FOBS-X: An Extensible Hybrid Functional-
Object-Oriented Scripting Language

James Gil de Lamadrid
Bowie State University, Bowie, MD, USA, 20715
jgildelamadrid@bowiestate.edu

Features of FOBS

- A single, simple, elegant data type called a FOB, that functions both as a function and an object.
- Stateless programming. In the runtime environment, mutable objects are not allowed. Mutation is accomplished, as in functional languages, by the creation of new objects with the required changes.

- A simple form of inheritance. A sub-FOB is built from another super-FOB, inheriting all attributes from the super-FOB in the process.
- A form of scoping that supports attribute overriding in inheritance. This allows a sub-FOB to replace data or behaviors inherited from a super-FOB.
- A macro expansion capability, enabling the user to introduce new syntax.

Simple FOBs

- A structure [*m* *i* -> *e* ^ ρ] where
  - *m* is a modifier (public, `+, protected `~, or argument, `$).
  - *i* is an identifier with a binding to expression *e*.
  - *e* is the value of the identifier.
  - ρ is the return value of the FOB, if invoked as a function.
- Example FOB: `` [`+x -> 3 ^ 6] ``

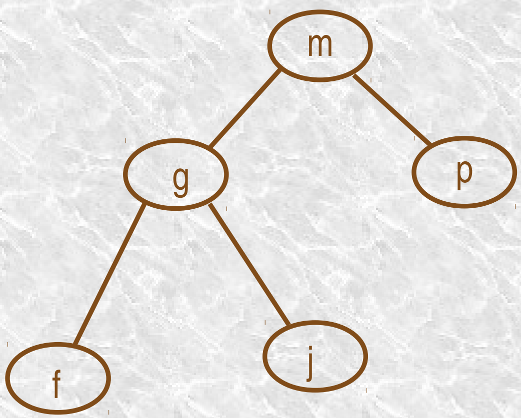
Primitive Types

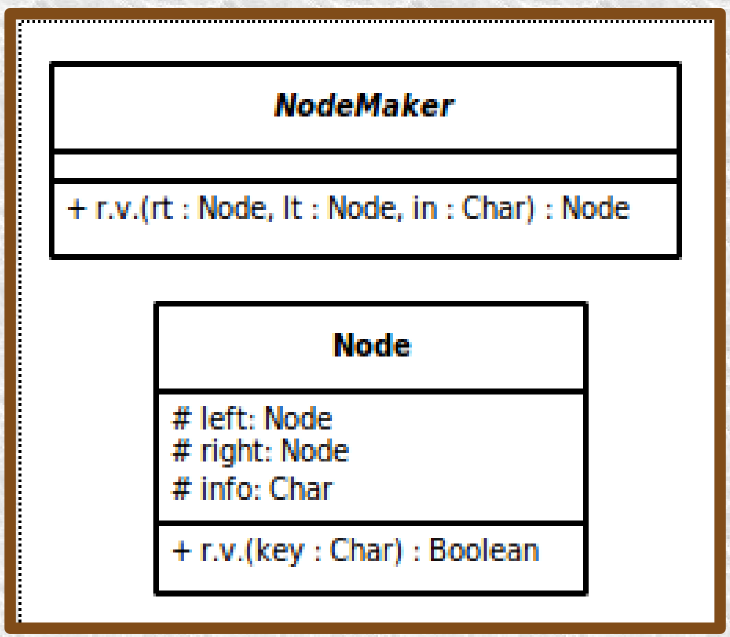
- Simple types: *Boolean*, *Char*, *Real*, *String*.
- Container type: *Vector.*
  - A heterogeneous immutable array with operations *head*, *tail*, *cons*, indexed read, and indexed write by copy.
  - Example: `[“abc”, 3, true]`.

Primitive Operations on FOBs

- Access – access the value of an identifier in a FOB:
  ```
  [`+x -> 3 ^ 6].x.
  ```
- Invoke – invoke a FOB as a function, giving the actual arguments in a Vector:
  ```
  [`$y -> _ ^ y.+[1]] [3]
  ```
- Combine – create a composite FOB out of two FOBS, implementing a simple form of inheritance with a FOB *stack*:
  ```
  [`+x -> 3 ^ _] ; [`$y -> _ ^ x.+[y]]
  ```
- FOB stacks:
  ```
  ([`+x -> 5 ^ _] ; [`$a -> _ ^ _] ;
  [`$b -> _ ^ a.*[b]]) [9, 2]
  ```

More Complex Example






```
## definition of the NodeMaker FOB
([NodeMaker ->
        [`$rt -> _ ^ _] ;
        [`$lt -> _ ^ _] ;
        [`$in -> _ ^ _] ;
        [`~Node ->
                [`~left -> lt ^ _] ;
                [`~right -> rt ^ _] ;
                [`~info -> in ^ _] ;
                [`$key -> _ ^
                        [`~a1 -> info.=[key] ^ _] ;
                        [`~a2 -> FOBS.isEmpty[left].|[a1].if[false,
                                left[key]] ^ _];
                        [`~a3 -> FOBS.isEmpty[right].|[a1].if[false,
                                right[key]]^_];
                        [`+a4 -> a1.|[a2].|[a3] ^ _].a4]
        ^ Node]
^ _] ;
## build the tree
[`+tree ->
        NodeMaker['m', NodeMaker['g', NodeMaker['f', _, _],
                NodeMaker['j', _, _]], NodeMaker['p', _, _]]
^_])
## search for 'f'
.tree['f']
#.
```

FOBS Semantics

- Variable Overriding – Redefinition completely overrides lower definition:

```
[`$m -> 'a' ^ m.toInt[]] ;
[`+m -> 3 ^ m]
```

- Argument Substitution – Actual arguments are substituted for formals by stacking on new definitions:

```
([`$r -> 5 ^ _] ;
[`$s -> 3 ^ r.+[s]]) [10, 6]
```

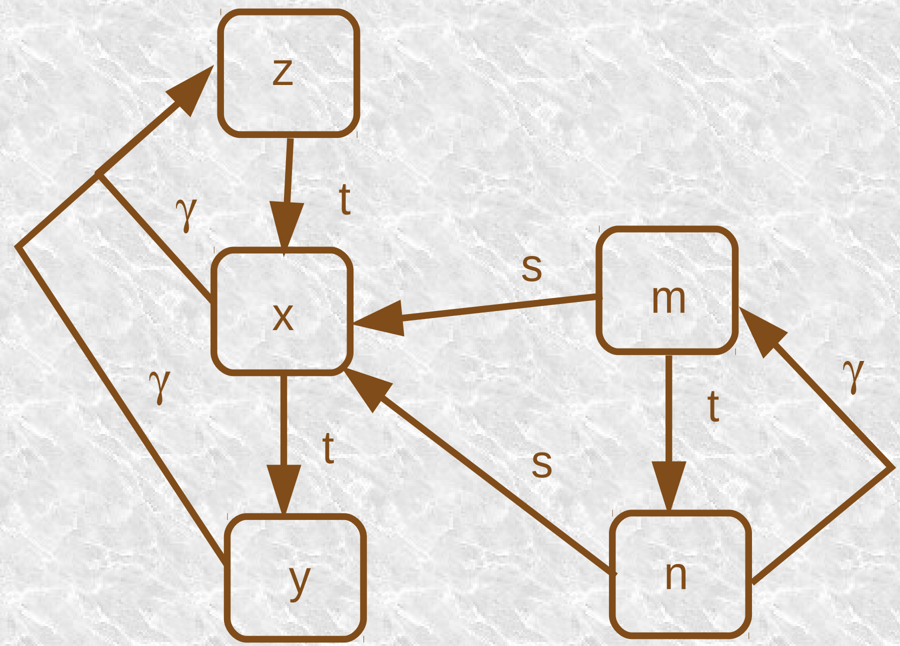
becomes

```
[`$r -> 5 ^ _] ;
[`$s -> 3 ^ r.+[s]] ;
[`+r -> 6 ^ r.+[s]] ;
[`+s -> 10 ^ r.+[s]]
```

- After binding the formal to the actual arguments, the ρ expression is evaluated.
- Variable Scope – A combined lexical and dynamic scope system is used.
- Pointers are used in the FOB to implement the scoping.
  - *s*: The enclosing FOB.
  - *t*: The FOB below in the FOB stack.
  - γ: The top FOB in the FOB stack.

- Example:

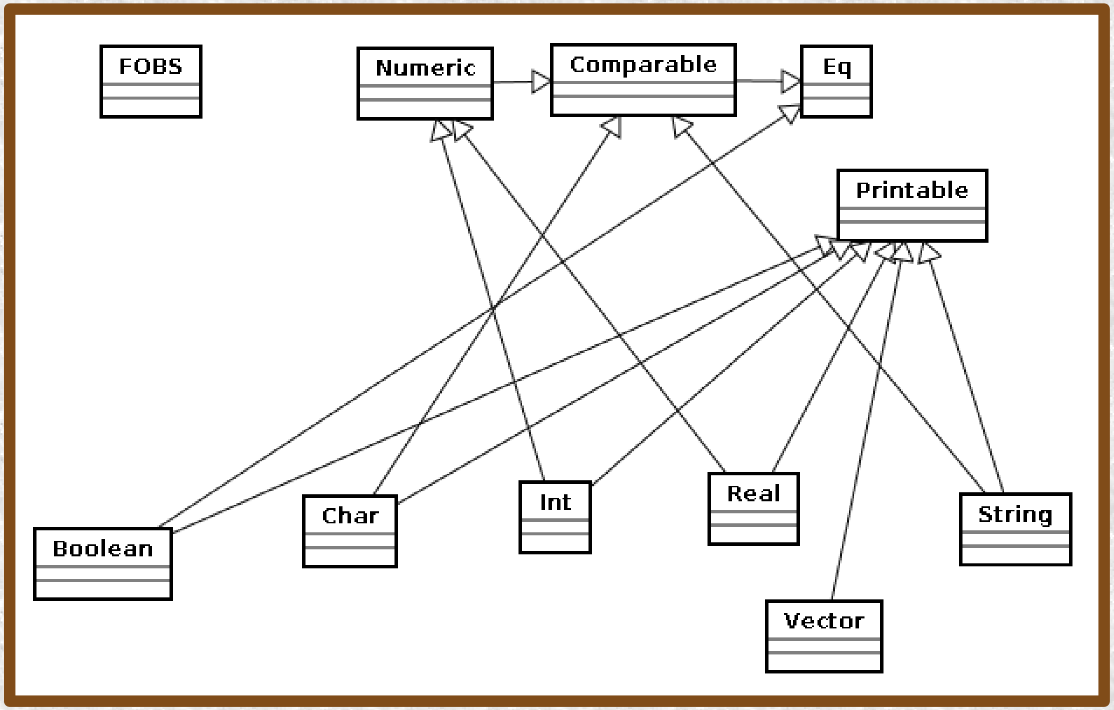
```
[`~y -> 1^_] ;
[`~x ->
   [`+n -> y + m
   ^ n] ;
   [`~m -> 2 ^_]
^_] ;
[`~z -> 3 ^x.n]
```



- Search order, starting at the variable reference.
  - Go to the top of the stack using the γ pointer.
  - Search down the stack using the *t* pointers.
  - Find the next lexical stack out using the *s* pointer
  - Repeat the process.

Library Structure

- The Library Contains:
  - Utility FOBS:
    - Numeric
    - Comparable
    - Eq
    - Printable
  - Primitive FOBS:
    - Boolean
    - Char
    - Int
    - Real
    - Vector
    - String
  - *FOBS* FOB: System utilities



Example Operations

<table>
<tr><th>Library FOB</th><th>Operation</th><th>Description</th></tr>
<tr><td>Boolean</td><td>b.if[x, y]</td><td>If boolean value b is true, return x, otherwise return y</td></tr>
<tr><td></td><td>b.&[x]</td><td>Return the boolean value of the expression $b \wedge x$</td></tr>
<tr><td></td><td>b.|[x]</td><td>Return the boolean value of the expression $b \vee x$</td></tr>
<tr><td></td><td>b.![]</td><td>Return the boolean value of the expression $\neg b$</td></tr>
<tr><td>Eq</td><td>e.=[x]</td><td>Return the boolean value of the expression $e = x$</td></tr>
</table>

Macro Expansion

- Macro definitions are of the form:
  <*S1* → *S2*: *P*, *d*>
  - *S1*: search string, including wild-cards.
  - *S2*:replacement string, including wild-cards.
  - *P*: priority of the macro (0 – 19).
  - *d*: direction of the macro (*r*, right-to-left, *l*, left-to-right).
- Macros allow the syntax of FOB-X to be almost completely redefined.

Example Macro

```
< #?multiplicand * #?multiplier →
  ( #?multiplicand .*
  [ #?multiplier ] ) : 9 , l >
```

- Wild-cards:
  - `#?multiplicand`
  - `#?multiplier`
- Matching `x * y`:

  `#?multiplicand ← x`, `#?multiplier ← y`
- Output: `(x.*[y])`

Macro Details

- Macros are expanded in passes, one pass per priority, highest priority first, implementing precedence levels
- Macros are scanned for in the indicated direction, implementing associativity.
- After a match, macro processing restarts the current priority pass, allowing macros that contain macros of the same or lessor priority.
- Wild-cards match *atoms*; tokens, and balanced bracketed sequences of atoms.
  - Bracketing characters:
    “(“, “)”, “{“, “}”, “[“, and “]”

Macro Syntax

- Keywords:
  - `#defleft`, `#defright`
  - `#as`, `#level`, `#end`
- Moving “*” to infix:
  - Move the operator, and change its name at priority 9.
  - Change the name back at priority 0.
  - Avoids having the “*” reprocessed by the same macro.

```
## numeric multiply
   operator
#defleft
   #?op1 * #?op2
#as
   ( #?op1 .:*: [ #op2 ] )
#level
   9
#end
#defleft
   :*:
#as
   *
#level
   0
#end
```

Standard Extension (SE-FOBS-X)

- Allow infix notation for most operators.
- Eliminate the cumbersome syntax associated with declaring a FOB.
- Introduce English keywords to replace some of the more cryptic notation.
- Allow some parts of the syntax to be optionally omitted.

Example FOB Declaration Macro

- `fob`, `ret`, `val`, “\” keywords are used to define a FOB stack.
- Example:

```
fob{x ret{3 * 5}\}
```

expands to

```
([`~x -> _ ^
  (3.*[5])] ; _)
```

```
#defleft
  fob { #?id ret
  { #*ret } \ #*x }
#as
  ( [ `~ #?id -> _ ^
  #*ret ] ; fob { #*x }
  )
#level
  3
#end
```

Further FOB Stack Example

- A two-FOB stack

```
fob{
  public x val{3} \
  y val{5} ret{x + y} \
}
```

- It expands to:

```
([`+x -> 3 ^ _] ;
 ([`~y -> 5 ^ (x.+[y])] ; _))
```

- Modifiers, `val` parts, and `ret` parts are all optional, using default values of protected, and the empty FOB.

A Larger Example

```
#use #SE
## definition of the NodeMaker FOB
(fob{
  NodeMaker
  val{
    fob{
      argument rt \
      argument lt \
      argument in \
      Node
      val{
        fob{
```

```
left val {lt} \
right val {rt} \
info val {in} \
argument key
ret{
  (fob{
    a1 val {info = key} \
    a2
    val{
      if {nofob left | a1}
      then {false}
      else {left[key]}
    } \
    a3
    val{
      if {nofob right | a1 | a2}
      then {false}
      else {right[key]}
    } \
```

```
            public a4 val{a1 | a2 | a3} \
            } ).a4
          } \
        }
      }
      ret {Node} \
    }
  } \
  ## build the sample tree
  public tree
  val{
    NodeMaker['m', NodeMaker['g', NodeMaker['f', _, _],
      NodeMaker['j', _, _]], NodeMaker['p', _, _]]
  } \
} )
## use the main FOB tree variable to search for 'f'
.tree['f']
#.
#!
```

Features of the Example

- Directives: *#!*, end of script, *#.*, end of expression.
- *#use*: Install an extension by loading the macro definitions, and installing a module in the library.
- New syntax: *if* construct, and the *nofob* operator replacing the *isEmpty* operator from the *FOBS* FOB.
- *public* and *argument* modifier names.
- The *fob-val-ret* construct with optional parts.

Conclusion

- A Simple core-FOBS-X provides a combined object-oriented and functional environment, with a simple construct.
- A macro processor allows the language syntax to be reconfigured to a large degree.
- Future work: In the future the library will be configurable using the *FOBS* FOB, allowing interface with the scripted environment.