Democritus Language Final Report

Amy Xu	Emily Pakulski	Amarto Rajaram	Kyle Lee
xx2152	enp2111	aar2160	kpl2111
Project Manager	Language Guru	System Arhictect	Tester

May 8, 2016

Contents

1 Introduction **4**
1.1 Motivation . . . 4
1.2 Goals . . . 4

2 Language Tutorial **5**
2.1 Setup and Installation . . . 5
2.2 Compiling Your Code . . . 6
2.3 Writing Code . . . 6
2.4 Getting Started . . . 6
Declarations . . . 6
Types . . . 6
Primitives . . . 6
Structs . . . 7
Operators . . . 7
Binary Operators: . . . 7
Unary Operators: . . . 7
2.5 Control Flow . . . 7
Conditional Branching . . . 7
Loops and Iteration . . . 8
2.6 Multithreading and Atomicity . . . 9

3 Language Reference Manual **10**
3.1 Data types . . . 10
Primitive Types . . . 10
int . . . 10
boolean . . . 10
Complex Types . . . 10
string . . . 10
struct . . . 10
3.2 Lexical Conventions . . . 10
Identifiers . . . 10
Literals . . . 11
Tokens . . . 11
Punctuation . . . 12
Semicolon . . . 12
Curly Brackets . . . 12
Parentheses . . . 12
Comments . . . 12
3.3 Expressions and Operators . . . 12
Declaration and Assignment . . . 12

Arithmetic Operations . . . 13
Addition and Subtraction . . . 13
Multiplication . . . 13
Division . . . 13
Modulus . . . 13
Bit Shifting . . . 14
Boolean Expressions . . . 14
Equality . . . 14
Negation . . . 14
Comparison . . . 14
Chained Expressions . . . 14
Pointers and References . . . 15
Array access . . . 15
Operator Precedence and Associativity . . . 15
3.4 Statements . . . 15
Expressions . . . 15
Declarations . . . 15
Control Flow . . . 16
`if, elif, else` . . . 16
Looping with `for` . . . 16
3.5 Functions . . . 17
Overview . . . 17
Built-in Functions . . . 17
3.6 Concurrency . . . 17
Overview . . . 17
Atomic Inline Declarations . . . 18
Atomic Parameter Declarations . . . 18
Spawning Threads . . . 18

4 Project Plan **19**
4.1 Workflow . . . 19
4.2 Development Process . . . 19
4.3 Git Logs . . . 19

5 Architecture Overview **20**
5.1 Compiler Overview . . . 20
The Scanner . . . 20
The Parser . . . 20
The Semantic Analyzer . . . 20
The Code Generator . . . 20
5.2 C Bindings . . . 20
`pthread()` . . . 20
`sockets` API . . . 20
File I/O . . . 20

6 Testing **21**
6.1 Integration Testing . . . 21
Aside: Unit Testing . . . 21
6.2 The Test Suite and Automated Regression Testing . . . 21

7 Lessons Learned **22**
7.1 Amy 22
7.2 Emily 22
7.3 Amarto 22
7.4 Kyle 22

8 Code Listing **23**

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].

1.1 Motivation

1.2 Goals

[1] http://c2.com/cgi/wiki?PythonPhilosophy

2. Language Tutorial

Democritus is a strongly-typed, imperative language with standard methods for conditional blocks, iteration, variable assignment, and expression evaluation. In this chapter, we will cover environment configuration as well as utilizing Democritus' basic and more advanced features.

2.1 Setup and Installation

To set up the Democritus compiler, OCaml and LLVM must be installed. Testing and development was done in both native Ubuntu 15.04 and Ubuntu 14.04 running on a virtual machine.

```
sudo apt-get install m4 clang-3.7 clang-3.7-doc libclang-common-3.7-dev
   libclang-3.7-dev libclang1-3.7 libclang1-3.7-dbg libllvm-3.7-ocaml-dev
   libllvm3.7 libllvm3.7-dbg lldb-3.7 llvm-3.7 llvm-3.7-dev llvm-3.7-doc llvm
   -3.7-examples llvm-3.7-runtime clang-modernize-3.7 clang-format-3.7 python
   -clang-3.7 lldb-3.7-dev liblldb-3.7-dbg opam llvm-runtime
```

For Ubuntu 15.04, we need the matching LLVM 3.6 OCaml Library.

```
sudo apt-get install -y ocaml m4 llvm opam
opam init
opam install llvm.3.6 ocamlfind
eval `opam config env`
```

For Ubuntu 14.04:

```
sudo apt-get install m4 llvm software-properties-common

sudo add-apt-repository --yes ppa:avsm/ppa
sudo apt-get update -qq
sudo apt-get install -y opam
opam init

eval `opam config env`

opam install llvm.3.4 ocamlfind
```

After setting up the environment, clone the git repository into your desired installation directory:

```
git clone https://github.com/DemocritusLang/Democritus.git
```

2.2 Compiling Your Code

To build the compiler, `cd` into the Democritus repository, and run `make`.

If building fails, try running `eval ‘opam config env‘`, which should update your local environment use OPAM packages and compilers. It's recommended to add the above command to your shell's configuration file if you plan on developing with Democritus.

To compile code, simply run

```
./Democritus < filename.demo > outfile.lli
```

To run compiled code, call `lli` on the output:

```
lli outfile.lli
```

2.3 Writing Code

Code can be written in any text file, but Democritus source files should have the `.demo` extension by convention. Democritus programs consist of global function, struct, and variable declarations. Only the code inside main() will be exectued at runtime. At this time, linking is not included in the Democritus compiler; all code should be written and compiled from a single `.demo` source file.

2.4 Getting Started

Declarations

Functions are declared with the `<function func_name(a type, b type) return_type>` syntax. Variables are declared with the `<let var_name var_type;>` syntax. Statements are terminated with the semicolon `;`. Note that all variable declarations must happen before statements (including assignments) in any given function.

```
function triangle_area(base int, height int) int{
  return base*height/2;
}
```

Types

Primitives

Primitive types in Democritus include booleans, integers, and strings. The void type is also used for functions.

```
function main() int{
  let s string;
  let foo int;
  let bar bool;

  bar = true;
  s = "Hello, World!"
  foo = 55;
  bar = false;

  return 0;
}
```

Structs

Structs are declared at the global level with the `<struct struct_type { named fields }>` syntax.

```
struct Person{
  let education string;
  let name string;
  let age int;
  let working bool;
}


function main() int{
  let p Struct person;
  p.name = "Joe"
  p.education = "Bachelor's";
  p.age = 25;
  p.working = false;

  return 0;
}
```

Operators

Democritus includes the 'standard' set of operators, defined as follows:

Binary Operators:

- artithmetic: `+, -, *, /`
- logical: `==, !=, <, <=, >, >=, && (and), || (or)`

Unary Operators:

- artihmetic: `-`
- logical: `! (not)`

Logical expressions return a boolean value.

The expressions on each side of a binary operation must be of the same type. The `&&, ||,` and `!` operators must be called on boolean expressions.

2.5 Control Flow

As an imperative language, Democritus executes statements sequentially from the top of any given function to the bottom. Branching and iteration is done similarly to many other imperative languages.

Conditional Branching

Conditional branching is done with:

```
if(boolean expression)
{
  /* do something here */
}

else
{
  /* do alternative here */
}
```

Here is an example of conditional branching in Democritus:

```
struct Person{
  let education string;
  let name string;
  let age int;
  let working bool;
}


function main() int{
  let p Struct person;
  p.name = "Joe"
  p.education = "Bachelor's";
  p.age = 25;
  p.working = false;

  if(p.working){
    print(p.name);
    print(" works.\n");
  }else{
    print(p.name);
    print(" is looking for work.\n");
  }

  return 0;
}
```

Loops and Iteration

Iteration can be done either via a `while` or `for` loop. A `while(e1)` loop requires `e1` to be boolean conditional statement. A `for(e1; e2; e3)` loop requires three expressions; `e1` is called prior to entering the loop, `e2` is a boolean conditional statement for the loop, and `e3` is called after each iteration. Both `e1` and `e3` may be empty expressions. Each of the following functions should print 42.

```
function main() int{
  let i int;
  i = 0;
  while(i<42){
    i = i+1;
  }
  print_int(i)
```

```
  return 0;
}
```

```
function main() int{
  let i int;
  for(i = 0; i<42; i++){
    i = i+1;
  }
  print_int(i)
  return 0;
}
```

2.6 Multithreading and Atomicity

3. Language Reference Manual

3.1 Data types

Primitive Types

int

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

boolean

A 1-bit true or false value.

Complex Types

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.

3.2 Lexical Conventions

In this subsection, 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`.

Identifiers

Identifiers for Democritus will be defined as follows: 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. The regular expression defining identifiers is as follows:

```
['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']*
```

An example of declarations with identifiers:

```
2wrongID int;    /* not a valid identifier declaration */
mySecond float;    /* valid */
my_Second char;    /* valid */
```

Literals

Literals, simply a sequence of numbers, may be identified with the regular expression

```
['0'-'9']+           (* Int *)
['0'-'9']*'.'['0'-'9']+    (* Float *)
```

Tokens

The list of tokens used in Democritus are as follows:

```
| '('      { LPAREN }
| ')'      { RPAREN }
| '{'      { LBRACE }
| '}'      { RBRACE }
| ';'      { SEMI }
| ':'      { COLON }
| ','      { COMMA }
| '+'      { PLUS }
| '-'      { MINUS }
| '*'      { TIMES }
| '%'      { MOD }
| ">>"     { RSHIFT }
| "<<"     { LSHIFT }
| '/'      { DIVIDE }
| '='      { ASSIGN }
| "=="     { EQ }
| "!="     { NEQ }
| '<'      { LT }
| "<="     { LEQ }
| ">"      { GT }
| ">="     { GEQ }
| "&&"     { AND }
| "||"     { OR }
| "!"      { NOT }
| "if"     { IF }
| "else"   { ELSE }
| "elif"   { ELIF }
| "for"    { FOR }
| "return" { RETURN }
| "int"    { INT }
| "float"  { FLOAT }
| "char"   { CHAR }
| "boolean"   { BOOLEAN }
| '*' { POINTER }
| '&' { AMPERSAND }
| "function" { FUNCTION }
| "void"   { VOID }
| "struct" { STRUCT }
| "string" { STRING }
| "true"   { TRUE }
| "false"  { FALSE }
| "break"  { BREAK }
```

```
| "continue" { CONTINUE }
| "atomic" { 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.

Punctuation

Semicolon

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

```
statement SEMI
```

Curly Brackets

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.

```
LBRACE statements RBRACE
```

Parentheses

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

```
LPAREN expression RPAREN
```

Comments

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

3.3 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
- an array access operation
- a struct access

Declaration and Assignment

Democritus requires all named variables to be declared with its type at the top of each function. Named variables are declared with the `let [ID] type` syntax. Assignment to these variables may be done with `=`.

```
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 */
```

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 */
```

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 */
```

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 subsection for usage.

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 */
```

Operator Precedence and Associativity

Precedence	Operator	Description	Associativity
1	`()`	Parenthesis	Left-to-right

2	`()`
	`{}`
	`[]`
		Function call
		Array creation s
		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

3.4 Statements

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 */
```

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';
```

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 */
}
```

3.5 Functions

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;
}
```

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

3.6 Concurrency

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.

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! */
}
```

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.

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;
  }
```

4. Project Plan

4.1 Workflow

4.2 Development Process

4.3 Git Logs

5. Architecture Overview

5.1 Compiler Overview

The Scanner

The Parser

The Semantic Analyzer

The Code Generator

5.2 C Bindings

`pthread()`

`sockets` API

File I/O

6. Testing

6.1 Integration Testing

Aside: Unit Testing

6.2 The Test Suite and Automated Regression Testing

7. Lessons Learned

7.1 Amy

7.2 Emily

7.3 Amarto

7.4 Kyle

8. Code Listing