Asteroid
The Programming Language

Dr Lutz Hamel
Dept. of Computer Science & Statistics
University of Rhode Island

asteroid-lang.org
Asteroid: The Programming Language

- The Asteroid programming language is,
  - modern
  - application-oriented
  - open-source
  - dynamically typed
  - multi-paradigm
  - heavily influenced by Python, Rust, ML, and Prolog
  - currently under development at the University of Rhode Island

- Project page: https://asteroid-lang.org

- A cloud-based version is available for this talk: https://replit.com/@lutzhamel/asteroid-talk-f22

- Documentation: https://asteroid-lang.readthedocs.io
Design Objectives

- Seamless integration of imperative, functional, and object-oriented programming.
- Full support of first-class patterns.
- Expressive, conversational syntax geared towards use in a classroom setting.
“Hello, World!”

- As is tradition when looking at a new programming language...hello world...

```plaintext
1  -- the customary hello world program
2
3  load system io.
4
5  io @println "Hello World!".
```

```
> asteroid hello.ast
Hello World!
```

hello.ast
Imperative Programming

- Should look familiar.
- Here is an imperative version of computing a factorial...

```plaintext
-- compute the factorial iteratively
-- this program makes use of the fact that multiplication
-- is commutative, i.e. 1*3*2 = 3*2*1

-- load modules
load system io.
load system type.

-- our factorial function
function fact with n do
  let val = 1.
  while n > 1 do
    let val = val*n.
    let n = n-1.
  end
  return val.
end

-- talk to the user
let x = type @tointeger (io @input "Enter a positive integer: ").
io @println ("The factorial of "+x+" is "+(fact x)).
```
Something a bit more interesting – the bubble sort.

Note the access operator ‘@’ for list element access.

‘@’ is a universal access operator:
- Member functions
- Tuple components
- List elements

---

```ast
1  -- the bubble sort
2  load system io.
3
4  -- sort list l in ascending order
5  function bubblesort with l do
6    loop -- forever
7      let swapped = false.
8      for i in 0 to len(l)-2 do
9        if l[@(i+1)] < l[@i] do -- out of order
10           let (l[@i],l[@(i+1)]) = (l[@(i+1)],l[@i]).
11             let swapped = true.
12       end
13     end
14     if not swapped do
15       break. -- done!
16     end
17  end -- loop
18  return l.
19 end
20
21  -- sort a list
22  let k = [6,5,3,1,8,7,2,4].
23  io @println ("unsorted list: "+k).
24  io @println ("sorted list: "+(bubblesort k)).
```

bubble.ast
Strongly Typed

- Asteroid supports several type hierarchies,
  
  boolean < integer < real < string
  list < string
  tuple < string
  none (or ‘()’)

- These are all built-in types.
- User defined types are introduced with the ‘structure’ keyword (more on that later).
  - User defined types do not belong to any hierarchy
- No generics,
  - Dynamic typing together with duck typing cover most of the use cases of generics in Asteroid.
Functional Programming

- Asteroid has a complete functional sublanguage.
  - ‘asteroid –F’ turns the Asteroid interpreter into a functional language interpreter.
  - Lisp/Scheme style functional programming – no monads or algebraic data types here.
  - But Asteroid offers pattern-matchable objects similar to Rust.
The functional version of the factorial computation...

```
  -- compute the factorial recursively

  load system io.
  load system type.

  function fact
    with 1 do
      1
    with n do
      n*fact(n-1).
  end

  let x = type @tointeger (io @input "Enter a positive integer: ").
  io @println ("The factorial of "+x+" is "+(fact x)).
```
Something a bit more interesting – the quick sort

Note:
- \([1|2,3] = [1,2,3]\)

```plaintext
qsort
with [] do -- empty list
    []
with [a] do -- single element list
    [a]
with [pivot|rest] do -- head-tail operator
    function filter -- local function
        with (e, [], fcmp) do
            []
        with (e, [a|rest], fcmp) do
            [a]+filter(e, rest, fcmp)
            if fcmp(a, e)
            else filter(e, rest, fcmp)
        end
    let less=filter(pivot, rest, lambda with (x,y) do x < y).
    let more=filter(pivot, rest, lambda with (x,y) do x >= y).
    qsort less + [pivot] + qsort more.
end
io @println (qsort [3,2,1,0]).
```
Asteroid allows you to “mix ‘n match” paradigms.

E.g. in the QuickSort we keep the functional multi-dispatch with structural pattern matching but replace the ‘filter’ functions with a ‘for’ loop from the imperative paradigm.

Our experience is that the various paradigms complement each other in a very natural way.

```
  -- Quicksort
  load system io.
  function qsort
      with [] do -- empty list
          [].
      with [a] do -- single element list
          [a].
      with [pivot|rest] do -- head-tail operator
          let less=[].
          let more=[].
          for e in rest do -- iteration instead of recursion
              if e < pivot do
                  let less = less + [e].
              else do
                  let more = more + [e].
              end
          end
          qsort less + [pivot] + qsort more.
      end
  end
  io @println (qsort [3,2,1,0]).
```
Function Calls

- In the functional programming tradition, Asteroid’s function calls are constructed by juxtaposing a function with a value, e.g. `fact 3`.

- The implication is that all functions have only a single argument. If you want to pass more than one value to a function you have to construct a tuple of values, e.g. `foo (1,2)`.

- Syntactically this looks the same as a function call to `foo` in Python but semantically it is very different – call `foo` with the value `(1,2)` in Asteroid as apposed to call `foo` with the list of values `(1,2)` in Python.

- This slight change of perspective enables effective pattern matching in the multi-dispatch within function definitions in Asteroid.
Function Calls

- The interpretation of function arguments as a list of values has unexpected implications in Python
  - $\text{foo (1,2)} \neq \text{foo ((1,2))}$, but
  - $(1,2) = ((1,2))$

- Inconsistent handling of parenthesized tuples!

In Asteroid it works as expected…
Asteroid implements a very clean and intuitive framework for higher-order programming, e.g. the ‘map’ function:

- A program that creates a list of alternating positive and negative ones.
- The list constructor [1 to 10] constructs a list of values [1, 2,...,10].
- The first map turns this list into the list [1,0,1,...0].
- The second map turns that list into the list [1,-1,1,-1,...,-1].

```map.ast
1 -- higher-order programming with the 'map' function
2 -- this program that creates a list of alternating 1 and -1
3
4 load system io.
5 load system math.
6
7 let a = [1 to 10] @map (lambda with x do math @mod (x,2))
     @map (lambda with x do 1 if x else -1).
8
9 io @println a.
```

```
Higher-Order Programming

- The improvements in the conceptual framework for higher-order programming in Asteroid are non-trivial.
- Let's try the same program in Python...

```python
l = [x for x in range(11)]
iter = map(lambda x: x%2, l)
out = list(map(lambda x: 1 if x else -1, iter))
print(out)
```

Compared to Asteroid:
```
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.
```
Structures

- Like in many modern programming languages such as Rust and Go, classes have given way to structures with member functions in Asteroid,
  - No member protection
  - No inheritance
  - But object identity (‘this’)

```plaintext
rect.ast

--- rectangle structure
load system io.

structure Rectangle with
  data xdim.
  data ydim.
  function area with () do -- member function
    return this@xdim * this@ydim.
  end
end

let r = Rectangle(4,2). -- default constructor
io @println ("The area of Rectangle ("+r@xdim+","+r@ydim+") is "+r@area()).
```
Asteroid

```rust
struct Rectangle with
data xdim.
data ydim.
function area with () do -- member function
  return this@xdim * this@ydim.
end
end
```

Go

```go
type rect struct {
  width int
  height int
}

func (r *rect) area() int {
  return r.width * r.height
}
```
First-Class Patterns

- The support of first-class patterns implies that patterns can be
  - stored in variables
  - passed to/from functions
- Asteroid implements the idiom
  - Patterns as values and values as patterns
First-Class Patterns

- In classical pattern matching patterns are syntactically static – consider the quick sort

```scala
-- functional implementation of quicksort
load system io.

function qsort
  with [] do -- empty list
    []
  with [a] do -- single element list
    [a]
  with [pivot|rest] do -- head-tail operator
    function filter -- local function
      with (e,[],fcmp) do
        []
      with (e,[a|rest],fcmp) do
        [a]+filter(e,rest,fcmp)
      if fcmp(a,e)
        else filter(e,rest,fcmp)
    end
    let less=filter(pivot,rest,l ambiguous with (x,y) do x < y).
    let more=filter(pivot,rest,l ambiguous with (x,y) do x >= y).
    qsort less + [pivot] + qsort more.
  end
io println (qsort [3,2,1,0]).
```
Pattern Reuse

- First-class patterns are values and therefore dynamic in their nature.
- First-class patterns enable pattern reuse.

\[ n \cdot \text{pos_int} \equiv n \text{ if } n \text{ is } \ast \text{pos_int} \]
Pattern Factoring

- Patterns can become quite complex, first-class patterns allow us to break patterns into smaller chunks.
- In the process we can also give sub-patterns meaningful names making pattern expressions much more comprehensible.

---

```plaintext
-- first-class patterns: factoring

load system io.

-- without first-class patterns
function foo1 with (x if (x is %integer) or (x is %real), y) do
    io @println (x,y).
end

-- with first-class patterns
let scalar = pattern v if (v is %integer) or (v is %real).

function foo2 with (x:*scalar, y) do
    io @println (x,y).
end

foo1 (1,2).
foo2 (1,2).
```
First-class patterns can act like types

Here we use the first-class pattern ‘Shape’ to define a subtype polymorphic function
Structures and Objects

- In Asteroid the ‘let’ statement is a pattern-match statement of the form:
  \[
  \text{let } \langle\text{pattern}\rangle = \langle\text{value}\rangle
  \]

- We pattern-match objects for data decomposition!

```plaintext
1 -- pattern matching on objects
2
3 structure A with
4   data a.
5   data b.
6 end
7
8 let o = A(1,2). -- call constructor
9 let A(x,y) = o. -- pattern match object o
10 assert(x==1 and y==2).
```

struct-pat.ast
Structures and Objects

Here are some fun pattern-match identities on objects using first-class and static patterns.

```
1  -- some pattern-match identities on objects
2
3  structure A with
4     data a.
5     data b.
6  end
7
8  let A(1,2) = A(1,2).  -- pattern A(1,2) matching new object A(1,2)
9  let o = A(1,2).      -- variable o as pattern for new object A(1,2)
10 let *o = A(1,2).     -- object o as pattern matching new object A(1,2)
11 let *o = o.         -- object o as pattern matching object o
```

ident.ast
Asteroid in the Classroom

- In CSC301 (Foundations of PLs) I use Asteroid mostly to teach functional programming concepts,
  - “Everything is a Value”
  - Higher-order programming
  - Pattern matching
- In CSC493 (Multi-Paradigm Programming) we look deeply into the different programming paradigms and study how they interact
  - In particular, we look at the effect first-class patterns have on programming in general
- The fact that Asteroid is dynamically typed and basically looks familiar to most students let’s us get to the interesting bits very quickly…
  - …in contrast to using something like Haskell or ML where we would have to wrangle the type system in non-trivial ways before we get to the interesting bits.
  - …or Lisp/Scheme where we would have to wrestle the uncommon syntax before we get to the interesting bits.
Future Work

- **Near term,**
  - We are developing a compiler for Asteroid that produces native code.
  - Key to this development is the Asteroid Virtual Machine (AVM) framework.

- **Long term,**
  - Asteroid has a niche as a development platform for performant programs within the WebAssembly (https://webassembly.org) framework geared toward frontend developers that are not used to working in C or Rust.
Thank You!

- I wanted to take this opportunity to thank the folks who have contributed to this project over the years, in particular,
  - Ariel Finkle
  - Calvin Higgins
  - Christian Tropeano
  - Oliver McLaughlin
  - Theodore Henson
  - Timothy Colaneri

- If you are interested in programming language design and implementation, we are always looking for contributors!
Questions?

- [lutzhamel@uri.edu](mailto:lutzhamel@uri.edu)
- or stop by my office for a chat.

- Homepage
  - [https://asteroid-lang.org](https://asteroid-lang.org)

- Example code at
  - [https://replit.com/@lutzhamel/asteroid-talk-f22](https://replit.com/@lutzhamel/asteroid-talk-f22)