

# Democritus Language Reference Manual

Amy Xu  
xx2152

Emily Pakulski  
enp2111

Amarto Rajaram  
aar2160

Kyle Lee  
kpl2111

March 4, 2016

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Data types</b>	<b>4</b>
2.1	Primitive Types . . . . .	4
	int . . . . .	4
	float . . . . .	4
	char . . . . .	4
	boolean . . . . .	4
	pointer . . . . .	4
2.2	Complex Types . . . . .	4
	Array . . . . .	4
<b>3</b>	<b>Lexical Conventions</b>	<b>5</b>
3.1	Identifiers . . . . .	5
3.2	Keywords . . . . .	5
3.3	Punctuation . . . . .	6
	; . . . . .	6
	{ and } . . . . .	6
	( and ) . . . . .	6
	Comments . . . . .	6
<b>4</b>	<b>Expressions and Operators</b>	<b>7</b>
4.1	Assignment . . . . .	7
4.2	Arithmetic Operations . . . . .	7
	Addition and Subtraction . . . . .	8
	Multiplication . . . . .	8
	Division . . . . .	8
	Modulus . . . . .	8
	Bit Shifting . . . . .	8
4.3	Boolean Expressions . . . . .	8
	Equality . . . . .	9
	Negation . . . . .	9
	Comparison . . . . .	9
	Chained Expressions . . . . .	9
4.4	Pointers and References . . . . .	9
4.5	Array access . . . . .	9
4.6	Operator Precedence and Associativity . . . . .	10

<b>5</b>	<b>Statements</b>	<b>11</b>
5.1	Expressions . . . . .	11
5.2	Declarations . . . . .	11
5.3	Control Flow . . . . .	11
	<b>if, elif, else</b> . . . . .	11
	Looping with <b>for</b> . . . . .	12
<b>6</b>	<b>Functions</b>	<b>13</b>
6.1	Overview . . . . .	13
6.2	Built-in Functions . . . . .	13
<b>7</b>	<b>Concurrency</b>	<b>15</b>

# Chapter 1

## Introduction

Democritus is a programming language with a static type system and native support for concurrent programming via its `atomic` keyword, with facilities for both imperative and functional programming. Democritus is compiled to the LLVM (Low Level Virtual Machine) intermediate form, which can then be optimized to machine-specific assembly code. Democritus' syntax draws inspiration from contemporary languages, aspiring to emulate Go and Python in terms of focusing on use cases familiar to the modern software engineer, readability, as well as having “one – and preferably only one – obvious way to do it”<sup>1</sup>.

---

<sup>1</sup><http://c2.com/cgi/wiki?PythonPhilosophy>

# Chapter 2

## Data types

### 2.1 Primitive Types

#### **int**

A standard 32-bit two's-complement signed integer. It can take any value in the inclusive range (-2147483648, 2147483647).

#### **float**

A 64-bit floating precision number, represented in the IEEE 754 format.

#### **char**

An 8-bit ASCII character.

#### **boolean**

A 1-bit boolean may take a true or false value.

#### **pointer**

An 64-bit pointer holds the value to a location in memory; they operate similarly to those found in C.

### 2.2 Complex Types

#### **Array**

A fixed-size array, allocated on the stack (thus requiring the size to be defined at declaration) of other primitive types. An array object can be accessed by C array notation, such as `list1[0]`.

## Chapter 3

# Lexical Conventions

In this section, we will cover the standard lexical conventions for Democritus. Similarly to languages such as C, Algol, or Pascal, Democritus is a free-format language. Thus, the parser will discard whitespace characters such as ' ', `\t`, and `\n`.

### 3.1 Identifiers

Identifiers for Democritus will be defined in the same way as they are in most other languages; any sequence of letters and numbers without whitespaces and is not a keyword will be parsed as an identifier. Note that, as in other languages, identifiers cannot begin with a number. Somewhat different, however, is the order of variable declarations; in Democritus, declarations are made following the *varname vartype* structure.

```
2wrongID int;    /* Not a valid identifier */  
mySecond float; /* Valid */  
my_Second char; /* Valid */
```

### 3.2 Keywords

The list of reserved keywords used in Democritus are as follows:

```
if  
else  
elif  
for  
return  
int  
float  
char  
boolean  
function  
void  
string  
true  
false  
break  
continue  
atomic
```

These words have been reserved by the compiler and hold special meaning within the language. Though most are self-explanatory, we will delve into their usage later on.

### **3.3 Punctuation**

**;**

Similarly to C, the semicolon ‘;’ is required to terminate any statement in Democritus.

**{ and }**

For the sake of keeping the language free-format, curly braces are used to delineate between separate and nested blocks. These braces are required even for single-statement conditional and iteration loops.

**( and )**

To assert precedence, expressions may be encapsulated within parentheses to guarantee order of operations.

#### **Comments**

For now, comments are initiated with `/*` and closed with `*/`. They cannot be nested.

## Chapter 4

# Expressions and Operators

An expression consists a combination of any of the following:

- a literal value
- a variable name
- a binary operation
- a unary operation
- array access

### 4.1 Assignment

Assignment is done with `=`. As mentioned above, variables are declared with the `varname vartype` syntax. Variables can be assigned to a single value or to the result of an expression.

```
x float = 4.0;
y int = 5/2 + 1; /* y = 3 */
```

Array assignment is done with Java-like syntax. Note that the size of the array must be specified in the declaration.

```
x int[5] = {0, 1, 2, 3, 4};
```

Pointer types are denoted with a `*` which must be attached to the primitive type that they reference.

```
x int = 4; /* x = 4 */
y int* = &x; /* *y = 4 */
```

### 4.2 Arithmetic Operations

Democritus supports all the arithmetic operations standard to most general-purpose languages like C and Java. Note that casting is not built into the language; this functionality will instead be implemented through the standard library.



## Addition and Subtraction

Addition works with the + character, behaving as expected.

```
x int = 4;
y int = 2;
x = x+y;    /* x = 6 */
y = y-x     /* y = -4 */
```

## Multiplication

Multiplication follows the same rules as well.

```
x int = 4;
y int = 2;
x = x*y+y;  /* x = 10 */
```

## Division

Democritus will default to integer division, unless both types provided are floats.

```
x int = 5;
y int = 2;
x = x/y;    /* x = 2 */

a float = 4.0;
b float = 2.0;
a = b/a;    /* a = 2.0 approximately */
```

## Modulus

The remainder of an integer division operation can be accessed via the modulo % operator.

```
x int = 8;
y int = 5;
x = x%y;    /* x = 3 */
```

## Bit Shifting

Integers can be bit-shifted with >> and <<.

```
x int = 9;
y int = x>>1;  /* y = 4 */
x = y<<2;      /* x = 16 */
```

## 4.3 Boolean Expressions

Democritus features all of the standard logical operators, following Java-style syntax. Each expression will return a boolean value of true or false.

## Equality

Equality is tested with the `==` operator. Inequality is tested with `!=`.

```
x int = 8;
y int = 8;
z boolean = (x==y);    /* z = true */
z = (x==(y+1));         /* z = false */
z = (x!=(y+1));         /* z = true */
```

## Negation

Negation is done with `!`, a unary operation.

## Comparison

Democritus also features the `<`, `<=`, `>`, and `>=` operators.

```
x int = 9;
y int = 8;
z int = 8;

x>y;    /* true */
y>=z;   /* true */
z<x;    /* true */
```

## Chained Expressions

Boolean expressions can be chained with `&&` and `||`, representing **and** and **or**, respectively. These operators have lower precedence than any of the other boolean operators described above. The **and** operator has a higher precedence than **or**.

```
x int = 9;
y int = 8;
z int = 8;

(x>y && y<x);    /* false */
(x>y || y<x);    /* true */
(x>y && y<x || z==y) /* true */
```

## 4.4 Pointers and References

Pointers and dereferencing utilizes the same syntax as C. The unary operator `&` gives a variables address in memory, and the operator `*` dereferences a pointer. See the assignment section for usage.

## 4.5 Array access

Array access is done with `[i]` where *i* is the index being accessed.

```
x int[5] = {0,1,2,3,4};
y int = x[2];    /* y = 2 */
```

## 4.6 Operator Precedence and Associativity

Precedence	Operator	Description	Associativity
1	()	Parenthesis	Left-to-right
2	() { } [ ]	Function call Array creation Array subscript	Left-to-right
3	* & ! -	Dereference Address-of Negation Unary minus	Right-to-left
4	* / %	Multiplication Division Modulo	Left-to-right
5	+ -	Addition Subtraction	Left-to-right
6	>> <<	Bitwise shift right Bitwise shift left	Left-to-right
7	>> = << =	For relational > and ≥ respectively For relational < and ≤ respectively	Left-to-right
8	== !=	For relational = and ≠ respectively	Left-to-right
9	&&	Logical and	Left-to-right
10		Logical or	Left-to-right
11	=	Assignment	Right-to-left

# Chapter 5

## Statements

### 5.1 Expressions

An expression statement consists of an expression followed by a semicolon. Expressions in expression statements will be evaluated, and its value calculated.

```
a int = 500;
s char = 'a';
2+4-3;          /* Not used, thrown away */
```

### 5.2 Declarations

A declaration specifies a variable's name and type, in that order. Values may also be initialized in the declaration

```
x int;
y char = '4';
```

### 5.3 Control Flow

**if**, **elif**, **else**

An **if** statement causes a block (encapsulated by { and }) to be entered if the specified condition evaluates to true.

An **elif** allows an alternate condition to be specified.

An **else** is entered if the 'if' and 'elif's are not entered.

A boolean expression encapsulated within parentheses is required for every **if** and **elif**. **Elif** and **else** belong to the first preceding **if** statement.

```
x int = 1;
if (x == 1)
{
    print( x = 1! );
}
```

```

elif (x == 2)
{
    print(  x  = 2!  );
}

else
{
    print(  f a i l  );
}

```

## Looping with for

Democritus eliminates the **while** structure, replacing it instead with a modified **for** loop. For can be used to iterate by providing an initialization, termination condition, and update:

```

for(i int = 0; i < 10; i++)
{
    /* Some code here */
}

```

It can also be used as a while loop providing only one condition:

```

for(x<10)
{
    /* Some code here */
}

```

# Chapter 6

## Functions

### 6.1 Overview

Functions can be defined in Democritus to return one or no data type. Functions are evaluated via eager evaluation and the function implementation must directly follow the function header.

A function appears in the form:

```
function [function name]([type:formal_arg, ... ]):[return type]{  
    [function implementation]  
    return [variable of return type]  
}
```

**Note:** all functions need **return** statements at the end (no falling off the end). A void **return** is simply a return with nothing following it.

Functions may be recursive and call themselves:

```
function recursive_func(i:int):void{  
    if(i < 0){  
        return;  
    }else{  
        print    h i    ;  
        recursive_func(i-1);  
    }  
}
```

Functions may be called within other functions:

```
function main():void{  
    recursive(3);  
    return;  
}
```

### 6.2 Built-in Functions

A handful of functions are natively built into Democritus for user flexibility and ease of usage. There are:

- `print(s:string)`: takes in a string (standard library functions will convert from other data types to strings)
- `thread(f:function, [arg1:type, arg2:type, ...])`: takes in function and function args

## Chapter 7

# Concurrency