DSP Jockey

Language Reference Manual
COMS W4115 Programming Languages and Translators
Professor Stephen Edwards

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

Table of Contents

Introduction

The purpose of DSPJockey is to provide a language that makes it easy for programmers to express signal processing algorithms. We are aiming to provide the tools necessary to conveniently manipulate signals. To facilitate this, we include a functional programming-like syntax that allows easy definition of recursive functions that manipulate signals. In addition to this, our language also has a notion of global time which corresponds to the current time in the stream that we are processing. Since many DSP algorithms involve time manipulations, we feel that the time parameter makes it very intuitive to express algorithms. We have included some code samples at the end to show how natural it is to express signals and to pass them through filters to get output signals.

Lexical Elements

Comments

DSPJockey allows for single-line and multiline comments, similar to C-style comments.

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

```
multi-line comment:
/* this is
multiline comment
*/
```

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. DSPJockey is case sensitive.

Keywords

Keywords are specific identifiers that the language uses to denote certain types or objects. They cannot be overloaded.

Keyword:	**Meaning/Description:**
let	used to declare a new variable
dec	used to declare a new mathematical function
int	data type that represents an integer
float	data type that represents a floating point number
if, else, elseif	specify conditional statement

Signal	keyword used to declare a new signal stream
print	used to print information to standard out
to	used to specify a range (from a to b)
bool	data type that represents a Boolean value
true, false	used to denote boolean keyword true, false
sum	is the keyword to denote the summation
return	return a value
char	data type that represents a character
array	data type that represents a list of values

Constants

Constants are either the Boolean types true and false or just a plain sequence of digits.

For example *1.241, 1,* and *true* are all constants.

Separators

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

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

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

```
statement 1;
statement 2;
```

Whitespace

Whitespace is represented by tab and blank characters. Whitespace is ignored by the compiler and is only used to separate lexical tokens from each other.

Operators

Operators are given in detail under the *Expressions and Operators* section.

Data Types

Basic Types

DSPJockey has five basic data types Integer, Float, Boolean, Char 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.

`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
`char`	8 bit datatype designated mostly for holding character Information for ASCII conversion
`array`	A standard list, style array which can be used to collect any of the four previous types of data

Builtin Types

Additionally DSPJockey has one special data type called `stream`, which is used to represent an ongoing signal. A Stream 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 512 samples. additionally the current value of the signal is always stored in the array at index t or the current time. previous samples are accessed by subtracting an integer value from t. for example

```
stream signal;
s1 = signal[t];
s2 = signal[t-1];
```

in this case `s1` would be set to the value of the current sample of the signal while `s2` would be set to the value of the previous sample.

Expressions and Operators

Here, we describe the syntax of expressions within our language. A given expression is a sequence of operands and operators that evaluate to some value. Expressions in DSPJockey are evaluated left to right. However, this does not supersede the precedence of operators.

Variables

A variable is an expression whose type and value is the same as the type of the expression that it has been designated to.

Constants

A constant is an expression whose type can be assigned to some boolean, string or number (float or integer).

Parenthesized expressions

Parenthesis are used to clearly depict and modify operator precedence. Other than that, a parenthesized expression will have the same type as a non-parenthesized expression.

Binary Operator Expressions

A binary operator expression is an expression that can be formed using a binary operator on two individual expressions. Such a complex expression will have the form given by *expression1 binary_operator expression2,* where binary_operator can be a logical, arithmetic or relational operator.

Logical Operators

Operator	Description
&&	Conjunction
||	Disjunction

We have two logical operators, && and ||, which are used to do a logical operation on a given input. The inputs specified to a logical operator must evaluate to a boolean and as such the result of a logical operation is also a boolean. Where conflict may arise, the logical && operator has precedence over ||

Arithmetic Operators

Operator	Description
/	division operator
*	multiplication operator
+	additive operator
-	subtraction operator

The input operands specified to an arithmetic operator must be numbers. The result of the arithmetic expressions will be the number obtained by applying the arithmetic operation on the operands. Where conflict may arise, the precedence of the operators is as specified in the table above - from highest precedence to lowest.

Comparison/relational Operators

Operator	Description
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to
==	equal to
!=	not equal to

The inputs to a comparison operator should be of the same type to enable comparison. The return of such a relational expression will be a boolean value determined by evaluating the expression. All relational operators have the same precedence in an expression.

Array Index Operator
We use [] to index into an array or a signal and access the value at the specified location.

Function Expressions
Expressions relating to the creation and calling of functions are detailed under the section ***Functions.***

Statements

A statement does not have a specific value and type. Instead, a statement is typically used for its side effects. The following are types of statements in DSPJockey.

Conditional Statements
These types of statements can take on of the following forms:

`if ( expression ) statement`
`if ( expression ) statement else statement2`
`if ( expression ) statement elseif ( expression ) statement2 else statement3`

For any of the cases above, the expressions must be items that evaluate to give a boolean. In the first form, if the boolean expression is true, then the statement is executed. Otherwise, the statement is ignored. In the second form, if the expression evaluates to true, then the first statement is executed. Otherwise, statement2 shall be executed. For the last form, we have a series of *if ...elseif* sequences. The statement associated with the first expression that evaluates to true will be executed. if none of the expressions are true, then the statement under the last *else* shall be executed.

Assignment Statements
An assignment statement uses the operator = to form assignment statements. Such statements will have the format *lvalue = expression ,* where the value of the expression is evaluated and stored in lvalue.

Return Statements
These statements use the *return* keyword to return a value. The keyword is used in conjunction with an expression as follows:

```
return expression;
```

Function Call Statements
Statements associated with function calls are detailed under the section ***Functions.***

Functions

In our language, we have two different notions of functions. One of them, in the mathematical sense is a function which takes an input value, performs some mathematical operations on it and returns the value of the operation. The other type of functions are functions like in traditional languages that are a way to separate programming logic. From here, they will be referred to as mathematical functions and logical functions.

Function Declarations

To declare a mathematical function, we start with the keyword *dec*, which lets the compiler know that we will be performing mathematical operations with the input values and that this will be a recursive function. Here is a simple declaration of a low pass filter function

```
dec lowpass_filter(orig_signal, dt, rc) {...}
```

or more generally,

```
function-definition:
       dec function-declarator function-body
```

where

```
function-declarator:
       declarator (input_signal, parameter-listopt)
```

```
parameter-list:
       identifier
       identifier, parameter-list
```

The logical functions are declared as below:

```
function-definition:
       type-specifier-opt function-declarator function-body
```

where

```
function-declarator:
       declarator (parameter-listopt)
```

```
parameter-list:
       identifier
       identifier, parameter-list
```

Function Definitions

Again, the type of function definition depends on the type of function that we are trying to write. For mathematical functions, the function structure should follow a roughly equation-like syntax with the beginning lines of the function doing intermediate calculations and the last line modifying the signal we want to operate on.

```
function-body:
       { statement-list signal-assignment }
```

Having the signal on the last line also enforces the return value of functions of type dec, which is the signal type.

Logical functions are defined in a similar manner but return values are specified with the return keyword for functions that have a return value.

```
function-body:
       { declaration-listopt statement-list }
```

Calling Functions

To call user-defined mathematical functions in this language, you have to create a new signal to be assigned to the returned signal from the function. Then, you can just use the assignment operator to assign the function return value to the new signal.

```
let new_signal = Signal[]
new_signal = func(original_signal);
```

To call logical functions, you can just call the function with the parameters that the function requires.

```
func_name(<function parameters>);
```

Programming Structure and Scope

Program Structure

DSPJockey programs can exist in a single file or across many by importing functions from other files or libraries through the use of the import function. All programs must be written in files with the extension ".dspj". 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 section for example purposes

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.dspj**/
int i;

void set_i(){
	i =2;
}

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

The variable i will originally be set to 2 by the set_i function then be set to 3 later in the main function.

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,

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

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

```
    set_i();
    print  i;
}
```

will return i = 2 since set i only defines an int i within it's own scope and will not modify main's variable i.

Sample Programs

Our language is based on an implicit time parameter t, used to access and modify our signals. t gives us access to the current time step in any calculation that we will be doing and to access previous times, we would subtract integers from t.

/**********Low Pass Filter**********/

```
// Given an input signal, we would like to output a signal that is  low passed
dec lowpass_filter(orig_signal, dt, rc) {
      alpha = dt/(rc+dt);
      new_signal[t] = alpha * orig_signal[t] + (1-alpha) *
new_signal[t-1];
}

// Code to read in samples into a stream
int main() {
      let sig = Signal[];
      let new_signal = Signal[];
      sig = stdin;
      new_signal = lowpass_filter(sig, 10, 10);
      return 0;
}
```

/**********FIR Filter**********/

$$y[n] = b_0x[n] + b_1x[n-1] + \cdots + b_Nx[n-N]$$
$$= \sum_{i=0}^{N} b_i \cdot x[n-i],$$

where:

- x[n] is the input signal,
- y[n] is the output signal,
- N is the filter order; an Nth-order filter has (N+1) terms on the right-hand side
- b_i is the value of the impulse response at the *i*'th instant for $0 \le i \le N$ of an Nth-order FIR filter. If the filter is a direct form FIR filter then b_i is also a coefficient of the filter .

```
let coef_array = Array[size];
dec fir_filter(input_signal, coef_array) {
```

```
        // Setting each sample of the output signal to the sum of the coefficients multiplied by
        // the time shifted input array
        output_signal[t] = sum i=0 to coef_array.size :
coef_array[i]*input_signal[t-i];
}
let output_signal = Signal[];
output_signal = fir_filter(input_signal, coef_array);
```

/**********IIR Filter**********/

$$y[n] = \frac{1}{a_0}\left(\sum_{i=0}^{P} b_i x[n-i] - \sum_{j=1}^{Q} a_j y[n-j]\right)$$

where:

- P is the feedforward filter order
- b_i are the feedforward filter coefficients
- Q is the feedback filter order
- a_i are the feedback filter coefficients
- x[n] is the input signal
- y[n] is the output signal.

```
dec iir_filter(input_signal, coef_array_a, coef_array_b) {
        //here we are taking a sum over the input or output signals and their
        //coefficient arrays signal so that we can take the difference for an FIR filter
        xsum = sum i=0 to coef_array_a.size : coef_array[i]
              *input_signal[t-i];
        ysum = sum j=1 tp coef_array_b.size : coef_array_b[i]
              *output_signa[t-j];
        //the output of the IIR will subtract ysum (output sum) from xsum (input sum) and then
multiply that difference value by 1 over the first value in the 'a' feedback filter coefficient array
        output_signal[t] = 1/coef_array_a[0]*(xsum - ysum)
}
let coef_array_a = Array[size];
let coef_array_b = Array[size];
let output_signal = Signal[];
output_signal = iir_filter(input_signal, coef_array_a, coef_array_b);
```

```
/************Unit step************/
dec create_unit_step(amplitude, time) {
      if t < time
            sig[t] = 0;
      else
            sig[t] = amplitude;
}
let sig = Signal[];
sig = create_unit_step(10, 10);
```