Programação Avançada

- Marcos Santos, nº 94051
- João Marques , nº 90865

Make_Class

```
C1 = make_class(:C1, [], [:a])
C2 = make_class(:C2, [], [:b, :c])
C3 = make_class(:C3, [C1, C2], [:d])
```

Make_class

```julia
struct Class
    name :: Symbol
    superclasses :: Vector{Class}
    slots :: Set{Symbol}
end

function make_class(name, superclasses, slots)
    for sc in superclasses
        slots = [slots..., sc.slots...]
    end
    Class(name, superclasses, Set(slots))
end
```

DefClass

```
@defclass(C1, [], a)
@defclass(C2, [], b, c)
@defclass(C3, [C1, C2], d)
```

```
macro defclass(name, superclasses, slots...)
    :($(esc(name)) = make_class($(Meta.quot(name)), $superclasses, $slots))
end
```

Make_instance

```
c3i1 = make_instance(C3, :a=>1, :b=>2, :c=>3, :d=>4)
c3i2 = make_instance(C3, :b=>2)
```

```
make_instance(class, slot_val...) = Instance(class, slot_val...)
```

Make_instance

```
struct Instance
    class :: Class
    slot_val :: Dict{Symbol, Any}
    function Instance(class, slot_val...)
        dict = Dict()
        for (s,v) in slot_val
            if any(x -> x == s, class.slots)
                dict[s] = v
            else
                error("Slot ", s, " is missing")
            end
        end
        new(class, dict)
    end
end
```

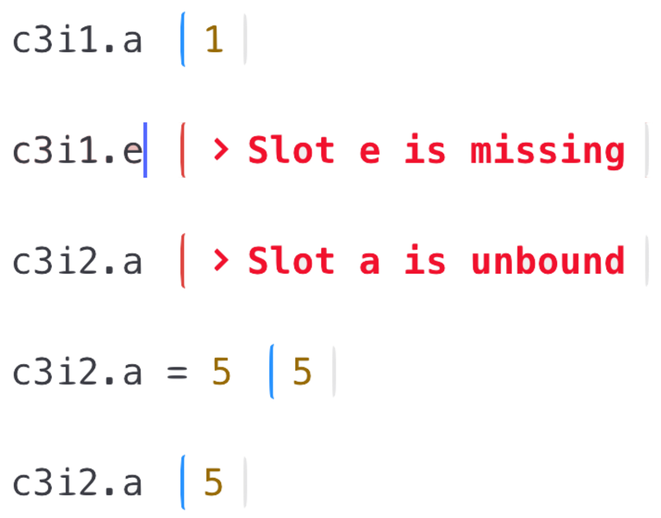
Get/Set_slot

```
get_slot(c3i2, :b)      2

set_slot!(c3i2, :b, 3)  3
```

Get/Set_slot

```julia
function get_slot(x::Instance, field::Symbol)
    slot_val = getfield(x, :slot_val)
    if all(sv -> sv[1] != field, slot_val)
        error("Slot ", field, any(slot -> slot == field
                , getfield(x, :class).slots) ?  " is unbound" : " is missing")
    end
    slot_val[field]
end

function set_slot!(x::Instance, field::Symbol, value)
    if all(slot -> slot != field, getfield(x, :class).slots)
        error("Slot ", field, " is missing")
    end
    getfield(x, :slot_val)[field] = value
end
```

Get/Set_property



```
c3i1.a          1

c3i1.e          > Slot e is missing

c3i2.a          > Slot a is unbound

c3i2.a = 5      5

c3i2.a          5
```

```julia
Base.getproperty(x::Instance, field::Symbol) = get_slot(x, field)
Base.setproperty!(x::Instance, field::Symbol, value) = set_slot!(x, field, value)
```

Def_generic/method

```julia
mutable struct Method
    types :: Vector{Symbol}
    lambda :: Function
end

mutable struct Generic
    name :: Symbol
    parameters :: Vector{Symbol}
    methods :: Vector{Method}
end
```

Def_generic/method

```
macro defgeneric(expr)
    name = expr.args[1]
    parameters = expr.args[2:end]
    if length(parameters) != length(Set(parameters))
        error("Duplicate variable name")
    end
    :($(esc(name)) = Generic($(Base.Meta.quot(name)), $parameters, $[]))
end
```

Def_generic/method

```julia
macro defmethod(expr)
    name = expr.args[1].args[1]
    parameters = expr.args[1].args[2:end]
    body = expr.args[2].args[2]

    varnames = [p.args[1] for p in parameters]
    vartypes = [p.args[2] for p in parameters]

    if length(varnames) != length(Set(varnames))
        error("Duplicate variable name")
    end

    :(defmethod($(name), $vartypes, ($(varnames...),)->$body))
end
```

Def_generic/method

```
function defmethod(gen, vartypes, lambda)
    if length(vartypes) != length(gen.parameters)
        error("Required ", length(gen.parameters),
            " parameter(s) but ", length(vartypes), " given")
    end

    for m in gen.methods
        if m.types == vartypes
            m.lambda = lambda
            return
        end
    end

    gen.methods = [gen.methods..., Method(vartypes, lambda)]
end
```

Def_generic/method

```julia
function getEffectiveMethod(methods, args...)
    args_classes = [getfield(a, :class) for a in args]

    for ac in expand(args_classes)
        method = findApplicable(methods, ac)
        if method != nothing
            return method
        end
    end

    error("No applicable method")
end

(f::Method)(args...) = f.lambda(args...)
(f::Generic)(args...) = getEffectiveMethod(f.methods, args...)(args...)
```

Def_generic/method

```
function recursive(idx, args_classes, ignore)
    ret = ignore ? [] : [[c.name for c in args_classes]]
    for class in args_classes[idx].superclasses
        ret = [ret..., recursive(
                    idx
                    , [args_classes[1:idx-1]..., class, args_classes[idx+1:end]...]
                    , false)...]
    end
    return ret
end

function expand(args_classes)
    expanded = [[c.name for c in args_classes]]
    for idx in range(1,length=length(args_classes))
        expanded = [expanded..., recursive(idx, args_classes, true)...]
    end
    return expanded
end
```

Def_generic/method

```
function findApplicable(methods, args_classes)
    for method in methods
        if method.types == args_classes
            return method
        end
    end
    return nothing
end
```