



The Let Statement & Basic Pattern Matching

- The let statement is a pattern-match statement in Asteroid,

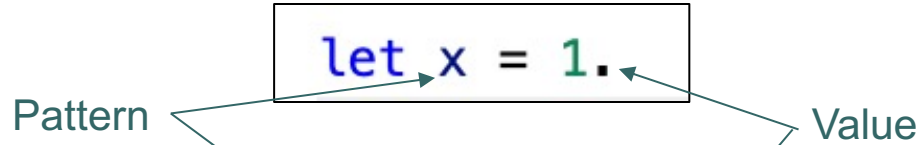
```
let <pattern> = <value>.
```

- where the pattern on the left side of the equal sign is matched against the value of the right side of the equal sign.
- Simple patterns are expressions that consist purely of constructors and variables



The Let Statement & Basic Pattern Matching

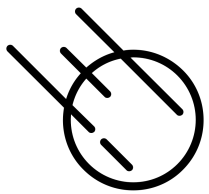
- When the pattern is just a single variable then the let statement looks like an assignment statement,



- However, statements like,

A diagram illustrating a let statement where both the pattern and the value are constructors. The statement is `let 1 = 1.` enclosed in a box. An arrow labeled "Pattern" points from the text "Pattern" to the first `1`. Another arrow labeled "Value" points from the text "Value" to the second `1.`.

- are completely legal,
 - the `1` on the left is a constructor viewed as pattern, the `1` on the right is a constructor viewed as a value.
 - highlighting the fact that the let statement is not equivalent to an assignment statement.



Basic Patterns

- In programs values are represented by constructors,
 - 1
 - "Hello, World!"
 - [1,2,3]
- Any structure that cannot be reduced any further consists purely of constructors and is the **minimal representation** of a value
 - [1,2] + [3] and 1 + 1 can be reduced further
 - [1,2,3] and 2 cannot be reduced further
- The latter are considered the minimal representations of the former values and consist purely of constructors
- The idea of values being represented purely by constructors is important for patterns and pattern matching



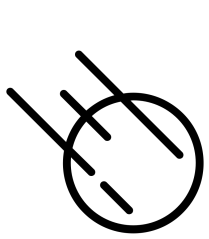
Basic Patterns

- **Def:** A pattern is an expression that consists of constructors and possibly variables.

```
[lutz$ asteroid
Asteroid Version 1.1.4
(c) University of Rhode Island
Type "asteroid -h" for help
Press CTRL-D to exit
[ast> let 1 = 1.
[ast> let 2 = 1 + 1.
[ast> let 1+1 = 2.
error: pattern match failed: term and pattern disagree on struct
[ast> let 1+1 = 1+1.
error: pattern match failed: term and pattern disagree on struct
ast> █
```

```
[lutz$ asteroid
Asteroid Version 1.1.4
(c) University of Rhode Island
Type "asteroid -h" for help
Press CTRL-D to exit
[ast> let [x,2,y] = [1]+[2]+[3].
[ast> x
1
[ast> y
3
ast> █
```

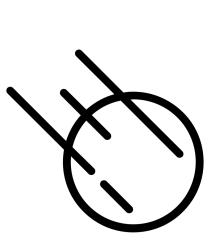
```
[lutz$ asteroid
Asteroid Version 1.1.4
(c) University of Rhode Island
Type "asteroid -h" for help
Press CTRL-D to exit
[ast> let x = 1.
[ast> x
1
[ast> let (x,2) = (1,2).
[ast> x
1
ast> █
```



Basic Patterns

```
[lutz$ asteroid
Asteroid Version 1.1.4
(c) University of Rhode Island
Type "asteroid -h" for help
Press CTRL-D to exit
[ast> structure A with
[....   data a.
[....   data b.
[.... end
[ast> let o = A(1,2). -- construct object
[ast> let A(1,2) = o.
[ast> let A(x,y) = o.
[ast> x
1
[ast> y
2
ast> █
```

- This also works for user defined structures/objects
- The expression `A(1,2)` on the left side is considered a constructor and also a pattern
- We can insert variables into the constructor, `A(x,y)`, for easy access to the components of the object `o`
 - destructuring



Destructuring

- The idea of destructuring is fundamental to pattern matching
- It makes access to substructures much more readable (and efficient).

Without structural pattern matching

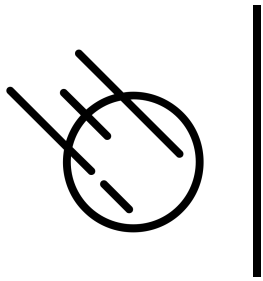
```
let p = (1,2).    -- create a structure
let x = p@0.      -- access first component
let y = p@1.      -- access second component
assert (x==1 and y==2).
```

In004/destruct1.ast

With structural pattern matching

In004/destruct2.ast

```
let p = (1,2).    -- create a structure
let (x,y) = p.    -- structural pattern matching, access to components
assert (x==1 and y==2).
```



Destructuring

- Here is another example using structures and objects

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

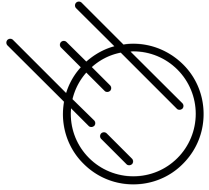
let joe = Person("Joe", 32, "Cook").  -- construct an object
let Person(n,a,p) = joe.             -- pattern match object

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



Basic Pattern Matching Summary

- The let statement
let <pattern> = value .
- On the right side of equal sign constructors represent values
 - Operators/functions are allowed
- On the left side they represent structure
 - Operators/functions are **not** allowed
 - Constructors must minimally represent structure



Pattern Matching in Python

- Limited pattern matching available with the assignment statement
 - Called **destructuring** assignment

```
[>>> (x,y) = (1,2)
[>>> x
1
[>>> y
2
[>>> [a,b,c] = [1,2,3]
[>>> a
1
[>>> b
2
[>>> c
3
>>> █
```



Pattern Matching in Python

- The match statement as of 3.10 provides a bit more functionality

```
>>> o = (1,2)
>>> match o:
...     case (1,2):
...         print("matched")
...     case _:
...         raise ValueError("not matched")
...
matched
>>> █
```

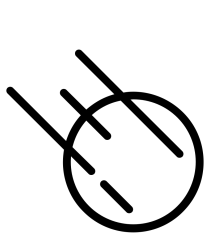
ln004/destruct1.py

```
class Person:
    def __init__(self, name, age, profession):
        self.name = name
        self.age = age
        self.profession = profession

joe = Person("Joe", 32, "Cook")

match joe:
    case Person(name=n, age=a, profession=p):
        pass
    case _:
        raise ValueError("match error")

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

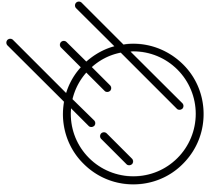


Conditional Pattern Matching

```
ast> let (x,y) if x==y = (1,1).  
ast> let (x,y) if x==y = (1,2).  
error: pattern match failed: conditional pattern match failed  
ast> █
```

```
ast> let x if x >= 0 = 1.  
ast> let x if x >= 0 = -11.  
error: pattern match failed: conditional pattern match failed  
ast> █
```

- Only assign a pair if the two component values are the same
- Only assign positive values to x



The is Predicate & Type Patterns

Note: a predicate is a function/operator that always returns true or false. No other return value is permitted.

- The is predicate is of the form
 <value> is <pattern>
and returns true if the value matches the pattern otherwise it will return false
- The is predicate allows us to do pattern matching in expressions

```
[ast> [1,2] is [x,2].  
true  
[ast> x  
1  
ast> █
```



The is Predicate & Type Patterns

- Type patterns are patterns of the form
 %<type name>
 and match all instances of the <type name>
- All built-in types have associated type patterns such as
 %integer, %real, %string etc.
- User defined types are also supported,
 %<user defined type name>

```
[ast> let %integer = 1.  
[ast> let %integer = 1.0.  
error: pattern match failed: expected type 'integer' got a term of type 'real'  
ast> █
```

```
[ast> struct MyType with  
error: expected 'EOF' found 'with'.  
[ast> structure MyType with  
[....    data a.  
[....    data b.  
[.... end  
[ast> let %MyType = MyType(1,2).  
[ast> let %MyType = 3.  
error: pattern match failed: expected type 'MyType' got an object of type 'integer'  
ast> █
```



Advanced Pattern Match Expressions

- We can combine conditional pattern matching with type patterns and the is predicate to express sophisticated patterns
- E.g., only assign a value to x if it is an integer value

```
[ast> let x if x is %integer = 1.  
[ast> x  
1  
[ast> let x if x is %integer = 1.0.  
error: pattern match failed: conditional pattern match failed  
ast> █
```



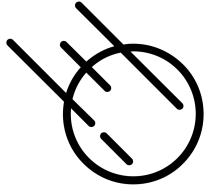
Advanced Pattern Match Expressions

- Here are some additional examples,

```
[ast> let x if (x is %real) and (x > 0.0) = 3.14.  
[ast> x  
3.14
```

```
ast> load system math.  
ast> let x if (x is %integer) and not math @mod (x,2) = 4.  
ast> x  
4  
ast> let x if (x is %integer) and not math @mod (x,2) = 5.  
error: pattern match failed: conditional pattern match failed  
ast> let x if (x is %integer) and not math @mod (x,2) = 4.0.  
error: pattern match failed: conditional pattern match failed  
ast> █
```

Note: 'mod' is the modulus function



Named Patterns

- The simple conditional pattern
x if x is <pattern>
appears a lot in Asteroid programs
- Named patterns of the form
x:<pattern>
represent a shorthand for the simple
conditional patterns above
- E.g.

```
[ast> let p if p is (x,y) = (1,2).  
[ast> p  
(1,2)  
[ast> let p:(x,y) = (1,2).  
[ast> p  
(1,2)  
ast> █
```




Named Patterns

- This shorthand notation is especially useful when combined with type patterns,

```
ast> let y if y is %integer = 1.  
ast> y  
1  
ast> let y:%integer = 1.  
ast> y  
1  
ast> █
```



Named Patterns

- Beware: even though named patterns with type patterns look like a declarations they are not!
- They are pattern match statements; consequently, implicit type conversions we are used to from other programming languages do not work!

```
ast> let x:%real = 1.  
error: pattern match failed: expected type 'real' got a term of type 'integer'  
ast> let x:%real = 1.0.  
ast> x  
1.0  
ast> █
```



Pattern Matching with Regular Expressions

- Regular expressions are patterns that can be applied to strings
- e.g., the regex
“a(b)*”
matches any string that starts with an a followed by zero or more b's.
- In Asteroid regular expressions are considered patterns and therefore we can write expressions like
“abbbb” is “a(b)*”
- Asteroid's regex syntax follows Python's regex syntax
 - <https://docs.python.org/3/library/re.html>



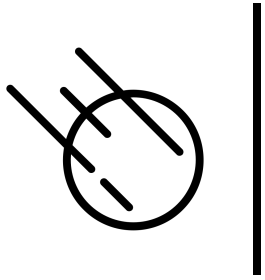
Pattern Matching with Regular Expressions

```
ast> "abbba" is "a(b)*a".
true
ast> "10101" is "(0|1)+".
true
ast> "-1001" is "-?(0|1)+".
true
ast> "1001" is "-?(0|1)+".
true
ast> "1002" is "-?(0|1)+".
false
ast> █
```

Note: $(a)^+ = a(a)^*$

Pattern matching with regex

```
1  -- using pattern matching to test whether
2  -- a specific element exists on a list
3
4  load system io.
5  load system type.
6
7  let l = ["turkey", "goose", "chicken", "blue jay"].
8
9  if type @tostring l is ".*blue jay.*" do
10 |   io @println "the Blue Jay is on the list".
11 else do
12 |   io @println "Blue Jay was not found".
13 end
```



Reading

- The Let Statement

- asteroid-lang.readthedocs.io/en/latest/User%20Guide.html#the-let-statement