DSPJockey

A Programming Language Built for Signal Processing

Vanshil Shah (vs2409), Brian Bourn (bab2177)
Addisu Petros (aep2157), Abhinav Mishra (anm2147)

COMS 4115 Programming Languages and Translators
Prof. Steven Edwards

1. Introduction

1.1 Motivation:

In the modern world, Digital Signal Processing is used in many areas such as telecommunications, DJ/Audio mixing, and even in fields such as finance for algorithmic trading. Within digital signal processing itself, there are many mathematical operations that can be created and defined such as convolution, filtering, time shifting, and more. However, implementing this functionality requires the notion of time, because signals exist over a period be it discrete or continuous. Currently, there aren't many languages that simply allow users to build and manipulate Signals using a notion of a global time.

Therefore, the main idea behind our language is to provide a simple framework that would enable interested parties to write programs that can conveniently manipulate signals. Programmers would be able to create signals with relative ease in an environment that allows for the straightforward representation and manipulation of signals.

Since this language will be able to manipulate signals it opens up a variety of different possibilities. With the ability to modify signals, the language will be able to support writing any DSP function such as Fourier Transforms, basic phase shifting, amplitude/frequency modulation, etc. One possible end user application could be an Electronic music generator. Digital signal processing is even relevant when it comes to the financial stock market where there are many DSP applications used in essential data modeling and market analysis. Therefore, as signal processing is ubiquitous, this language has many applications that can be used to create programs for a myriad of industries.

1.2 Language Features:

- Signal and Array data types for simple Signal creation and manipulation
- Global time so that one can access a signal at a current time and at a previous time by just subtracting from the global time
- Normal C-like functionality such as binary operations, relational operations, variable assignment, global and local scope, functions, basic C data types, etc.
- Summation functionality using sum keyword (very useful for filters and for digital signal processing in general)

1.3 Keywords:

Keyword	**Meaning/Description**
let	used to declare a new signal variable
int	data type that represents an integer
float	data type that represents a floating point number
Signal	keyword used to declare a new signal stream
Array	data type that represents a list of values which all have the same type
print	used to print information to standard out
to	used to specify a range (from a to b) in sum
bool	data type that represents a Boolean value

`return`	return a value
`true, false`	value of a Boolean
`sum`	is the keyword to denote the summation
`if, else`	specify if, else conditional statement
`for`	specify for loop conditional
`while`	specify while loop conditional
`Stream`	used to declare signal stream
`Time`	used to access signal time

1.4 Primitive Data Types:

integer	A 32-bit number which represents only whole real numbers. (default Signed, can be declared as Unsigned)
float	A 32-bit Allows for the representation of numbers with fractional parts.
bool	A single bit data type used for true false statements. 1 for true 0 for false

1.5 Aggregate Data Types:

`Signal`	used to represent Signal data type, all the values are of type float and can be accessed using the time keyword.
`Array`	standard list style array where all elements are of the same type

1.6 Operators:

Operator	**Meaning**
+	Addition
-	Subtraction
*	Multiplication
/	Division
^	Exponential
=	Assign
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to
=	Assignment
==	equals
!=	not equal to

1.7 Control-Flow:

Control-flow is done by using if-else statements when there is only one iteration. When there are multiple iterations for and while loops are used. If-else statements do not necessarily require an else.

2. Language Tutorial

2.1 Environment Settings:
To compile and run DSPJockey programs you must have the g++ compiler, which is used for compiling C++ programs.

2.2 Building the Compiler/Running Programs:
Compilation of the DSPJockey compiler requires Ocaml (version 4.0 and above). The libraries, ocamllex and ocamlyacc are also required. Type make in the build directory to build all the source files.

To run the .dj program, do the following

```
./run_compiler.sh <name of .dj file> [optional <name of output file>]
```

2.3 Simple Hello World Program:
Just like C, DSPJockey requires a main function. DSPJockey also does not support void as a "return type."

Here is a simple hello world program.

```
//hello-world.dj
int main(){
      print("hello world");
}
```

The generator will read the DSPJockey code and output C++ code. Here is the output of the compiler:

```
//main.cpp
#include <iostream>
#include <fstream>
#include <iomanip>
#include <string>
using namespace std;


int main(){
cout << "hello world" << endl;
}
```

Test the program to make sure it prints "hello world."

2.4 Variables and Data Types:

Variables hold values in memory that can be written to or read from. The variable's data type can either be of a primitive data type (String, Boolean, int, float) or an aggregate data type (Array and Signal).

2.4.1 Primitive Data Type Variable Declaration:

```
int x;
x = 33;
float y = 2.33;
String z = "bye";
```

2.4.2 Aggregate Data Types:

The two aggregate data types, Array and Signal are very important to the language as they are used to build and manipulate streams. They are declared in much the same manner, the only difference being that the array's length must be declared while a signal's length is guaranteed to be 1024.

2.4.2.1 Array:

To create and initialize an array the let keyword must be used.

```
let arr =Array[10];
```

To access an array element, use an index that can range from 0 to the size you declared minus 1.

For example if the third element needs to be accessed:

```
float x = arr[2];
```

2.4.2.2 Signal:

Signal is the most important data type in DSPJockey and used to build and manipulate signals.

To create and initialize a signal:

```
let sig = Signal[];
```

Signal access is based on using "time". If the user just uses the keyword time, the latest value in Signal is accessed.

```
float y = sig[time];
```

A value in a signal that is in a previous time can also be accessed by subtracting the discrete time (an integer) from time.

For example, if one wants to access a value two samples before the current time.

```
float z = sig[time-2];
```

The advantage to using signal is that when you perform an operation on a signal it happens over the whole signal. For example:

```
sig[time] = sig[time] +1
```

will increment all the samples in the signal by one.

2.5 Operators:
DSPJockey uses the same operators as C for arithmetic operators, assignment operators, and comparison. To learn more see the Language Reference Manual for more information on our different data operators.

2.6 Comments:
Commenting is identical to C. For multi-line comments begin with /* and then end with */ while for single line comments use the // at the beginning of the line or statement.

2.7 Control Flow:
Control Flow
Our language has the ability to use the same conditional and looping structure as C and java. The if, while and for statements are identical

```
//else is optional
if ( boolean_condition ) {
// some code
}
else {
//more code
}

//while statement
while ( boolean_condition ) {
//some code
}

//for loop
for(initialization; boolean_condition; iteration_step){
//some code
}
```

2.8 Built-in Functions:
The two built-in functions that DSPJockey provides are print and Sum.

2.8.1 Print:
The print function is just used for printing to standard out. It is called in the following manner

```
print(5); //prints 5
print"hello world"; //prints hello world
```

```
int x = 5;
print x; // prints 5
```

2.8.2 Sum:

The sum function is used for performing a summation operation. It takes in the start and end index, which need to be integers and the expression to evaluate it on. Here is how you would use it:

```
sum x = 1 to 3: n+1;
sum would be equal to (1+1) + (2+1) + (3+1) = 9.
```

2.9 Functions:

Function declaration and calling a function is identical to what you would see in C or C++ with the exception that there are two different types of functions, normal functions and stream functions. Normal functions are used to modify primitive types and arrays, stream functions are used to modify signals as well as the other types. Signals can only be created and modified in Stream functions

```
//function declaration
int function(args){
//some code
}

//stream functions
stream x(args){
//some code
}
```

Additionally calling the function is done the same as well

```
int result = function(float a);
```

would call function with argument a, which has type float, and store the return value in result. Note that we have to explicitly state the type of result.

2.4.8 Comprehensive Examples:

To view examples where all of the features of the language are combined, refer to Chapter 6, which is the test section.

3. Language Reference Manual

3.1 Lexical Conventions:
In DSPJockey, there are different kinds of tokens which include identifies, keywords, constants, strings, and comments.

3.1.1 Comments:
DSPJockey allows for single-line and multi-line comments that are similar to C-style commenting. /* introduces the comment and */ ends the comment. The previous notation is mainly used for multi-line commenting while for single-line commenting, // can be used at the beginning of the line to comment out that whole line.

Example:
single line comment:

```
//this is a single line comment
```

multi-line comment:

```
/* this is a
multiline comment */
```

3.1.2 Identifiers:
Identifiers are used to identify variables and functions. Each identifier can contain a combination of digits, letters, and the underscore character, although the identifier must start with a letter. Letters can be lowercase and/or uppercase ASCII characters. Digits are the ASCII characters 0-9. Identifiers in DSPJockey are case sensitive.

3.1.3 Constants:
Constants in DSPJockey mainly refer to literals that can be a boolean, a float, or an int.

Float constants: These contain of an integer part and a decimal point. It is ok to not have the decimal part included but in that case it just makes more logical sense to use int.

Integer constants: simply contain the integer part without any decimal place. If any decimal is found in the int type an error will be thrown.

String constants: In DSPJockey, you can also have string constants that consist of a series of characters delimited by quotation marks.

3.1.4 Whitespace:
Whitespace is represented by tab and blank characters in DSPJockey. It is ignored by the compiler and is mainly used to separate lexical tokens from each other.

3.1.5 Keywords:

Keywords refer to specific identifiers that are used to denote certain types or objects. These **cannot** be overloaded and are reserved for the language only.

Here is the list of keywords in DSPJockey:

Keyword	**Meaning/Description**
`let`	used to declare a new signal variable
`int`	data type that represents an integer
`float`	data type that represents a floating point number
`Signal`	keyword used to declare a new signal stream
`Array`	data type that represents a list of values which all have the same type
`print`	used to print information to standard out
`to`	used to specify a range (from a to b) in sum
`bool`	data type that represents a Boolean value
`return`	return a value
`true, false`	value of a Boolean
`sum`	is the keyword to denote the summation
`if, else`	specify if, else conditional statement
`for`	specify for loop conditional
`while`	specify while loop conditional
`Stream`	used to declare signal stream
`Time`	used to access signal time

3.1.6 Separators:

The only separators besides whitespace and new line for lexical analysis in DSPJockey are the comment (,) which is used for separating arguments in passed into a function and the semicolon (;) which is used for separating statements in a block of code.

The comma character (,) is used to separate tokens in a list or tokens in the arguments to a function.

```
int lowpass_filter(orig_signal, dt, rc)
      { /* code */
}
```

The semicolon (;) character is used to separate statements in a block of code.

```
      statement 1;
```

```
statement 2;
```

3.2 Data Types

3.2.1 Basic Types:

DSPJockey has four basic data types Integer, Float, Boolean, and Array. These data types can be used without reservation or import of an outside library in any part of a DSPJockey Program. These types can also be used to build objects or libraries. Note that integers can be compared with floats and vice versa. Floats and integers can also be compared with Booleans (0 for false and 1 for true).

integer	A 32-bit number which represents only whole real numbers. (default Signed, can be declared as Unsigned)
float	A 32-bit Allows for the representation of numbers with fractional parts.
Boolean	A single bit data type used for true false statements. 1 for true 0 for false
array	A standard list, style array which can be used to collect any of the four previous type of data

For array access, use the square-bracket notation ([ ]). For instance, if we have an array x and we want to access the fourth element, we would do x[3] (because the elements start at 0).

3.2.2 Built-in Types:

DSPJockey has one special data type called signal, which is used to represent an ongoing signal. A signal has many of the same aspects as a standard array, however it differs in that it is constantly updated and only allows access to the previous 1024 samples. Additionally, the current value of the signal is always stored in the array at index time or the current time. Subtracting an integer value from time accesses previous samples. Signals can only contain floats.

Signals can be used to do operations that require the notion of time. The whole point of this language and the implementation of signal are to exploit it as has the notion of time at discrete intervals.

```
let x = Signal [] ;
float s1 = signal[time];
float s2 = signal[time-2];
```

Note that other types can be built using both primitive and built-in data types. In addition, if a function returns a value, its type must be a supported build-in or primitive data type.

3.3 Operators:

DSPJockey supports a variety of operators that can be used for binary operator expressions, variable assignment, summation functionality, and more. The following is a list of the operators.

Operator	**Meaning**
+	addition
-	subtraction
*	multiplication
/	division
^	exponential
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to
=	assignment
==	equals
!=	not equal to

3.4 Expressions:

Expressions can be the following (expr stands for expression).

expr :==

Integer Literal

String Literal

Float Literal

Boolean Literal

Id Assign

Binop Expression

Function call

Parentheses Expression

Summation

Basic Op (Signal or Array Operation)

3.4.1 Integer Literal:

This just represents a single integer. For example, just the number 5 represents an expression.

3.4.2 String Literal:

This just represents a string constant, which as described before is delimited by quotes. For example "Hello world" is an example of a simple string literal.

3.4.3 Float Literal:

This just represents a single float number. A float doesn't necessarily have to contain a decimal point. For example, 5.5 and 5 are both valid examples for valid floating-point numbers.

3.4.4 Boolean Literal:

This just represents a Boolean type, so either true or false.

3.4.5 Id Assign:

The following is the syntax tree for Id Assign.
expr :==
Id assign_opt :==
Id | Id ASSIGN expr

where the ASSIGN keyword represents the assign operator, = , used for assigning variables.

Essentially, this is used for assigning variables where id represents the identifier for the variable and ASSIGN refers to the assignment operator so you have the variable being assigned to some sort of expression.

For example int x = 5*5 is an example in which x is the id and 5*5 is the right hand side expression.

3.4.6 Binary Operation:

Binary operation is one in which an operator separates two expressions, one on the left hand side, the other on the right hand side. The format is essentially expr op expr. The operator is an operator from section 3.4 and it can be used to return a new expression, or to return a Boolean comparing the two expressions.

Here is the syntax tree:
expr :==
b_expr:==

expr PLUS expr

| expr MINUS expr
| expr TIMES expr
| expr DIVIDE expr
| expr EXP expr
| expr EQ expr
| expr NEQ expr
| expr LT expr
| expr LEQ expr
| expr GT expr
| expr GEQ expr

These expressions can be grouped into the following categories: additive operators, multiplicative operators, and relational operators.

3.4.6.1 Additive Operators:
The first two are additive operators include the addition and subtraction (+ and – respectively) operators and they group left to right.

3.4.6.2 Multiplicative Operators:
The next two are multiplicative operators include the multiply and divide (* and / respectively) operators and they group left to right.

Additive and Multiplicative Operators return another expression that is in the form of a literal.

Ex. (5*5) + (5/1) = 30

3.4.6.3 Relational Operators:
These operators group from left to right and return a Boolean type as they are comparing the left hand side expression to the right hand side expression. The operators in order are less than (<), less than or equal to (<=), greater than (>), and greater than or equal to.

3.4.7 Function Call:

expr:==
ID LPAREN actual_opts RPAREN:==
ID LPAREN RPAREN | ID LPAREN actuals_list RPAREN :==
ID LPAREN RPAREN | ID LPAREN expr RPAREN | ID LPAREN
actuals_list COMMA expr RPAREN

Looking at the way function call is defined above, it can be seen that a function is identified by an id. To call the function it must have left parentheses and right parentheses, (). It is optional to put arguments inside the parentheses and if there are multiple arguments, they are separated by a comma.

The arguments themselves are an expression, which was defined and explained above. The copy of the result of each of the expressions is available in the scope of the function block.

The types of the expressions in the argument list must match the types of the implicit parameters of the function.

When the function is called, each of the expressions in the actuals_list is evaluated from left to right. Control of execution is then given to the function specified by the identifier. A function may call itself.

Argument passing is done through pass-by-reference. The result of evaluating the function call is the value returned by the function called, where the type of value corresponds to the return type of the function.

DSPJockey provides a built in print function.

print(expr): any expression can be printed which includes string literals.

3.4.8 Parentheses Expression:
This is simply an expression closed in parentheses. Function calls basically use a more complex version of this.

expr:==
LPAREN expr RPAREN

3.4.9 Basic Operation (Array or Signal)

expr:==
basic_op :==
ID LBRACKET LITERAL RBRACKET assign_opt | ID LBRACKET time_expr RBRACKET assign_opt :==

ID LBRACKET LITERAL RBRACKET |
ID LBRACKET LITERAL RBRACKET ASSIGN expr |
ID LBRACKET TIME RBRACKET |
IDLBRACKET TIME MINUS LITERAL RBRACKET |
ID LBRACKET TIME RBRACKET ASSIGN expr |
ID LBRACKET TIME MINUS LITERAL RBRACKET ASSIGN expr

Although the syntax tree looks a bit complicated, the overall functionality and usage is pretty straightforward. In basic op it can be seen that we have one basic operation without time_expr and one with it. time_expr is used for signal while the one without it is used for array.

basic_op is basically used to assign a value to a specific slot in an array or at a specific time in the signal. For assigning to an array, just use the square-bracket notation (ex. a[3]

= 5). The Signal or Array can only have values of the float type.

For signal assignment, the time must be taken into account. The time is part of the global scope and so one can either assign a value to the signal at is current time (second to last expression) or at a previous time.

So, if one wanted to assign a value to 2 timeslots before the current time one would do a[time-3] = 2.

3.5 Declarations:

3.5.1 Basic Type Variable Declaration:

variable :==

prim ID assign_opt SEMI :==
{{
typ
name
exp
}}
(STRING | INT | FLOAT | BOOL) ID ASSIGN expr SEMI | (STRING | INT | FLOAT | BOOL) ID SEMI

As seen by the above syntax tree, a variable of primitive type is declared by first specifying the type itself, String, int, float, or Boolean. Next the identifier for the variable is specified which is just the name for the variable.

That is followed by two cases. In the first case, the initial value for the variable is set to the result of the expression, which must evaluate to the same type as what was used on the left hand side. This is followed by a semicolon, which ends the statement.

The second case is just used for declaring variables as it is just the name of the variable followed by a semicolon.

Ex. Case 1: int x = 5
Case 2: int x;

3.5.2 Signal and Array Declaration:

create_basic :==
LET ID ASSIGN basic LBRACKET lit_opt RBRACKET :==
LET ID ASSIGN (ARRAY|SIGNAL) LBRACKET RBRACKET
| LET ID ASSIGN (ARRAY|SIGNAL) LBRACKET LITERAL RBRACKET

The syntax for arrays and signals is straightforward. Using the let keyword followed by the identifier for the array or signal and then the assignment operator, state the type (either array or signal).

This is followed by two cases, one in which there is an integer literal in the square brackets and one, which contains nothing in the square brackets.

The literal in the second case is used to specify the size of the array or signal. This initializes the array to the size specified and thus space on the stack is allocated for it.

The first case does not initialize the array or signal and does not allocate any space on the stack for it.

3.5.3 Function Declaration

fdecl :==
    prim ID LPAREN formal_opt RPAREN LBRACE stmt_list RBRACE |
    STREAM ID LPAREN formal_opt RPAREN LBRACE stmt_list RBRACE

where

prim :==
    STRING | INT | FLOAT | BOOL

formal_opt :==
/* nothing */ | formal_list :==
/* nothing */ | formal | formal_list COMMA formal

stmt_list :==
/* nothing */ | stmt_list stmt

The first type of function declaration is used for declaring functions with a return type that corresponds to a primitive data type. After the type is specified it is followed by the function identifier. Next the parentheses contain the formal options, or parameter list, which can either, be nothing or a list of arguments that are each separated by a comma.

A function declaration declares a function that accepts the parameters given by the parameter list. The identifiers for these parameters will be available in the function body. In addition, the parameter list contains primitive, array, and/or signal declarations.

Following the arguments you have the actual function code, which is contain inside the left and right braces. A function will contain a list of statements and hence the stmt_list or statement list, which is recursively defined as a list of statements, is contained between the left brace and right brace. When the function is called it evaluates the given block. A function cannot be modified after its declaration.

3.6 Statements

stmt :==
    expression

return statement

block

if with else

if with no else

variable

for loop

while loop

print

create basic

sum

3.6.1 Expression:

A statement can simply be an expression, which was explained above, followed by a semicolon. An expression can range from a simple literal to a complex binary operation. Note that the value is discarded (unless it's variable assignment) but the side effects of the expression still occur.

3.6.2 Return statement:

stmt :==
    RETURN expr SEMI

Return statements are used in functions to signify the end of the function by returning a value, which has the same data type as specified in the function declaration to the caller of the function. The value in this case is the value of the expression.

3.6.3 Block:

stmt :==
    LBRACE stmt_list RBRACE

A block is used for defining the function body. It contains multiple statements and so when a block is executed, each of the statements in the statement list is executed in order. Scoping is based on blocks. Blocks allow for the grouping of multiple statements.

3.6.4 If/Else Conditional statement:

stmt :==
IF LPAREN expr RPAREN stmt ELSE stmt

The expression inside the parentheses is evaluated and if it is non-zero or true, the first statement is executed. If the expression evaluates to zero or false, the second statement is executed.

3.6.5 If without Else Conditional Statement:

stmt :==
IF LPAREN expr RPAREN stmt %prec NOELSE

This is the same as before except that there is no else statement so if the expression returns 0 or false, nothing else is done for this statement and the program moves to the next statement.

3.6.6 For Statement:

stmt :==

FOR LPAREN expr_opt SEMI expr_opt SEMI expr_opt RPAREN stmt

expr_opt :==
/*nothing */ | expr

The for loop has three main parts. The first part is an expression that represents the starting index. The second expression is used to specify an expression that specifies the number of iterations the for loop should execute (it can either be for a specified number of times or until a condition is met). The last expression is used to determine what to do with the index at the end of each iteration.

The statement inside the for loop is executed until the second expression returns 0 or false.

3.6.7 While statement:

stmt :==
WHILE LPAREN expr RPAREN stmt

This is like the for loop except that the user has to take care of the index (if there is a need for one) inside of the while block. The while loop will keep iterating so far as the expression returns a value greater than or equal to 1 or true.

3.6.8 Print statement:

stmt :==

PRINT expr SEMI

The print statement is used for printing an expression to standard output. It takes an expression and prints the value of that expression to standard out.

3.6.9 Create Basic (Array or Signal):

stmt :==
  create_basic :==
    LET ID ASSIGN basic LBRACKET lit_opt RBACKET :==
      LET ID ASSIGN (ARRAY|SIGNAL) LBRACKET (/*nothing*/ | LITERAL) RBRACKET

This statement is used for the creation of arrays and signals. This was already explained above but here is a repeat of the explanation.

The syntax for arrays and signals is straightforward. Using the let keyword followed by the identifier for the array or signal and then the assignment operator, state the type (either array or signal).

This is followed by two cases, one in which there is an integer literal in the square brackets and one, which contains nothing in the square brackets.

The literal in the second case is used to specify the size of the array or signal. This initializes the array to the size specified and thus space on the stack is allocated for it.

The first case does not initialize the array or signal and does not allocate any space on the stack for it.

3.6.10 Summation

In DSPJockey, one can perform a summation over an expression by defining the beginning index, end index, and the expression itself. It is extremely useful when working with filter operations such as FIR Filters.

stmt:==
SUM ID ASSIGN LITERAL TO LITERAL COLON expr SEMI

It provides super simple functionality to perform a summation. Here is an example:

sum x = 1 to 3: n+1;
sum would be equal to (1+1) + (2+1) + (3+1) = 9.

3.7 Program Structure:

DSPJockey programs exist in a single file. All programs must be written in files with the extension ".dj". Programs must additionally contain a function named main, which is where

the program will begin running. A few basic programs are included in the next two sections.

3.8 Scope:
Variables may be referenced in several different contexts throughout a program, as such DSPJockey allows for both global and local scopes. Variables must be assigned before they are referenced, for instance

```
int x = y+7;
int y = 5;
```

will not work since y is referenced by x before it is assigned.

A global variable is declared at the beginning of a file and can be referenced and updated by any program. For example,

```
/***declaration.dj**/
 int i;


int set_i(){
 i =2;
return i;
}

int   main(){
      set_i();
      i = 3;
}
```

A local variable is declared somewhere in a function or a loop and is therefore available only to the function or loop in which it is declared. For instance,

```
int set_i(){
      int i;
      i=2;
return i;
}

int main(){
     int i;
     i =3;
     set_i();
    print  i;
}
```

set_i will return 2 since it is using the value of i in its local scope while in the main function print i will print 3 since it's using the i in its scope, which is the main global scope.

Essentially DSPJockey uses block scoping and each nested block creates a new scope. Variables of the same id in the new scope supersede the variables of the same id in the parent or global scope.

4. Project Plan

4.1 Planning:

The group used a combination of an iterative and feature-driven software development approach. For each step in the process, starting with the scanner, and ending with code generation and testing, the group discussed the feature requirements for that step and when the deadline was for implementing those features. Progress reports and group meetings were scheduled with Professor Edwards. These were especially important when the group was stuck at some point and didn't know how to move on. Dates weren't fully set from the beginning, mainly because the approach was to finish one part at a time as they built off of each other. However, as soon as one part was finished, the deadline for the next part was set. Meeting times within the group were also set from the beginning to avoid time conflicts.

4.2 Specification Process:

The initial features were planned during the writing of the Language Reference Manual. However, changes to the specification needed to be made consistently, which impeded progress as it required having to go back to the parser again. Some of the additional features had to be scrapped or modified to comply with the time for the project.

4.3 Development Process:

The development of the compiler required to follow a certain order, starting from the scanner, then going to the parser, followed by the abstract syntax tree, the semantic analyzer, and lastly code generation. The iterative approach was that for each step there would be a repeated cycle of feature development and testing in small increments to make sure that all of the features worked correctly and to avoid having to go back and change a lot at once. Each member was not just restricted to working on one part. Coding was often done in pairs to allow for collaborative feedback and multiple insights, especially since developing a compiler in a Ocaml is an extremely new process.

4.4 Testing Process:

As described in the development process, testing was done alongside with the feature development. So after a few features in say the parser were developed, testing would be done to make sure there aren't shift/reduce conflicts. Regression testing was also done for later steps by running the old tests to make sure that all the functionality remained intact.

4.5 Programming Style:

Two main languages were used in the software development cycle. Ocaml (including ocamllex and ocamlyacc) were used for the scanner, parser, ast, and sast. Then for code generation C++ was used. For editing the files, the vim editor was unanimously used, mainly because everyone was used to vim. Testing scripts were written in bash.

4.5.1 Ocaml Programming Style:

- for indentation, four spaces were used
- for pattern matching, the pipe character was not placed for the first case and was only placed for each successive case one space before the next case (i.e. for match statements)

- leave one line of whitespace between each function
- one line of whitespace between each let in a function
- underscore case used for naming functions and variables

4.5.2 C++ Programming Style:

- use vim indentation (gg=G command)
- opening curly brace for conditional should be on the same line as the conditional statement, ending brace should be one line after the last statement in the block of the conditional
- leave whitespace line between each function call
- followed normal ANSI-C programming style

4.5.3 Bash/Script Programming Style:

- hard tab for each statement inside a function
- each statement begins right at the beginning of the line (no space)
- whitespace line between things that had its own blocks such as functions, if/else conditionals, etc.

4.6 Project Timeline:

Date	Milestone
9/24/14	Project Proposal Completed
10/27/14	Language Reference Manual Drafted and Submitted
11/2/14	Scanner completed
11/10/14	Parser Completed
11/25/14	AST completed
12/5/14	SAST and Semantic Analyzer Completed
12/11/14	Code generation completed
12/15/14	Final Project Report Completed

4.7 Team Responsibilities:

Team Member	Responsibilities
Brian Bourn (Project manager)	Scanner, Parser, Code Generation
Vanshil Shah (Language Guru)	Scanner, Parser, AST, Semantic Analyzer
Abhinav Mishra (System Architect)	AST, SAST, Semantic Analyzer
Addisu Petros (Test/Validation)	Semantic Analyzer, Code Generation

4.8 Development Environment:

4.8.1 Programming Environment:
Everybody used the vim editor, as it was the most familiar and easy to use.

4.8.2 Version Control System:
Git was used as the version control system, as once again it was the most familiar. A private, shared repository was stored on **GitHub** for easy, universal access amongst the group members. It also allowed for separation of specific parts through branching, which was useful for reverting back to master in case something got messed up.

4.8.3 Project Management:
Asana was used for project management as it includes a calendar for setting deadlines, tracking of bugs, and communication without the need for email.

4.8.4 Documentation Environment/Storage of Documents:
All the important documents such as code samples, the proposal, and the LRM were stored on **Google Drive** in a shared folder. Initially, for creating and editing new documents, Google Drive was being used but it was leading to significant formatting issues. Hence, there was a transition where documents were being locally written in **Microsoft Word** and then shared via **Microsoft OneDrive** for editing. The final versions were stored in Google Drive.

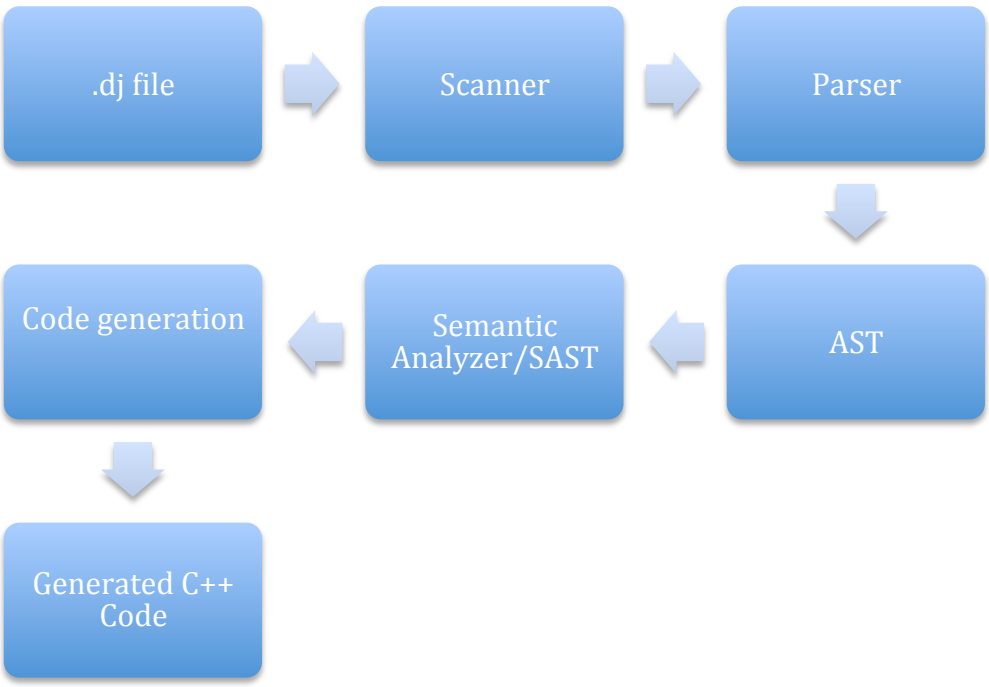
4.9 Project Log:

9/10/14	Team finalized
9/13/14	First team meeting, weekly times decided
9/20/14	Language Defined
9/24/14	Proposal Completed
10/1/14	Specific features of language decided
10/9/14	Expressions in language decided
10/15/14	Statements in language decided
10/24/14	Language Reference Manual drafted
10/26/14	Development environment created
11/2/14	Scanner mainly completed
11/7/14	Tests written, testing phase executed for scanner and parser
11/10/14	Parser completed
11/11/14	Begin working on AST-> expression and statements
11/16/14	Tests written for AST, testing phase for AST begins along with regression testing for Scanner and Parser

11/25/14	AST completed, string of variable, function, etc.
11/30/14	Began SAST and Analyzer (two separate files)
12/2/14	Tests written for Semantic Analyzer, testing phase for Semantic Analyzer begins along with regression testing
12/3/14	Type checking implemented, signal and array functionality added, edits made to parser and ast are made to comply with Semantic Analyzer
12/5/14	Semantic Analyzer completed, include scope checking, variable/function checking, etc., testing for Semantic Analyzer also complete
12/6/14	Code generation begins along with more regression tests
12/9/14	Tests are written for code generation
12/11/14	Signal and Array Code generation complete along with the testing of it
12/16/14	Final Report completed

5. System Architecture

5.1 Architecture Overview:



The components flow in the following order. The Ocaml libraries, ocamllex and ocamlyacc, were used to build the scanner and parser respectively (scanner.mll and parser.mly). The code generator is also written in Ocaml in the file, code_generation.ml. The ocamllex and ocamlyacc libraries automatically generate the interfaces from the scanner to parser and from the parser to the ast.

The semantic analyzer takes in the ast and returns the sast and performs the analysis on the sast using the analyzer. Then the code generator takes the sast and converts that to C++ code. The types are essentially converted from Ast type to Sast type.

The entry point to the semantic analyzer is in infer_prog at the bottom of the file, which takes in the ast and then adds the variable declarations and function declarations to the global environment scope. The ast components are then converted to sast components where more checking such as type checking and scope-checking are done.

The entry point to code generation is located in string_of_prog first concatenates all the variables and functions into one string an then also takes in each component of the sast and converts it to corresponding C++ code.

5.2 Component Interface Interaction:

5.2.1 .dj file (DSPJockey File):
This is simply the file that contains the code written in the DSPJockey Language.

5.2.2 Scanner (scanner.mll):
The scanner is written using ocamllex and essentially takes in the .dj source code and transforms it into a series of lexical tokens for the parser to interpret. The scanner can also detect the presence of invalid characters.

5.2.3 Parser (parser.mly):
The parser is written using ocamlyacc and takes in the tokens generated by the scanner and checks for syntax errors based on the context-free grammars defined in the parser for expression, statement, etc. After processing the tokens, the parser then produces the abstract syntax tree.

5.2.4 AST (ast.ml):
The abstract syntax tree defines the relationships between tokens and represents the syntactic structure of DSPJockey. The ast types such as variable, statement, expression, and primitive data types are defined in the ast.

5.2.5 Semantic Analyzer (sast.ml & analyzer.ml):
The semantic analyzer will take in the abstract syntax tree and go through the nodes of the tree converting the ast types to the annotated sast types so that the types such as expressions and statement are evaluated properly. The semantic analyzer also defines the environment of a scope, starting with the main global scope. Each scope is represented by a symbol table, which contains the variable identifiers and function identifiers. This is used for checking if a variable or function already exists in a local or parent scope (for say assignment expression). The semantic analyzer is extremely important for making sure things such as variable declaration and function calls are valid.

The analyzer file does all the main semantic analysis and converts the ast types to the annotated sast types after analysis. The sast file just simply contains those annotated types.

5.2.6 Code Generation (codegen.ml):
The code generation files takes in the annotated ast (the sast) and converts all those types into C++ code. It is essentially the compiler and represents the last step in the process. The C++ code can then be run to execute the program that was originally in the .dj file.

5.3 Top-level file (dspjockey.ml):
This file basically represents the whole diagram from above as it reads from standard in and then creates the program from the Parser and Scanner, which is then passed into the ast. Then the ast is passed to the sast whose contents are passed to the code generator.

It essentially provides the executable entry point to DSPJockey compiler after compilation and distributes the tasks to the individual compiler modules.

5.4 File Assignments:

Team Member	Responsibilities
Brian Bourn (Project manager)	Scanner, Parser, Code Generation
Vanshil Shah (Language Guru)	Scanner, Parser, AST, Semantic Analyzer
Abhinav Mishra (System Architect)	AST, SAST, Semantic Analyzer
Addisu Petros (Test/Validation)	Semantic Analyzer, Code Generation

6. Test Plan

The big rationale behind our testing implementation was to have a system that would be an integral part of our development process and help us accurately translate our ideas into code. To this end, we designed our testing suite to check new functionality as well as flush out possible edge-case errors.

6.1 Automation:
There was an initial plan to create a script that would run through all the tests but due to time and that the building up of the compiler was already handled in the run_compiler.sh, there was no creation of a test automation script.

6.2 Mechanism:
Based on the development process, the test plan was broadly categorized into three. They are outlined as follows:

6.2.1 Early stage testing:
In the beginning stages of our development process we wanted to insure that our scanner and parser were properly written. To this end we relied on ocamllex and ocamlyacc to insure that we had no shift/reduce conflicts and that our context free grammar was indeed logical.

6.2.2 Middle stage testing:
At this stage, our biggest focus was making sure that our DSPJockey code was being read in effectively and accurately. We had to make sure that the *ast* and *sast* were generating the correct tokens. For this, we added debugging statements in the *ast* and *sast* that would print out the tokens and types generated from our code. An example is presented below.

DSPJockey code to create a ramp signal:

```
stream hello_sig(float val) {

            let sig = Signal[];
            sig[time] = (val=val+1.0);
            print sig[time];
}

int main() {
            hello_sig(0.0);
}
```

Debugging print statement from the ast

```
([],
{ return type = intfname = "main"
   formals = []
Expr (Call hello_sig [(Float lit: 0..)])]}
```

```
{Stream return type = fname = "hello_sig"
   formals = [float val]
Signal sig -1.,
Expr (Signal sig[time - 0] = Id valBinop (Id val) Add (Float lit:
1..)),
Printing: Signal sig[time - 0] = .]})
```

By examining these outputs, we were able to ascertain that the code was properly type checked and tokenized.

6.2.3 Final stage testing:

The primary concern here was to validate the c++ code that was being generated from our code generator. For this, we wrote as many test cases as we could until we were satisfied that we had a properly functioning code generator. In general, we had multiple test cases to validate all the important aspects of our language. These test cases looked at :

- Signal creation
- Signal access/modification
- Signal Printing
- Array creation
- Array access/modification
- Summation statement evaluation
- Function declaration
- Function calling
- Type checking
- Scope checking
- Arithmetic evaluation
- Boolean statements evaluation
- For/while loop statement execution
- If/else statement execution

6.2.3.1 Examples of representative programs:

I) simple program to create a ramp signal (with a buffer size of 1024) and print the values of the signal

DSPJockey code:

```
stream hello_sig(float val) {

            let sig = Signal[];
            sig[time] = (val=val+1.0);
            print sig[time];
}
int main() {
            hello_sig(0.0);
}
```

Generated C++ code

```
#include <iostream>
```

```
#include <fstream>
#include <iomanip>
#include <string>
#include "libcirc/circ_buffer.h"
using namespace std;

void hello_sig(float val)
{
            circular_buffer sig;
            for (int xz=0; xz<1024; xz++) sig.set_value(val = (val +
1.));;
            for(int ew=0; ew<1024; ew++) cout << sig.value_at(0)
<<endl;
}
int main()
{
            hello_sig(0.);
}
```

II) Simple program to simulate an FIR filter

DSPJockey code

```
stream fir_filter() {

            let coef_array= Array[10];
            int x =0;
            while(x<10){
            coef_array[x]=5;
            x=x+1;
            }

            float val=0.0;
            let sig = Signal[];
            sig[time]= (val=val+1.0);

            let output_signal = Signal[];
            let output_signal[time] = Sum i=0 to 10 : coef_array[i] *
sig[time-1];
            print output_signal[time];

}
int main() {
            fir_filter();
}
```

Generated C++ code

```
#include <iostream>
#include <fstream>
#include <iomanip>
```

```
#include <string>
#include "libcirc/circ_buffer.h"
using namespace std;

void fir_filter()
{
            int coef_array[10];
            int x = 0;
            while ((x < 10)){
            coef_array[x]= 5;
            x = (x + 1);
            }

            float val = 0.;
            circular_buffer sig;
            for (int xz=0; xz<1024; xz++) sig.set_value(val = (val +
1.));;
            circular_buffer output_signal;
            for(int ew=0; ew<1024; ew++){
                        int u=0;
                        for(int i=0; i<10; i++){
                                    u+=         (coef_array[i]         *
sig.value_at(1));
                        }
                        output_signal.set_value(u);
            }
            for(int      ew=0;      ew<1024;      ew++)      cout      <<
output_signal.value_at(0) <<endl;

}
int main()
{
            fir_filter();
}
```

7. Lessons Learned

Brian Bourn (Project Manager):
I had fun with this project although I thought it was overall one of the most challenging things I've ever done at Columbia. This project really taught me the value of starting a project early and trying to stick to your project milestones. While our team started early we failed to reach each milestone on time. As Such we ended up with a lot to do at the very end, which leads me to my second lesson learned. The second thing I learned was that it's a great Idea to break up your project in to smaller pieces but it is important to plan the execution of the small chunks so that you can move on from where you are. For instance it's not a good Idea to have to last small chunk of your semantic analyzer to be writing program functionality. Overall this program was really fun to write but I wish I had more time to write it in.

Vanshil Shah (Language Guru):
When we were first starting the project, I was completely overwhelmed. I thought there was no way that 4 people without any experience in functional programming could create a compiler for a digital signal processing language. None of us had ever used Ocaml before taking this course and knew next to nothing about it. However, we progressed through the class and learned bits and pieces about Ocaml and compilers and eventually started with out language implementation. When we first started programming, we had no idea what any of the components were doing but now as we have completed the compiler, I feel very confident in my Ocaml programming abilities. I think the most important thing I have learned from this project is managing complexity and breaking down a big project into smaller, more manageable chunks. Once we started programming in smaller chunks, we found it much easier to see how everything was connected on a larger scale. Writing this compiler also helped me understand Ocaml much better and brought me to the realization that you only begin to understand a language when you write actual code in it. My advice to future teams is to really try to understand the components of the compiler first and then begin by breaking the project into smaller chunks. If you have a clear idea of what chunks the compiler needs, you will have a much easier time writing it. The interconnection on components is also very important and if you try to understand Ocaml on a deep level, it will make your life much easier. That and start earlier!

Abhinav Mishra (System Architect):
Overall the experience of building your own programming language was pretty interesting and enriching. I knew about some of the CS Theory concepts such as DFAs and Context-free grammars and how they were used for building languages but I never understood how that was applied to actually build a programming language. Therefore, in a general sense the main thing I learned was the overall process to build a compiler and how the computer science concepts around languages connect to the process of building a compiler and designing a new language.

The advice I would give to future students in to START EARLY! It's obviously really tough to do with all the other work in other classes but define a strict guideline for how many times per week the group should meet and for how many hours and follow that guideline to make sure you're making continual progress on time. Another thing I would suggest is that

when you're implementing the scanner and parser, think way ahead, in terms of time and overall functionality with the sast and code generation. It can be extremely annoying to go back to the parser and ast again as it impedes the overall progress of the project.

Addisu Petros (Test/Validation):
Personally, I found this project to be a lot of fun and very rewarding. Not only did I get to learn a new language in Ocaml but I also got to learn about how the whole compiler design process works. It was very great to see that the scanner and parser, which form the foundation of the language, can be so easily implemented using ocamllex and ocamlyacc. It was also amazing seeing how the individual parts like the scanner, parser and ast come together to form an entire language implementation.
The one thing I would recommend to other students is to read up on Ocaml a bit before starting the project. Trying to learn Ocaml while completing the project is extremely difficult and painful to do, especially since work builds up as the semester goes on. It's also harder to learn because in previous classes, we only used imperative languages.

8. Appendix

8.1 Ocaml Module Files:

8.1.1 scanner.mll:

```
(*
     Brian Bourn, Vanshil Shah
*)
{ open Parser }

rule token = parse
  [' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace *)
| "/*"     { comment lexbuf }           (* Comments *)
| '('      { LPAREN }
| ')'      { RPAREN }
| '{'      { LBRACE }
| '}'      { RBRACE }
| '['        { LBRACKET }
| ']'        { RBRACKET }
| ';'      { SEMI }
| ','      { COMMA }
| '+'      { PLUS }
| '-'      { MINUS }
| '*'      { TIMES }
| '/'      { DIVIDE }
| '^'      { EXP }
| '='      { ASSIGN }
| ':'        { COLON }
| "=="     { EQ }
| "!="     { NEQ }
| '<'      { LT }
| "<="     { LEQ }
| ">"      { GT }
| ">="     { GEQ }
| "if"     { IF }
| "elseif" { ELSEIF }
| "else"   { ELSE }
| "for"    { FOR }
| "while"  { WHILE }
| "return" { RETURN }
| "print"  { PRINT }
| "int"    { INT }
| "float"  { FLOAT }
| "bool"   { BOOL }
```

```
| "string" { STRING }
| "let"    { LET }
| "to"       { TO }
| "time"   { TIME }
| "stream" { STREAM }
| "Array"  { ARRAY }
| "Signal" { SIGNAL }
| "Sum"      { SUM }
| "true"   { BOOLEAN_LIT(true) }
| "false"  { BOOLEAN_LIT(false) }
| ['0'-'9']+'.'['0'-'9']+ as lxm { F_LIT(float_of_string
lxm) }    (* To recognize floats, can add exponential *)
| ['0'-'9']+ as lxm { LITERAL(int_of_string lxm) }
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm {
ID(lxm) }
| '"' ([^'"']+ as s) '"' { STRING_LIT(s) }
| eof { EOF }
| _ as char { raise (Failure("illegal character " ^
Char.escaped char)) }

and comment = parse
  "*/" { token lexbuf }
| _    { comment lexbuf }
```

8.1.2 parser.mly:

```
%{ open Ast %}

%token SEMI LPAREN RPAREN LBRACE RBRACE COMMA COLON
%token LBRACKET RBRACKET
%token PLUS MINUS TIMES DIVIDE ASSIGN EXP
%token EQ NEQ LT LEQ GT GEQ
%token RETURN IF ELSEIF ELSE FOR WHILE INT FLOAT BOOL
STRING
%token ARRAY
%token SUM TO LET STREAM SIGNAL TIME PRINT
%token <bool> BOOLEAN_LIT
%token TIME
%token <float> F_LIT
%token <int> LITERAL
%token <string> ID
%token <string> STRING_LIT
%token EOF

%nonassoc NOELSE
%nonassoc ELSE
%nonassoc ELSEIF
%right ASSIGN
```

```
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE
%left EXP
%start program
%type <Ast.program> program

%%

program:
   /* nothing */ { [], [] }
 | program vdecl { ($2 :: fst $1), snd $1 }
 | program fdecl { fst $1, ($2 :: snd $1) }

fdecl:
     prim ID LPAREN formals_opt RPAREN LBRACE stmt_list
RBRACE
     { { is_stream = false;
         fname = $2;
           ret_type = Some($1);
           formals = $4;
           body = List.rev $7 } }

   | STREAM ID LPAREN formals_opt RPAREN LBRACE stmt_list
RBRACE
     { { is_stream = true;
         fname = $2;
           ret_type = None;
         formals = $4;
         body = List.rev $7; } }

formals_opt:
    /* nothing */ { [] }
  | formal_list   { List.rev $1 }

formal_list:
    formal                { [$1] }
  | formal_list COMMA formal { $3 :: $1 }

formal:
     prim ID
     { {
          form_type = $1;
          form_name = $2;
     } }

variable:
```

```
      prim ID assign_opt SEMI
      { {
           typ  = $1;
           name = $2;
           exp  = $3;
      } }

vdecl:
     variable  { Variable_Dec($1) }

stmt_list:
    /* nothing */  { [] }
  | stmt_list stmt { $2 :: $1 }

stmt:
    expr SEMI { Expr($1) }
  | RETURN expr SEMI { Return($2) }
  | LBRACE stmt_list RBRACE { Block(List.rev $2) }
  | IF LPAREN expr RPAREN stmt %prec NOELSE { If($3, $5,
Block([])) }
  | variable   { Prim_Assign($1) }
  | IF LPAREN expr RPAREN stmt ELSE stmt    { If($3, $5,
$7) }
  | FOR LPAREN expr_opt SEMI expr_opt SEMI expr_opt RPAREN
stmt
     { For($3, $5, $7, $9) }
  | WHILE LPAREN expr RPAREN stmt { While($3, $5) }
  | PRINT expr SEMI { Print($2) }
  | prim ID ASSIGN SUM ID ASSIGN LITERAL TO LITERAL COLON
expr SEMI      { Sum($2, $5, $7, $9, $11) }
  | LET ID LBRACKET TIME RBRACKET ASSIGN SUM ID ASSIGN
LITERAL TO LITERAL COLON expr SEMI      { Basic_Sum($2, $8,
$10, $12, $14) }
  | create_basic SEMI    { $1 }

create_basic:
  LET ID ASSIGN basic LBRACKET lit_opt RBRACKET {
Basic_Dec($2, $4, $6) }

lit_opt:
     /* nothing */  { -1 }
  | LITERAL             { $1 }

expr_opt:
    /* nothing */ { Noexpr }
  | expr          { $1 }

expr:
```

```
    LITERAL
     { Literal($1) }
  | STRING_LIT
     { String_literal($1) }
  | F_LIT
     { Float_literal($1) }
  | BOOLEAN_LIT
          { Bool_literal($1) }
  | ID assign_opt
     { Id($1, $2) }
  | b_expr
     { $1 }
  | ID LPAREN actuals_opt RPAREN
     { Call($1, $3) }
  | LPAREN expr RPAREN
     { $2 }
  | basic_op
     { $1 }

b_expr:
    expr PLUS   expr
     { Binop($1, Add,   $3) }
  | expr MINUS  expr
     { Binop($1, Sub,   $3) }
  | expr TIMES  expr
     { Binop($1, Mult,  $3) }
  | expr DIVIDE expr
     { Binop($1, Div,   $3) }
  | expr EXP    expr
     { Binop($1, Exp,   $3) }
  | expr EQ     expr
     { Binop($1, Equal, $3) }
  | expr NEQ    expr
     { Binop($1, Neq,   $3) }
  | expr LT     expr
     { Binop($1, Less,  $3) }
  | expr LEQ    expr
     { Binop($1, Leq,   $3) }
  | expr GT     expr
     { Binop($1, Greater,  $3) }
  | expr GEQ    expr
     { Binop($1, Geq,   $3) }

basic_op:
     ID LBRACKET LITERAL RBRACKET assign_opt           {
Basic_Op(Array, $1, $3, "!Noid", $5) }
  | ID LBRACKET ID RBRACKET assign_opt       {
Basic_Op(Array, $1, -1, $3, $5) }
```

```
  | ID LBRACKET time_expr RBRACKET assign_opt {
Basic_Op(Signal, $1, $3, "!Noid", $5) }

time_expr:
     TIME                                { 0 }
  | TIME MINUS LITERAL                   { $3 }

assign_opt:
     /* nothing */ { Noexpr }
  | ASSIGN expr       { $2 }

prim:
    STRING  { String }
  | INT    { Int }
  | FLOAT { Float }
  | BOOL   { Bool }

basic:
     ARRAY     { Array }
  | SIGNAL     { Signal }

actuals_opt:
    /* nothing */ { [] }
  | actuals_list  { List.rev $1 }

actuals_list:
    expr                    { [$1] }
  | actuals_list COMMA expr { $3 :: $1 }
```

8.1.3 Ast.ml:

```
type op = Add | Sub | Mult | Div | Exp | Equal | Neq | Less
| Leq | Greater | Geq

type prim = Int | Float | Bool | String

type basic = Array | Signal

type expr =
    Literal of int
  | Id of string * expr
  | String_literal of string
  | Float_literal of float
  | Bool_literal of bool
  | Binop of expr * op * expr
  | Call of string * expr list
  | Basic_Op of basic * string * int * string *  expr
```

```
  | Noexpr

type variable = {
    typ         :    prim;
     name :    string;
    exp         :    expr;
}

type vdecl =
     Variable_Dec of variable

type stmt =
    Block of stmt list
  | Expr of expr
  | Return of expr
  | If of expr * stmt * stmt
  | For of expr * expr * expr * stmt
  | Prim_Assign of variable
  | While of expr * stmt
  | Sum of string * string * int * int * expr
  | Basic_Dec of string * basic * int
  | Basic_Sum of string * string * int * int * expr
  | Print of expr

type formal =
{
     form_type : prim;
     form_name : string;
}

type func_decl =
{
    is_stream  : bool;
    fname           : string;
    ret_type   : prim option;
    formals    : formal list;
    body       : stmt list;
}

type program = vdecl list * func_decl list

(* Low-level AST printing, to help debug the structure.
These functions are
   only for debugging (the -r flag) and can be removed. *)

let rec expr_s = function
   Literal(l) -> "Literal " ^ string_of_int l
 | Id(s, e) -> "Id " ^ s ^ expr_s e
```

```
 | Binop(e1, o, e2) -> "Binop (" ^ expr_s e1 ^ ") " ^
       (match o with Add -> "Add" | Sub -> "Sub" | Mult ->
"Mult" |
             Div -> "Div" | Exp -> "Exp" | Equal -> "Equal"
| Neq -> "Neq" |
                    Less -> "Less" | Leq -> "Leq" |
Greater -> "Greater" |
                    Geq -> "Geq") ^ " (" ^ expr_s e2 ^ ")"
 | String_literal(s) -> "String Lit: " ^ s ^ "."
 | Float_literal(f) -> "Float lit: " ^ string_of_float f ^
"."
 | Bool_literal(b) -> "Bool lit: " ^ string_of_bool b ^ "."
 | Call(f, es) -> "Call " ^ f ^ " [" ^
       String.concat ", " (List.map (fun e -> "(" ^ expr_s
e ^ ")") es) ^ "]"
 | Basic_Op(b, s, i, a, e) ->
               (match b with Signal -> "Signal" | Array ->
"Array") ^ " " ^
               s ^ (match b with Signal -> "[time - " |
Array -> "[")
               ^ string_of_int i ^ "] = " ^ expr_s e
 | Noexpr -> ""

let prim_s = function
    Int -> "int"
  | Float -> "float"
  | String -> "string"
  | Bool -> "bool"

let string_of_prim = function
    Int -> "int"
  | Float -> "float"
  | String -> "std::string"
  | Bool -> "bool"

let string_of_type = function
    Some(x) -> string_of_prim x
  | _ -> ""

let string_of_basic = function
    Array -> "Array"
  | Signal -> "Signal"

let string_of_var var =
    string_of_prim var.typ ^ " " ^ var.name ^ " " ^ expr_s
var.exp

let string_of_vdecl = function
```

```
     Variable_Dec(v) -> string_of_var v

let rec stmt_s = function
   Block(ss) -> "Block [" ^ String.concat ",\n"
                             (List.map (fun s -> "(" ^
stmt_s s ^ ")") ss) ^ "]"
  | Expr(e) -> "Expr (" ^ expr_s e ^ ")"
  | Return(e) -> "Return (" ^ expr_s e ^ ")"
  | If(e, s1, s2) -> "If (" ^ expr_s e ^ ") (" ^ stmt_s s1 ^
") (" ^
                                                  stmt_s s2 ^
")"
  | For(e1, e2, e3, s) -> "For (" ^ expr_s e1 ^ ") (" ^
expr_s e2 ^
                            ") (" ^ expr_s e3 ^ ") (" ^
stmt_s s ^ ")"
  | While(e, s) -> "While (" ^ expr_s e ^ ") (" ^ stmt_s s ^
")"
  | Print(s) -> "Printing: " ^ expr_s s ^ "."
  | Basic_Dec(s, b, i) -> (match b with Signal -> "Signal" |
Array -> "Array") ^ " " ^ s ^ " " ^ string_of_int i ^ "."
  | Prim_Assign(v) -> string_of_var v

let string_of_formal forml =
     string_of_prim forml.form_type ^ " " ^ forml.form_name

let func_decl_s f =
  "{"
  ^ (match f.is_stream with true -> "Stream" | false -> "")
^
  " return type = " ^ string_of_type f.ret_type ^ "fname =
\"" ^ f.fname ^ "\"\n   formals = [" ^
  String.concat ", " (List.map string_of_formal f.formals)
^ "]\n" ^
  String.concat ",\n" (List.map stmt_s f.body) ^
  "]}\n"


let program_s (vars, funcs) = "([" ^ String.concat ", "
(List.map string_of_vdecl vars) ^ "],\n" ^
  String.concat "\n" (List.map func_decl_s funcs) ^ ")"

(* "Pretty printed" version of the AST, meant to generate a
MicroC program
   from the AST.  These functions are only for pretty-
printing (the -a flag)
   the AST and can be removed. *)
let rec string_of_expr = function
```

```
    Literal(l) -> string_of_int l
  | Id(s, e) -> s ^ string_of_expr e
  | Binop(e1, o, e2) ->
      string_of_expr e1 ^ " " ^
      (match o with
     Add -> "+" | Sub -> "-" | Mult -> "*" | Div -> "/"
      | Exp -> "^" | Equal -> "==" | Neq -> "!="
      | Less -> "<" | Leq -> "<=" | Greater -> ">" | Geq ->
">=") ^ " " ^
      string_of_expr e2
  | String_literal(s) -> "String Lit: " ^ s ^ "."
  | Float_literal(f) -> "Float lit: " ^ string_of_float f ^
"."
  | Bool_literal(b) -> "Bool lit: " ^ string_of_bool b ^
"."
  | Call(f, el) ->
      f ^ "(" ^ String.concat ", " (List.map string_of_expr
el) ^ ")"
  | Basic_Op(b, s, i, a, e) ->
                (match b with Signal -> "Signal" | Array ->
"Array") ^ " " ^
                s ^ (match b with Signal -> "[time - " |
Array -> "[")
                ^ string_of_int i ^ "] = " ^ string_of_expr
e
  | Noexpr -> ""

let rec string_of_stmt = function
    Block(stmts) ->
      "{\n" ^ String.concat "" (List.map string_of_stmt
stmts) ^ "}\n"
  | Expr(expr) -> string_of_expr expr ^ ";\n";
  | Return(expr) -> "return " ^ string_of_expr expr ^
";\n";
  | If(e, s, Block([])) -> "if (" ^ string_of_expr e ^
")\n" ^ string_of_stmt s
  | If(e, s1, s2) ->  "if (" ^ string_of_expr e ^ ")\n" ^
      string_of_stmt s1 ^ "else\n" ^ string_of_stmt s2
  | For(e1, e2, e3, s) ->
      "for (" ^ string_of_expr e1  ^ " ; " ^ string_of_expr
e2 ^ " ; " ^
      string_of_expr e3  ^ ") " ^ string_of_stmt s
  | While(e, s) -> "while (" ^ string_of_expr e ^ ") " ^
string_of_stmt s
  | Print(s) -> "Printing: " ^ string_of_expr s ^ "."
  | Basic_Dec(s, b, i) -> (match b with Signal -> "Signal"
| Array -> "Array") ^ " " ^ s ^ " " ^ string_of_int i ^ "."
  | Prim_Assign(v) -> string_of_var v
```

```
let string_of_fdecl fdecl =
  (match fdecl.is_stream with true -> "Stream" | false ->
"")^ " type: "
  ^ string_of_type fdecl.ret_type ^ " "
  ^fdecl.fname ^ "(" ^ String.concat ", " (List.map
string_of_formal fdecl.formals) ^ ")\n{\n" ^
  String.concat "" (List.map string_of_stmt fdecl.body) ^
  "}\n"

let string_of_program (vars, funcs) =
  String.concat "" (List.map string_of_vdecl vars) ^ "\n" ^
  String.concat "\n" (List.map string_of_fdecl funcs)
```

8.1.4 sast.ml:

```
(*
     Abhinav Mishra, Vanshil Shah
*)
type t =
    Var
  | Func
  | String
  | Int
  | Float
  | Bool
  | Expression
  | Array (* type, int *)
  | Signal     (* signal of floats *)
  | No

type i_or_s =
     Int_ of int
  | String_ of string

type a_expr =
     Literal of t * int
  | String_literal of t * string
  | Bool_literal of t * bool
  | Float_literal of t * float
  | Binop of t * a_expr * Ast.op * a_expr
  | Id of t * string * a_expr * t
  | Call of t * string * a_expr list
  | Basic_Op of t * string * i_or_s * a_expr * t
  | Noexpr

type a_var = {
     a_typ          : t;
     a_name    : string;
```

```
     a_exp          : a_expr;
}

type a_stream = {
     str_typ        : t;
     str_name  : string;
}

type a_variable =
     Primitive of a_var
   | Basic of a_stream

type a_stmt =
    Block of a_stmt list
   | Expr of t * a_expr
   | If of a_expr * a_stmt * a_stmt
   | For of a_expr * a_expr * a_expr * a_stmt
   |  Prim_Assign of t * a_variable
   | While of a_expr * a_stmt
   | Print of t * a_expr
   | Return of a_expr
   | Basic_Sum of t * string * string * int * int * a_expr
   | Sum of t * string * string * int * int * a_expr
   | Basic_Dec of t * string * int  (* When creating an
Array or a signal, we just need the type, the name, and the
size *)

type a_vdecl =
     Variable_Dec of t * a_variable

type a_fdecl =
{
     a_is_stream    :    bool;
     a_fname        :    string;
     a_ret_type     :    t option;
     a_formals :    Ast.formal list;
     a_locals  :    a_variable list;
     a_old_body     :    Ast.stmt list;
     a_body         :    a_stmt    list;
}

type a_program = a_vdecl list  * a_fdecl list
```

8.1.5 analyzer.ml:

```
(*
     Abhinav Mishra, Addisu Petros, Vanshil Shah
*)

open Ast
open Sast

type symbol_table = {
                parent    : symbol_table option;
      mutable functions : a_fdecl list;
     mutable variables : a_variable list;
}

type translation_environment = {
     mutable func_return_type : t option;
                          scope : symbol_table;
}


let type_of_expr (ae : Sast.a_expr) : Sast.t =
     match ae with
      | Sast.Literal(t, _) -> t
      | Sast.String_literal(t, _) -> t
      | Sast.Bool_literal(t, _) -> t
      | Sast.Float_literal(t, _) -> t
      | Sast.Binop(t, _, _, _) -> t
      | Sast.Id(_, _, _, t) -> t
      | Sast.Call(t, _, _) -> t
      | Sast.Basic_Op(_, _, _, _, t) -> t
      | Sast.Noexpr -> Sast.No

let formal_type_conversion = function
     Ast.Int -> Sast.Int
   | Ast.Bool -> Sast.Bool
   | Ast.String -> Sast.String
   | Ast.Float -> Sast.Float

let basic_type_conversion = function
     Ast.Signal -> Sast.Signal
   | Ast.Array  -> Sast.Array

let f_type_conv = function
     Some(x) -> formal_type_conversion x
   | _ -> Sast.No

let find_var name var = match var with
```

```
      Primitive({a_name=n; _}) -> n=name
    | Basic({str_name=n; _}) -> n=name

let rec find_variable (scope : symbol_table) name =
    try
          List.find (find_var name) scope.variables
    with Not_found ->
          match scope.parent with
            Some(parent) -> find_variable parent name
          | _ -> raise Not_found

let find_function (scope : symbol_table) name =
     let rec get_global_scope scope = match scope.parent
with
          | None -> scope
          | Some(parent) -> (get_global_scope parent)
     in
     let g_scope = get_global_scope scope in
     try
          List.find (fun {a_fname=s; _} ->
                         s = name) g_scope.functions
     with
     Not_found ->
          raise Not_found

let add_variable (scope : symbol_table) var =
     scope.variables <- var :: scope.variables;
     var

let string_of_binop = function
     Add -> "Add"
   | Sub -> "Sub"
   | Mult -> "Mult"
   | Div -> "Div"
   | Exp -> "Exp"
   | Equal -> "Equal"
   | Neq -> "Neq"
   | Less -> "Less"
   | Leq -> "Leq"
   | Greater -> "Greater"
   | Geq -> "Geq"

let types_equal t1 t2 =
     match t1,t2 with _, _ -> if (t1 = t2) then true else
false

(*let functioncall env call = *)
```

```
let rec expr env = function
   Ast.Literal(v) -> Sast.Literal(Sast.Int, v)
 | Ast.String_literal(v) ->
Sast.String_literal(Sast.String, v)
 | Ast.Bool_literal(v) -> Sast.Bool_literal(Sast.Bool, v)
 | Ast.Float_literal(v) -> Sast.Float_literal(Sast.Float,
v)
 | Ast.Binop (e1,o,e2) ->
   (
     let e1 = expr env e1
    and e2 = expr env e2 in

     let t1 = type_of_expr e1 in
     let t2 = type_of_expr e2 in

     match o with
          (Add | Sub | Mult | Div | Exp) ->
               if types_equal t1 t2
                    then Sast.Binop(t1, e1, o, e2)
               else
                    raise(Failure("Binop " ^
(string_of_binop o) ^ " have invalid operands."))

       | (Equal | Neq | Less | Leq | Greater | Geq) ->
               if (types_equal t1 t2)
                    then Sast.Binop(Sast.Bool, e1, o, e2)
               else
                    raise(Failure("Binop " ^
(string_of_binop o) ^ " have invalid operands."))
     )
  | Ast.Id(vname, exp) -> (
     let vdecl = try
          find_variable env.scope vname;
     with Not_found ->
          print_endline vname;
          raise(Failure("Variable not found"))
     in

     let ex = expr env exp in
     let ex_type = type_of_expr ex in

     let var = match vdecl with
          Primitive(x) -> x
       | Basic(_) -> raise(Failure("Referring to signal"))
     in

     if var.a_typ <> ex_type && ex_type <> Sast.No
     then
```

```
          raise(Failure("Assign types don't match"))
     else
          print_endline "";

     match var.a_typ with
          Sast.Int ->
               Sast.Id(Sast.Var, vname, ex, Sast.Int)
       | Sast.Float ->
               Sast.Id(Sast.Var, vname, ex, Sast.Float)
       | Sast.Bool ->
               Sast.Id(Sast.Var, vname, ex, Sast.Bool)
       | Sast.String ->
               Sast.Id(Sast.Var, vname, ex, Sast.String)
       | _ ->
               raise(Failure("Incorrect variable type"))

     )
  | Ast.Call(f_name, exprs) ->
     let func_found = find_function env.scope f_name in
     let t_ret_type = func_found.a_ret_type in
     let formal_types = List.map (fun {form_type=s; _} ->
formal_type_conversion s) func_found.a_formals in
     let checked_exprs = List.map (expr env) exprs in
     let checked_types = List.map type_of_expr
checked_exprs in

     let rec typesMatch list1 list2 = match list1,list2
with
          | [], [] -> true
          | [], _ -> raise(Failure(" found parameters when
expecting none "))
          | _ , [] -> raise(Failure(" found no parameters
when expecting parameters"))
          | _ , _ ->
               try
                    ( types_equal (List.hd(list1))
(List.hd(list2)) ) &&
                         typesMatch (List.tl(list1))
(List.tl(list2))
               with Failure("hd") ->
                    raise(Failure(" trying List.hd on [] in
exprCheck:Ast.Call"))
     in

     let types_same = typesMatch formal_types checked_types
in

     if types_same
```

```
      then
           match t_ret_type with
                Some(ret_val) ->
                     Sast.Call(ret_val, f_name,
checked_exprs)
              | None ->
                     Sast.Call(Sast.No, f_name,
checked_exprs)
      else
           raise(Failure("Arg types not same"))

   | Ast.Basic_Op(b_type, name, lit, id, exp) ->
      (* If we're not in a stream function we can't do this
kind of op *)
      if env.func_return_type <> None
      then
           raise(Failure("Can't do signal/array ops in
normal functions"))
      else
           (* Lets find the signal first *)
           let vdecl = try
                find_variable env.scope name
           with
                Not_found ->
                     raise(Failure("Array/Signal not
found"))
           in

           let a_type = basic_type_conversion b_type in

           let var = match vdecl with
                Primitive(_) -> raise(Failure("Basic
operation on wrong type of variable"))
              | Basic(x) ->
                if x.str_typ <> a_type
                then
                     raise(Failure("Type of signal/array
incorrect"))
                else
                     x
           in
           ignore (var);
           (* Evaluate the expression *)
           let a_exp = expr env exp in

           if id <> "!Noid"
           then
                let id_found = try
```

```
                    ignore (find_variable env.scope id);
               with
                    Not_found ->
                         raise(Failure("ID doesn't exist"))
               in
               ignore (id_found);

          (* The literal thats in Basic_Op is going to be
               dependent on the a_type

               if a_type is a signal, then lit=0 means we
just have t
                                                  lit>0 means t-
lit
               if a_type is an array, then lit means
Array[lit]
          *)
               Sast.Basic_Op(a_type, name, String_(id),
a_exp, Sast.Float)
          else
               Sast.Basic_Op(a_type, name, Int_(lit),
a_exp, Sast.Float)

  | Ast.Noexpr ->
    Sast.Noexpr

let rec stmt env = function
     Ast.Block(st_list) ->
          let sl = List.map (fun s -> stmt env s) st_list
in
          Sast.Block(sl)
  | Ast.Sum(var, lit, i1, i2, ex) ->
     (*                                          *)
     let vdecl = try
         ignore (find_variable env.scope var);
          raise(Failure("Identifier already exists"))
     with
          Not_found ->  (
               let new_vdecl =
Sast.Primitive({a_typ=(Sast.Int); a_name=var;
a_exp=Sast.Noexpr;}) in
               ignore (add_variable env.scope new_vdecl);
          )
     in
     vdecl;

     let new_vdecl = Sast.Primitive({a_typ=(Sast.Int);
a_name=lit; a_exp=Sast.Noexpr;}) in
```

```
      ignore (add_variable env.scope new_vdecl);

      let exp = expr env ex in

      let t1 = type_of_expr exp in

      Sast.Sum(t1, var, lit, i1, i2, exp)

  | Ast.Basic_Sum(s1, s2, i1, i2, exp) ->
      let vdecl = try
           find_variable env.scope s1
      with
           Not_found ->
                raise(Failure("Signal doesn't exit"))
      in

      let var = match vdecl with
           Primitive(x) -> raise(Failure("Incorrect variable
type"))
         | Basic(x) -> x
      in

      (* Var is a variable of type a_stream *)
      if var.str_typ <> Sast.Signal
      then
           raise(Failure("Array type doesn't belong here"));

      let new_vdecl = Sast.Primitive({a_typ=(Sast.Float);
a_name=s2; a_exp=Sast.Noexpr;}) in
      ignore (add_variable env.scope new_vdecl);

      let ex = expr env exp in

      let t1 = type_of_expr ex in

      Sast.Basic_Sum(t1, var.str_name, s2, i1, i2, ex)
  | Ast.Return(ex) ->
           (match env.func_return_type with
           | None    -> raise(Failure("Invalid return
statement")))
           | Some(x) ->
               let exp = expr env ex in
               let typ = type_of_expr exp in
               if typ <> x
               then
                    raise(Failure("Invalid return type"))
               else
                    Sast.Return(exp)
```

```
          )

  | Ast.Expr(exp) ->
     let ex =  expr env exp in

     Sast.Expr(Sast.Expression, ex)

  | Ast.If(exp, st1, st2) ->
     let exp1 = expr env exp in
     let typ = type_of_expr exp1 in

     if typ <> Sast.Bool
     then
          raise(Failure("If argument invalid"))
     else
          Sast.If(exp1, stmt env st1, stmt env st2)

  | Ast.For(e1, e2, e3, st) ->
     let exp1 = expr env e1 in
     let exp2 = expr env e2 in
     let exp3 = expr env e3 in

     let typ2 = type_of_expr exp2 in

     if typ2 <> Sast.Bool
     then
          raise(Failure("For loop incorrect"))
     else
          Sast.For(exp1, exp2, exp3, stmt env st)

  | Ast.While(exp, st) ->
     let ex = expr env exp in

     let typ = type_of_expr ex in

     if typ <> Sast.Bool
     then
          raise(Failure("While condition requires boolean
expression"))
     else
          Sast.While(ex, stmt env st)

  |  Ast.Prim_Assign(var) ->
     (* 1. Evaluate the type of rhs expression, make sure
its the same as prim_type
          2. Make sure the variable name doesnt exist in
this scope
     *)
```

```
      let vdecl = try
          ignore (find_variable env.scope var.name);
           raise(Failure("Identifier already exists"))
      with
      Not_found ->  (
           (*                                        *)
           let typ = formal_type_conversion var.typ in
           let a_expr = expr env var.exp in
           let type_expr = type_of_expr a_expr in

           if type_expr <> No && typ <> type_expr
           then
                raise(Failure("Type of ID doesn't match
expression"))
           else (
                let new_vdecl = Sast.Primitive({a_typ=(typ);
a_name=var.name; a_exp=a_expr;}) in
                ignore (add_variable env.scope new_vdecl);
                Sast.Prim_Assign(Sast.Var, new_vdecl)
           )
      ) in
      vdecl

   | Ast.Basic_Dec(name, b_type, l_opt) ->
      (* If this is not a stream function then we can't work
with signals*)
      if env.func_return_type <> None
      then
           raise(Failure("Can't declare basic types in
normal functions"))
      else
           let vdecl = try
                ignore (find_variable env.scope name);
                raise(Failure("Variable already exists"))
           with
                Not_found ->
                     (* Create the signal *)
                     let typ = basic_type_conversion b_type
in

                     if typ = Sast.Array && l_opt = -1
                     then
                          raise(Failure("Need size for
array"));

                     let new_vdecl =
Sast.Basic({str_typ=typ; str_name=name;}) in
                     ignore (add_variable env.scope
```

```
new_vdecl);
                    Sast.Basic_Dec(typ, name, l_opt)
          in
          vdecl
  | Ast.Print(exp) ->
     let exp1 = expr env exp in
     let typ1 = type_of_expr exp1 in

     Sast.Print(typ1, exp1)

let check_vdecls env = function
     Ast.Variable_Dec(var) ->
          let vdecl = try
               ignore (find_variable env.scope var.name);
               raise(Failure("Variable already exists"))
          with
            | Not_found ->
               (*                                        *)
               let typ = formal_type_conversion var.typ in
               let a_expr = expr env var.exp in
               let type_expr = type_of_expr a_expr in

               if type_expr <> No && typ <> type_expr
               then
                    raise(Failure("Type of ID doesn't match
expression"))
               else (
                    let new_vdecl = Primitive({a_typ=(typ);
a_name=var.name; a_exp=a_expr;}) in
                    ignore (add_variable env.scope
new_vdecl);
                    Sast.Variable_Dec(Sast.Var, new_vdecl)
               )
          in
          vdecl

let create_new_env parent_env ret_type =
     let new_scope = {
          parent= Some parent_env.scope;
          functions=[];
          variables=[];
     } in
     let new_env = {
          func_return_type = ret_type;
          scope = new_scope;
     } in
     new_env
```

```
let get_global_environment =
     let new_scope = {
          parent=None;
          functions=[];
          variables=[];
     } in
     let new_env = {
          func_return_type=None;
          scope=new_scope;
     } in
     new_env

let variable_from_formal formal =
     let new_var = Primitive({a_typ =
formal_type_conversion formal.form_type;
a_name=formal.form_name; a_exp=Sast.Noexpr})  in
     new_var

let add_function globe_env fn =
     let func_exists = try
          ignore (find_function globe_env.scope fn.fname);
          raise(Failure("Function already exists"))
     with
          Not_found ->
               (* Convert the return type from
                    Ast.prim option -> Sast.t option
               *)
               let new_ret_type = f_type_conv fn.ret_type
in

               (* Create temporary body to fill in our
function *)
               let new_body = [] in
               let new_function =
{a_is_stream=fn.is_stream;
                                        a_fname=fn.fname;
                                        a_ret_type=(match
fn.is_stream with

     false -> Some new_ret_type
                                                    | true
-> None);

     a_formals=fn.formals;
                                        a_locals=[];
                                        a_old_body=fn.body;
                                        a_body=new_body;}
in
```

```
                (* Add the function to the global scope *)
                globe_env.scope.functions <-
new_function::globe_env.scope.functions;

     in
     func_exists

let string_of_opt = function
     Some(_) -> print_endline "Has a value"
   | None -> print_endline "Is stream"

let string_of_variables = function
     Primitive(x) -> print_endline ("Primitive " ^
x.a_name)
   | Basic(x) -> print_endline ("Stream " ^ x.str_name)

let check_function_bodies globe_env fn =
     print_endline ("==============Checking function: " ^
fn.a_fname ^ "==============");

     (* Create a new environment *)
     let f_env = create_new_env globe_env fn.a_ret_type in

     (* Convert all of the formals to variables so that we
can
          add them to the scope of the current function
     *)
     let vars_from_formals = List.map variable_from_formal
fn.a_formals in
     ignore (List.map (add_variable f_env.scope)
vars_from_formals);

     (* Create new body thats typechecked *)
     let new_body = List.map (stmt f_env) fn.a_old_body in

     let new_function = {a_fname = fn.a_fname;
                         a_is_stream = fn.a_is_stream;
                         a_ret_type = fn.a_ret_type;
                         a_formals = fn.a_formals;
                         a_locals =
f_env.scope.variables;
                         a_old_body = [];
                         a_body = new_body;
     } in
     print_endline "***Function Locals***";
     ignore (List.map string_of_variables
new_function.a_locals);
```

```
     print_endline "==============Finished checking
function==============";
     new_function

let string_of_functions fn =
     print_endline fn.a_fname

let string_of_scope scope =
     ignore (List.map string_of_variables scope.variables);
     List.map string_of_functions scope.functions

let print_env env =
     string_of_scope env.scope

let infer_prog program =
     let vdecls, fdecls = program in
     let global_env = get_global_environment in
     let a_vdecls = List.map (check_vdecls global_env)
vdecls in
     ignore (List.map (add_function global_env) fdecls);
     let a_fdecls = List.map (check_function_bodies
global_env) global_env.scope.functions in

     print_endline "==============Printing
Environment==============";
     ignore (print_env global_env);
     print_endline "==============Finished printing
Environment==============";

     (a_vdecls, a_fdecls)
```

8.1.6 codegen.ml:

```
(*
     Brian Bourn, Addisu Petros
*)
open Sast

type str_list ={
mutable   stream_list    : string list;
}

let __list = { stream_list=[]; }

let add_stream _list name =
     _list.stream_list <- name :: _list.stream_list;
     name
```

```
let find_in_list name =
     let found = try
          List.find (fun(s) -> s=name) __list.stream_list
     with
          Not_found ->
               "Not in list"
     in
     found

let empty_list _list =
     _list.stream_list <- []

let string_of_t = function
    Sast.Int -> "int"
  | Sast.Float -> "float"
  | Sast.String -> "string"
  | Sast.Bool -> "bool"
  | _ -> ""

let rec string_of_expr = function
    Sast.Literal(_, l) -> string_of_int l
  | Sast.Id(_, s, e, _) -> s ^(match e with Noexpr -> "" |
_ -> " = " ^
                        string_of_expr e ^ "")
  | Sast.Binop(_, e1, o, e2) ->
      "(" ^ string_of_expr e1 ^ " " ^
      (match o with
     Ast.Add -> "+" | Ast.Sub -> "-" | Ast.Mult -> "*" |
Ast.Div -> "/"
      | Ast.Exp -> "^" | Ast.Equal -> "==" | Ast.Neq ->
"!="
      | Ast.Less -> "<" | Ast.Leq -> "<=" | Ast.Greater ->
">"
      | Ast.Geq -> ">=")
      ^ " " ^string_of_expr e2 ^")"
  | Sast.String_literal(_, s) ->  "\"" ^ s ^ "\""
  | Sast.Float_literal(_, f) ->  string_of_float f
  | Sast.Bool_literal(_, b) -> string_of_bool b
  | Sast.Call(_, f, el) ->
      f ^ "(" ^ String.concat ", " (List.map string_of_expr
el) ^ ")"
  | Sast.Basic_Op(b, s, i, e, j) ->
          (match b with
        Signal -> (match e with
          Sast.Noexpr ->
                    s ^
                    let str =
                         let n = find_in_list s in
```

```
                                    if (n = "Not in list")
                                    then (
                                         ignore (add_stream
__list s);
                                         ".value_at(" )
                                    else
                                         ".peek("
                         in
                         str
                         ^ (match i with
              Sast.String_(i) -> i
            | Sast.Int_ (i) -> string_of_int i
            ) ^")"
                 | _ -> "for (int xz=0; xz<1024; xz++) " ^
                         s ^
                         let str =
                              ignore (add_stream __list s);
                              ".set_value("
                         in
                         str
                   ^ string_of_expr e ^ ")" ^ ";"
          )(*".set_value(" ^ (match e with Sast.Basic_Op
(_,s,_,_) -> s ^ ".value_at(0)" | _ -> string_of_expr e )^
")" *)
          |  Array -> s ^ "[" ^
            (match i with
              Sast.String_ (i)-> i
              | Sast.Int_(i) -> string_of_int i
            ) ^ "]" ^ (match e with Sast.Noexpr -> "" | _ ->
"= " ^ string_of_expr e)
        )
    | Sast.Noexpr -> ""


let string_of_var var = match var with
      Primitive(x) ->
           string_of_t x.a_typ ^ " " ^ x.a_name ^
                 (match x.a_exp with Noexpr -> ";\n" | _ ->
" = " ^
                      string_of_expr x.a_exp ^ ";\n")
  | Basic(_) -> ""

let string_of_vdecl = function
     Sast.Variable_Dec(_, v) -> string_of_var v


let rec string_of_stmt = function
    Sast.Block(stmts) ->
```

```
      "{\n" ^ String.concat "" (List.map string_of_stmt
stmts) ^ "}\n"
  | Sast.Expr(_, expr) ->
          empty_list __list;
          string_of_expr expr ^ ";\n";
  | Sast.Return(expr) -> "return " ^ string_of_expr expr ^
";\n";
  | Sast.If(e, s, Block([])) -> "if (" ^ string_of_expr e ^
"){\n" ^ string_of_stmt s^"}\n"
  | Sast.If(e, s1, s2) ->  "if (" ^ string_of_expr e ^
"){\n" ^
      string_of_stmt s1 ^ "\n}else{\n" ^ string_of_stmt s2
^ "}"
  | Sast.For(e1, e2, e3, s) ->
      "for (" ^ string_of_expr e1  ^ " ; " ^ string_of_expr
e2 ^ " ; " ^
      string_of_expr e3  ^ "){\n " ^ string_of_stmt s^
"\n}"
  | Sast.While(e, s) -> "while (" ^ string_of_expr e ^
"){\n " ^ string_of_stmt s^"}"
  | Sast.Print(t, s) -> (*"printf(\" "^
                        (match t with
                               String -> "%s\", \""
^string_of_expr s ^ "\");"
                             | Int -> "%d\",
"^string_of_expr s ^ ");"
                             | Float -> "%f\",
"^string_of_expr s ^ ");"
                             (*| Var -> (match s.a_typ
with
                                       String -> "%s\",
"^s.a_name ^ ");"
                                     | Int -> "%d\",
"^s.a_name ^ ");"
                                     | Float -> "%f\",
"^s.a_name ^ ");"
                             ) *)

                        )  *)

               empty_list __list;
               (match t with Sast.Float -> "for(int ew=0;
ew<1024; ew++) " | _ -> "" ) ^ "cout << " ^  string_of_expr
s ^  " <<endl;\n"
  | Sast.Prim_Assign(_, v) -> string_of_var v
  | Sast.Basic_Dec(t, n, l) -> (match t with Signal ->
"circular_buffer "^ n ^";\n"
                              | Array -> "int " ^ n ^ "["
```

```
^ string_of_int l ^ "]" ^ ";\n")
  | Sast.Sum(_,var,s, e1, e2, e3) -> empty_list __list;
"int " ^ var ^"=0; \n for(int " ^s^"=" ^ string_of_int e1 ^
"; " ^s^"<=" ^ string_of_int e2 ^"; " ^s^"++){\n" ^ var
^"+=" ^ string_of_expr e3 ^ ";\n}"
  | Sast.Basic_Sum(_,var, s,e1,e2, e3) -> empty_list
__list;  "for(int ew=0; ew<1024; ew++){\n" ^ "int u=0; \n "
^
     "for(int " ^ s ^ "=" ^ string_of_int e1 ^ "; " ^  s ^
"<" ^ string_of_int e2 ^ "; " ^ s ^ "++){\n " ^ "u+= " ^
string_of_expr e3 ^ ";\n} \n " ^
     var ^ ".set_value(u); \n}"


let string_of_a_fdecl fdecl =
  (match fdecl.a_is_stream with true -> "void " | false ->
"int " (*string_of_t fdecl.a_ret_type*))
  ^fdecl.a_fname ^ "(" ^ String.concat ", " (List.map
Ast.string_of_formal fdecl.a_formals) ^ ")\n{\n" ^
  String.concat "" (List.map string_of_stmt fdecl.a_body) ^
  "\n}"

let string_of_prog (vars, funcs) =
    "#include <iostream>\n#include <fstream>\n#include
<iomanip>\n#include <string>\n
     #include \"libcirc/circ_buffer.h\"\nusing namespace
std;\n"^
  String.concat "" (List.map string_of_vdecl vars) ^ "\n" ^
  String.concat "\n" (List.map string_of_a_fdecl funcs)
```

8.1.7 dspjockey.ml:

```
(*
     Vanshil Shah
*)
type action = Raw | Ast | Sast | Codegen

let string_of_action = function
     Raw -> "Raw"
  | Ast -> "Ast"
  | Sast -> "Sast"
  | Codegen -> "Codegen"
let filename = ref ""
let opts = ref Raw

let speclist = [
     ("-r", Arg.Unit (fun _ -> opts := Raw), "Raw");
     ("-a", Arg.Unit (fun _ -> opts := Ast), "Ast");
     ("-s", Arg.Unit (fun _ -> opts := Sast), "Sast");
```

```
    ("-c", Arg.Unit (fun _ -> opts := Codegen), "Codegen");
    ("-f", Arg.Set_string filename, "Setting filename");
  ]

let usage = "usage: " ^ Sys.argv.(0) ^ " [-r] [-a] [-s] [-
c] [-f <Filename>]"

let () =
     Arg.parse
          speclist
          (fun x -> raise (Arg.Bad ("Bad Argument : " ^
x)))
          usage;

     let in_channel = open_in !filename in
     let lexbuf = Lexing.from_channel in_channel in
     let program = Parser.program Scanner.token lexbuf in
     match !opts with
          Raw -> print_string (Ast.program_s program)
     |    Ast -> let listing = Ast.program_s program in
                      print_string listing
     |    Sast -> (try
                           ignore (Analyzer.infer_prog
program);
                           print_endline "Semantic analysis
complete"
                      with
                           Failure(x) -> print_endline x)
    |   Codegen ->
           let main_out = open_out "main.cpp" in
              let a_program = try
                   Analyzer.infer_prog program
              with
                   Failure(x) ->
                        print_endline ("Semantic analysis
failed: " ^ x);
                        ([],[])
              in
              if a_program <> ([],[])
              then
                   let listing = Codegen.string_of_prog
a_program
               in output_string main_out listing
```

8.2 Circular Buffer C++ Files:

8.2.1 circ_buffer.h:

```
/**
 * Addisu Petros
 * Vanshil Shah
 */
#ifndef __CIRC_BUFFER_H__
#define __CIRC_BUFFER_H__

#define MAX_SIZE 1024

class circular_buffer {
public:
     float buffer[MAX_SIZE];
     int cur_index;

     circular_buffer();

     bool empty();
     float peek(int offset);
     float value_at(int offset);
     void set_value(int value);


};

#endif
```

8.2.2 circ_buffer.cpp:

```
/**
 * Addisu Petros
 * Vanshil Shah
 *
 */
#include "circ_buffer.h"

circular_buffer::circular_buffer() {
     for(int i = 0; i<MAX_SIZE; i++)
          buffer[i] = 0.0;
     cur_index = 0;
}

float circular_buffer::peek(int offset) {
     if(cur_index-offset < 0){
```

```
            return buffer[MAX_SIZE - offset];
        }
        else{
            return buffer[(cur_index) -offset];
        }
}

float circular_buffer::value_at(int offset) {
        float val;
        if(cur_index-offset < 0){
            val=buffer[MAX_SIZE - offset];
        }
        else{
            val = buffer[cur_index-offset];
        }
        cur_index++;
        if(cur_index == MAX_SIZE) {
            cur_index = 0;
        }

        return val;
}

void circular_buffer::set_value(int value) {
        buffer[(cur_index)] = value;
        cur_index++;
        if(cur_index == MAX_SIZE) {
            cur_index = 0;
        }
}
```

8.3 Compiler Makefile:

```
OBJS = ast.cmo parser.cmo scanner.cmo analyzer.cmo
codegen.cmo dspjockey.cmo

TESTS =

TARFILES = Makefile testall.sh scanner.mll parser.mly \
        ast.ml analyzer.ml codegen.ml dspjockey.ml \
        $(TESTS:%=tests/test-%.mc) \
        $(TESTS:%=tests/test-%.out)

dspjockey : $(OBJS)
        ocamlc -o dspjockey $(OBJS)

.PHONY : test
test : dspjockey testall.sh
```

```
	./testall.sh

scanner.ml : scanner.mll
	ocamllex scanner.mll

parser.ml parser.mli : parser.mly
	ocamlyacc parser.mly

%.cmo : %.ml
	ocamlc -w A -c $<

%.cmi : %.mli
	ocamlc -w A -c $<

dspjockey.tar.gz : $(TARFILES)
	cd .. && tar czf dsp_jockey.tar.gz
$(TARFILES:%=microc/%)

.PHONY : clean
clean :
	rm -f dspjockey parser.ml parser.mli scanner.ml
testall.log output \
	*.cmo *.cmi *.out *.diff *.cpp

# Generated by ocamldep *.ml *.mli
ast.cmo:
ast.cmx:
parser.cmo: ast.cmo parser.cmi
parser.cmx: ast.cmx parser.cmi
scanner.cmo: parser.cmi
scanner.cmx: parser.cmx
parser.cmi: ast.cmo
codegen.cmo: ast.cmo
codegen.cmx: ast.cmx
analyzer.cmo: ast.cmo sast.cmo
analyzer.cmx: ast.cmx sast.cmx
```

8.4 run_compiler.sh (for building the compiler):

```
# Brian Bourn, Vanshil Shah
OUTPUT=$(./dspjockey -c -f $1)

#echo $OUTPUT

if [[ $OUTPUT == *"Semantic analysis failed"* ]]
then
	exit
fi
```

```
if [ $# -ge 2 ]
then
     touch $2
     echo Generating executable $2
     g++ main.cpp -o $2 -I./libcirc libcirc/circ_buffer.cpp
else
     echo Generating executable a.out
     g++ main.cpp -I./libcirc libcirc/circ_buffer.cpp
fi
```

8.5 Tests:

8.5.1 Dsp_Hello_World.dj (hello world signal test 1):

```
stream hello_sig(float val) {

     let sig = Signal[];
     sig[time] = (val=val+1.0);
     print sig[time];
}

int main() {
     hello_sig(0.0);
}
```

8.5.2 Dsp_Hello_World2.dj (hello world signal test 2):

```
stream hello_sig(float val) {

     let sig = Signal[];
     sig[time] = sig[time]+1.0;
     print sig[time];
}

int main() {
     hello_sig(0.0);
}
```

8.5.3 binop.dj (binop operator test):

```
int x;
int y;
float f;
bool z;

int main() {
     if(2>1){
          print "Hello World";
```

```
     }
}
```

8.5.4 binop2.dj (binop operator test 2):

```
int x;
int y;
float f;
bool z;

int main() {
     if(1>2){
          print "Hello World";
     }else{
          print "Goodbye World";
     }
}
```

8.5.5 test-arith1.dj (arithmetic test 1):

```
int main (){

     int x;
     x=5;
     x=6;
     print x;

     if (x==6){
          print "arith2 passed";
     }
     else {
          print "arith2 failed";
     }

}
```

8.5.5 test-arith2.dj (arithmetic test 2):

```
int test(int a){
     if( 5*5+25 == a) {
          print "arith3 passed";
     }
     else {
          print "arith3 failed";
     }

}
int main (){
```

```
test(50);

}
```

8.5.6 test-arith3.dj (arithmetic test 3):

```
int main (){

print "haha";

}
```

8.5.7 test-arith4.dj (arithmetic test 4):

```
float x;
int test() {
     x=7;
     float y=2.0;
     float z=10.0;

     if( ((x/y) == 3.5)) {
          print "arith5 passed";
     }
     else{
          print "arith5 failed";
     }

}
int main () {
     test();
}
```

8.5.8 test-arith5.dj (arithmetic test 5):

```
int x;
int main (){
     float y=10.0;
     for (x=0; x<5; x=x+1){
          y=y*10.0;
     }

     if(y==1000000.0){
          print "arith5 passed";
     }
     else {
          print "airth5 failed";
     }
```

```
}
```

8.5.9 test-arith6.dj (arithmetic test 6):

```
int x;
int main (){
     float y=10.0;
     string f="this is f";
     print f;
     while (x<5) {
          y=y*10.0;
          x=x+1;
     }

     if(y==1000000.0){
          print "arith6 passed";
     }
     else {
          print "airth6 failed";
     }
}
```

8.5.10 print.dj (print function test):

```
int x;
int y;
float f;
bool z;

int main() {
     print "Hello World";
}
```

8.5.11 summation.dj (summation formula test):

```
int main()
{
     int a = 0;
     int x = Sum i=0 to 100 : i;
     print x;

}
```

8.5.12 test-array.dj:

```
main(){
/*int x = 5;*/
let sig = Signal[];
```

```
let arr = Array[10];
}
```

8.5.13 test-array2.dj (test array 2):

```
int x;
int b = 10;
float z = 10.0;
int a = 5;
stream str_func(int a) {
     int w = 10;
     let arr = Array[10];
}


int main() {
     int r = 5;
     str_func(10);
}
```

8.5.14 test-array3.dj (test array 3):

```
stream str_func(int a) {
     int w = 10;
     let arr = Array[2];

     arr[0]=5;
     print arr[0];

}


int main() {
     str_func(10);
}
```

8.5.14 test-array-assign.dj (array assignment test):

```
main(){
/*int x = 5;*/
let sig = Signal[];
sig[time] = 5;
let arr = Array[10];
arr[11] = 5;
arr[0] = cool;

}
```

8.5.15 test-bool.dj (bool test):

```
bool x;
bool y = false;
int test() {
     x= true;

     print x;
     print y;
     print x==y;

}


int main() {
     test();
}
```

8.5.15 test-types1.dj (type test 1):

```
int main (){
     float y=10.0;
     int x = 10;

     if(y!=x){
          print "types1 passed";
     }
     else {
          print  "types1  failed,  int  should  not  equal
float";
     }
}
```

8.5.16 test-types2.dj (type test 2):

```
int main (){
     int x=10;
     string y= "10";

     if(y!=x){
          print "types2 passed";
     }
     else {
          print  "types2  failed,  int  should  not  equal
string";
     }
}
```

8.5.17 test-types3.dj (type test 3):

```
int main (){
     int x=10;
     bool y= true;

     if(y!=x){
          print "types2 passed";
     }
     else {
          print "types2 failed, int should not equal bool";
     }
}
```

8.5.18 test-signal.dj (signal test):

```
main(){
/*int x = 5;*/
let sig = Signal[];
}
```

8.5.19 test-sig-assign.dj (signal assign test 1):

```
int x;
float z = 10.0;
int a = 5;
int b = 10;

stream str_func(int a) {
     int w = 10;
     let sig = Signal[];
     sig[time] = 100;
}

int main() {
     str_func(10);
}
```

8.5.20 test-sig-assign2.dj (signal assign test 2):

```
main(){
/*int x = 5;*/
let sig = Signal[];
sig5[time] = 5;
}
```

8.5.21 test-sig-assign3.dj (signal assign test 3):

```
int x;
float z = 10.0;
int a = 5;
```

```
int b = 10;

stream str_func(int a) {
      int w = 10;
      let sig = Signal[];
      let sig2 = Signal[];
      sig[time] = 100;
      sig2[time] = 1.0+sig[time];
}

int main() {
      str_func(10);
}
```

8.5.22 test-sig-call.dj (signal call):

```
main(){
let sig = Signal[];
sig=create_unit_step(10,10);
}

create_unit_step(amplitude, time){
print "haha";
}
```

8.5.23 multi_func.dj (multi-function):

```
int x;
float z = 10.0;
int a = 5;

int main() {
      int r = 5;
}

int b = 10;
stream str_func(int a) {
      int w = 10;
      let arr = Array[10];
      let sig = Signal[];
}
```

8.5.24 unit_step.dj (unit_step signal test):

```
stream unit_step(float amplitude, float t) {

      let sig2 = Signal[];
```

```
     float current = sig2.get_current();

     if(t<current) {
          sig2[time]=0;
     }

     else {
          sig2[time]=amplitude;
     }


}


int main() {
     unit_step(2.0,5.0);
}
```

8.5.25 lowpass_filter.dj (lowpass filter test):

```
stream lowpass_filter(float dt, float rc) {
     float alpha = dt/(rc+dt);
     float val=0.0;

     let sig = Signal[];
     sig[time]= (val=val+1.0);
     let sig2 = Signal[];

     sig2[time]=   alpha   *   sig[time]   +   (1.0-alpha)   *
sig2[time-1];

     print sig2[time];

}

int main() {
     lowpass_filter(2.0,5.0);
}
```

8.5.26 fir_filter.dj (fir filter test):

```
stream fir_filter() {

     let coef_array= Array[10];
     int x =0;
     while(x<10){
     coef_array[x]=5;
     x=x+1;
```

```
     }

     float val=0.0;
     let sig = Signal[];
     sig[time]= (val=val+1.0);

     let output_signal = Signal[];
     let   output_signal[time]   =   Sum   i=0   to   10   :
coef_array[i] * sig[time-1];
     print output_signal[time];

}

int main() {
     fir_filter();
}
```