The design of VLAM Compyle

A tower of languages from python to C++ using scheme

Marc-Antoine Desroches
mad@vlam.ca

Contents of the presentation

- Presentation of Compyle
- Examples
- The big picture & the tower of languages
- Conclusion

What is Compyle

Compyle is a compiler.

It translates a subset of the Python programming language into C++.

What Compyle does differently

- No type annotations in source code → generic code
- Just-in-time
  - the C++ translation of a function is generated every time it is called with a different set of argument types
- Generates multi-threaded code
- Emphasis on performance over the support of the whole of Python
  - meant to optimize the parts of your program you know are slow

Why

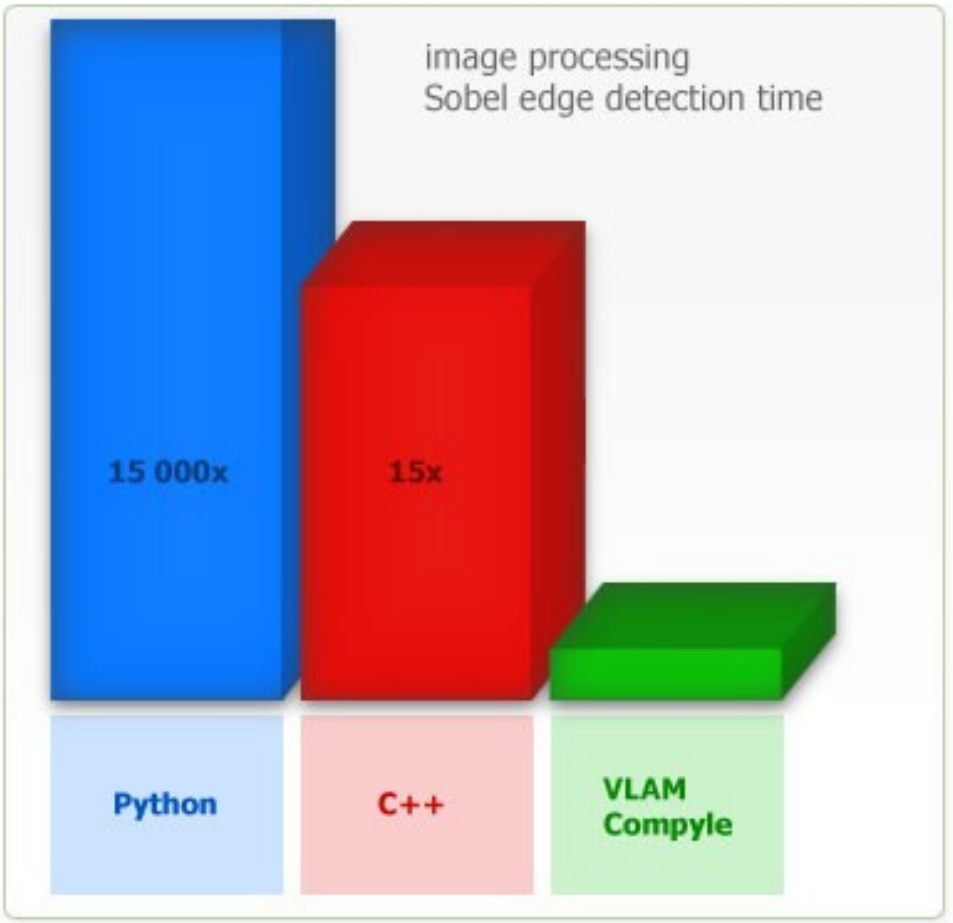


image processing
Sobel edge detection time
15 000x
15x
Python
C++
VLAM
Compyle

- Presentation of Compyle
- Examples
- The big picture & the tower of languages
- Conclusion

Hello world

```
from vlam_compyle import jit
s = """
def hello():
    print "Hello world!"
"""
m = jit.module( s )
m.hello()
```

Hello world : generated C++

```
__declspec(dllexport)void hello ()
{
   ((std::cout << "Hello world!") << std::endl);
};
;
```

A less trivial example

```
s = """
def called( x, y ):
    return x + y


def test_overload( x, y ):
    return called( x + 1, y ) + called( x + 1.5, y )
"""
module = jit.module( s )
self.assertEqual( 8.5, module.test_overload( 1, 2 ) )
```

A less trivial example

```
int32 called (int32 x, int32 y)
{
    return (x + y);
};
float64 called (float64 x, int32 y)
{
    return (x + y);
};
__declspec(dllexport)float64 test_overload (int32 x, int32 y)
{
    return (called((x + 1), y) + called((x + 1.5), y));
};
```

2D arrays & threads

```
s = """
def test_buffer_exp( dest, zero ):
    dest = (zero + 1) * 2
"""
m = jit.module( s )


buffer_size = ( 1024, 1024 )
buffer0 = numpy.zeros( buffer_size, dtype=numpy.ubyte )
buffer_dest = numpy.empty( buffer_size, dtype=numpy.ubyte )
m.test_buffer_exp( buffer_dest, buffer0 )
test = buffer_dest == 2
assert( test.all() )
```

2D arrays & threads

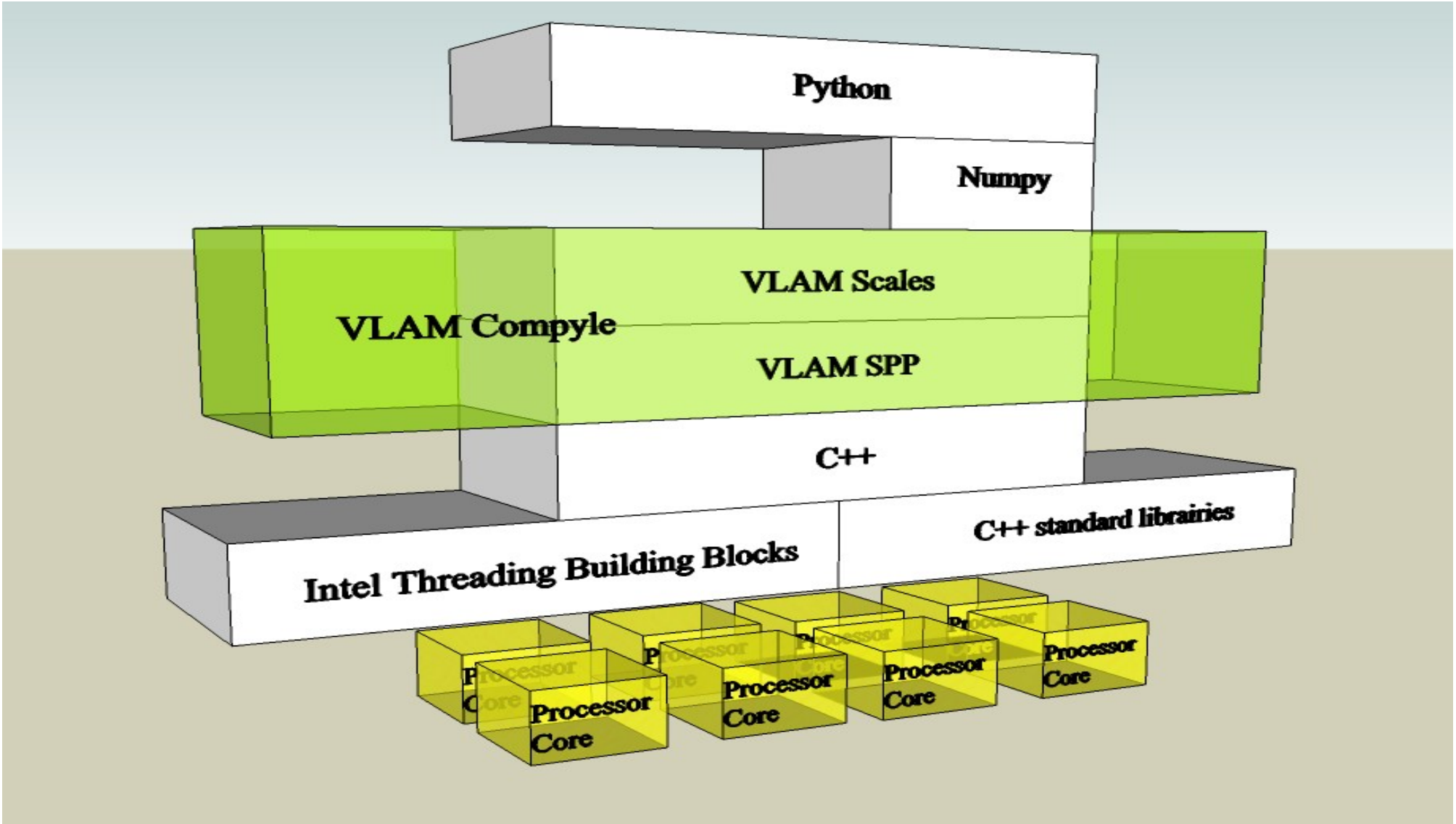
```
__declspec(dllexport)void test_buffer_exp (
        uint8* dest_ptr, int dest_height, int dest_width,
        uint8* zero_ptr, int zero_height, int zero_width)
{
    tbb::parallel_for(
        tbb::blocked_range<int>(0, ((0 == 0) ? dest_height : dest_width)),
            g300( dest_ptr, dest_height, dest_width,
                    zero_ptr, zero_height, zero_width));
};
```

2D arrays & threads

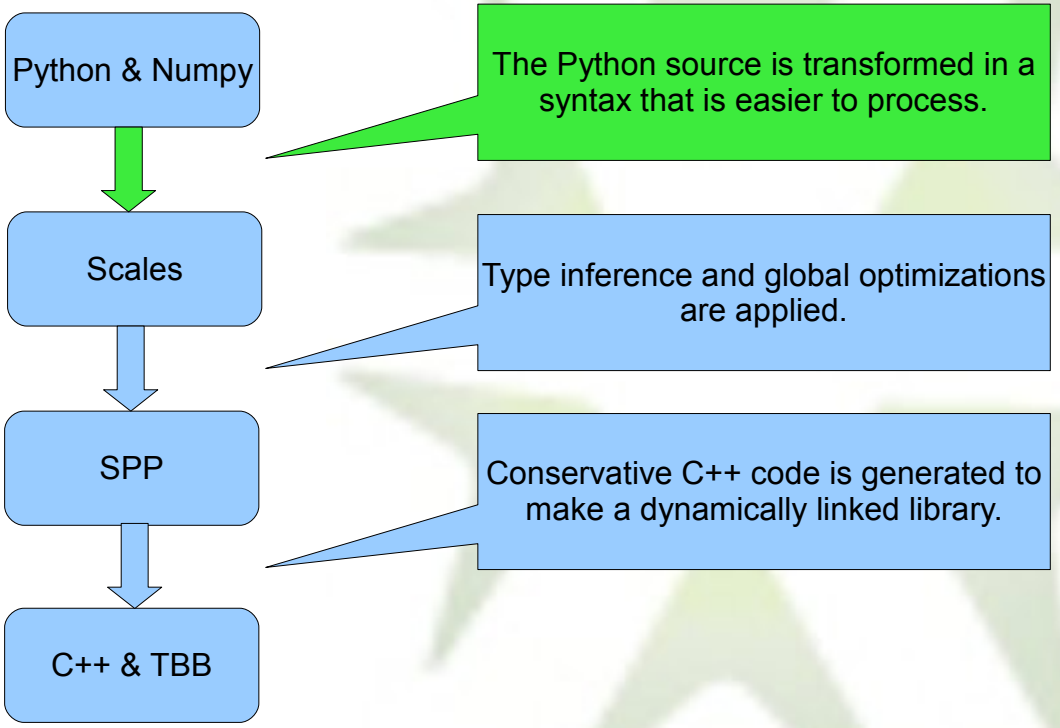
```
struct g300

{

    mutable uint8* dest_ptr; int dest_height; int dest_width;

    mutable uint8* zero_ptr; int zero_height; int zero_width;

    g300 (uint8* __in_dest_ptr, int __in_dest_height, int __in_dest_width, uint8* __in_zero_ptr, int __in_zero_height, int __in_zero_width) :dest_ptr(__in_dest_ptr), dest_height(__in_dest_height),
dest_width(__in_dest_width), zero_ptr(__in_zero_ptr), zero_height(__in_zero_height), zero_width(__in_zero_width)

    {

    };

    void operator() (const tbb::blocked_range<int>& g301) const

    {

        for( int __row_id__sym5 = g301.begin(); (__row_id__sym5 < g301.end()); (++ __row_id__sym5) )

        {

            for( int __col_id__sym6 = 0; (__col_id__sym6 < ((0 == 1) ? dest_height : dest_width)); (++ __col_id__sym6) )

            {

                (dest_ptr[((dest_width * __row_id__sym5) + __col_id__sym6)] =
((zero_ptr[((zero_width * __row_id__sym5) + __col_id__sym6)] + 1) * 2));

            };

        };

    };

};
```

- Presentation of Compyle
- Examples
- The big picture & the tower of languages
- Conclusion

The big picture



The tower of languages



Python → Scales

```
s = """
def test_for( x ):
    for i in range(10):
        x += i
    return x
"""
fun = jit.module( s ).test_for
self.assertEqual( 46, fun( 1 ) )
```

Python → Scales

(file_input (stmt (compound_stmt (funcdef (NAME def) (NAME test_for) (parameters (LPAR \() (varargslist (fpdef (NAME x))) (RPAR \))) (COLON :) (suite (stmt (compound_stmt (for_stmt (NAME for) (exprlist (expr (xor_expr (and_expr (shift_expr (term (factor (power (atom (NAME i)))))))))) (NAME in) (testlist (test (or_test (and_test (not_test (comparison (expr (xor_expr (and_expr (shift_expr (term (factor (power (atom (NAME range)) (trailer (LPAR \() (arglist (argument (test (or_test (and_test (not_test (comparison (expr (xor_expr (and_expr (shift_expr (term (factor (power (atom (NUMBER 10)))))))))))))))) (RPAR \)))))))))))))))) (COLON :) (suite (stmt (simple_stmt (small_stmt (expr_stmt (testlist (test (or_test (and_test (not_test (comparison (expr (xor_expr (and_expr (shift_expr (term (factor (power (atom (NAME x))))))))))))))) (augassign (PLUSEQUAL +=)) (testlist (test (or_test (and_test (not_test (comparison (expr (xor_expr (and_expr (shift_expr (term (factor (power (atom (NAME i))))))))))))))))))))))) (stmt (simple_stmt (small_stmt (flow_stmt (return_stmt (NAME return) (testlist (test (or_test (and_test (not_test (comparison (expr (xor_expr (and_expr (shift_expr (term (factor (power (atom (NAME x)))))))))))))))))))))))))

Python → Scales

Using scheme macros, the previous python syntax tree is transformed to Scales:

```
(defun test_for (x)
        (begin
            (for-in i (range 10)
                (+= x i))
            (return x)))
```

Scales is

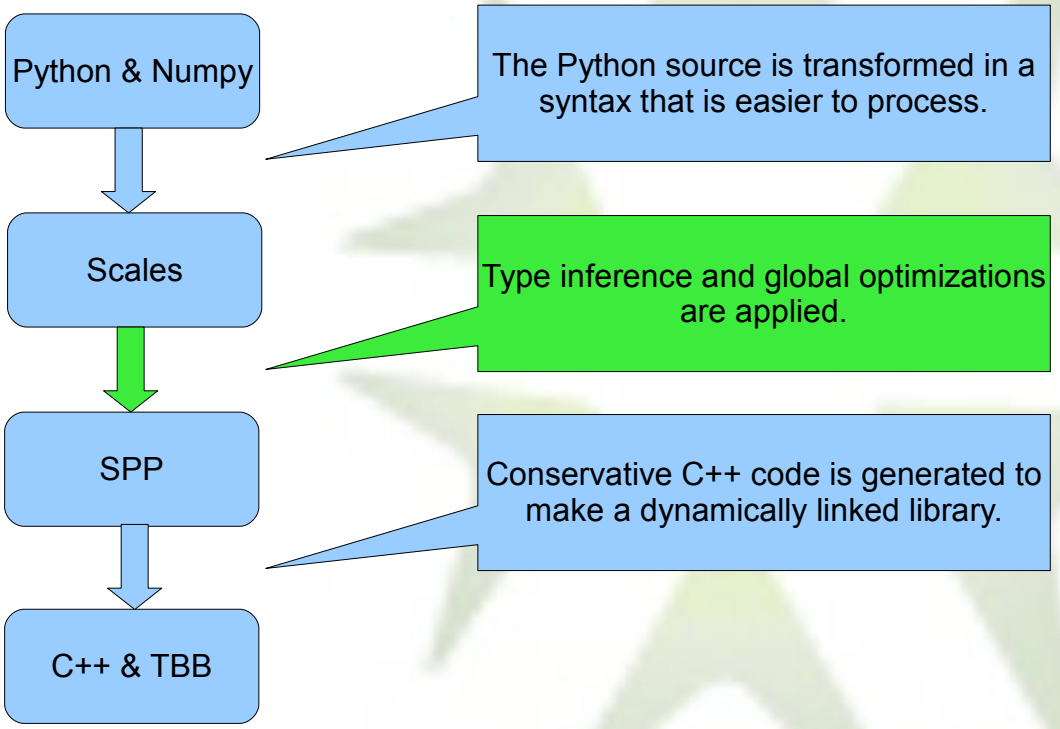
- Lispy syntax
  - with macros
- Python semantics

Scales is programmable

Scales intrinsics are user-programmable

```
(defmacro convolve_h formals
  (let ((dst (list-ref formals 0))
        (src (list-ref formals 1))
        (coefs (list-ref formals 2))
        (norm (list-ref formals 3))
        (row-name (genname "row")))
    `(p-for-in ,row-name (range (array-ref (member-ref ,src shape) 0))
               (convolve_row_h ,row-name ,dst ,src ,coefs ,norm))))
```

Scales → SPP


Python & Numpy
Scales
SPP
C++ & TBB
The Python source is transformed in a syntax that is easier to process.
Type inference and global optimizations are applied.
Conservative C++ code is generated to make a dynamically linked library.

Scales → SPP

From :

```
(defun test_for (x)
        (begin
            (for-in i (range 10)
                (+= x i))
            (return x)))
```

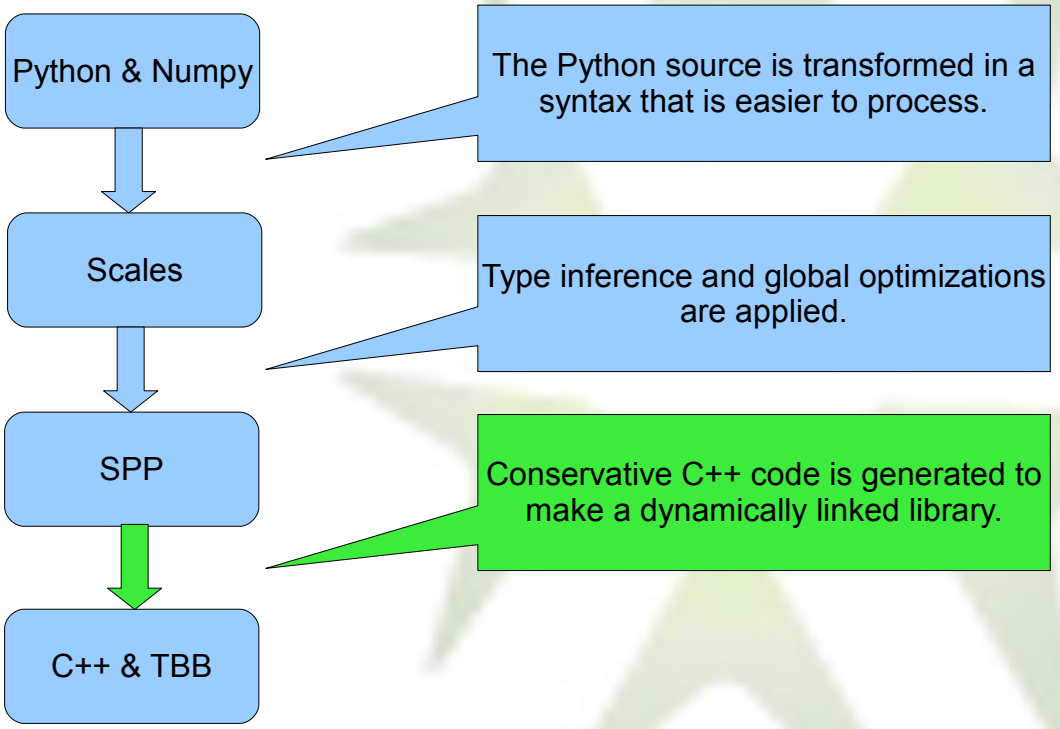
When called with typeof(x) == int32, transformed to:

```
(sppx
 (sppx-function int32 test_for ((int32 x))
            (seq (for (int i 0) (< i 10) (++ i)
                (+= x i))
             (return x))))
```

SPP

- Lispy (programmable) syntax
- Semantics of C++

SPP → C++


Python & Numpy
Scales
SPP
C++ & TBB
The Python source is transformed in a syntax that is easier to process.
Type inference and global optimizations are applied.
Conservative C++ code is generated to make a dynamically linked library.

SPP → C++

```
(function float test_array()
        (decl float[10] array)
        (= (array-ref array 3) 0.0)
        (return (array-ref array 3)))
```

Becomes

```
float test_array ()
{
    float array[10];
    (array[3] = 0.0);
    return array[3];
};
```

SPP is programmable

```
(spp
 (let-syntax ((for-range
               (syntax-rules ()
                 ((_ type var begin end body ...)
                  (for (type var begin) (< var end) (++ var) body ...)))))

   (function int main ()
             (decl float[256] array)
             (for-range int i 0 256
                        (= (array-ref array i) 0.0))
             (return 0))))
```

- Presentation of Compyle
- Examples
- The big picture & the tower of languages
- Conclusion

Conclusion

The challenges of the design

- Error messages : tracing the mistake to the original python source line
- Trying to keep Scales from becoming too pythonic
- Steep learning curve

Conclusion

The benefits of the design

- Easy to build good tests
- Creation of languages that are valuable on their own
- Strong separation between modules, clean interface
  - We could have a mix&match of backends & frontends
- A programming language is a rich metaphor, with a strong theorical basis
  - features are built a the right level of abstraction