SStats Language Reference Manual
Phil Park
pwp2106

1. Introduction

2. Lexical Conventions:

2.1 Comments:

Characters /* introduce a comment which are terminated with the characters */. Comments can run for multiple lines and must be terminated.

2.2 Identifiers:

An identifier is a sequence of letters and digits with the first being alphabetic. The underscore "_" also counts as an alphabetic. Identifiers are case sensitive.

2.3 Keywords:

The following identifiers are reserved for keyword usage and cannot be used for any other purpose:

int	if	continue
float	elif	return
boolean	else	function
null	for	True
char	while	False
	break	var

2.4 Constants:
SStats has several kinds of constants:

2.4.1 Integer Constants:
An integer constant is a sequence of digits, considered to be decimal

ex: 123456

2.4.2 Character Constants:
A character constant is 1 or 2 characters enclosed in single quotes " ' ". Within a chracter constant the single quote must be preceded by a backslash "\". The backslash character functions as an escape character. Certain non-graphic characters and the backslash itself can be escaped according to the following table:

Newline	\n

Tab	\t
\	\\

ex: 'a'

2.4.3 Floating Constants:
SStats uses the definition of floating constant as defined in the C language reference manual repeated here: "A floating constant consists of an integer part, a decimal point, a fraction part, an e, and an optionally signed integer exponent. The integer and fraction parts both consist of a sequence of digits. Either the integer part or the fraction part (not both) may be missing; either the decimal point or the e and the exponent (not both) may be missing. Every floating constant is taken to be double-precision"

ex: 1.2, 3e5, 5e-6.5

2.4.4 Strings:
A string is a sequence of characters surrounded by double quotes " " ". The type of string is an array of characters. In a string, the double quote character " must be escaped using backslash.

"This is a \"string\""

2.4.5 Arrays and Matrices:
SStats has a particular focus on arrays and matrices. An array is a variable type that stores multiple instances of data. The data stored must be all of the same type. In SStats, the array is used as the list data type. Arrays are declared using brackets '[]'.

ex: var int[] myArray = [] /* initializes an empty array*/

Values can be declared by listing them inside the brackets separated by spaces.

ex: var int[] myArray = [1 2 3 4]

Matrices in SStats are two-dimensional arrays. They can be declared by separating rows within the matrix with semicolons.

ex: var int[] myArray = [1 2 3; 4 5 6; 7 8 9]

Note that a matrix declared this way must have the lengths of its row vectors be consistent across the entire matrix.

ex: var int[] myArray = [1 2 3; 4 5 6; 7] /* this is invalid */

2.5 General Punctuation

Using the previously mentioned identifiers and literals, primary expressions can be made in

SStats.

Parentheses are used for a variety of purposes: function declaration, function argument declaration and specifying order of operations in an expression

```
function foo() { /*function declaration */
}

foo() /* function call */
```

Braces are used to indicate control flow blocks including function bodies.

```
{
/* this is a block */
}
```

Brackets are used for array and matrix expressions as mentioned in section 2.4.5

2.6 Operations

Operators:

The following operators are implemented in SStats:

+	Addition	-	Subtraction	%	Modulo
*	Multiplication	/	Division		
++	increment by 1	--	decrease by 1		
.*	Element-wise Multiplication			./	Element-wise Division
.+	Element-wise Addition			.-	Element-wise Subtraction
'	Matrix Transpose				
=	Value binding				
!	Negation	<	Less-than	>	Greater-than
==	Equal to	!=	Not equal to	>=	Greater or equal to
&&	And	||	Or	<=	Less than or equal to

2.7 Expressions:

Expressions are listed in sections in order of their precedence:

2.7.1 Primary Expressions
Primary expressions are evaluated left to right

2.7.1.1 Identifier (as defined in 2.2)
2.7.1.2 Constant
2.7.1.3 String
2.7.1.4 (expression)

Parentheses surrounding an expression elevates its order of precedence and has no other effect on type or value of the expression

2.7.1.5 primary-expression [expression]

Brackets following a primary expression is used for accessing elements within an array. The expression must evaluate to type int.

2.7.1.6 primary-expression (list-of-expression)

A primary expression followed by parentheses with an optional list of expressions is a funtion call.

2.7.2 Unary Operators

Unary operators are right-associative

2.7.2.1 ! expression

! applies the logical negation operator to a boolean expression

2.7.2.2 - expression

results in the negative of the expression and retains the same type otherwise. Expression must be of type int or float

2.7.2.3 expression ++

increments the expression by 1, type must be int or float

2.7.2.4 expression --

same as ++ but decrements by 1 instead

2.7.2.5 expression '

transposes expression. Expression must be of type array

2.7.3 Multiplicative Operators

Multiplicative operators are left-associative

2.7.3.1 expression * expression

Binary operator * indicates mathematical multiplication. Accepted data types are int and float and array. In the case where operands are int or float: if one of the expressions is of type float, the expression results in type float. Otherwise the result is the same type as the operands inputted. If the expressions are arrays, the operator * indicates matrix multiplication and the mathematical restrictions are applied here in addition to having both expressions be of type array.

2.7.3.2 expression / expression

/ indicates division, the same type rules for * apply here.

2.7.3.3 expression % expression

% indicates the modulo operation. Both expressions must be of type int and the result will be int.

2.7.3.4 expression .* expression

.* indicates element-wise multiplication. The first operand must be of type array and the second must be float or int. The result is another array that has had the operation * expression applied to every single element in the array.

2.7.3.5 expression ./ expression

./ indicates element-wise division. The same restrictions for .* apply here.

2.7.4 Additive Operators
Additive operators are left-associative

2.7.4.1 expression + expression

Binary operator + indicates addition. Accepted types are int, float and array. If one operator is float, the result will be float. Like with multiplication, if one operand is of type array the other also must be array. In this case, matrix addition is performed and thus the additional restriction of arrays being the same size is also required.

2.7.4.2 expression - expression

- indicates the difference between expressions. Same restrictions for + apply here

2.7.4.3 expression .+ expression

.+ indicates element-wise addition. The first operand must be of type array and the second must be float or int. The result is another array that has had the operation + expression applied to every single element in the array.

2.7.4.4 expression .- expression

.- indicates element-wise subtraction. The same restrictions for .+ apply here.

2.7.5 Relational Operators
2.7.5.1 expression < expression
2.7.5.2 expression <= expression
2.7.5.3 expression > expression
2.7.5.4 expression >= expression

The operators < (less than), > (greater than), <= (less than or equal to) and >= (greater than or equal to) all yield 0 if the specified relation is false and 1 if it is true.

2.7.6 Equality Operators
2.7.6.1 expression == expression
2.7.6.2 expression != expression

2.7.7 expression && expression

&& evaluates to 1 if both expressions evaluate to 1 and returns 0 otherwise

2.7.8 expression || expression

|| evaluates to 1 if either or both expressions evaluate to 1 and 0 otherwise

2.7.9 lvalue = expression

lvalue must be an identifier or an access of an element within an array specified by an indentifier.

The = operator replaces whatever value is referred by lvalue with the result of the expression

2.8 Declarations:
Declarations are used to define the type of an individual identifier.

declaration:

type-specifier declaration-option

2.8.1 Type specifier:
SStats has the following type specifiers:
int
float
boolean
char

2.8.2 Declaration options:

declaration-option:

[]

()

(expression)

2.8.2.1 []
Indicates that the declaration will be an array of type-specifier
2.8.2.2 ()
Indicates that the declaration is a function
2.8.2.3 (expression)
Indicates that the declaration is the result of a function called with expression

2.9 Statements:
2.9.1 Expression statement
Most statements will be expression statements which have the form

expression ;

2.9.2 Conditional
There are four forms of conditional statements:

if (expression) statement

if (expression) statement else statement

if (expression) statement elif (expression) statement

if (expression) statement elif (expression) statement else statement

The elif segment can be present an arbitrary number of times. The expressions are executed in order and the first non-zero triggers its corresponding statement. "else" is executed if all previous expressions in the statement resolve to 0.

2.9.3 While Loop

while (expression) statement

While the expression evalutes to a non-zero value the statement is repeatedly executed.

2.9.4 For Loop

for (expression-1; expression-2; expression-3) statement

This statement is equivalent to

expression-1;

while (expression-2) {

statement

expression-3;

}

2.9.5 Break

break;

Break terminates the smallest enclosing while or for statement. The statement following the terminated statement is then executed.

2.9.6 Continue

continue;

Continue causes control to go to the loop continuation portion of the local while or for statement.

2.9.7 Return

return (expression) ;

Return returns the value of expression to the caller of the function.

2.10 Scope Rules

Sstats is block scoped. That is variables exist within the context of the block they are initialized in. Most often the block will be contained within a function but can be extended to conditional and loop statements.

2.11 Program Structure

Sstats programs must include a main() function at the end of the program that will execute any containing statements.

FizzBuzz example:

```
function fizzbuzz(int n) {
	for(int i=0; i<n; i++) {
		if (i % 15 == 0) {
			print("FizzBuzz");
		}
		elif (i % 5 == 0) {
			print("Fizz");
		}
		elif (i % 3 == 0) {
			print("Buzz");
		}
		else {
			print(i);
		}
	}
}

function main() {
	fizzbuzz(100);
```

```
}
```