
'}'
     | INT id '(' ')' '{' compound_statement '}'
     | DOUBLE id '(' parameter_list ')' '{'
compound_statement '}'
     | DOUBLE id '(' ')' '{' compound_statement '}'
     | STRING id '(' parameter_list ')' '{'
compound_statement '}'
     | STRING id '(' ')' '{' compound_statement '}'
     ;

parameter_list
     : parameter ',' parameter_list
     | parameter
     ;
```

```
parameter
     : DATASET id
     | INT id
     | DOUBLE id
     | STRING id
     | INPUT id
     | OUTPUT id
     | VOID
     ;

compound_statement
     : variable_list statement_list
     | variable_list
     | statement_list
     ;

variable_list
     : variable variable_list
     | variable
     ;

variable
     : INT id ';'
     | DOUBLE id ';'
     | DATASET id ';'
     | STRING id ';'
     | INPUT id ';'
     | OUTPUT id ';'
     ;

statement_list
     : statement statement_list
     | statement
     ;

statement
     : return_statement
     | iteration_statement
     | selection_statement
     | '{' compound_statement '}'
     | '{' '}'
     | expression_statement
     ;
```

```
expression_statement
     : expression ';'
     | ';'
     ;

return_statement
     : RETURN expression_statement
     ;

iteration_statement
     : FOR '(' expression ';' expression ';' expression ')'
statement
     | WHILE '(' expression ')' statement
     | DO statement WHILE '(' expression ')' ';'
     ;

selection_statement
     : IF '(' expression ')' statement
     | IF '(' expression ')' statement ELSE statement
     ;

expression
     : write_expression
     | read_expression
     | assignment_expression
     ;

read_expression
     :  read_expression READ id
     | id
     ;

write_expression
     :  write_expression WRITE assignment_expression
     | id
     ;

assignment_expression
     : id '=' assignment_expression
     | simple_expression
     ;

simple_expression
     : add_expression NE simple_expression
     | add_expression IE simple_expression
```

```
     | add_expression LTE simple_expression
     | add_expression GTE simple_expression
     | add_expression LT simple_expression
     | add_expression GT simple_expression
     | add_expression
     ;
add_expression
     : mul_expression '+' add_expression
     | mul_expression '-' add_expression
     | mul_expression
     ;
mul_expression
     : mod_expression '*' mul_expression
     | mod_expression '/' mul_expression
     | mod_expression
     ;
mod_expression
     : join_expression '%' mod_expression
     | join_expression
     ;
join_expression
     : term ATTRIBUTE_JOIN join_expression
     | term BLOCK_JOIN join_expression
     | term
     ;

term
     : factor AND term
     | factor OR term
     | '-' term
     | NOT term
     | factor
     ;

factor
     : '(' expression ')'
     | id '(' ')'
     | id '(' argument_list ')'
     | id
     | id '[' info_list ']' '{' coordinate_list '}'
     | id '{' coordinate_list '}' '[' info_list ']'
     | id '[' info_list ']'
     | id '{' coordinate_list '}'
     | string
     | num
```

```
     ;

argument_list
     :  expression ',' argument_list
     | expression
     ;

info_list
     : info ',' info_list
     | info
     ;

info
     : expression
     | VOID
     ;

coordinate_list
     : coordinate ',' coordinate_list
     | coordinate
     ;

coordinate
     : expression ':' expression
     ;

id
     : IDENTIFIER
     ;
num
     : NUMBER
     ;
string
     : STRING_LITERAL
     ;


%%
```

WORKS CITED

[Kernighan 1988] Kernighan, B.W., and Ritchie, D.M., *The C Programming Language (2nd Edition)*. Englewood Cliffs, NJ: Prentice-Hall, 1988

[Ledorf 2006] Lerdorf, Rasmus, Kevin Tatroe and Peter MacIntyre, *Programming PHP*, Sebastopol, CA: O'Reilly Media, 2006

[Louden 1997] Louden, Kenneth C., *Compiler Construction: Principles and Practice,* Mason, OH: Cengage Learning Company, 1997

[Rhodes 2004] Rhodes, Philip J., *Granite: A Scientific Database Model and Implementation*, Dissertation at the University of New Hampshire, Durham, New Hampshire, 2004

[Rhodes and Ramakrishnan 2005] Rhodes, Philip J. and Sridhar Ramakrishnan, "Iteration Aware Prefetching for Remote Data Access*". Proc. 1st International IEEE Conference on e-Science and Grid Technologies (e-Science05),* Melbourne*,* 2005

[Rhodes, Tang et al. 2005] Rhodes, Philip J., Xuan Tang, R. Daniel Bergeron, and Ted M. Sparr "Iteration Aware Prefetching for Large Multidimensional Scientific Datasets". *Proc. SSDBM '05,* Santa Barbara*,* 2005

[Yan 2006] Yan, Baoqiang and Philip J. Rhodes, "An Iteration Aware Multidimensional Data Distribution Prototype for Computing Clusters", *Proc. Cluster '06,* 2006

VITA

Education

Degrees

The University of Minnesota, St. Paul, MN

- Ph.D. Applied Economics Expected Graduation Date May 2015

The University of Mississippi, Oxford, MS

- M.S. Computer and Information Science Expected Graduation Date August 2011
- M.A. Economics May 2011
- M.A. Journalism December 2010
- B.A. Computer and Information Science August 2007
- B.A. Journalism and Economics August 2007
- B.S. Math August 2007

Jackson Academy, Jackson, MS

- High School Diploma May 2002

Work Experience

Businesses

President/CEO

- *Thigpen Media Group, LLC* March 2011 to present
  Thigpen Media Group is a company dedicated to providing our audience with the finest content available, whether it be via the video, print or online. We have tried to leverage our vast technical experience to try to place this content in the most visually appealing light possible. Our current media properties include Stay Right for Life and Stay Left for Life.
- *Thigpen Consulting, LLC* March 2011 to present
  Thigpen Consulting is a business focused on providing an outlet for my indepedent contracting. These contract jobs use my skills as a writer, a programmer, a designer, a mathematician and an economist. A few of these jobs are referenced in my resume below, but only in cases where a formal title exists. Prior to the founding of this company, I would just get paid directly for my work.

Computer Science

Webmaster

- *MrMagazine.com*, January 2009 to present

Teaching

- *Mississippi Center for Supercomputer Research* September 2007 to May 2008
  Helped put together MPI and OpenMP teaching aids and taught classes on MPI, OpenMP, Mathematica and various other software packages or utilities used on MCSR systems.

Programming

- *University of Mississippi Office of Research and Sponsored Programs* September 2006 to May 2008
  Programmed web database applications in Javascript, PHP and MySQL, using the model-view-control design pattern and implementing standard security features to prevent security risks like session stealing.

Journalism

Editor

- *MrMagazine.com*, August 2008 to January 2009

Managing Editor

- *Samir Husni's Guide To New Magazines*, March 2009
- *MrMagazine.com*, January 2009 to present
- *MrMagazine.com*, June 2006 to January 2007
- *Samir Husni's Guide To New Magazines*, March 2006

Layout/Art Director

- *Journal of Sports Media*, March 2006 to March 2008
- *Our Voice*, the official newsletter of The University of Mississippi Association of Black Journalists, Fall 2005 to Spring 2006
- *The Daily Mississippian*, July 2005 to May 2006 (News Section)

Senior Editor

- *M Magazine*, January 2007
- *MrMagazine.com*, January 2006 to May 2006

Copy Editor

- *The Daily Mississippian*, April 2006 to August 2006

Assistant Editor

- *Samir Husni's Guide To New Magazines*, March 2008
- *Samir Husni's Guide To New Magazines*, March 2007

Writing

- *Co-ed Magazine*, November 2005
  Wrote a feature on the chair of the department of journalism, Samir Husni, the introduction to the college section on Ole Miss, an article about Oxford's social scene and an article about Greek life at Ole Miss. All of the articles appeared in the winter 2005 issue of the magazine.
- *The Ole Miss*, October 2004 to June 2006
  Wrote articles for the student life section, academics section and distinctions section of the yearbook and took photographs.
- *The Daily Mississippian*, July 2004 to August 2009
  Wrote editorials for the opinion section, wrote articles in the arts and life section and took photographs.

Internships

- *MSNBC*, September 2008
  Helped with the network's coverage of the 2008 presidential debate at The University of Mississippi. Tasks included setting up the staging area and the work room; running scripts between the work area and the stage; escorting talent back and forth between press shots; and helping break down all of the gear after the debate was over.
- *Athletics Media Relations Student Worker*, January 2004 to August 2009
  Worked the basketball games for the television crew, passed out stat sheets during the basketball games, put together press books for the basketball games, did the injury reports for the football games, helped set up the home and visiting teams media rooms, took and transcribed quotes from the visiting team and helped set up on game day.

Other Journalism Experience

- *Mississippi Scholastic Press Association Summer Workshop*, June 2006
  Taught people how to use Adobe InDesign and Adobe Photoshop.
- *Mississippi Scholastic Press Association Dow Jones Minority Workshop*, June 2005
  Taught people how to use Adobe InDesign and Adobe Photoshop and designed the template for the workshop's newspaper.
- *Assistant to Dr. Samir Husni, chair of the Department of Journalism*, May 2005 to present
  Wrote articles, helped students in the lab, did layout for the department newsletter, did layout for presentations, wrote scripts for the department web site and helped maintain department

computers. I also helped scan magazine covers and enter data for *Samir Husni's Guide to New Magazines*.

Other Experience

- *Associated Student Body Student Services Committee*, April 2004 to April 2005
  Responsibilities included gathering information to be used in governing decisions, writing reports, and helping come up with new and innovative ways to serve the students.
- *Associated Student Body Communications Committee*, April 2003 to April 2004
  Responsibilities included writing press releases, gathering information to be used in public relations articles in the press book and coming up with public relations fund raising ideas.
- *Warren A. Hood*, June 2001 to July 2001
  Taught classes geared toward helping campers attain the cooking, pioneering and wilderness survival merit badges.

Awards and Honors

- Member of Upsilon Pi Epsilon, an international honor society for the computing and information disciplines (Inducted Spring 2009). The membership requirements can be found at http://upe.acm.org/member_requirements.htm.
- Member of the UM programming team at the Consortium for Computer Sciences in Colleges: Mid-South Conference at Arkansas Tech University. (Spring 2008)
- Member of the UM programming team at the Consortium for Computer Sciences in Colleges: Mid-South Conference at The University of Louisiana Monroe. (Spring 2007)
- Member of the UM programming team at the Association for Computing Machinery International Collegiate Programming Contest's Southeastern United States Regional (Fall 2006)
- Member of Pi Mu Epsilon, a mathematical honor society (Inducted Fall 2006). The membership requirements can be found at http://www.pme-math.org/organization/whatispme.html
- Placed 2nd in Best Newspaper Page Layout Designer Category at the Southeast Journalism Conference (Spring 2006)
- Selected for the University of Mississippi's Who's Who class of 2005-2006.

Community Involvement

- Alumnus of *Sigma Nu Fraternity*
- *Campus Crusade for Christ* August 2002 to present
- Volunteered with *Oxford Veteran's Home* on January 2003
- Inducted into Order of the Arrow in scouts
- Obtained rank of Eagle scout March 2002
- Volunteered with *Mission Mississippi* on December 2001
- Volunteered with *American Red Cross* on August 2001
- Volunteered with *Hudspeth* on March 2001
- Volunteered with *Mississippi Soil and Water Conservation* on March 2001
- Volunteered with *First Baptist Church Jackson West Park Project* on December 2000
- Volunteered with *Stew Pot* from December 1999 to February 2002
- Volunteered with *Baptist Children's Village* from June 1999 to July 1999.

Research

Parallel Compression

- I worked on a program called czip for the Cell Broadband Engine. As part of my work, I took a program that had originally been written by Andy Anderson for the Cell Broadband Engine using Cell s.d.k. 1.0. His program used Mark Adler and Jean-loup Gailly's zlib to compress the files and produced output in a series of multiple chunks that was sufficiently gzip compliant that they could be unzipped using gunzip. Using this program as a base, I converted the code to cell s.d.k. 2.1, eliminated the existing 2 gig file size barrier, made it to where the program output the zipped files name in the header (making the file fully gzip compliant), optimized the program to print only one header and trailer for files less than 2 gig and made it to where it broke compressed files into 2

gigabyte chunks for compressed files larger than 2 gigabytes. My work on this was presented at a Mississippi Academy of Sciences conference in the Spring of 2008.

- I am also working on a parallel version of bgzip2 for the Cell Broadband Engine that will be based on the current parallel version of bgzip2.

Web Publishing

- For my senior project in computer science, I worked on a web publishing platform that allowed people to upload a magazine onto a server for others to view and customize by rearranging the content.

Magazines

- I have performed research covering the economic theory behind magazine business decisions. This work builds on a lot of the research in the field of antitrust.
- I have helped enter data on new magazine launches for *Samir Husni's Guide to New Magazines*, which is an annually published guide on all of the new magazine launches for each year.

Programming Languages

- For my master's project in computer science, I developed a programming language called FASPL - the fast adaptive spatial programming language - to aid researchers in generating XDDL files, which are files used to perform joins in the Granite database system.