

# Imperative Programming – Foundations

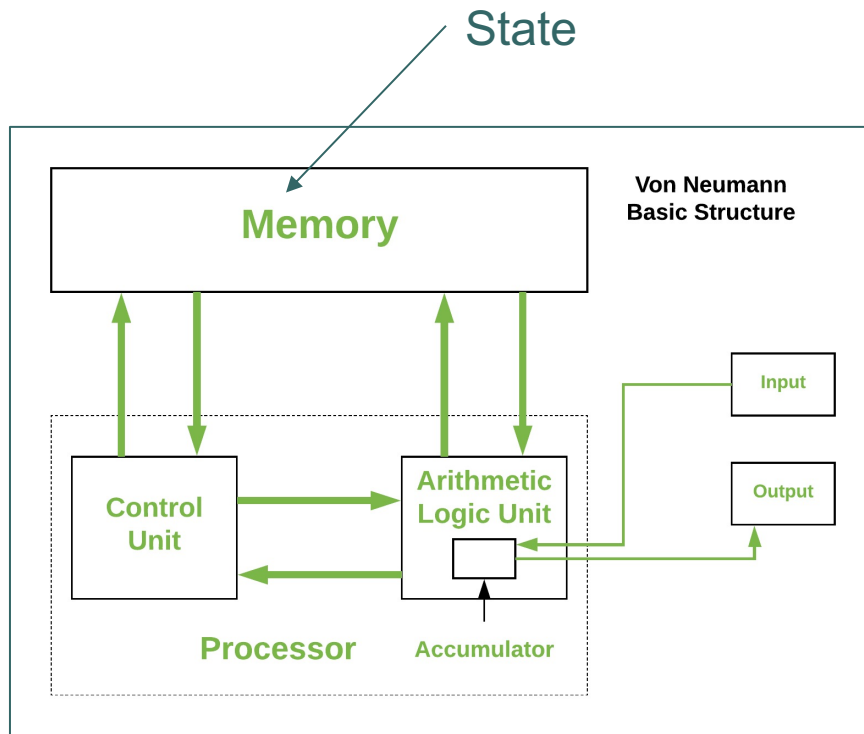
- The origins of imperative programming
- Types



# The von Newman Architecture



John von Newman, Hungarian mathematician, 1903-1957.



- John von Newman's computing model gave rise to the notion of imperative programming
- Assembly/machine instructions directly manipulate processor memory
  - Imperative in the sense that each instruction states what memory will look like after it executes
- The contents of the memory defines the state of the computation at any particular point in time



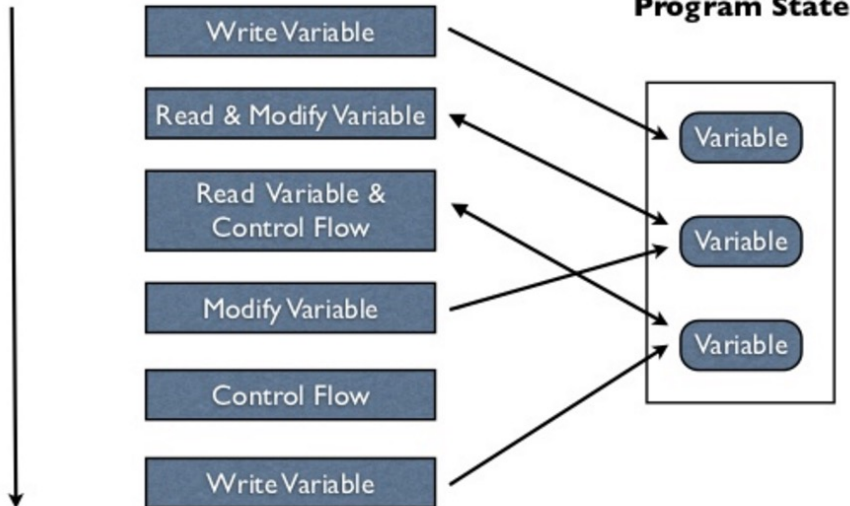
# Imperative Programming – Foundations

- In higher-level languages memory is abstracted into variables
  - This includes array/list variables
- Assembly/machine instructions are abstracted into programming language syntax
  - BUT, the assignment statement is still imperative, it tells us exactly what memory looks like after it executes.



# Imperative Programming – Foundations

Program Flow



## Imperative programming –

- Explicit statements that change the program state
- The program state is defined by the values assigned to the variables in a program
- The most common way to change the state in imperative programming is through an **explicit assignment of a new value to an existing variable**



# Imperative Programming – Foundations

- An example of an imperative program

```
-- sum the elements of a list
load system io.

-- initialize state
let lst = [1,2,3].
let sum = 0.

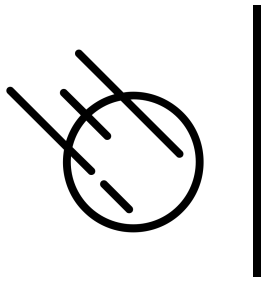
-- modify state each time around the loop by
--   (1) assigning a new value to x from the list
--   (2) incrementing sum by x
for x in lst do
|   let sum = sum + x.
end

io @println sum.
```



# Imperative Programming – Foundations

- Let's review basic type theory for programming languages
- This is important in order to understand
  - Type hierarchies
  - Type checking
  - Type promotion



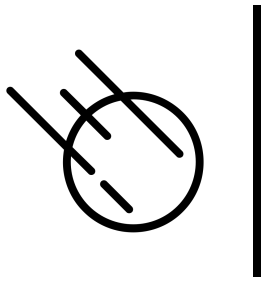
# Types

**A Type is a Set of Values**

Consider the Rust statement:

```
let n : i32 = 3;
```

Here we constrain `n` to take on any value from the set of all 32bit integer values.



# Types

**Def:** A type is a set of values.

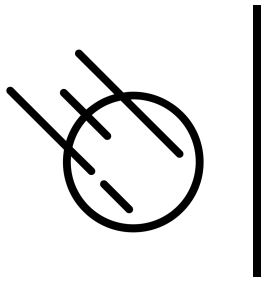
**Def:** A primitive type is a type that is built into the language, e.g., integer, string.

**Def:** A constructed type is a user defined type, e.g., any type introduced by the user.  
In Asteroid this is done through the 'structure' statement.

Example: Asteroid, primitive type

<u>q:%real = 1.1;</u>	}	q is of type real, only a value that is a member of the set of all real values can be assigned to q.
type real $\Rightarrow$ set of all possible real values		



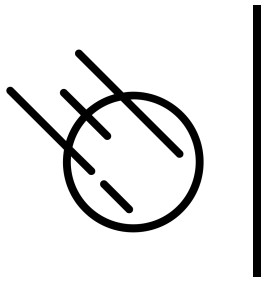


# Types

Example: Rust, constructed type

```
struct Rectangle {  
    xdim: i32,  
    ydim: i32,  
}  
  
fn main() {  
    let r:Rectangle = Rectangle { xdim: 3, ydim: 4 };  
}
```

Now the variable `r` only accepts values that are members of type `Rectangle`;  
☞ object instantiations of struct `Rectangle`.



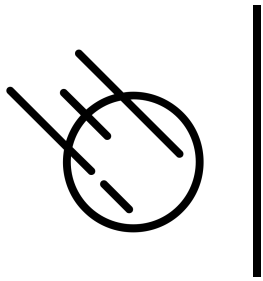
# Types

Example: Asteroid, constructed type

```
structure Rectangle with
  data xdim.
  data ydim.
end

let r:%Rectangle = Rectangle(4,2).
```

an element of  
type Rectangle.



# Subtypes

**Def:** a subtype is a subset of the elements of a type.

Example: C

Short is a subtype of int: `short < int`

The notation  $A < B$  means  
A is a subtype of B.

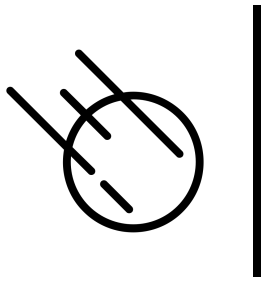
Observations:

- (1) converting a value of a subtype to a values of the super-type is called widening type conversion. (safe)
- (2) converting a value of a supertype to a value of a subtype is called narrowing type conversion. (not safe)

Example: C, partial type hierarchy

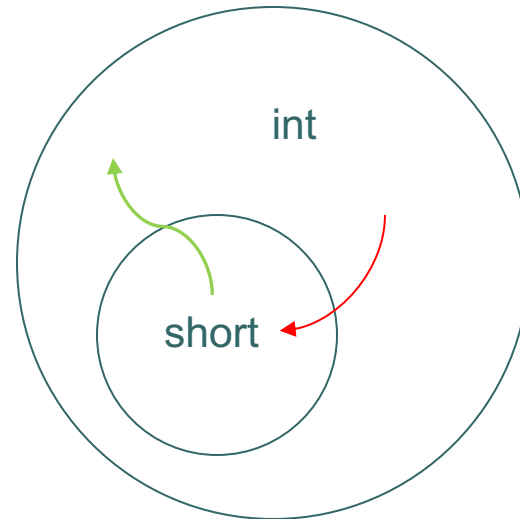
`char < short < int < float < double`

Subtypes give rise to type hierarchies and  
type hierarchies allow for automatic type  
coercion – widening conversions!

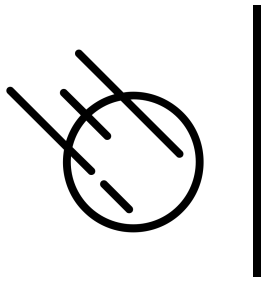


# Subtypes

- A convenient way to visualize subtypes is using Venn diagrams
- Consider,  
`short < int`
- It is easy to see that the shorts are a subset of the integer values
- The green arrow represents a widening type conversion is always safe
- The red arrow represents a narrowing type conversion and is never safe

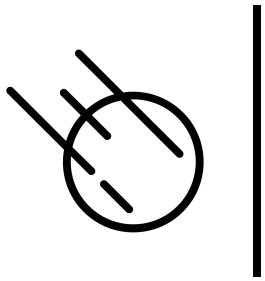


e.g. In Rust we have `i16 < i32`



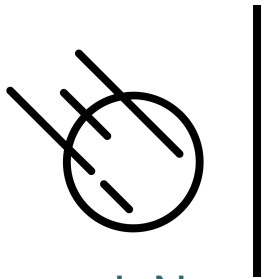
# Why do we use types?

- Types allow the language system to assist the developer write better programs. Type mismatches in a program usually indicate some sort of programming error.
  - Static type checking – check the types of all statements and expressions at compile time.
    - Rust
  - Dynamic type checking – check the types at runtime.
    - Asteroid
    - Python



# Type Equivalence

- Fundamental to type checking is the notion of type equivalence:
  - Figuring out whether two type description are equivalent or not
  - This is trivial for primitive types
  - But not so straight forward for constructed types like class/struct objects.



# Type Equivalence

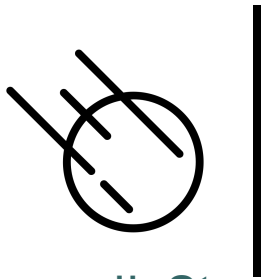
- I. Name (nominal) Equivalence – two objects are of the same type if and only if they share the same type name.

Example: Rust – constructed type

```
1 struct Type1 {x:i64, y:i64}
2 struct Type2 {x:i64, y:i64}
3
4 fn main () {
5     let x: Type1 = Type1{x:1,y:2};
6     let y: Type2 = x;
7     println!("{:?}",y);
8 }
```

**Error;** even though the types look the same, their names are different, therefore, Rust will not compile.

☞ Rust uses name equivalence



# Type Equivalence

II. Structural Equivalence – two objects are of the same type if and only if they share the same type structure.

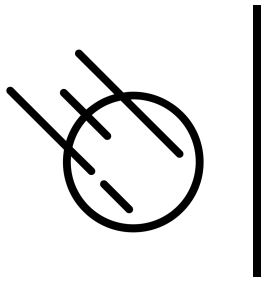
Example: Haskell

```
1  type Type1 = (Integer, Integer)
2  type Type2 = (Integer, Integer)
3
4  x :: Type1
5  y :: Type2
6
7  x = (1, 2)
8  y = x
```

Even though the type names are different, Haskell correctly recognizes this statement.

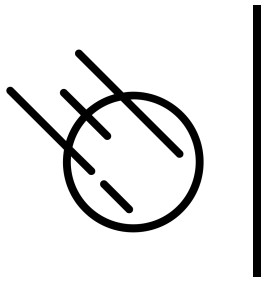
👉 Haskell uses structural equivalence.





# Type checking

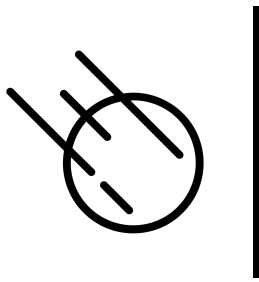
- Type checking refers to the process of making sure that all expressions and statements are properly typed.



# Type Checking

- Here is the Python type checker in action
  - int and str are not part of a common type hierarchy.

```
Python 3.8.10 (default, Nov 14 2022, 12:59:47)
[GCC 9.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> "my string" + 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
>>> █
```

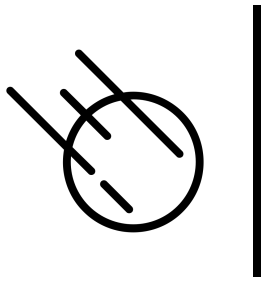


# Type Checking

- Here is the type checker of the Rust compiler in action
  - i16 < i32

```
fn main () {  
    let x:i32 = 3;  
    let y:i16 = 2*x;  
    print!("{}",y);  
}
```

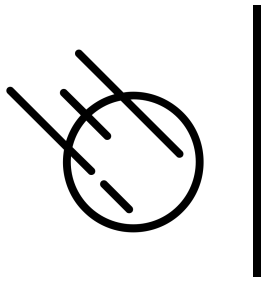
```
ubuntu$ rustc assign.rs  
error[E0308]: mismatched types  
--> assign.rs:3:16  
  
3 |     let y:i16 = 2*x;  
  |               ^^^ expected `i16`, found `i32`  
  |               |  
  |               expected due to this  
  
help: you can convert an `i32` to `i16` and panic if the converted value wouldn't fit  
  
3 |     let y:i16 = (2*x).try_into().unwrap();  
  |                   ~~~~~  
  
error: aborting due to previous error  
  
For more information about this error, try `rustc --explain E0308`.  
ubuntu$
```



# Type Checking in Asteroid

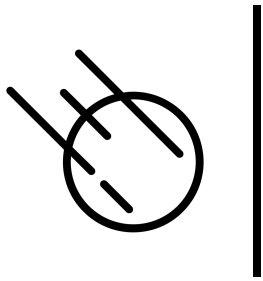
- The Asteroid type checker in action
  - Integer < real

```
Asteroid Version 1.1.4
(c) University of Rhode Island
Type "asteroid -h" for help
Press CTRL-D to exit
[ast> let x:%real = 3.1.
[ast> let y:%integer = x.
error: pattern match failed: expected type 'integer' got a term of type 'real'
ast> █
```



# Type Promotion

- Convert a subtype to a supertype (automatically)
  - Widening conversion
- This usually happens at the operator level



# Type Promotion - Python

- The addition operation is only defined for operands of the same type
- In order to apply the operator in a mixed-type situation one of the operands needs to be promoted
  - If promotion is not possible then flag a type error

```
Python 3.8.10 (default, Nov 14 2022, 12:59:47)
[GCC 9.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> isinstance(3.5 + 1, float)
True
>>> █
```

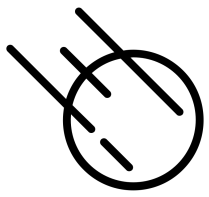
Promotion int → float



# Type Promotion - Asteroid

```
Asteroid Version 1.1.4
(c) University of Rhode Island
Type "asteroid -h" for help
Press CTRL-D to exit
ast> load system type.
ast> type @gettype (3.5 + 1).
real
ast> █
```

Promotion integer → real



# Imperative Programming – Asteroid

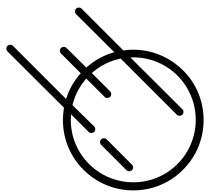
- Let's take a closer look at the imperative aspects of Asteroid
- We start with the type system





# Primitive Types & Constants in Asteroid

- 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



# Type Hierarchies

- Asteroid arranges primitive data types in a type hierarchy,
  - `boolean < integer < real < string`
- As we have seen, type hierarchies facilitate automatic type promotion

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

In002/let2.ast

Type promotion: plus as string concatenate op



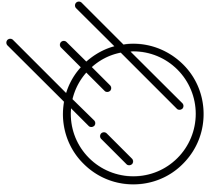
# Structured Data Types

- 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.
- Lists are mutable objects whereas tuples are immutable.
- Some examples,

Note:  $(1,) \neq (1)$

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

In002/let1.ast



# Structured Data Types

- 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,

```
Asteroid Version 1.1.4
(c) University of Rhode Island
Type "asteroid -h" for help
Press CTRL-D to exit
ast> load system io.
ast> io @println ("this is my list: " + [1,2,3]).
this is my list: [1,2,3]
ast> █
```



# The None Type

- Asteroid supports the **none** type.
- The **none** type has only one member
  - A constant named **none**.
  - The empty set of parentheses **()** can be used as a shorthand for the **none** constant.
  - That is: **none = ()**

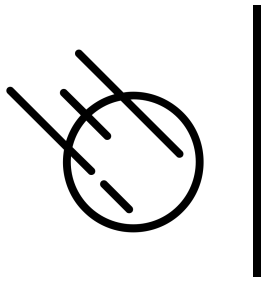


# Other Data Types

- In Asteroid we also have additional data types:
  - function
  - pattern
  - user defined data types via structures

```
load system type.  
  
-- define a function  
function inc with x do  
|   return x+1.  
end  
  
-- show that 'inc' is of type 'function'  
assert (type @gettype(inc) == "function").
```

In002/ftype.ast



# Reading

- The Basics

- [asteroid-lang.readthedocs.io/en/latest/User%20Guide.html#the-basics](https://asteroid-lang.readthedocs.io/en/latest/User%20Guide.html#the-basics)