Naturally

Programming in natural language

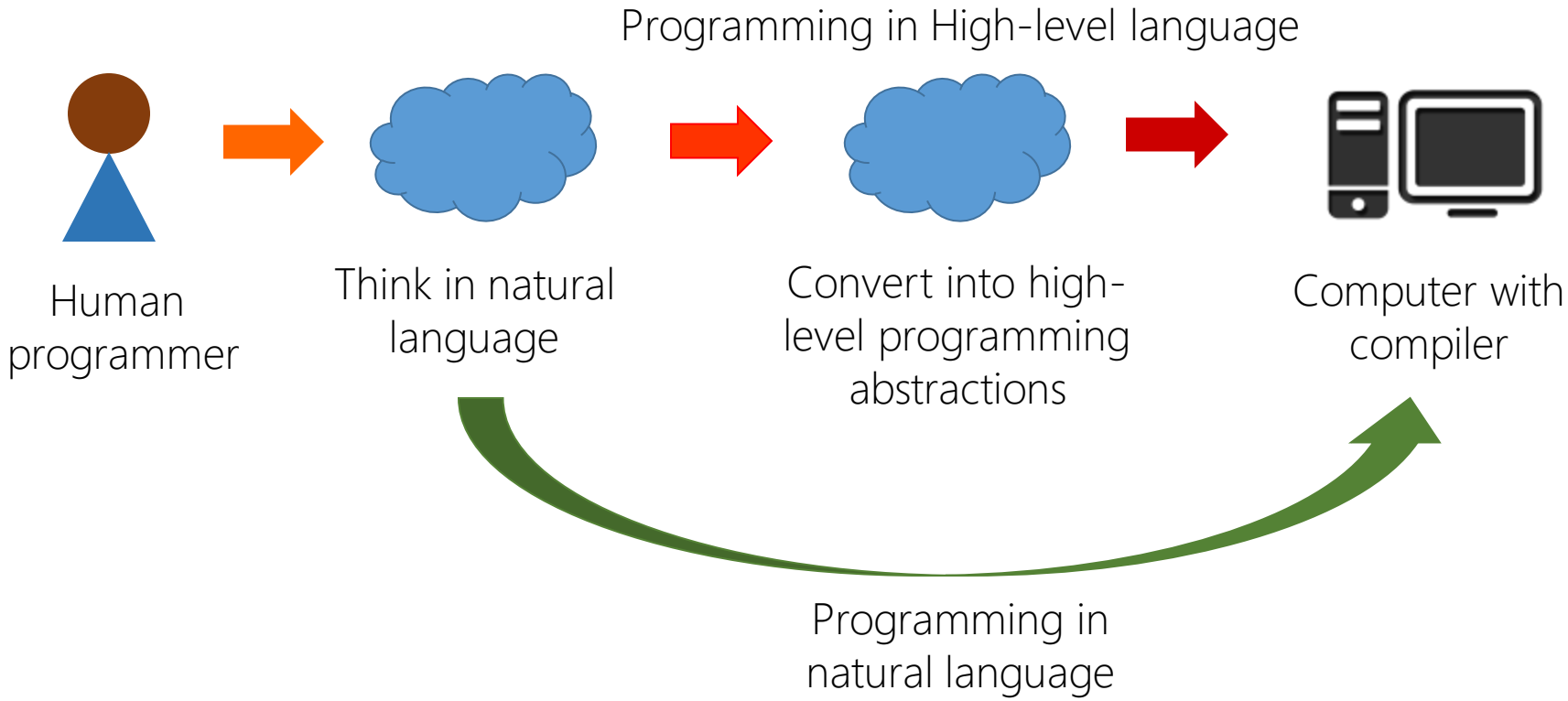
Kushagra Mehta
2013A7PS006H

What is Naturally?

- Naturally attempts to fill the void of natural language programming by expanding the subset of natural (English) language understood by a machine.
- Existing natural language programming languages are
  - Inform 7 – For creating interactive fiction
  - Shakespeare – An esoteric language whose syntax is loosely based on Shakespearean plays
  - Wolfram Alpha and Mathematica – For querying into a knowledge base and general mathematical functionality
- We don't have a natural programming language to meet general purpose programming needs. Hence, Naturally.

Why program in natural language?

- Cut one step short in programming abstraction.



- Lowered barrier into programming, faster scripting, self-documenting code etc. – benefits are numerous!

Concepts of Natural Language Programming

- The smallest executable unit is a sentence.
- Every sentence necessarily must compile unambiguously into a high level language procedure call or expression.

- The challenge of natural language programming is weighing two factors,

At the end of the day, programming languages must involve debugging logic, not ambiguity.

How does Naturally solve this?

While designing Naturally, we first thought about common operations done using a programming language

- Declaring variables, defining functions, invoking functions, defining classes, instantiating objects, evaluating expressions, conditional and iterative flow of logic.
- i.e we do a lot of small atomic, orthogonal operations which can be combined in different ways to produce an infinite number of programs.

Therefore, we identify common natural language constructs of specifying these operations and produce the syntax for the language.

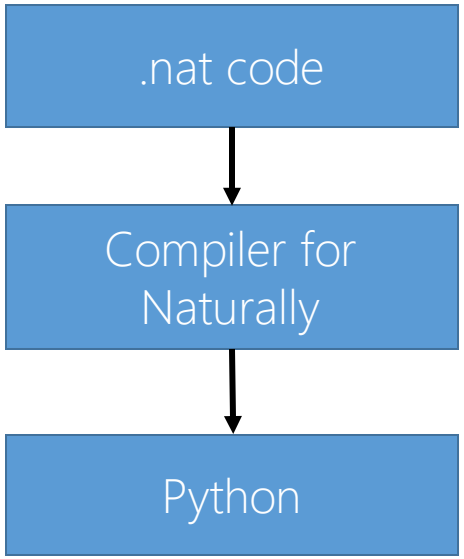
Further, we must allow for expanding syntax to include more natural ways of specification without introducing significant ambiguity.

Solution – synonyms for existing constructs.

Implementation

Naturally can be defined as a meta-language or a programming language which produces output in another language. This is similar to lex/yacc.

Naturally converts English structures defined by its syntax and converts it into Python code for execution. The conversion is done by simple translation into atomic Python statements.



Written in Naturally

Written in Python

- The Naturally compiler implements a simple lexer, parser and compiler interface which performs simple textual substitutions and limited top-down parsing.
- It relies on Python for type system, garbage collection and runtime execution.

Data Types

Naturally builds from orthogonal data types which can be used to build very complex data structures.

Naturally does not support classes/user-defined types as of now.

Primitives

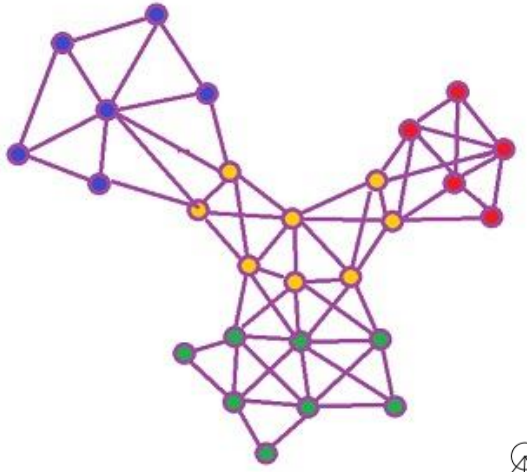
- integer
- float
- boolean
- string - string of length 1 is considered a character

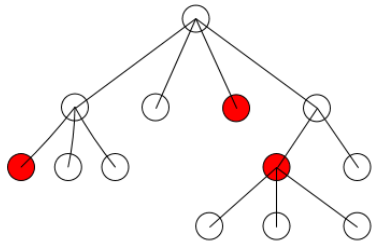
Complex

- list - dynamic array of primitives or complex data types, not necessarily same type
- dictionary - collection of key: value pairs with efficient retrieval, creation and removal

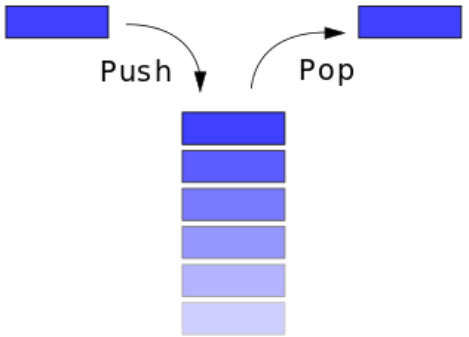
Data Types

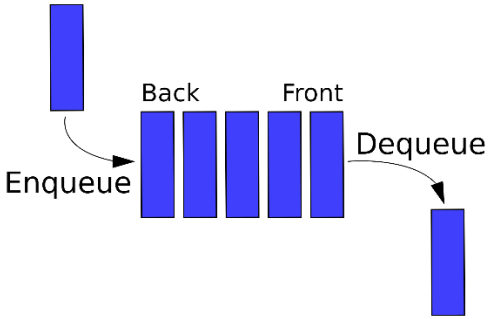
Most data structures can be easily built using the provided data types. Classes can be implemented using a dictionary.



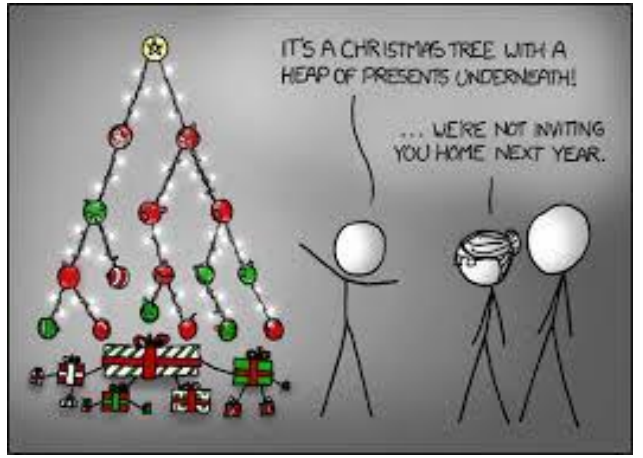


Graphs, trees – Implemented with a dictionary with node as key, and a list of adjacent nodes as value





Stacks, queues, heaps – Implemented easily using a list



Data Type System

Naturally uses a type system characterised by:

- Strong typing – The type of a value does not suddenly change without an explicit conversion or reassignment.
- Dynamic typing – Runtime objects have a type as opposed to static typing where variables have a type

Examples

```
set c to 5
read b
print b plus c          // Will throw error at runtime
                        // read takes input into b as a string, adding to an integer
                            cannot be done without a cast
```

Variable declarations

Variable declarations may be done in a number of ways:

- let <identifier> be <value/expression>
- set <identifier> to <value/expression>
- define <identifier> as <value/expression>
- declare <identifier> as <value/expression>

This showcases synonymous constructs for the same operation, to support different writing styles.

Scope of a variable is global if not declared under any indent level, else is local to its current indent level and immediate lower levels.

Flow constructs - conditional

Naturally provides for the four major programming flows

- sequential, conditional, iterative, recursive

Conditional flow is represented through the following block,

```
if <booleanexpression>,
        <statements>
else if/elif <booleanexpression>,
        <statements>
else,
        <statements>
```

- Multiple statements may be under the scope of an if/elif/else
- All statements which fall under the scope of an if/elif/else are indented by atleast one tab from it.
- else if and elif are synonymous constructs.

Flow constructs - iterative

Naturally provides for the four major programming flows

- sequential, conditional, iterative, recursive

Iterative flow is represented through the following block,

while <booleanexpression>,
    <statements>

- Multiple statements may be under the scope of a while block
- Statements which fall in the scope of the while must be indented by atleast one tab from it.

Block/function/global scope is thus implemented through a hierarchy of indentations.

Functions

Functions are first class citizens in Naturally – functions may be treated like a value and can even be passed around.

Functions are defined in the following manner,

define/declare function <functionname> (as),
    <functionbody>

We may also pass arguments to a function as,
define/declare function <functionname> (as) with arguments <formalargumentlist>,
    <functionbody>

Statements in the scope of a function are indented by atleast one tab from it.

Functions

Functions are invoked in the following manner:

call <functionname>

call <functionname> with <actualargumentlist>

value from <functionname>

value from <functionname> with <actualargumentlist>

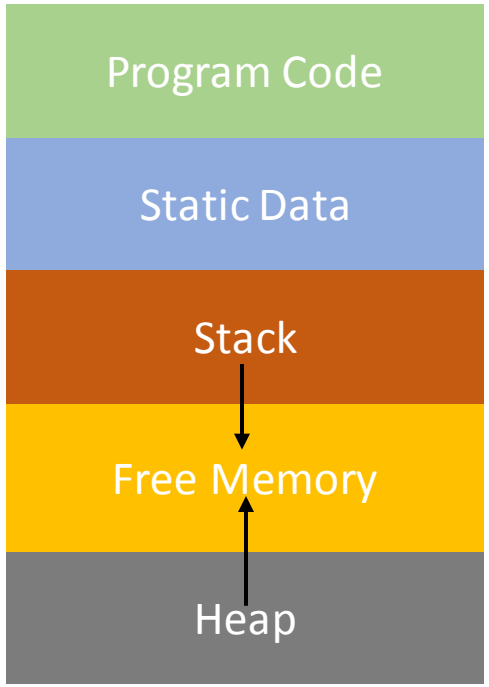
The ‘value from’ construct is meant to be used when the return value from a function is to be used in an expression.

Return statements in a function are as:
return <valuelist>

Multiple return values are therefore possible.

Parameter passing and return values

The CPython implementation of Python relies on the memory organization used by C for its programs.



// Global variables and constants

// Function activation records – local variables, parameters, return values

// Dynamically allocated memory – used for lists, dictionaries

Parameter passing and return values

- Parameters are passed through the activation record of the functions. Activation records are stored on the stack and each record has space for arguments and local variables.

  Calling a function -> push activation record on stack

  Returning from function -> pop activation record off stack

- Although we say multiple return values can be passed from the callee to caller, it is actually just an address of a tuple. A tuple is an immutable list in Python.
- Tuple packing and unpacking mechanism is used for return value passing and receiving.

I/O

Naturally provides a very simple I/O interface.

Input can be taken using,

input/read <variable> // <variable> is assigned user input with string type

Output can be printed using,

print <variablelist>

Memory Management

Naturally relies on the memory management and garbage collection features provided by the underlying CPython interface.

Naturally's memory allocation and deallocation is therefore automatic, there is no need for preallocation or deallocation.

The memory management system has two components:

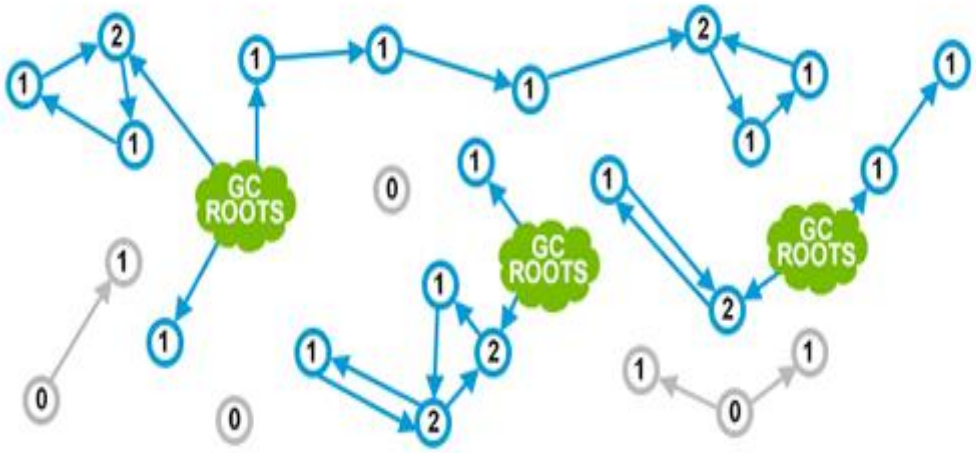
- Reference counting invoked with every reference or dereference
- Garbage collector invoked periodically to remove reference cycles, subject to a threshold

At every period, if the number of allocations minus the number of deallocations crosses the threshold, the garbage collector is invoked.

Memory Management

Reference Count

All Naturally data types and structures are an underlying Python object, each with a reference_count field.

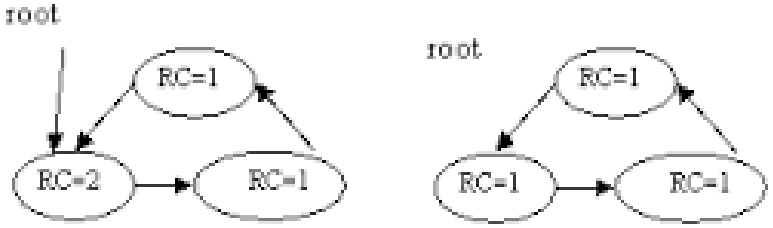


reference_count is updated if

- creation of new reference
- overwriting of reference
- assignment
- recycled

If reference_count falls to 0, it implies the object is no longer required, and can be recycled.

Reference counting suffers from not able to reclaim objects connected in a loop. Lists and dictionaries can create such loops.



Memory Management

Garbage Collection

- Garbage collection in Python is similar to that provided by JVM i.e. generational collection. It uses three generations.

- Traditional mark and sweep, stop and copy methods of garbage collection fail to work due to implementation of extensions in Python. There is a risk of freeing objects still referenced from somewhere.

- The GC first tries to find the set of unreachable objects.
  This is safer since the algorithm fails, it will be as good as normal GC.
  No risk of deleting referenced objects.

Since only containers (lists, dictionaries) can create reference cycles, we optimize our search by maintaining a doubly linked list of containers.

Memory Management

Garbage Collection

- Cost of doing this is 12 bytes of extra storage per container object.
- gc_refs is set to reference count initially.
- For each container object, find the referenced objects, and decrease their gc_refs.
- All container objects with gc_refs>1 are referenced from outside, cannot be freed.
- Any objects referenced by the above objects also cannot be freed.
- The rest of the objects form a cycle and can be freed.

THANK YOU