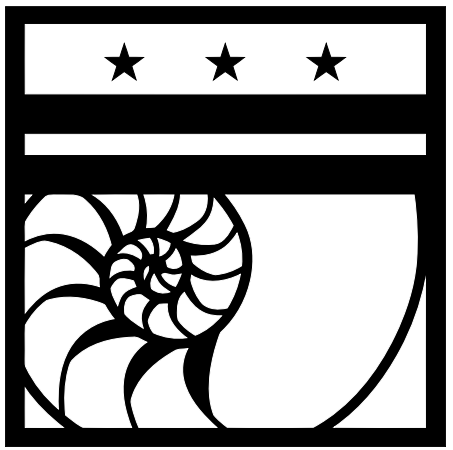
Your very own GP Interpreter

Natural language and Artificial Intelligence Group



Genetic Algorithms

- Uses principals of Darwinian evolution to breed good individuals
- Mutates individuals to create randomness in the population
- Over time these individuals tend toward the best
- Hopefully the GA will find a sufficient solution to the given problem

Remember Two Weeks Ago

- We talked about GAs
- We used GAs to evolve the coefficients of a parabola
- We used a data sample to figure out the error of the individual

Goal Parabola

$$3.14159 \cdot X + 1.61803 \cdot X^2$$

GA Individuals

$$3.14159 \cdot X + 1.61803 \cdot X^2$$

Genetic Programming

- Uses the same principals of Darwinian evolution as Genetic Algorithms
- Evolves **programs** not parameters
- This can include control structures such as loops, conditionals and even recursion (if you do it right)

Genetic Programming

- Individuals consist of functions, literals and variables
- Mutations change functions into different functions and literals and variables into different literals or variables
- Crossover copies part of a program into it's child

GP Individual

$$3.14159 \cdot X + 1.61803 \cdot X^2$$

Remember One Week Ago

- Todd evolved programs to create digital imagery
- He cited a nice paper by Karl Sims on "Artificial Evolution for Computer Graphics"
- Todd and Karl used trees to represent programs

Tree Based GP

- Tree GP is the oldest tradition of Genetic Programming
- Started by John Koza in 1985
- Used Lisp S-Expressions to represent a program

Why Lisp?

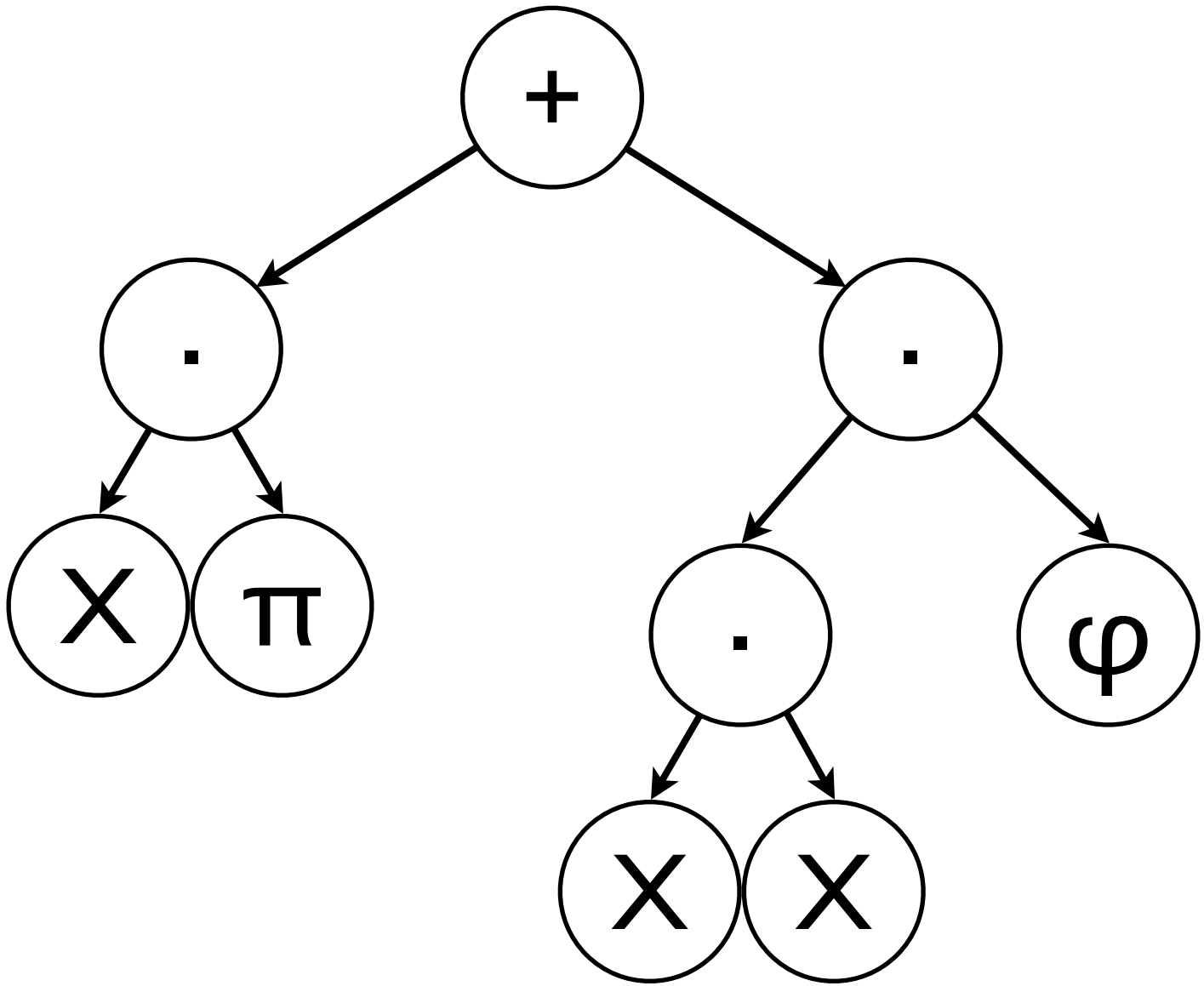
Lisp is *almost* always syntactically valid

- This allowed for the generation of random expressions with little worry of syntax errors
- S-Expressions can be manipulated as lists making programming mutation and crossover easier

GP Tree Individual

$$(+\ (\cdot\ \pi\ X)\ (\cdot\ \varphi\ (\cdot\ X\ X)))$$

Tree Parabola



Error Cases

- Even with the almost always syntactically validness of the S-Expression representation there are still error cases

Error Case Compensation

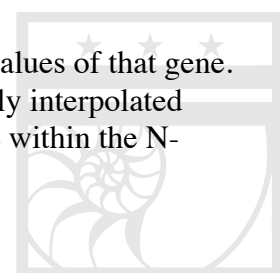
1. Any node can mutate into a new random expression. This allows for large changes, and usually results in a fairly significant alteration of the phenotype.
2. If the node is a scalar value, it can be adjusted by the addition of some random amount.
3. If the node is a vector, it can be adjusted by adding random amounts to each element.
4. If the node is a function, it can mutate into a different function. For example *(abs X)* might become *(cos X)*. If this mutation occurs, the arguments of the function are also adjusted if necessary to the correct number and types.
5. An expression can become the argument to a new random function. Other arguments are generated at random if necessary. For example *X* might become *(* X .3)*.
6. An argument to a function can jump out and become the new value for that node. For example *(* X .3)* might become *X*. This is the inverse of the previous type of mutation.
7. Finally, a node can become a copy of another node from the parent expression. For example *(+ (abs X) (* Y .6))* might become *(+ (abs (* Y .6)) (* Y .6))*. This causes effects similar to those caused by mating an expression with itself. It allows for sub-expressions to duplicate themselves within the overall expression.

Crossovers can be performed by sequentially copying genes from one parent, but with some frequency the source genotype is switched to the other parent. This causes adjacent genes to be more likely to stick together than genes at opposite ends of the sequence. Each pair of genes has a *linkage* probability depending on their distance from each other.

Each gene can be independently copied from one parent or the other with equal probability. If the parent genes each correspond to a point in N-dimensional genetic space, then the genes of the possible children using this method correspond to the 2^N corners of the N-dimensional rectangular solid connecting the two parent points. This method is the most commonly used in this work and is demonstrated in figure 2. Two parent plant structures are shown in the upper left boxes, and the remaining forms are their children.

Each gene can receive a random percentage, p, of one parent's genes, and a 1 - p percentage of the other parent's genes. If the percentage is the same for each gene, linear *interpolation* between the parent genotypes results, and the children will fall randomly on the line between the N-dimensional points of the parents. If evenly spaced samples along this line were generated, a *genetic dissolve* could be made that would cause a smooth transition between the parent phenotypes if the changing parameters had continuous effects on the phenotypes. This is an example of utilizing the underlying genetic representation for specific manipulation of the results. Interpolation could also be performed with three parents to create children that fall on a triangular region of a plane in the N-dimensional genetic space.

Finally, each new gene can receive a random value between the two parent values of that gene. This is like the interpolation scheme above, except each gene is independently interpolated between the parent genes. This method results in possible children anywhere within the N-dimensional rectangular solid connecting the parent points.



Error Cases

- Most error cases stem from two problems
  - Arity - Function specify how many arguments they take
  - Data Types - Functions specify what data types they take
- Crossover and Mutation must be aware of the error cases

Our Goal

Make an interpreter for a language that is *always* syntactically valid

Time to bust out your laptop

Goals

- Make functions not picky about arity
- Make functions not picky about data types

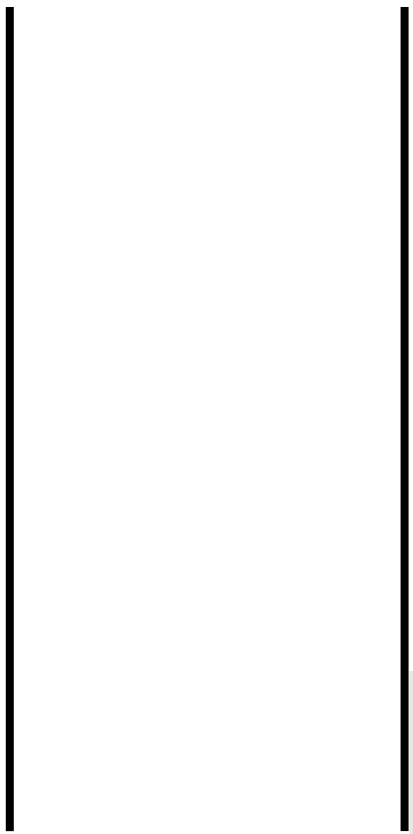
Our Representation

- Based on the PushGP programming language
- Stack based representation
- Reverse Polish Notation to represent programs

(Think Fourth on crack)

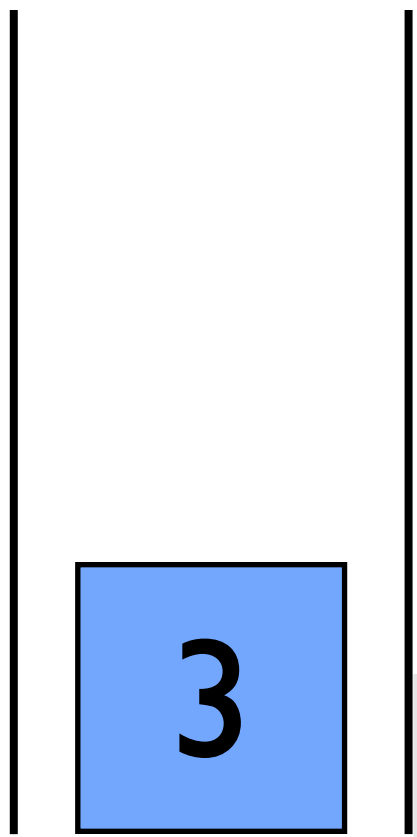
Reverse Polish Notation

3 4 +



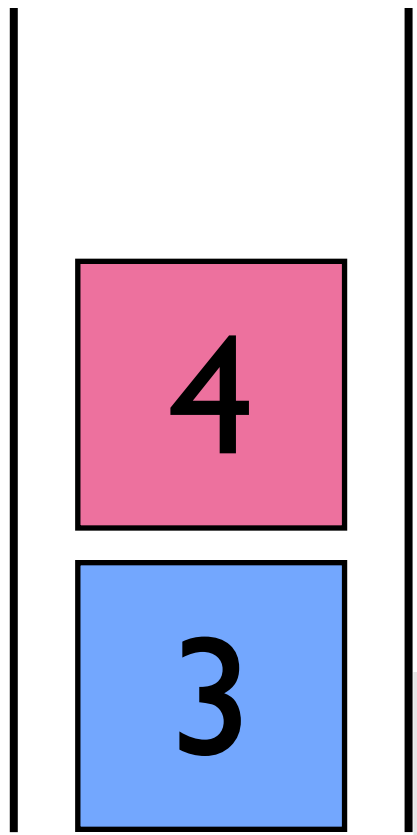
Reverse Polish Notation

3 4 +



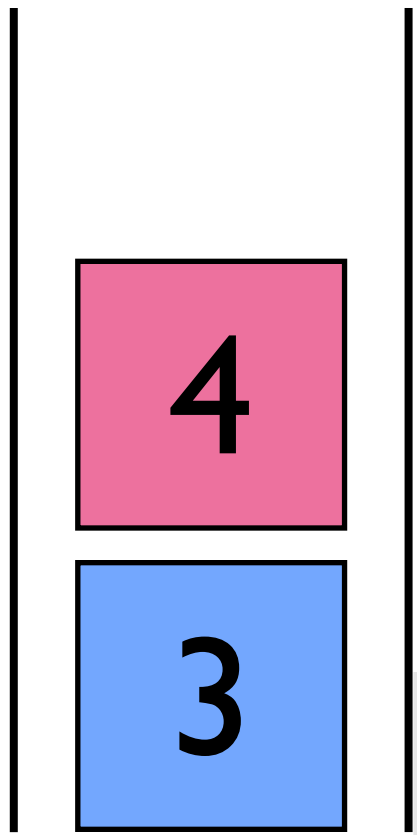
Reverse Polish Notation

4 +

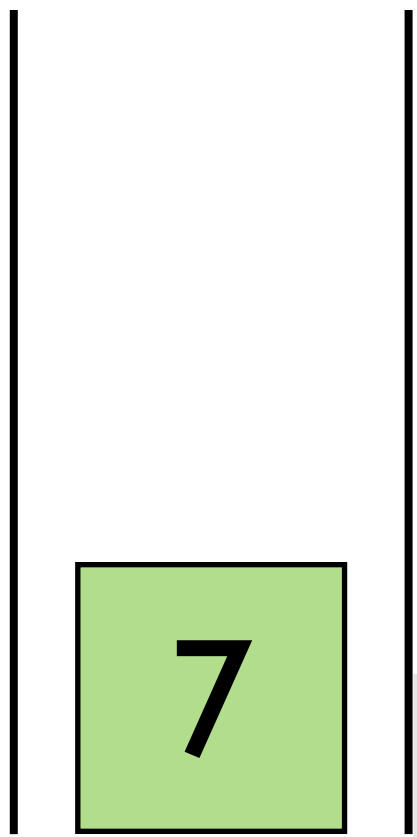


Reverse Polish Notation

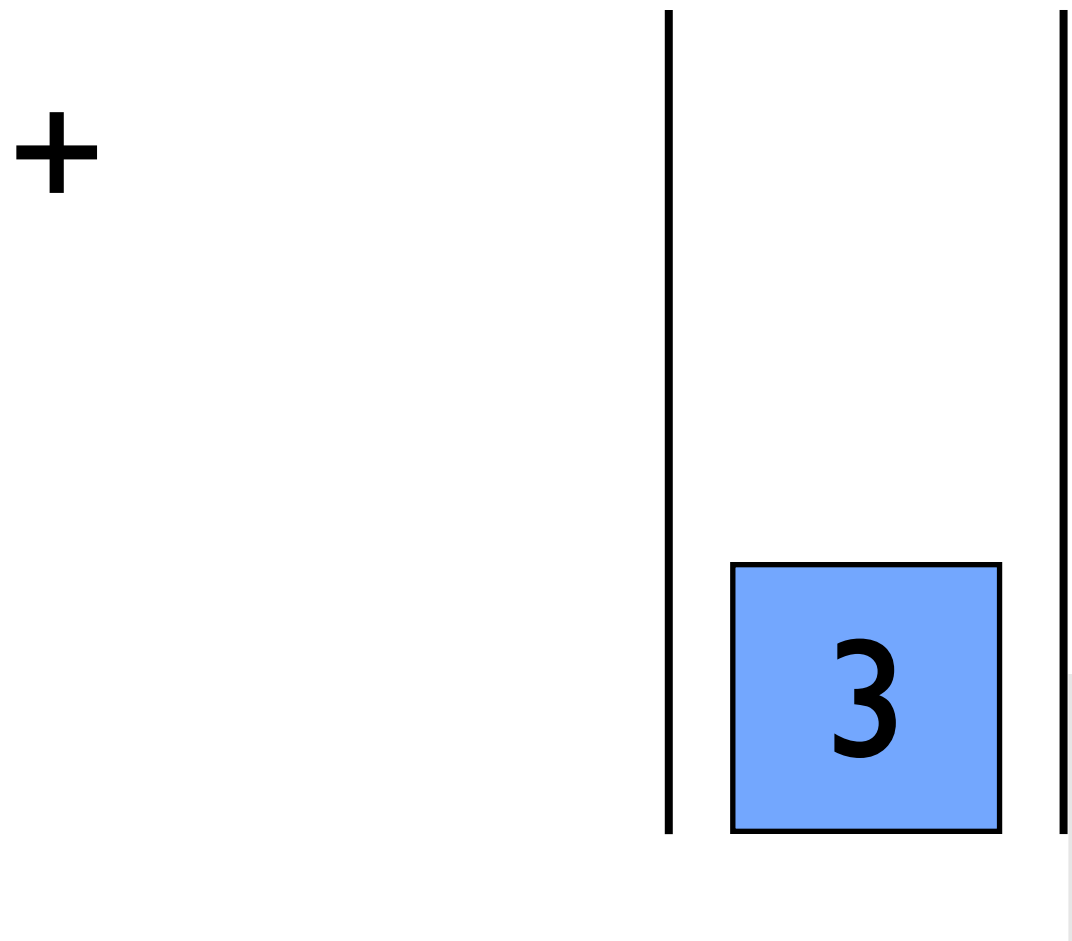
+



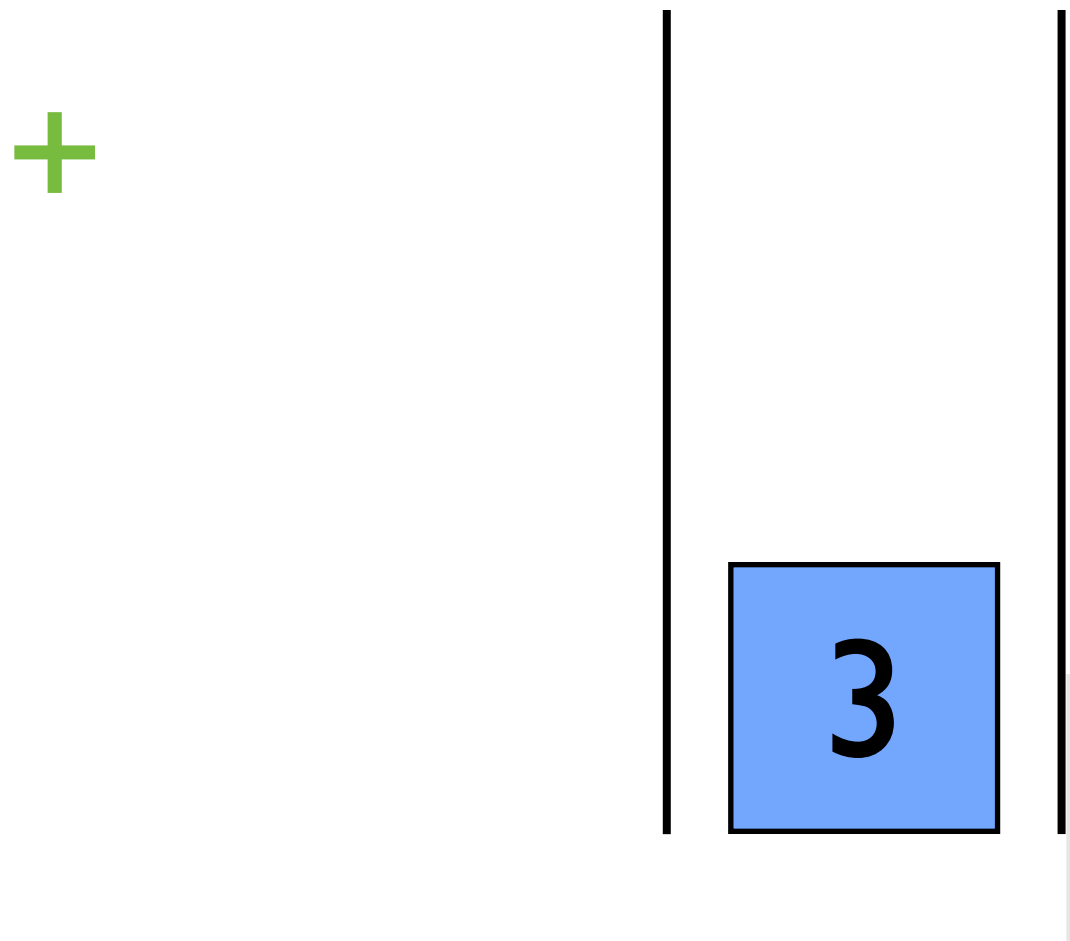
Reverse Polish Notation



Arity Error

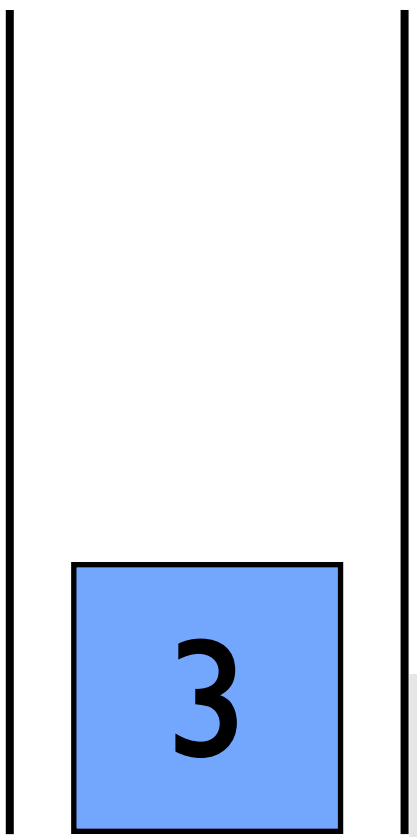


Arity Error

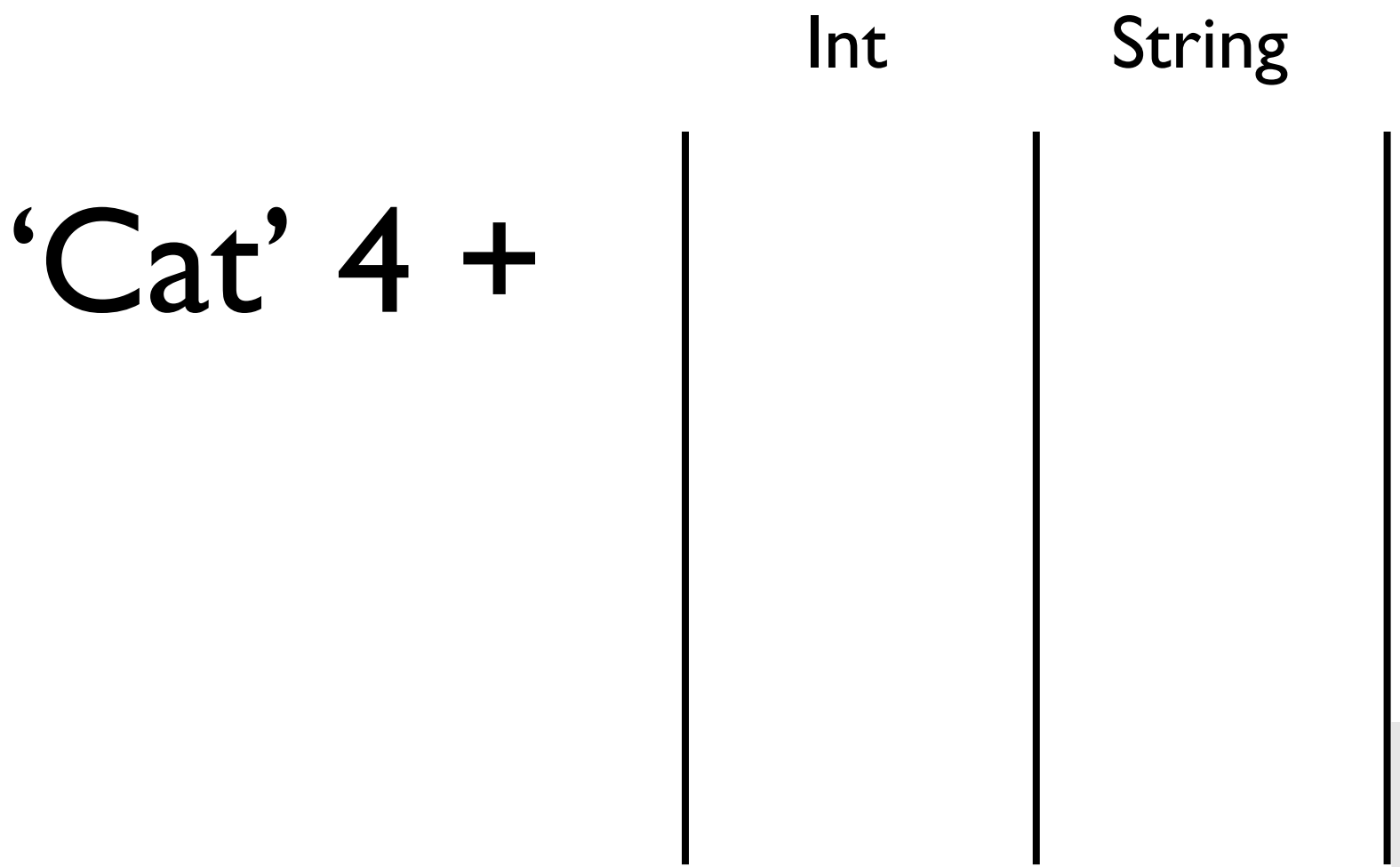


Arity Error

It NOOPd.

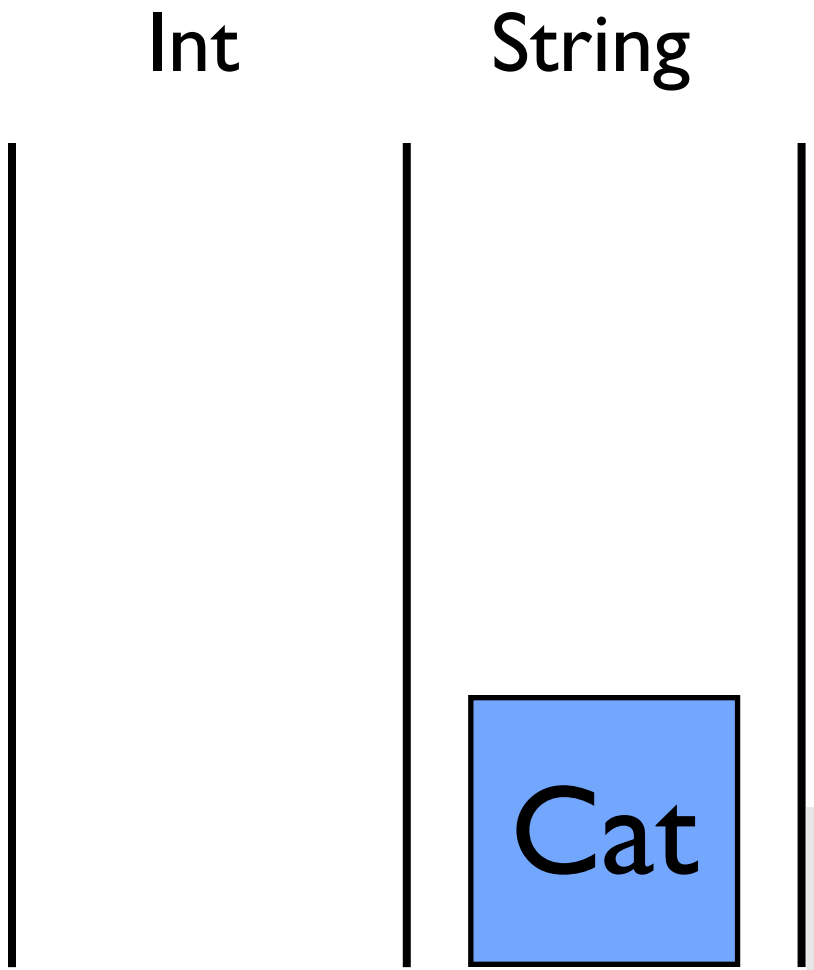


Data Error

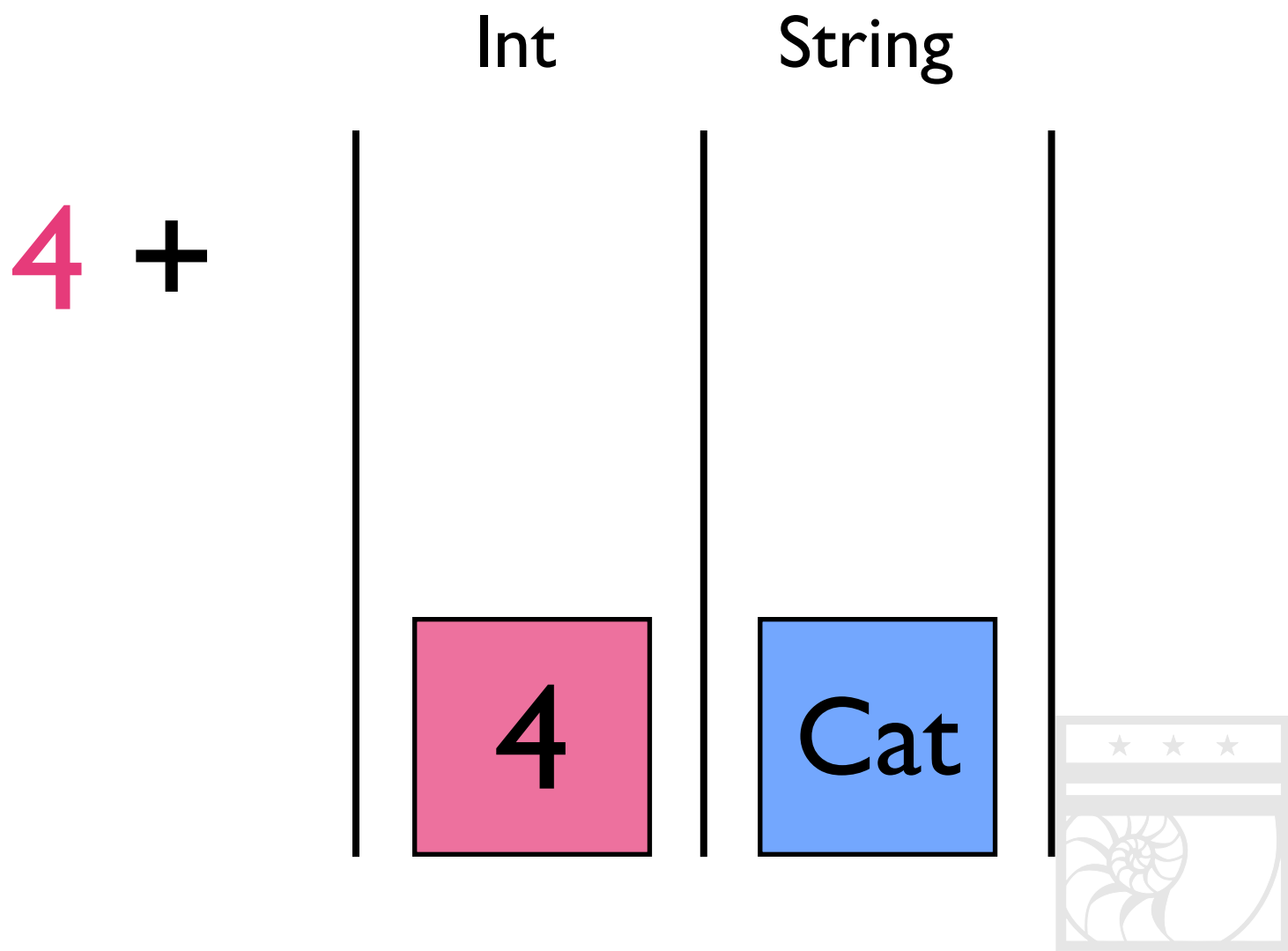


Data Error

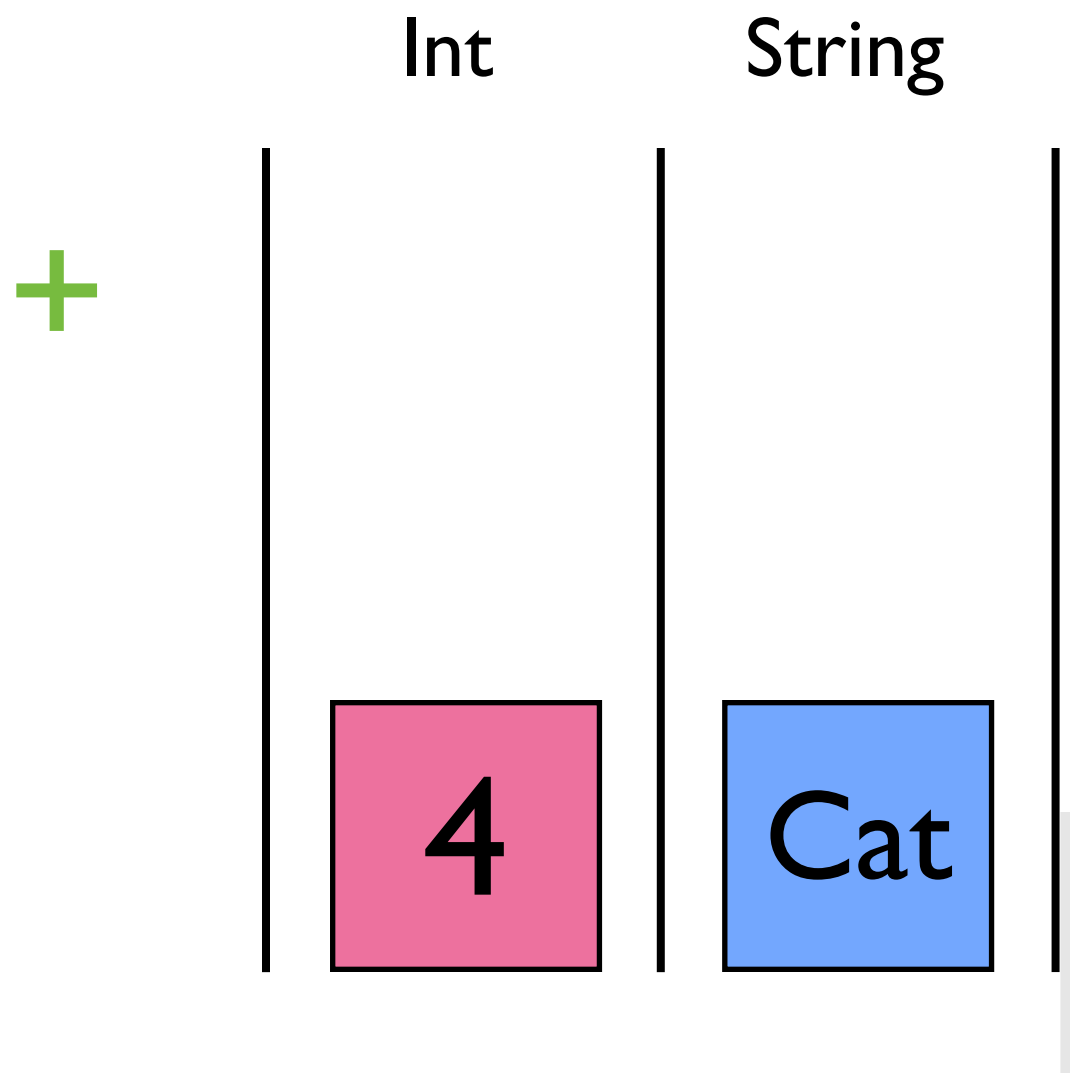
‘Cat’ 4 +



Data Error

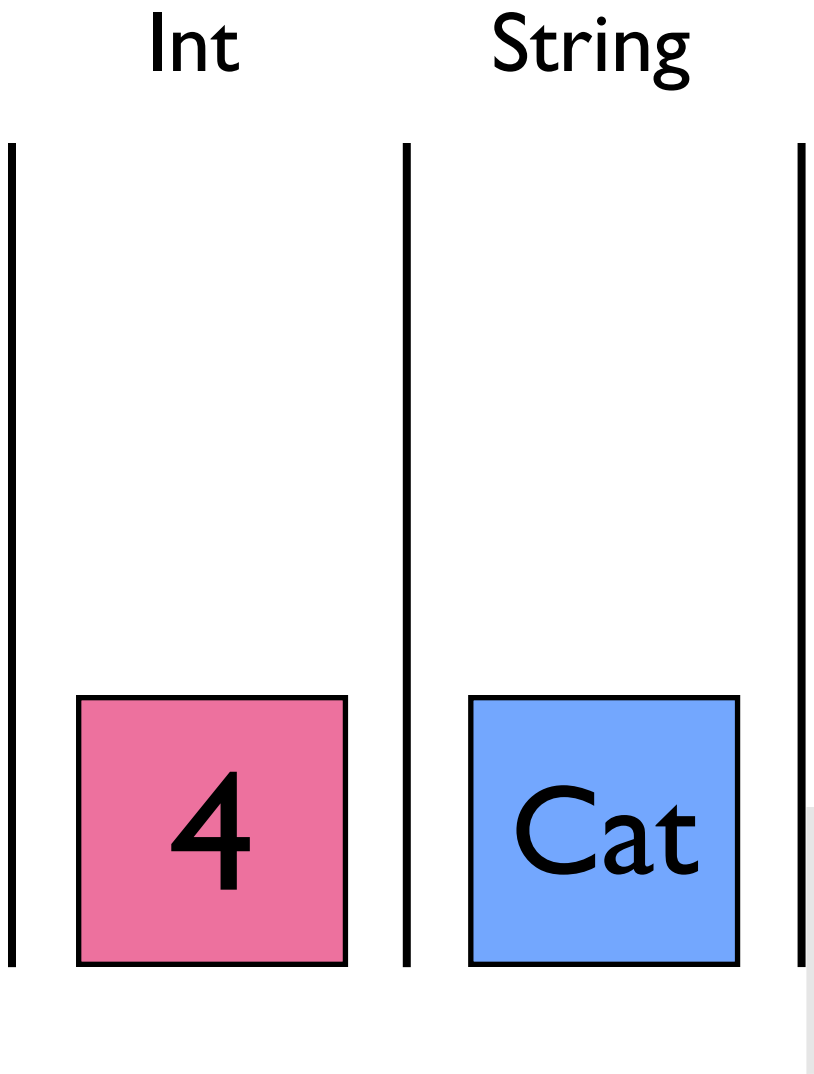


Data Error



Data Error

It NOOPd.



What Does This Give Us?

- The ability to randomly create lists of functions and literals without worry
- No more syntactic errors
- No more weird typing systems

Lets build it!

Challenges

- Use hash tables and function literals to make it nice and general
- Implement `ATOI` and `toString` functions
- Implement an `Vector` stack and vector multiplication and scalar multiplication
- Implement conditionals
  (Hint: Need a boolean stack)
- Implement while loops
  (Hint: Put your program on a stack)