

# Cesium : A New Language

William B. Hart

November 23, 2012

## Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>The Cesium interpreter</b>	<b>2</b>
<b>3</b>	<b>Code comments</b>	<b>2</b>
<b>4</b>	<b>Literals</b>	<b>2</b>
4.1	Integer literals . . . . .	2
4.2	Floating point literals . . . . .	3
4.3	Character literals . . . . .	3
4.4	String literals . . . . .	4
<b>5</b>	<b>Expressions</b>	<b>4</b>
5.1	Tuples . . . . .	4
5.2	Binary arithmetic operators . . . . .	4
5.3	Binary relational operators . . . . .	5
5.4	Ternary “if” expressions . . . . .	5
<b>6</b>	<b>Variables and assignment</b>	<b>6</b>
6.1	Global variables . . . . .	6
6.2	Tuple assignment . . . . .	6
6.3	Tuple unpacking . . . . .	6

## 1 Introduction

Cesium is a new programming language designed primarily for efficiency, easy interface with C libraries and with a straightforward high-level syntax. It uses the LLVM Jit to give very performant code, even when run in interactive mode. The language is polymorphic, statically typed (with type inference) and imperative in style, with some functional features.

The language fills a niche between Python and C. Python is not terribly performant, but has a very simple and flexible syntax. C is fiddly and low level, statically compiled and not interactive, but very fast.

Cesium is inspired by Julia, but is more suited to computer algebra, rather than numerical tasks. Of course Cesium can be used as a general purpose language, but design decisions are influenced by the desire to make it useful to a certain class of mathematical/computational people.

## 2 The Cesium interpreter

The Cesium interpreter is started by typing `./cs` at the command line.

When the interpreter starts, a prompt is displayed ready to accept Cesium code.

If you type an expression and press enter, Cesium immediately executes the expression and a return value is displayed. If there is no return value, `none` is printed.

Note that end-of-lines are significant in Cesium. In many cases they can be elided by addition of certain keywords. However, in general, statements must finish with an end-of-line.

## 3 Code comments

Cesium allows you to add comments to your code, which are ignored by the parser. For example, `/* this is a comment */`.

Comments are allowed anywhere that a space is allowed.

```
if 1 < 2 /* check if 1 is less than 2 */
  3 /* if it is, return 3 */
else
  4 /* if not, return 4 */
end /* end of if */
```

## 4 Literals

We first describe how to enter various types of constant literals in Cesium.

### 4.1 Integer literals

Here are some examples of signed and unsigned integers in Cesium:

```
1234
12u8
109i8
278009804u
234567i32
```

Integers with no suffix are signed and either 32 or 64 bits, depending on whether a 32 or 64 bit binary of Cesium is being used.

Integers which are unsigned are given the suffix `u`.

The other suffixes, e.g. `i8`, `u32`, etc., specify whether the number is signed (with a `i`) or unsigned (with a `u`) and how many bits the type is. The valid numbers of bits are 8, 16, 32, 64.

For example, the literal `109i8` is the integer 109 stored as a signed 8 bit value.

The suffixes are necessary in Cesium because it uses type inference to infer the type of a value from the value itself. In the case of numbers this is not possible without the extra annotation.

In practice, only unannotated types are required unless one has some specific bit fiddling to do or if one is interfacing with a C program which makes use of specific types.

The Cesium types for integer values are `int8`, `uint8`, `int16`, `uint16`, etc., depending on the suffix the value has.

## 4.2 Floating point literals

There are two types of floating point values in Cesium, `float` and `double` values.

Floating point values must have a decimal point and have an optional exponent. The exponents begin with an optional sign, followed by an integer.

Without a suffix, floating point values are considered to be double precision floating point values. However, if they have the suffix `f` they are considered to be single precision floats.

Here are some example floating point values:

```
1.0
2.1f
123.4
1.3e+12
1.4E-11f
```

For example, `1.4E-11f` is the single precision floating point value  $1.4 \times 10^{-11}$ .

## 4.3 Character literals

Cesium has a `char` type for single ASCII characters. To represent a character one simply wraps it in single quote marks.

```
'c'
's'
','
','
```

Some special characters are also available. These characters all begin with a backslash:

```
'\n' /* new line */
'\r' /* carriage return */
'\t' /* tab */
'\0' /* null character */
'\' ' /* backslash */
'\\" /* double quotes */
'\'' /* single quotes */
```

## 4.4 String literals

One can represent strings in Cesium by putting them between double quotation marks:

```
"this is a string"  
"123"  
" %51"
```

To include a double quotation mark in a string, it must be escaped with a backslash:

```
"this \"string\" contains double quotes"
```

The type of a string value in Cesium is `string`.

## 5 Expressions

When an expression is entered at the Cesium prompt, it immediately evaluates the expression and prints the result. For example, typing the expression `1 + 1` at the prompt results in the value `2`.

We now discuss the various ways of constructing expressions in Cesium.

### 5.1 Tuples

A tuple of values can be entered by separating the individual expressions by commas and wrapping in parentheses. Note that a tuple with only one entry must have a comma before the final parenthesis to distinguish it from a set of parentheses around an expression which are acting as a grouping rather than as a tuple constructor.

Here are some examples of tuples expressions:

```
(1, 2, 3)  
(0,)   
(3, 'c', "string", (2.3, 1.1f, 3u8))
```

A tuple can be thought of as an anonymous data type in Cesium. It can contain inhomogeneous data, i.e. the types of the expressions in the tuple do not need to be the same. This is different to arrays, which must have homogeneous data. However, unlike arrays, tuples cannot be accessed by index. To extract values from a tuple, one needs to use tuple unpacking, described later.

The type of a tuple value is given by a tuple of types, e.g. the tuple `(1, 'c', "string")` has type `(int, char, string)`.

### 5.2 Binary arithmetic operators

To perform computations with values, one has the binary arithmetic operators `+`, `-`, `*`, `/` and `%`, representing binary addition, subtraction, multiplication, division and remainder, respectively.

These operators are defined for each of the integer and floating point types.

The ordinary mathematics rules for operator precedence and associativity apply. But to evaluate in a different order, one can group using parentheses.

Here are some example expressions using the arithmetic operators:

```

1 + 2
1 - 2*3 - (4 - 7*6)
2.3 / 4.0
12.7 % 1.2
11 % 3

```

### 5.3 Binary relational operators

A binary relational operator takes two values, performs some comparison on them and then returns a `bool` depending on the result of the comparison.

The relational operators available in Cesium are `<`, `>`, `<=`, `>=`, `==` and `!=`.

Here are some expression using these operators:

```

1 < 2
12.1 >= 15.0
7 != 7
4i8 == 4i8

```

Each of them returns either `true` or `false` depending on whether the stated relation holds or not.

### 5.4 Ternary “if” expressions

An `if` expression in Cesium is an expression which takes one of two values, depending on the outcome of a comparison. It is best illustrated with an example:

```
if 1 < 2 then 3 else 4 end
```

This expression has the value 3 if `1 < 2`, otherwise it has the value 4.

There is no requirement that an `if` expression must be all on one line. For example, the following is equivalent to the above.

```

if 1 < 2 then
  3
else
  4
end

```

This format also allows for additional lines of code to be inserted in each branch (see `if` statements later in the documentation for examples).

The expressions in each branch of an `if` expression must have the same type. This then becomes the type of the entire expression. Thus, for example, the type of the `if` expression in the above example is `int`.

For convenience, the `then` keyword may be elided in the multiline version of an `if` expression:

```

if 1 < 2
  3
else
  4
end

```

## 6 Variables and assignment

### 6.1 Global variables

To define a variable in Cesium, one uses an assignment to assign a given value to that variable. It is not necessary to declare variables in advance. They begin to exist at the moment they are first assigned a value.

Here are some examples of assignment:

```
i = 1
mytup = (1, 'c', "string")
v = i + 1
s = if 1 < 2 then 3 else 4 end
```

If a variable is defined at the top level (at the Cesium prompt), it is called a global variable and its value can be used or changed anywhere after that point.

### 6.2 Tuple assignment

Sometimes it is convenient to be able to assign a whole lot of values at once. This can be done with tuple assignment. For example, the following code defines variables `i`, `j` and `k` (if they don't already exist), and assigns the values 0, 1 and "string", respectively, to them:

```
(i, j, k) = (0, 1, "string")
```

Note that tuple assignment is done efficiently by the compiler without actually creating a tuple. It's simply syntactic sugar for multiple assignment.

### 6.3 Tuple unpacking

Sometimes one has a value which is a tuple and one wants to extract the values individually. This is called tuple unpacking.

For example, suppose that, at some point in our code, we have:

```
t = (0, 1, "string")
```

We can extract the three values in the tuple `t` as follows:

```
(a, b, c) = t
```

Now, `a` will have the value 0, `b` the value 1 and `c` the value "string".

Of course, the compiler does not actually create a tuple on the left hand side. It merely assigns the three values from inside the tuple to the given variables.

Tuple assignment and unpacking can be combined, e.g.

```
(a, b, (c, d)) = (1, 2, (3, 4))
```

Here tuple assignment is used to assign the values of `a` and `b` without creating a tuple. Then the tuple (3, 4) is created and unpacked using tuple unpacking into `c` and `d`.

The compiler could optimise this case further by not creating the tuple (3, 4) and simply assigning the values directly to `c` and `d`. However, this is not important as the code above could also be written as follows, achieving the same thing efficiently:

```
(a, b, c, d) = (1, 2, 3, 4)
```