Asteroid

Release 1.1.4

Lutz Hamel, Tim Colaneri, Oliver McLaughlin, Ariel Finkel, Theodo

Jan 05, 2023

CONTENTS

Asteroid is an open source, dynamically typed, multi-paradigm programming language heavily influenced by Python, Rust, ML, and Prolog currently under development at the University of Rhode Island. Asteroid implements a new programming paradigm called pattern-matching oriented programming. In this new programming paradigm patterns and pattern matching are supported by all major programming language constructs making programs succinct and robust. Furthermore, patterns themselves are first-class citizens and as such can be passed to and returned from functions as well as manipulated computationally.

CHAPTER

ONE

CONTENTS

1.1 Installation

Asteroid is available from the PyPI project website pypi.org/project/asteroid-lang and is installed using:

```
$ pip install asteroid-lang
```

The should work on Unix-like and Windows systems, though you may have to use *pip3* or some other variation.

> Don't forget to add the *bin* directory where *pip* installs programs to your *PATH* variable.

In addition to installing Asteroid directly on your machine, there is also a cloud-based Linux virtual machine that is completely set up with an Asteroid environment and can be accessed at Repl.it.

1.2 Running the Asteroid Interpreter

You can now run the interpreter from the command line by simply typing *asteroid*. This will work on both Windows and Unix-like systems as long as you followed the instructions above. To run asteroid on Unix-like systems and on our virtual machine,

```
$ cat hello.ast
-- the obligatory hello world program

load system io.

io @println "Hello, World!".

$ asteroid hello.ast
Hello, World!
$
```

On Windows 10 the same thing looks like this,

```
C:\> type hello.ast
-- the obligatory hello world program

load system io.

io @println "Hello, World!".
```

(continues on next page)

(continued from previous page)

```
C:\> asteroid hello.ast
Hello, World!
C:\>
```

As you can see, once you have Asteroid installed on your system you can execute an Asteroid program by typing:

```
asteroid [flags] <program file>
```

at the command prompt. Asteroid also supports an interactive mode:

```
$ asteroid
Asteroid Version 1.1.3
Type "asteroid -h" for help
Press CTRL-D to exit
ast> load system io.
ast> io @println "Hello, World!".
Hello, World!
ast>
```

1.3 Welcome To Asteroid!

Thank you for visiting our page about Asteroid! If you have gone to the Asteroid Documentation then you have seen that Asteroid is a pattern-matching oriented language. If you have not heard of that programming paradigm (type of programming language) before, Asteroid is one of the first of its kind. Here we provide a brief introduction to Asteroid geared towards C++ programmers.

1.3.1 Pattern Matching at the Core of Things

Pattern-matching is the idea of extracting values from a structure by applying a pattern to that structure. If the pattern contains variables then these variables will be instantiated with values from the structure during a successful match. Consider the structure `(1,2)`. If we apply the pattern `(1,x)` to that structure then the `1` will be matched and the variable `x` will be instantiated with the value `2`. The interesting part about pattern matching is that they can also fail to match. Consider again our structure `(1,2)` but now we want to apply the pattern `(y,1)`. This pattern match will fail because the second component of our structure and the pattern do not match. Because of this mismatch the variable `y` will also not be instantiated. Perhaps the simplest pattern match is between a value and a variable as a pattern. In that case the variable is simply instantiated with the value.

A very interesting aspect of pattern matching is that it provides a powerful way to inspect the values passed to a function. Consider the C++ function `f` defined here,

```
void f(int x) {
  if (x == 0)
    std::cout << "zero\n";
  else
    std::cout << "not zero\n";
}
```

If the integer value passed to the function is equal to zero then it prints out the word `zero`, otherwise it prints out the words `not zero`. Here the programmer has to use conditionals in order to decide which output to produce. Using conditionals is not a bad practice, however, functions that use pattern tend to look less cluttered.

Here is the same function written in Asteroid using pattern matching,

```
load system io.

function f
   with 0 do
      io @println("zero").
   with x do
      io @println("not zero").
   end
```

Here the value `0` and the variable `x` appearing right after the `with` keywords are patterns that are applied to the function value. If the function is called with the value `0` then the value will be matched by the first pattern and the associated print statement is executed. If the value with which the function was called is anything but `0` then the second pattern will match and that associated print statement will be executed. A variable can be considered a pattern that will match any value (unless other conditions have been placed on that variable).

The interesting part about patterns is that they can include all kinds of additional information like, for example, that the value a pattern is to match is a positive integer. Consider the following example of the recursive factorial implementation. Let's look at that function in C++,

```
int fact(int x) {
  if (x == 0)
     return 1;
  else if (x > 0)
     return x * fact(x-1);
  else
    throw "undefined for negative values";
}
```

Factorials are only defined over positive integers. So notice that in this case we have to declare the argument as an integer argument and then later in the function we need to make sure that the integer value being passed in is in the correct range of values in order for the computation to be successful. With pattern matching in Asteroid we can accomplish all this right in the patterns that we apply to the input argument,

```
-- define patterns matching positive or negative integer values
let POS_INT = pattern %[(k:%integer) if k > 0]%.
let NEG_INT = pattern %[(k:%integer) if k < 0]%.

-- define our factorial function
function fact
    with 0 do
        return 1
    with x:*POS_INT do            -- use first pattern
        return x * fact(x-1).
    with x:*NEG_INT do            -- use second pattern
        throw Error("undefined for "+x).
    end
```

The first two lines of the program create the patterns that define what it means to be a positive or negative integer. For example, the first pattern will only match a value that is an integer whose value is larger than zero. Later in the program, these patterns get dereferenced (which means retrieved from where they are stored in memory) using the * operator. Notice that we have a similar setup here as with the `f` function we looked at earlier. If the `0` pattern matches then we will just return the value `1`. The line after that is saying "with the argument `x` and the pattern POS_INT (or in other words, if the argument is positive), recursively find the factorial of the number" and the last `wth` line is saying "with the

argument x and the pattern NEG_INT (if the argument is negative), throw an error". Notice that patterns allow us to precisely define what we mean by positive or negative integers in one place and then use these patterns in our function.

Pattern matching can be applied in a lot of places in Asteroid. But one other place is perhaps more prevalent than any other, which is pattern matching in Asteroid's `let` statement. The `let` statement is Asteroid's version of the assignment statement with a twist though: the left side of the = sign is not just a variable but is considered a pattern. For simple assignments there is no discernible difference between assignments in Asteroid and assignments in other languages,

```
let x = val.
```

Here, the variable `x` will match the value stored in `val`. However, because the left side of the = sign is a pattern we can write something like this,

```
let x:%[(k:%integer) if mod(k,2)==0]% = val.
```

where `x` will only match the value of `val` if that value is an even integer value. The fact that the left side of the = is a pattern allows us to write things like this,

```
let 1 = 1.
```

which simply states that the value 1 on the right can be matched by the pattern 1 on the left. Having the ability to pattern match on literals is convenient for statements like these,

```
let (1,x) = p.
```

This `let` statement is only successful for values of `p` which are pairs where the first component of the pair is the value 1. **The thing to remember is that the let statement is not entirely equivalent to the assignment operator in other languages, even though it may look like that.**

1.3.2 Object-Oriented Programming in Asteroid

The term object-oriented in programming means that code is broken up into classes and objects. Think of classes as **user defined data types**. While this may sound intimidating, there are many uses of object-oriented programming that can be used to help write efficient, clean code. For instance, there may be a time where you have to write code for software that pertains to families. While you could use tuples or arrays to represent this data, objects and classes are an even better way to achieve this feat. Take a look at this code in C++ that has the class for a family:

```
class Family {
public:
    std::string parent;
    std::string child1;
    std::string child2;

    // constructor
    Family(std::string p, std::string c1, std::string c2) {
        this->parent = p;
        this->child1 = c1;
        this->child2 = c2;
    }
};
```

Now if you want to create an instance or object of the Family class, you could write this line to do so:

```
Family *myfamily = new Family("Jim", "Bob", "Ann");
```

where the properties parent is "Jim", child1 is "Bob" and child2 is "Ann". Now if you wanted to access one of these properties, you could do,

```
std::cout << myfamily->child1; // while this looks intimidating, all this is doing is␣
↪dereferencing child1
```

Classes and objects are an easier way to store data that may not fit with any data structure that a language currently has. Asteroid implements object-orientation via structures, an approach it shares with the programming language Rust. In Asteroid the above example would be written as,

```
structure Family with
    data parent.
    data child1.
    data child2.

    -- constructor
    function __init__ with (p:%string, c1:%string, c2:%string) do
       let this @parent = p.
       let this @child1 = c1.
       let this @child2 = c2.
    end
end
```

And you can create an object from that structure by doing,

```
let myfamily = Family("Jim", "Bob", "Ann").
```

Notice how similar the construction of objects are in both languages. **Think of structures in Asteroid as classes in C++, and in both languages these allow you to instantiate objects** (that means if you have programmed with classes and objects in C++, creating structures in Asteroid should be trivial). Something else to note is that similar to Rust and Go, **Asteroid does not have inheritance for classes**. That is why programming in Asteroid is sometimes referred to as object-based programming rather than object-oriented programming.

We can access substructures of objects with the access operator `@`,

```
io @println (myfamily @child1).
```

which will print out the name of the first child.

The name of the class above can now be considered a user defined data type and can appear wherever built-in data type names can appear. For instance it can appear in a pattern restricting the values a particular variable can take on,

```
let f:%Family = myfamily.
```

Since we are talking about the `let` statement in conjunction with objects, Asteroid allows pattern matching on objects! This allows for easy access to substructures of objects,

```
let Family(parent,first,second) = myfamily.

assert(parent is "Jim").
assert(first is "Bob").
assert(second is "Ann").
```

Here we are matching the object stored in `myfamily` again the pattern `Family(parent,first,second)` and the variables will be instantiated with appropriate values from the `data` members of the object.

Now that you understand the two different paradigms that Asteroid is made out of, you can start writing your programs in it and explore the versatility of patterns, pattern-matching and object-oriented programming.

1.3.3 How to Get Started in Asteroid

Now that you know what principles Asteroid is made of, you can now get started writing programs in it. Directions to install Asteroid can be found here. After you installed Asteroid correctly, you can write your first program. The first one you can write is a simple hello world program, which looks something like:

```
load system io. -- header that allows the programmer to print things out to the screen␣
↪and to accept input

io @println "Hello, World!".
```

After you have written your first program, you can run the program by typing in the following line in your terminal:

```
asteroid <name of program>
```

where the name of the program is the name of the file that you want to run.

Make sure that you are in the same folder in your terminal of the file that you are trying to run!

Notice how the `@` symbol is used in two different places (this is common in programming languages, where one operator can be used multiple ways). In Asteroid, modules (which was the `load system io.` line at the top of our files) are actually objects, so to access a method in a module, you use the `@` symbol. So in this example, the module is the `io` module and we want to use the `println` method in that module, which is why you see the `@` symbol in there. **A module is a group of code that has already been written (typically by the developers of the language) which can be used again in other people's programs.**

Here is the complete list of modules in Asteroid.

Some important things to note in Asteroid:

- Most statements must end with a period (this is equivalent to using a semicolon in C++)
- In order to print things, you must include the `load system io.` in your program before you attempt any output.
- lines that start with `--` are comment lines
- If you see a line that looks like (`x:%integer`), that is used to match any value of a given type. (The `%integer` pattern matches any integer value and can be used with any other type in Asteroid.)

If you would like more information about Asteroid, please see the Asteroid reference guide and user guide.

1.4 Asteroid User Guide

1.4.1 Introduction

Asteroid is a modern, application-oriented, multi-paradigm programming language supporting first-class patterns. The language is heavily influenced by Python, Rust, ML, and Prolog. Furthermore, Asteroid is dynamically typed and makes pattern matching one of its core computational mechanisms. When we talk about pattern matching we mean both structural pattern matching as well as regular expression matching.

In this document we describe the major features of Asteroid and give plenty of examples. If you have used a programming language like Python or JavaScript before, then Asteroid should appear very familiar. However, there are some features which differ drastically from other programming languages due to the core pattern-matching programming paradigm with first-class patterns. Here are just two examples:

Example: All statements that look like assignments are actually pattern-match statements. For example if we state,

```
let [x,2,y] = [1,2,3].
```

that means the subject term `[1,2,3]` is matched to the pattern `[x,2,y]` and `x` and `y` are bound to the values 1 and 3, respectively. By the way, there is nothing wrong with the following statement,

```
let [1,2,3] = [1,2,3].
```

which is just another pattern match without any variable instantiations.

Example: Patterns in Asteroid are first-class citizens of the language. This is best demonstrated with a program. Here is a program that recursively computes the factorial of a positive integer and uses first-class patterns in order to ensure that the domain of the function is not violated,

```
-- define first-class patterns
let POS_INT = pattern (x:%integer) if x > 0.
let NEG_INT = pattern (x:%integer) if x < 0.

-- define our factorial function
function fact
    with 0 do
        return 1
    with n:*POS_INT do            -- use first pattern
        return n * fact (n-1).
    with n:*NEG_INT do            -- use second pattern
        throw Error("undefined for "+n).
    end
```

As you can see, the program first creates patterns and stores them in the variables `POS_INT` and `NEG_INT` and it uses those patterns later in the code by dereferencing those variables with the * operator. First-class patterns have profound implications for software development in that pattern definition and usage points are now separate and patterns can be reused in different contexts.

These are just two examples where Asteroid differs drastically from other programming languages. This document is an overview of Asteroid and is intended to get you started quickly with programming in Asteroid.

1.4.2 The Basics

As with most programming languages we are familiar with, Asteroid has variables (alpha-numeric symbols starting with an alpha character) and constants. Constants are available for all the primitive data types,

- `integer`, e.g. `1024`
- `real`, e.g. `1.75`
- `string`, e.g. `"Hello, World!"`
- `boolean`, e.g. `true`

Asteroid arranges these primitive data types in a type hierarchy,

`boolean` < `integer` < `real` < `string`

Type hierarchies facilitate automatic type promotion. Here is an example where automatic type promotion is used to put together a string from different data types,

```
let x:%string = "value: " + 1.
```

Here we associate the string "value: 1" with the variable x by first promoting the integer value 1 to the string "1" using the fact that integer < string according to our type hierarchy and then interpreting the + operator as a string concatenation operator.

Asteroid also supports the built-in data types:

- list
- tuple

These are structured data types in that they can contain entities that belong to other data types. Both of these data types have constructors which are possibly empty sequences of comma separated values enclosed by square brackets for lists, e.g. [1,2,3], and enclosed by parentheses for tuples, e.g. (x,y). For tuples we have the caveat that the 1-tuple is represented by a value followed by a comma to distinguish it from parenthesized expressions, e.g. (3,) the 1-tuple versus (3) the parenthesized expression. Here are some examples,

```
let l = [1,2,3].  -- this is a list
let t = (1,2,3).  -- this is a tuple
```

As we said above, in order to distinguish it from a parenthesized value the single element in a 1-tuple has to be followed by a comma, like so,

```
let one_tuple = (1,).  -- this is a 1-tuple
```

Lists are mutable objects whereas tuples are immutable. Lists and tuples themselves are also embedded in type hierarchies, although very simple ones:

- list < string
- tuple < string

That is, any list or tuple can be viewed as a string. This is very convenient for printing lists and tuples,

```
load system io.
io @println ("this is my list: " + [1,2,3]).
```

Here the + operator acts like a string concantenation operator with the list [1,2,3] promoted to a string according to the above type hierarchy.

Asteroid supports the none type. The none type has only one member: A constant named none. However, it turns out that the null-tuple, a tuple with no components indicated by (), also belongs to this type rather than the tuple type discussed earlier. But the none data type only has one constant, this implies that () and none mean the same thing and can be used interchangeably. That is, the following let statements will succeed,

```
let none = ().
let () = none.
```

showing that () and none are equivalent and pattern-match each other. The none data type itself does not belong to any type hierarchy.

We should mention here that because functions and patterns are both first-class citizens in Asteroid we also have the types function and pattern,

```
load system type.

-- define a function
```

(continues on next page)

(continued from previous page)

```
function inc with x do
   return x+1.
end

-- show that 'inc' is of type 'function'
assert (type @gettype(inc) == "function").
```

Just like the `none` type, neither of these types are part of a type hierarchy but we can use them in type patterns (see below).

By now you probably figured out that statements are terminated with a period and that comments start with a `--` symbol and continue till the end of the line. You probably also figured out that the `let` statement is Asteroid's version of assignment even though the underlying mechanism is a bit different as we will see when we discuss pattern matching in more detail.

1.4.3 Data Structures

Lists

In Asteroid the `list` is a fundamental, built-in data structure. A trait it shares with programming languages such as Lisp, Python, ML, and Prolog. Below is a list reversal example program. Notice that lists are zero-indexed and elements of a list are accessed via the `@` operator,

```
load system io.

let a = [1,2,3].          -- construct list a
let b = [a@2, a@1, a@0].  -- reverse list a
io @println b.
```

The output is: `[3,2,1]`.

We can achieve the same effect by giving a list of index values (a slice) to the `@` operator,

```
load system io.

let a = [1,2,3].     -- construct list a
let b = a@[2,1,0].   -- reverse list a using slice [2,1,0]
io @println b.
```

In Asteroid lists are considered objects with member functions that can manipulate list objects. We could rewrite the above example as,

```
load system io.

let a = [1,2,3].
let b = a @reverse (). -- reverse list using member function 'reverse'
io @println b.
```

The `@` operator is Asteroid's general access operator. It allows you to access either individual elements, slices, or member functions of a list. It also allows for access to members and functions of tuples and objects. Notice that in order to access the `println` function of the `io` module we also use the `@` operator. This is because in Asteroid, system modules are objects, so you must use `@` to access the functions of the module.

For a comprehensive treatment of available member functions for lists and tuples please see the reference guide. We look at objects later on in this guide.

Besides using the constructor for lists which consists of the square brackets enclosing comma separated elements we can use list comprehensions to construct lists. In Asteroid a list comprehension consist of a range specifier together with an optional step specifier allowing you to generate integer values within that range,

```
load system io.

-- build a list of odd values
let a = [1 to 10 step 2].  -- list comprehension
io @println ("list: " + a).

-- reverse the list using a slice computed as comprehension
let slice = [4 to 0 step -1]. -- list comprehension
let b = a@slice.
io @println ("reversed list: " + b).
```

The output is,

```
list: [1,3,5,7,9]
reversed list: [9,7,5,3,1]
```

Asteroid's simple list comprehensions in conjunction with the `map` function for lists allows you to construct virtually any kind of list. For example, the following program constructs a list of alternating 1 and -1,

```
load system io.
load system math.

let a = [1 to 10] @map (lambda with x do math @mod (x,2))
                  @map (lambda with x do 1 if x else -1).

io @println a.
```

where the output is,

```
[1,-1,1,-1,1,-1,1,-1,1,-1]
```

Tuples

As we saw earlier, the tuple is another fundamental, built-in data structure that can be found in Asteroid. Below is an example of a tuple declaration and access.

```
let a = (1,2,3).    -- construct tuple a
let b = a@1.        -- access the second element in tuple a, tuples are 0-indexed
assert (b == 2).    -- assert that the value of the second element is correct
```

Lists and tuples may be nested,

```
-- build a list of tuples
let b = [("a","b","c"),
         ("d","e","f"),
         ("g","h","i")].
```

(continues on next page)

(continued from previous page)

```
-- Access an element in the nested structure.
assert (b@0@1 == "b").
```

As we have mentioned earlier, unlike lists, tuples are immutable. This means that their contents cannot be changed once they have been declared. The following program demonstrates this,

```
load system io.

let b = ("a","b","c"). -- build a tuple

try
    let b@1 = "z". -- attempt to modify an element in the tuple
catch Exception (kind,message) do
    io @println (kind+": "+message).
end.
```

Which will print out the following message:

```
SystemError: term '(a,b,c)' is not a mutable structure
```

Should we want to change the contents of an already declared tuple, we would need to abandon the original and create a new one with the updated contents. When to use tuples and when to use lists is really application dependent. Tuples tend to be preferred over lists when representing some sort of structure, like abstract syntax trees, where that structure is immutable meaning, for example, that the arity of a tree node cannot change.

Structures and Objects

You can introduce custom data structures using the `structure` keyword. For example, the following statement introduces a structure of type `A` with data members `a` and `b`,

```
structure A with
   data a.
   data b.
end
```

Structures differ from lists and tuples in the sense that the name of the structure acts like a type tag. So, when you define a new structure you are in fact introducing a new type into your program.

For each structure Asteroid creates a default constructor that instantiates an object from that structure. The default constructor copies the arguments given to it into the data member fields in the order that the arguments and data members appear in the program text. Also, the data fields of an object are accessed via their names rather than index values. Here is a simple example that illustrates all this,

```
-- define a structure of type A
structure A with
    data a.
    data b.
end

let obj = A(1,2).     -- default constructor, a<-1, b<-2
assert (obj@a == 1).  -- access first data member
assert (obj@b == 2).  -- access second data member
```

The following is a more involved example,

```
load system io.

structure Person with
    data name.
    data age.
    data gender.
end

-- make a list of persons
let people = [
    -- use default constructors to construct Person objects
    Person("George", 32, "man"),
    Person("Sophie", 46, "woman"),
    Person("Oliver", 21, "man")
].

-- retrieve the second person on the list and use pattern
-- matching on Person objects to extract member values
let Person(name,age,gender) = people@1.

-- print out the member values
io @println (name + " is " + age + " years old and is a " +  gender + ").
```

The output is,

```
Sophie is 46 years old and is a woman.
```

The `structure` statement introduces a data structure of type `Person` with the three data members `name`, `age`, and `gender`. We use this data structure to build a list of persons. One of the interesting things is that we can pattern match the generated data structure as in the second `let` statement in the program to extract information from a `Person` object.

In addition to the default constructor, structures in Asteroid also support user specified constructors and member functions. We'll talk about those later when we talk about OO programming in Asteroid.

1.4.4 The Let Statement

The `let` statement is a pattern matching statement of the form,

```
let <pattern> = <value>.
```

where the pattern on the left side of the equal sign is matched against the value of the right side of the equal sign. When the pattern consist of just a single variable then the let statement can be viewed as Asteroid's version of the assignment statement, e.g.,

```
let x = 1.
```

However, statements like,

```
let 1 = 1.
```

where we pattern match the pattern 1 on the left side to the value 1 on the right side are completely legal and highlight the fact that the let statement is not equivalent to an assignment statement.

Simple patterns are expressions that consist purely of constructors and variables. Constructors themselves consist of constants, list and tuple constructors, as well as user defined structures. The advantage of pattern matching is that it provides direct access to substructures of a particular value. Consider that we want to access the constituent values of the pair `(1,2)`. In a non-pattern-matching approach we would have to access each of these constituent values one-by-one,

```
let p = (1,2).
let x = p@0.
let y = p@1.
assert (x==1 and y==2).
```

But in a pattern-matching approach we can write a `let` statement with a pattern that looks like a pair with the variables `x` and `y` where we expect our values to be,

```
let p = (1,2).
let (x,y) = p.
assert (x==1 and y==2).
```

Matching the pattern against the value `(1,2)` stored in `p` first matches the pair structure against the pair value and the matches the variables to the appropriate substructures. Once the variables have been matched to value the `let` statement declares the variables in the current scope and they become available for computation.

The following is an example involving structures and objects,

```
structure Person with
    data name.
    data age.
    data profession.
end

let joe = Person("Joe", 32, "Cook").  -- construct an object
let Person(n,a,p) = joe.              -- pattern match object

assert (n=="Joe" and a==32 and p=="Cook").
```

We first construct an object `joe` with the first `let` statement and the use pattern matching to desctructure it with the second `let` statement binding its substructures to the variables `n`, `a`, and `p`.

Asteroid supports special patterns called type patterns that match any value of a given type. For instance, the `%integer` pattern matches any integer value. Here is a simple example,

```
let %integer = 1.
```

This let statement succeeds because `1` is an integer value can be pattern-matched against the type pattern `%integer`.

Asteroid also supports something called a named pattern were a (sub)pattern can be given a name and that name will be instantiated with a term during pattern matching. For example,

```
load system io.

let t:(x,y) = (1,2).  -- using a named pattern on lhs
io @println t.
```

Here, the construct `t:(x,y)` is called a named pattern and the variable `t` will be unified with the term `(1,2)`, or more generally, the variable will be unified with term that matches the pattern on the right of the colon. The program will print,

```
(1,2)
```

Named patterns are a shorthand notation for conditional patterns; in this case,

```
let t if t is (x,y) = (1,2).
```

We can combine type patterns and named patterns to give us something that looks like a variable declaration in other languages. In Asteroid, though, it is still just all about pattern matching. Consider,

```
load system io.
load system math.

let x:%real = math @pi.
io @println x.
```

The left side of the `let` statement is a named type pattern that matches any real value, and if that match is successful then the value is bound to the variable `x`. Note that even though this looks like a declaration, it is in fact a pattern matching operation. The program will print the value `3.141592653589793`.

Beware of the fact that even though the `let` statement above looks like a declaration of a real variable it is not; it is a pattern match statement enforcing that the value assigned to `x` matches the pattern `%real`. Since this is a pattern match statement, this also means that standard type promotions such as promoting integers to reals during assignments in other programming languages do not apply here. For example, in Asteroid the following let statement fails,

```
let x:%real = 1.
```

because 1 is an integer value and does not match the pattern `%real`.

1.4.5 Flow of Control

Control structure implementation in Asteroid is along the lines of any of the modern programming languages such as Python, Swift, or Rust. For example, the `for` loop allows you to iterate over lists without having to explicitly define a loop index counter. In addition, the `if` statement defines what does or does not happen when certain conditions are met in a very familiar way. For a list of all control statements in Asteroid, please take a look at the reference guide.

As we said, in terms of flow of control statements there are really not a lot of surprises. This is because Asteroid supports loops and conditionals in a very similar way to many of the other modern programming languages. For example, here is a short program with a `for` loop that prints out the first six even positive integers,

```
load system io.

for i in 0 to 10 step 2 do
    io @println i.
end
```

The output is,

```
0
2
4
6
8
10
```

Here is another example that iterates over lists,

```
load system io.
load system util

let indexes = ["first","second","third"].
let birds = ["turkey","duck","chicken"].

for (ix,bird) in util @zip (indexes,birds) do
    io @println ("the "+ix+" bird is a "+bird).
end
```

The output is,

```
the first bird is a turkey
the second bird is a duck
the third bird is a chicken
```

In the loop we first create a list of pairs using the `zip` function, over which we then iterate pattern matching on each of the pairs on the list with the pattern `(ix,bird)`.

The following is a short program that demonstrates an `if` statement,

```
load system io.
load system type.

let x = type @tointeger (io @input "Please enter an integer: ").

if x < 0 do
    let x = 0.
    io @println "Negative, changed to zero".
elif x == 0 do
    io @println "Zero".
elif x == 1 do
    io @println "One".
else do
    io @println "Something else".
end
```

Even though Asteroid's flow of control statements look so familiar, they support pattern matching to a degree not found in other programming languages and which we will take a look at below.

1.4.6 Functions

Functions in Asteroid resemble function definitions in functional programming languages such as Haskell and ML. Here functions definitions have a single formal argument and function calls are expressed via juxtaposition of the function name and the single actual argument. Here is a simple example,

```
function double with i do -- pattern match the actual arg with i
    return 2*i.
end

let d = double 2.  -- function call via juxtaposition, no parentheses necessary
assert (d == 4).
```

In the `with` expression we pattern match the actual argument that is being passed in against the variable `i`. Also note that the function call is expressed via juxtaposition, no parentheses necessary.

If we wanted to pass more than a single value to a function we have to create a tuple and then pass that tuple to the function like in this example,

```
function reduce with (a,b) do -- pattern match the actual argument
    return a*b.
end

let r = reduce (2,4).  -- function call via juxtaposition
assert (r == 8).
```

Even though the function call looks like a traditional function call like in Python it is not. The underlying mechanism is quite different: on the call site we construct a tuple that holds all our values which is then passed to the function as the only parameter. Within the function that tuple is pattern matched and whatever variables are instantiated during this pattern match can be used within the function body.

In Asteroid functions are multi-dispatch, that is, a single function can have multiple bodies each attached to a different pattern matching the actual argument. The following is the quick sort implemented in Asteroid where each `with` clause introduces a new function body with its corresponding pattern,

```
load system io.

function qsort
    with [] do -- empty list pattern
        return [].
    with [a] do -- single element list pattern
        return [a].
    with [pivot|rest] do -- separating the list into pivot and rest of list
        let less=[].
        let more=[].

        for e in rest do
            if e < pivot do
                less @append e.
            else
                more @append e.
            end
        end

        return qsort less + [pivot] + qsort more.
    end

-- print the sorted list
io @println (qsort [3,2,1,0])
```

The output is as expected,

```
[0,1,2,3]
```

Notice that we use the multi-dispatch mechanism to deal with the base cases in the first two `with` clauses. In the third `with` clause we use the pattern `[pivot|rest]` to match the input list. Here the variable `pivot` matches the first element of the list, and the variable `rest` matches the remaining list. This remaining list is the original list with its first element removed. The function body then implements the pretty much standard recursive definition of the quick

sort. Just keep in mind that function calls are expressed via juxtaposition of function name and actual argument; no parentheses necessary.

As you have seen in a couple of occasions already in the document, Asteroid also supports anonymous or `lambda` functions. Lambda functions behave just like regular functions except that you declare them on-the-fly and they are declared without a name. Here is an example using a `lambda` function,

```
load system io.

io @println ((lambda with n do n+1) 1).
```

The output is `2`. Here, the `lambda` function is a function that takes a value and increments it by one. We then apply the value `1` to the function and the print function prints out the value `2`.

1.4.7 Pattern Matching

Pattern matching lies at the heart of Asteroid. We saw some of Asteroid's pattern matching ability when we discussed the `let` statement. Here is a more general discussion of pattern matching.

Pattern Matching in Expressions: The Is Predicate

We can also have pattern matching in expressions using the `is` predicate. Consider the following example,

```
load system io.

let p = (1,2).

if p is (x,y,z) do
    io @println ("it's a triple with: "+x+","+y+","+z)
elif p is (x,y) do
    io @println ("it's a pair with: "+x+","+y).
else do
    io @println "it's something else".
end
```

Here we use patterns to determine if `p` is a triple, a pair, or something else. Pattern matching is embedded in the expressions of the `if` statement using the `is` predicate. The output of this program is,

```
it's a pair with: 1,2
```

Pattern matching with the `is` predicate can happen anywhere expressions can be used. That means we can use the predicate also on the right side of `let` statements,

```
let true = (1,2) is (1,2).
```

This is kind of strange looking but it succeeds. Here the left operand of the `is` predicate is a term and the right operand is a pattern. Obviously this pattern match will succeed because the term and the pattern look identical. The return value of the `is` predicate is then pattern matched against the `true` pattern on the left of the = symbol.

We can also employ pattern matching in loops. In the following program we use the `is` predicate to test whether a list is empty or not while looping,

```
load system io.

let list = [1,2,3].

repeat do
    let [head|tail] = list. -- pattern match with head/tail operator
    io @println head.
    let list = tail.
until list is []. -- pattern match with is predicate
```

The output is,

```
1
2
3
```

The example employs pattern matching using the head-tail operator in order to iterate over the list elements and print them. The termination condition of the loop is computed with the `is` predicate.

Pattern Matching in Function Arguments

As we have seen earlier, Asteroid supports pattern matching on function arguments in the style of ML and many other functional programming languages. Here is an example that uses pattern matching on function arguments using binary tree data structures,

```
structure Node with -- internal tree node with a value
    data value.
    data left_child.
    data right_child.
end

structure Leaf with -- leaf node with a value
    data value.
end

-- traverse a tree and collect all the values in the tree in a list
function traverse
    with Leaf(v) do
        return [v].
    with Node(v,l,r) do
        return [v] + traverse l + traverse r.
end

let tree = Node(1,Leaf(2),Leaf(3)).
assert (traverse(tree) == [1,2,3]).
```

The structures `Node` and `Leaf` allow us to construct binary trees with embedded values. The `traverse` function traverses such trees and collects the values embedded in a tree on a list and returns that list. Notice the pattern matching on the tree node constructs in the `with` clauses of the `traverse` funtion.

Conditional Pattern Matching

Asteroid allows the user to attach conditions to patterns that need to hold in order for the pattern match to succeed. This is particularly useful for restricting input values to function bodies. Consider the following definition of the `factorial` function where we use conditional pattern matching to control the kind of values that are being passed to a particular function body,

```
load system io.

function factorial
    with 0 do
        return 1
    with n if (n is %integer) and (n > 0) do
        return n * factorial (n-1).
    with n if (n is %integer) and (n < 0) do
        throw Error("factorial is not defined for "+n).
end

io @println ("The factorial of 3 is: " + factorial 3).
```

Here we see that first, we make sure that we are being passed integers and second, that the integers are positive using the appropriate conditions on the input values. If we are being passed a negative integer, then we throw an error.

The above factorial program can be simplified by rewriting the first condition on `n` in the conditional patterns as a named pattern. We can also take advantage of the fact that the last expression evaluated in a function body provides an implicit return value. This gives us,

```
load system io.

function factorial
    with 0 do
        1
    with (n:%integer) if n > 0 do
        n * factorial (n-1).
    with (n:%integer) if n < 0 do
        throw Error("factorial is not defined for "+n).
end

io @println ("The factorial of 3 is: " + factorial 3).
```

The parentheses as they appear in the conditional pattern expressions are necessary.

Pattern Matching in For Loops

We have seen pattern matching in `for` loops earlier. Here we show another example. This combines structural matching with regular expression matching in `for` loops that selects certain items from a list. Suppose we want to print out the names of persons that contain a lower case 'p',

```
load system io.

structure Person with
    data name.
    data age.
end
```

(continues on next page)

(continued from previous page)

```
-- define a list of persons
let people = [
    Person("George", 32),
    Person("Sophie", 46),
    Person("Oliver", 21)
].

-- print names that contain 'p'
for Person(name if name is ".*p.*",age) in people do
    io @println name.
end
```

Here we pattern match the `Person` object in the `for` loop and then use a regular expression to see if the name of that person matches our requirement that it contains a lower case 'p'. The output is `Sophie`.

Pattern Matching in Try-Catch Statements

Exception handling in Asteroid is very similar to exception handling in many of the other modern programming languages available today. The example below shows an Asteroid program that throws one of two exceptions depending on the randomly generated value `i`,

```
load system io.
load system random.
load system type.

structure Head with
    data val.
end

structure Tail with
    data val.
end

try
    let i = random @random ().
    if i >= 0.5 do
        throw Head i.
    else do
        throw Tail i.
    end
catch Head v do
    io @println ("you win with "+type @tostring (v,type @stringformat (4,2))).
catch Tail v do
    io @println ("you loose with "+type @tostring (v,type @stringformat (4,2))).
end
```

The `Head` and `Tail` exceptions are handled by their corresponding `catch` statements, respectively. In both cases the exception object is unpacked using pattern matching and the unpacked value is used in the appropriate message printed to the screen.

It is worth noting that even though Asteroid has builtin exception objects such as `Error`, you can construct any kind of object and throw it as part of an exception.

1.4.8 Structures, Object-Oriented Programming, and Pattern Matching

We saw structures such as,

```
structure Person with
    data name.
    data age.
end
```

earlier. It is Asteroid's way to create custom data structures. These structures introduce a new type name into a program. For instance, in the case above, the `structure` statement introduces the type name `Person`. Given a structure definition, we can create objects from that structure. For example,

```
let scarlett = Person("Scarlett",28).
```

The right side of the `let` statement invokes the default constructor for the structure in order to create an object stored in the variable `scarlett`. We can access members of the object,

```
load system io.

structure Person with
    data name.
    data age.
end

let scarlett = Person("Scarlett",28).
-- access the name field of the structure instance
io @println (scarlett@name).
```

Asteroid allows you to attach functions to structures. In member functions the object identity is available through the `this` keyword. For example, we can extend our `Person` structure with the `hello` function that uses the `name` field of the object,

```
load system io.

structure Person with
    data name.
    data age.
    function hello with none do
        io @println ("Hello, my name is "+this@name).
    end
end

let scarlett = Person("Scarlett",28).
-- call the member function
scarlett @hello ().
```

This program will print out,

```
Hello, my name is Scarlett
```

The expression `this@name` accesses the `name` field of the object the function `hello` was called on. Even though our structures are starting to look a bit more like object definitions, pattern matching continues to work in the same way from when we discussed structures. The only thing you need to keep in mind is that you cannot pattern match on a

function member field. From a pattern matching perspective, a structure consists only of data fields. So even if we declare a structure like this,

```
load system io.

structure Person with
    data name.
    -- the function is defined in the middle of the data fields
    function hello with none do
        io @println ("Hello, my name is "+this@name).
    end
    data age.
end

-- pattern matching ignores function definitions
let Person(name,age) = Person("Scarlett",28).
io @println (name+" is "+age+" years old").
```

where the function `hello` is defined in the middle of the data fields, pattern matching simply ignores the function definition and pattern matches only on the data fields. The output of the program is,

```
Scarlett is 28 years old
```

Here is a slightly more involved example loosely based on the dog example from the Python documentation. The idea of the dog example is to have a structure that describes dogs by their names and the tricks that they can perform. Rather than using the default constructor, we define a constructor for our instances with the `__init__` function that performs some basic type checking on its arguments using type patterns and then initializes the data members of the object. Here is the program listing for the example in Asteroid,

```
load system io.
load system type.

structure Dog with
   data name.
   data tricks.
   function __init__ with (name:%string, tricks:%list) do -- constructor
      let this@name = name.
      let this@tricks = tricks.
   end
end

let fido = Dog("Fido",["play dead","fetch"]).
let buddy = Dog("Buddy",["sit stay","roll over"]).
let bella = Dog("Bella",["roll over","fetch"]).

let dogs = [fido,buddy,bella].

-- print out all the dogs that know how to fetch
for (Dog(name,tricks) if type @tostring tricks is ".*fetch.*") in dogs do
   io @println (name+" knows how to fetch").
end
```

After declaring the structure we instantiate the dogs with their respective trick repertoires and put them on a list. The last couple of lines of the program consist of a `for` loop over a list of our dogs. The `for` loop is interesting because here we use structural, conditional, and regular expression pattern matching in order to only select the dogs that know

how to do `fetch` from the list of dogs. The pattern is,

```
Dog(name,tricks) if type @tostring tricks is ".*fetch.*"
```

The structural part of the pattern is `Dog(name,tricks)` which simply matches any dog instance on the list. However, that match is only successful if the conditional part of the pattern holds,

```
if type @tostring tricks is ".*fetch.*"
```

This condition only succeeds if the `tricks` list viewed as a string matches the regular expression `".*fetch.*"`. That is, if the list contains the word `fetch`. The output is,

```
Fido knows how to fetch
Bella knows how to fetch
```

1.4.9 Patterns as First-Class Citizens

A programming language feature that is promoted to first-class status does not change the power of a programming language in terms of computability but it does increase its expressiveness. Think functions as first-class citizens of a programming language. First-class functions give us `lambda` functions and `map`, both powerful programming tools.

The same is true when we promote patterns to first-class citizen status in a language. It doesn't change what we can and cannot compute with the language. But it does change how we can express what we want to compute. That is, it changes the expressiveness of a programming language.

In Asteroid first-class patterns are introduced with the keyword `pattern` and patterns themselves are values that we can store in variables and then reference when we want to use them. Like so,

```
let p = pattern (x,y).  -- define a first-class pattern
let *p = (1,2).         -- use the first-class pattern
```

The left side of the second `let` statement dereferences the pattern stored in variable `p` and uses the pattern to match against the value `(1,2)`.

Here we look at three examples of how first-class patterns can add to a developer's programming toolbox.

Pattern Factoring

Patterns can become very complicated especially when conditional pattern matching is involved. First-class patterns allow us to control the complexity of patterns by breaking patterns up into smaller subpatterns that are more easily managed. Consider the following function that takes a pair of values. The twist is that the first component of the pair is restricted to primitive data types of Asteroid,

```
function foo with (x if (x is %boolean) or (x is %integer) or (x is %string),y) do
    io @println (x,y).
end
```

That complicated pattern for the first component of the input pair completely obliterates the overall structure of the parameter pattern and makes the function definition difficult to read.

We can express the same function with a first-class pattern,

```
let TP = pattern q if (q is %boolean) or
                      (q is %integer) or
```

(continues on next page)

(continued from previous page)

```
                        (q is %string).

function foo with (x:*TP,y) do
    io @println (x,y).
end
```

It is clear now that the main input structure to the function is a pair and the conditional type restriction pattern has been relegated to a subpattern stored in the variable TP.

Pattern Reuse

In most applications of patterns in programming languages specific patterns appear in many spots in a program. If patterns are not first-class citizens the developer will have to retype the same patterns over and over again in the various different spots where the patterns occurs. Consider the following program snippet,

```
function fact
    with 0 do
        return 1
    with (n:%integer) if n > 0 do
        return n * fact (n-1).
    with (n:%integer) if n < 0 do
        throw Error("fact undefined for negative values").
end

function sign
    with 0 do
        return 1
    with (n:%integer) if n > 0 do
        return 1.
    with (n:%integer) if n < 0 do
        return -1.
end
```

In order to write these two functions we had to repeat the almost identical pattern four times. First-class patterns allow us to write the same two functions in a much more elegant way,

```
let POS_INT = pattern (x:%integer) if x > 0.
let NEG_INT = pattern (x:%integer) if x < 0.

function fact
    with 0 do
        return 1
    with n:*POS_INT do
        return n * fact (n-1).
    with *NEG_INT do
        throw Error("fact undefined for negative values").
end

function sign
    with 0 do
        return 1
    with *POS_INT do
```

(continues on next page)

(continued from previous page)

```
        return 1.
    with *NEG_INT do
        return -1.
end
```

The relevant patterns are now stored in the variables `POS_INT` and `NEG_INT` which are then used in the function definitions.

Constraint Patterns

Sometimes we want to use patterns as constraints on other patterns. Consider the following (somewhat artificial) example,

```
let x: (v if (v is %integer) and v > 0) = some_value.
```

Here we want to use the pattern `v if (v is %integer) and v > 0` purely as a constraint on the pattern `x` in the sense that we want a match on `x` only to succeed if `some_value` is a positive integer. The problem is that this constraint pattern introduces a spurious binding of the variable `v` into the current environment which might be undesirable due to variable name clashes. Constraint patterns address this. We can rewrite the above statement as follows,

```
let x: %[v if (v is %integer) and v > 0]% = some_value.
```

By placing the pattern `v if (v is %integer) and v > 0` within the `%[...]%` operators the pattern still functions as before but does not bind the variable `v` into the current environment.

The most common use of constraint patterns is the prevention of non-linear patterns in functions. Consider the following program,

```
load system io.

let POS_INT = pattern %[v if (v is %integer) and v > 0]%.

function add with (a:*POS_INT,b:*POS_INT) do
   return a+b.
end

io @println (add(1,2)).
```

Without the `%[...]%` operators around the pattern `v if (v is %integer) and v > 0` the argument list pattern for the function `(a:*POS_INT,b:*POS_INT)` would instantiate two instances of the variable `v` leading to a non-linear pattern which is not supported by Asteroid. With the `%[...]%` operators in place we prevent the pattern `v if (v is %integer) and v > 0` from instantiating the variable `v` thus preventing a non-linearity to occur in the argument list pattern.

Sometimes we need to use constraint patterns instead of straightforward patterns in order to avoid non-linearities but we also want controlled access to the variables these constraint patterns declare. We achieve this by using the `bind` keyword at the pattern-match site. Consider the following program,

```
-- declare a pattern that matches scalar values
let Scalar = pattern %[p if (p is %integer) or (p is %real)]%.

-- declare a pattern that matches pairs of scalars
let Pair = pattern %[(x:*Scalar,y:*Scalar)]%.
```

(continues on next page)

(continued from previous page)

```
-- compute the dot product of two pairs of scalars
function dot2d
   with (*Pair bind [x as a1, y as a2], *Pair bind [x as b1, y as b2]) do
      a1*b1 + a2*b2
end

assert(dot2d((1,0),(0,1)) == 0).
```

In the function definition of `dot2d` we see that the `Pair` pattern is used twice to make sure that the function is called with a pair of pairs as its argument. However, in order to compute the dot product of those two pairs we need access to the values each pair matched. We use the `bind` keyword together with an appropriate binding term list to extract the matched values. For the first pair we map `x` and `y` to `a1` and `a2` and for the second pair we map `x` and `y` to `b1` and `b2`, respectively.

As a quick aside, the `as` construction in the binding term list is only necessary when trying to resolve non-linearities otherwise the binding term list can just consist of the variable names appearing in the pattern that you want to bind into the current scope.

1.4.10 Basic Asteroid I/O

I/O functions are defined as member functions of the `io` module. The `println` function prints its argument in a readable form to the terminal. Recall that the + operator also implements string concatenation. This allows us to construct nicely formatted output strings,

```
load system io.

let a = 1.
let b = 2.
io @println ("a + b = " + (a + b)).
```

The output is

```
a + b = 3
```

We can use the `tostring` function defined in the `type` module to provide some additional formatting. The idea is that the `tostring` function takes a value to be turned into a string together with an optional `stringformat` formatting specifier object,

```
type @tostring(value[,type @stringformat(width spec[,precision spec])])
```

The width specifier tells the `tostring` function how many characters to reserve for the string conversion of the value. If the value requires more characters than given in the width specifier then the width specifier is ignored. If the width specifier is larger than than the number of characters required for the value then the value will be right justified. For real values there is an optional precision specifier.

Here is a program that exercises some of the string formatting options,

```
load system io.
load system type.
load system math.

-- if the width specifier is larger than the length of the value
```

(continues on next page)

(continued from previous page)

```
-- then the value will be right justified
let b = type @tostring(true,type @stringformat(10)).
io @println b.

let i = type @tostring(5,type @stringformat(5)).
io @println i.

-- we can format a string by applying tostring to the string
let s = type @tostring("hello there!",type @stringformat(30)).
io @println s.

-- for floating point values: first value is width, second value precision.
-- if precision is missing then value is left justified and zero padded on right.
let r = type @tostring(math @pi,type @stringformat(6,3)).
io @println r.
```

The output of the program is,

```
     true
   5
                hello there!
3.142
```

Notice the right justification of the various values within the given string length.

The `io` module also defines a function `print` which behaves just like `println` except that it does not terminate print with a newline.

Another useful function defined in the `io` module is the `input` function that, given an optional prompt string, will prompt the user at the terminal and return the input value as a string. Here is a small example,

```
load system io.

let name = io @input("What is your name? ").
io @println ("Hello " + name + "!").
```

The output is,

```
What is your name? Leo
Hello Leo!
```

We can use the type casting functions such as `tointeger` or `toreal` defined in the `type` module to convert the string returned from `input` into a numeric value,

```
load system io.
load system type.

let i if i > 0  = type @tointeger(io @input("Please enter a positive integer value: ")).

for k in 1 to i do
    io @println k.
end
```

The output is,

```
Please enter a positive integer value: 3
1
2
3
```

Finally, the function `read` reads from `stdin` and returns the input as a string. The function `write` writes a string to `stdout`.

1.4.11 The Module System

A module in Asteroid is a file with a set of valid Asteroid statements. You can load this file into other Asteroid code with the statement:

```
load "example_path/example_filename".
```

or:

```
load example_modulename.
```

The search strategy for a module to be loaded is as follows,

1. raw module name - could be an absolute path
2. search in current directory
3. search in directory where Asteroid is installed
4. search in subdirectory where Asteroid was started

Modules defined by the Asteroid system should be loaded with the keyword `system` in order to avoid any clashes with locally defined modules. If the `system` keyword is used then Asteroid only searches in its system folders rather than in user directories.

Say that you wanted to load the `math` module so you could execute a certain trigonometric function. The following Asteroid program loads the `math` module as well as the `io` module. Only after loading them would you be able to complete the sine function below,

```
load system io.
load system math.

let x = math @sin( math @pi / 2 ).
io @println("The sine of pi / 2 is " + x + ".").
```

Both the function `sin` and the constant value `pi` are defined in the `math` module. In addition, the `io` module is where all input/output functions in Asteroid (such as `println`) come from. If you want the complete list of modules, make sure to check out the reference guide here.

1.4.12 More on Exceptions

This section will give further information on how to work with exceptions, or unexpected conditions that break the regular flow of execution. Exceptions generated by Asteroid are `Exception` objects with the following structure,

```
structure Exception with
    data kind.
    data value.
end
```

The `kind` field will be populated by Asteroid with one of the following strings,

- `PatternMatchFailed` - this exception will be thrown if the user attempted an explicit pattern match which failed, e.g. a let statement whose left side pattern does not match the term on the right side.
- `NonLinearPatternError` - this exception occurs when a pattern has more than one variable with the same name, e.g. `let (x,x) = (1,2).`
- `RedundantPatternFound` - this exception is thrown if one pattern makes another superfluous, e.g. in a multi-dispatch function definition.
- `ArithmeticError` - e.g. division by zero
- `FileNotFound` - an attempt of opening a file failed.
- `SystemError` - a general exception.

In addition to the `kind` field, the `value` field holds a string with some further details on the exception. Specific exceptions can be caught by pattern matching on the `kind` field of the `Exception` object. For example,

```
load system io.

try
    let x = 1/0.
catch Exception("ArithmeticError", s) do
    io @println s.
end
```

The output is,

```
integer division or modulo by zero
```

Asteroid also provides a predefined `Error` object for user level exceptions,

```
load system io.

try
    throw Error("something worth throwing").
catch Error(s) do
    io @println s.
end
```

Of course the user can also use the `Exception` object for their own exceptions by defining a `kind` that does not interfere with the predefined `kind` strings above,

```
load system io.

try
```

(continues on next page)

(continued from previous page)

```
    throw Exception("MyException","something worth throwing").
catch Exception("MyException",s) do
    io @println s.
end
```

The output here is,

```
something worth throwing
```

In addition to the Asteroid defined exceptions, the user is allowed to construct user level exceptions with any kind of object including tuples and lists. Here is an example that constructs a tuple as an exception object,

```
load system io.

try
    throw ("funny exception", 42).
catch ("funny exception", v) do
    io @println v.
end
```

The output of this program is `42`.

Now, if you don't care what kind of exception you catch, you need to use a `wildcard` or a variable because exception handlers are activated via pattern matching on the exception object itself. Here is an example using a `wildcard`,

```
load system io.

try
    let (x,y) = (1,2,3).
catch _ do
    io @println "something happened".
end
```

Here is an example using a variable,

```
load system io.
load system type.

try
    let (x,y) = (1,2,3).
catch e do
    io @println ("something happened: "+type @tostring(e)).
end
```

In this last example we simply convert the caught exception object into a string and print it,

```
something happened: Exception(PatternMatchFailed,pattern match failed: term and pattern
lists/tuples are not the same length)
```

1.4.13 More on Multi-Dispatch

With the `qsort` function above we saw functional programming style dispatch where the `with` clauses represent a case analysis over a single type, namely the input type to the function. However, Asteroid has a much broader view of multi-dispatch where the `with` clauses can represent a case analysis over different types. In order to demonstrate this type of multi-dispatch, we show the example program from the multi-dispatch Wikipedia page written in Asteroid,

```
load system io.
load system type.

let pos_num = pattern %[x if type @isscalar(x) and x > 0]%.

structure Asteroid with
   data size.
   function __init_
      with v:*pos_num do
         let this @size = v.
      end
end

structure Spaceship with
    data size.
   function __init_
      with v:*pos_num do
         let this @size = v.
      end
end

-- we use first-class pattern SpaceObject to
-- express that both asteroids and space ships are space objects.
let SpaceObject = pattern %[x if (x is %Asteroid) or (x is %Spaceship)]%.

-- multi-dispatch function
function collide_with
  with (a:%Asteroid, b:%Spaceship) do
    return "a/s".
  with (a:%Spaceship, b:%Asteroid) do
    return "s/a".
  with (a:%Spaceship, b:%Spaceship) do
    return "s/s".
  with (a:%Asteroid, b:%Asteroid) do
    return "a/a".
  end

-- here we use the first-class pattern SpaceObject as a
-- constraint on the function parameters.
function collide with (x:*SpaceObject, y:*SpaceObject) do
  return "Big boom!" if (x@size > 100 and y@size > 100) else collide_with(x, y).
end

io @println (collide(Asteroid(101), Spaceship(300))).
io @println (collide(Asteroid(10), Spaceship(10))).
io @println (collide(Spaceship(101), Spaceship(10))).
```

Each `with` clause in the function `collide_with` introduces a new function body with its corresponding pattern. The function bodies in this case are simple `return` statements but they could be arbitrary computations. The output of the program is,

```
Big boom!
a/s
s/s
```

1.5 Asteroid Reference Guide

1.5.1 Language Syntax

Note: In the following descriptions <something>? denotes an optional something in a piece of syntax. We also use the notation <something>* which means that something can appear zero or more times in a program. Capitalized words are keywords where `FOR` represents the keyword `for` and `END` represents `end`.

Statements

Assert

Syntax: `ASSERT exp '.'?`

If the expression of the assert statement evaluates to a value equivalent to the Boolean value `false` an exception is thrown otherwise the statement is ignored.

For example, the statement,

```
assert (1+1 == 3).
```

will generate a runtime error but the statement,

```
assert (1+1 == 2).
```

will be ignored once the expression has been evaluated.

Break

Syntax: `BREAK '.'?`

The break statement immediately breaks out of the closest surrounding looping structure. Execution will continue at the statement right after the loop. Issuing a break statement outside of a looping structure will lead to a runtime error.

As an example we break out of the indefinite loop below when `i` is equal to 10,

```
let i = 0.

loop
   let i = i+1.
   if i==10 do
      break.
   end
```

(continues on next page)

(continued from previous page)

```
end

assert (i==10).
```

Expressions at the Statement Level

Expressions at the statement level are supported. However, they do not have any effect on the computation unless they contain side effects with one exception: In the absence of an explicit return statement, the value of the last expression evaluated in a function body is considered the return value of the function.

An example,

```
function inc
   with i do
      i+1.
   end
```

Notice that because the expression `i+1` is the last statement evaluated in the function body its value becomes the return value of the function.

For-Loop

Syntax: `FOR pattern IN exp DO stmt_list END`

In a for-loop the expression must evaluate to either a list or a tuple. The pattern is then matched to each component of the expression value sequentially starting with the first component. The loop body is executed for each successful match.

In the following program the body of the loop is executed exactly once when the pattern matches the tuple `(1, "chicken")`,

```
let tuple_list = [
        (0,"duck"),
        (1,"chicken"),
        (2,"turkey")
        ].

for (1,bird) in tuple_list do
   assert(bird is "chicken").
end
```

Function-Definition

Syntax: FUNCTION function_name WITH pattern DO stmt_list (WITH pattern DO stmt_list)* END

Function definitions in Asteroid can have one or more function bodies associated with single function name. A function body is associated with a particular pattern that is matched against the actual argument of the function call. If the match is successful then the associated function body is executed. If the match is not successful then other pattern/body pairs are tried if present. If none of the patterns match the actual argument then this constitutes a runtime error. Patterns are tried in the order they appear in the function definition.

The following is a definition of the `sign` function,

```
function sign
   with x if x >= 0 do
      return 1.
   with x if x < 0 do
      return -1.
end
```

Here the first function body returns 1 if the actual argument is greater or equal to zero. The second function body return -1 if the actual argument is less than zero.

Global

Syntax: `GLOBAL variable_name (',' variable_name)* '.'?`

The `global` statement allows the developer to declare a variable as global within a function scope and this allows the developer to set the value of a global variable from within functions.

Consider the following code snippet,

```
let x = 0.

function foo
   with none do
      global x.
      let x = 1.
end

assert(x==0).
foo().
assert(x==1).
```

The `global` statement within the function `foo` indicates that the `let` statement on the following line should assign a value to the global variable `x`.

If-Then-Else

Syntax: `IF exp DO stmt_list (ELIF exp DO stmt_list)* (ELSE DO? stmt_list)? END`

If the first expression evaluates to the equivalent of a Boolean `true` value then the associated statements will be executed and the execution continues after the `end` keyword. If the expression evaluates to the equivalent of a Boolean `false` then the expressions of the optional `elif` clauses are evaluated if present. If one of them evaluates to the equivalent of a Boolean value `true` then the associated statements are executed and execution continues after the `end` keyword. Otherwise the statements of the optional `else` clause are executed if present and again flow of control is transferred to the statements following the if-statement.

As an example consider the following `if` statement that determines what kind of integer value the user supplied,

```
load system io.
load system type.

let x = type @tointeger (io @input "Please enter an integer: ").

if x < 0 do
```

(continues on next page)

(continued from previous page)

```
    io @println "Negative".
elif x == 0 do
    io @println "Zero".
elif x == 1 do
    io @println "One".
else do
    io @println "Positive".
end
```

Let

Syntax: `LET pattern = exp '.'?`

The `let` statement is Asteroid's version of the assignment statement with a twist though: the left side of the = sign is not just a variable but is considered a pattern. For simple assignments there is no discernible difference between assignments in Asteroid and assignments in other languages,

```
let x = val.
```

Here, the variable `x` will match the value stored in `val`. However, because the left side of the = sign is a pattern we can write something like this,

```
load system math.
let x:%[(k:%integer) if math @mod(k,2)==0]% = val.
```

where `x` will only match the value of `val` if that value is an even integer value. The fact that the left side of the = is a pattern allows us to write things like this,

```
let 1 = 1.
```

which simply states that the value 1 on the right can be matched by the pattern 1 on the left. Having the ability to pattern match on literals is convenient for statements like these,

```
let (1,x) = p.
```

This `let` statement is only successful for values of `p` which are pairs where the first component of the pair is the value 1.

Loop

Syntax: `LOOP DO? stmt_list END`

The `loop` statement executes the statements in the loop body indefinitely unless a `break` statement is encountered.

Repeat-Until

Syntax: `REPEAT DO? stmt_list UNTIL exp '.'?`

Repeatedly execute the statements in the loop body until the expression evaluates to the equivalent of a Boolean `true` value.

Here is an example of a program that prints out the elements of a list,

```
load system io.

let l = ["bmw", "volkswagen", "mercedes"].

repeat
   let [element|l] = l.
   io @println element.
until l is [].
```

Return

Syntax; `RETURN exp? '.'?`

Explicitly return from a function with an optional return value.

Structure

Syntax: `STRUCTURE type_name WITH data_or_function_stmts END`

The `structure` statement introduces a composite data type that defines a physically grouped list of variables under one name. The variables within a structure can be declared as data members or as function members. Unless a member function was declared as a constructor (an `__init__` function) structures are instantiated using a default constructor. The default constructor copies the arguments given to it into the data member fields in the order that the data members appear in the structure definition and as they appear in the parameter list of the constructor. We often refer to instantiated structures as objects. Member values of objects are accessed using the access operator `@`. Here is a simple example,

```
-- define a structure of type A
structure A with
    data a.
    data b.
end

let obj = A(1,2).       -- call default constructor
assert( obj @a == 1 ).  -- access first data member
assert( obj @b == 2 ).  -- access second data member
```

We can use custom constructors to enforce that only certain types of values can be copied into an object,

```
-- define a structure of type Person
structure Person with
    data name.
    data age.
    function __init__ with (name:%string,age:%integer) do -- constructor
       let this @name = name.
```

(continues on next page)

(continued from previous page)

```
       let this @age = age.
    end
    function __str__ with none do
      return this@name+" is "+this@age+" years old".
    end
end

let betty = Person("Betty",21).  -- call constructor
assert( betty @name == "Betty" ).
assert( betty @age == 21 ).

load system type.
assert(type @tostring betty is "Betty is 21 years old").
```

Note that object identity is expressed using the `this` keyword. Here we also supplied an instantiation of the `__str__` function that allows us to customize the stringification of the object. See the last line where we cast the object `betty` to a string. Without the `__str__` function Asteroid uses a default representation of the object as a string. The `__str__` function does not accept any arguments and has to return a string.

Try-Catch

Syntax: `TRY DO? stmt_list (CATCH pattern DO stmt_list)+ END`

This statement allows the programmer to set up exception handlers for exceptions thrown in the code of the `try` part of the statement. Notice that you can set up one or more handlers within the `catch` part of the statement. If there are more than one handlers then they are searched in order starting with the first. Handlers are selected via pattern matching on the exception object. The handler code of the first `catch` clause whose pattern matches the exception object is executed.

Below is an example of a `try-catch` statement where the code in the `try` part generates a division-by-zero exception. The exception object is pattern-matched in the `catch` clause and processed by the associated handler,

```
load system io.

try
    let x = 1/0.
catch Exception("ArithmeticError", s) do
    io @println s.
end
```

For more details on exceptions please see the User Guide.

Throw

Syntax: `THROW exp '.'?`

Allows the developer to throw an exception. Any object can serve as an exception object. However, Asteroid provides some predefined exception objects. For more details on exceptions please see the User Guide.

While-Loop

Syntax: `WHILE exp DO stmt_list END`

While the expression evaluates to the equivalent of a Boolean `true` value execute the statements in the body of the loop. The loop expression is reevaluated after each loop iteration.

Here is an example that prints out a sequence of integer values in reverse order,

```
load system io.

let i = 10.

while i do
   io @println i.
   let i = i-1.
end
```

The loop terminates once `i` becomes zero which is the equivalent to a Boolean value `false`.

Expressions

All the usual arithmetic, relational, and logic operators,

```
+, -, *, /, ==, =/=, <=, <, >=, >, and, or, not
```

are supported in Asteroid. For extended mathematical operations such as `mod` (modulus) or `sin` (sine) see the `math` module. Here we discuss expression constructions that are particular to Asteroid.

Substructure Access

Syntax: `structure_exp @ index_exp`

Asteroid provides the uniform substructure access operator `@` for all structures which includes lists, tuples, and objects. For example, accessing the first element of a list is accomplished by the expression,

```
[1,2,3] @0
```

Similarly, given an object constructed from structure `A`, member values are accessed by name via the `@` operator,

```
structure A with
   data a.
   data b.
end

let obj = A(1,2).
assert( obj @a == 1 ).  -- access member a
```

Head-Tail Operator

Syntax: `element_exp | list_exp`

This operator works in one of two ways. In the first way it allows you to pre-append an element to a list,

```
let [1,2,3] = 1 | [2,3].
```

It can also be nested,

```
let [1,2,3] = 1 | 2 | 3 | [].
```

In the second way it works as a pattern to deconstruct a list into its first element and the remainder of the list, the list with its first element removed,

```
let h | t = [1,2,3].
assert(h == 1).
assert(t == [2,3]).
```

You can put optional brackets around the operator to highlight the fact that we are dealing with a list,

```
let [h | t] = [1,2,3].
```

The Is Predicate

Syntax: `exp IS pattern`

This operator matches the structure computed by the expression on the left side against the pattern on the right side of the operator. If the match is successful it returns the Boolean value `true` and if not successful then it returns the Boolean value `false`. All regular rules of pattern matching apply such as instantiating appropriate variable bindings in the current scope.

Example,

```
if v is (x,y) do
   io @println "success".
   assert(isdefined "x").
   assert(isdefined "y").
else
   io @println "not matched".
   assert(not isdefined "x").
   assert(not isdefined "y").
end
```

The In Predicate

Syntax: `exp IN list_exp`

This predicate returns `true` if the value computed by the expression on the left in contained in the list computed by the list expression on the right. It is an error if the expression on the right does not compute a list.

Example,

```
let true = 1 in [1,2,3].
```

The Eval Function

The `eval` function allows you to evaluate Asteroid expressions. If the expression is a string then the contents of the string is treated like Asteroid code and is interpreted accordingly in the current interpreter environment. If that code produces a value then the `eval` function will return that value, e.g.,

```
let a = eval "1+1".
assert(a == 2).
```

If the expression to be evaluated is a simple, structural pattern then the pattern is evaluated as a constructor where variables are instantiated from the current environment. For example,

```
let p = pattern (x,y)
let x = 1.
let y = 2.
let o = eval p.
assert(o is (1,2)).
```

List Comprehensions

Syntax: `start_exp TO end_exp (STEP exp)?`

This expression constructs a list starting with an element given by the start expression up to the value of the end expression with a given step. If the step expression is not given then a step value of 1 is assumed. The comprehension can be placed between optional square brackets.

Examples,

```
let [0,1,2,3,4] = 0 to 4.
let [0,-2,-4,-6] = [0 to -6 step -2].
```

Function Calls

Syntax: `exp exp`

Function calls are defined by function application, more specifically by juxtaposition of expressions. Here, the first expression has to evaluated to a function expression and the second expression has to evaluate to an appropriate actual function parameter. Notice that function calls are defined in terms of a single function parameter. If you would like to pass more than one value to a function then you have to create a tuple. For example, if the function `foo` needs two values to be passed to it then you need to create a tuple, e.g. `foo (1,2)`. In that respect function calls differ drastically from function calls in languages like C/C++ or Python.

Examples,

```
let val = (lambda with i do i+1) 1.
assert(val == 2).

function foo with (q,p) do q+p end
```

(continues on next page)

(continued from previous page)

```
let val = foo (1,2).
assert(val == 3).
```

If-Else Expressions

Syntax: `then_exp IF bool_exp ELSE else_exp`

If the boolean expression evaluates to true then this expression returns the value of the first expression. Otherwise it will return the value of the last expression.

Example,

```
let val = "yup" if b else "nope".
```

If `b` evaluates to true then this expression returns the string `"yup"` otherwise it returns the string `"nope"`.

First-Class Patterns

Syntax: `PATTERN exp`
Syntax: `'*' exp (BIND '[' ID (AS ID)? (',' ID (AS ID)?)*']')?`

This construction allows the user to construct a pattern as a value using the `pattern` keyword. The advantage of patterns as values is that they can be stored in variables or passed to or from functions. As an example we construct a pattern which is a pair where the first component is the constant `1` and the second component is the variable `x` and we store this pattern in the variable `p` for later use,

```
let p = pattern (1,x).
```

The pattern derefence operator * allows us to retrieve patterns from variables, e.g.

```
let *p = (1,2).
```

Here the pair `(1,2)` is matched against the pattern stored in the variable `p` such that `x` is bound to the value `2`.

The optional `bind` term together with an appropriate list of variable names allows the user to selectively project variable bindings from a constraint pattern into the current scope. The `as` keyword allows you to rename those bindings. Consider the following program,

```
let Pair = pattern %[(x,y)]%.

-- bindings of the variables x and y are now visible as a and y respetively
let *Pair bind [x as a, y] = (1,2).
assert( a == 1).
assert(y == 2).
```

At the second `let` statement we bind the `x` as `a` and `y` from the hidden scope of the constraint pattern into our current scope.

Type Patterns

Syntax: `'%'type_name`

Type patterns match all the values of a particular type. Type patterns exist for all the Asteroid builtin types and are also available for user defined types introduced via a `structure` command.

Example,

```
let true = 1 is %integer.
```

Named Patterns

Syntax: `name_exp ':'  pattern`

Named patterns allow you to bind the term matched by the pattern to a variable. Here the name expression has to evaluate to either a variable, object member variable, or list location.

Example,

```
let x:%integer = val.
```

The variable `x` will be bound to the value of `val` if that value matches the type pattern `%integer`.

Named patterns are a syntactic short hand for the equivalent conditional pattern,

```
name_exp if name_exp is pattern
```

That means the following two `let` statements are equivalent,

```
let x:(q,p) = (1,2).
let x if x is (q,p) = (1,2).
```

Conditional Patterns

Syntax: `pattern IF cond_exp`

In conditional patterns the pattern only matches if the condition expression evaluates to true.

Example,

```
load system math.
let k if (math @mod(k,2) == 0) = val.
```

Here `k` only matches the value of `val` if that value is an even number.

Pure Constraint Patterns

Syntax: `%[ pattern ]% (BIND '[' ID (AS ID)? (',' ID (AS ID)?)*']')?`

A pure constraint pattern is a pattern that does not create any bindings in the current scope. Any pattern can be turned into a pure constraint pattern by placing it between the `%[` and `]%` operators.

Example,

```
let pos_int = pattern %[(x:%integer) if x > 0]%
let i:*pos_int = val.
```

The first line defines a pure constraint pattern for the positive integers. Notice that the pattern internally uses the variable `x` in order to evaluate the conditional pattern but because it has been declared as a pure constraint pattern this value binding is not exported to the current scope during pattern matching. On the second line we constrain the pattern `i` to only the positive integer values using the pure constraint pattern stored in `p`. This pattern match will only succeed if `val` evaluates to a postive integer.

Asteroid Grammar

The following is the complete grammar for the Asteroid language. Capitalized words are either keywords such as `FOR` and `END` or tokens such as `STRING` and `ID`. Non-terminals are written in all lowercase letters. The grammar utilizes an extended BNF notation where <syntactic unit>* means zero or more occurrences of the syntactic unit and <syntactic unit>+ means one or more occurrences of the syntactic unit. Furthermore, <syntactic unit>? means that the syntactic unit is optional. Simple terminals are written in quotes.

```
////////////////////////////////////////////////////////////////////////////////
// statements

prog
  : stmt_list

stmt_list
  : stmt*

stmt
  : '.' // NOOP
  | LOAD SYSTEM? (STRING | ID) '.'?
  | GLOBAL id_list '.'?
  | ASSERT exp '.'?
  | STRUCTURE ID WITH struct_stmts END
  | LET pattern '=' exp '.'?
  | LOOP DO? stmt_list END
  | FOR pattern IN exp DO stmt_list END
  | WHILE exp DO stmt_list END
  | REPEAT DO? stmt_list UNTIL exp '.'?
  | IF exp DO stmt_list (ELIF exp DO stmt_list)* (ELSE DO? stmt_list)? END
  | TRY DO? stmt_list (CATCH pattern DO stmt_list)+ END
  | THROW exp '.'?
  | BREAK '.'?
  | RETURN exp? '.'?
  | function_def
  | exp '.'?
```

(continues on next page)

(continued from previous page)

```
function_def
  : FUNCTION ID body_defs END

body_defs
  : WITH pattern DO stmt_list (WITH pattern DO stmt_list)*

data_stmt
  : DATA ID

struct_stmt
  : data_stmt  '.'?
  | function_def '.'?
  | '.'

struct_stmts
  : struct_stmt*

id_list
  : ID (',' ID)*

////////////////////////////////////////////////////////////////////////////////
// expressions/patterns

exp
  : pattern

pattern
  : PATTERN WITH? exp
  | '%[' exp ']%' binding_list?
  | head_tail

head_tail
  : conditional ('|' exp)?


conditional
  : compound (IF exp (ELSE exp)?)?

compound
  : logic_exp0
      (
         (IS pattern) |
         (IN exp) |
         (TO exp (STEP exp)?) |
      )?

logic_exp0
  : logic_exp1 (OR logic_exp1)*

logic_exp1
  : rel_exp0 (AND rel_exp0)*
```

(continues on next page)

(continued from previous page)

```
rel_exp0
  : rel_exp1 (('==' | '=/=' ) rel_exp1)*

rel_exp1
  : arith_exp0 (('<=' | '<'  | '>=' | '>') arith_exp0)*

arith_exp0
  : arith_exp1 (('+' | '-') arith_exp1)*

arith_exp1
  : call_or_index (('*' | '/') call_or_index)*

call_or_index
  : primary (primary | '@' primary)* (':' pattern)?

////////////////////////////////////////////////////////////////////////////////
// primary expressions/patterns

primary
  : INTEGER
  | REAL
  | STRING
  | TRUE
  | FALSE
  | NONE
  | ID
  | '*' call_or_index  binding_list?
  | NOT call_or_index
  | MINUS call_or_index
  | PLUS call_or_index
  | ESCAPE STRING
  | EVAL primary
  | '(' tuple_stuff ')'
  | '[' list_stuff ']'
  | function_const
  | TYPEMATCH           // TYPEMATCH == '%'<typename>



binding_list
  : BIND binding_list_suffix

binding_list_suffix
   : binding_term
   | '[' binding_term (',' binding_term)* ']'

binding_term
  : ID (AS ID)?

tuple_stuff
  : exp (',' exp?)*
  | empty
```

(continues on next page)

(continued from previous page)

```
list_stuff
  : exp (',' exp)*
  | empty

function_const
  : LAMBDA body_defs
```

1.5.2 Notes on Function Argument Notation

Functions in Asteroid are multi-dispatch functions and therefore can be called with a variety of input configurations. This is reflected in the documentation of built-in functions and functions belonging to modules: when a function can be called with different input argument configurations then the documentation reflects this by providing different argument configuration separated by a '|' symbol. E.g.,

> list @pop () | ix:%integer

indicating that the list member function `pop` can be called either with the empty argument `()` or with a single integer value.

1.5.3 Builtin Functions

getid x
: Returns a unique id of any Asteroid object as an integer.

hd x:%list
: Returns the first element of a list. It is an error to apply this function to an empty list.

isdefined x:%string
: Returns true if a variable or type name is defined in the current environment otherwise it returns false. The variable or type name must be given as a string.

len x
: Returns the length of x. The function can only be applied to lists, strings, tuples, or structures.

range stop:%integer | (start:%integer, stop:%integer) | (start:%integer, stop:%integer, inc:%integer)
: Compute a list of values depending on the input values:

1. If only the stop value is given then the list [0 to stop-1] is returned.
2. If the start and stop values are given then the list [start to stop-1] is returned.
3. If in addition to the start and stop values the inc values is given then the list [start to stop-1 step inc] is returned.

tl x:%list
: Returns the rest of the list without the first element. It is an error to apply this function to an empty list.

1.5.4 List and String Objects

In Asteroid, both `lists` and `strings,` are treated like objects in the OO sense. Due to this, they have member functions that can manipulate the contents of those objects.

Lists

A **list** is a structured data type that consists of square brackets enclosing comma-separated values. Member functions on lists can be called on the data structure directly, e.g.:

```
[1,2,3] @length ()
```

Member Functions

list @append item
Adds the item to the end of the list.

list @clear ()
Removes all items from the list.

list @copy ()
Returns a shallow copy of the list.

list @count item
Returns the number of times item appears in the list.

list @extend item
Extend the list by adding all the elements from the item to the list where the item is either a list or a tuple.

list @filter f:%function
Returns a list constructed from those elements for which function f returns true.

list @index item | (item, loc(startix:%integer) | (item, loc(startix:%integer, endix:%integer))
Returns a zero-based index of the first element whose value is equal to item. It throws an exception if there is no such item. The argument loc allows you to specify startix and endix and are used to limit the search to a particular subsequence of the list. The returned index is computed relative to the beginning of the list rather than the startix argument.

list @insert (ix:%integer, item)
Insert the item into the list at the position i. This means that `a@insert(0, x)` inserts x at the front of the list, and `a@insert(a@length(), x)` is equivalent to `a@append(x)`.

list @join join_str:%string
Turns the list into a string using join_str between the elements. The string is returned as the return value from this function.

list @length ()
Returns the number of elements within the list.

list @map f:%function
Applies the function f to each element of the list in place. The modified list is returned.

list @member item
Returns true only if item exists on the list.

list @pop () | ix:%integer
Removes the item at the given position in the list and returns it. If no index is specified removes and returns the last item in the list.

list @reduce f:%function | (f:%function, init)
Reduce the list to a value by applying the function f to all the members of the list. The function f has to be a function with two arguments where the first argument is the accumulator. If no initial value is given then the first element of the list is assumed to be the first accumulator value. In order to illustrate, we have:

```
let value = [1,2] @reduce (lambda with (x,y) do x+y, 0).
assert(value == 3).
```

is equivalent to

```
let l = [1,2].
let value = 0.
for i in range(l@length()) do
      let value = (lambda with (x,y) do x+y) (value,l@i).
end
assert(value == 3).
```

list @remove item
Removes the first element from the list whose value is equal to item. It throws an exception if there is no such item.

list @reverse ()
Reverses the elements of the list in place and returns the reversed list.

list @shuffle ()
Creates a random permutation of the list in place and returns the randomized list.

list @sort () | reverse:%boolean
Sorts the items of the list in place and returns the sorted list. If the boolean reverse is set to true then the sorted list is reversed.

Strings

A string is a sequence of characters surrounded by double quotes. In Asteroid, single characters are represented as single character strings. Similar to lists the member functions of strings can be called directly on the data structure itself, e.g.:

```
"Hello there" @length ()
```

Member Functions

string @explode ()
Returns the string as a list of characters.

string @flip ()
Returns a copy of the string with its characters in the reverse order.

string @index item:%string | (item:%string, loc(startix:%integer)) | (item:%string, loc(startix:%integer, endix:%integer))
Returns an integer index of the item in the string or none if item was not found. The argument loc allows you to specify startix and endix and are used to limit the search to a particular substring of the string. The returned index is computed relative to the beginning of the full string rather than the startix.

string @length ()
Returns the number of characters within the string.

string @replace (old:%string, new:%string) | (old:%string, new:%string, count:%integer)
Return a copy of the string with all occurrences of regular expression old replaced by the string new. If the argument count is given, only the first count occurrences are replaced.

string @split () | sep:%string | (sep:%string, count:%integer)
Return a list of the words in the string, using sep as the delimiter. If count is given then at most count splits are done (thus, the list will have at most count+1 elements). If count is not specified or -1, then there is no limit on the number of splits (all possible splits are made). Consecutive delimiters are not grouped together and are deemed to delimit empty strings. For example:

```
let s = "1,,2" @split ",".
assert (s == ["1", "", "2"]).
```

The sep argument may consist of multiple characters. For example:

```
let s = "1<>2<>3" @split "<>".
assert (s == ["1", "2", "3"]).
```

Splitting an empty string with a specified separator returns `[""]`. If sep is not specified or is None, a different splitting algorithm is applied: consecutive whitespace is regarded as a single separator, and the result will contain no empty strings at the start or end if the string has leading or trailing whitespace. Consequently, splitting an empty string or a string consisting of just whitespace with a none separator returns `[]`.

string @tolower ()
Returns a copy of the string in all lower case letters.

string @toupper ()
Returns a copy of the string in all upper case letters.

string @trim () | what:%string
Returns a copy of the string with the leading and trailing characters removed. The what argument specifies the set of characters to be removed. If omitted trim defaults to removing whitespace. The what argument is not a prefix or suffix; rather, all combinations of its characters are stripped.

1.5.5 Asteroid Modules

There are a number of system modules that can be loaded into an Asteroid program using `load system <module name>`. The modules are implemented as objects where all the functions of that module are member functions of that module object. For example, in the case of the `io` module we have `println` as one of the member functions. To call that function:

```
load system io.
io @println "Hello there!".  -- println is a member function of the io module
```

bitwise

This module defines bitwise operations on integers. It supports the following functions,

bitwise @band (x:%integer, y:%integer)
Performs the bitwise AND operation and returns the result as an integer.

bitwise @bclearbit (x:%integer, i:%integer)
Clear the ith bit in x and returns the result as an integer.

bitwise @blrotate (x:%integer, i:%integer)
Performs the bitwise left rotate operation by i bits and returns the result as an integer.

bitwise @blshift (x:%integer, y:%integer)
Performs the bitwise left shift operation where x is shifted by y bits and returns the result as an integer.

bitwise @bnot x:%integer
Performs the bitwise NOT operation and returns the result as an integer.

bitwise @bor (x:%integer, y:%integer)
Performs the bitwise OR operation and returns the result as an integer.

bitwise @brrotate (x:%integer, i:%integer)
Performs the bitwise right rotate operation by i bits and returns the result as an integer.

bitwise @brshift (x:%integer, y:%integer)
Performs the bitwise right shift operation where x is shifted by y bits and returns the result as an integer.

bitwise @bsetbit (x:%integer, i:%integer)
Sets the ith bit in x and returns the result as an integer.

bitwise @bsize x:%integer
Returns the bit size of x.

bitwise @bxor (x:%integer, y:%integer)
Performs the bitwise XOR operation and returns the result as an integer.

hash

This module implements a hash for key-value pairs. It supports the following functions,

hash @hash ()
Returns a new hash object of type __HASH__.

__HASH__ @aslist ()
Returns the hash as a list of key-value pairs.

__HASH__ @get key
Return the value associated with the given key as long as it can be found otherwise an exception will be thrown.

__HASH__ @insert (key, value) | pairs:%list
Given a pair of the format (key, value) insert it into the table. Given a list of the format:

```
[(key1, val1), (key2, val2), ...]
```

insert all the key-value pairs on the list into the hash.

io

This module implements Asteroid's I/O system. The module defines three I/O streams,

1. __STDIN__ - the standard input stream.
2. __STDOUT__ - the standard output stream.
3. __STDERR__ - the standard error stream.

Furthermore, the module supports the following functions,

io @close file:%__FILE__
Closes the file where file is a file descriptor of type __FILE__.

io @input () | prompt:%string
Ask the user for input from __STDIN__. The input is returned as a string. If prompt is given it is printed and then input is read from terminal.

io @open (name:%string, mode:%string)
Returns a file descriptor of type __FILE__. The mode string can be "r" when the file will only be read, "w" for only writing (an existing file with the same name will be erased), and "a" opens the file for appending; any data written to the file is automatically added to the end. Finally, "r+" opens the file for both reading and writing.

io @print item
Prints item to the terminal (__STDOUT__). No implicit newline is appended to the output.

io @println item
Prints item to the terminal (__STDOUT__) with an implicit newline character.

io @read () | file:%__FILE__
Read a file and return the contents as a string. If no file is given the __STDIN__ stream is read.

io @readln () | file:%__FILE__
Reads a line of input from a file and returns it as a string. If no file is given the __STDIN__ stream is read.

io @write what:%string | (file:%__FILE__, what:%string)
Write what to a file. If file is not given then it writes to the __STDOUT__ stream.

io @writeln what:%string | (file:%__FILE__, what:%string)
Write what to a file and append a newline charater. If file is not given then it writes to __STDOUT__.

math

The math module implements mathematical constants and functions. An example:

```
load system io.
load system math.

let x = math @sin( math @pi / 2 ).
io @println("The sine of pi / 2 is " + x + ".").
```

Constants

math @pi
The mathematical constant = 3.141592..., to available precision.

math @e
The mathematical constant e = 2.718281..., to available precision.

Power and logarithmic functions

math @exp x:%integer
Returns e raised to the power x, where e = 2.718281... is the base of the natural logarithm.

math @log x | (x, base:%integer)
If only argument x is the input, return the natural logarithm of x (to base e). If two arguments, (x, base:%integer), are given as input, return the logarithm of x to the given base, calculated as log(x)/log(base).

math @pow (b, p:%integer)
Return b raised to the power p. The return type depends on the type of the base.

math @sqrt x
Return the square root of x as a real.

Number-theoretic and representation functions

math @abs x
Return that absolute value of x. The return type depends on the type of x.

math @ceil x:%real
Returns the ceiling of x: the smallest integer greater than or equal to x.

math @floor x:%real
Returns the floor of x: the largest integer less than or equal to x.

math @gcd (a:%integer, b:%integer)
Returns the greatest common denominator that both integers share.

math @isclose (a:%real, b:%real) | (a:%real, b:%real, t:%real)
Return true if the values a and b are close to each other and false otherwise. Default tolerance is 1e-09. An alternative tolerance can be specified with the t argument.

math @mod (v,d)
Implements the modulus operation. Returns the remainder of the quotient v/d.

Trigonometric functions

math @acos x
Returns the arc cosine of x in radians. The result is between 0 and pi.

math @asin x
Returns the arc sine of x in radians. The result is between -pi/2 and pi/2.

math @atan x
Returns the arc tangent of x in radians. The result is between -pi/2 and pi/2.

math @cos x
Returns the cosine of x radians.

math @sin x
Returns the sine of x radians.

math @tan x
Returns the tangent of x radians.

Hyperbolic functions

math @acosh x
Returns the inverse hyperbolic cosine of x.

math @asinh x
Returns the inverse hyperbolic sine of x.

math @atanh x
Returns the inverse hyperbolic tangent of x.

math @cosh x
Returns the hyperbolic cosine of x.

math @sinh x
Returns the hyperbolic sine of x.

math @tanh x
Returns the hyperbolic tangent of x.

Angular conversion

math @degrees x
Converts angle x from radians to degrees.

math @radians x
Converts angle x from degrees to radians.

os

This module provides a portable way of using operating system dependent functionality.

Process Parameters

os @argv
The list of command line arguments passed to an Asteroid script. argv[0] is the name of the Asteroid script (it is operating system dependent whether this is a full pathname or not). In interactive mode argv[0] will be the empty string.

os @env
A hash table where keys and values are strings that represent the process environment. For example,

os @env @get "HOME"

is the pathname of your home directory (on some platforms), and is equivalent to getenv("HOME") in C.

os @platform
This string contains a platform identifier.

Functions

os @basename path:%string
Return the base name of pathname path. This is the second element of the pair returned by passing path to the function split. Note that the result of this function is different from the Unix basename program; where basename for '/foo/bar/' returns 'bar', the basename function returns an empty string ("").

os @chdir path:%string
Change the current working directory to path.

os @dirname path:%string
Return the directory name of pathname path. This is the first element of the pair returned by passing path to the function split.

os @exists path:%string
Return true if path refers to an existing path or an open file descriptor. Returns false for broken symbolic links. On some platforms, this function may return False if permission is not granted to execute stat on the requested file, even if the path physically exists.

os @exit () | v:%integer | msg:%string
Signaling an intention to exit the interpreter. When an argument value other than none is provided it is considered a status value. If it is an integer, zero is considered "successful termination" and any nonzero value is considered "abnormal termination" by shells and the like. Most systems require it to be in the range 0–127, and produce undefined results otherwise. Some systems have a convention for assigning specific meanings to specific exit codes, but these are generally underdeveloped; Unix programs generally use 2 for command line syntax errors and 1 for all other kind of errors. If none is given as an argument value then is it is considered to be a successful exit equivalent to passing a zero. If a string is passed then it is printed printed to __STDERR__ and results in an exit code of 1. In particular, sys.exit("some error message") is a quick way to exit a program when an error occurs.

os @getdir ()
Return a string representing the current working directory.

os @getpathtime path:%string | (path:%string,flag:%boolean)
Returns a triple with (creation, access, modification) times. By default the return value is a triple of real numbers giving the number of seconds since 1/1/1970. If the flag is set to true then a triple of strings is returned where each string represents the respective local time. Throws an exception if the file does not exist or is inaccessible.

os @getsize path:%string
Return the size, in bytes, of path. Throws exception if the file does not exist or is inaccessible.

os @isfile path:%string
Return true if path is an existing regular file. This follows symbolic links.

os @isdir path:%string
Return true if path is an existing directory. This follows symbolic links.

os @join (path1:%string,path2:%string)
Join path1 and path2 components intelligently. The return value is the concatenation of path and any members of *paths with exactly one directory separator following each non-empty part except the last, meaning that the result will only end in a separator if the last part is empty. If the second component is an absolute path, the first component is thrown away.

On Windows, the drive letter is not reset when an absolute path component (e.g., r'foo') is encountered. If a component contains a drive letter, all previous components are thrown away and the drive letter is reset. Note that since there is a current directory for each drive, os.path.join("c:", "foo") represents a path relative to the current directory on drive C: (c:foo), not c:foo.

os @split path:%string
Split the pathname path into a pair, (head, tail) where tail is the last pathname component and head is everything leading up to that. The tail part will never contain a slash; if path ends in a slash, tail will be empty. If there is no slash in path, head will be empty. If path is empty, both head and tail are empty. Trailing slashes are stripped from head unless it is the root (one or more slashes only). Also see the functions dirname and basename.

os @splitdrive path:%string
Split the pathname path into a pair (drive, tail) where drive is either a mount point or the empty string. On systems which do not use drive specifications, drive will always be the empty string. In all cases, drive + tail will be the same as path.

On Windows, splits a pathname into drive/UNC sharepoint and relative path.

If the path contains a drive letter, drive will contain everything up to and including the colon.

os @splitext path:%string
Split the pathname path into a pair (root, ext) such that root + ext == path, and the extension, ext, is empty or begins with a period and contains at most one period. If the path contains no extension, ext will be the empty string.

os @syscmd cmd:%string
Execute a command in a subshell. This is implemented by calling the Standard C function system, and has the same limitations. If command generates any output, it will be sent to the interpreter standard output stream. The C standard does not specify the meaning of the return value of the C function, so the return value of this function is system-dependent.

pick

The pick module implements pick objects that allow a user to randomly pick items from a list of items using the pickitems function. An example:

```
load system io.
load system pick.

let po = pick @pick([1 to 10]).
let objects = po @pickitems 3.
io @println objects.
```

pick @pick l:%list
Construct a pick object of type __PICK__.

__PICK__ @pickitems () | n:%integer
Return items randomly picked from the list l. If no input is provided then pickitems will return a single, randomly picked item from the list. If an integer value n is given then a list of n randomly picked items from the list l is returned. The picked item list is constructed by sampling the list l with replacement.

random

The random module implements random number generation.

random @randint (lo:%integer,hi:%integer) | (lo:%real,hi:%real)
Return a random value N in the interval lo <= N <= hi. The type of the random value depends on the types of the values specifying the interval. If the interval is specified with integers then a random integer value is returned. If the interval is specified with real numbers then a real value is is returned, and for everything else an exception is thrown.

random @random ()
Return a random real number in the range [0.0, 1.0).

random @seed x:%integer
Provide the seed value x for the random number generator.

set

The set module implements Asteroid sets as lists. Unlike lists, sets do not have repeated elements. Use the set member function toset to turn any list into a list that represents a set (remove repeated items).

set @diff (a:%list,b:%list)
Return the difference set between sets a and b.

set @intersection (a:%list,b:%list)
Return the intersection of sets a and b.

set @toset l:%list
Return list l as a set by removing repeated elements.

set @union (a:%list,b:%list)
Return the union of sets a and b.

set @xunion (a:%list,b:%list)
Return the elements in a or b but not both.

sort

The sort module defines a parameterized sort function over a list. The sort function makes use of a user-defined order predicate on the list's elements to perform the sort. The QuickSort is the underlying sort algorithm. The following is a simple example:

```
load system io.
load system sort.
let sl = sort @sort((lambda with (x,y) do true if x<y else false),
                    [10,5,110,50]).
io @println sl.
```

prints the sorted list:

```
[5,10,50,110]
```

sort @sort (p:%function,l:%list)
Returns the sorted list l using the predicate p.

stream

The stream module implements streams that allow the developer to turn any list into a stream supporting interface functions like peeking ahead or rewinding the stream. A simple use case:

```
load system io.
load system stream.

let s = stream @stream [1 to 10].
while not s @eof() do
   io @print (s @get() + " ").
end
io @println "".
```

which outputs:

```
1 2 3 4 5 6 7 8 9 10
```

stream @stream l:%list
Returns a stream object of type __STREAM__.

__STREAM__ @append x
Adds x to the end of the stream.

__STREAM__ @eof ()
Returns true if the stream does not contain any further elements for processing. Otherwise it returns false.

__STREAM__ @get ()
Returns the current element and moves the stream pointer one ahead. Returns none if no elements left in stream.

__STREAM__ @map f:%function
Applies function f to each element in the stream.

__STREAM__ @peek ()
Returns the current element available on the stream otherwise it returns none.

__STREAM__ @rewind ()
Resets the stream pointer to the first element of the stream.

type

The type module defines type related functions and structures. Here is a program that exercises some of the string formatting options:

```
load system io.
load system type.
load system math.

-- if the width specifier is larger than the length of the value
-- then the value will be right justified
let b = type @tostring(true,type @stringformat(10)).
io @println b.

let i = type @tostring(5,type @stringformat(5)).
io @println i.

-- we can format a string by applying tostring to the string
let s = type @tostring("hello there!",type @stringformat(30)).
io @println s.

-- for floating point values: first value is width, second value precision.
-- if precision is missing then value is left justified and zero padded on right.
let r = type @tostring(math @pi,type @stringformat(6,3)).
io @println r.
```

The output of the program is,

```
     true
   5
                hello there!
3.142
```

Notice the right justification of the various values within the given string length.

Type Conversion

type @tobase (x:%integer,base:%integer)
Represents the given integer x as a numeral string in different bases.

type @toboolean x
Interpret x as a Boolean value.

type @tointeger (x:%string,base:%integer) | x
Converts a given input to an integer. If a base value is specified then the resulting integer is in the corresponding base.

type @toreal x
Returns the input as a real number.

type @tostring x | (x,type @stringformat(width:%integer,precision:%integer,scientific:%boolean))
Converts an Asteroid object to a string. If format values are given, it applies the formatting to the string object.

Type Query Functions

type @islist x
Returns true if x is a list otherwise it will return false.

type @isnone x
Returns true if x is equal to the value none.

type @isscalar x
Returns true if x is either an integer or a real value.

type @gettype x
Returns the type of x as a string.

A simple example program using the `gettype` function,

```
load system type.

let i = 1.
assert(type @gettype(i) == "integer").
```

util

The util module defines utility functions and structures that don't really fit into any other modules.

util @achar x
Given a decimal ASCII code x, return the corresponding character symbol.

util @ascii x:%string
Given a character x, return the corresponding ASCII code of the first character of the input.

util @cls ()
Clears the terminal screen.

util @copy x
Given the object x, make a deep copy of it.

util @ctime x:%real
Given a real value representing seconds since 1/1/1970 this function converts it to a suitable string representation of the date.

type @sleep x
Sleep for x seconds where the x is either an integer or real value.

type @time ()
Returns the local time as a real value in secs since 1/1/1970.

type @unzip x:%list
Given a list of pairs x this function will return a pair of lists where the first component of the pair is the list of all the first components of the pairs of the input list and the second component of the return list is a list of all the second components of the input list.

type @zip (list1:%list,list2:%list)
Returns a list where element i of the list is the tuple (list1@i,list2@i).

vector

The vector defines functions useful for vector arithmetic. Vectors are implemented as lists. Here is a simple example program for the `vector` module:

```
load system io.
load system vector.

let a = [1,0].
let b = [0,1].

io @println (vector @dot (a,b)).
```

which prints the value `0`.

vector @add (a:%list,b:%list)
Returns a vector that contains the element by element sum of the input vectors a and b.

vector @dot (a:%list,b:%list)
Computes the dot product of the two vectors a and b.

vector @mult (a:%list,b:%list)
Returns the element by element vector multiplication of vectors a and b.

vector @op (f:%function,a:%list,b:%list) | (f:%function,a:%list,b if type @isscalar(b)) | (f:%function,a if type @isscalar(a),b:%list)
Allows the developer to vectorize any function f. Applying scalar values to vectors is also supported by this function.

vector @sub (a:%list,b:%list)
Returns the element by element difference vector.

1.5.6 Interfacing Asteroid with Python

Asteroid allows integration with Python in one of two ways. First, we can call the Asteroid interpreter from within a Python program and second, we can embed Python code directly within an Asteroid program. We start with looking at calling the Asteroid interpreter from Python.

Calling Asteroid from Python

Calling Asteroid from within a Python program is nothing more than calling Asteroid's `interp` function with a string representing an Asteroid program as its argument. In order to make this work you will have to make sure that the Python interpreter can find the Asteroid modules. Here we assume that you have installed Asteroid with the `pip` installer. Once you have installed Asteroid you will have to point the `PYTHONPATH` environment variable to the directory where `pip` installed the Asteroid modules. You can easily find out where the modules are installed by issuing the `show` command,

```
ubuntu$ pip3 show asteroid-lang
Name: asteroid-lang
Version: 1.1.3
Summary: A pattern-matching oriented programming language.
Home-page: https://asteroid-lang.org
```

(continues on next page)

(continued from previous page)

```
Author: University of Rhode Island
Author-email: lutzhamel@uri.edu
License: None
Location: /home/ubuntu/.local/lib/python3.8/site-packages
Requires: numpy, pandas, matplotlib
Required-by:
ubuntu$
```

The `Location` field tells us where the Asteroid modules have been installed. Under Ubuntu we can now create an environment variable that points to that directory as follows,

```
ubuntu$ export PYTHONPATH=/home/ubuntu/.local/lib/python3.8/site-packages
ubuntu$
```

Now that Python knows how to find the Asteroid modules we can import the Asteroid interpreter into any Python program using,

```
from asteroid.interp import interp
```

where the `interp` function takes a string representing of an Asteroid program as an argument. Let's test drive this in the Python interactive shell,

```
ubuntu$ python3
Python 3.8.10 (default, Nov 26 2021, 20:14:08)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from asteroid.interp import interp
>>> interp('load system io. io @println "Hello, World!".')
Hello, World!
>>>
```

For more detailed information on the `interp` function do a `help(interp)` at the interactive Python prompt. Even though we have shown this example under Linux, analogous approaches should work on both Windows and macOS.

Not only can we execute the Asteroid interpreter from Python but we can also access its state to look up the results of a computation for example. Here is a slight variation of the program above where the Asteroid program computes the string value containing the greeting but we are actually printing the value from Python,

```
# import Asteroid modules
from asteroid.interp import interp
from asteroid.state import state

# run the interpreter to compute the greeting string
interp('let s = "Hello World!".')

# retrieve the greeting string from the interpreter state
# notice the pair of values a symbol table lookup produces:
# one for the type of the value and one for the actual value
(type,val) = state.symbol_table.lookup_sym('s')
print(type)
print(val)
```

The program prints out,

```
string
Hello World!
```

Embedding Python into an Asteroid Program

Using Asteroid's `escape` expression allows us to embed arbitray Python code into an Asteroid program,

```
-- Printing hello once from each environment

-- print hello from Asteroid
load system io.
io @println "Hello World from Asteroid!".

-- print hello from Python
escape
"
print('Hello World from Python!')
".
```

Please note that the format of the Python code in the escaped string should follow the same guidelines as the Python code embedded in strings handed to the Python exec function.

Not only does the `escape` expression give you access to the Python environment but it also gives you access to the current Asteroid interpreter state including its symbol table. That means we can access any variable defined in the Asteroid environment from Python,

```
let s = "Hello World!".

escape
"
(type, val) = state.symbol_table.lookup_sym('s')
print(type)
print(val)
".
```

Notice that a symbol table lookup produces a pair of values where the first value represents the type of the value stored in the symbol table and the second value is the actual value stored. In this case our program prints out,

```
string
Hello World!
```

That is the type of the value is a string and the value is the actual string `Hello World!`.

Since `escape` represents an expression we can also return values from the Python code using a special `__retval__` variable. The only trick is that we have to remember that values in Asteroid are pairs consisting of type information and values. Here is a very simple program that exercises that part of the Python API,

```
load system io.

let i = escape
"
global __retval__  # access the return value register

```

(continues on next page)

(continued from previous page)

```
__retval__ = ('integer', 101)
".

io @println i.
```

This program will print out the value 101 from Asteroid even though that value was created within the Python environment. Notice that we have to access the return value register `__retval__` with the `global` statement in the Python code.

We can pull all of this together and write an Asteroid function that performs its computations in Python,

```
function inc with i do return escape
"
# access return value register
global __retval__
# lookup the value of the formal argument
(type, val) = state.symbol_table.lookup_sym('i')

# only perform the increment if the value is an integer
if type != 'integer':
   raise ValueError('not an integer')
else:
   __retval__ = (type, val+1)
".
end

-- call inc and make sure the result is correct
let k = inc(1)
assert(k == 2).
```

Of course the function is just an illustration of how to use the Python API. This type of computation is much easier to express in Asteroid directly,

```
function inc
   with i:%integer do
      i+1
   end

let k = inc(1)
assert(k == 2).
```

The Foreign Type Tag

When working in the hybrid Asteroid-Python environment it is sometimes useful to be able to embed values in an Asteroid program that have no direct representation in Asteroid. This is where the `foreign` type tage comes into play. Consider the following program that uses Pandas dataframes within an Asteroid program,

```
--------------------------------------------------------------------------
function pack
--------------------------------------------------------------------------
-- this function packs four real values into a Pandas dataframe
with (a:%real,b:%real,c:%real,d:%real) do return escape
```

(continues on next page)

(continued from previous page)

```
"
global __retval__
# we can ignore type info here because we checked it above
(_, aval) = state.symbol_table.lookup_sym('a')
(_, bval) = state.symbol_table.lookup_sym('b')
(_, cval) = state.symbol_table.lookup_sym('c')
(_, dval) = state.symbol_table.lookup_sym('d')

import pandas as pd
df = pd.DataFrame({'x':[aval,bval], 'y':[cval,dval]})
__retval__ = ('foreign', df)
"
end

--------------------------------------------------------------------------
function dump
--------------------------------------------------------------------------
-- dump the Pandas dataframe to stdout
with df do escape
"
(dftype, dfval) = state.symbol_table.lookup_sym('df')
if dftype != 'foreign':
   raise ValueError('expected data frame')
print(dfval)
"
end

--------------------------------------------------------------------------
function access
--------------------------------------------------------------------------
-- access an element of the Pandas dataframe at row r and column c
with (df,r:%integer,c:%integer) do return escape
"
global __retval__
(dftype, dfval) = state.symbol_table.lookup_sym('df')
if dftype != 'foreign':
   raise ValueError('expected data frame')
# we can ignore type info here because we checked it above
(_, rval) = state.symbol_table.lookup_sym('r')
(_, cval) = state.symbol_table.lookup_sym('c')
# make sure the ret value conforms to the Asteroid value structure
__retval__ = ('real', dfval.iloc[rval,cval])
"
end

--------------------------------------------------------------------------
function sum
--------------------------------------------------------------------------
-- sum down the columns of the dataframe and return a pair of values,
-- one component for each column
with (df) do return escape
"
```

(continues on next page)

(continued from previous page)

```
global __retval__
(dftype, dfval) = state.symbol_table.lookup_sym('df')
if dftype != 'foreign':
   raise ValueError('expected data frame')
# sum the value down the columns
sum = list(dfval.sum(axis=0))
# construct our tuple, note the type information
__retval__ = ('tuple', [('real',sum[0]),('real',sum[1])])
"
end

--------------------------------------------------------------------------
-- exercise our machinery
let df = pack(1.0,2.0,3.0,4.0).
dump(df).
assert(access(df,1,1) == 4).
assert(sum(df) == (3.0,7.0)).
```

The dump function generates the following output,

```
     x    y
0  1.0  3.0
1  2.0  4.0
```

Pandas dataframes are not directly usable in Asteroid but by writing thin Python wrappers and taking advantage of the `escape` expression the `foreign` type tag we can embed Pandas functionality into Asteroid. As an additional step we could wrap these individual functions into a `structure` with the dataframe as a data member and the functions as member functions of that structure. As an example of this approach see the dataframe.ast system module.

1.6 Asteroid in Action

This document was inspired by Andrew Shitov's excellent book Using Raku: 100 Programming Challenges Solved with the Brand-New Raku Programming Language. Here we use Asteroid to solve these programming challenges.

1.6.1 Section: Using Strings

Challenge: Hello, World!

> Print 'Hello, World!'

The canonical `Hello, World!` program. The easiest way to write this in Asteroid is,

```
load system io.

io @println "Hello, World!".
```

Output:

```
Hello, World!
```

Challenge: Greet a person

> Ask a user for their name and greet them by printing 'Hello, <Name>!'

Here is our first solution using a separate function for each of the steps,

```
load system io.

io @print ("Enter your name: ").
let name = io @input().
io @print ("Hello, "+name+"!").
```

Letting the function `input` do the prompting,

```
load system io.

let name = io @input("Enter your name: ").
io @println ("Hello, "+name+"!").
```

Doing everything in one step,

```
load system io.

io @println ("Hello, "+io @input("Enter your name: ")+"!").
```

Challenge: String length

> Print the length of a string.

In order to print the length of a string we can use the function `len` available in the `util` module,

```
load system io.

load "util".
io @println (len("Hello!")).
```

Output:

```
6
```

We can also use the string member function `length` in order to compute the length of the string,

```
load system io.

io @println ("Hello!" @length()).
```

Output:

```
6
```

Challenge: Unique digits

> Print unique digits from a given integer number.

In order to accomplish this we take advantage of the string `explode` function and the `sort` function on lists. Finally we use the `reduce` function to map a list with repeated digits to a list with unique digits,

```
load system io.

function unique with (x,y) do
    if not (x @member(y)) do
        return x @append(y).
    else do
        return x.
    end
end

let digits = "332211" @explode()
                      @sort()
                      @reduce(unique,[]).
io @println digits.
assert(digits == ["1","2","3"]).
```

Output:

```
[1,2,3]
```

Probably the most noteworthy characteric about this program is the `reduce` function. The `reduce` function applies a binary function to a list. The first argument of the binary function acts like an accumulator, and the second argument gets instantiated with the elements of the list to be processed. In our function `unique`, the variable `x` is the accumulator with an initial value of `[]`. The function tests whether the element `y` is in the list. If it is not, then it adds it to the list. Otherwise, it just returns the accumulator unchanged.

1.6.2 Section: Modifying String Data

Challenge: Reverse a string

> Print a string in the reversed order from right to left.

We use the `explode` function to turn a string into a list of characters. Then, we reverse the list and turn it back into a string using the `join` function,

```
load system io.

let str = "Hello, World!" @explode()
                          @reverse()
                          @join("").
io @println str.
assert(str == "!dlroW ,olleH").
```

Output:

```
!dlroW ,olleH
```

Challenge: Removing blanks from a string

> Remove leading, trailing, and double spaces from a given string.

```
load system io.
let str = "   Hello  ,   World    !   " @trim()
                                        @replace("  ","").
io @println str.
assert(str == "Hello, World!").
```

Output:

```
Hello, World!
```

Challenge: Camel case

> Create a camel-case identifier from a given phrase.

In this task, we will form the `CamelCase` variable for names from a given phrase. Names created in this style are built of several words, each of which starts with a capital letter.

```
load system io.

function title with w do
    let letter_list = w @tolower()
                        @explode().
    let first_letter = letter_list @0
                                   @toupper().
    if letter_list @length() > 1 do
        let title_case = ([first_letter] + letter_list @[1 to letter_list@length()-1])␣
↪@join("").
    else
        let title_case = first_letter.
    end
    return title_case.
end

let str = "once upon a time".
let camel_str = str @split()
                    @map(title)
                    @join("").
io @println camel_str.
assert(camel_str == "OnceUponATime").
```

Output:

```
OnceUponATime
```

Challenge: Incrementing filenames

> Generate a list of filenames like file1.txt, file2.txt, etc.

```
load system io.

let root = "file".
let ext = ".txt".

for i in 1 to 5 do
    io @println (root+i+ext).
end
```

Output:

```
file1.txt
file2.txt
file3.txt
file4.txt
file5.txt
```

Challenge: Random passwords

> Generate a random string that can be used as a password.

In our solution we take advantage of Asteroid's `Pick` object. The `Pick` object maintains a list of items that we can randomly select from using the `pick` member function. As input to the `Pick` object, we compute a bunch of lists of characters that are useful for password construction. The function `achar` converts a decimal ASCII code to a single character string.

```
load system io.
load system type.
load system util.
load system pick.
load system random.

random @seed(42).

-- make up lists of symbols useful for password construction
let int_list = [0 to 9] @map(type @tostring).
let lc_list = [97 to 122] @map(util @achar). -- lower case characters
let uc_list = [65 to 90] @map(util @achar). --upper case characters
let sp_list = ["!","_","#","$","%","*"].
-- build the overall pick list of symbols
let pick_list = int_list+lc_list+uc_list+sp_list.

-- generate the password and print it.
let pwd = pick @pick pick_list @pickitems 15 @join("").
io @println pwd.

assert (pwd == "e3zvshdbS43brt#")
```

Output:

```
e3zvshdbS43brt#
```

Challenge: DNA-to-RNA transcription

> Convert the given DNA sequence to a compliment RNA.

We'll not dig deep into the biology aspect of the problem. For us, it is important that the DNA is a string containing the four letters A, C, G, and T, and the RNA is a string of A, C, G, and U. The transformation from DNA to RNA happens according to the following table:

```
DNA: A C G T
RNA: U G C A
```

We will solve this programming problem using Asteroid's first-class patterns. We could have solved this with just testing equality on DNA characters. However, using first-class patterns is more general and can be applied to problems with a more structured mapping relationship.

```
load system io.

let dna2rna_table =
    [
      ("A","U"),
      ("C","G"),
      ("G","C"),
      ("T","A")
    ].

function dna2rna with x do
    for (dna,rna) in dna2rna_table do
        if x is *dna do
            return rna.
        end
    end
    throw Error("unknown dna char "+x).
end

let dna_seq = "ACCATCAGTC".
let rna_seq = dna_seq @explode()
                      @map(dna2rna)
                      @join("").
io @println rna_seq.

assert(rna_seq == "UGGUAGUCAG").
```

Output:

```
UGGUAGUCAG
```

Challenge: Caesar cipher

> Encode a message using the Caesar cipher technique.

The Caesar code is a simple method of transcoding the letters of the message so that each letter is replaced with the letter that occurs in the alphabet N positions earlier or later. For example, if N is 4, then the letter e becomes a, f is transformed to b, etc. The alphabet is looped so that z becomes v, and letters a to d become w to z.

```
load system io.
load system util.

let achar = util @achar.
let ascii = util @ascii.

let encode_table = [119 to 122] @map(achar) + [97 to 118] @map(achar).

function encode with (v:%string) if len(v) == 1 do
    -- only lowercase letters are encoded
    if not (ascii(v) in [97 to 122]) do
        return v.
    else
        return encode_table @(ascii(v)-ascii("a")).
    end
end

function decode with (v:%string) if len(v) == 1 do
    -- only lowercase letters are decoded
    if not (ascii(v) in [97 to 122]) do
        return v.
    else
        return encode_table @(ascii(v)-ascii("w")+4).
    end
end

let message = "hello, world!"
let secret = message @explode()
                     @map(encode)
                     @join("").
io @println secret.

assert (secret == "dahhk, sknhz!")

let decoded_msg = secret @explode()
                         @map(decode)
                         @join("").
io @println decoded_msg.

assert (decoded_msg == "hello, world!")
```

Output:

```
dahhk, sknhz!
hello, world!
```

1.6.3 Section: Text Analysis

Challenge: Plural endings

> Put a noun in the correct form — singular or plural — depending on the number next to it.

In program outputs, it is often required to print some number followed by a noun, for example:

```
10 files found
```

If there is only one file, then the phrase should be `1 file found` instead.

```
load system io.

for n in 0 to 5 do
    io @println (n+" file"+("s " if n>1 or n==0 else " ")+"found").
end
```

Output:

```
0 files found
1 file found
2 files found
3 files found
4 files found
5 files found
```

Challenge: The most frequent word

> Find the most frequent word in the given text.

In our solution we use a hash table to count the number of word occurences.

```
load system io.
load system util.
load system hash.

-- text generated at 'https://www.lipsum.com/'
let text = "Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed
accumsan magna quis risus commodo, et pellentesque dui cursus. Sed quis risus
libero. Cras et mattis libero, eget varius nisi. Phasellus ultrices, augue non
dictum eleifend, nunc elit blandit velit, a viverra risus enim in tellus.
Maecenas quis ante eget turpis rhoncus rhoncus eget ut mauris. Suspendisse nec
erat sed nunc tempus hendrerit. Nunc dictum nunc molestie eleifend tempus.
Praesent cursus lorem diam, sed mattis velit vehicula scelerisque. Nunc iaculis
rhoncus ante. Etiam quam nisi, fermentum et euismod a, vulputate eu elit.
Suspendisse tincidunt ligula quis interdum blandit. Quisque sed aliquam tellus.
Pellentesque ac lacus pulvinar, ornare purus ac, viverra ex. Donec quis pharetra
dolor.

In ac massa tortor. Cras sagittis luctus scelerisque. Morbi a neque sed tortor
ultrices dapibus. Mauris pretium vitae massa non auctor. Cras egestas ex ante,
ac ullamcorper ante dignissim eget. Fusce bibendum justo ut enim luctus, id
volutpat diam lacinia. Mauris sit amet ante risus.
```

(continues on next page)

(continued from previous page)

```
Nullam rhoncus ultricies dui. Etiam vel metus vehicula, pellentesque felis ut,
suscipit nunc. Sed nec interdum lorem. Maecenas odio erat, vestibulum nec
dapibus id, commodo vitae libero. Nulla sed urna sit amet nunc commodo finibus
sed vel elit. Aliquam euismod feugiat nisi quis placerat. Aliquam libero nisl,
ultrices non est at, sagittis hendrerit dui. Quisque id sem lorem. Nam ultricies
metus id ultrices molestie. Pellentesque elementum consequat nibh, nec convallis
lorem ullamcorper in. Etiam vitae mi tellus. Etiam accumsan massa sit amet dolor
tincidunt iaculis. Nam ullamcorper blandit sem id bibendum. Quisque elementum
ipsum ac sapien blandit vehicula."

-- get rid of punctuation, turn to lower case, and split into words.
-- Note: we could have employed richer regular expressions to clean up the text here
let wl = text @replace("\.","")
              @replace(",","")
              @tolower()
              @split().

-- put the words into a hash table, the value is the count of the words
let ht = hash @hash().
for w in wl do
    if not ht @get(w) do
        ht @insert(w,1).
    else do
        ht @insert(w,ht @get(w)+1).
    end
end

-- get the contents of hash table and find the most frequent word
let (keys,values) = util @unzip(ht@aslist()).
let values_sorted = values @copy()
                           @sort(true).
let most_frequent_word = keys @(values @index(values_sorted @0)).
io @println most_frequent_word.

assert (most_frequent_word == "sed").
```

Output:

```
sed
```

Challenge: The longest common substring

> Find the longest common substring in the given two strings.

Let us limit ourselves with finding only the first longest substring. If there are more common substrings of the same length, then the rest are ignored. There are two loops (see also Task 17, The longest palindrome) over the first string (`stra`). These use the index method to search for the substring in the second string (`strb`).

```
load system io.

let stra = "the quick brown fox jumps over the lazy dog".
```

(continues on next page)

(continued from previous page)

```
let strb = "what does the fox say?".
let common = "".

for startix in 0 to stra @length()-1 do
    for endix in startix to stra @length()-1 do
        let s = stra @[startix to endix].
        if strb @index(s) and s @length() > common @length() do
            let common = s.
        end
    end
end

if common do
    io @println ("The longest common substring is '"+common+"'.").
else do
    io @println ("There are no common substrings.").
end

assert (common == " fox ").
```

Output:

```
The longest common substring is ' fox '.
```

Challenge: Anagram test

> Tell if the two words are anagrams of each other.

An anagram is a word, phrase, or name formed by rearranging the letters of another, such as `cinema`, formed from `iceman`.

```
load system io.

let str1 = "cinema".
let str2 = "iceman".

function normalize with str do
    return str @explode()
               @sort()
               @join("").
end

if normalize(str1) == normalize(str2) do
    io @println "Anagrams".
else do
    io @println "Not anagrams".
end

assert (normalize(str1) == normalize(str2)).
```

Output:

```
Anagrams
```

Challenge: Palindrome test

> Check if the entered string is palindromic.

A palindrome is a string that can be read from both ends: left to right or right to left.

```
load system io.

let str = "Was it a rat I saw?".

function clean with str:%string do
    return str @tolower()
               @replace("[^a-z]","").
end

-- only keep lower case letters
let clean_str = clean(str).

-- check if it is palidromic
if clean_str == clean_str @flip() do
    io @println "Palindromic".
else do
    io @println "Not palindromic".
end

assert (clean_str == clean_str @flip()).
```

Output:

```
Palindromic
```

Challenge: The longest palindrome

> Find the longest palindromic substring in the given string.

The main idea behind the solution is to scan the string with a window of varying width. In other words, starting from a given character, test all the substrings of any length possible at that position. Now, extract the substring and do the check similar to the solution of Task 16, Palindrome test. Here, we have to be careful to check the palindrome without taking into account the non-letter characters, but saving the result as part of the original string.

```
load system io.

let str = "Hello, World!".

function clean with str:%string do
    return str @tolower()
               @replace("[^a-z]","").
end

function palindrome_test with str:%string do
```

(continues on next page)

(continued from previous page)

```
    let clean_str = clean(str).
    if clean_str == clean_str @flip() do
        return true.
    else do
        return false.
    end
end

-- create the moving window over the string
let longest_palindrome = "".

for i in 0 to str @length()-2 do
    for j in i+1 to str @length()-1 do
        let str1 = str @[i to j].
        if palindrome_test(str1) and
           str1 @length() > longest_palindrome @length() do
            let longest_palindrome = str1.
        end
    end
end

io @println longest_palindrome.
```

Output:

```
o, Wo
```

Challenge: Finding duplicate texts

> Find duplicate fragments in the same text.

We do this by finding and hashing N-grams after the appropriate preprocessing. We will use `N=3`.

```
load system io.
load system hash.

-- text from "www.lipsum.com"

let str = "Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed
malesuada sapien nec neque suscipit, non rutrum arcu scelerisque. Nam feugiat
sapien porta ipsum accumsan, eget maximus diam volutpat. Pellentesque elementum
in orci quis pretium. Donec dignissim nunc lectus, id ornare urna varius ut.
Praesent semper faucibus vehicula. Aliquam luctus sapien at lorem malesuada,
eget suscipit felis facilisis. Suspendisse velit lectus, mollis sit amet tempor
eget, faucibus ut nulla. Vestibulum et elementum dolor, a vehicula ipsum. Morbi
ut fringilla nisi. Fusce congue rutrum orci nec porta. Ut laoreet justo vel
turpis sodales vehicula. Nulla porttitor nisl id odio eleifend sodales.

Suspendisse blandit tristique enim id laoreet. Etiam vel aliquet dui, quis
tempus magna. Donec blandit volutpat felis egestas tincidunt. Integer placerat
luctus mi non pharetra. Donec aliquet nisl orci, egestas elementum nunc bibendum
a. Morbi nec risus aliquet, viverra nunc in, molestie odio. Curabitur
```

(continues on next page)

(continued from previous page)

```
pellentesque, ante eget dictum aliquam, felis leo bibendum libero, vel bibendum
lorem velit eget ex. Lorem ipsum dolor sit amet, consectetur adipiscing elit.
Vestibulum pretium tellus quis ante vulputate, pretium tincidunt ipsum dapibus.
Praesent congue, ipsum ut sagittis tempus, lacus nisi dapibus dui, aliquam porta
metus odio ut neque. Aliquam vitae faucibus dolor. Nulla iaculis lorem non
mauris viverra, ut malesuada nibh aliquam. Nam bibendum sit amet massa in
dignissim. Nam posuere nunc ante, at viverra diam rhoncus vel.

Aliquam mollis sagittis nulla. Maecenas faucibus eu dui eget accumsan.
Suspendisse sit amet fermentum sapien. Nunc vitae mi nibh. Mauris condimentum
vestibulum imperdiet. Quisque at vehicula dui. Integer sit amet volutpat arcu.
Maecenas efficitur leo tortor, non ullamcorper magna tempor non. Sed efficitur
quis metus ut pulvinar. Proin nunc felis, congue sit amet nibh placerat,
tincidunt mattis nunc. Duis efficitur lacus a orci porttitor, sed molestie risus
tempor.

Sed tincidunt ipsum at urna sollicitudin feugiat. Ut mollis orci quis massa
dictum facilisis. Maecenas non elementum mauris. Sed rutrum orci faucibus,
tristique nunc nec, mattis ante. Pellentesque habitant morbi tristique senectus
et netus et malesuada fames ac turpis egestas. In hac habitasse platea dictumst.
Morbi pellentesque dolor sit amet nunc tincidunt, ut rutrum ante vulputate.
Nullam pretium, mi sed condimentum luctus, ipsum nunc dictum lorem, vel
ultricies nibh mi ut sem. Nam volutpat id libero eget mollis.

Vestibulum eget velit eros. Phasellus sit amet vestibulum odio, vel malesuada
quam. Mauris dictum erat eu ligula mollis laoreet. Phasellus ut ante auctor,
hendrerit ipsum et, fermentum magna. Etiam nec eros elementum, consectetur nibh
ac, ullamcorper ligula. Aliquam sed porttitor sapien. Nulla tincidunt, turpis
vitae venenatis aliquet, quam purus elementum diam, in tincidunt orci diam sed
nulla. Cras pellentesque non diam quis sollicitudin. Duis suscipit lectus dui,
eu varius metus pretium sit amet.

Nulla eu ex velit. Ut non justo semper, gravida erat quis, vehicula est.
Suspendisse nunc dui, iaculis id purus sit amet, rutrum commodo lacus. Aenean
consequat turpis a est vestibulum, ac accumsan nibh dapibus. Nam blandit
scelerisque lectus, eu pellentesque arcu ornare non. Fusce ac gravida diam. Ut
in fringilla eros. Sed metus augue, porta quis vehicula at, pellentesque et
mauris. Duis sodales lacus sit amet condimentum placerat. In blandit tristique
nulla eget malesuada. Sed congue finibus neque at semper. Etiam pellentesque
egestas urna, ut lobortis odio euismod et. Phasellus aliquet quam purus, quis
ullamcorper sem mollis eu.

Mauris quis ullamcorper nisi. Aenean quam nulla, sodales eu faucibus in, mattis
a nulla. Nullam pulvinar pretium justo eu mattis. Aliquam rutrum ipsum vitae leo
maximus ultrices. Donec ut pulvinar nisi. Sed pharetra, turpis dictum lobortis
egestas, quam massa venenatis enim, dapibus efficitur dolor mauris eu felis.
Donec vulputate ultrices justo sit amet condimentum. Donec id posuere nulla. In
vestibulum mi in lectus commodo dignissim. Quisque vestibulum egestas arcu sit
amet finibus. Proin commodo aliquet neque quis maximus.

Nulla facilisi. Sed gravida aliquet diam in congue. Mauris vehicula justo ac
sollicitudin laoreet. Mauris enim mi, auctor id magna eget, feugiat sollicitudin
```

(continues on next page)

(continued from previous page)

```
leo. Vivamus ornare ornare commodo. Suspendisse ut dui quis enim porta pretium.
Praesent vitae lacus fermentum, posuere orci ac, imperdiet massa. Nulla
hendrerit id nisl sed maximus. Vivamus commodo lacus eu condimentum bibendum.
Suspendisse porttitor sem eget dolor aliquet congue. Pellentesque tristique
augue at quam hendrerit dignissim. Aenean a congue dui. Vestibulum ante ipsum
primis in faucibus orci luctus et ultrices posuere cubilia curae; Integer ante
lacus, commodo et enim sed, auctor egestas metus.

Aliquam a urna id risus tincidunt rutrum. Nunc facilisis, tortor ac suscipit
aliquam, ante neque tincidunt mi, nec ullamcorper lectus ligula vel urna.
Suspendisse lobortis at felis sit amet facilisis. Pellentesque velit lacus,
porttitor vitae eros rutrum, convallis blandit erat. Pellentesque nec mi
viverra, volutpat dui in, rutrum lacus. Ut non venenatis leo. Praesent
sollicitudin magna porttitor lorem elementum molestie non a turpis. Suspendisse
potenti.

Donec malesuada iaculis laoreet. Nunc ut volutpat ante, ut consequat tortor.
Phasellus posuere, ipsum quis dignissim iaculis, nisl felis ullamcorper ligula,
quis placerat sem sapien nec ante. Cras suscipit ut magna nec lacinia. Donec
ipsum nibh, imperdiet non aliquam eu, maximus id ante. Pellentesque vitae felis
felis. Aliquam et diam sed nulla volutpat vestibulum molestie non lacus.
Praesent porta et lacus auctor fermentum. In hac habitasse platea dictumst.
Aliquam erat volutpat. Etiam at ligula orci. Class aptent taciti sociosqu ad
litora torquent per conubia nostra, per inceptos himenaeos."

let word_list = str @tolower()
                   @replace("[^a-z0-9_]"," ")
                   @split().
let ht = hash @hash().

-- create N-grams
for i in 0 to word_list @length()-3 do
    -- Note: make this code more general
    let n_gram = [word_list@i, word_list@(i+1), word_list @(i+2)] @join(" ").
    -- put the N-gram into a hash table, the value is the count of the N-gram in the␣
↪text.
    if not ht @get(n_gram) do
        ht @insert(n_gram,1).
    else do
        ht @insert(n_gram,ht @get(n_gram)+1).
    end
end

for ((n_gram,cnt) if cnt > 1) in ht @aslist() do
    io @println (n_gram+": "+cnt).
end
```

Output:

```
lorem ipsum dolor: 2
ipsum dolor sit: 2
dolor sit amet: 3
```

(continues on next page)

(continued from previous page)

```
sit amet consectetur: 2
amet consectetur adipiscing: 2
consectetur adipiscing elit: 2
in hac habitasse: 2
hac habitasse platea: 2
habitasse platea dictumst: 2
aliquet quam purus: 2
diam sed nulla: 2
sit amet condimentum: 2
```

1.6.4 Section: Using Numbers

Challenge: Pi

> Print the value of pi.

```
load system io.
load system math. -- definition of pi

io @println (math @pi).
```

Output:

```
3.141592653589793
```

Other constants are also available.

```
load system io.
load system math.

let tau = 2 * math @pi.

io @println (math @e).
io @println tau.
```

Output:

```
2.718281828459045
6.283185307179586
```

Challenge: Factorial!

> Print the factorial of a given number.

By definition, the factorial of a positive integer number N is a product of all the integers numbering from 1 to N, including N. Our first solution is based on the direct implementation of the definition above using the list `reduce` function.

```
load system io.

let n = 3.
```

(continues on next page)

(continued from previous page)

```
let fact = [1 to n] @reduce(lambda with (a,b) do return a*b).
io @println fact.
assert (fact == 6).
```

Output:

```
6
```

Our second solution uses the recursive definition of factorial,

```
     | 1       if  x = 0,
x! = | x(x-1)! if  x > 0,
     | undef   if  x < 0,
```

where $x \in Int$. Here, each case specifies what value the function should return if the predicate applied to the input is true. The last case is of some interest because it states that the function is undefined for negative integers.

```
load system io.

let POS_INT = pattern with (x:%integer) if x > 0.
let NEG_INT = pattern with (x:%integer) if x < 0.

function fact
    with 0 do
        return 1
    with n:*POS_INT do
        return n * fact (n-1).
    with n:*NEG_INT do
        throw Error("factorial is not defined for "+n).
    end

io @println ("The factorial of 3 is: " + fact (3)).
assert (fact(3) == 6).
```

Output:

```
The factorial of 3 is: 6
```

Challenge: Fibonacci numbers

> Print the Nth Fibonacci number.

Fibonacci numbers are defined by the recurring formula:

```
f_n = f_{n-1} + f_{n-2}
```

You can assign two values at a time (**Challenge: Swap two values**). You can use that technique for calculating the next Fibonacci number from the previous two. To bootstrap the algorithm, the two first values are needed. In one of the definitions of the Fibonacci row, the first two values are both 1.

Here we give an iterative solutions. It is clear that there exists a trivial recursive solution by implementing the above formula.

```
load system io.

let n = 10. -- compute the 10th Fib number

let (f_1,f_2) = (1,1).
for i in 3 to n do
    let (f_1,f_2) = (f_1+f_2,f_1).
end

io @println f_1.
assert (f_1 == 55)
```

Output:

```
55
```

Challenge: Print squares

> Print the squares of the numbers 1 through 10.

Of course this is straightforward, with a `for-loop` over a list. Here we show another solution using the list `map` function.

```
load system io.

let sq = [1 to 10] @map(lambda with x do return x*x).

io @println sq.

assert (sq == [1,4,9,16,25,36,49,64,81,100])
```

Output:

```
[1,4,9,16,25,36,49,64,81,100]
```

Challenge: Powers of two

> Print the first ten powers of two.

Just as in the previous challenge, we skip the naive loop solution and give a solution using the `map` function.

```
load system io.
load system math.

let p2 = [0 to 9] @map(lambda with x do return math @pow(2,x)).

io @println p2.

assert (p2 == [1,2,4,8,16,32,64,128,256,512])
```

Output:

```
[1,2,4,8,16,32,64,128,256,512]
```

Challenge: Odd and even numbers

> Print the first ten odd numbers. Print the first ten even numbers.

We start with printing the first ten odd numbers,

```
load system io.
load system math.

let odd = []
for (n if math @mod(n,2) =/= 0) in 1 to 10 do
    let odd = odd + [n].
end

io @println odd.
assert(odd == [1,3,5,7,9])
```

Output:

```
[1,3,5,7,9]
```

Now the even numbers,

```
load system io.
load system math.

let even = []
for (n if math @mod(n,2) == 0) in 1 to 10 do
    let even = even + [n].
end

io @println even.
assert(even == [2,4,6,8,10])
```

Output:

```
[2,4,6,8,10]
```

Challenge: Compare numbers approximately

> Compare the two non-integer values approximately.

Comparing non-integer numbers (which are represented as floating-point numbers) is often a task that requires approximate comparison. In Asteroid this can be accomplished with the `isclose` function availabel in the `math` module.

```
load system io.
load system math.

-- not equal under the default tolerance of 1E-09
assert (not math @isclose(2.0,2.00001)).
```

(continues on next page)

(continued from previous page)

```
-- equal under the user defined tolerance of 0.0001
assert (math @isclose(2.0,2.00001,0.0001)).
```

Challenge: Prime numbers

> Decide if the given number is a prime number.

Prime numbers are those that can be divided only by 1, and by themselves.

```
load system io.
load system math.

function isprime with x do
    if x >= 2 do
        for y in range(2,x) do
            if not math @mod(x,y) do
                return false.
            end
        end
    else do
        return false.
    end
    return true.
end

io @println (isprime 17).
io @println (isprime 15).

assert (isprime(17)).
assert (not isprime(15)).
```

Output:

```
true
false
```

Challenge: List of prime numbers

> Print the list of the first ten prime numbers.

```
load system io.
load system math.

function isprime with x do
    if x >= 2 do
        for y in range(2,x) do
            if not math @mod(x,y) do
                return false.
            end
        end
```

(continues on next page)

(continued from previous page)

```
    else do
        return false.
    end
    return true.
end

let cnt = 0.
for (n if isprime(n)) in 1 to 1000000 do
    io @println n.
    let cnt = cnt+1.
    if cnt == 10 do
        break.
    end
end
```

Output:

```
2
3
5
7
11
13
17
19
23
29
```

Challenge: Prime factors

> Find the prime factors of a given number.

Prime factors are the prime numbers that divide the given integer number exactly.

```
load system io.
load system math.

function isprime with x do
    if x >= 2 do
        for y in range(2,x) do
            if not math @mod(x,y) do
                return false.
            end
        end
    else do
        return false.
    end
    return true.
end

function primes with x do
    let lp = [].
```

(continues on next page)

(continued from previous page)

```
    for (n if isprime(n)) in 1 to x do
        let lp = lp+[n].
    end
    return lp.
end

let n = 165.
let factors = [].
let primes_list =  primes(n).
let ix = 0.

while n > 1 do
    let factor = primes_list @ix.
    let ix = ix+1.
    if not math @mod(n,factor) do
        let ix = 0.
        let n = n/factor.
        let factors = factors+[factor].
    end
end
io @println factors.

assert (factors == [3,5,11])
```

Output:

```
[3,5,11]
```

Challenge: Reducing a fraction

> Compose a fraction from the two given integers — numerator and denominator — and reduce it to lowest terms.

5/15 and 16/280 are examples of fractions that can be reduced. The final results of this task are 1/3 and 2/35. Generally, the algorithm of reducing a fraction requires searching for the greatest common divisor, and then dividing both numerator and denominator by that number. For our solution we use the function `gcd` available in the `math` module.

```
load system io.
load system math.

-- fraction a/b
let a = 16.
let b = 280.

-- reduce fraction
let gcd_val = math @gcd(a,b).
let numerator = a/gcd_val.
let denominator = b/gcd_val.
io @println numerator.
io @println denominator.

-- show that original and reduced fraction are the same value
assert (a/b == numerator/denominator).
```

Output:

```
2
35
```

Challenge: Divide by zero

> Do something with the division by zero.

Asteroid is an eager language, that is, expressions are evaluated as early as possible. We can trap division-by-zero errors using a try-catch block.

```
load system io.

try
    io @println (42/0).
catch Exception(_,m) do
    io @println m.
end
io @println "We are still alive...".
```

Output:

```
integer division or modulo by zero
We are still alive...
```

1.6.5 Section: Random Numbers

Challenge: Generating random numbers

> Generate a random number between 0 and N.

Asteroid has two random number generation functions: `random()` generates a random real value in the interval $[0.0,1.0)$ and `randint(a,b)` that generates a random value in the interval $[a,b]$. The type of the random value generated depends on the type of the values a and b specifying the interval.

```
load system io.
load system random.
load system util.
load system type.

let randint = random @randint.

random @seed(42).

io @println (random @random()).          -- random value in [0.0,1.0)
io @println (randint(0.0,1.0)).  -- random value in [0.0,1.0]
io @println (randint(0,1)).      -- always 0 or 1

-- generating a random number in the appropriate interval
let n = 10.
io @println (randint(0.0,type @toreal(n))).
io @println (randint(0,n)).
```

Output:

```
0.6394267984578837
0.025010755222666936
1
2.4489185380347624
2
```

Challenge: Neumann's random generator

> Implement Von Neumann's random number generator (also known as Middle-square method).

This algorithm is a simple method of generating short sequences of four-digit random integers. The method has its drawbacks, but for us, it is an interesting algorithmic task. The recipe has these steps:

1. Take a number between 0 and 9999.
2. Calculate the square of it.
3. If necessary, add leading zeros to make the number 8-digit.
4. Take the middle four digits.
5. Repeat from step 2.

To illustrate it with an example, let's take the number 1234 as the seed. On step 2, it becomes 1522756; after step 3, 01522756. Finally, step 4 extracts the number 5227.

```
load system io.
load system util.
load system type.

let n = 1234.
let sq = n*n.
let sq_str = type @tostring(sq).
if sq_str @length() < 8 do
    let prefix = [1 to 8-sq_str@length()] @map(lambda with _ do return "0")
                                          @join("").
    let sq_str = prefix + sq_str.
end
let rstr = sq_str @[2 to 5].
let rval = type @tointeger(rstr).
io @println rval.

assert (rval == 5227)
```

Output:

```
5227
```

Challenge: Histogram of random numbers

> Test the quality of the random generator by using a histogram to visualise the distribution.

The quality of the built-in generator of random numbers fully depends on the algorithm the developers of the compiler used. As a user, you cannot do much to change the existing generator, but you can always test if it delivers numbers uniformly distributed across the whole interval.

In our solution, we generate 10 random integers between 0 and 9. We then count how many times each of the integers have been generated. If it is a decent random number generator, all numbers should have been generated roughly an equal number of times.

```
load system io.
load system random.

let hist = [0 to 9] @map(lambda with _ do return 0).

for _ in range(10000) do
    let ix = random @randint(0,9).
    let hist @ix = hist @ix +1
end

io @println hist.
```

Output:

```
[944,1032,1015,968,981,986,1014,1058,989,1013]
```

1.6.6 Section: Mathematical Problems

Challenge: Distance between two points

> Calculate the distance between the two points on a surface.

There are two points on a surface, each with their own coordinates, x and y. The task is to find the distance between these two points. A straightforward solution would be to use the Pythagorean theorem:

```
load system io.
load system math.

let x = [10, 3].
let y = [9, 1].
let d = (math @sqrt(math @pow(x@0-y@0,2) + math @pow(x@1-y@1,2))).
io @println d.

assert (d == 2.23606797749979)
```

Output:

```
2.23606797749979
```

Another approach is using the math identity,

```
||a|| = sqrt(a . a)
```

where `.` represents the dot product. In our case `a` would be the distance vector between points `x` and `y`,

```
load system io.
load system math.
load system vector.

let x = [10, 3].
let y = [9, 1].
let a = vector @sub(x,y).
let d = math @sqrt(vector @dot(a,a)).
io @println d.

assert (d == 2.23606797749979)
```

Output:

```
2.23606797749979
```

The interesting part about the second approach is that it is completely dimension independent. Note that except for the definition of the vectors x and y dimension never plays a part in the definition of the program.

Challenge: Standard deviation

> For the given data, calculate the standard deviation value (sigma).

Standard deviation is a statistical term that shows how compact data distribution is. The formula is the following:

$$\sigma = \sqrt{\frac{1}{n-1}} \sum_i (\bar{x} - x_i)^2$$

where n is the number of elements in the array x; $\bar{x}$ is its average value (**Challenge: Average on an array**).

```
load system io.
load system math.

let values = [727.7, 1086.5, 1091.0, 1361.3, 1490.5, 1956.1].

let avg = values @reduce(lambda with (x,y) do return x+y) / values @length().
let diff_sq = values @map(lambda with x do return math @pow(x-avg,2)).
let numerator = diff_sq @reduce(lambda with (x,y) do return x+y).
let denominator = values @length() -1.
let sigma = math @sqrt(numerator/denominator).
io @println sigma.

assert (sigma == 420.96248961952256)
```

Output:

```
420.96248961952256
```

Challenge: Polar coordinates

> Convert the Cartesian coordinates to polar and backward.

Polar coordinates are a convenient way of representing points on a surface with the two values: distance from the centre of coordinates, and the angle between the vector and the pole axis. The conversion formulae between the Cartesian and polar systems, which is valid for **positive** `x` and `y`, are the following:

```
x = r cos(psi)
y = r sin(psi)
r = sqrt(x^2 + y^2)
psi = arctan(x/y)
```

These expressions can be implemented as-is in the code:

```
load system io.
load system math.

-- define common math functions locally so the
-- formulas are easy to read
let cos = math @cos.
let sin = math @sin.
let sqrt = math @sqrt.
let pow = math @pow.
let atan = math @atan.

function polar_to_cartesian with (r,psi) do
    -- return a tuple: (x,y)
    return (r*cos(psi),r*sin(psi)).
end

function cartesian_to_polar with (x,y) do
    -- return a tuple: (r,psi)
    return (sqrt(pow(x,2)+pow(y,2)),atan(y/x)).
end

let (r,psi) = cartesian_to_polar(1,2).
let (x,y) = polar_to_cartesian(r,psi).

io @println (x,y).

-- show that the recovered coordinates are the same
-- we started with
assert (math @isclose(1,x,0.0001) and math @isclose(2,y,0.0001)).
```

Output:

```
(1.0000000000000002,2.0)
```

For the **negative** `x` and `y`, the Cartesian-to-polar conversion is a bit more complicated. Depending on the quadrant of the point, the `psi` value is bigger or smaller than `pi`. When `x` is zero, it is either `-pi/2` or `pi/2`. All these variants can be implemented by using `with` clauses and conditional matching, as demonstrated below:

```
load system io.
load system math.
```

(continues on next page)

(continued from previous page)

```
load system util.
load system type.

-- define common math functions locally so the
-- formulas are easy to read
let cos = math @cos.
let sin = math @sin.
let sqrt = math @sqrt.
let pow = math @pow.
let atan = math @atan.
let pi = math @pi.
let toreal = type @toreal.

function polar_to_cartesian with (r,psi) do
    -- return a tuple: (x,y)
    return (r*cos(psi),r*sin(psi)).
end

function cartesian_to_polar with (x,y) do
    return (sqrt(pow(x,2)+pow(y,2)),cartesian_to_psi(x,y)).
end

function cartesian_to_psi
    with (x,y) if x > 0  do
        return atan(toreal(y)/x).
    with (x,y) if x < 0 and y >= 0 do
        return atan(toreal(y)/x)+pi.
    with (x,y) if x < 0 and y < 0 do
        return atan(toreal(y)/x)-pi.
    with (x,y) if x == 0 and y > 0 do
        return pi/2.
    with (x,y) if x == 0 and y < 0 do
        return -pi/2.
    with (x,y) if x == 0 and y == 0 do
        return none.
    end

let (r,psi) = cartesian_to_polar(-3,5).
let (x,y) = polar_to_cartesian(r,psi).

io @println (x,y).

-- show that the recovered coordinates are the same
-- we started with
assert (math @isclose(-3,x,0.0001) and math @isclose(5,y,0.0001)).
```

Output:

```
(-2.999999999999999,5.000000000000001)
```

Challenge: Monte Carlo method

> Calculate the area of a circle of radius 1 using the Monte Carlo method.

The Monte Carlo method is a statistical method of calculating data whose formula is not known. The idea is to generate a big number of random numbers and see how many of them satisfy the condition.

To calculate the area of a circle with a radius of 1, pairs of random numbers between 1 and 1 are generated. These pairs represent the points in the square in the center of coordinates with sides of length 2. The area of the square is thus 4. If the distance between the random point and the center of the square is less than 1, then this point is located inside the circle of that radius. Counting the number of points that landed inside the circle and the number of points outside the circle gives the approximate value of the area of the circle, as soon as the area of the square is known. Here is the program.

```
load system io.
load system math.
load system random.

let sqrt = math @sqrt.
let pow = math @pow.
let randint = random @randint.

random @seed(42).

let inside = 0.
let n = 10000.
for _ in 1 to n do
    let point = (randint(-1.0,1.0),randint(-1.0,1.0)).
    if sqrt(pow(point@0,2)+pow(point@1,2)) <= 1.0 do
        let inside = inside+1.
    end
end
let area = 4.0 * inside / n.
io @println area.

assert (area == 3.1392).
```

Output:

```
3.1392
```

Challenge: Guess the number

> Write a program that generates a random integer number between 0 and 10, asks the user to guess it, and says if the entered value is too small or too big.

First, a random number needs to be generated. Then the program must ask for the initial guess and enter the loop, which compares the guess with the generated number.

```
load system io.
load system random.
load system util.
load system type.

```

(continues on next page)

(continued from previous page)

```
random @seed(42).

let n = random @randint(0,10).
let guess = type @tointeger(io @input("Guess my number between 0 and 10: ")).
while guess =/= n do
    if guess < n do
        io @println "Too small.".
    elif guess > n  do
         io @println "Too big.".
    end
    let guess = type @tointeger(io @input("Try again: ")).
end
io @println "Yes, this is it!".
```

Challenge: Binary to integer

> Convert a binary number to a decimal integer.

In Asteroid this is straightforward using the built-in `tointeger` function, passing it a string representation of the binary number and the base.

```
load system io.
load system type.

let bin = "101101".
let int = type @tointeger(bin,2).
io @println int.

assert (int == 45).
```

Output:

```
45
```

Challenge: Integer as binary, octal, and hex

> Print a given integer number in the binary, octal, and hexadecimal representations.

In Asteroid this is easily done with the `tobase` function.

```
load system io.
load system type.

let tobase = type @tobase.
let tointeger = type @tointeger.

let val = 42.

io @println (tobase(val,2)).  -- bin
io @println (tobase(val,8)).  -- oct
io @println (tobase(val,16)). -- hex
```

(continues on next page)

(continued from previous page)

```
-- make sure that conversions are correct in both directions
assert (tointeger(tobase(val,2),2) == val).
assert (tointeger(tobase(val,8),8) == val).
assert (tointeger(tobase(val,16),16) == val).
```

Output:

```
101010
52
2A
```

Challenge: Sum of digits

> Calculate the sum of digits of a given number.

Pretty straightforward using string and list manipulation.

```
load system io.
load system type.

let number = 139487854.


let s = type @tostring number @explode()
                              @map(type @tointeger)
                              @reduce(lambda with (x,y) do return x+y).
io @println s.

assert (s == 49).
```

Output:

```
49
```

Challenge: Bit counter

> Count the number of bits set to 1 in a binary representation of a positive integer number.

If we remove all the zeros from a binary number, then we are left with only 1 characters which we can then count.

```
load system io.

let bits = "1010101" @replace("0","")
                     @length().
io @println bits.

assert (bits == 4).
```

Output:

```
4
```

Challenge: Compose the largest number

> Given the list of integers, compose the largest possible number by concatenating them.

The easiest way to achieve that is to treat the numbers as strings, sort them alphabetically in descending order, concatenate the pieces to a single string, and get the resulting integer.

```
load system io.
load system type.

let a = type @tointeger([67, 8, 1, 5, 45] @map(type @tostring) @sort(true) @join("")).
io @println a.

assert (a == 8675451).
```

Output:

```
8675451
```

Challenge: Convert to Roman numerals

> Convert an integer number to a Roman numerals string.

Roman numbers are not a direct translation of the decimal system. In this task, we assume that the number is not more than 3999, which is the maximum a regular Roman number can reach.

Let's use the algorithm that keeps the table of pre-calculated sequences of Roman letters. This is so that we don't have to check when III becomes IV, or when another I appears after V, etc.

In the program below, there are four such sequences: for thousands, hundreds, tens, and ones. The program iterates over the digits of the number in the decimal representation and chooses one of the values from the array of lists stored in the `roman_hash` table.

```
load system io.
load system math.
load system util.
load system hash.
load system type.

let roman_hash = hash @hash().
roman_hash @insert(1000,["","M","MM","MMM"]).
roman_hash @insert(100,["","C","CC","CCC","CD","D","DC","DCC","DCCC","CM"]).
roman_hash @insert(10,["","X","XX","XXX","XL","L","LX","LXX","LXXX","XC"]).
roman_hash @insert(1,["","I","II","III","IV","V","VI","VII","VIII","IX"]).

let n = 2018.
let p10 = range(type @tostring n @length()) @map(lambda with x do return math @pow(10,x))
                                            @reverse().
let digits = type @tostring n @explode()
                              @map(type @tointeger).
let z = util @zip(digits, p10).
```

(continues on next page)

(continued from previous page)

```
io @println z.
let roman = "".
for (d,p) in z do
    let roman = roman + roman_hash @get(p) @d.
end
io @println roman.

assert (roman == "MMXVIII")
```

Output:

```
[(2,1000),(0,100),(1,10),(8,1)]
MMXVIII
```

Challenge: Spelling numbers

> Write an integer number below one million in words.

Human languages have many inconsistencies, especially in the most frequent constructs. Spelling numbers seems to be a simple task, but due to a number of small differences, the resulting program is quite big.

The program is listed on the next page. Let's discuss the algorithm first.

Take a number; for example, 987,654. The rules for spelling out the groups of three digits, 987 and 654, are the same. For the first group, the word thousand must be added.

Now, examine a group of three digits. The first digit is the number of hundreds, and it has to be spelled only if it is not zero. If it is not zero, then we spell the digit and add the word hundred.

Now, remove the leftmost digit, and we've got two digits left. If the remaining two digits form the number from 1 to 20, then it can be directly converted to the corresponding name. The names for the numbers from 0 to 10 are obviously different. The names for the numbers from 11 to 19 have some commonalities, but is it still easier to directly prepare the names for all of them.

For the larger numbers (21 to 99), there are two cases. If the number is dividable by 10 then a name for 20, 30, 40, etc. is taken. If not, then the name is built of the name of tens and the name for units, joined with a hyphen, such as forty-five.

The zero name appears only in the case when the given number is zero.

```
load system io.
load system math.

let mod = math @mod.

let names = ["zero","one","two","three","four","five","six","seven","eight","nine",
             "ten","eleven","twelve","thirteen","fourteen","fifteen",
             "sixteen","seventeen","eighteen","nineteen","twenty","thirty",
             "forty","fifty","sixty","seventy","eighty","ninety"].

function spell_number
    with (n:%integer) if n < 20 do
        return names @n.
    with (n:%integer) if n < 100 do
        let r = names @(n / 10 + 18).
```

(continues on next page)

(continued from previous page)

```
        let r = r + ("-" + names @(mod(n,10))) if mod(n,10) else "".
        return r.
    with (n:%integer) if n < 1000 do
        return spell_part(n,100,"hundred").
    with (n:%integer) if n < 1000000 do
        return spell_part(n,1000,"thousand").
    end

function spell_part
    with (n:%integer,base:%integer,name:%string) do
        let r = spell_number(n/base) + " " + name.
        return r + " " + spell_number(mod(n,base)) if mod(n,base) else r.
    end

io @println (spell_number 15).
io @println (spell_number 75).
io @println (spell_number 987654).
io @println (spell_number 1001).
```

Output:

```
fifteen
seventy-five
nine hundred eighty-seven thousand six hundred fifty-four
one thousand one
```

1.6.7 Section: Manipulating Lists and Arrays

Challenge: Swap two values

> Swap the values of two variables.

In Asteroid, there is no need to use temporary variables to swap the values of two variables. Just use tuples on both sides of the equation:

```
let (b,a) = (a,b).
```

Consider the complete program:

```
load system io.

let (a,b) = (10,20).
let (b,a) = (a,b).
io @println ("a = "+a,"b = "+b).

assert ((a,b) is (20,10)).
```

Output:

```
(a = 20,b = 10)
```

This program prints the swapped values:

```
(a = 20,b = 10)
```

This approach also works with elements of an array:

```
load system io.

let a = [3,5,7,4].
let (a@2,a@3) = (a@3,a@2).
io @println a.

assert (a is [3,5,4,7]).
```

Output:

```
[3,5,4,7]
```

Challenge: Reverse a list

> Print the given list in reverse order.

```
load system io.

let a = [10, 20, 30, 40, 50].
io @println (a @reverse()).

assert(a == [50,40,30,20,10]).
```

Output:

```
[50,40,30,20,10]
```

Challenge: Rotate a list

> Move all elements of an array N positions to the left or to the right.

Asteroid does not have a built-in `rotate` function. However, such a function is easily constructed through slicing lists (see `vix` below).

```
load system io.
load system math.

function rotate with (l:%list,i:%integer) do
    let n = l @length().
    let vix = range n @map(lambda with x do return math @mod(x+i,n)).
    return l @vix.
end


let a = [1, 3, 5, 7, 9, 11, 13, 15].
let b = rotate(a,3).
let c = rotate(a,-3).
io @println a.
```

(continues on next page)

(continued from previous page)

```
io @println b.
io @println c.

assert(b == [7,9,11,13,15,1,3,5] and c == [11,13,15,1,3,5,7,9]).
```

Output:

```
[1,3,5,7,9,11,13,15]
[7,9,11,13,15,1,3,5]
[11,13,15,1,3,5,7,9]
```

Challenge: Randomize an array

> Shuffle the elements of an array in random order.

This is easily accomplished with the built-in `shuffle`.

```
load system io.
load system random.

random @seed(42).
let b = [1 to 20] @shuffle().
io @println b.

assert(b == [20,6,15,5,10,14,16,19,7,13,18,11,2,12,3,17,8,9,1,4]).
```

Output:

```
[20,6,15,5,10,14,16,19,7,13,18,11,2,12,3,17,8,9,1,4]
```

Challenge: Incrementing array elements

> Increment each element in an array.

For this we use Asteroid's `vector` module, which can handle incrementing a vector with a scalar.

```
load system io.
load system vector.

let a = [1 to 10].
let b = vector @add(a,1).
io @println b.

assert(b == [2,3,4,5,6,7,8,9,10,11]).
```

Output:

```
[2,3,4,5,6,7,8,9,10,11]
```

Challenge: Adding up two arrays

> Take two arrays and create a new one whose elements are the sums of the corresponding items of the initial arrays.

Again, here we take advantage of Asteroid's `vector` module. Note that the two vectors have to be of the same length in order to add them together.

```
load system io.
load system vector.

let a = [10 to 20].
let b = [30 to 40].
let c = vector @add(a,b).
io @println c.

assert(c == [40,42,44,46,48,50,52,54,56,58,60]).
```

Output:

```
[40,42,44,46,48,50,52,54,56,58,60]
```

The vector module defines a function called `op` that allows you to combine two vectors using any arbitrary binary function. Rewriting the above program using `op`,

```
load system io.
load system vector.

let a = [10 to 20].
let b = [30 to 40].
let c = vector @op((lambda with (x,y) do return x+y),a,b).
io @println c.

assert(c == [40,42,44,46,48,50,52,54,56,58,60]).
```

Output:

```
[40,42,44,46,48,50,52,54,56,58,60]
```

As we said above, any arbitrary binary function. Consider the relational operator < expressed as a lambda function,

```
load system io.
load system vector.
load system random.

random @seed(42).

let a = [1 to 10] @shuffle().
let b = [1 to 10] @shuffle().
let c = vector @op((lambda with (x,y) do return x<y),a,b).
io @println c.

assert(c == [false,true,false,false,false,true,false,false,true,true]).
```

Output:

```
[false,true,false,false,false,true,false,false,true,true]
```

Challenge: Exclusion of two arrays

> From the given two arrays, find the elements of the first array which do not appear in the second one.

Here we use Asteroid's `set` module.

```
load system io.
load system set.

let a = [1 to 10].
let b = [5 to 15].
let c = set @diff(a,b).
io @println c.

assert(c @sort() == [1,2,3,4]).
```

Output:

```
[2,3,1,4]
```

1.6.8 Section: Information Retrieval

Challenge: Sum of the elements of an array

> Find the sum of the elements of an array of integers.

```
load system io.

let a = [4, 6, 8, 1, 0, 58, 1, 34, 7, 4, 2].
let s = a @reduce(lambda with (x,y) do return x+y).
io @println s.

assert (s == 125).
```

Output:

```
125
```

If summing up elements that are greater than 10,

```
load system io.

let a = [4, 6, 8, 1, 0, 58, 1, 34, 7, 4, 2].
let f = (lambda with (x,y) do return x+(y if y > 10 else 0)).
let s = a @reduce(f,0).
io @println s.

assert (s == 92).
```

Output:

```
92
```

Challenge: Average of an array

> Find the average value of the given array of numbers.

```
load system io.

let a = [7, 11, 34, 50, 200].
let avg = a @reduce(lambda with (x,y) do return x+y)/a @length().
io @println avg.

assert (avg == 60).
```

Output:

```
60
```

Challenge: Is an element in a list?

> Tell if the given value is in the list.

```
load system io.

let array = [10, 14, 0, 15, 17, 20, 30, 35].
let x = 17.
io @println ((x+" is in the list") if array @member(x) else (x+" is not in the list")).
```

Output:

```
17 is in the list
```

We can also use a reduction function to solve this,

```
load system io.

let array = [10, 14, 0, 15, 17, 20, 30, 35].
let x = 17.

if array @reduce(lambda with (acc,i) do return true if i==x else acc,false) do
    io @println (x+" is in the list").
else
    io @println (x+" is not in the list").
end
```

Output:

```
17 is in the list
```

Challenge: First odd number

> Find the first odd number in a list of integers.

The easiest way to do this is with a reduction,

```
load system io.
load system math.
load system util.
load system type.

let mod = math @mod.

let array = [2, 4, 18, 9, 16, 7, 10].
let odd = array @reduce(lambda with (acc,i) do return i if type @isnone(acc) and mod(i,
↪2) else acc,none).
io @println odd.
```

Output:

```
9
```

Challenge: Take every second element

> Form a new array by picking every second element from the original array.

```
load system io.
load system math.

let array = [20 to 30] @filter(lambda with x do return math @mod(x,2)).
io @println array.

assert (array == [21,23,25,27,29]).
```

Output:

```
[21,23,25,27,29]
```

We can use an index vector to accomplish the same thing,

```
load system io.
load system math.

let a = [20 to 30].
let array = a @[1 to a @length()-1 step 2] .
io @println array.

assert (array == [21,23,25,27,29]).
```

Output:

```
[21,23,25,27,29]
```

Challenge: Number of occurrences in an array

> Count how many times a particular element appears in the array.

```
load system io.
load system math.

let dt = ["apple",
          "pear",
          "grape",
          "lemon",
          "peach",
          "apple",
          "banana",
          "grape",
          "pineapple",
          "avocado"].

let cnt = dt @count("grape").
io @println cnt.

assert (cnt == 2).
```

Output:

```
2
```

Challenge: Finding unique elements

> Print all unique elements of the given array.

Converting a list to a set will remove all duplicate elements in the list.

```
load system io.
load system set.

function unique with lst:%list do
    return set @toset lst @sort().
end

let a = unique([2, 3, 7, 4, 5, 5, 6, 2, 10, 7]).

io @println a.

assert (a == [2,3,4,5,6,7,10])
```

Output:

```
[2,3,4,5,6,7,10]
```

Challenge: Minimum and maximum

> Find the minimum and the maximum numbers in the given list of integers.

```
load system io.

function max with lst:%list do
    return lst @sort(true) @0.
end

function min with lst:%list do
    return lst @sort() @0.
end

let v = [7, 6, 12, 3, 4, 10, 2, 5, 15, 6, 7, 8, 9, 3].

let a = max v.
let b = min v.

io @println a.
io @println b.

assert (a == 15 and b == 2).
```

Output:

```
15
2
```

Challenge: Increasing sequences

> Check if the given array contains increasing (or decreasing) numbers.

```
load system io.
load system type.

let a = [3, 7, 19, 20, 34].
let b = type @toboolean(a @reduce(lambda with (x,y) do return y if x<y else false)).

io @println b.

assert (b).
```

Output:

```
true
```

1.6.9 Section: Multi-Dimensional Data

Challenge: Transpose a matrix

> Take a matrix and print its transposed version.

In Asteroid a matrix can be represented by nested lists, like so,

```
let m = [[1,2],
         [3,4]].
```

The transpose of this matrix is,

```
let m = [[1,3],
         [2,4]].
```

In a square matrix computing the transpose is just a matter of swapping around the elements. However, here we will solve the more general problem for non-square matrices,

```
let m = [[1,2],
         [3,4],
         [5,6]].
```

with its transpose,

```
let m = [[1,3,5],
         [2,4,6]].
```

The procedure:

```
load system io.

function transpose with m do
    -- figure out the dimensions
    let xdim = m @0 @length().
    let ydim = m @length().

    -- reserve space for the transpose
    -- first we do the ydim of new matrix
    let mt = range(xdim).
    for y in mt do
        let mt @y = range(ydim).
    end

    -- swap the elements
    for x in range(xdim) do
        for y in range(ydim) do
            let mt @x @y = m @y @x.
        end
    end
```

(continues on next page)

(continued from previous page)

```
    return mt.
end

function print_matrix with m do
    io @println "".
    for r in m do
        for e in r do
            io @print (e + " ").
        end
        io @println ("").
    end
    io @println "".
end

let m = [[1,2],
         [3,4]].

let mt = transpose(m).

io @println ("The transpose of:").
print_matrix m.
io @println ("is:").
print_matrix mt.
io @println ("").

let m = [[1,2],
         [3,4],
         [5,6]].

let mt = transpose(m).

io @println ("The transpose of:").
print_matrix m.
io @println ("is:").
print_matrix mt.
io @println ("").

assert(mt == [[1,3,5],[2,4,6]]).
```

Output:

```
The transpose of:

1 2
3 4

is:

1 3
2 4
```

(continues on next page)

(continued from previous page)

```
The transpose of:

1 2
3 4
5 6

is:

1 3 5
2 4 6
```

Challenge: Sort hashes by parameter

> Sort a list of hashes using data in their values.

This task is commonly performed to sort items where the sortable parameter is one of the values in the hash. For example, sorting a list of people by age.

```
load system io.
load system hash.
load system sort.
load system random.

let randint = random @randint.

random @seed(42).

-- hash of names with ages
let ht = hash @hash().
ht @insert("Billie",randint(20,50)).
ht @insert("Joe",randint(20,50)).
ht @insert("Pete",randint(20,50)).
ht @insert("Brandi",randint(20,50)).

-- export the hash as a list of pairs
let lst = ht @aslist().

-- define our order predicate on a
-- list of pairs where the second
-- component holds the order info
function pairs with ((_,x),(_,y)) do
    return true if x < y else false.
end

-- print out the sorted list
io @println (sort @sort(pairs,lst)).

assert (sort @sort(pairs,lst) == [("Pete",20),("Joe",23),("Billie",40),("Brandi",43)])
```

Output:

```
[(Pete,20),(Joe,23),(Billie,40),(Brandi,43)]
```

Challenge: Count hash values

> For a given hash, count the number of occurrences of each of its values.

For example, a hash is a collection mapping a car's license plate to the colour of the car or a passport number to the name of the street where the person lives. In the first example, the task is to count how many cars of each colour there are. In the second example, we have to say how many people live on each street. But let's simply count the colours of fruit.

```
load system io.
load system hash.
load system sort.

let fruit_hash = hash @hash().
fruit_hash @insert("apple","red").
fruit_hash @insert("avocado","green").
fruit_hash @insert("banana","yellow").
fruit_hash @insert("grapefruit","orange").
fruit_hash @insert("grapes","green").
fruit_hash @insert("kiwi","green").
fruit_hash @insert("lemon","yellow").
fruit_hash @insert("orange","orange").
fruit_hash @insert("pear","green").
fruit_hash @insert("plum","purple").

let fruit_lst = fruit_hash @aslist().

let color_hash = hash @hash().
for (_,color) in fruit_lst do
    if not color_hash @get(color) do
        color_hash @insert(color,1).
    else
        color_hash @insert(color, color_hash @get(color) +1).
    end
end
let color_lst = color_hash @aslist().

function pairs with ((_,x),(_,y)) do
    return true if x < y else false.
end

io @println (sort @sort(pairs,color_lst)).
```

Output:

```
[(red,1),(purple,1),(yellow,2),(orange,2),(green,4)]
```

Challenge: Product table

> Generate and print the product table for the values from 1 to 10.

We will do this with an outer loop and a `map` function.

```
load system io.
load system type.

function format with v do
    let maxlen = 3.
    let vstr = type @tostring v.
    return [1 to maxlen-len(vstr)] @map(lambda with _ do return " ") @join("") + vstr.
end

for i in 1 to 10 do
    io @println ([1 to 10] @map(lambda with x do return format(i*x)) @join(" ")).
end
```

Output:

```
 1   2   3   4   5   6   7   8   9  10
 2   4   6   8  10  12  14  16  18  20
 3   6   9  12  15  18  21  24  27  30
 4   8  12  16  20  24  28  32  36  40
 5  10  15  20  25  30  35  40  45  50
 6  12  18  24  30  36  42  48  54  60
 7  14  21  28  35  42  49  56  63  70
 8  16  24  32  40  48  56  64  72  80
 9  18  27  36  45  54  63  72  81  90
10  20  30  40  50  60  70  80  90 100
```

Challenge: Pascal triangle

> Generate the numbers of the Pascal triangle and print them.

The Pascal triangle is a sequence of rows of integers. It starts with a single 1 on the top row, and each following row has one number more, starting and ending with 1, while all of the other items are the sums of the two elements above it in the previous row. It is quite obvious from the illustration:

```
      1
     1 1
    1 2 1
   1 3 3 1
  1 4 6 4 1
 1 5 10 10 5 1
1 6 15 20 15 6 1
```

To calculate the values of the next row, you may want to iterate over the values of the current row and make the sums with the numbers next to it. Let us use the functional style that the language offers. Consider the fourth row, for example: 1 3 3 1. To make the fifth row, you can shift all the values by one position to the right and add them up to the current row:

```
  13310
+ 01331
-------
14641
```

We can easily accomplish this with our `vector` module. Given the vector of the fourth row,

```
[1,3,3,1]
```

we create two new vectors,

```
[1,3,3,1,0]
```

and

```
[0,1,3,3,1]
```

We then add them together,

```
vector @add([1,3,3,1,0],[0,1,3,3,1]) = [1,4,6,4,1]
```

The only thing that is left to do is to iterate appropiately and format the output.

```
load system io.
load system vector.
load system util.
load system type.

let triangle = [[1]].
let ix = 0.

for i in 1 to 6 do
    let v = triangle @ix.
    let v1 = [0] + v.
    let v2 = v + [0].
    let new_v = vector @add(v1,v2).
    let triangle = triangle + [new_v].
    let ix = ix + 1.
end

for r in triangle do
    io @println (r @map(lambda with v do return type @tostring v) @join(" ")).
end
```

Output:

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1
```

The program prints the first seven rows of the Pascal triangle. The rows are not centred, and are aligned to the left side. As an extra exercise, modify the program so that it prints the triangle as it is shown at the beginning of this task. For example, you can first generate rows and keep them in a separate array and then, knowing the length of the longest string, add some spaces in front of the rows before printing them.

1.7 ADB Reference Guide

The Asteroid Debugger (ADB) is a source code debugger for the Asteroid programming language.

It supports the following:

- Breakpoints
- Single stepping at the source level
- Stepping through function calls
- Stack frame inspection
- Evaluation of arbitrary Asteroid code
- Macros
- REPL Instances

ADB features a unique "explicit" mode which details much of the pattern matching and underlying mechanics of Asteroid allow developers to debug pattern matches. Explicit mode details many of the steps of pattern matching, function calling, statement execution, and return values. Explicit mode is very experimental and is under current research. As a result, explicit mode is currently incomplete.

1.7.1 Usage

```
asteroid --adb <FILENAME>
```

```
asteroid -g <FILENAME>
```

1.7.2 Debugging sessions

The debugger's prompt `(ADB)` allows the user to enter commands to effect the source environment and debugger behavior.

Each debugging session will have a few pieces of information:

Filename and line number

```
[/home/user/test.ast (1)]
```

The current line which will be executed

```
-->> let p = pattern %[(x:%integer) if x > 0 and x < 100]%.
```

The command prompt

```
(ADB)
```

ADB runs like any other debugger, here's a small example session where we see running commands, listing the program, and setting and continuing to breakpoints.

```
[/home/user/test.ast (1)]
-->> let p = pattern %[(x:%integer) if x > 0 and x < 100]%.
(ADB) ll
----- Program Listing -----
>  1 let p = pattern %[(x:%integer) if x > 0 and x < 100]%.
   2
   3 let d = pattern %[(x:*p, y:*p, z:*p)]%.
   4 let x:*p = 99.
   5
   6 let t:*d = (1,2,3).
   7 [EOF]
(ADB) break 6
(ADB) continue
----- Breakpoint -----
[/home/user/test.ast (6)]
-->> let t:\*d = (1,2,3).
(ADB)
```

1.7.3 Commands

Below is a list of the commands available to the debugger. Most of which can be shortened. These shortenings are shown in parenthesis (*c(ommand)*).

List

`l(ist)` lists the lines around the currently executing line

Longlist

`ll (longlist)` lists the entire program

Step

`s(tep)` through to the next statement or through function call.

Continue

`c(ontinue) cont(inue)` continue execution until the next breakpoint.

Next

`n(ext)` continue onto the next executing line at the current scope.

Until

`u(ntil) ?lineno` By default, continue execution until a line with a greater number than the current one is reached. Given an optional line number, continue execution until a line number greater than or equal to that number is reached

Return

`(r(et))urn` Continue execution until the return of the current function is reached

Breakpoints

`b(reak) number\*` set a breakpoint at one or more lines. Running without any arguments lists your breakpoints.

Example: `b 1 2 3`, `break`.

Conditional breakpoints can be set in the same way, just attach `if eval("condition")` after each breakpoint number.

Example:

```
-- Set a conditional breakpoint on 10 and normal breakpoints on 11, 15, and 23.
b 10 if eval("x == 123") 11 15 23
```

Delete

`d(elete) (number)+` `del(ete) (number)+` delete a list of breakpoints.

Example:

```
del 1 5 8 9
```

Macro

`macro (name) (command list)`. Define a macro. Running just `macro` lists your macros.

Example macro that continues to a breakpoint and prints the value of `x`:

```
macro gox = c; eval("io@println(x)");
```

Eval

eval("asteroid code") Evaluate the asteroid code between quotes. Works exactly like a single-line repl.

Example, print out the value of `x`:

```
eval("x")
```

!

`!` Open up a repl in the current context

__retval__

`(_)_retval` Print the most recent return value

Help

`h(elp) (command)?` gives help for a given command. Running just `help` shows all available commands. Example:

```
h macro
help break
```

Retval

`((r)et)val` Prints the most recent return value

<

< move up one stack frame

>

> move down one stack frame

Where

`w(here)` displays the frame stack and the currently active frame.

Explicit

`e(xplicit) (on|off)?` By default, this command run without an argument toggles explicit mode. If given a literal `on` or `off`, explicit mode will be switched to the corresponding state.

Explicit

```
-- Toggle Explicit mode
explicit
e

-- Turn on/off
explicit on
e on
explicit off
e off
```

Quit

`q(uit)` Quits the current ADB session