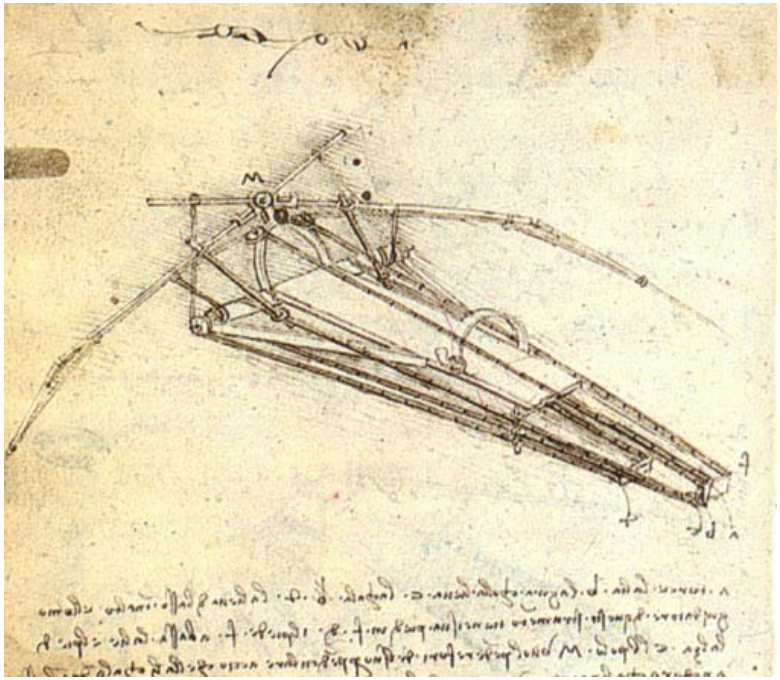
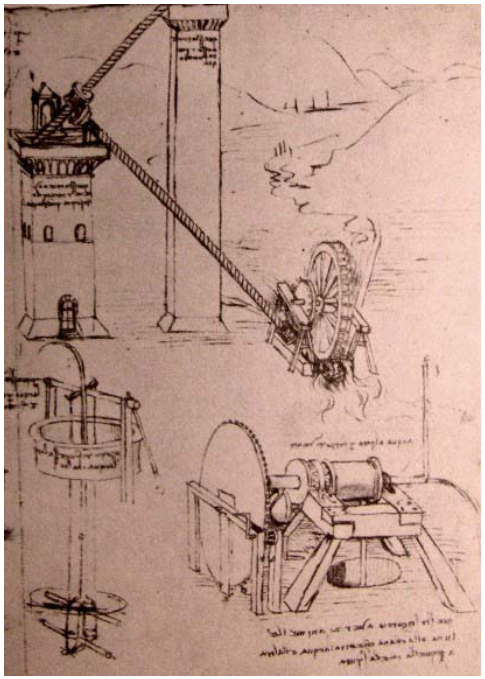
HY540 – Advanced Topics in Programming Language Development

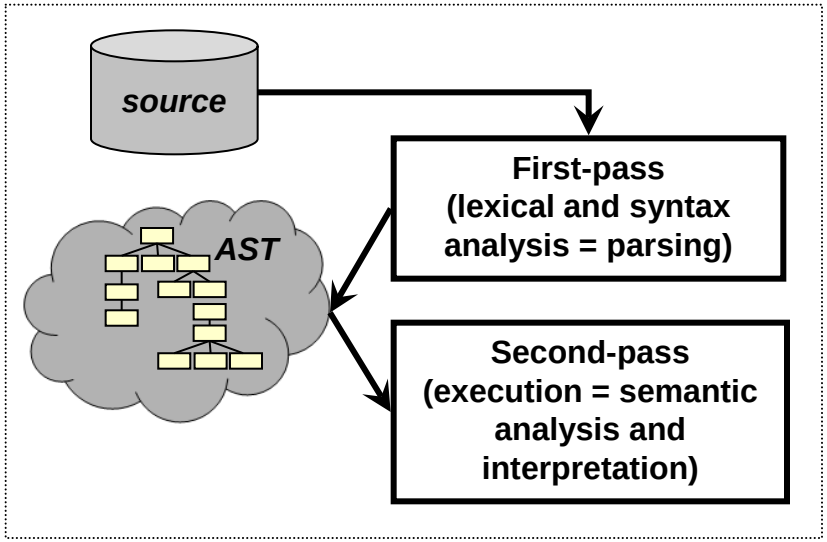


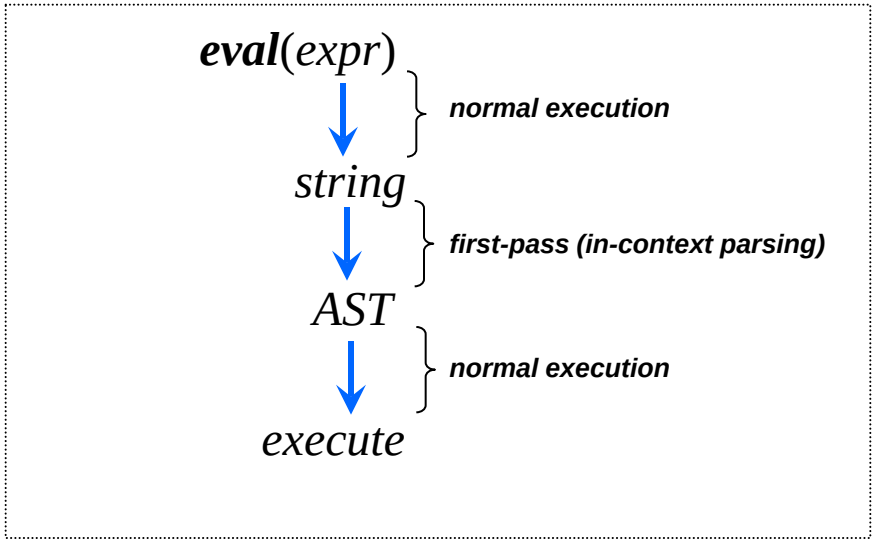


Project

Project (1/11)

- *Small interpreted meta language named* ***sin***
  - compile to AST and run
  - no *eval*() is required (*optional*)





Project (2/11)

- *Untyped object-based* with dynamically typed variables (like *JavaScript*)
- Objects *ex nihilo* as field dictionaries
  - Object construction expressions are optional
    - { *[*<***key***> <***value***> ***(,***<***key***> <***value***>***)**** *]* }
    - `a = { x 1, y 2, z f() };`
    - `a= {}; a.x = 1; a.y =2; a.z = f();`
  - If object construction expressions are not supported, you need basically the expression *{}* to imply "*make a new object*"

Project (3/11)

- Basic reference counting for objects
  - very primitive garbage collection not working for cyclic references
- All key types are allowed but are treated as strings just before hashing
  - `t[0]` ≡ `t["0"]`
  - `t[t]` ≡ `t["object(0x12)")`
    - assuming unique serial number for objects, else you may use the native object address or reference
  - Reserve a *`keys`* and *`size`* key that can't collide with user-defined keys
    - *`keys`* = object with all keys, *`size`* = number of fields
    - e.g. `a.#keys` and `a.#size`

Project (4/11)

- Simple syntax but always *of your choice*
  - *You may want to justify that*
  - Don't be biased or affected by the syntax of my previous or later examples
- Statement based
  - Notion of a global program as a series of statements with the following simplifications
    - Nested functions but no access to outer locals (only globals)
    - No closures
    - No lambda (anonymous functions) - *optional*
    - No true methods (only syntactic sugar)
      - `a->f(b)` ≡ `a.f(a, b)` assuming `->` means method invocation
    - No functors - *optional*
    - No function expressions - *optional*

Project (5/11)

- Function environment *fenv* is a hash table *Environment* (internal object) carrying
  - Arguments
  - Locals
- Global variables are stored in a global *genv* that is used for variables outside functions
- Inside functions *genv* and *fenv* are normally accessed
  - with every function call getting its reserved *fenv* copy
  - created at *call site (caller)* with the actual arguments and then expanded at the *callee site (called)* with the locals

Project (6/11)

- Extensible library functions programmed in native code
  - the language in which you make the ***sin***
  - *Environment* as parameter to library functions
    - to extract actual arguments
  - You need also a way to return a value from library functions
    - you can reserve a *retval* field in the *Environment* object
    - the environment is also to be destroyed at the *call site* after the invocation returns and the return value is (optionally) taken

Project (7/11)

- You will need to support native objects (pointers or references) as types in ***sin***
  - those are returned by library functions and they are created and managed exclusively by native code
- You will need this feature since
  - ASTs will be native objects
  - handled by the set of tree manipulation library functions you will create and install

Project (8/11)

- Basic metaprogramming features of ***sin***
  - **.<** *expr* **>.** *shift to meta level (expr → AST)*
  - .~ *var* *assume var already carries an AST*
  - .! *expr* *compile (execute) an AST (meta expression)*
- *Optional features*
  - .@ *expr* *compile a string (runtime) expr to an AST*
  - ***eval****(expr)* is .!.@*expr*
  - .# *expr* *unparse a meta expression (AST→text)*
- *There is no separation of metafunctions from normal functions*
  - *In principle, any program expression may be meta code*
  - *You can use .! to compile a manually produced AST via lib functions*

Project (9/11)

- Metaprograms you may develop for testing *(1/2)*
  - simple local optimizations
    - optimized versions of functions like power
    - loop unrolling for statically known number of iterations
  - adding diagnostics into functions
    - wrapping invocations to specific functions with *before* and *after* messages
  - aspectual transformations
    - simple *advice* for simple code patterns like adding *Design by Contract* calls in methods
      - in every function *f* with a *self* first argument add the following code:

```
precond = self.pref_f; if (precond) precond(self);
```

Project (10/11)

- Metaprograms you may develop for testing *(2/2)*
  - generating object factories
    - Meta functions accepting pairs of slot identifiers and initial values, producing a respective object factory function

```
st_factory = .!factory("name", "", "address", "", "semester", 1);
student = st_factory.new();
```

  - static function analysis
    - exit paths — *compile warnings*
    - simple dead code elimination — *if (false), return; <code>*
    - assignment in condition — *compile warnings*
  - static function style checker
    - function size in statements
    - expression complexity

Project (11/11)

- *Rules and terms*
  - Up to four persons
  - Any implementation language allowed
  - No report needed
  - Examination is a formal presentation
  - Examples from three categories at least required
  - Presentation date (*final*) is Wednesday 09/09/09
  - Plenary discussions with all teams biweekly possible