Cesium : A New Language

William B. Hart

February 9, 2013

Contents

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 some cases they can be replaced with 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
1234U
123u
```

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 with the suffix `U` or `u` are unsigned 32 or 64 bit integers.

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 or with the suffix `d`, 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
1.2d
123.4
1.3e+12
1.4E-11f
```

For example, `1.4E-11f` is the single precision floating point value 1.4×10^{-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:

```
s = 1
mytup = (1, 'c', "string")
v = s + 1
t = if s < 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 `a`, `b` and `c` (if they don't already exist), and assigns the values `0`, `1` and `"string"`, respectively, to them:

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

7 Control flow statements

The path through a program is directed by control flow statements. We now discuss the various control flow statements available in Cesium.

7.1 The *if* statement

The `if` statement executes a series of statements if a given condition is true.

Here is an example of code fragment incorporating an `if` statement:

```
s = 1
t = 2
if s < 2
   t = t + 1
   s = t + 3
end
```

As with an `if` expression, the condition can be followed by the keyword `then`, and if the code that is to be executed is a simple statement, the entire thing can be written on one line:

```
s = 1
if s < 2 then s = 3 end
```

The branch of an `if` statement does not define a new lexical scope. Thus variables defined inside the `if` statement also exist outside it.

No value is returned by an `if` statement, and so `none` will be printed after evaluating an `if` statement at the Cesium prompt.

7.2 The *if..else* statement

The `if..else` statement executes the statements in one of two branches, depending on a given condition.

Here is an example of code fragment incorporating an `if..else` statement:

```
s = 1
t = 2
if s < 2
   t = t + 1
   s = t + 3
else
   s = 4
end
```

Note that a branch of an `if..else` statement does not define a new lexical scope. Thus variables defined inside the `if..else` statement also exist outside it.

As with the `if` statement and `if` expression, the condition can be followed by the keyword `then`, and if each branch only contains a single simple statement, then the entire thing can be written on one line:

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

No value is returned by an `if..else` statement, and so `none` will be printed after evaluating an `if..else` statement at the Cesium prompt.

Note that the branches of an `if..else` statement do not define a new lexical scope. Thus variables defined inside a branch of an `if..else` statement also exist outside the statement.

Recall that an `if..else` statement whose branches both end in expressions (which must be of the same type) is called an `if` expression. An `if` expression does return a value, whose type is the same as that of the two expressions.

Here is an example of a multi-line `if` expression.

```
s = 1
t = 2
if s < 2
   t = t + 1
   s + t
else
   s = 4
   7
end
```

In this example, the `if` expression will have the value `4` because the first branch is taken and the final expression in that branch is `s + t`, which has the value `4`.

7.3 The *while* statement

A simple loop in Cesium can be obtained with the `while` statement. It continues to execute its body in a loop so long as a given condition is true.

Here is a simple example of a `while` loop:

```
c = 0
s = 0
while c < 1000000 do
   s = s + c
   c = c + 1
end
```

If there is only a single simple expression in the body of a while loop, the entire thing can be written on a single line.

In the multiline version, the `do` keyword can be elided. Thus the example above can be written:

```
c = 0
s = 0
while c < 1000000
   s = s + c
   c = c + 1
end
```

The body of a `while` statement does not define a new lexical scope. Thus variables defined inside the `while` statement also exist outside it.

A `while` statement does not return a value, and so `none` is printed if one is evaluated at the Cesium prompt.

8 User defined types

8.1 The *type* declaration

Just as a tuple is an anonymous type in Cesium, the user can define types with names using the `type` statement.

The body of a type statement contains a list of fields, each of which is given a type. Any Cesium or user defined type can be used for such a type declaration.

Here is a simple example of declaring a new type called `blah`.

```
type blah
  a : int
  b : double
  c : (int, char, string)
end
```

This type contains three fields, `a` of type `int`, `b` of type `double` and `c` which has the tuple type `(int, char, string)`.

Type declarations in Cesium can refer to other user defined types, even ones which don't exist yet. So long as the type is not instantiated before all the required types are defined, the compiler will not complain about undefined type references.

Here is an example of a type referring to another type which is defined subsequently:

```
type mytype1
  a : int
  b : mytype2
end

type mytype2
  c : int
  d : double
end
```

The advantage of a `type` declaration over a tuple is that the fields inside the type can be referred to by name (see the dot operator below).

8.2 Default constructors

Once a new type has been defined using a `type` statement, Cesium automatically creates a default constructor for the type of the same name. A constructor is a function which takes a set of values as parameters and returns an instance of the newly defined type with those values inside.

For example, the `blah` type declaration above automatically creates a `blah` function which can be called as follows:

```
v = blah(1, 2.3, (2, 's', "hello"))
```

The value `v` now has type `blah` with the given data inside it.

8.3 The dot operator

Given a value whose type is user defined using a `type` statement, the individual fields of the value can be accessed by name using the dot operator.

For example, to access the field `c` of the value `v` of type `blah` in the above examples, we use the expression `v.c`.

Here are some examples of this:

```
type circle
   x : int
   y : int
   radius : int
end

c = circle(20, 20, 10)
m1 = c.radius + c.x

c.x = 40
```

In the expression `m1`, we initially have that `c.x` has the value `20`. This value is then used in the expression for `m1` before it is subsequently changed to `40`.

Note that when a value of user defined type is assigned to another variable, the new variable points to the same actual data. Thus modifying the fields of the second variable modifies the value of the original.

For example:

```
type t1
   x : int
end

c = t1(20)
d = c
d.x = 40
```

In this example, `c.x` starts with the value `20`. But `d.x` aliases the same data and thus the final value of `c.x` is `40`.