GASSP Programming Language Reference Manual

Adam Fowler (ajf2177), Patrycja Przewoznik (pap2154),
Samuel Weissmann (spw2136), Swan Htet (sh3969), Yuanxin Yang (yy3036)

February 24, 2021

Contents

1. Introduction

GASSP is a statically typed object-oriented general purpose programming language with its roots in C++ and Java. As an object-oriented language, GASSP does not attempt to needlessly reinvent the wheel, but rather it adopts an Umbridgean worldview: to preserve what must be preserved, perfect what can be perfected, and prune practices that ought to be prohibited. Through this philosophy we hope to embrace the strengths of the object-oriented paradigm without falling into the common trap of trying to turn a tool into a toolbox and while striving to shield the user from unintended consequences.

With GASSP, we aim to provide a simple and intuitive language free of syntactic ambiguity that does its best to limit the users' ability to create unpredictable or unintended behaviors. It features strong static typing, replaces null with the option type, does not perform type coercion, does not permit function overloading, and avoids undefined behaviors wherever possible.

2. Lexical Conventions

2.1 Comments

Multi-line comments are enclosed inside `/* */`. GASSP does not support single line comments.

```
/* This is a multi-
line comment. It's readable;
but not poetry */
```

2.2 Identifiers

GASSP recognizes ASCII symbols, and identifiers may be constructed from a combination ASCII letters, digits, and underscores. Identifiers must begin with a letter or underscore and cannot override keywords.

2.3 Keywords

Keywords are reserved in GASSP and they cannot be used as identifiers.
GASSP keywords include:

`if`	`string`	`false`	`array`
`else`	`pass`	`int`	`class`
`switch`	`while`	`float`	`inherits`
`case`	`break`	`bool`	`void`
`default`	`continue`	`char`	`return`
`for`	`true`		

2.4 Operators
GASSP features a standard set of built-in operators that can be broadly classified into four categories.

Arithmetic	`+ - * / % ** ++ --`
Comparison	`== != < <= > >=`
Logical	`&& || !`
Bitwise	`& | ^ << >> ~`

2.4.1 Arithmetic Operators:
Arithmetic operators can be used on integers and floats (with one exception, see 2.4.4). Operands must be of the same type.

2.4.2 Comparison Operators:
Comparison operators can be used on integers, floats, chars, and strings. The operands must be of the same type. Strings and chars are compared on their ASCII lexicographic order.

2.4.3 Logical Operators:
Logical operators can be used on booleans or expressions that evaluate to booleans.

2.4.4 String Operators
The + operator can be used for string concatenation. Both operands must be of type string.

2.4.5 Operator Precedence
From highest to lowest precedence.

Operator	Description	Associativity
`++` `--`	Increment Decrement	Right to left
`!` `~`	Logical not Bitwise not	Right to left
`**`	Exponentiation	Right to left
`* / %`	Multiplicative	Left to right

`+ -`
`+`
	Additive
	String concatenation
		Left to right

`>> <<`	Shifts	Left to right
`&`	Bitwise and	Left to right
`^`	Bitwise xor	Left to right
`|`	Bitwise or	Left to right
`== !=`	Equality	Left to right
`< > <= >=`	Comparison	Left to right
`&&`	Logical and	Left to right
`||`	Logical or	Left to right
`=`	Assignment	Right to left

2.5 Literals
Literals are constant values for one of GASSP's built-in types (i.e. a primitive or string).

2.5.1 Integer Literals
An integer literal is any sequence of one or more decimal digits. Integer literals may not begin with a 0.

2.5.2 Float Literals
A float literal is a sequence of one or more decimal digits followed by a decimal point (`.`) and another sequence of one more decimal digits.

2.5.3 Boolean Literals
Boolean literals are represented with the keywords `true` and `false`.

2.5.4 Char Literals
A char literal is any of the ASCII values enclosed in single quotation marks.

2.5.5 String Literals
A string literal is any sequence of chars enclosed in double quotation marks.

3. Data Types
All data in GASSP falls into one of two categories: primitives or objects. Data types are not inferred at runtime and need to be explicitly declared.

3.1 Primitives
GASSP has four primitive data types. It should be noted that unlike in some other common languages, the char data type can not be used as an integer value, and attempting to do so will result in an error.

Name	Size	Operators	Syntax
`int`	4 bytes	All	`int x = 3;`
`float`	8 bytes	All	`float x = 3.14;`
`boolean`	1 byte	Logical, Comparison	`bool x = true;`
`char`	1 byte	Comparison	`char x = 'x';`

3.2 Objects
All objects have an associated type and value with it. The value may be either a primitive or a reference to another object. GASSP includes strings and arrays as built-in objects. Object data and methods are accessed using familiar dot (.) notation as follows:

```
bankAcc.balance(); /* prints "0" */
bankAcc.deposit(500);
bankAcc.balance(); /* prints "500" */
```

3.2.1 Arrays
Arrays are fixed length and type defined. Arrays are mutable in the sense that their contents may be mutated or overridden, but their size and the type of data they hold is fixed.

```
int[n] int_arr; /* declares an n-length array of ints */
int[] int_arr = [1,2,3];
```

Arrays may be accessed using square bracket notation as follows:

```
int[] int_arr = [1,2,3];
int_arr[0]; /* returns 1 */
int_arr[1] = 10; /* array is now [1,10,3] */
```

3.2.2 Strings
Strings in GASSP are immutable, but can be accessed with square bracket notation similar to arrays.

```
string str1 = "hello";
string str2 = str1 + " world"; /* "hello world" */
string str3 = str2[4]; /* "o" */
```

Additionally, strings support `str.upper()` and `str.lower()` functions, where `str` is a string object and they return the same string in all uppercase or all lowercase letters, respectively.

3.2.3 User Defined Objects

GASSP allows users to define their own objects by means of a class system. GASSP also supports inheritance. The basis class syntax is as follows:

```
class Haiku inherits Poetry {

     /* instance variables,
     constructor, and other class
     methods belong here */
}
```

4. Statements

Statements are executed in the sequence they are written, except when otherwise stated. Statement scope is demarcated with curly braces. The different types of statements are defined in the following sections.

4.1 Expression Statements

Expression statements evaluate to a particular value. Expressions are terminated with a semicolon and take the form:

expression `;`

4.1.1 Operators

4.1.2 Operator Precedence

4.2 if and if-else Statements

The `if` statement is composed of a boolean expression, a statement that is executed if the boolean expression evaluates to true, and an optional `else` statement that is executed if the boolean expression evaluates to false. Mandatory curly braces define the scope of the statements, and else ambiguity is resolved by attaching an else statement to the most recently encountered elseless `if` of the same scope level. The statements take the following forms:

`if` *boolean expression* { *statement* }
`if` *boolean expression* { *statement* } `else` { *statement* }

4.3 Switch Statements

The `switch` statement is composed of an expression, one or more `case` statements, an optional `default` statement, and takes the following form:

`switch` *expression* { *statement(s)* }

The `case` statements are composed of an expression and a statement, while the `default` statement has only another statement. These statements take the following forms:

`case` *expression* { *statement* }
`default` { *statement* }

When executed, the `switch` statement evaluates its expression, compares that value to the value of the case expressions in the order that they appear, and passes control to the first `case` statement with a matching expression value, or the `default` statement if it is included and no matching value exists. If no statement can be executed, control resumes normally.

4.4 While Statements

The `while` statement is composed of an expression and statement in the following form:

`while` (*expression*) { *statement* }

The statement will be repeatedly executed as long as the expression evaluates to true.

4.5 For Statements

The `for` statement is composed of three expressions and a statement in the following form:

`for` (*expression-1* ; *expression-2* ; *expression-3*) { *statement* }

Expression-1 initializes the loop with a value; expression-2 specifies a condition that is checked before every iteration and that must evaluate to true for the loop to continue running; and expression-3 is evaluated at the end of each iteration, and is typically used to update a loop control variable.

4.6 Return Statements

The `return` statement returns control to the caller of a function, and optionally passes back the value of an expression. It has the following form:

`return ;`
`return` (*expression*) `;`

All functions in GASSP must include a return statement and void functions are not permitted.

5. Functions

5.1 User Defined Functions

Functions in GASSP are a type of statement that may take in zero or more arguments and executes. Functions may have a type or be of type “void”. Typed functions must return an expression that evaluates to their type, while void functions return nothing. Function type must be specified in function’s definition. Function arguments are specified as a list of typed parameters in the function definition. Functions are defined and called as follows:

```
int sumOfInt(int x, int y) {
     return (x+y);
}

gout(sumOfInt(1,4)); /* prints “5” */
```

5.2 Standard Library

GASSP standard library consists of a small number of built-in functions.

5.2.1 Mathematical Functions

GASSP has the following mathematical functions

Min and Max

The min and max functions take in two arguments of the same type and return the smaller value or larger value respectively. Arguments may be ints or doubles. If the arguments have the same value, the result is that same value.

```
int x = min(5, 9); /* x = 5 */
int y = max(5, 9); /* y = 9 */
```

sqrt(a)

Returns the positive square root of a double value. The argument can be integer or double type. For results that exceed the precision of a double, the number is truncated.

```
double x = sqrt(10); /* x = 3.16227766017 */
```

random()

Returns a double with a value in the range [0, 1).

```
double x = random(); /* x = 0.5568515217910215 */
```

5.2.2 Sorting and I/O Functions
GASSP includes the following sorting and I/O functions.

quicksort(a)
Takes in an array of comparable objects and sorts them in ascending order using the quicksort algorithm.

```
int[] a = [10,8,23,5];
int[] s = quicksort(a); /* s is now {5, 8, 10 ,23} */
```

gout(a)
Prints the argument to stdout. The argument can be of type integer, double, string and boolean.

```
int a = 5;
gout(a); /* prints 5*/
```

gin()
Takes the input from stdin and returns the input.

```
int a = gin(); /* a will have an integer value from stdin */
```

6. Examples
GCD implemented in GASSP

```
int gcd(int a, int b) {
    if (b == 0) {
        return a;
    }
    return(b, a%b);
}
```

Quicksort implemented in GASSP

```
/* initially low and high refer to index 0 and index n-1, where n
is the size of the array
*/

int partition (int[] a, int low, int high)
{
      int pivot = a[high];
      int i = (low-1);

      for (int j = low; j <= high-1; j++)
      {
            if (a[j] <= pivot)
            {
                  i++;
                  int temp = a[i];
                  a[i] = a[j];
                  a[j] = temp;
            }
      }
      int temp = a[i+1];
      a[i+1] = a[high];
      a[high] = temp;
      return (i+1);
}

void quickSort(int a[], int low, int high)
{
      if (low < high)
      {
            int split = partition(a, low, high);  /* split is the
partition index */
            quickSort(a, low, split-1);
            quickSort(a, split+1, high);
      }
}
```