Democritus Language Reference Manual

Amy Xu	Emily Pakulski	Amarto Rajaram	Kyle Lee
xx2152	enp2111	aar2160	kpl2111

March 6, 2016

Contents

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, emphasizing readability, and having "one – and preferably only one – obvious way to do it"[1].

[1] 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, represnted in the IEEE 754 format.

char

An 8-bit ASCII character. We include the extended ASCII set, so we use all 256 possible values.

boolean

A 1-bit true or false value.

pointer

A 64-bit pointer that holds the value to a location in memory; very similar to those found in C.

2.2 Complex Types

Array

A fixed-size array, allocated on the stack and containing other primitive types. The size must be defined at declaration. An array object can be accessed by C array notation, such as `list1[0]`.

string

An immutable array of characters, implemented as a native data type in Democritus.

struct

A struct is a simple user-defined data structure that holds various primitives, similar to the ones found in C.

Chapter 3

Lexical Conventions

In this section, we will cover the standard lexical conventions for Democritus. As with languages such as C, Algol, or Pascal, Democritus is a free-format language. 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 declaration */
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

;

As in C, the semicolon ';' is required to terminate any statement in Democritus.

{ and }

In order to keep the language free-format, curly braces are used to delineate 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 of a combination of any of the following:

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

4.1 Assignment

Assignment is done with `=`. As mentioned above, variables and 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 computed 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 operations utilize 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 shift right Bitwise shift left	Left-to-right
7	>> = <<=	For relational $>$ and $\geq$ respectively For relational $<$ and $\leq$ respectively	Left-to-right
8	`== !=`	For relational $=$ and $\neq$ 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 preceeding `if` statement.

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

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

else
{
    print("fail");
}
```

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. The syntax is reminiscent of Scala, although Democritus doesn't support implied returns. A function appears in the form:

```
function [function name]([formal_arg type, ... ]) 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_fun(i-1);
    }
}
```

Functions may be called within other functions:

```
function main() void{
    recursive_func(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

7.1 Overview

Democritus intends to cater to modern software engineering use cases. Developments in the field are steering us more and more towards highly concurrent programming as the scale at which software is used trends upward.

With this in mind, Democritus adds support for the `atomic` keyword, used as a modifier at the declaration step. The keyword can be used both in an inline declaration and when declaring a function's types. We also wrap the $pthread_t$ datatype and related functions.

7.2 Atomic Inline Declarations

Under the hood, declaring a variable with the `atomic` keyword embeds a locking structure into the type, as well as exposing the `lock()` and `unlock()` functions. If the keyword is used with a standard data type, the compiler replaces the normal version of that type with a version that includes the lock and the functions described above.

```
  }
    x int;
    x.lock(); x.unlock(); /* undefined! */

    y atomic int;
    y.lock(); y.unlock(); /* defined! */
}
```

7.3 Atomic Parameter Declarations

A function whose formal parameters are `atomic` will throw a compile-time error if a non-atomic type is passed in. The idea is to force the programmer to document which functions are safe to use in a multi-threaded context and which are not.

```
  }
function [function name]([formal_arg atomic type]) atomic type {
  formal_arg.lock();
  /* do something */
  formal_arg.unlock();
```

```
  return formaL_arg
}
```

Naturally, rather than calling `lock()` and `unlock()` manually, the programmer can implement atomic operations.

7.4 Spawning Threads

To spawn threads, Democritus uses a wrapper around the C-language `pthread` family of functions.

The $thread_t$ data type wraps $pthread_t$.

To spawn a thread, the thread function takes a variable number of arguments where the first argument is a function and the remaining optional arguments are the arguments for that function. It returns an error code.

The `detach` boolean determines whether or not the parent thread will be able to join on the thread or not.

```
  }
thread(f function, boolean detach, [arg1 type, arg2 type, ...]) int;
  }
```