Odd

Opposing Discrete and Definite Heuristics

Language Reference Manual

Alexandra Medway, Alex Kalicki, Daniel Echikson, Lily Wang
afm2134, avk2116, dje2125, lfw2114

Contents

1. Introduction

1.1 Philosophy and Motivation

I see your boundless form everywhere, the countless arms, bellies, mouths, and eyes; Lord of All, I see no end, or middle or beginning to your totality" - Arjuna to Krishna, Bhagavad-Gita.

As programmers, we are often forced to think and program in terms of definite binaries: 0 or 1, if-else, do-while, one answer or some finite number of answers. The real world, however, is not so determinate or discrete. The real world is fluid. The real world operates on chance and spectrums of possibility. As Arjuna remarks, the problems we seek solutions to frequently have no apparent beginning, middle, or end. They must be conceived of in their totality. We understand this to be the programmer's job.

The programmer must take real-world problems - problems that present themselves as neither obviously discrete nor definite - and come up with solutions that can be computed on machines that operate within the realm of the discrete and definite. We understand the programmer to be a translator of sorts, from the uncertainty of the real to the general certainty of the virtual. The motivation for *Odds* is to ease this process of translation. We recognize the need to be able to compute not only on definite values, but also on discrete and non-discrete distributions, and continuous ranges of numbers. In implementing these structures as an essential part of *Odds*, we hope to create a programming language that more seamlessly reflects the manner in which problems and solutions are posed in the real world, that is, the world of fluidity and uncertainty.

1.2 Language Description

Odds is a functional programming language that uses a simple and straightforward syntax. *Odds* centers around mathematical distributions and expresses operations on them in a direct and uncomplicated way.

Distributions support standard operations such as addition and multiplication. In addition to these simple operations, users have the option of sampling the distribution in order to apply complex calculations on portions of the data. This will allow the user to easily create simulations on ranges of data using a Monte Carlo approach.

2. Lexical Conventions

2.1 Identifiers

Identifiers in *Odds* consist of a combination of alphabetical characters, underscores, and numbers. Numbers are forbidden to be the first character of identifiers. Identifiers are case sensitive.

Valid	**Invalid**

`helloWorld`
`_helloWorld`
`h3ll0W0r1d`
	`3ll0W0r1d`
	`hello.World`
	`hello World`

2.2 Reserved Words

Odds reserves the following words. They may not be used in a program as identifiers:

```
 if        set
then      state
else      return
true      false
void
```

2.3 Literals

2.3.1 Numerics

In *Odds*, there are five types of literals. The four primary ones are integer, float, string, and boolean. The final literal type, list, will be discussed in section 3.2.

There are two numerical literal types: integer literals and floating point literals. Integer literals consist of a sequence of digits with an optional negative sign in front to indicate negativity. Floating point literals consist of optional digits in the beginning followed by a period and at least one digit after the period. Like integer literals, floating point literals can have an optional negative sign in front to indicate negativity. The following table depicts valid integer and floating point literals.

	Integer Literals	Floating Point Literals

Valid	`6`
	`192`
	`04`
		`1.67`
		`.4`
		`234.19`

Invalid	`1,000`
	`+15`
		`1.`
		`4.1.1.5`
		`+6.23`

2.3.2 Boolean

Odds also has boolean literals that can be one of two values: `true` or `false`. They denote the values of logical true and logical false respectively.

2.3.3 String

String literals are delimited by double-quotes. To place a double-quote within a string, a backslash is placed before the double-quote to escape it.

2.4 Punctuators

Type	Explanation
`( )`	Function declarations, Function Calls
`< >`	Distribution type, operators
`[ ]`	Lists
`;`	Sequencer
`,`	List Delimiter, Distribution Range Delimiter
`|`	Distribution function Delimiter
`->`	Function Delimiter
`"`	String Delimiter

2.5 Comments

Odds supports multi-line comments. A backslash followed by an asterisk denotes the start of a comment, and an asterisk followed by a backslash denotes the end of the comment. Comments may not be nested and must be closed before the end of the file is reached.

	Comment

Valid	`/* I can write anything here */`
	
	`/*`
	`* This is a`
	`* multiline`
	`* comment`
	`*/`

Invalid	`/* Nested comments are /* not */ okay */`

3. Types, Operators, and Expressions

3.1 Basic Types

Odds has five basic types. These basic types are explained below:

Type	Example	Explanation
int	`1, -213, 24`	The int type is a Signed 32-bit two's complement integer with a minimum value of -2^{31} and maximum value of $2^{31} - 1$. An optional `'-'` is placed at the front to designate as negative.
float	`1.24, .23, -27.0`	The float types is a Double-precision 64-bit IEEE 754 floating point number. Floats consist of an optional integer part, a decimal, a mandatory fraction part. An optional `'-'` is placed at the front to designate as negative.
bool	`true, false`	The bool data type has only two possibilities: `true` or `false`. Used in control flow.
string	`"23%", "Edwards is a great and benevolent`	The string type is a sequence of characters. Note *Odds* has no char type, thus even single

	`teacher.”, “y&632@”, “say: \” hello, world\””`	characters are expressed as strings. All strings are delimited by double-quotes. To place a double-quote within the string itself, escape with backslash.
void	`void`	The void type has only one value, void. It is used to represent the return type of expressions that return ‘no value.’ Expressions that are evaluated only for their side-effects, such as printing functions, return the void type.

3.2 Lists

Lists are *Odds’* basic ordered collection type. They are homogenous, i.e. consisting of only one data type. They are singly linked lists and thus have *O(1)* insertion time but *O(n)* access time. Lists are non-mutable.

Lists are delimited by square-brackets and the values within the list are comma-separated. They may be initialized as empty or containing any number of values. Below are a few examples of binding a list to an identifier using list literals:

```
set one_to_six = [1, 2, 3, 4, 5, 6]
/*one_to_six is a list of ints from 1 to 6 */

set bool_list = [true, false, false]
/*bool_list is a list of bools*/

set empty  = []
/*empty is an empty list*/

set error_list = [42, 42.0]
/* error_list is an illegal list and
 * will throw an error because it is
 * non-homogenous. It has an int and a float
 */
```

3.3 Casting

There is no implicit casting in *Odds*. All casting must therefore be done explicitly.

All casting functions follow the pattern: *x*_to_*y* where *x* is the type being converted (argument of the casting function) and *y* is the desired type (return type of the casting function). For example, `int_to_float` takes as an argument an integer and returns that integer's floating point equivalent.

Casting rules are detailed in the standard library section of this manual.

3.4 Arithmetic Operations

Odds' has six basic arithmetic operators. Because *Odds* has no implicit casting, you cannot mix floats and ints while using any of the arithmetic operators. All integer arithmetic yields an integer. All float arithmetic yields a float.

Operator	Example	Explanation

`**`	`2 ** 3,`
	`2.0 ** 1.5,`
	`3.0 ** -3.0`
		The Exponentiation operator takes the number on the left-hand side and raises it to the power of the number on the right-hand side.

`*`	`2 * 4,`
	`-2 * 4,`
	`23.42 * 0.0`
		The Multiplication operator takes the number on the left-hand side and multiplies it by the number on the right-hand side

`/`	`3 / 9,`
	`-3 / 3,`
	`27.2 / .2`
		The Division operator takes the number on the left-hand side and divides it by the number on the right-hand side.

`+`	`3 + 3,`
	`2.0 + 2.0,`
	`4.72 + -.72`
		The Addition operator takes the number on the left-hand side and adds it to the number on the right-hand side.

`-`	`6 - 9,`
	`8.34 - 9.37,`
	`8 - -2`
		The Subtraction operator takes the number on the right-hand side and subtracts it from the number on the left-hand side.

`%`	`6 % 9,`
	`7.2 % -3.2,`
	`42.0 % 42.0`
		The Modulo operator takes the number on the left-hand side and mods it by the number on the right-hand side.

3.5 Relational Operators

Odds has six basic relational operators. Mixing ints and floats is not allowed when using relational operators. All return a bool: `true` or `false`.

Operator	Example	Explanation
`<`	`1 < 2, 2.0 < -74.2`	The LessThan operator tests if the number on the left-hand side is less than the number on the right-hand side.
`>`	`4 > 7, 2.2 > 42.2`	The GreaterThan operator tests if the number on the left-hand side is greater than the number on the right-hand side.
`<=`	`1 <= 2, 2.0 <= 2.0`	The LessThanOrEqual operator tests if the number on the left-hand side is less than or equal to the number on the right-hand side.
`>=`	`2 >= 42, -72.0 >= 1.4`	The GreaterThanOrEqual operator tests if the number on the left-hand side is greater than or equal to the number on the right-hand side.
`==`	`0 == 0, -.73 == -.6`	The Equals operator tests if the number on the left-hand side is equal to the number on the right hand side.
`!=`	`1 != 1, 42.0 != .3`	The NotEquals operator tests if the number on the left-hand side is not equal to the number on the right hand side.

*Note that not all the examples above evaluate to `true`.

3.6 Logical operators

Odds has three logical operators. The expressions on each side of the operator must evaluate to a bool. All operators return a bool: `true` or `false`.

Operator	Example	Explanation
`&&`	`true && false`	Logical And
`||`	`true || false`	Logical Or
`!`	`! true`	Logical Not

3.7 Binding Operator, Set, and State

Remember, all expressions in *Odds* return a value.

3.7.1 Binding using `set` and `=`

To bind an identifier to a value or function, one uses the combination of the *set* keyword and = operator.

All bindings follow the pattern `set` *x* = *y* where *x* is the identifier and *y* is the value or function that the identifier, *x*, is being bound to. Any expression that starts with `set` is a special type of expression, a statement, and does not return any value.

For example:

```
set num = 7
/* binds the integer 7 to the identifier num */

set A_pls = “Edwards is good”
/* binds a string literal to the identifier A_pls */

set is_true = true && (true || false)
/* binds the result of a logical expression to the identifier
is_true */
```

3.7.2 `state`

Though all expressions return a value in *Odds*, sometimes there is no need to capture the actual return value. Using state allows you to ignore the return value. Like *set,* an expression that begins with *state* is a statement that does not return a value.

All uses of state follow the pattern `state` *x* where *x* is an expression whose return value you wish to ignore.

For example,

```
state print(“Teamwork!”)
/* calls the function, print, which
 * prints the string “Teamwork!”. Then
 * ignore the value print returns,(print
 * returns void).
 */
```

3.8 Sequencing

Odds' sequencing operator is `';'`. It functions very similarly to OCaml's sequencing operator. Important to note is that in *Odds,* lines/statements do not need to end in `';'` as they do in languages like C, C++, and Java.

Sometimes, however, it is necessary to write two statements in a place where you would normally only be able to write one. For example, say you wanted to call two functions in the body of an `if` statement, you would need to write:

```
if conditional then func1(); func2() else func3()
```

This means if *`conditional`* is `true`, call func1 and then, after it has completed its execution, call func2. The last sequenced statement is always the value returned by the control flow expression. In the example above, if *`conditional`* is `true`, the `if-`statement returns the value returned by `func2`.

If, however, your whole program or a function consisted of calling `func1` and `func2`, i.e. you were not calling them in the body of an `if` statement, then you could just write:

```
    state func1()
    state func2()
```

It will be made clearer in sections 4 and 5 when it is necessary to use the sequencing operator.

3.9 Precedence and Order of Operations

The precedence of operators is listed below from highest precedence to lowest precedence:

Operators	Explanation
`**`	Exponentiation
`*, /, %`	Multiplication, Division, Remainder
`+, -`	Addition, Subtraction
`==, !=, <=, >=, <, >`	Relational Operators
`!`	Logical NOT

`&&`	Logical AND
`||`	Logical OR

*All operators above are left associative

4. Control Flow

The control-flow statements of *Odds* specify the order in which computation is to be performed, as well as decision-making about which computations should be run.

4.1 Statements

An expression such as `x = 0` or `i + 1` or `print(...)` becomes a *statement* when it is preceded by the keyword `set` or `state`, as in

```
    set x = 0
    set i = i + 1
    state print("Go team!")
```

The `set` keyword indicates the intent to bind a value or function to an identifier, whereas the `state` keyword evaluates the following expression but discards the return value.

4.2 Identifier Scope

Identifiers in *Odds* are scoped to the function in which they are enclosed. There is no concept of global variables in the language. An identifier declared in a function is local to that function - `x` in function `func1` is independent from `x` declared in function `func2`. After returning from a function, any attempt to reference an identifier declared within that function results in undefined behavior as the identifier is considered out of scope and no longer usable.

```
set x = 4
set increment = (n) ->
     set x = 1      /* does not affect outer x */
     return n + x
set y = increment(x)     /* y == 5 */
/* x is still equal to 4 */
```

Identifiers referenced within functions that have not been defined in the function take on their previous value in the program. If the identifier has not been defined previously, an error results:

```
set a = 5
set adda x = x + a                  /* x + 5 */
set a = 10
state print(int_to_string(adda(0)))     /* 5 */
set adda x = x + a                  /* x + 10 */
state print(int_to_string(adda(0)))     /* 10 */
```

The following section will discuss defining and calling functions in further detail.

4.3 If, Then, Else

The if-then-else structure is used to express decisions and program control based off the results of those decisions. Formally, the syntax is

```
if expression then statement1 else statement2
```

The *expression* is evaluated; if it is equivalent to the boolean value `true`, *statement1* is executed. Conversely, if *expression* has the boolean value `false`, *statement2* is executed instead.

Because of the mandatory if-then-else structure, conditionals can be nested:

```
if expression1 then
     if expression2 then statement1 else statement2
else statement3
```

The above code will check *expression1*; if *expression1* evaluates to `true`, then the program will proceed to check *expression2*, evaluating *statement1* on `true` and *statement2* on `false`. If *expression1* evaluates to `false`, the program will instead evaluate *statement3*.

The mandatory structure also makes writing unambiguous multi-way else-if conditionals a breeze:

```
if expression1 then statement1
else if expression2 then statement2
else if expression3 then statement3
else statement4
```

Finally, the sequencing operator '`;`' discussed above comes into its own when attempting to execute multiple statements under one branch of a conditional:

```
state if expression1 then
     set x = "expression #1"; state print(x)
else
     set x = "expression #2"; state print(x)
```

Moreover, all control-flow statements return the last sequenced statement. So, the above example could also be written as:

```
set x =
     if expression1 then "expression #1"
     else "expression #2"
state print(x)
```

5. Functions and Program Structure

Functions allow programmers to break large tasks into smaller ones in order to allow for better code reuse and more readable work. *Odds* makes it easy to define and call your own functions in order to write algorithms and perform complex tasks.

5.1 Function Definition

In order to be called in various parts of a user program, a function must first be defined. Functions definitions take the form:

```
(argument_1, argument_2, …, argument_n) ->
     declarations and statements
     return statement
```

In order to better understand the process of function definition in *Odds*, let us break this function into its respective components and examine each individually. To begin with, we have the function signature:

```
(argument_1, argument_2, …, argument_n)
```

The function signature defines a list of identifiers used to refer to user-passed arguments within the function. The function name may be followed by a list of one or more argument names, each of which corresponds to a mandatory value that must be passed when the function is called. All arguments in *Odds* are passed by value.

The function signature is followed by the delimiter "`->`", and an arbitrarily long list of declarations and statements. These statements can declare local function identifiers, call other functions, or execute complex control flow logic as described in the preceding sections.

Finally, the function must end with a mandatory `return` *`statement`*. This statement uses the `return` keyword, followed by the value the function will return when called. Functions in which it is not necessary to return a value must end with `return void`.

Function definitions are usually bound to identifiers using the `set` statement, just as we would normally bind values to identifiers. In the previously given example,

```
set increment = (n) ->
     set x = 1
     return n + x
```

we are defining a function that takes one argument, bound to `n` within the function, and returns `n+1`. We bind this function to the identifier `increment` for later use. The code

```
set is_sum_even = (a, b) ->
     return if a + b % 2 == 0 then true else false
```

defines a function that takes two arguments with identifiers `a` and `b` within the function, and returns the boolean `true` if the sum of the arguments is even or `false` if the sum is odd. The code binds this function to the `is_sum_even` identifier using the `set` statement.

5.2 Calling Functions

Calling previously defined functions in *Odds* is extremely straightforward and mimics the workflow found in many other contemporary languages. Given a function

```
set increment = (n) -> return n + 1
```

you call the function by listing its identifier followed by the arguments to be passed in:

```
state print(int_to_string(increment(4))  /* 5 */
```

Anonymous functions, described below, are called by defining the function and immediately providing arguments with which to call the function.

5.3 Nested Functions

Odds supports the definition of nested functions in order to further break down functionality into its component parts. While these types of functions are most useful when making recursive calls, as discussed below, they can also be used for more straightforward computation as well.

Nested functions are defined within their enclosing functions just as local, function identifiers are bound. The nested function, being local to its enclosing shell, can not be called outside the scope with which it is defined:

```
     set is_sum_even = (a, b) ->
          set is_even = (n) -> return n % 2 == 0
          return is_even(a + b)
     state is_sum_even(2, 4)        /* true */
     state is_even(5)               /* error */
```

5.4 Anonymous Functions

In addition to declaring functions and binding their definition to identifiers, users can also define "anonymous functions" that are not bound to a name. These functions are often applicable when the functionality is only needed for an ephemeral amount of time to render a direct result:

```
     set is_4_even = ((n) -> return n % 2 == 0)(4)
     /* is_4_even == true */
```

Anonymous functions can also be used as a return type. Here, the function normal takes a mean and standard deviation and returns a function mapping a value `x` to its weight within the distribution:

```
set my_normal = (mean, stdev) ->
     return (x) ->
          set exp = -1.0*((x - mean)**2.0/(2.0 * stdev)**2.0)
          return 1 / (stdev * (2.0 * PI) ** (0.5)) * EUL ** exp

set my_standard_normal = normal(0.0, 1.0)
state print(int_to_string(my_standard_normal(0.5)))
```

5.5 Recursion

Like many languages, *Odds* supports the ability for a function to recursively call itself. Such functions typically have a base case that defines the point at which recursion ends and a recursive call if it has not yet ended. For example, one could define the Euclidean algorithm as follows:

```
set gcd = (a, b) ->
     return if b == 0 then a else gcd(b, a % b)
state print(int_to_string(gcd(48, 36)))  /* 12 */
```

6. Distributions

6.1 Definition

A distribution is a *range* (measurable set of data) to which a function of a discrete variable is applied. This function will map the set of data to a new set of weighted outcomes. However if no function is applied, the data will have a uniform distribution (as explained below).

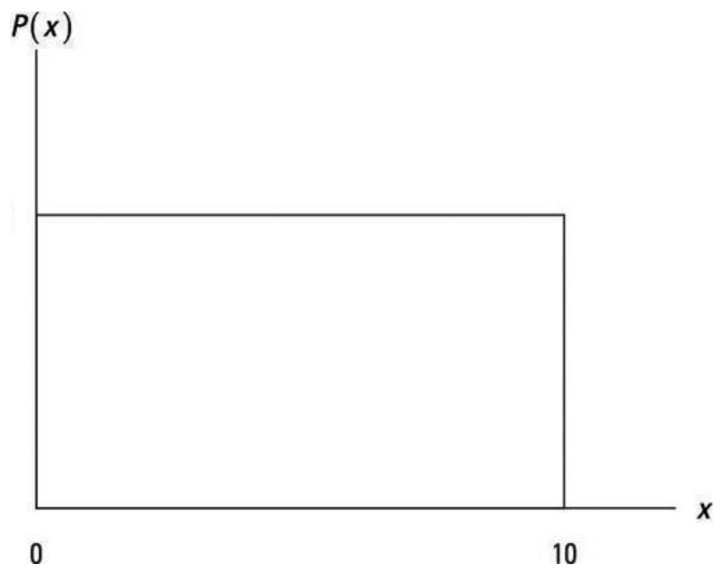
6.2 Declaration

To create a distribution in *Odds*, users need to specify the measurable set they are interested in. The measurable set of values, also called a *range*, indicates a continuous set of numbers. Declaring a distribution over the range 0 to 10 can be done in the following way:

```
set a = <0, 10>
```

Notice that in the definition of distribution, this range of numbers is meant to be mapped by a function to a set of outcomes. However, the declaration above has no such function. This is because if users don't specify a function to apply to a distribution range, *Odds* assumes a uniform distribution. Therefore, the distribution above, `a`, can be visualized with the following graph:

Figure A



Let’s create a new distribution which has a function associated with it. Functions associated with distributions, also called *maps*, are a bit different from standard functions within *Odds*. *Maps* take in one parameter of type float or int, and return one parameter of type float or int. Let’s create a function with these specifications called ‘squared.’ This function will map the input *x* to an output, x^2. We declare our function in the following way:

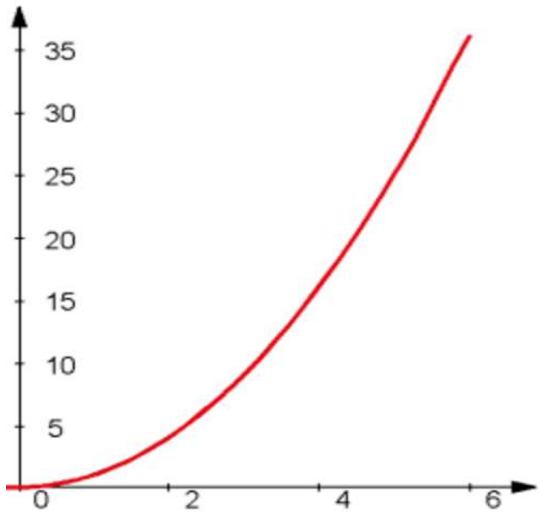
```
set squared(x) -> return x**2
```

We can then create our distribution d in the following way:

```
set b = <0, 6> | squared
```

This creates a distribution, `b`, which can be visualized with the following graph:

Figure B



Applying a function to a distribution creates weight within the distribution. Looking at the graph above, we see that a value of 4 with a weight of 16 is 4 times more likely to occur than a value of 2 with a weight of 4. Compare this to `a` in which each value is equally weighted: a value of 4 is equally as likely to occur as a value of 2.

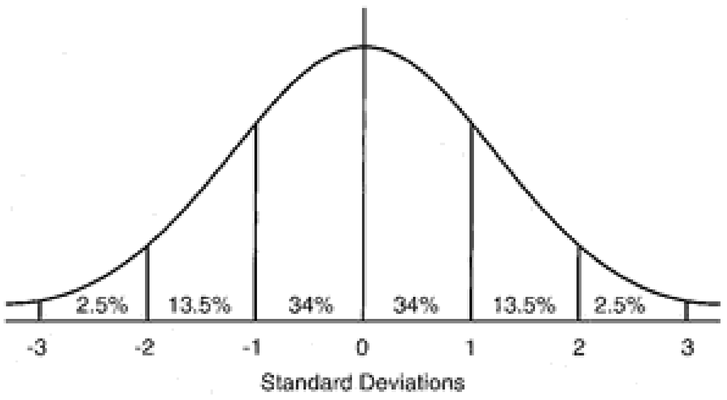
6.3 Built-In Distributions

Distributions can be used for a number of purposes, including statistical probability. *Odds* has several built in probability distributions: the *uniform* distribution, *normal* distribution, *binomial* distribution, and the *gamma* distribution. To apply these distributions to a range of numbers, the user can declare a distribution in the following way using the built-in probability distribution keyword:

```
set c = <-3, 3> | normal
```

This creates a distribution, `c`, over the range -3 to 3 with the *normal* distribution applied.

Figure C

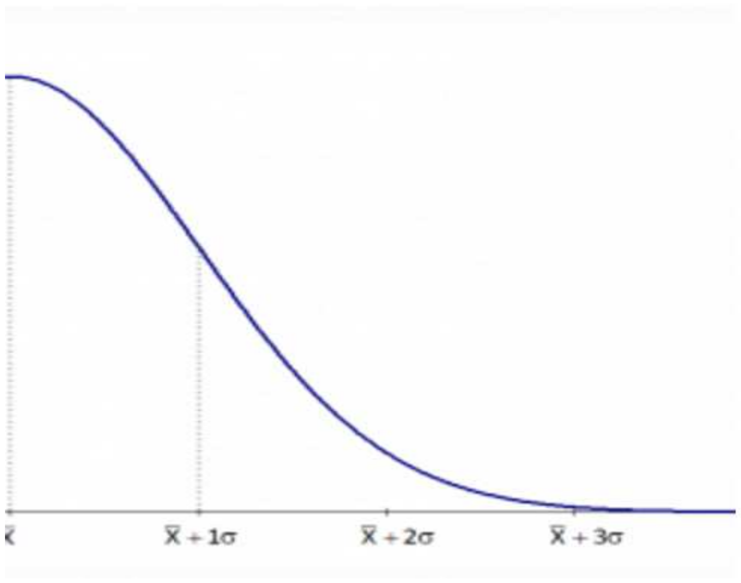


The normal distribution is centered about 0 with a standard deviation of 1. If the user doesn't center the range about 0, the distribution will be skewed.

```
set x = <0, 3> | normal
```

In a visual representation of `x`, we can see that the data is skewed.

Figure D



To create a normal distribution centered around a point other than 0, with a standard deviation other than 1, users must use *operators* as described in the next section.

Additionally, *Odds* supports *binomial* and *gamma* distributions, which can be created by replacing “normal” in the declaration for `c` with the distribution one is interested in.

6.4 Operations

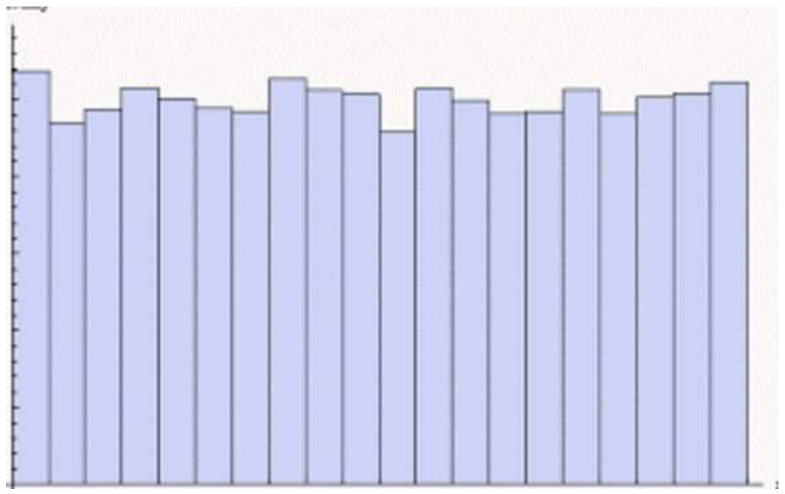
6.4.1 Sampling

There are a number of operations one can use on distributions. The first of which is *sampling*. Sampling has several advantages, as it allows the user to work with discrete values rather than a continuous range. The sampling operator can be used on a distribution, d, in the following way:

```
set d = a<100>
```

We had defined `a` to be a uniform distribution across the range 0 to 10. Thus, we would expect our sample of 100 values to reflect the continuous distribution.

Figure E



As one can see in the example above, because we have sampled discrete values and are working with experimental data, the sample doesn't have the same horizontal appearance as the distribution. However, this is a result of working with experimental data rather than theoretical data, and is to be expected.

Sampling is especially useful with statistical distributions. We create a second sample, `sample`, below using `d` from above.

```
set sample = d<10>
```

We defined d above to be a normal distribution over the range -3 to 3. Looking at Figure C one sees the percentage of values distributed within a specific standard deviation range. In the normal distribution, 68% of the values are within one standard deviation from the mean. We would expect our sample to also have ~68% of its values between -1 and 1. In `sample` above, *Odds* would select the 10 values randomly from within the `d`, given the weight of its distribution.

6.4.2 Addition and Subtraction

6.4.2.1 Constants

To transform a distribution by shifting it a constant value, one has the option of adding or subtracting constants to it.

```
set distribution = <-1, 1> | normal
/* Normal distribution from -1 to 1 */

set transformation = distribution + 5
/* Transformation is now a normal distribution
 * ranging in values from 4 to 6, with a mean
 * of 5, and the same standard deviation as
 * the original distribution
 */
```

6.4.2.2 Distributions

Two distributions of different variables can be combined into a single distribution of one variable by using the addition or subtraction operator.

```
set e = <0, 3>  /* Uniform distribution */
set f = <3, 6> /* Uniform distribution */
```

It helps to think about a distribution as a sampled list of discrete values rather than continuous values when visualizing addition. Let's choose a sample with precision value of 1, thus we are working with two lists of `[0, 1, 2, 3]` and `[3, 4, 5, 6]`. Adding the values of a distribution is not as simple as adding the lists together. Rather, each element in the first distribution must be summed with each element in the second distribution. We can then visualize addition with the following:

$$\sum\sum (e[i] + f[j])$$

As one can see, the smaller the step value between each i and j, the more precise the addition. Ideally, the step value is 0, and the distribution is entirely continuous rather than discrete. Rather than performing this complicated mathematical operation, *Odds* makes addition as simple as the following:

```
set g = e + f
```

The calculation of `g`, with precision value 1, would result in the following list:

```
[3, 4, 5, 6, 4, 5, 6, 7, 5, 6, 7, 8, 6, 7, 8, 9]
```

Sorted, we get the following:

```
[3, 4, 4, 5, 5, 5, 6, 6, 6, 6, 7, 7, 7, 8, 8, 9]
```

Notice that we no longer are working with a uniform distribution, as the value 6 appears 4 times, whereas the value 3 appears only once. Thus, 6 has a heavier weight than 3, and when sampled, is 4 times more likely to occur.

However, `g` does not have to be treated like the discrete list of numbers we see above, and can be treated as a continuous distribution which represents the combination of `e` and `f` with the operation from above.

6.4.3 Multiplication

6.4.3.1 Constants

Multiplying a distribution by a constant greater than one *stretches* the values and their corresponding weights. Dividing a distribution by a constant greater than one *contracts* the values and their corresponding weights.

```
set h = <-1, 1> | normal
set i = h * 3
/* i is a normal distribution across the range -3 to 3
 * centered about 0 with a standard deviation of 3
 */
```

6.4.3.2 Distributions

Just like addition, the multiplication operator will combine two distributions into one. If we have `e` and `f` from above, then the multiplication of the two distributions would be done with the following mathematical formula:

$$\sum\sum\ (e[i]\ *\ f[j])$$

To do this calculation in *Odds*, a user can create `j`– the multiplication of `e` and `f` – with the following line of code:

```
set j = e * f
```

6.4.4 Exponentiation

Distributions can only be exponentiated by a constant. This applies the exponentiation operator to every value in the distribution, as if the distribution were to be treated as discrete values.

```
set u = <1, 2>
/* u is a uniform distribution across the range 1 to 2 */

set v = u ** 2
/* v is a distribution across the range 1 to 4, with weights
following the previous distribution, squared */
```

6.4.5 Cropping

Distributions are defined by a range of numbers. If the user decides they wish to work with a subset of this range, they have the option to *crop* the distribution.

```
set w = <-10, 10> | normal
/* A normal distribution across the range -10 to 10 */

set z = w<-1, 1>
/* z is a normal distribution across the range -1 to 1, a subset
of w */
```

With *cropping*, you maintain the same distribution as before but remove values outside the cropped range.

6.5 Usage

Distributions allow a user to work with a range of values rather than discrete numbers. This poses a number of advantages to working with standard lists.

6.5.1 Function Application

If a user is interested in the way a function will affect a range of data, they can create a distribution and apply functions in order to transform it. This is far simpler than creating a list, and running loops to apply the function, especially if the user is unsure how precise they want their sample to be. To work with the discrete values, the user always has the option of sampling the data.

6.5.2 Monte Carlo

The *Monte Carlo* method is a way of obtaining numerical results given a broad range of data on which computation would be tedious. These tedious calculations are unnecessary in *Odds*, as any normally tedious calculation can be done easily by using the distribution creating a distribution, transforming it using the distribution operators, and sampling it to receive measurable data.

7. Standard Library

7.1 String Operations

Odds has a number of built-in functions to simplify operations on strings. These operations are detailed below:

Operator	Example	Result	Explanation
substring	`substr(“hello”, 0, 3)`	`“hel”`	The new string generated from the substring function is the span of the input string from the first numbered index (inclusive) to the second numbered index (exclusive).
concatenation	`“hello” + “ ” + “world”`	`“hello world”`	The concatenated string is sum of the strings in the input function.
str_len	`str_length(“”)`	`0`	Returns the length of the string.
==	`“hi” = “bye”`	`false`	Checks strings for equality (case sensitive).

7.2 List Operations

Lists are an important data type in *Odds*. Remember, all lists in *Odds* are homogenous and immutable.

There are two ways to create a list. A user can create a list with a set of literals, or with the included *make* function.

```
set l = [1, 2, 3]
/* A list of 1, 2, 3 */

set m = make_list(3, 0)
/* A list of size 3, all initialized to 0 */
```

There are numerous functions included in the standard library, detailed below. Note that all the functions below are returning completely new lists, they are not modifying references or anything like that; lists are immutable.

Operator	Example	Result	Explanation
make	`make_list(3, 0)`	`[0, 0, 0]`	*make* takes in two arguments, the size of the list to be created, and initialization values.
concatenation	`concat([1,2], [3,4])`	`[1,2,3,4]`	The concatenated list is the first list combined with the second list.

fold	`set str_concat = (cur, str) -> return cur + str`
	
	`fold(str_concat, “”, [“h”, “i”])`
		`“hi”`	Apply a function to a partial result and an element of the list to produce the next partial result. Moves from the end of the list to the head

iterate	`iterate(print, [“1”, “2”, “3”])`	`/*prints*/ 1 2 3`	Apply a function to each element of a list; produce a void result.
reverse	`reverse([1, 2, 3])`	`[3, 2, 1]`	Reverse the order of the elements of a list.
map	`map(int_to_string, [1, 2, 3])`	`[“1”, “2”, “3”]`	Apply a function to each element of a list to produce another list
get	`get(2, [1, 2, 3])`	`3`	Returns an element at a specific index in a list
head	`head([1, 2, 3])`	`1`	Returns the head of the list
tail	`tail([1, 2, 3])`	`[2, 3]`	Returns a list of all elements but the head.
put	`put(1, [2, 3])`	`[1, 2, 3]`	Append an element to the beginning of the list in *O(1)* time.
insert	`insert(1, 2, [1, 3])`	`[1, 2, 3]`	inserts the specified value before the specified index

remove	`Remove(0, [0,1,2,3])`	`[1, 2, 3]`	removes the value at the specified index
list_length	`list_length([1,2,3])`	`3`	Returns the length of the list

7.3 Mathematical Constants

Because *Odds* can be used for numerous mathematical purposes, such as modeling and distribution, the user is provided with built-in mathematical constants.

Constant	Value	Definition
PI	`3.14159...`	Pi.
EUL	`2.71828...`	Euler's constant, also referred to as 'e.'

7.4 Casting

Users have the option to typecast a number of data types within *Odds*. The following operations are valid:

Function	Example	Result	Explanation
int_to_float	`int_to_float(1)`	`1.0`	Converts an int to a float with a decimal value of 0.
float_to_int	`float_to_int(2.1)`	`2`	Floors the value of the float, returning the integer only.
int_to_string	`int_to_string(1)`	`"1"`	Returns a string representation of the input integer.
float_to_string	`float_to_string(1.2)`	`"1.2"`	Returns a string representation of the input float.
int_to_bool	`int_to_bool(0)`	`false`	Returns 'false' if the input integer is 0, true otherwise.

float_to_bool	`float_to_bool(1.2)`	`true`	Returns ‘false’ if the input float is 0, true otherwise.

7.5 Printing

Users only have the ability to print strings. This can be done with the following statement:

```
state print(“hello”)
```

Citation for Figures:

a. http://www.dummies.com/how-to/content/how-to-graph-the-uniform-distribution.html
b. http://raider.mountunion.edu/ma/MA125/Spring2011/Chapter10/RelationsOnSets.html
c. http://medical-dictionary.thefreedictionary.com/Normal+distribution+curve
d. https://www.spcforexcel.com/knowledge/basic-statistics/normal-distribution
e. http://davegiles.blogspot.com/2011/08/visualizing-random-p-values.html