Dynamically-typed Language

- Everything is a value
- No type declarations
- Examples of dynamically-typed languages
  - APL, Io, JavaScript, Lisp, Lua, Objective-C, Perl, PHP, Python, Ruby, Scheme, Smalltalk
## Dynamic vs. Static

<table>
<thead>
<tr>
<th>Static</th>
<th>Dynamic</th>
</tr>
</thead>
<tbody>
<tr>
<td>▪ Types are associated with variables</td>
<td>▪ Types are associated with runtime values</td>
</tr>
<tr>
<td>▪ Each variable is of only one type during its lifetime</td>
<td>▪ Variables may point to values of differing types during their lifetime</td>
</tr>
<tr>
<td>▪ Variable types declared or inferred</td>
<td>▪ Often no variable declaration</td>
</tr>
<tr>
<td>▪ Type checking done at compile time</td>
<td>▪ Type checking often done at run-time</td>
</tr>
</tbody>
</table>
## Dynamic vs. Static (In Practice)

<table>
<thead>
<tr>
<th>Static</th>
<th>Dynamic</th>
</tr>
</thead>
<tbody>
<tr>
<td>▪ Most often occurs in compiled languages</td>
<td>▪ Most often occurs in interpreted languages</td>
</tr>
<tr>
<td>▪ Well suited for class-based OO programming</td>
<td>▪ Well suited for prototype-based OO programming</td>
</tr>
<tr>
<td>▪ Usually results in faster compiled code, but often takes longer to compile</td>
<td>▪ May allow compilers and interpreters to run faster, but produces slower (compiled) code</td>
</tr>
<tr>
<td>▪ Easier to optimize</td>
<td>▪ More difficult to optimize</td>
</tr>
<tr>
<td>▪ Code generally more verbose</td>
<td>▪ Code generally more succinct</td>
</tr>
<tr>
<td>▪ Meta-programming more cumbersome to write</td>
<td>▪ Well suited for meta-programming</td>
</tr>
</tbody>
</table>
Dynamic vs. Static (In Practice)

Double Layered Hash Table:

C# 1.0

```csharp
Hashtable ht = new Hashtable();
ht["something"] = new Hashtable();
((Hashtable)ht["something"])["someObj"] = new SomeObj();
((SomeObj)((Hashtable)ht["something"])["someObj"]).SomeMethod();
```

Python

```python
ht = {}
ht["something"] = {}
ht["something"]["someObj"] = SomeObj()
ht["something"]["someObj"].SomeMethod();
```
Type Inference

- Optional type declaration
  - Allow variables to be declared so as to enable optimization of the (compiled) code

- Standard type inference
  - Use standard type inferencing techniques to determine static type information

- Type prediction
  - Use usage patterns to “guess” what type a variable (an object?) is based on what operation it is used in
Python

- Supports structured and prototype based object oriented programming
- Uses Duck Typing
  - No type checking at compile time. Instead, an operation on an object fails at runtime if it does not support that operation
- Strongly typed
PyPy – Python Compiler

- Compiler analyses “live” programs
  - Programs are read into the Python interpreter and initialized
- Flow Object Space used to construct the control flow graphs by running through each possible control path of the code and recording operations performed on abstract objects
- Flow graphs are in Static Single Information (SSI) form
  - Extension of Static Single Assignment form in which each variable is used in only one basic block
  - All variables that are not dead at the end of a block are explicitly carried over and renamed
- Resulting flow graphs are passed to the annotator that performs type inferencing
PyPy’s Type Annotator

- Assigns annotations to each variable in the control flow graph
  - Annotations describe the possible run-time objects that a variable can contain
- Flows annotations forward
  - Type of a variable in Python can only be deduced by how it is produced, not by how it is used
  - Starting from an entry function, with user-specified annotations for it’s arguments, annotations are flowed through the blocks following calls recursively
  - Used fixed point algorithm in situation of loops – if previously annotated variables are too restrictive, generalize them and process the loop again
PyPy’s Type Annotator

Lattice of annotations
PyPy’s Type Annotator

Lattice of annotations
Self

- Prototype based Object Oriented language
- Implements full messages passing
- Objects consist of named slots, each of which contain a reference to another object
- Objects with source code associated with them are called methods
- When a message is sent to an object (called a “receiver”) the slots of the object (and recursively, parents of the object) are search for a match
  - If found, its content is evaluated and returned as the result of the message send
- Primitive operations have succeed and fail cases
  - Flow of control normally rejoins after the result is computed
Self Code Example

Self code:

```plaintext
sumTo: upperBound = ( 
  | sum <- 0 | 
  to: upperBound Do: [ 
    | :index | 
    sum: sum + index ]. 
  sum )
```

C code:

```c
int sumTo(int self, int upperBound) {
  int sum = 0;
  int index;
  for (index = self; index <= upperbound; index++)
    sum = sum + index;
  return sum;
}
```
SELF Compiler

- Extracts static type information
- Compiles several copies of a procedure, each customized for a specific receiver type
- Splits calls after a join, placing a copy on each control path optimized for a particular receiver type
- Predicts types that are likely but unknown by static type inference and inserts run-time tests to verify predictions
- Implements other standard optimization techniques
  - compile-time message lookup, aggressive procedure inlining, etc. …
SELF Compiler – Customized Compilation

- Provides type information for any calls to self (assuming no dynamic inheritance)
- Compiles copies of a method customized by characteristics of the calling site
- All subsequent calls to the method sharing the same characteristic, call the copy optimized for that calling site
SELF Compiler – Message Splitting

- Provides type information for the successful results of primitive operations
- When control paths merge with different result types for each path, a subsequent message can be “split”
  - The messages send is “push” up past the merge point
  - Each copy of the message send can then be further optimized
SELF Compiler – Type Prediction

- Provides type information for some messages not covered by customized compilations or message splitting
- Certain messages are more likely to be sent to some types of receivers than others
- From benchmarking measurements, types of receivers can be predicted
  - eg. 90% of the time operators +, -, and < have integer arguments
- A run-time type check is inserted and the message is split with a copy compiled on each branch
  - The “success” message is inlined while the “fail” message remains a full message send