 # silver: Reduced and Reflective Programming

- MIT licensed open source transpiler and JIT engine of entire packaged applications.
- Build and execute in module.
- Import from anywhere and describe installation of external libraries through **import** keyword.
- *'will take 3 days to make an alpha experimental...' -8 weeks ago when it wasn't even 3% designed*

*# **import** keyword*

import encapsulates software versioning, installation, and effective #**define** and #**include**

We describe building flags where you interface with your entry points. Quickly files like this become top-level build files.
However, it's silver language, not CMake, Makefile, or various build-scripting engines.
It's the same language you use for your app, and it's your objects. Runs by transpiled C99, or JIT.
silver serves a scripted, or middleware use-case and debugs with symbol compatible C99

import

name: skia *# this is a token type, you are allowed to express yourself including-hyphens*
source: '//skia.googlesource.com/skia@f4f4f4'
shell: "build.sh" *# cmake, meson, shell args exists for compilation facility*
links: "skia"
includes: ["skia/skia.h", "skia/skia2.h"]
defines: [DEF:1]

silver module example

widget.si
int something: 2

import widget
 # token only is for silver modules
 # can access widget.something

import widget **as** w-another
*# alias a module name with **as** keyword*
 # can now access w-another.something
 # use of - is two-fold: minus operator or character in token
*# **one must separate** A - B in operations to avoid **forming-tokens***
 # symbols-are-this-way in file-systems, with underscore being disallowed on URL;
this-is-ok: performs tasks without the need to re-map reflected names for web/majority use-cases

*# operators **** *** / // + - require white-space due to this logic*
*# **cast** methods*
*# **operator** methods – classes operate with typed arguments*
*# **expr** blocks - connect and mold code into operator-like expression blocks*

 # reduction of (parenthesis, arg framing), { code block } into syntax

int, **string** len-and-str [**string** arg]

  return arg.length, arg.upper-case *# all returns must return the same types*

int arg, **string** arg2: len-and-str [arg: '{test}'] *# optional to use named args, similar to* ***python … make use of the multiple returns***

*# **mod** – form of tapestry in middle-ware medium (imagined: **Novis** et al)*

*# more-than-class: enabled syntax for use-case, since you can **mod**ify them.*

mod-name [[*template-args* ::] *construct-args*] *controlled-code*

  or-code-in-block (not both)

 # We have a construct for syntax – Now start coding the language? How peculiar!

 # … implications, or implementations of such

mod if

  intern bool condition

  if [**bool** condition]

  expr [**code** fn] **@if** [condition, fn] *# calls the **A-type** method in silver design-time*

*## **mod** keyword capabilities*

silver as an element and given thermodynamic properties is most compatible with the idea of being able to 'mod' any mods from any point.

*It's maliable, moldable: **reasoning**: name-trait-attrib.*

*The last to describe (end-user) can describe last, and effectively alter the implementation – **intern**'s too (**see foot-note: 1**)*

  *(1) one can still describe internal to the module and have it hidden by making it an **intern** module's member*

Implicit trust seems to provide better results than implicit/automatic denial

  i. for open source, let's cooperate-with rather than dictate-to each other. Keano [this is a still a dictation…]

##

 # meta-described classes unless intern specified

 # broadcasting assignments are natural, and primitive types can broadcast as copies

 # expressions at chain length is up for assignment with multiple returns, except when declared in args

 # the requests are applicable to implemention-level only

***method** [arguments [**for-impl.** :: requests]]*

*## **template-args :: args** or **just-args** or **template-args***

***expr** methods are selected based on the data of expression, as matched by described members.*

more control over 'control-flow' because we are all first class in our design access.

  *** if you perform expression on something it isnt binding to, then it's an error at design stage*

##

template [**T**]
mod print
  expr [**string** message, **any**[] args]
    string m = message.format [args]
    @**print** [m] *# for many functions there is a basic @run-time call*

template [**T**]
mod switch
  intern T value
  intern bool has-default
  intern default default-node
  expose alias case**:** case [**T**] *# when using a **switch** [T] we use a **case** [T] – redefining this allows compatible instancing or error at design time*
  intern case[] cases
  switch [**T** value] *# this template is still used by switch [any-value] – we do not specify T as its been figured out at the argument*
  expr [**case**[] cases] *# cases will be auto-copied*
  compile [**case**[] cases] *# compile-time checks – design silver's validation checks for your keywords*
    @error-if cases.length == 0, 'no cases given'
    if [T enumerable]
      int count: T.tokens.length
      int min: T.min, max: T.max
      bool[T] bits-set
      for [case c] cases
        if bits-set.contains [c.value] @error *'case covered'*

template [**T**]
mod case
  T value
  code fn
  case [**T** value]
  expr [**code** fn] fn:fn

template [**T**]
mod default: case
  # code should be peer-aware, meaning it can look at the code around it, know where its being embedded – more reduction potential
  # silver's declarative approach: by enabling mount with type selectors we allow syntax to be consumed in the way its expressed, for your data
  # another error type is no suitable-selector found, when there is >= 1 mount selector
  mount [**switch**] # this is a valid way of using types – not until you have more than one do you need a variable name
    switch.has-default: true *# instance members first, then static – you don't have both in natural circumstances*
    switch.default-node: this
 # this part i am unsure of in how the : causes the expression split – the lack of variable name? …
int i: 0
switch [i]
  case 0: **return** 1 *# the colon-token ':' ends the expression segment on that line (so mod does not absorb it) – it can never be assign in these cases*
  case 2: **goto** 3
  case 3: **return** 2
  default: return 3

*# **methods with no args is not expressed with []: that's an error,***
 # so you can have property-like methods and reduced, prettier syntax
 # we can still get its address by method.address

*# **intern** is about external mod visibility and if you are reflecting the member*
 # If you are modifying an existing mod, you are certainly stating intention in code.
*# We have been **mod** ('ifying) each other's code for decades – this facilitates that more – more of a block-chain ready language.*
 # Evaluation of the network could-be how much improvement there is to a given module in relation

operator keyword

operator methods are performed on neighboring expression; with operator you may horizontally-flavor your language.
below, our cast method
you may invoke operators by instance-name[arg, arg2]
array-name[2] or array-name 2 or 2 array-name

```
proto is-keyword
mod keyword observes is-keyword

mod break : keyword
    int levels
    keyword reg: null
    break [ int levels: 1 ]
    register [ keyword kw ]
```

top-level mod templates

```
template [ T ]
mod range
   T from
   T to
   range [ T from, T to ]
   # matching precedence goes to first described class
   bool operator [ T check ]       # operator args:    object [ go-in-here ]    or-for-single-args-only:   go-in-here  object    or    go-in-here  object
       return check >= from && check <= to

template [ T ]
mod from
   T from
    range operator [ to b ] return [ from, b.to ]
    from [ T from ]

template [ T ]
mod to
   T to
    to [ T to ]
```

 # another name for this is prototype – they can't be instanced but can be conformed to

```
proto expr-user
```

```
    # ... performs a collection, with compatible elements going into any ( all )
    expr [ code fn ]

proto accepts-break
    accept [ break ] # added during design
```

*## **expose** keyword*

```
    Available to user in scope; once described
    Typically these will be passed in direct from template args, when the user describes the variables themselves
##

# here we are defaulting int
# the name can be yours if you need data for quantified range-progress
template [ T state-bind:int  ]
mod for : keyword observes expr-user, accepts-break
    expose T state-bind
    intern range iter
    for [ range iter ]
    for [ ] # a range with a step of 0 should be infinite
    accept [ break ]
        break.register this
    expr [ code fn ]
        @for-state-range iter.start, iter.end, iter.step, ref state-bind, fn

# express for statement
for [ :: ]
    print '1 way to infinite loop\n'

# templates should have a sub-expression for macro
template [ macro M  ]
mod while
    expr [ code fn ]
        for [ :: ]
            if [ ! @if [ M, fn ] ]
                break

# express while statement
while [ true ]
    print [ 'another way' ]

# a for statement with a range 0 to 10 (includes 10); up-to could allow for 1 unit less
for [ int a :: from 0 to 10 ]
    print '{ a } ... a = %i\n', [ a ]
    print "user-level interpolation only: a = %i\n", [ a ]
```

```
# map responds to first, and last, so that gives us a range of field
# field should have expose'd members for selecting
# if they do not map properly with the class operated on, then it will error
for [ string key, int value :: map ]
    print '{ key } and { value }'


mod field
    expose any key   # expose works in context of pulling args
    expose any value

mod item
    item prev: null  # null is different from a default-state – one null allows us to avoid getting into recursion, too
    item next: null
    field element  # element's value can hold a value, or key and value; an item has everything it needs for identity

# this helps one reduce code and take the load off the mods that implement
# this gives proto objects some over-arching controls which it can perform Reflection on, if needed
proto itemized
    item  first[]
    item   last[]
    int     size[]
    clear [ ]
        print 'printing this line at the itemized proto method, then invoking the mods clear'
        :: clear [ ]      # climb back in scope one, which in this case will always give us the mod
# only a proto's call to a method of the same implementation will call that user method (approach interfaces as controllers)

mod list observes itemize
# observes is seen as implements, but also provides optional method invocation from your own –
# application-level broadcast, useful as a signal handling scheme; its also an interface into any arb systems and can be architected-as
# for all of the times a given mod is augmented, it will accumulate its effective observing interface-traits
    intern item f: null
    intern item l: null
    int      count           # allow range to respect our count without iterating for it
    item first[] return f
    item  last[] return l
    int     size[] return count

    template [ T ]
    T operator [ int index ]
        int i: 0
        if [ index > 0 ]
            for [ item cur :: range [ f, l, count ] ]
                if [ i = index ] return cur
                    i += 1
        @error 'out of bounds '
        return T[]
```

```
    template [ T ]
    T pop [ ]
        @error-if [ ! f, 'list is empty' ]
        item o: l
        l: l.prev
        if [ !l ] f: null
        count -= 1
        return o.field.value

    clear [ ] # with no type given, always returns the instance
        while [ f ]  f:f.next
        l:      null
        count: 0

# end-list

mod map
    list[]   hash       # hash [ key % map-size ] -> list of fields to match key
    list     fields
    int      count
    const num map-size 64

    intern hash-fields [ string key ]
        return hash [ key.hash % hash.size ]

    map [ ]
        hash.set-size [ map-size ]    # construct bucket classes

    template [ T ]
    T operator [ string key ]
        list f : hash-fields [ key ]
        for [ field i :: f ] if [ i.key = key ] return i.value
        f += field [ key: key, value: T[] ]
        return f.last.value

    bool remove [ string key ]
        int rem-matches [ list f, int sum ] :
            int index: 0
            for [ field i :: f ]
                if [ i.key = key ]
                    f.remove [ index ]
                    return sum + 1
                index += 1
            return 0
        int success: rem-matches [ hash-fields [ key ],  rem-matches [ fields, 0 ] ]
```

```
        @assert [ success = 2 || success = 0 ]
        return success = 2

    num size []
        return count


mod import
      import [ token token-name ]
          @import token-name
```

silver should always import *.si in the default group (language, math, data)
 # the trait checks are checking for 'implements'
*# **important-note:** proto-col's, interfaces and traits are contained in model*

*# **alias** keyword – closest aliases always win in the type selection*

 # this is to allow us to not see map and array everywhere, but something more user-described
 # override them to change the map or array type

 # the lambdas generated would be fairly C99 efficient, basically the same as implementing it in the run-time
 # lets express as much in silver as we can
 # its the language of the user. you can do whatever you want, to use reflection as much as possible in the process.

```
template [ E: any ]
mod array
     # we are pulling from alias space, rather than traditional name space
    intern ref E elements    # references hide inside any
    read-only int size, count  # you can't write to labels as externs
    array [ int size ]
        elements: @new-null [ E, size ]   # @new-null is a basic run-time for managing our own vector allocations
        size: size
    E operator [ int index ]  return @element-at [ elements, index ]
     delete [ int index ]
        for [ int i :: index + 1 to count ]
            elements[ i - 1 ]: elements[ i ]
     append [ E e ]
        if [ size = count ]
              size: 32 + size * 4
              elements: @resize [ elements, count, size ] # ref buffers are ref counted internally
        elements [ count++ ] : e
```

 # now we have a way to use .NET style type syntax for array types
 # also, swap out array for your own method and keep the syntax.

```
template [ E : any ]
alias E [] : array [ E ]
```

 # the types work at design-time here in order to facilitate the most you can do while retaining context and control

 # the syntax is basic, but the options are not limited – we could re-alias to an unordered map or a strict-type-based one (non-any)

template *[* ***K :*** *hashable,* ***V :*** *any]*

alias *K [V] :* ***map***

***enum*** *type*

 # enums are enumerable at runtime, with a default of your liking [otherwise set to first by default]

enum *etype* ***default*** *token*

token: 1

token-two

 # lets enumerate the etype

for [***symbol*** *:: etype.symbols]*

print ' enumerable { e-type.name } / { symbol.name } : { symbol.value } '

mod e-user

etype e : etype.token-two

 # invoking ***silver*** *executable*

silver main.si arg1:1 *# graph and run with optional args*

silver –compile main.si arg1:1 *# graph, transpile-C99, compile and run-exe with args*

silver –compile-only main.si *# compile without running*