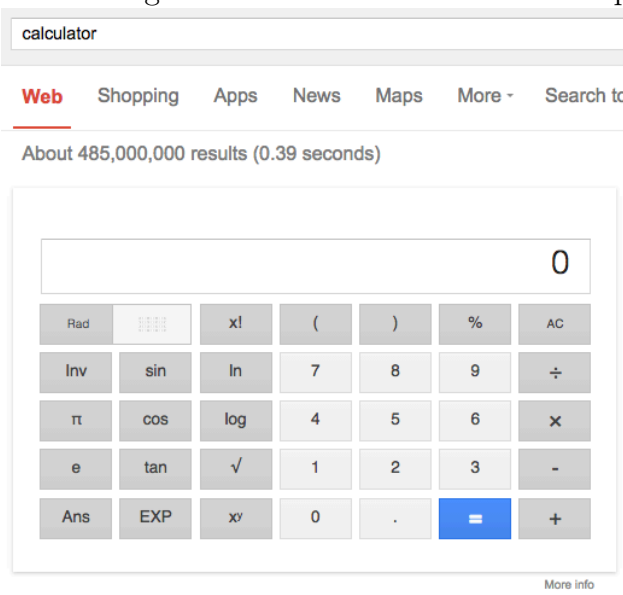
Figure 1: Screenshot of a calculator provided by the Google search engine.



1 Variables

1.1 Assignment

The Google calculator has a button `Ans` to refer to the answer to the previous evaluation. This is a form of memory. The last answer is stored in a specific place in memory for retrieval when `Ans` is used. In some calculators, more advanced memory features are possible. For some, it is possible to push values onto a stack of values for them to be referred to at a later time. This proves useful for complicated expressions, say, as the expression can be broken into smaller intermediate steps to be computed. These values can then be appropriately combined. This strategy is a good one, though the memory buttons can make its implementation a bit cumbersome.

With `Julia`, as with other programming languages, it is very easy to refer to past evaluations. This is done by *assignment* whereby a computed value stored in memory is associated with a name. The name can be used to look up the value later.

Assignment in `Julia` is handled by the equals sign and takes the general form `variable_name = value`. For example, here we assign values to the variables `x` and `y`

```
x = sqrt(2)
y = 42
```

42

In an assignment, the right hand side is returned, so it appears nothing has happened. However, the values are there, as can be checked by typing their name

```
x
```

1.4142135623730951

Just typing a variable name (without a trailing semicolon) causes the assigned value to be displayed.

Variable names can be reused, as here, where we redefine `x`:

```
x = 2
```

2

`Julia` is referred to as a "dynamic language" which means (in most cases) that a variable can be reassigned with a value of a different type, as we did with `x` where first it was assigned to a floating point value then to an integer value. (Though we meet some cases - generic functions - where `Julia` balks at reassigning a variable if the type if different.)

More importantly than displaying a value, is the use of variables to build up more complicated expressions. For example, to compute

$$\frac{1 + 2 \cdot 3^4}{5 - 6/7}$$

we might break it into the grouped pieces implied by the mathematical notation:

```
top = 1 + 2*3^4
bottom = 5 - 6/7
top/bottom
```

3 9 . 3 4 4 8 2 7 5 8 6 2 0 6 8 9

1.1.1 Examples

Example Imagine we have the following complicated expression related to the trajectory of a projectile with wind resistance:

$$\left(\frac{g}{k v_0 \cos(\theta)} + \tan(\theta)\right) x + \frac{g}{k^2} \ln\left(1 - \frac{k}{v_0 \cos(\theta)} x\right)$$

Here g is the gravitational constant 9.8 and v_0, θ and k parameters, which we take to be 200, 45 degrees, and 1/2 respectively. With these values, the above expression can be computed when $x = 100$:

```
g = 9.8
v0 = 200
theta = 45
k = 1/2
x = 100
a = v0 * cosd(theta)
(g/(k*a) + tand(theta))* x + (g/k^2) * log(1 - k/a*x)
```

9 6 . 7 5 7 7 1 7 9 1 6 3 2 1 6 1

By defining a new variable `a` to represent a value that is repeated a few times in the expression, the last command is greatly simplified. Doing so makes it much easier to check for accuracy against the expression to compute.

Example A common expression in mathematics is a polynomial expression, for example $-16x^2 + 32x - 12$. Translating this to `Julia` at $x = 3$ we might have:

```
x = 3
-16*x^2 + 32*x - 12
```

-6 0

This looks nearly identical to the mathematical expression, but we inserted `*` to indicate multiplication between the constant and the variable. In fact, this step is not needed as Julia allows numeric literals to have an implied multiplication:

```
-16x^2 + 32x - 12
```

-6 0

1.2 Where math and computer notations diverge

It is important to recognize that `=` to `Julia` is not in analogy to how $=$ is used in mathematical notation. The following `Julia` code is not an equation:

```
x = x^2
```

9

What happens instead? The right hand side is evaluated (`x` is squared), the result is stored and bound to the variable `x` (so that `x` will end up pointing to the new value `4` and not the old one `2`); finally the value computed on the right-hand side is returned and in this case displayed, as there is no trailing semicolon to suppress the output.

This is completely unlike the mathematical equation $x = x^2$ which is typically solved for values of x that satisfy the equation (0 and 1).

Example Having `=` as assignment is usefully exploited when modeling sequences. For example, an application of Newton's method might end up with this expression:

$$x_{i+1} = x_i - \frac{x_i^2 - 2}{2x_i}$$

As a mathematical expression, for each i this defines a new value for x_{i+1} in terms of a known value x_i. This can be used to recursively generate a sequence, provided some starting point is known, such as $x_0 = 2$.

The above might be written instead with:

```
x = 2
x = x - (x^2 - 2) / (2x)
```

1 . 5

Repeating this last line will generate new values of `x` based on the previous one - no need for subscripts. This is exactly what the mathematical notation indicates is to be done.

1.3 Context

The binding of a value to a variable name happens within some context. For our simple illustrations, we are assigning values, as though they were typed at the command line. This stores the binding in the `Main` module. `Julia` looks for variables in this module when it encounters an expression and the value is substituted. Other uses, such as when variables are defined within a function, involve different contexts which may not be visible within the `Main` module.

The `varinfo` function will list the variables currently defined in the main workspace. There is no mechanism to delete a single variable.

Shooting oneselves in the foot. `Julia` allows us to locally redefine variables that are built in, such as the value for `pi` or the function object assigned to `sin`. For example, this is a perfectly valid command `sin=3`. However, it will overwrite the typical value of `sin` so that `sin(3)` will be an error. The binding to `sin` occurs in the `Main` module. This shadows that value of `sin` bound in the `Base` module. Even if redefined in `Main`, the value in base can be used by fully qualifying the name, as in `Base.sin(pi)`. This uses the convention `module_name.variable_name` to look up a binding in a module.

1.4 Variable names

`Julia` has a very wide set of possible names for variables. Variables are case sensitive and their names can include many Unicode characters. Names must begin with a letter or an appropriate Unicode value (but not a number). There are some reserved words, such as `try` or `else` which can not be assigned to. However, many built-in names can be locally overwritten. Conventionally, variable names are lower case. For compound names, it is not unusual to see them squished together, joined with underscores, or written in camelCase.

```
value_1 = 1
a_long_winded_variable_name = 2
sinOfX = sind(45)
__private = 2     # a convention
```

2

1.4.1 Unicode names

Julia allows variable names to use Unicode identifiers. Such names allow `julia` notation to mirror that of many mathematical texts. For example, in calculus the variable ϵ is often used to represent some small number. We can assign to a symbol that looks like ϵ using `Julia`'s LaTeX input mode. Typing `\epsilon[tab]` will replace the text with the symbol within `IJulia` or the command line.

```
ϵ = 1e-10
```

1 . 0 e - 1 0

Entering Unicode names follows the pattern of "slash" + LaTeX name + `[tab]` key. Some other ones that are useful are `\delta[tab]`, `\alpha[tab]`, and `\beta[tab]`, though there are hundreds of other values defined.

For example, we could have defined `theta` (`\theta[tab]`) and `v0` (`v\_0[tab]`) using Unicode to make them match more closely the typeset math:

```
θ = 45; v_0 = 200
```

200

There is even support for tab-completion of emojis such as `\:snowman:[tab]` or `\:koala:[tab]`

Example As mentioned, as of `Julia v"0.7.0"`, the value of e is bound to the unicode value `\euler[tab]` and not the letter `e`, so Unicode entry is required to access this constant This isn't quite true. The `MathConstants` module defines `e`, as well as a few other values accessed via unicode. This is loaded with the accompanying `CalculusWithJulia` package. Alternatively, one can define `e = exp(1)` and use the variable as desired.

1.5 Tuple assignment

It is a common task to define more than one variable. Multiple definitions can be done in one line, using semicolons to break up the commands, as with:

```
a = 1; b = 2; c=3
```

3

For convenience, `Julia` allows an alternate means to define more than one variable at a time. The syntax is similar:

```
a, b, c = 1, 2, 3
```

```
(1, 2, 3)
```

This sets `a=1`, `b=2`, and `c=3`, as suggested. This construct relies on *tuple destructuring*. The expression on the right hand side forms a tuple of values. A tuple is a container for different types of values, and in this case the tuple has 3 values. When the same number of variables match on the left-hand side as those in the container on the right, the names are assigned one by one.

The value on the right hand side is evaluated, then the assignment occurs. The following exploits this to swap the values assigned to `a` and `b`:

```
a, b = b, a
```

```
(2, 1)
```

Example, finding the slope Find the slope of the line connecting the points $(1, 2)$ and $(4, 6)$. We begin by defining the values and then applying the slope formula:

```
x0, y0 = 1, 2
x1, y1 = 4, 6
m = (y1 - y0) / (x1 - x0)
```

1 . 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3

Of course, this could be computed directly with `(6-2) / (4-1)`, but by using familiar names for the values we can be certain we apply the formula properly.

1.6 Questions

⊛ Question
Let $a = 10$, $b = 2.3$, and $c = 8$. Find the value of $(a - b)/(a - c)$.

⊛ Question
Let `x = 4`. Compute $y = 100 - 2x - x^2$. What is the value:

⊛ Question
What is the answer to this computation?

```
a = 3.2; b=2.3
a^b - b^a
```

⊛ Question
For longer computations, it can be convenient to do them in parts, as this makes it easier to check for mistakes.

For example, to compute

$$\frac{p - q}{\sqrt{p(1 - p)}}$$

for $p = 0.25$ and $q = 0.2$ we might do:

```
p, q = 0.25, 0.2
top = p - q
bottom = sqrt(p*(1-p))
ans = top/bottom
```

What is the result of the above?

⊛ Question
Using variables to record the top and the bottom of the expression, compute the following for $x = 3$:

$$y = \frac{x^2 - 2x - 8}{x^2 - 9x - 20}.$$

⊛ Question
Which if these is not a valid variable name (identifier) in `Julia`:

1. `some_really_long_name_that_is_no_fun_to_type`
2. `aMiXeDcAsEnAmE`
3. `5degreesbelowzero`
4. `fahrenheit451`

⊛ Question
Which of these symbols is one of `Julia`'s built-in math constants?

1. `pi`
2. `I`
3. `E`
4. `oo`

⊛ Question
What key sequence will produce this assignment

```
δ = 1/10
```

1. `\delta[tab] = 1/10`
2. `$\\delta$ = 1/10`
3. `delta[tab] = 1/10`

⊛ Question
Which of these three statements will **not** be a valid way to assign three variables at once:

1. `a,b,c = 1,2,3`
2. `a=1, b=2, c=3`
3. `a=1; b=2; c=3`

⊛ Question
The fact that assignment *always* returns the value of the right hand side *and* the fact that the = sign associates from right to left means that the following idiom:

```
x = y = z = 3
```

Will always:

1. Assign all three variables at once to a value of 3
2. Create 3 linked values that will stay synced when any value changes
3. Throw an error