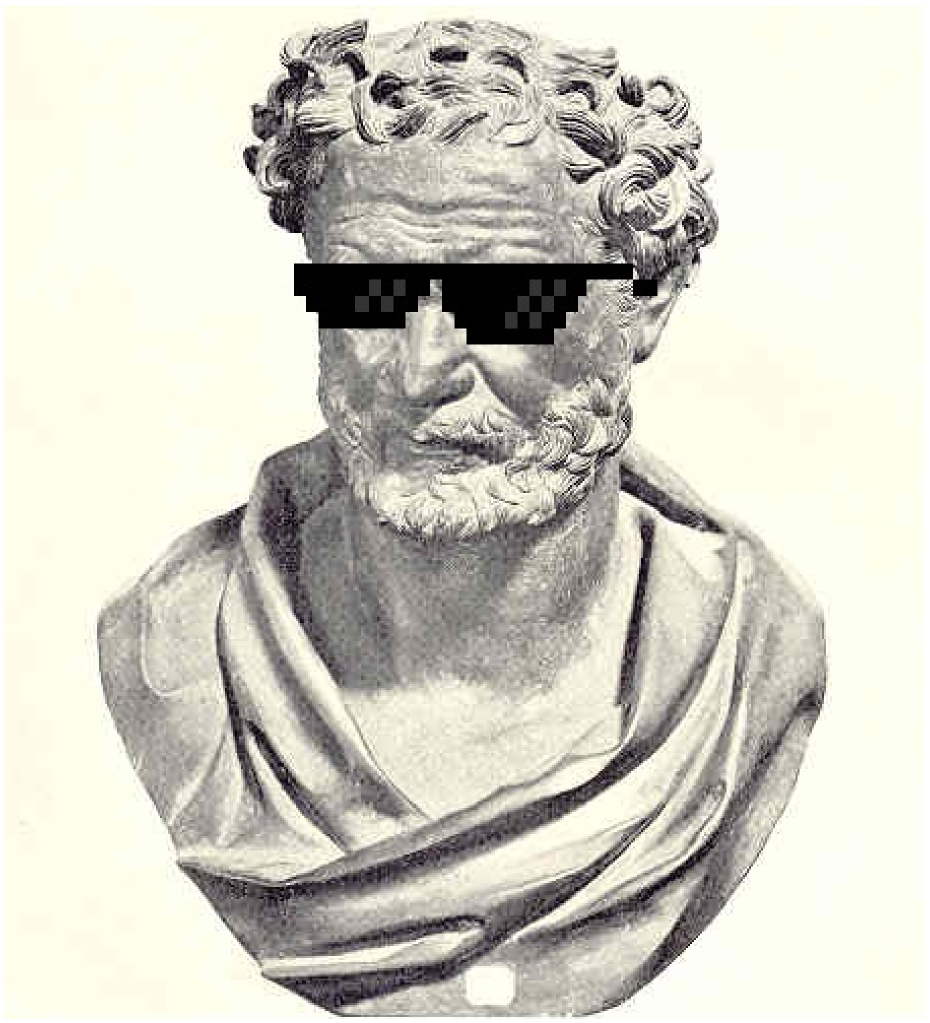

Democritus: An Atomic Language

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

February 10, 2016

1 Introduction

Named after the father of atomic theory, Democritus is a programming language with a static type system and added support for concurrent programming, with facilities for both imperative and functional programming. Democritus is compiled to the LLVM (Low Level Virtual Machine) intermediate form, which is then converted to optimized machine code. Democritus' syntax draws inspiration from contemporary languages, aspiring to emulate Go and Python in terms of focusing on excelling in use cases familiar to the modern software engineer[1] as well as emphasizing readability and having "one – and preferably only one – obvious way to do it"[2]. Programs written by large teams that leverage concurrent programming paradigms will benefit from the `atomic` keyword that Democritus offers as well as the unambiguous syntax conventions.

1.1 Why Democritus?

Democritus offers a low-level, performance-oriented language with strong support for concurrent programming and elegant, efficient syntax. It is ideal for concurrency-oriented applications such as transactional software and task-scheduling. The `atomic` keyword allows a programmer to create methods that abstract away locking primitives from others calling methods that require an atomic parameter.

Finally, the simple and intuitive syntax allows large teams to easily unify their coding style. Ideally, Democritus would have a linter that ships with the language, reminiscent of Golang's `fmt`.

Democritus is compiled down to the LLVM intermediate representation (IR) which then produces optimized programs for several architectures (such as ARM, PowerPC, x86, etc). As a result, a Democritus programmer is relieved from worrying about portability.

2 Feature Specification and Usage

Democritus attempts to stay low-level, while providing as many common-sense or overly tedious functions for free. The features we intend to implement are as follows:

2.1 Concurrency support

1. `atomic` keyword
   Democritus attempts to make concurrent programming easier by allowing users to instantiate objects with an atomic keyword. Under the hood, this initializes the struct with a mutex built into it, and defines the functions `lock()` and `unlock()` as part of this struct (see below).

   Functions that take an atomic parameter are those that will lock on the variable. Atomic variables can't be passed to functions that don't require atomic variables, as this means mixing thread-safe code with non-thread-safe code.

   Ideally, we would support a compiler flag that ignores all locks, so that Democritus code could be compiled to run on a uniprocessor system.

2. `lock():int, unlock():int`

   These functions are built into any instantiated atomic type. They lock and unlock on the mutex built into the type.

[1]https://golang.org/doc/faq#Origins
[2]http://c2.com/cgi/wiki?PythonPhilosophy

3. `function thread(f:function, ...):int`

   The `thread()` function call allows the current process to spawn a new thread. The new thread starts execution at the function passed by thread. If specified, the remaining parameters are passed to the function.

To keep the scope of the language minimal, we'd like to start with locks and not broach semaphores and other more complex synchronization primitives.

2.2 Control flow

1. `for`

   Democritus takes inspiration from Go syntax for its loop notation. Instead of `while`, loops are notated with `for`. This is because many users are familiar with the 3-part `for` syntax and that notation can encompass the `while` notation, enforcing uniformity in style across the language.

2. `if, elif, else`

   Democritus denotes its conditionals control flow structure with `if`, `elif`, and `else`. We chose to use `elif` instead of `else if` because it is less ambiguous to have one word instead of a combination of two words that each have other meanings.

3. `break, continue`

   While there's some controversy over when to use these constructs, common wisdom is that there is a time and place for `break` and `continue`. Furthermore, they map well to our LLVM IR target.

2.3 Memory management

1. `function new(val:type T):int`

   The `new()` function is akin to a `malloc()` in C with a built in call to `sizeof()`. In other words, if T is a typedefined struct, `new(T)` will allocate enough memory for a T struct on the heap.

2. `Function delete(val:type T): int`

   `delete()` is the inverse of `new()`. It deallocates the memory allocated in `new()`.

In these functions, the compiler determines the size of type T by looking up the type of the value passed in. These function declarations are not intended to suggest that Democritus supports generic programming (it does not).

2.4 Primitive data types

1. `int`: basic integer data type. No decimal.
2. `float`: basic decimal data type.
3. `char`: ascii character.
4. `boolean`: true or false.
5. `pointer`: integer that is treated as a memory address.
6. `function`: takes a set of inputs and returns an output.

2.5 Complex datatypes

1. `list` (array)
2. `struct`
3. `string`

Structs and lists are common to have as a default data type. We decided to include a built-in string data type because we believe that strings are so frequently used that it?s worth prioritizing support for clean string syntax.

2.6 I/O

1. `function read(fd:int,s:string,n:int):int`

   Read `n` bytes from the file `fd` into the buffer. Return number of bytes read.
2. `function write(fd:int,s:string,n:int):int`

   Write `n` bytes from `fd` into the buffer. Return number of bytes written.

2.7 Use cases

The atomicity of Democritus can be useful in several scenarios such as:

- Database Transactions

  An airplane company wants to sell the seats to a flight online. This may lead to problems as several customers may be browsing and initiating transactions on the same seat at the same time. Due to these concurrent actions, the company must ensure that a transaction locks so that only one customer pays and receives the seat while the other customers neither pay nor receive the seat.
- Dealing with repeats and loops (see code snippet)

  We want to record all web pages by recursively visiting a page and accessing its hyperlinks. Since hyperlink repeats and loops occur often, an efficient way of traversing through the Internet and recording unique pages would be to lock the set we are writing the records to.
- File locking

  A forum allows users to edit their posts. However, it needs to prevent users from editing posts while it does some routine maintenance. The forum needs to lock posts during its maintenance so that users are not able to submit edits even if they had already entered the editing state at the time the maintenance began.

2.8 Code samples

`web_crawler.dem` demonstrates Democritus' ability to easily handle concurrency issues and multi-threading. It creates an atomic set of strings, and for every link on a page it finds that is unvisited, it adds it to the set and spins off a new thread where it recursively follows the link. The set is by necessity atomic in order to prevent concurrency issues from multiple threads reading from and writing to the set at the same time.

```
web_crawler.dem
/* Assume this function parses an HTML document and returns
all the valid links in it as a list of strings. */
function get_links(link: string): [string] = {
  /* ... */
}

/* This function demonstrates the use of an atomic standard library data
type and how to create new threads. */
function handle_link(s: atomic set, root: string): void = {
  links = get_links(root);

  for each root in links {
    s.lock();
    if root not in s {
      /* add this string to set. add() is part of STL, and
since s is of an atomic type, it locks. */
      add(s, root);
      s.unlock();
      /* new thread with this function and parameters. */
      thread(handle_link, s, link);
    } else {
      s.unlock();
    }
  }
}


function discover_all_links(root: string) [atomic set] = {
  atomic set s = new(atomic set);
  handle_link(s, root);
  return s;
}
```

`stock_transactions.dem` uses a variant on the canonical bank account example for concurrency issues to demonstrate the syntactic ease of the Democritus language. The `buy_stocks()` function allows a user to purchase stocks from a market, which uses an atomic map to represent its inventory. The map must be atomic to prevent multiple users from carrying out transactions at the same time, artificially depleting the inventory beyond its capacity.

```
stock_transactions.dem
/* ... */
function buy_stocks(market: atomic map, user_portfolio:
 atomic map<string, int>, number: int, s: string): void = {
  if market.get(s) < number {
    except   Stock unavailable!   ;
  }else {
    market.put(s, market.get(s) - number);
    user_portfolio.put(s, user_portfolio.get(s) + number);
  }
}
/* ... */
```