SASSi Language Reference Manual

Group Leader: Carl Morgan, Paul Salama, Xiaotang Zhang

1 Lexical conventions

1.1 Comments

Comments are enclosed by the characters "/*" and "*/". Also, the "//" style of commenting for a single line applies.

1.2 Identifiers

An identifier is under the same rules that bind their name in C. It must consist of letters, digits, and the underscore or "_" for long names. The first character must be a letter. All identifiers are case sensitive meaning that "A" is considered completely different from "a"; this assists in checking uniqueness.

1.3 Keywords

The following words are reserved and thus can not be used as identifiers by the user:

for	if	else	return	include		procedure	true	false	plot	print
bool	int	float	double	string	char	vectori	vectord		load	
const	xconst	err	pi	exp	inf					

1.4 Numbers

An integer is a set of digits preceded by an optional "-" signifying a negative integer.
A floating-point number follows the same rules that are applied in C. It will consist of digits, an optional decimal point ".", another set of digits, and an "E" or "e" for exponential numbers followed by a signed integer. Integers and floating point numbers will be distinguishable. It should be noted, a floating-point number need not have an exponential part. Also, regular integers will be accepted.

1.5 String Literals

String literals will be defined as anything confined within a set of double quotes. Double quotes inside the string literal will be preceded by a backslash, i.e. \".

1.6 Characters

Characters will be supported as follows: a single quote, ', a letter, number, or single symbol, closed with a single quote,'. As described previously a double quote inside of a string literal will be preceded by a backslash.

1.7 Other Tokens

There are a few reserved characters or pairs of characters that must be used correctly in the language. The reserved characters and pairs are as follows:

{	}	(	)	[	]	
,	;	+	-	*	/	
=	==	+=	-=	*=	/=	
<=	>=	<	>	<>	++	--

2 Types

bool: Boolean values that can either be **true** or **false** (and True/False and TRUE/FALSE)
int: standard 32-bit integers
float: 32-bit IEEE floating format
double: 64-bit IEEE floating format
string: a string of characters
char: a single character
procedure: a user defined procedure
vectori: a M by 1 vector of **int**s
vectord: a M by 1 vector of **double**s

3 Expression

3.1 Primary Expression

Primary expressions are identifiers, constants, procedure calls, and access to vector types. They also include any expression in between "(" and ")".

3.2 Identifier

An identifier is a left-value expression. It will be evaluated to some value delimited by the restrictions of its type.

3.3 Constant

A constant is a right-value expression. It will be evaluated to the constant itself.

3.4 Procedure Call

A procedure call consists of a procedure identifier, followed by the appropriate list of parameters that are enclosed in "(" and ")". The list of parameters contains zero or more identifiers all separated by commas. Each parameter is an expression and each procedure call is itself left-valued.

3.5 Access to Vector types

The primary expression consists of a vector identifier or other left valued expression. This is followed by a list of indexes enclosed by "[" and "]". The list of indexes can contain as many numbers as the vector is defined to hold. The expression itself is left valued.

3.6 (expression)

A parenthesized expression is a primary expression. It returns the value of the enclosed expression. The presence of the parentheses gives the expression a higher priority than the other expressions in the statement.

3.7 Arithmetic expressions

Arithmetic expressions take primary expressions as operands.

Unary Arithmetic operators

The operators "+" and "-" can be prefixed to an expression. The "+"operator returns the expression in a positive form and the "-" returns the expression in a negative form. They are applicable only to the following: int, double, float, vectori, and vectord.

Multiplicative Operators

Binary operators "*" and "/" indicate multiplication and division respectively. They are grouped left to right. They are applicable only to the following: int, double, float, vectori, and vectord.

Additive operators

Binary operators "+" and "-" indicate addition and subtraction respectively. They are grouped left to right. They are applicable only to the following: int, double, float, vectori, and vectord.

3.8 Relational Expressions

Binary relational operators "<=", ">=", "==", "<", ">", and "<>" indicate if the first operand is less than or equal to, greater than or equal to, equal to, less than, greater than, or not equal to the second operand respectively. A **bool** value is returned in the case that both operands are numbers (i.e. not characters, strings, or vectors), and exactly two operands are needed.

Only "==" and "<>" are used in the vectors. If two vectors are compared these will tell you if they are exactly equal or exactly not equal by checking the values in each vector. Also, these only work if the vectors being compared have the same size.

4 Vector Indices

Vector indices can have only one component (i.e. m[3]). Index components can be a single arithmetic expression with an integer compatible value. Any 1-by-1 sub-vector is handled just as an element that matches the type of the vector.

5 Variable and Constant Declarations

There are two types of data: variables and constants. Constants must be defined at the beginning of any program or procedure. Constants can not under any circumstances be changed. The form they take for declaring is rigid and must be followed as below:

const
type somename = *number*;
.
.
xconst

The constants must be declared with their type. This is for type checking at compile time. The little dots are used to denote that you can put more statements in the **const** section. Notice that the const section is ended with a **xconst** this is used to denote that you are exiting a constant section and any other data type declarations are for variables. This is also a quick way to separate the data types and saves you time from typing something ridiculously long(i.e. static const int x=5;).

Variable can be declared anywhere in the program. It takes a similar form as the constant declaration and looks like :

type somename;

or

type somename=*initial value*;

The second version of the variable declaration takes an *initial value* and automatically assigns the variable to that value. This notation saves space. Also, it should be noted that the *initial value* has to match the type that the variable is. The *type* must be one of the ones defined previously in the type section(Section 2).

6 Statements

Statements are the basis of a program. They take on form and structure with certain notation. Most instructions are executed sequentially but recursion is a viable option. Items that are to be replaced by the user are in *italics*.

6.1 Statements in "{" and "}"

A group of zero or more statements can be enclosed by curly brackets. This modularizes the program. Any statements that are contained within curly brackets are grouped and run together in a sequential manner.

6.2 Assignments

An assignment can take one of two forms. The first assignment form is a direct assignment in the form of:

Left-valued expression = Right-valued expression ;

This form assigns whatever is on the right of the equal sign to whatever is on the left. The second assignment form is a mathematical assignment in the form of:

Left-valued expression + = Right-valued expression ;
Left-valued expression -= Right-valued expression ;
*Left-valued expression *= Right-valued expression* ;
Left-valued expression /= Right-valued expression ;

This form assigns whatever is on the left hand side to itself added to, subtracted from, multiplied by, or divided by whatever is on the right hand side respectfully.

6.3 Conditional Statements

The conditional statement can take one of two forms. The first form is a simple if and looks exactly like:

```
if ( relational expression )
{
        statement
}
```

This form executes whatever is within the curly brackets if the logical expression evaluates to true. Otherwise, it skips execution of what is within the curly brackets and continues with the program. The second form is the first version followed by an else clause. It looks like:

```
if ( relational expression )
{
        statement
}
else {
        statement
    }
```

This executes the same way as the first form except for the fact that if the logical expression evaluates to false then what is enclosed within the curly brackets after the else statement are evaluated. It is also important to know that these two forms of conditional statements can be nested within each other. By nesting conditional statements you effectively reduce execution time.

6.4 For Loops

All loops in the language take the form of a for loop. This helps in reducing the chances of having an infinite loop but does not eliminate the possibility all together. A for loop will take the following form:

```
for ( assignment ; relational expression ; action )
{
        statement
}
```

The *assignment* is a basic assignment to a variable that is going to be used in the *relational expression* and the *action*. An example of this would be i=0, where i is an integer variable. The *relational expression* follows the rules defined previously and must contain the variable used in the *assignment*. The *action* is either an increment or decrement denoted by the variable name used in the *assignment* followed by "++" or "--"

respectively. The statements enclosed within the curly brackets are executed as long as the *relational expression* evaluates to **true**.

6.5 Procedure Calls

Procedures are different than any other expression. They follow the following form:

procedure_name (*parameter list*);

The *procedure_name* must be a procedure that the user has already defined and must be spelled the correct way. The *parameter list* is the name of the appropriate variables separated by a comma (i.e. foobar(who, knows, why)). The *parameter list* may also be empty, this means that the procedure doesn't take anything. The *parameter list* variables must match the types they are being sent to be assigned. This prevents type mismatching like assigning an **int** to a procedure variable that requires a **double**.

6.6 Return Statements

The return statement can be used within the procedure definition body. It returns either a variable's value or a number to the caller. It is mandatory that you end the statement with a semicolon. A basic return statement would look like:

return 0;

6.7 Including Other Files

The keyword include is used to include other files. The form is as follows:

include <*path name*>;

This allows you to include previous work or procedures that you wish. It also reduces the amount of memory you consume by letting you choose which things you wish to include in your palette of procedures. The *path name* must be exact. The file will not be included if the *path name* entered is not correct. Also, this declaration must be made at the top of your program.

7 Procedure Definitions

A procedure takes only one form but what may be accomplished is unlimited. It follows the form below:

```
procedure return_type procedure_name ( variable list)
{
        statements
}
```

The *return_type* must be one of the predefined types or **void** if nothing is returned. The *procedure_name* is any name that you give but it must follow the guidelines defined by **identifiers**. The *variable list* is a list of parameters defined in the form of type and then identifier and each is separated by a comma. An example of this is:

```
procedure void doesnothing(int x, double y)
{

}
```

It is important to note that the *variable list* can be empty. You will also notice that it is not necessary to have any statements in the body of the procedure. It will compile with no statements. It is of course, your prerogative to have a procedure that does nothing so it is supported.

8 Internal Functions

8.1 Print

The function print() has four different versions. The first version takes a variable as an argument and prints out the value of the variable. This is only used for non vector types. It takes the form:

```
print(v);
```

The second version takes a string and prints it out to the screen. The string must be enclosed in double quotes and takes the form:

```
print("Who knows why you name something foo?");
```

The third version takes just a character and prints it out to the screen. The character must be enclosed in single quotes and takes the form:

```
print('a');
```

The fourth and final version is used only for printing vectors. It looks the same as the first version in the program you make but it behaves quite differently. It prints out a row of values of the vector you pass it. An example:

print(m); where m is a vector type

This would return a print out of the vector m similar to:

3 3 1 8

8.2 Plot

The plot procedure draws a plot that you request. The type of plots aren't limited to just the ones listed below because you can implement your own plot type. The plot command follows the form:

plot(*type_of_plot*, *type_of_data*);

The *type_of_data* can be only one of two types: vector or ordered series. The ordered series take the form:

[x1,x2…], [y1,y2…]

This form is simple to implement and allows the user just to put in the numbers they wish to plot. The vector already has its numbers defined so the call reacts exactly the same with either type. The *type_of_plot* falls under a user defined type or one of the following predefined types:

bar: a bar graph
pie: a pie chart
line: a line graph
error: a line graph with error bars added
curve: an exponential curve graph
best_fit_exp: an exponential curve graph that is to the best fit
best_fit_line: a line graph that is to the best fit

It is important to note that a program utilizing an error plot must have a variable of type **double** that is called **err**. The **err** value must be set before the plot call is made. This value can be changed after the plot call and thus is not necessarily a constant.

Since the user might only want to see a section of the plot, the language allows the domain and range to be set with the procedures:

setrange(*float miny, float maxy*) ;
setdomain(*float minx, float maxx*) ;

In addition to the already defined versions of plotting, the language also supports expression plotting. An expression plot is used for when you have a mathematical expression that you want to use to plot the results with in a boundary of values. It takes the form of:

plotfxn(*expression*, *minx*, *miny*, *maxx*, *maxy*, *interval*) ;

The *expression* is the expression that the user gave to define how the plot will behave. The *minx*, *miny*, *maxx*, and *maxy* are the minimum x, minimum y, maximum x, and maximum y values resptectively. This allows the user freedom in deciding how accurate and what space they wish to see the expression they defined plotted. The *interval* is obviously the interval between each x value that is to be inputted into the expression.

8.3 Built In Statistic Constants

These are constants that are used in mathematical evaluations. They are provided as a convenience to the user.

```
int inf=maxInt;
```

This is the value for infinity. It is defined at the maxInt of the system or the largest integer that can be represented by your computer.

```
double pi= 3.14159265358979323846264338327795;
```

This is the value of pi. Since pi is a very important number and curious within it self, we have defined it to a semi-exact point. If we were to implement it to the millionth decimal it would be quite a waste of space and take considerable time to be used in calculation.

```
double exp=2.71828182846;
```

This is the value of the exponential. This is included because it is used in various other function and is frequently used in mathematical calculations. Please note that imaginary numbers are not supported and there are no tools for conversion from an exponential notation to the imaginary notation.

8.4 Built In Statistic Procedures

Each of the following procedures is an available function to the user. This is the default set of statistical procedures. Others are implemented and can be used, they however must be included by putting an include statement in your program. The following do not require any include statements to be added to your program:

procedure double mean (vectord theVector)

procedure double mean (vectori theVector)

This procedure takes in a vector of doubles as its only argument. It finds the mean of all the elements with in that vector and returns the value as a double. The second version is the same except the values in the vector are all **int**s instead of **double**s. From this point on assume that anywhere you see a procedure defined for one vector type that it also applies to the other(writing both would be a waste of paper and you know how we need trees).

procedure int median(vectori theVector)

This procedure takes in a vector of ints as its only argument. It finds the median of all the elements within that vector and returns the value as an int. The return type matches the type of elements in the vector.

procedure void sort(vectori theVector)

This procedure takes in a vector of ints as its only argument. It sorts the entire vector in ascending order. This is very useful for an n-by-one vector.

procedure int range(vectori theVector)

This procedure takes in a vector of ints as its only argument. It finds the highest and lowest values within the vector and returns the difference between them.

procedure float square(float f)

This procedure takes in a float as its only argument. It returns the value that was passed in squared. This procedure can only be used on variables of type int, float, or double.

procedure float variance(vectori theVector)

This procedure takes in a vector of ints. It finds the variance of the vector and returns it as a float.

procedure float standard_deviation(vectord v)

This procedure takes in a vector of doubles as its only variable. It returns the standard deviation of the elements that are stored in the vector.

procedure vectori intersect(vectori theVector1,vectori theVector2)

This procedure takes in two vectors of the same type. It returns a vector of the same type that was passed. The vector returned contains the intersection of the two vectors passed in which means that it contains all the values that were in the first and all the values that were in the second with no repeated values. This is helpful when dealing with set theory.

procedure int[] union(vectori theVector1,vectori theVector2)

This procedure takes in two vectors of the same type. It returns a vector of the same type that was passed. The vector returned contains the union of the two vectors passed in which means that it contains all the values that were in the first and in the second but not any values that were just in one or the other. This is another helpful set theory procedure.

procedure int contains(vectori theVector,int number)

This procedure takes in a vector of integers and a integer. The procedure searches through the vector that was passed in for the integer that was passed in to the procedure. If the integer is in the vector then the index of that number is returned. If the integer is not found a negative one is returned. A zero is not returned since it could be the first element in the vector which is denoted as zero.

procedure vectori complement(vectori theVector, int range)

This takes in a vector of integers and an integer that is the range. The vector that is returned is a the complement of the vector passed in. This also is used in set theory.

procedure int factorial(int x)

This procedure is essential for statistics. It takes in an integer and returns the factorial of that integer. This will be used internally for both the nCr and nPr functions defined below.

procedure int nCr(int x,int y)

This is known as the Choose function. It is implemented using the factorial procedure discussed previously. If you were to read the mathematical version you would read this as x choose y. This is a fundamental tool used in most distributions so it was essential to included it.

procedure int nPr(int x,int y)

This is known as the Permutation function. It also is implemented using the factorial procedure discussed previously. If you were to read the mathematical version you would read this as x with permutations y.

procedure float prob(int n, int N)

This is the probability function. It returns the value of n divided by N. Float and double versions of this are also implemented. This is absolutely needed for almost all of statistics.

procedure int size(vectori theVector)

This procedure is very simple yet very helpful. It takes in a vector of integers and returns the number of elements in the vector. This is used a lot for size checking and is very helpful.

procedure float probInter(vectori A, vectori B)

This procedure is used in set theory. It is used to find the probability that A intersects B. This probability is then returned.

procedure float condProb(vectori A, vectori B)

This procedure is also used in set theory. The full name is conditional probability. In mathematics it is denoted Prob(A|B). This function is fundamental for such things as Bayes' Law. The value returned is the prob that A will occur given that B occurs.

procedure int IndepProb(vectori A, vectori B)

The full name of this procedure is independence probability. It is the probability that A will occur independently of B occurring or not occurring. This procedure is a test so it returns an integer to define the independence. It returns 0 if the two vectors are not independent and 1 if they are independent.

procedure float Expectation (Vector events, Vector probabilities)

This procedure returns the mathematical expectation of discrete events with given probabilities.

Discrete distributions
f(x) is the probability function and p(X) is the probability for all x <= X.

procedure float f_binomialDistri(int x, int n, float p)
procedure float p_binomialDistri(int x, int n, float p)

The procedure takes in a value x which is smaller than n and returns the binomial distribution with probability p.

procedure float f_geometricDistri(int x, float p)
procedure float p_geometricDistri(int x, float p)

This is a procedure that takes in a value x and the probability p and returns the geometric distribution

procedure float f_hypergeometricDistri(int x, int n, int a, int b)
procedure float p_hypergeometricDistri(int x, int n, int a, int b)

This is a procedure that takes in a value of x which is less than the value of n. The values a and b are used for the choice function which is a small part of the hypergeometric distribution. The procedure performs the hypergeometric distribution and returns a float .

procedure float f_negHypergeometricDistri(int x, int a, int b)
procedure float p_negHypergeometricDistri(int x, int a, int b)

This is the negative hypergeometric distribution function. It works similar to the hypergeometric distribution but in a semi-reverse order.

procedure float f_poissonDistri (int x, float l)
procedure float p_ poissonDistri (int x, float l)

This is the famous Poisson Distribution. It is a very useful function and defines what amount of the bell curve is covered using only an integer x and a float l.

Continuous distribution
p(X) returns the probability for all x <= X

procedure float p_ normalDistri (float x, float sigma, float u)

This procedure performs the Normal Distribution.

procedure float p_ standardDistri (float x)

This procedure performs the Standard Distribution using only a float x.

procedure float p_ zDistri (float x)

This is the famous Z-distribution. It is one of the most used distributions in statistics and thus we must include it.

procedure float p_ tDistri (float x)

This is a student t distribution.

procedure sampling_mean (vector x)

This procedure takes a vector and does the mean of a sample with in the vector.

procedure float sampling_standard_deviation(vector x, int N, int n)

This is the procedure for doing a standard deviation. N is the size of the population and n is the size of the sample in the vector x.

procedure float sampling_zValue(float x, float u, float sd)

This is the central limit theorem, get z value for x with given mean and standard deviation.

procedure float errorOfEstimate (int n, float precision, float sigma)

This procedure calculates the maximum error of estimate.

procedure float sampleSize (float maxError, float sd, float precision)

This is a procedure using sample size for estimating u given maximum error, stand deviation and precision.

procedure vector large_sample_confidence_interval (int mean, int n, float precision, float sigma)
This is a large-sample confidence interval for mean , suitable for n >= 30. It returns a pair of data, the lower bound and the upper bound

procedure vector small_sample_confidence_interval (int mean, int n, float precision, float stDev)
This is a small-sample confidence interval for mean, suitable for n < 30. It returns a pair of data, the lower bound and the upper bound

procedure vector linearRegression_leastSquare (Vector x, Vector y)

This procedure returns a and b for equation Y = a + b X. It is used only for the least square linear regressions.

procedure Vector multipleLinearRegression (Vector x1, Vector x2, Vector y)

This procedure returns b0, b1 and b2 for equation Y = b0 + b1X1 + b2X2. It is used only for multiple linear regressions.

procedure Vector nonlinearRegression_polynomial (Vector x, Vector y)

This procedure returns a, b, c for equation $Y = a + bX + c X^2$. It is used only for polynomial nonlinear regressions.

procedure Vector nonlinearRegression_exponential (Vector x, Vector y)

This procedure returns a, b for equation $Y = a * b^X$. It is used for exponential nonlinear regressions.

9 Sample Program

```
/* This is where constants are defined for the entire program
Here, constant a equals 6 */
const
int a=6;
xconst

/* This is the declarations section
Here integer x is declared, as well as vectord Victor- a vector of length six of doubles */
int x;
vectord[6] Victor;

/* This procedure takes an input integer, and uses the f_negHypergeometricDistri
procedure 6 times , for the values of i from 0-5. It saves the value of each iteration in the
vectord whatsyour. Finally it returns the vectord whatsyour. */
procedure vectord myprocedure(int y)
{int i;                          //declares integer i
 vectord[6] whatsyour;           //declares vectord whatsyour- a length-6 vector of doubles
 for(i=0;i<6;i++)
        {
          whatsyour[i]=f_negHypergeometricDistri(y, a, i);
        }
 return whatsyour;
}

/* These are some basic assignments. First setting int x =3, then calling the procedure
myprocedure with the input x which equals 3. Then the values in vectord Victor are
whatever myprocedure returns. Then print prints out the values of Victor on a line. */
x=3;
Victor=myprocedure(x);
print(Victor);

/* This is an if-else conditional statement, which will only produce a plot if a/2==3, and
it should be since a is defined as 6. If the conditional is true, the plot function produces a
line plot with (0-5) as the x values, and the vectord Victor as the y values. */
if((a/2)==3)
{
 plot(line,[0,1,2,3,4,5],Victor);
}
else    {
          print("Error, you should never see this message! \n");
        }
```