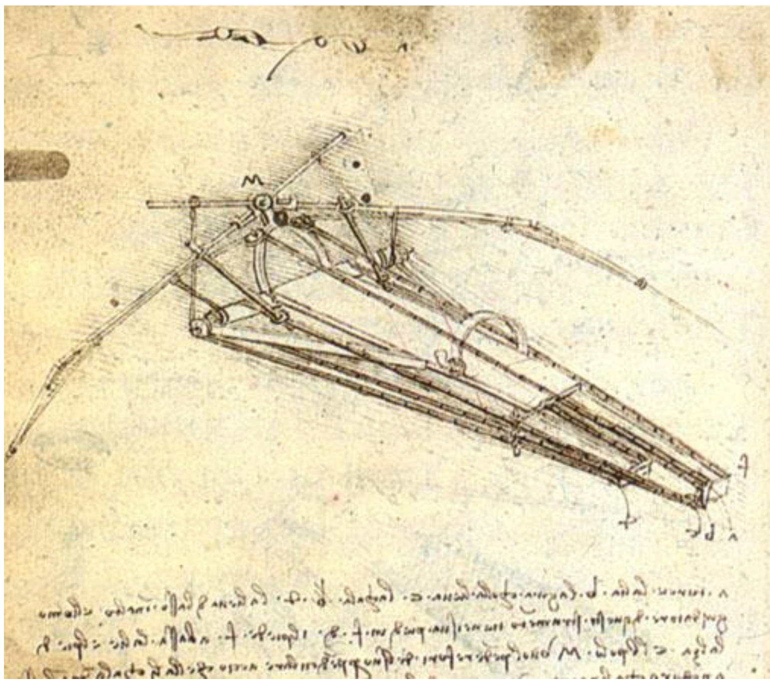
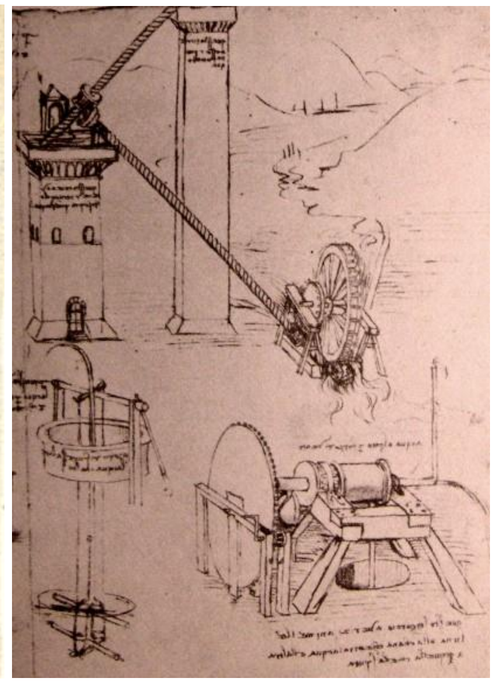
HY540 – Advanced Topics in Programming Language Development

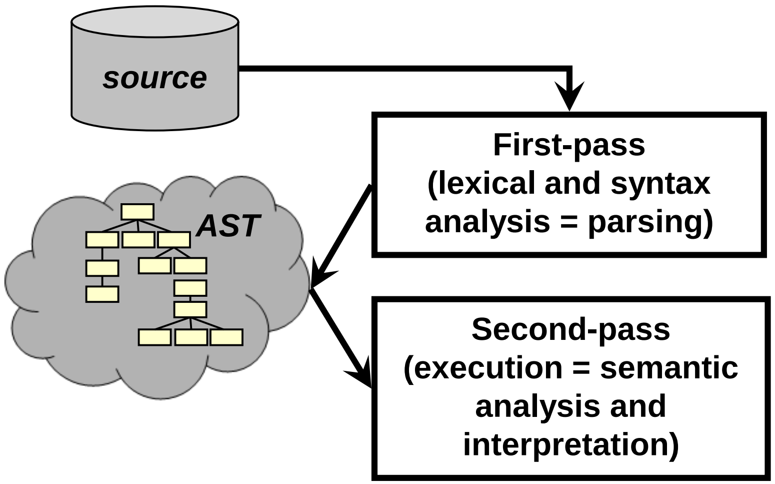
Untyped Language Interpreter

Untyped Language Interpreter Language

- *Untyped object-based* language with dynamically typed variables (like *JavaScript*)
- *Statement based*, i.e. program code is a series of statements
  - Typical statements: conditional, control flow, block
  - Function definitions as statements
  - Typical expressions: arithmetic, relational, boolean, assignments, function calls
  - Objects *ex nihilo* as field dictionaries
    - Created through object constructor expressions
    - Reference counted and subject to garbage collection

Untyped Language Interpreter
High level overview

- *Lexical analysis* – as usual (e.g. using lex)
- *Syntactic analysis* – as usual (e.g. using yacc)
  - Instead of the typical syntax directed translation we just build the program *Abstract Syntax Tree (AST)*
- *AST Interpretation*
  - AST traversal that:
    - Creates and maintains the execution environment
    - Introduces functions and variables to the environment respecting scoping rules
    - Executes program statements

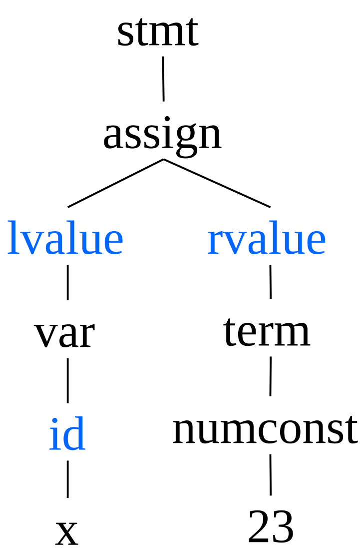


Untyped Language Interpreter Basic building blocks (1/2)

- *AST structure*
  - Generic AST node class supporting arbitrary children and custom properties
  - Optionally an AST visitor to support tree traversals
- *Environment*
  - Acts as a *symbol table*, keeping variable and function scope information
  - Acts as an *execution stack*, mapping variables to values
- Both can be implemented using the same dictionary-based class used for objects
  - Supporting an AST visitor for a dictionary-based object requires having ordered, numeric indices for children

Untyped Language Interpreter Basic building blocks (2/2)

- ASTs using objects




```
{ type : "stmt",
  child : <object> }

  { type : "assign",
    lvalue : <object>,      { type : "var",
    rvalue : <object> }       id : "x" }

     { type : "term",
       child : <object> }

        { type : "numconst",
          value : 23 }
```

- Environments using objects

```
{ x : <object>,     { type : "variable",
  f : <object> }      value: 23 }

                    { type : "program_func",
                      value: <object> }    AST Node object
```

Untyped Language Interpreter Environments (1/5)

- We have 3 kinds of environments
  - **Block environments**
    - Introduced by *blocks*
    - Hold variables and functions declared within the block
    - A block environment is also used for the global scope
  - **Function environments**
    - Introduced by *function calls*
    - Hold call actual arguments (values) mapped to formals
    - May also act as block environments, holding variables and functions declared within the function body
  - **Closure environments**
    - Introduced by *function definitions*
    - Hold a snapshot of the function execution environment (closure)

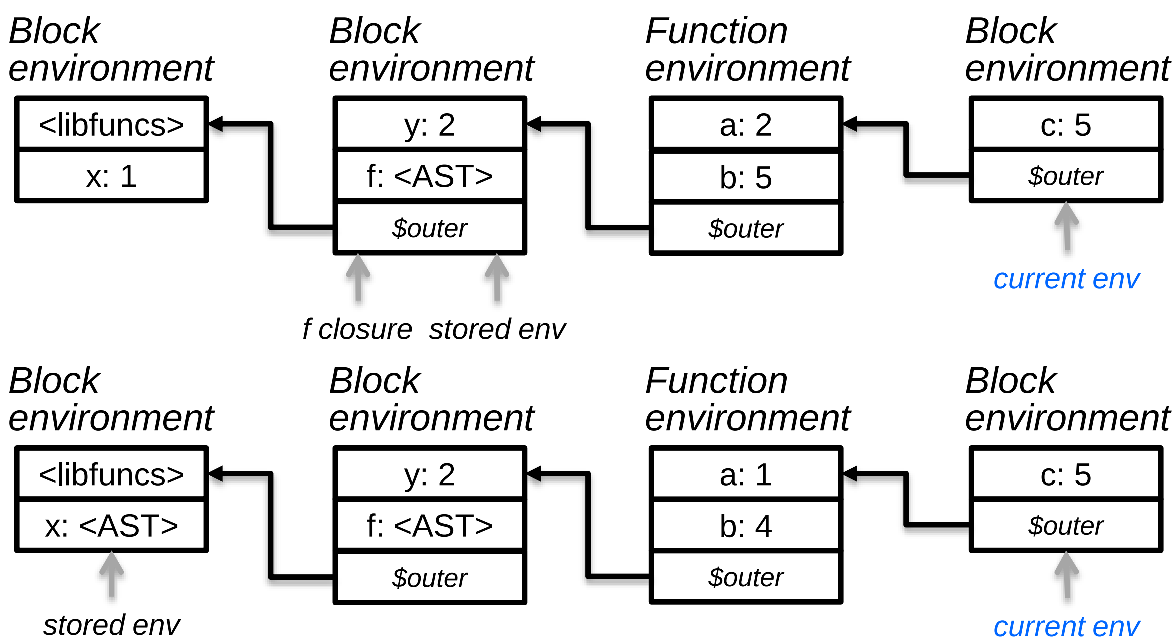
Untyped Language Interpreter Environments (2/5)

- Environments are linked in an *environment chain*
  - New environments are linked with previous ones through object references using a special index "$outer"
    - So as not to collide with any user symbols in the environment
  - *Do not explicitly destroy environments going out of scope*
    - They are objects, so just decrease their reference counter and let them be garbage collected when no longer needed
- Function calls create *new environment chains* that start with the function closure environment
  - We need to maintain a stack with environment chains
  - Environment creation and lookup is performed on the *top chain*
- For each chain, we only keep track of the innermost environment (for the top chain it is the current)
  - Other environments are accessible through the chain

Untyped Language Interpreter Environments (3/5)

- For lookup, we start from the current environment and navigate through enclosing environments using the $outer links
  - When function environments are involved, we are guarantied not to miss symbols as the closure environment chain already has access to any symbol visible within the function
    - The closure chain may include other function, block or closure environments

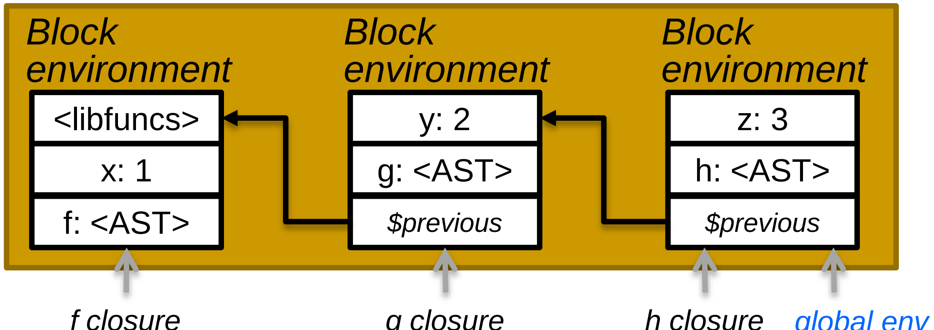
```
x = 1;
{
  y = 2;
  function f(a) {
    b = 3 + a;
    {
       local c = 4 + a;
    }
  }
  f(y); //1st example
  x = f;
}
x(1); //2nd example
```



Untyped Language Interpreter Environments (4/5)

- Function closures link to outer environments but should only have access to variables declared ***prior to*** the function definition (lexical scoping)
- To support this, block environments (global or nested) are *sliced* across function definitions
  - Slices are linked together using the special index "$previous“
  - Lookup within the slices of a block is performed through the **$previous** chain (***local lookup***)
  - Lookup across different blocks is performed through the **$outer** chain ***(normal lookup)***

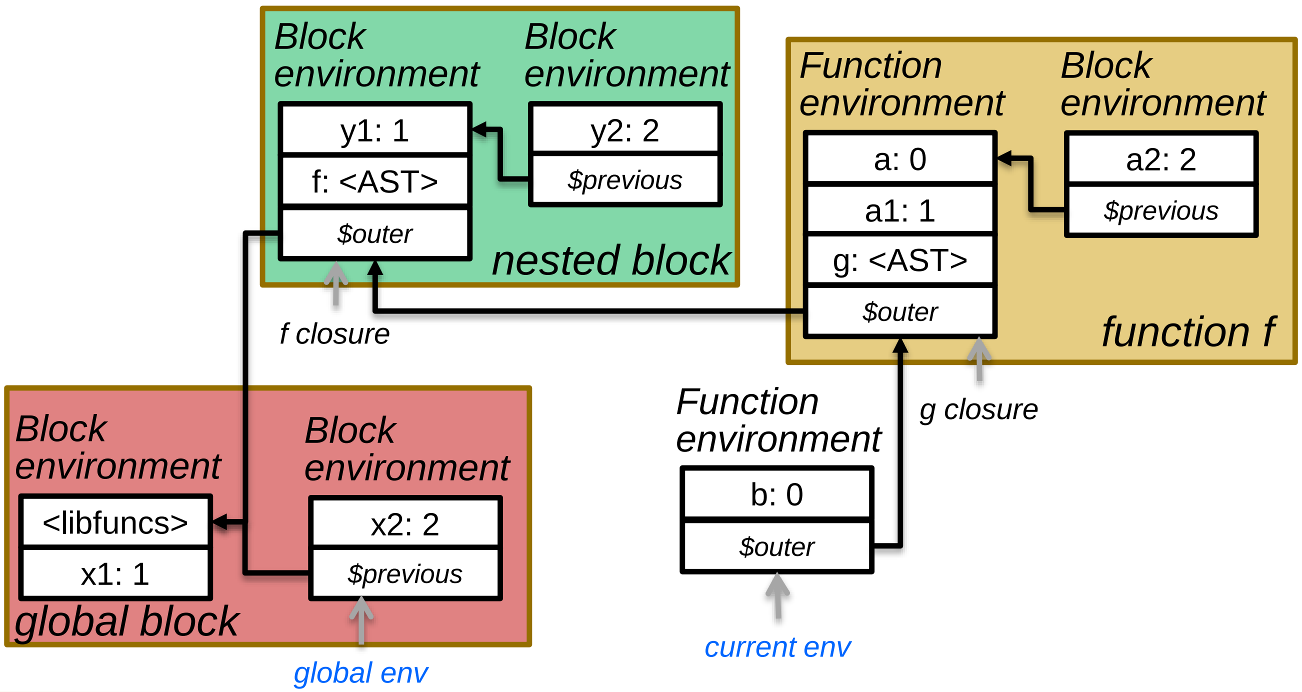
```
x = 1;
function f(){ return x; }
y = 2;
function g(){ return x + y; }
z = 3;
function h(){ return x + y + z; }
```



Untyped Language Interpreter Environments (5/5)

- Inner block slices also dictate slices in outer blocks
  - When popping a nested block, if it contained slices we *introduce a new slice for the outer block* (that will become the current)

```
x1 = 1;
{
  y1 = 1;
  function f(a){
    a1 = 1;
    function g(b){}
    a2 = 2;
    return g;
  }
  y2 = 2;
  x1 = f;
}
x2 = 2;
x1(0)(0);
```



Untyped Language Interpreter Closures (1/2)

- *Closures* support **lexically scoped name binding**
  - Used in languages with first class functions
  - Allow access to enclosing scope variables even when out of scope
- To support closures, we treat **function values as pairs** of a function address (AST) and a snapshot of their environment
- For the snapshot of the environment we simply use a reference to the current (most recent) environment
  - Allows sharing across execution system and function closures with minimal memory overhead
  - More importantly, it supports *write access to closure variables*
- The snapshot of the environment is taken at function definition, i.e. in the interpretation of the funcdef AST
- As an object, the environment is subject to garbage collection, so a closure will be automatically collected when not needed

Untyped Language Interpreter
Closures (2/2)

- With the discussed infrastructure, it is trivial to support *first-class closures*, or *glassbox closures*

```
function f(x) { return (function() { return x; }); }

f1 = f(1);
print(f1()); // prints 1

f1.x = 2;     // closure is a first class referenceable object
print(f1()); // prints 2
```

- We just need to extend the *object_get* and *object set* implementations to also accept functions and *lookup directly in their closure environment*
  - If lookup fails an error should be reported

Untyped Language Interpreter
Named and optional parameters (1/3)

- *Named parameters* allow specifying function call arguments associated with *the formal parameter name* rather than the position in the parameter list
  - e.g. *function rect(x, y, width, height) {…}*
  - *rect(10, 20, 30, 40); rect(x:10, y:20, width:30, height: 40);* same as *rect(width:30, height: 40, x:10, y:20);*
- *Optional parameters* allow omitting function call arguments for parameters having default values
  - Default values are typically constant
    - *function point(x = 0, y = 0) {…}*
  - But we can also have default values with arbitrary expressions evaluated using the current environment
    - *function randomize(seed = currenttime()) {…}*

Untyped Language Interpreter
Named and optional parameters (2/3)

- Named parameters requirements:
  - **Syntactic extensions** for function call
    - `argument: expression | IDENT ':' expression;`
  - Using an object dictionary (both numeric and string indices) instead of an array for the actuals arguments
    - If nil values cannot be stored in an object we need a special index, e.g. "$size" to count actual parameters
  - Extra error checking
    - Unexpected named parameters
    - Positional parameters appearing after named parameters
    - Parameters with both positional and named value

Untyped Language Interpreter
Named and optional parameters (3/3)

- Optional parameter requirements:
  - **Syntactic extensions** for function definition
    - `formal: IDENT | IDENT '=' expression;`
  - **Evaluating the default value expression** in the context of the callee during the insertion of formal argument symbols to the function environment to allow e.g. *function f(x, y = x + 1) {…}*
  - Extra error checking ensuring that all default arguments are at the end of the parameter list after any required parameters
    - This does not apply when combined with named parameters

Untyped Language Interpreter AST Interpretation (1/4)

- An ***Evaluate*** function is used for each AST node type:
  - Value EvaluateVar(AST ast);
  - Value EvaluateAddExpr(AST ast);
  - Value EvaluateIfStmt(AST ast);
- There is also a generic *Evaluate* function that uses a dispatcher to trigger the appropriate function based on the input AST node type
- *Evaluate* functions of non-leaf nodes recursively call *Evaluate* for their children and perform their operation
- No need to break long expressions to 3 address code
  - Temporary variables are implicitly created on the host language stack

```
Value EvaluateAddExpr(AST ast) {
  Value left = Evaluate(ast.Get("left"));
  Value right = Evaluate(ast.Get("right"));
  return left + right;
}
```

Untyped Language Interpreter AST Interpretation (2/4)

- No need to break control flow statements into more primitive instructions
  - We directly reuse the host language stmts

```
Value EvaluateIfStmt(AST ast) {
  Value cond = Evaluate(ast.Get("cond"));
  if (cond) Evaluate(ast.Get("stmt"));
  return null;
}
```

- Special attention is needed for *break*, *continue*, *return*
  - We use execution flags IsBreaking, IsContinuing, IsReturning that are set by the jump statements and handled respectively by their enclosing loop or function
    - The *return* statement may also optionally set the retval register
  - For multiple statements & blocks, the ***statement list*** evaluation checks the flags to transfer execution to the enclosing stmt
  - An alternative is to transfer control flow from one evaluation context to another using *exceptions*

Untyped Language Interpreter AST Interpretation (3/4)

- **Exception based control flow**
  - Jump statements throw exceptions
    - *BreakException*, *ContinueException*, *ReturnException* (maybe with value)
  - Interested clients catch exceptions and act accordingly
    - ForStmt / WhileStmt catch to repeat or terminate the loop
    - Blocks / Functions catch to cleanup the environment and rethrow
    - Function calls catch to continue past the return statement
    - Top level code catches to report any errors
  - *Be careful, as inconsistent code jumps will be difficult to debug*

```
Value EvaluateBreak(AST ast) {
  throw BreakException;
}

Value EvaluateContinue(AST ast) {
  throw ContinueException;
}
```

```
Value EvaluateWhileStmt(AST ast) {
  while(Evaluate(ast.Get("cond")))
    try { Evaluate(ast.Get("stmt")); }
    catch(BreakException e) { break; }
    catch(ContinueException e) { continue; }
  return null;
}
```

Untyped Language Interpreter
AST Interpretation (4/4)

- **Variable**
  - Lookup in the Environment chain
  - Declaration by use, so first use creates a symbol in the environment
  - Need to differentiate between lvalue and rvalue usage
    - Explicit *Symbol EvaluateLvalue(AST ast);* to be used in assignments
- **Function declaration**
  - Create function symbol in the environment associated with AST node
- **Function call (caller actions)**
  - Evaluate the arguments and put them into an arguments table
  - Invoke the target function (its value should be a function address)
- **Function invocation (callee actions)**
  - Maps the arguments table to the function formals
  - Creates the symbols in the function environment
  - Executes the function body
- The function call result is taken from the retval register

Untyped Language Interpreter
Summary of important evaluation actions

- Environment actions
  - push_slice() : push_scope(), link below via **$previous**
  - push_nested(): push_scope(), link below via **$outer**
  - **push_scope_space(), pop_scope_space()**
- START_PROGRAM
  - push_scope_space[empty]
- BLOCK_ENTER
  - push_nested()
- FUNC_DEF
  - add func_node in top_scope()
  - make function value *<func_node, top_scope()>*
  - push_slice()
- BLOCK_EXIT
  - slice_outer ← *top_scope() has* **$previous**
  - **while** *top_scope() has* **$previous do** pop_scope()
  - **assert** top_scope() has **$outer** pop_scope()
  - **if** slice_outer **then** push_slice()
- CALL
  - push_scope_space[*callee closure*]
  - push_scope() and compose actuals table (*SHADOWING ALLOWED*)
  - execute the call
- FUNC_ENTER
  - assign actuals
- RETURN_VAL
  - assign it to a retval register
- FUNC_EXIT
  - pop_scope() ←this is for actuals
  - pop_scope_space()

Untyped Language Interpreter
Library functions

- Extensible library functions programmed in *native code*
  - Receive user supplied arguments from the function call Environment
    - The environment is passed as a parameter to library functions
  - Optionally return a value by setting the retval register
    - The retval register may be part of the function environment or global
- Standard libfuncs
  - print, typeof, object_keys, object_size, **eval**

Untyped Language Interpreter eval (1/2)

- The *eval()* library function performs *in-place evaluation* of code provided in string form
  - Symbols within the code string are bound in the local execution context
- The code string is initially parsed to AST
  - This is a separate AST from the main program AST
- The AST is then executed
  - This is a **recursive invocation** of the interpreter
  - To simulate interpretation in the execution context of *eval*, we must manually pop the activation record of *eval* itself
    - And push it back after AST evaluation but before returning from eval for it to be normally popped from the execution stack

Untyped Language Interpreter eval (2/2)

- The code string of ***eval*** may contain an expression, a statement, or a list of statements
  - Thus parsing requires more than just the original parser
- Naive implementation: *use multiple distinct parsers*
  - *Bad approach due to code replication*
- *Better alternative*: extend the original parser
  - Add a new token (terminal) for each extra parse form
  - Add new productions from the start symbol to the extra parse forms with the matching tokens as prefixes
    - e.g. `start: stmts | PARSE_EXPR expr;`
  - Extend the lexer to first return any *user supplied artificial tokens* and then continue with typical lexical analysis
  - When initiating parsing, push the artificial token matching the target parse form to the lexer

Untyped Language Interpreter Metaprogramming (1/3)

- Basic metaprogramming features
  - .< *expr* >. — *shift to meta level (expr → AST)*
  - .~ *var* — *assume var already carries an AST*
  - .! *expr* — *compile (execute) an AST (meta expression)*
- *Additional features*
  - .@ *expr* — *compile a string (runtime) expr to an AST*
    - ***eval****(expr)* is similar to .!.@*expr*, but the latter replaces the code with the result once and for all, while eval will be reevaluated on reentry (e.g. loop, function)
  - .# *expr* — *unparse a meta expression (AST→ text)*
- *No separation between metafunctions and normal functions*
  - *In principle, any program expression may be meta code*
  - *You can use .! to compile a manually produced AST*

Untyped Language Interpreter Metaprogramming (2/3)

- 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);
```

Untyped Language Interpreter Metaprogramming (3/3)

- 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); or...
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

Untyped Language Interpreter Embedding (1/3)

- Language embedding is when we support interoperation between the target language (embedded) code and the native (host) code
  - e.g. C and Lua, C++ and Java (JNI), C++ and Delta
- Host ← Embedded
  - Library functions
- Host → Embedded
  - Invocation of function values
    - Typically by overloading the function call operator
  - Native hosted objects, e.g. Delta externids

Untyped Language Interpreter Embedding (2/3)

- Function invocations from host code can typically only pass direct function arguments
  - Any variables part of the function closure environment refer only to *embedded language code*, and cannot be directly mapped to host code
- We would like to transparently associate closure environment variable with host code variables
  - Embedded code executes normally
  - Closure variable assignments set and get operations directly operate on host code variables
  - *Advanced embedding support*
  - This requires extending the embedded language values

Untyped Language Interpreter Embedding (3/3)

- The disjoint union used for holding the embedded language values is extended with a field for **native value references**
  - This is another disjoint union with all possible native types
    - e.g. int *, double *, bool *
    - Also, a dedicated **void*** along with a type tag for arbitrary classes
  - Assignment and value retrieval for reference values is performed through the corresponding type
    - May involve appropriate type casting
    - May use explicit getter and setter functions supplied by the host code during construction of a reference value
- Host language code can access the function closure environment and set closure variables with reference values linking directly to native variables