Rincon Research Corporation

Dynamic, Recursive, Heterogeneous Types in Statically Typed Languages

Richard T. Saunders – Rincon Research Corporation
Dr. Clinton Jeffery – University of Idaho

Python dict

```
>>> v = "abc"
>>> v = 1     # dynamic values

# heterogeneous types in dict
>> d = { 'a':1, 'nest': {'b':3.14}}

# recursive, cascading lookup, insert
>>> print d['nest']['b']
>>> d['nest']['new'] = 17.6  # insert
```

Adding Key-Value Dicts to C++

- Most Dynamic Languages have a notion of a dictionary of key-value pairs
  - Python dict
  - Unicon/Icon table
  - Lua tables
  - Javascript objects
  - Ruby Hash (key-value store)
- The dict is really easy to use!

  Key-Value Store:
  - Associates a key with any kind of value

```
>>> d = { 'a': 1, 'b':2.2, 'c':'three' }
>>> print  d['a']  # key of 'a', value 1
1
```

- … no real equivalent in C++…

… using C++ features to make it easier

- We can add something like dict to C++
- Paradoxically: static features of C++ make dynamic features easier (??)
  - Function Overloading
  - Operator Overloading
  - User-Defined Conversions
  - Type Selection
  - Type Inference

Why Python dicts?

Goal: Make dynamic, recursive, heterogeneous, dictionaries as easy to use in C++ as Python

- Why?
  - Most major projects span multiple languages
    - Scripting languages (Python, JavaScript, Ruby) are the front-end, gluing together components
    - High-performance languages (FORTRAN, C/C++) form the hardcore backend
  - The front-end languages and the back-end languages need a common *currency* for communication: the Python dictionary

Outline

- Definitions
- History/Lessons Learned
- Val, Tab, Arr framework
  - Overloading
  - User-Defined Conversions
  - Cascading Insertion and Lookup
- Boost any type
- Conclusion

Definition: Dynamically Typed

- *Dynamically Typed Language*: The type of a variable is determined by the value in the variable at runtime
  - Python, Ruby, Lisp, Unicon are dynamically typed languages
- Python:

```
>>> a = 1         # a is a int
>>> a = 2.2       # Nope! Now it's a float
>>> a = "three" # Now it's a string
```

- The type is dynamic and bound at runtime

Rincon Research Corporation

Definition: Statically Typed

- *Statically Typed Language*: The type of a variable is bound at compile-time: that variable can only hold values of that type.
  - FORTRAN, C, C++, Java are statically-typed languages
- C++ Example:

```
int a = 1;
a = 2.2;
 // converts 2.2 to an int (or ERROR)
a = "three";
 // ERROR: a can only hold int values
```

Heterogeneous vs. Homogeneous

- Usually apply term to containers
  - A container is *heterogeneous* if it can hold more than one type
  - A container is *homogeneous* if it can only hold one type

- C++ containers are homogeneous:

```
vector<int> v{1,2,3};  // array type for ints only
map<string, int> m; // key-value, but string->int only
```

- Python containers are heterogeneous:

```
a = [1, 2.2, 'three']  # array type, can hold any type
d = { 'a':1, 'b':2.2, 'c':'three' }
          # keys and values can mix types
```

Rincon Research Corporation

Definition: Recursive

- A container is *recursive* if it can contain types of itself
  - i.e., containers contain containers

```
>>> d = { 'a': 1, 'b': 2.2, 'c': {'d':1 } }
```

- Extension of heterogeneity
  - How well does the language support nested types?

```
// Python: trivial
>>> print d['c']['a']     # Easy to access
1

// C++: // Uhh … ???
map<string, map<string, int> > m;
   // Only contains maps of map?  Not really …
```

History (or "How I Became Obsessed with Dynamic Types in C++")

- 1996: Worked on Midas 2k: A C++ framework for doing DSP
  - Technical success, political failure
  - I work with engineers: simplicity of interface matters
- One major success: OpalValues, OpalTables
  - Everyone wrote a list of things that should migrate from Midas 2k to new system
  - Number 1 on everyone's list: OpalValues/OpalTables

OpalValue:Dynamic Types version 1.0

- OpalValue: A *dynamic* container for holding any basic type, or tables
- OpalTables: a *recursive*, *heterogeneous* key-value container
  - OpalTable ot = "{a=1, b=2.2}";
    - keys are a, b
    - Values are 1, 2.2
    - ot.get("b") returns 2.2
  - **Keys** are strings
  - **Values** are OpalValues (*heterogeneous*), which can also be OpalTables (*recursive*)

OpalValue Successes

- Expressing Dynamic, Recursive, Heterogeneous Types in C++
  - New
  - Useful (on everyone's list as a feature to migrate)
- Both textual and binary expression
  - OpalTables could be saved to file in both binary (fast) and textual (human-readable) form

OpalValue Failure: Insertion of Values was simple, but not trivial

- OpalValue o1 = Opalize(string("hello"))
- OpalValue o2 = OpalTable(); // empty table
- OpalValue o3 = Number(real_8(1.0));

- Wasn't consistent, sometimes needed Opalize
- Using Opalize is wordy
- Number!

OpalValue Failure: Extraction of Values was Terrible

- Number n = UnOpalize(ov, Number);
- int i = n;

- string s = UnOpalize(ov, string);

OpalValue Lesson 1: Number was a Mistake ….

- Having a container class to contain numbers was a mistake: all extractions had to go through an extra level of Number

```
Number n = UnOpalize(ov, Number);
real_8 r = n;
```

OpalValue Lesson 2: Number was a Mistake … but it Taught us Something

Rincon Research Corporation

```
• Number n1 = 1;        // int
• Number n2 = 2.2;      // double
• Number n3 = 3.3f;     // float

•  int ii = n1;           // get out an int
•  real_8 rr = n3;        // get out double
```

OpalValue Failures: Textual Representation was Non-standard

- Syntax "stovepipe creation", i.e., non-standard
  - { a = { 1,2,3}, b = { c="hello"} }
  - Remember, this was the pre-JSON and pre-XML era

- Lists and tables had the same syntax with { }
  - {1,2.2,"three"} same as {0=1,1=2.2, 2="three"}

Rincon Research Corporation

Lessons

- Extraction and Insertion must be trivial
- An extra Number class is a mistake
- Use standard textual representation
- "Holistic" lesson: Be careful when overloading
  - Conversions interact in strange ways
  - Ambiguous overloads or conversions => compiler complains

Rincon Research Corporation

Dynamic Types 2.0: Python: Wow!

- Python got Dynamic, Recursive, Heterogeneous Types right!

Python

```
>>> v = "abc"
>>> v = 1     # dynamic values

# heterogeneous types in dict
>>> d = { 'a':1, 'nest': {'b':3.14}}

# recursive, cascading lookup, insert
>>> print d['nest']['b']
>>> d['nest']['new'] = 17.6  # insert
```

Dynamic Types 2.0: Lessons

- Lessons learned:
  - Use Python dictionary syntax as much as possible
    - People like it
    - Easy to use
    - In Python, modules, classes and most major namespaces are implemented as Python dictionaries
      - because of this ubiquity, the `dict` is fast and easy to use
    - Textual format is "standard"
      - JSON is a subset of Python dictionary (almost)
      - Python is widely used

Dynamic Types 3.0: Var

- Var is a wrapper in C++ for manipulating Python data structures
  - Embed a Python interpreter into your C++ program
  - Tried to make Python easier to express in C++
- Successes:
  - Var: a dynamic type
  - Cascading inserts, lookups easy to express
- Failures:
  - Extracting info too wordy
  - Python interpreter required
  - Cascading inserts, lookups used a proxy …

"Final Version": the Val/Tab/Arr

- Goal: Make dynamic, recursive, heterogeneous dictionaries as easy to use in C++ as Python
- Why?
  - Most major projects span multiple languages
    - Scripting languages (Python, Javascript, Ruby) are the front-end, gluing together components
    - High-performance languages (FORTRAN, C/C++) form the hardcore backend
  - The front-end languages and the back-end languages need a common currency for communication: the Python dictionary

- Those who fail to learn the lessons of history are doomed to repeat them

Python

```
>>> v = "abc"
>>> v = 1      # dynamic values

# heterogeneous types in dict
>> d = { 'a':1, 'nest': {'b':3.14}}

# recursive, cascading lookup, insert
>>> print d['nest']['b']
>>> d['nest']['new'] = 17.6  # insert
```

```
Val v = "abc";
v = 1;    // dynamic values

// heterogeneous types in Tab
Tab d = "{'a':1,'nest': {'b':3.1.4}}";

// recursive, cascading lookup, insert
cout << d["nest"]["b"] << endl;
d["nest"]["new"] = 17.6;
```

Rincon Research Corporation

Basics: the Val

- Every variable in C++ must have a static type: we will use `Val` as the type representing *dynamic* values.
- `Val` is a simple dynamic container:
  - Strings
  - Dictionaries (Tab) and lists (Arr)
  - Can contain any primitive type: `int_1, int_u1, int_2, int_u2, int_4, int_u4, real_4, real_8, complex_8, complex_16.`

Static Overloading on Constructor

- Chooses type based on value
- Makes Val construction easy:

```
Val a = 100;          // int
Val b = 3.141592;     // real_8
Val c = 3.1415f;      // real_4
Val d = "hello";      // string
Val e = None;         // empty
Val t = Tab();        // dictionary
```

Val Implementation

- Implemented as a type-tag and a union
  - That's so 1980s!
  - Reasons:

(1) Union is fast and space-efficient

(2) Union is also thread and heap friendly

- avoid unnecessary heap allocation: minor lesson from M2k

(3) Intentional lack of virtual functions or pointers to functions means you can use the Val in cross-process shared memory

(4) Yes, use placement `new` and manual destructors

Overloading Constructor: Issue

- Has to be overloaded on *all* primitive types, or compiler complains
  - If you forget real_8, what does `Val v = 1.0` do?

```
Class Val {
    public:
     // Constructors on Val overloaded on all primitive types
     Val (int_u1 a) : …
     Val (int_1 a) : …
     Val (int_u2 a) : …
     Val (int_4 a) : …
     Val (int_u8 a) : …
     Val (int_8 a) : …
     Val (real_4 a) : …
     Val (real_8 a) : …
     Val (const string& s) : …
```

Why not use Templatized Constructor?

- Answer:

(1) We don't control it as well, and we have to control all primitive type conversions to avoid compiler ambiguities

(2) Some backwards compatibility issues:

users back at RedHat 3 and 4!

Overloading on Platform Dependent Types

- Result of many STL operations is a `size_t.` What is a `size_t`? Answer: Some unsigned int. Depends.
- May or may not be same as `int_u8` or `int_u4`. May be platform defined int
  - more likely, GNU quantity: like int, but considered a different type by C++ type system.
- On some platforms, will be a `int_u8/int_u4`; on others, not.

Val and `size_t` Interactions

- Want Val to work well with size_t:

```
Val v=sizeof(Blach);
```

- But above will NOT work on platforms where size_t is not an int_u4 or int_u8. We can work around it:

```
Val v=int_u8(sizeof(Blach));
```

- But this subverts the "simplicity" for the users

Old days: `#ifdef`

- In old C days, we would add a `#ifdef` and add a new constructor for machines where `size_t` is a new type:

```
class Val {
#ifdef SIZE_T_NOT_INT_U8
   Val(size_t) : …
#endif
}
```

Problem: manually check if `size_t` is available or not, have to manage macros

Rincon Research Corporation

New Days: Type Selection

- Use type selection technique from *Modern C++ Design*
  - Introduce a new dummy type called `OC_UNUSED_SIZE_T`
  - Introduce a new constructor `Val(ALLOW_SIZE_T)`

  - If the compiler notices that `size_t` is a unique int type
    - `ALLOW_SIZE_T` becomes typedeffed to `size_t`
  - else `size_t` is NOT a unique int (i.e., it is an `int_u4`), then
    - `ALLOW_SIZE_T` is typedeffed to `OC_UNUSED_SIZE_T`

Type Selection (Idea)

```
class OC_UNUSED_SIZE_T { };
template <class T> struct FindSizeT {
    typedef size_t Result;
};
template <> struct FindSizeT<int_u4> {
   typedef OC_UNUSED_SIZE_T Result;
}
typedef FindSizeT<size_t>::Result ALLOW_SIZE_T;

Class Val {
    Val (ALLOW_SIZE_T a) : …
    // all other overloads …
}
```

Rincon Research Corporation

By the way … also overload operator=

```
Val V = 1;          // constructor
V = 2.2;            // operator=
V = "three";        // operator=
V = None;
V = Tab();
```

User-Defined Conversion

- C++ has a unique feature called *user-defined conversions* which allow a type to export itself as a different type.

```
class IntRange {           // restricted to 0..99

    operator int () {…}    // allow IntRange
                           // to be used as int
};
int f(int i); // prototype for f:
              //f only takes an int argument

IntRange m;
f(m);    // ERROR?? No!! IntRange is allowed
         // to export itself as an int
```

Syntactic Sugar

```
IntRange m;
f(m);

// Above form is syntactic sugar for:

IntRange m;
int _outcasted_temp_ = m.operator int();
f(_outcasted_temp_);  // Legal C++!
```

User-Defined Conversions with Val

- Allows us to extract all types from Val with minimal typing. Val has user-defined conversions for all basic types as well as Tabs, Arrs and strings:

```
Val v = 3.141592;
double d = v;          // syntactic sugar

// same as
Val v = 3.141592;
double d = v.operator double();
```

Type of the variable INFORMS the conversion so you don't have to state explicitly which conversion is being used!

Val type and conversion mismatch?

- What if type in Val and outcast mismatch?

```
Val v = 3.141592;
int i = v;    // What happens?
```

- *Principle of Least Surprise*:
  - Do what C++ would do if you explicitly cast.
  - If not allowed, throw an exception (like a dynamic language would)

```
int i = static_cast<int>(3.141592); // cast to 3
```

Other Mismatched Conversions

```
Val v = 3.141;

float f = v;          // As C++:f=float(3.141);
int i = v;            // As C++:i=int(3.141);
Tab t = v;
   // NOT a table, throw exception!
```

Val Implementation

- Like Val constructor, the outcasts have to overload on all primitive types (and strings, Tab, Arr) or will run into massive compiler warnings:

```
class Val {
  operator int_u1();
  operator int_1();
  operator int_u2();
  operator int_4();
  operator int_u4();
  operator int_8();
  operator int_u8();
  operator ALLOW_SIZE_T();  // size_t different?
  operator real_4 ();
  operator real_8();
  …
};
```

Val Implementation (code)

Archaic implementation:
but we control all conversions

```
operator int_4 () {  // tag tells which union field
    switch (tag) {
      case 's': return int_4(u.s);  // int_1 union field
      case 'S': return int_4(u.S);  // int_u1 union field
      case 'i': return int_4(u.i);  // int_2 field
      case 'I': return int_4(u.I);  // int_u2 field
      case 'l': return int_4(u.l);  // int_4 field
      case 'L': return int_4(u.L);  // int_u4 field
      case 'x': return int_4(u.x);  // int_8 field
      case 'X': return int_4(u.X);  // int_u8 field
      case 'f': return int_4(u.f);  // real_4 field
      case 'd': return int_4(u.d);  // real_8 field
      …
}
```

Limitations

```
Val v = 1;
string s = v;    // Works: operator string() on Val

s = v;           // FAILS! Overloaded operator=
```

- Problem: STL string has its own operator= and user-defined outcast interferes (confuses compiler)
  - All these signatures interfere with each other

```
string& operator=(const string& str);
string& operator=(const char* s);
string& operator=(char c);
Val::operator string();// Which one to use?
```

Workaround

```
string s;
Val v = 1;
s = string(v);
  // forces user-defined conversion
```

It breaks the idea that the variable chooses the right user-defined conversion, but at least it's simple and not too much more typing

Idea is useful in longer code snippets:

```
Val freq, bw = … something …
real_8 lim = real_8(freq) + real_8(bw) * 2.0 + 1000;
```

Summary of Val: Easy to Put Values in and Take Values Out

```
// C++
Val v = "hello";  // Val is dynamic container
v = 17;
v = 1.0;          // easy to put values in

float d = v;     //  easy to take values out
string s = d;    //  easy way to stringize

# Python
v = 'hello'       # Python
v = 17
v = 1.0
d = float(v)
s = str(d)
```

Composite Type: Tab

- Like Python dict
- `class Tab : public OCAVLHashT<Val, Val, 8> { }`
  - AVL Tree where keys at nodes are hashed values
    - No hash information lost to modulo operations
    - So integer compares to find place in tree
    - Values at nodes are (key, value) pairs
      - Still have to keep key in case of hash collisions
  - Bounded hash table with no rehashing necessary
    - M2k was a soft realtime system, incremental data structures
- A `Tab` is like an incremental hash table of key-values pairs
  - Lookups happen by keys
    - Strings or ints (`Val`)
  - Values are any `Val` (including nested `Tab`s!)

Rincon Research Corporation

Composite Type: Arr

- Like Python list
  - (Python list is really implemented as a resizing array)
- `Arr` is an array of Val
  - Array is an OpenContainers concept (picklingtools.com)
  - Array has been optimized to fit into 32 bytes
    - (so can do placement new into Val)
  - Backwards compatibility (before STL was ubiquitous)

Dictionary Literal

- Python:

```
>>> d = { 'a':1, 'b':2.2,
          'c':[1,2.2, 'three'] }
```

- C++:

```
Tab t = "{'a':1, 'b':2.2, 'c':[1,2.2, 'three'] }";
```

Use string literal (since C++ doesn't support Python syntax)

Can cut-and-paste dictionaries between Python and C++ AS-IS

Note: single quote strings of Python makes string literal much easier to type in C++:

```
Tab t = "{\"a\":1, \"b\":2.2,
          \"c\":[1,2.2,\"three\"]}";
```

Dictionary Literal

- Easy to express large dictionary in C++:

```
Tab t = "{"
        "  'a':1, "
        "  'b': 2.2, "
        "  'c':[1, 2.2, 'three']"
        "}";
```

(String continuation across lines makes this work)

Python: Gold Standard

- Python:

```
>>> d = {'a':1, 'b':2.2, 'c':[1,2.2, 'three']}
>>> print d['a']          # lookup
1
>>> d['c'] = 555;         # insert
>>> print d
{ 'a':1, 'b':2.2, 'c':555}
```

C++ Lookup

- Val overloads operator[] and returns Val& and can be used in both insertion and lookup contexts. What user types:

```
Tab d = "{'a':1, 'b':2.2, 'c':[1,2.2,'three']}";
cerr << d["a"];  // lookup, note double quotes
```

- Lots of extra work happening for this to look nice: this is equivalent to the following (legal!) C++:

```
Tab t = "{'a':1, 'b':2.2, 'c':[1,2.2,'three']}";
Val _key_ = "a"; // Create a Val for the key
Val& _valref_ = d.operator[](_key_);
operator<<(cerr, _valref_);
```

C++ Insertion

- Using the Val&, can insert directly into a table

```
d["c"] = 555;  // what user types
```

- Long form (what C++ does for us … legal C++!):

```
Val _key_ = "c";
Val& _valref_ = d.operator[](_key_);
    // Not there:creates Val inside d
Val _newthing_ = 555;
_valref_ = _newthing_;
```

Insertion vs. Lookup

- Can't distinguish between lookup and insertion in C++ via constness (Meyers, "More Effective C++", Item 30)
- Overload both [] and ():
  - Both do same thing: return (some) reference to a Val&
  - EXCEPT: if key not there!
    - [] creates a new (empty) Val and returns reference to it
    - () throws an exception

```
cerr << t("not there");      // throws exception
t["not there"] = 100;        // allows insertion
```

Rincon Research Corporation

Python Gold Standard:Nested Insertion and Lookup

```
>>> d={'a':1, 'b':2.2, 'c':[1,2.2,'three']}
>>> print d['c'][1]        #  nested lookup
>>> d['c'][0] = 'one';   # nested insertion
```

C++ Nested Lookup

- User C++ Code:

```
Tab d="{'a':1, 'b':2.2, 'c':[1,2.2,'three']}";
cout << d("c")(1); // nested lookup
```

- This translates to:

```
Tab d="{'a':1, 'b':2.2, 'c':[1,2.2,'three']}";
Val _key1_ = "c";
const Val& _subc_=d.operator()(_key1_);
Val _key2_ = 1;
const Val& _subc1_ = _subc_.operator()(_key2_);
operator<<(cout, _subc1_);
```

C++ Nested Insert

- User C++ code:

```
d["c"][0] = "one";  // nested insert
```

- What's happening behind the scenes:

```
Val _key1 = "c";
Val& _subc_ = d.operator[](_key1);
Val _key2 = 0;
Val&_subc0_=_subc_.operator[](_key2);
_subc0_ = "one";
```

Putting It All Together

```
// C++
Tab d="{'a':1,'b':2.2,'c':[1,2.2,'three']}";
int v = d("c")(0);
v += 3;
d["c"][2] = v;
```

```
# Python
d = {'a':1, 'b':2.2, 'c':[1,2.2,'three']}
v = int(d['c'][0])
v+=3
d['c'][2] = v
```

Speed

- How does C++ dynamic Val compare to other dynamic languages?
  - No current benchmark comparing dictionaries of other languages (perfect for "Programming Language Shootout")
  - We compare C Python vs. C++ Val
    - C Python very stable, hand optimized over 10s of years

Pickle Test Suites

- Pickle: How fast can we iterate over a complex Table and extract dynamic information?
  - Python C version: raw C code extracting dynamic info and iterating over Python dicts at the speed of C
  - C++ Val version: raw C++ code extracting dynamic info and iterating over Tabs at the speed of C++
- UnPickle: How fast can we create dynamic objects and insert into tables?
  - Python C version: raw C unpickling and creating Python objects
  - C++ Val version: raw C++ unpickling and creating Vals

- Table is about 10000 keys of varying types of keys and lengths
  - Relatively shallow table (but a few nested dicts)

Speed Tests

	PicklingTools 1.3.1 C++ Val Object	C Python Version 2.7 PyObject
PickleText Pickle Protocol 0 Pickle Protocol 2	5.90 seconds 12.23 seconds 1.30 seconds	4.82 seconds 12.65 seconds 3.41 seconds
Unpickle Text Unpickle Protocol 0 Unpickle Protocol 2	23.40 seconds 7.24 seconds 4.34 seconds	38.19 seconds 7.13 seconds 3.66 seconds

Speed Tests Results

- Roughly comparable
  - C++ Val faster at pickling:
    - Much faster at iterating over complex table
  - Python C PyObject faster at unpickling
    - C Python does an optimization to cache recently used PyObjects (which speeds up caching, at the cost of thread neutrality)
      - Python GIL enables this optimization, not an option for Val
- This test tells us that C++ dynamic Val is on par with the Cpython's dynamic PyObjects

User-Defined Types

- Drawback: *Val can't hold arbitrary datatypes*
  - Only Tab, Arr, string, and primitive types
- Rather than force Val to try to adapt to other types, let other types become Vals!
  - Similar policy to XML: all types can be expressed as a composite of primitive types, string, and composite tables/lists:i.e., some combo of Vals
- SO! To work with user-defined types, make your class…
  - Construct from Val
  - Export to Val

User-Defined Types with Vals

```
class MyType {
  // Construct a MyType directly from a Val
  MyType (const Val& v)  // import from Val

  // Create a Val from MyType
  operator Val()         // export to a Val
};
```

Related Work

JSON: JavaScript Object Notation

representing dicts and lists in all languages

XML:

many people use for key-value dicts, lists

Environments have massive tools for handling XML

(netbeans, Eclipse)

…If only key-values were easier to deal with in statically typed languages

`Val` vs. Boost `any`

- Boost has `any` type
  - More general than `Val`, as it can hold any type
  - Suffers from clumsier interface because it is more general
- Val has been designed to look like dynamic languages

Cascades with Val

- Cascading inserts/lookups with Val are simple:

```
Tab t = "{'a':{'nest':1}}";
cout << t["a"]["nest"] << endl;
t["a"]["nest"] = 17;
```

Cascades with any (1)

- Much more complex, with many more casts

```
// Boost any approach: no literals,
// create table explicitly
map<string, any> t;
map<string, any> subtable;
subtable["nest"] = 1;
t["a"] = subtable;
```

Rincon Research Corporation

Cascades with any (2)

```
// Cascade lookup
any& inner = t["a"];
map<string, any>& inner_table =
  any_cast<map<string,any>&>(inner);
int r = any_cast<int>(inner_table["nest"]);
cout << r << endl;
```

Cascades with any (3)

```
// Cascade insert
any& inneri = t["a"];
map<string, any>& inneri_table =
    any_cast<map<string, any>& >(inneri);
any& nest = inner_table["nest"];
nest = 17;
```

Conclusions

- Work done to support Python dictionaries in C++:
  - All work available at http://www.picklingtools.com
  - Open Source (BSD license)
- Allows using dictionaries in both C++ and Python
  - Information can flow between front-end scripting languages and back-end optimization languages
  - Dictionary becomes currency of system