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

```
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
```

Def Class

```
@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_Slot/Set_Slot

```
get_slot(c3i2, :b)  | 2

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

Get_Slot/Set_Slot

```
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_Property/Set_Property

```
c3i1.a | 1
c3i1.e | > Slot e is missing
c3i2.a | > Slot a is unbound
c3i2.a = 5 | 5
c3i2.a | 5
```

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

Def_Generic/Def_Method

```
@defgeneric add(x,y)
@defmethod add(x::Int64, y::Int64) = x + y

add(2, 2)
```

Def_Generic/Def_Method

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

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

Def_Generic/Def_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, Dict()))
end
```

Def_Generic/Def_Method

```
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/Def_Method

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

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

Def_Generic/Def_Method

```
@defgeneric add(x,y)
@defmethod add(x::Int64, y::Int64) = x + y

add(2, 2)
```

Def_Generic/Def_Method

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

(f::Method)(args...) = f.lambda(args...)


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

    for types in getPermutations(argstypes)
        if haskey(methods, types)
            return methods[types]
        end
    end

    error("No applicable method")
end
```

Def_Generic/Def_Method

```
function expand(idx, argstypes)
    expanded = [[c.name for c in argstypes]]
    for type in argstypes[idx].superclasses
        expanded = [expanded..., expand(
                    idx
                    , [argstypes[1:idx-1]..., type, argstypes[idx+1:end]...])...]
    end
    return expanded
end

function getPermutations(argstypes)
    expanded = [[c.name for c in argstypes]]
    for idx in range(1, length=length(argstypes))
        for type in argstypes[idx].superclasses
            expanded = [expanded..., expand(
                        idx
                        , [argstypes[1:idx-1]..., type, argstypes[idx+1:end]...])...]
        end
    end
    return expanded
end
```

Def_Generic/Def_Method

```
[:IstStudent, :IstStudent]
[:Student, :IstStudent]
[:Person, :IstStudent]
[:Sportsman, :IstStudent]
[:IstStudent, :Student]
[:IstStudent, :Person]
[:IstStudent, :Sportsman]
```

Extensions

- Before/After Methods

```
@defmethod :before add(x::Int64, y::Int64) = print("(x + y) = ")
@defmethod :after add(x::Int64, y::Int64) = print(";")
```

Before/After Methods

```
mutable struct Generic
    name :: Symbol
    parameters :: Vector{Symbol}
    before_methods :: Dict{Array,Method}
    methods :: Dict{Array,Method}
    after_methods :: Dict{Array,Method}

    function Generic(class, parameters)
        new(class, parameters, Dict(), Dict(), Dict())
    end
end
```

Before/After Methods

```
macro defmethod(expr)
    :(@defmethod(nothing, $expr))
end

macro defmethod(qualifier, 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, $qualifier))
end
```

Before/After Methods

```
(f::Generic)(args...) = applyEffectiveMethods(f, args...)
```

Before/After Methods

```
function applyEffectiveMethods(f, args...)
    expanded = getPermutations([getfield(a, :class) for a in args])

    before = getEffectiveMethods(f.before_methods, expanded)
    method = getEffectiveMethods(f.methods, expanded)
    after = getEffectiveMethods(f.after_methods, expanded)

    if length(method) == 0
        if length(before) != 0 || length(after) != 0
            error("No primary method")
        else
            error("No applicable method")
        end
    end

    for b in before b(args...) end
    res = method[1](args...)
    for a in after a(args...) end

    return res
end
```

Before/After Methods

```
function getEffectiveMethods(methods, expanded)
    return [methods[types] for types in expanded if haskey(methods, types)]
end
```