regex

Matt Might

University of Utah

matt.might.net

source: xkcd.com

Join the google group!

Extra TA: Petey Aldous

Today

- Research advertisement
- What is lexical analysis?
- What's a regular expression?

Help wanted

Problems

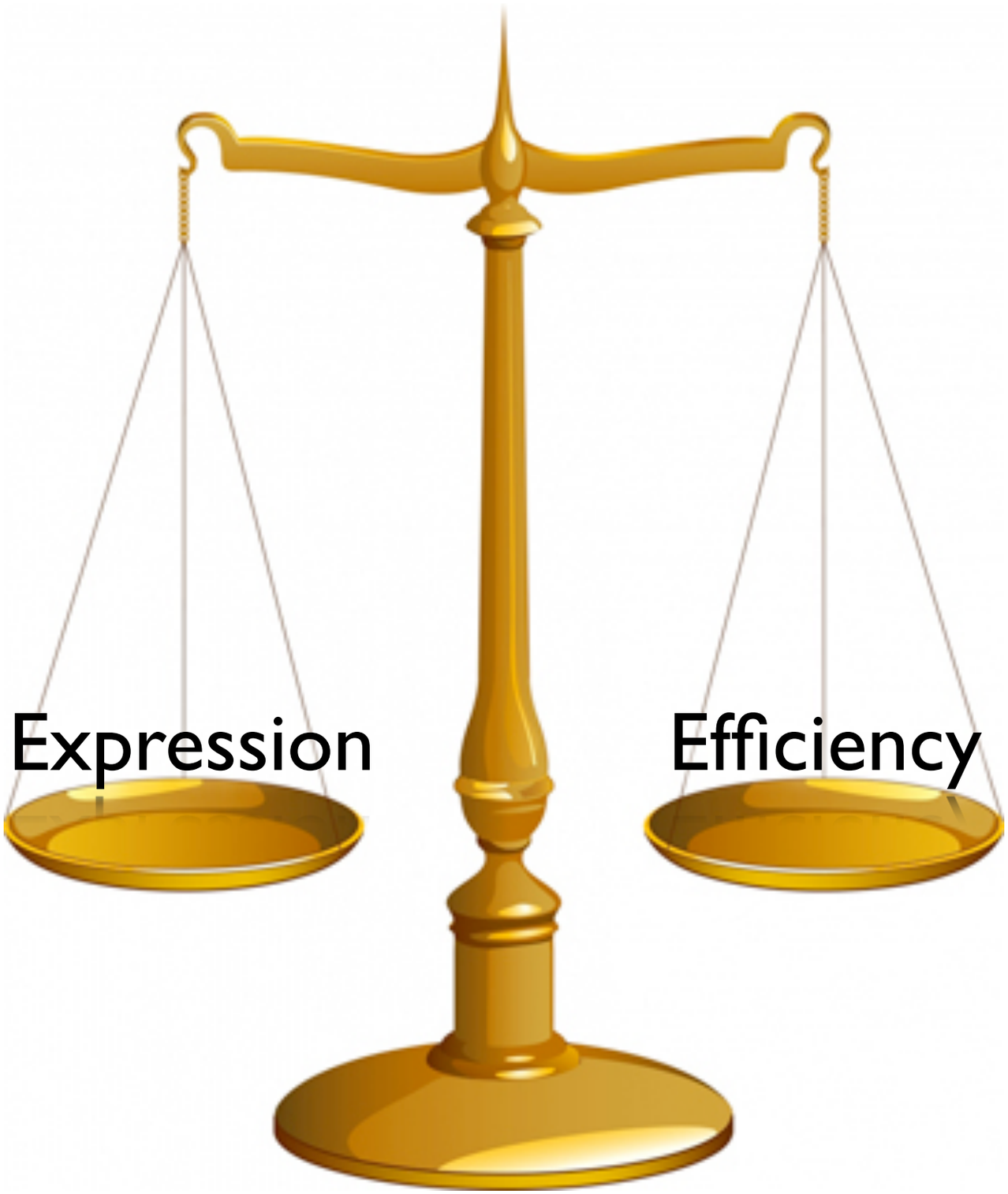
WANTED

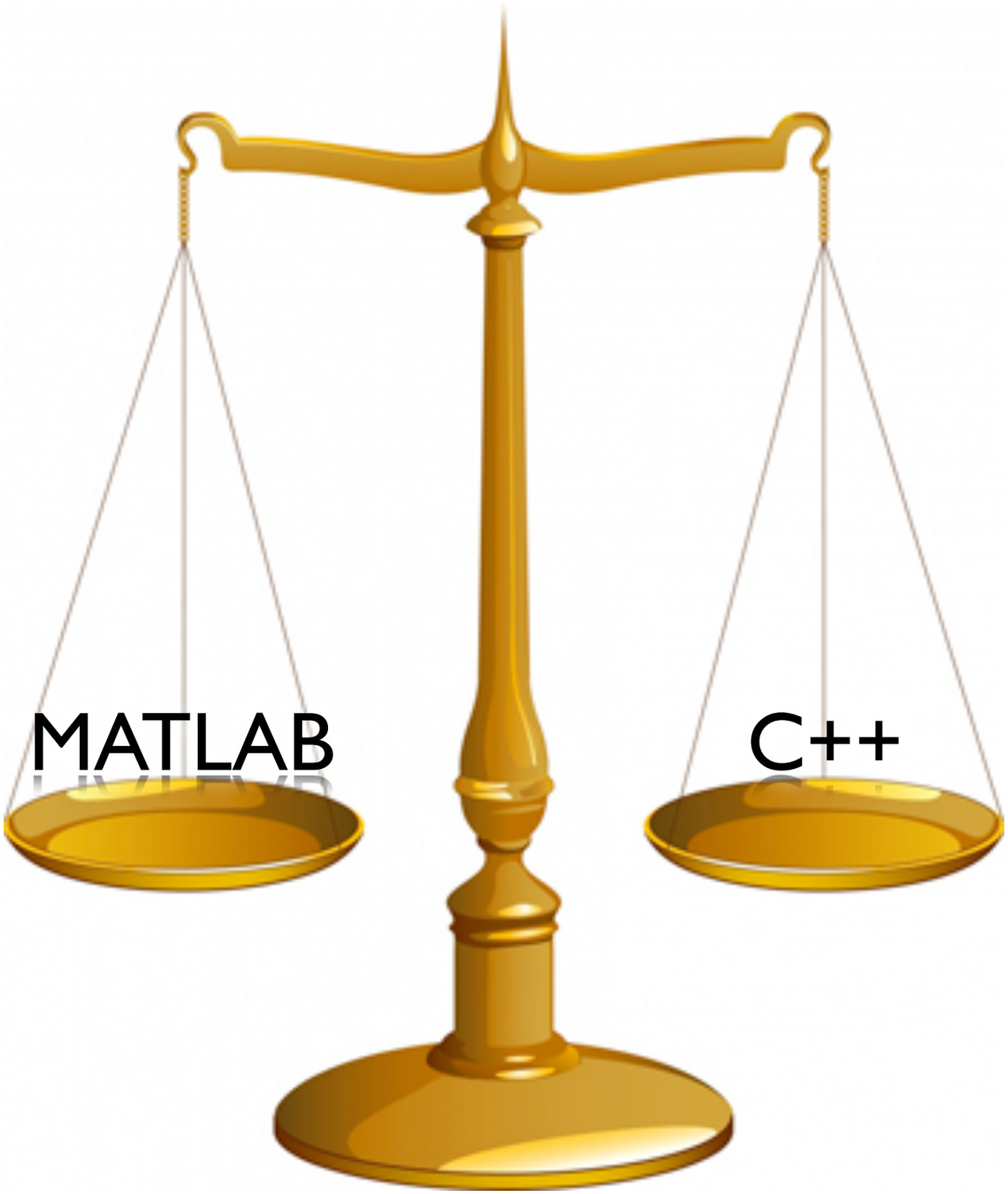
A language that accepts:

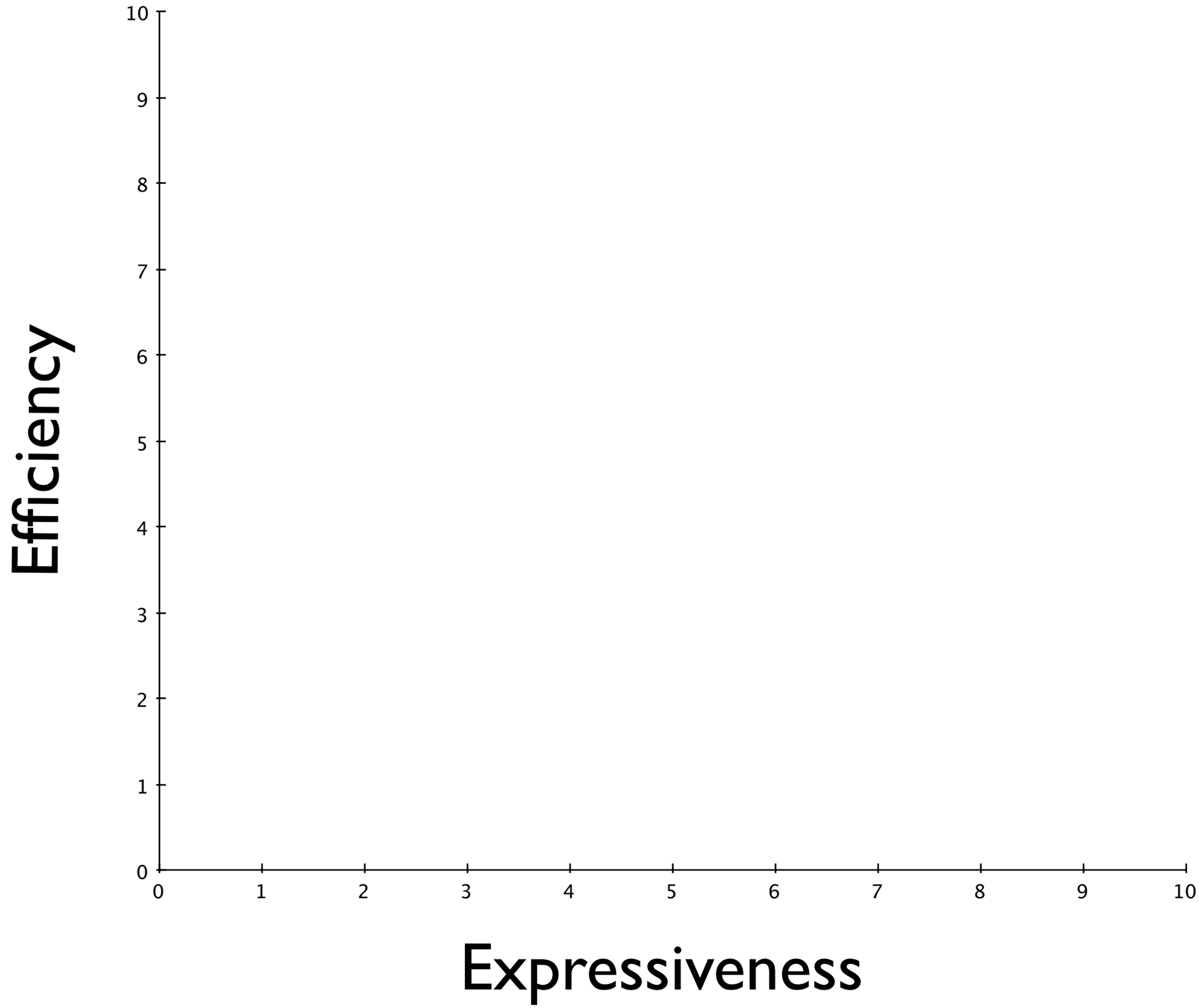
$$
\begin{aligned}
\frac{\partial \rho}{\partial t} &= -\nabla \cdot (\rho \mathbf{u}), \\
\frac{\partial (\rho \mathbf{u})}{\partial t} &= -\nabla \cdot (\rho \mathbf{u} \otimes \mathbf{u}) + \nabla \cdot \boldsymbol{\tau} - \nabla p + \rho \mathbf{g}, \\
\frac{\partial (\rho e_0)}{\partial t} &= -\nabla \cdot [\mathbf{u} (\rho e_0 + p)] + \nabla \cdot (\boldsymbol{\tau} \cdot \mathbf{u}) - \nabla \cdot \mathbf{q} + \rho \mathbf{g} \cdot \mathbf{u}, \\
\frac{\partial (\rho Y_i)}{\partial t} &= -\nabla \cdot (\rho Y_i \mathbf{u}) - \nabla \cdot \mathbf{J}_i + W_i \omega_i,
\end{aligned}
$$

but performs like this:

```
Field::const iterator ia = a.begin();
Field::const iterator ib = b.begin();
for(Field::iterator ic = c.begin();
    ic != c.end(); ++ic, ++ia, ++ib) {
 *ic = *ia + sin(*ib);
};
```

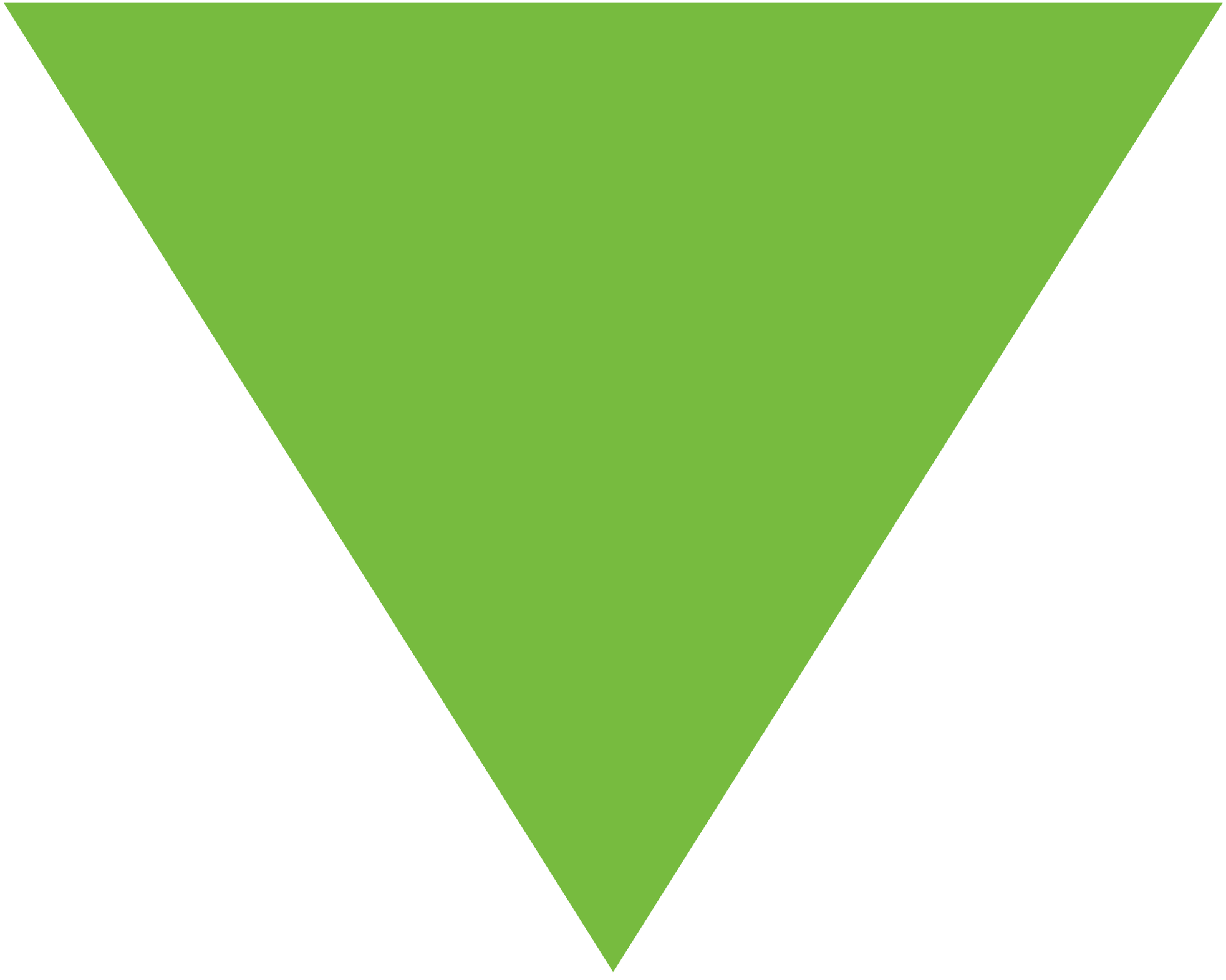

Expression
Efficiency

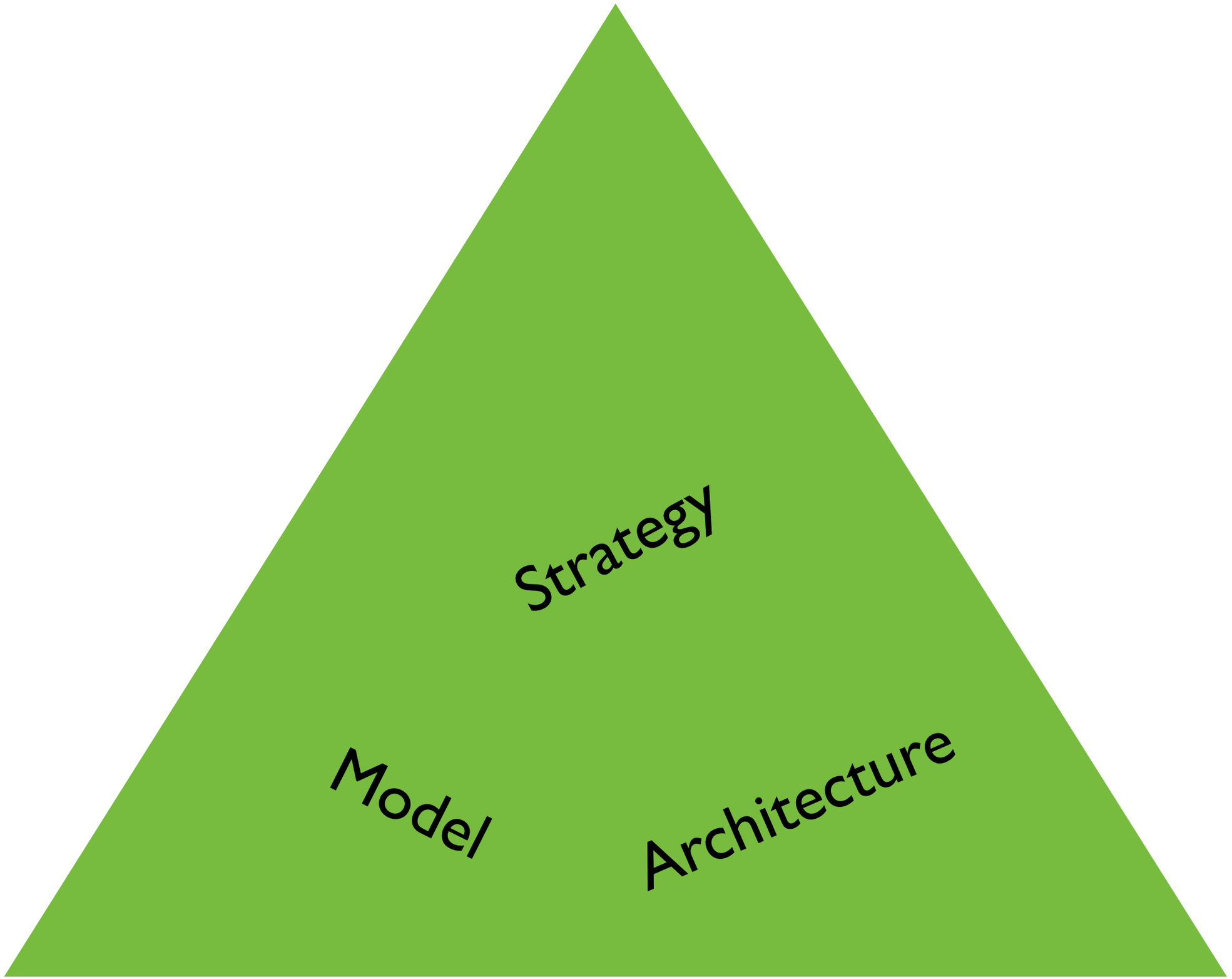

MATLAB
C++


Efficiency
Expressiveness
0
1
2
3
4
5
6
7
8
9
10

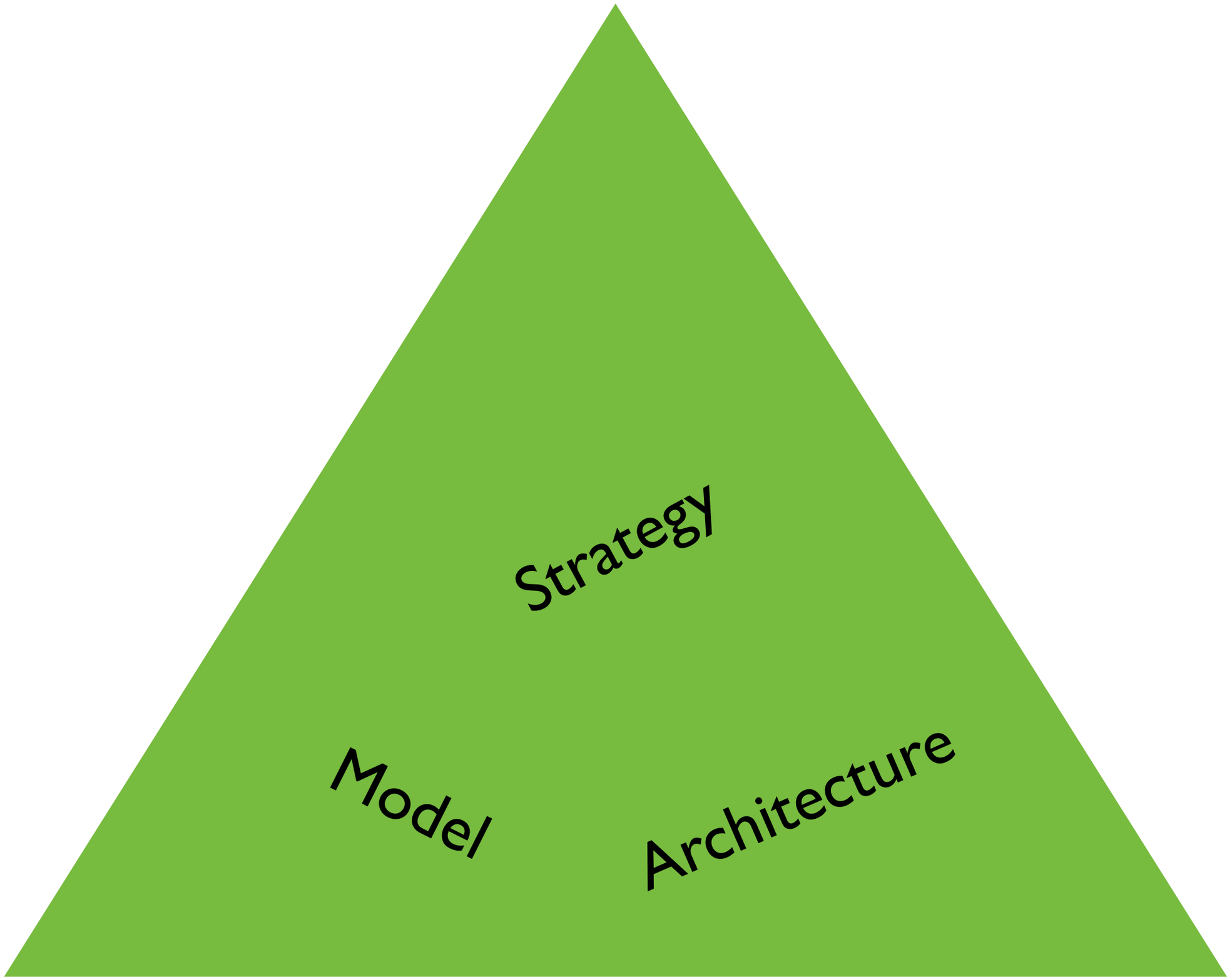

Efficiency
10
9
8
7
6
5
4
3
2
1
0
0
1
2
3
4
5
6
7
8
9
10
Expressiveness

Also...




Strategy
Model
Architecture

How do we solve both?


Strategy
Model
Architecture

Model

Architecture

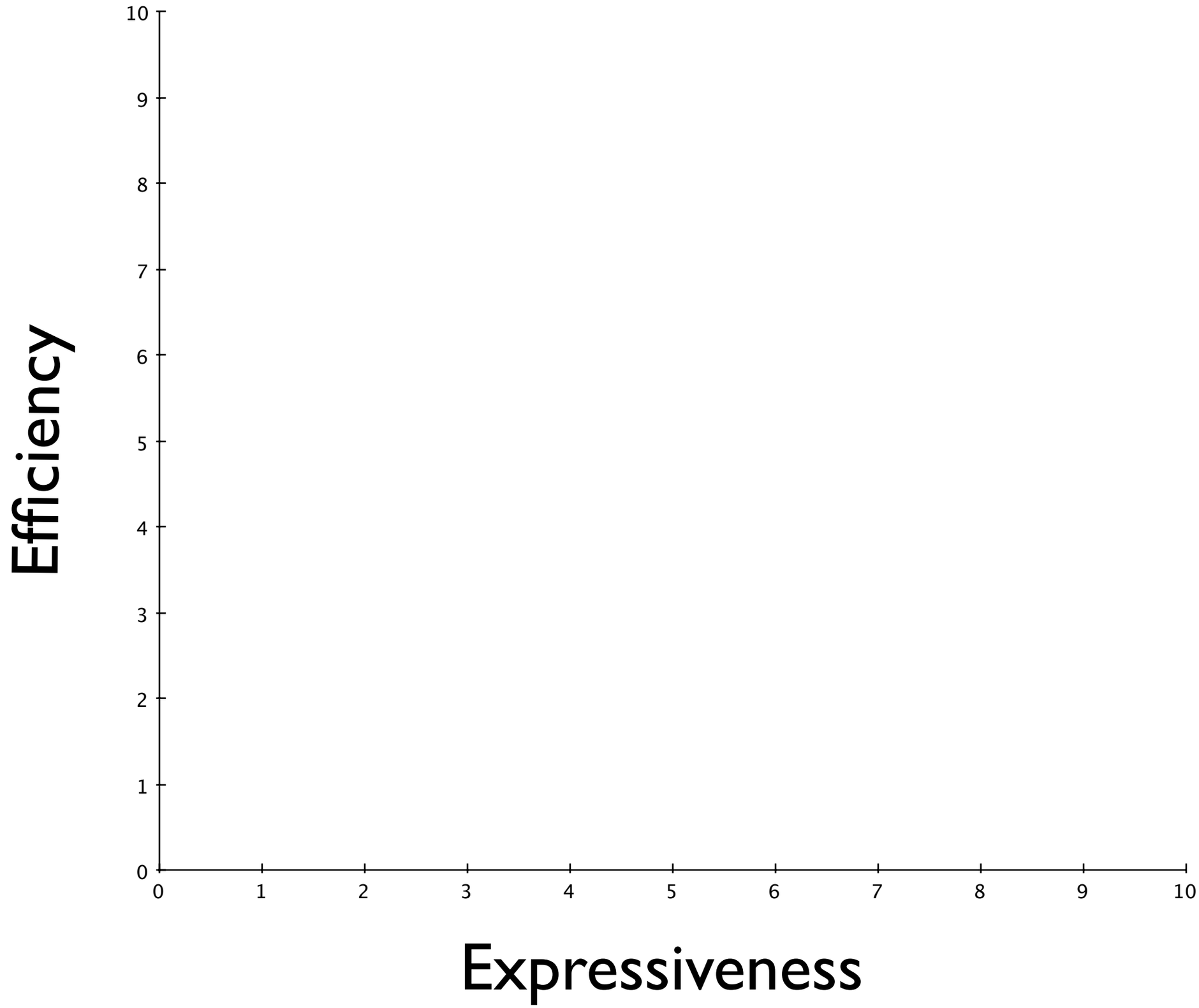
Strategy

Model

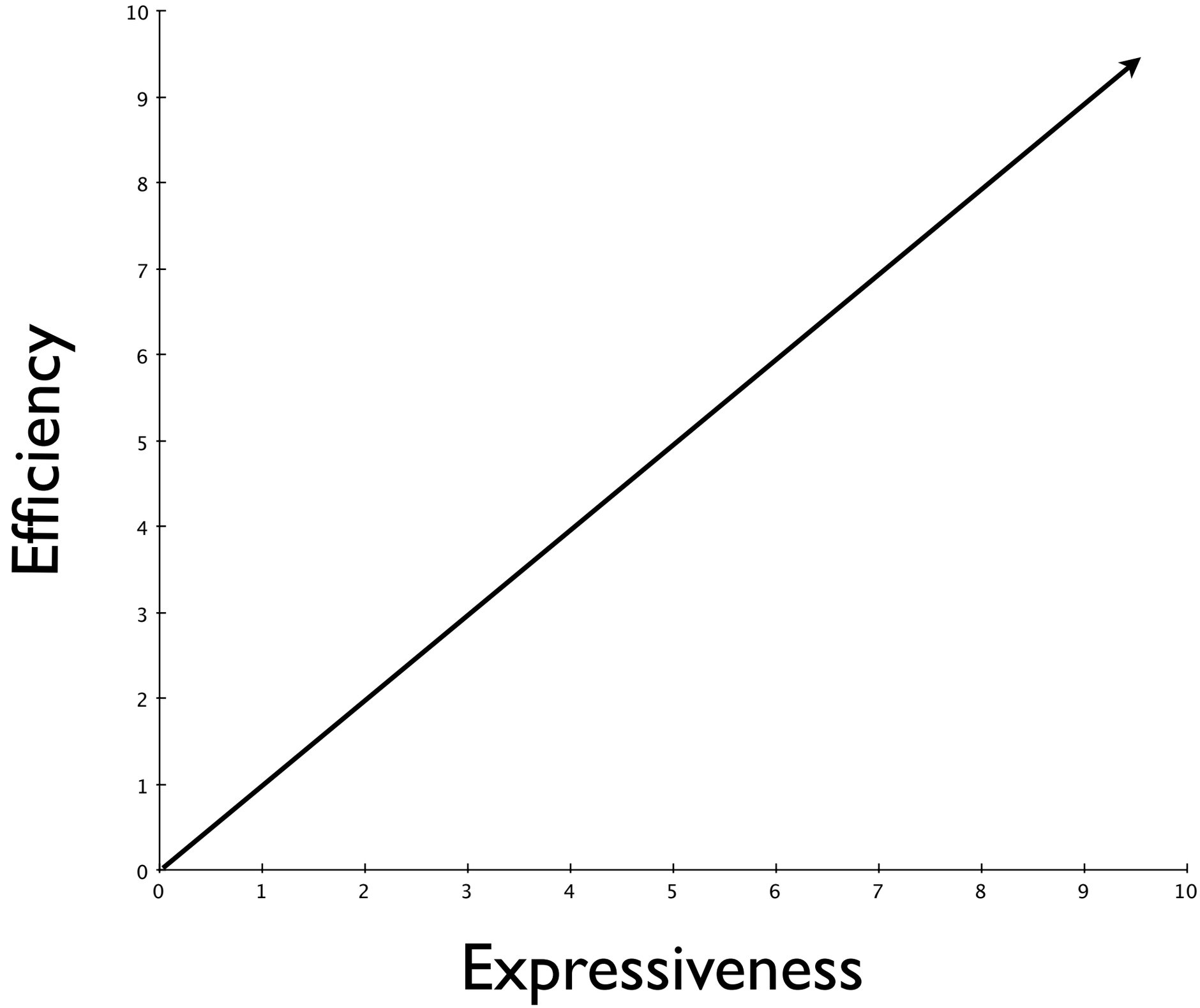
DSL

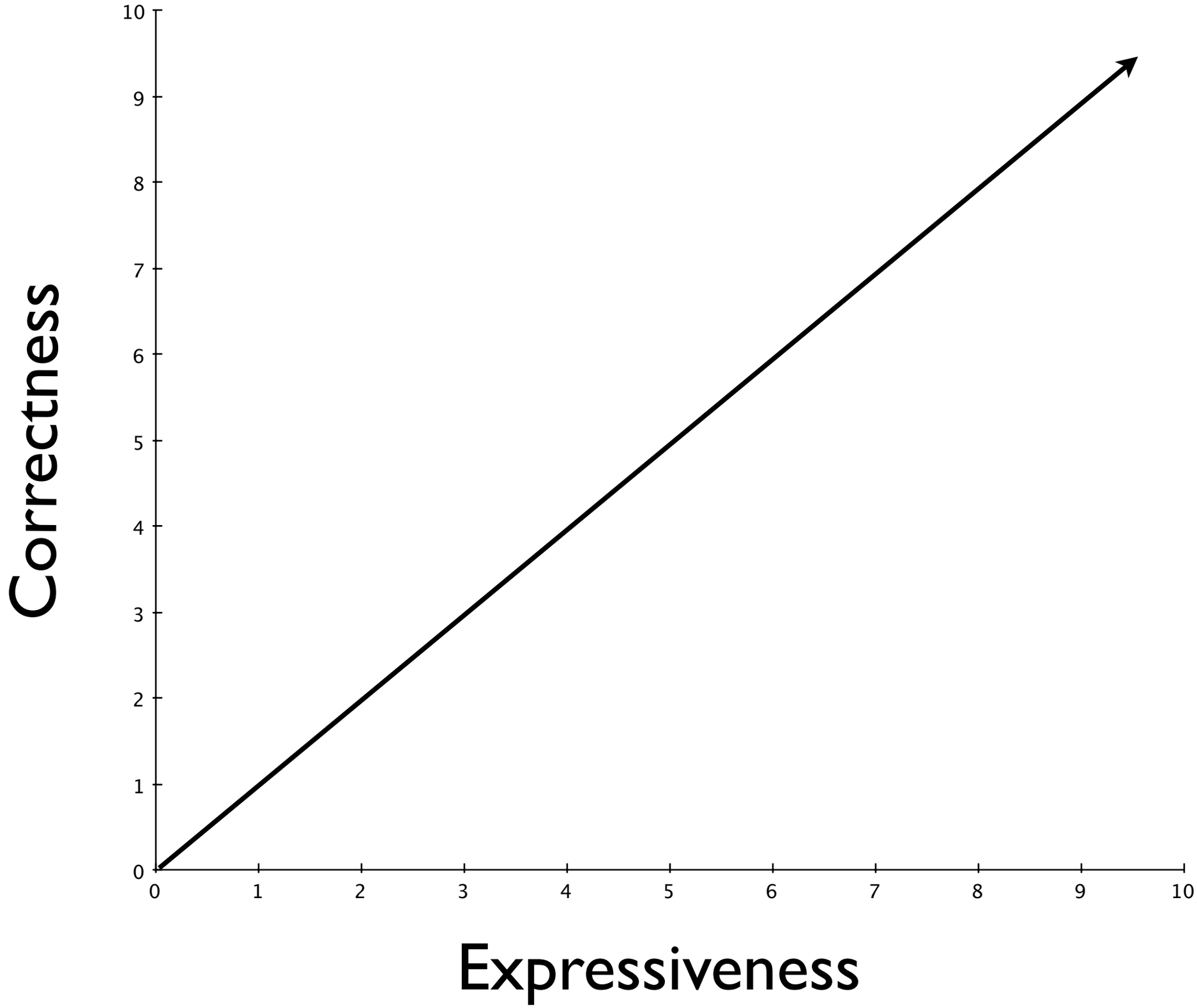
Architecture

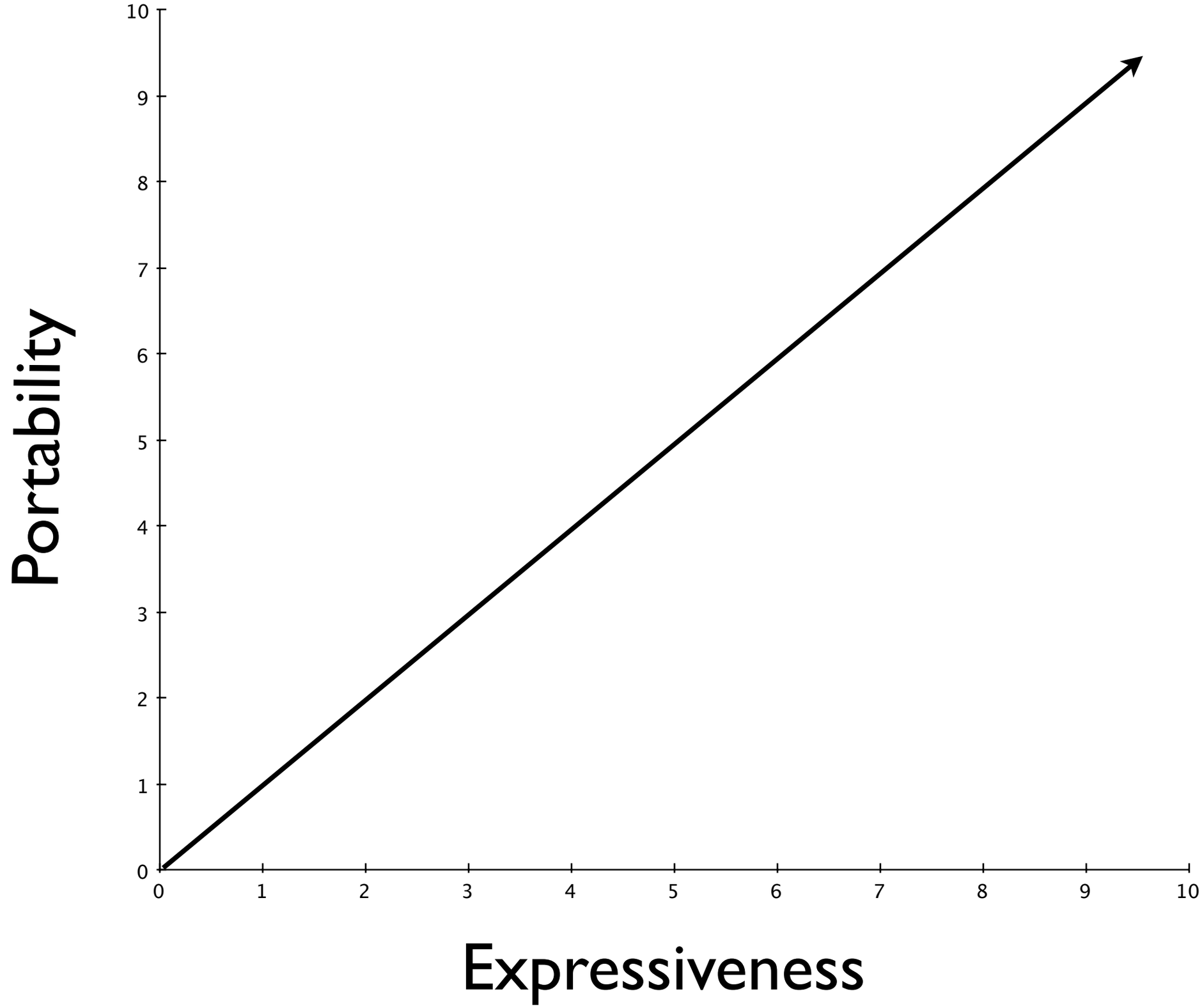
Strategy


Efficiency
Expressiveness
0
1
2
3
4
5
6
7
8
9
10


Efficiency
10
9
8
7
6
5
4
3
2
1
0
0
1
2
3
4
5
6
7
8
9
10
Expressiveness


Efficiency
Expressiveness
0
1
2
3
4
5
6
7
8
9
10


Correctness
10
9
8
7
6
5
4
3
2
1
0
0
1
2
3
4
5
6
7
8
9
10
Expressiveness


Portability
Expressiveness
0
1
2
3
4
5
6
7
8
9
10

But...

Legacy code matters.

C++ Template Metaprogramming

Domain Specific Languages

Embedded Domain Specific Languages

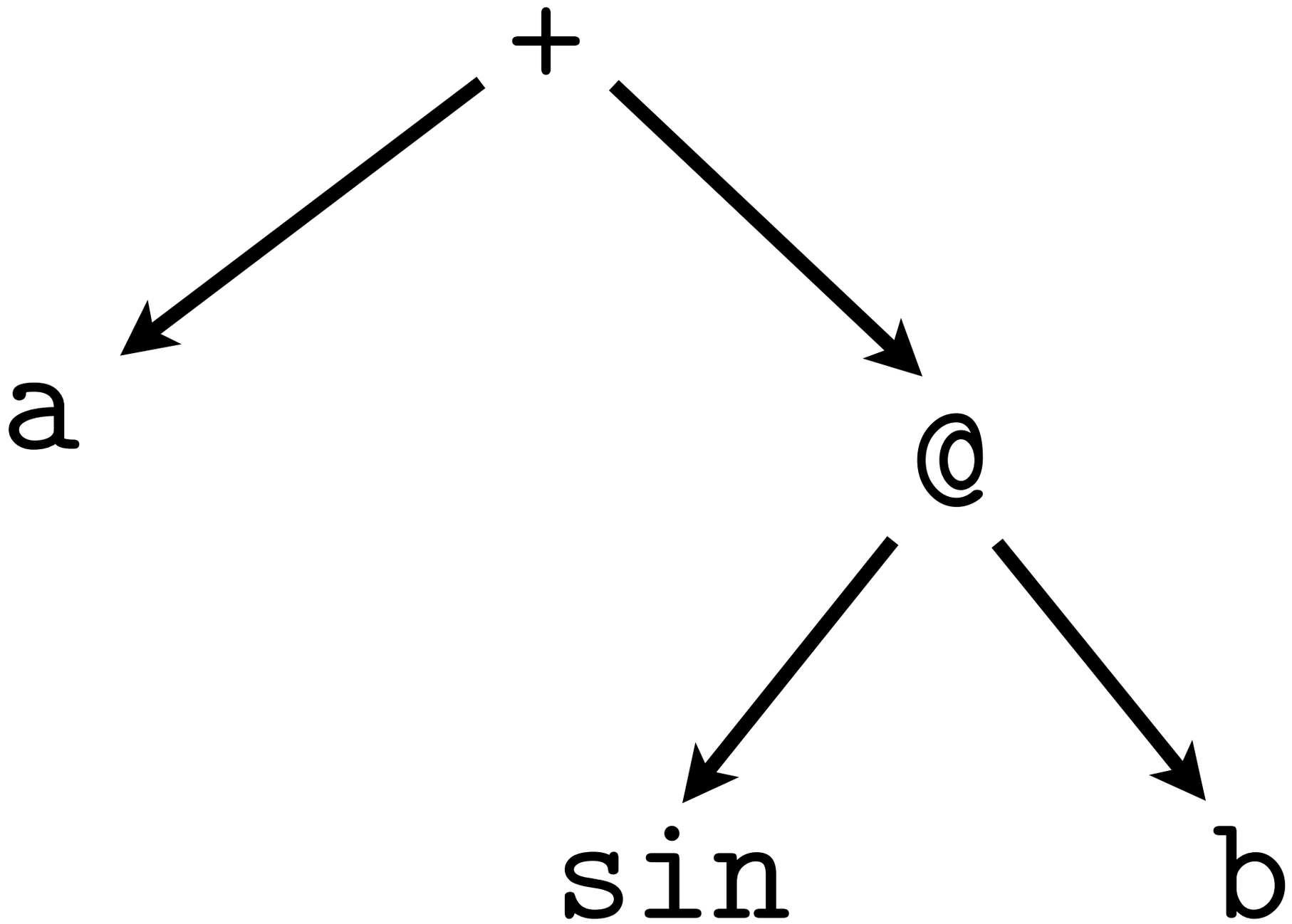
Example

$$\vec{c} = \vec{a} + \vec{sin}(\vec{b})$$

```
c <<= a + sin(b);
```

$$a + @$$

$$\sin b$$


+
a
@
sin
b

$$\vec{c} = \vec{a} + \vec{sin}(\vec{b})$$

```
c <<= a + sin(b); ⇒
```

Thread 1

```
Field::const_iterator ia1 = a1.begin();
Field::const_iterator ib1 = b1.begin();
for(Field::iterator ic1 = c1.begin();
    ic1 != c1.end();
    ++ic1, ++ia1, ++ib1) {
  *ic1 = *ia1 + sin(*ib1);
};
```

⋮

```
…
```

Thread *n*

```
Field::const_iterator ian = an.begin();
Field::const_iterator ibn = bn.begin();
for(Field::iterator icn = cn.begin();
    icn != cn.end();
    ++icn, ++ian, ++ibn) {
  *icn = *ian + sin(*ibn);
};
```

Needed

- Need: Very good with C++
- Want: Template meta-programming
- Want: Comfort with syntax trees

What is lexical analysis?

Characters in;

tokens out.

```
function id(x)
{
  return x ;  // comment
}
```

FUNCTION

```
id(x)
{
  return x ;  // comment
}
```

FUNCTION

IDENT(id)

```
(x)
{
  return x ;  // comment
}
```

```
                                        FUNCTION
                                        IDENT(id)
                                        LPAR

                    x)
{
  return x ;  // comment
}
```

```
                                                    FUNCTION
                                                    IDENT(id)
                                                    LPAR
                                                    IDENT(x)
                        )
{
  return x ;  // comment
}
```

```
                                    FUNCTION
                                    IDENT(id)
                                    LPAR
                                    IDENT(x)
                                    RPAR
{
  return x ;  // comment
}
```

```
function id ( x ) {  // comment
  return x ;
}
```

FUNCTION

IDENT(id)

LPAR

IDENT(x)

RPAR

LBRACE

```
function id ( x ) {
    x ; // comment
}
```

```
FUNCTION
IDENT(id)
LPAR
IDENT(x)
RPAR
LBRACE
RETURN
```

```
function id(x) {
                ; // comment
}
```

```
FUNCTION
IDENT(id)
LPAR
IDENT(x)
RPAR
LBRACE
RETURN
IDENT(x)
```

```
                                    FUNCTION
                                    IDENT(id)
                                    LPAR
                                    IDENT(x)
                                    RPAR
                                    LBRACE
                  // comment
}                                   RETURN
                                    IDENT(x)
                                    SEMI
```

```
}
```

FUNCTION

IDENT(id)

LPAR

IDENT(x)

RPAR

LBRACE

RETURN

IDENT(x)

SEMI

FUNCTION

IDENT(id)

LPAR

IDENT(x)

RPAR

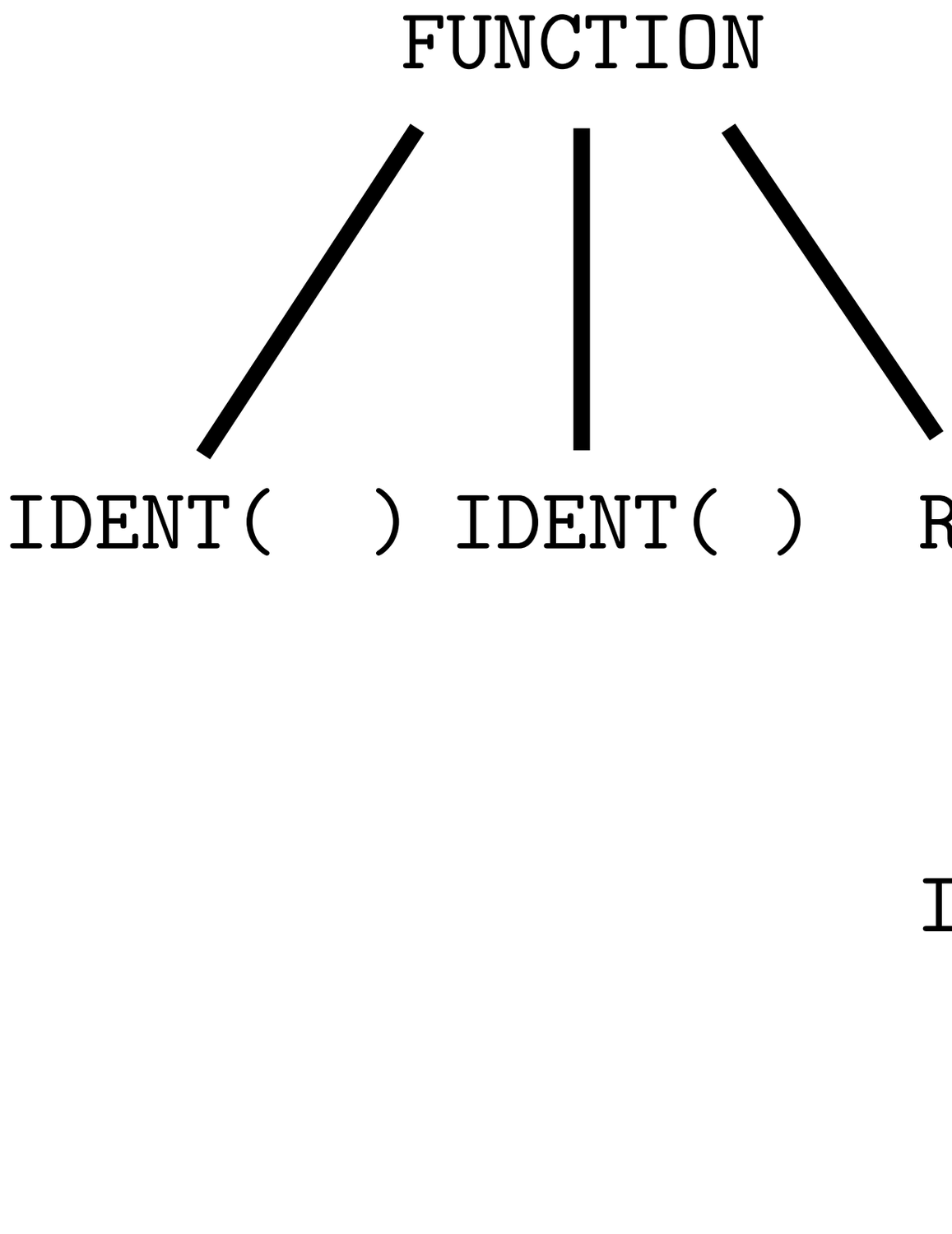
LBRACE

RETURN

IDENT(x)

SEMI

RBRACE



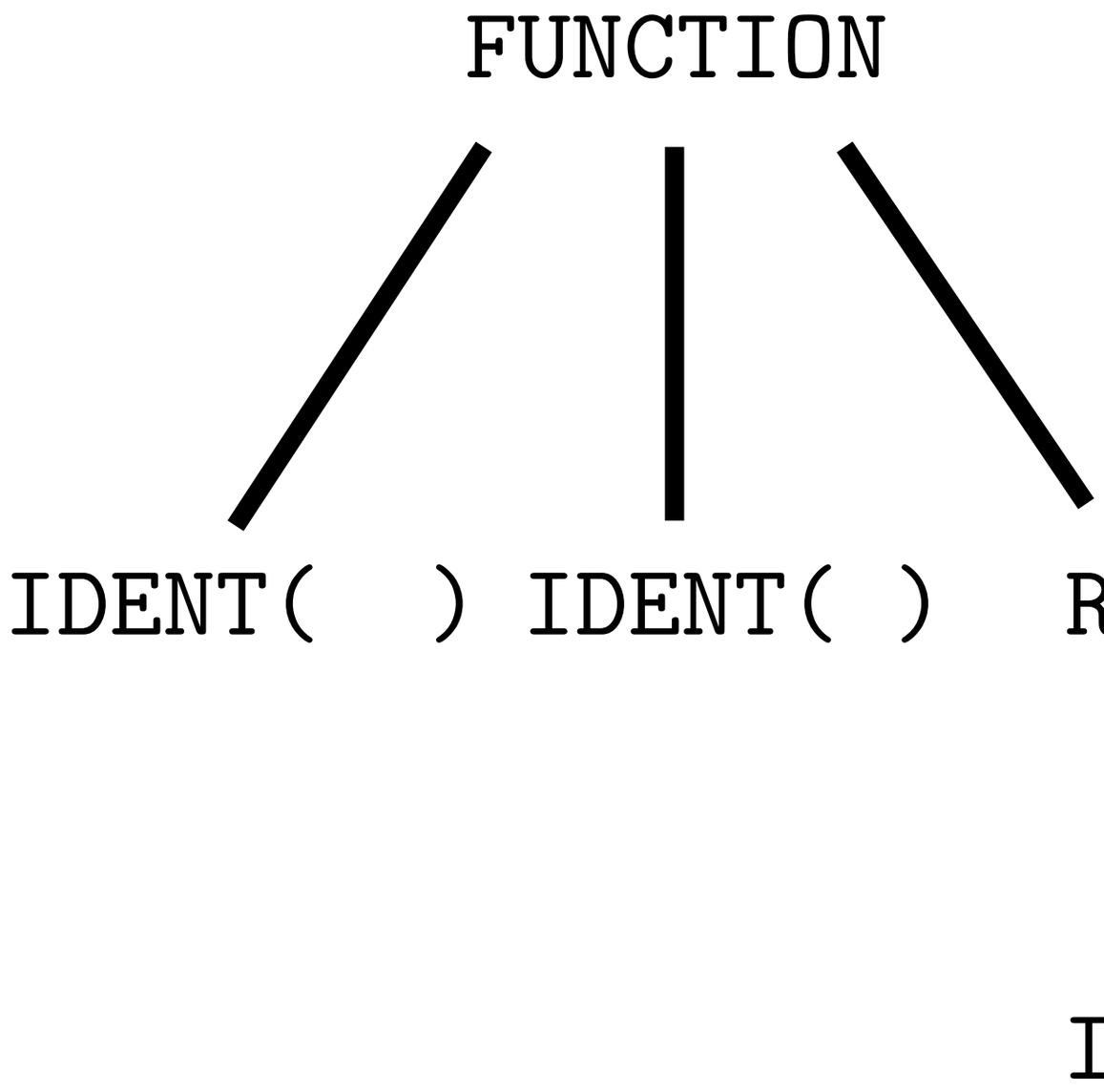
LPAR

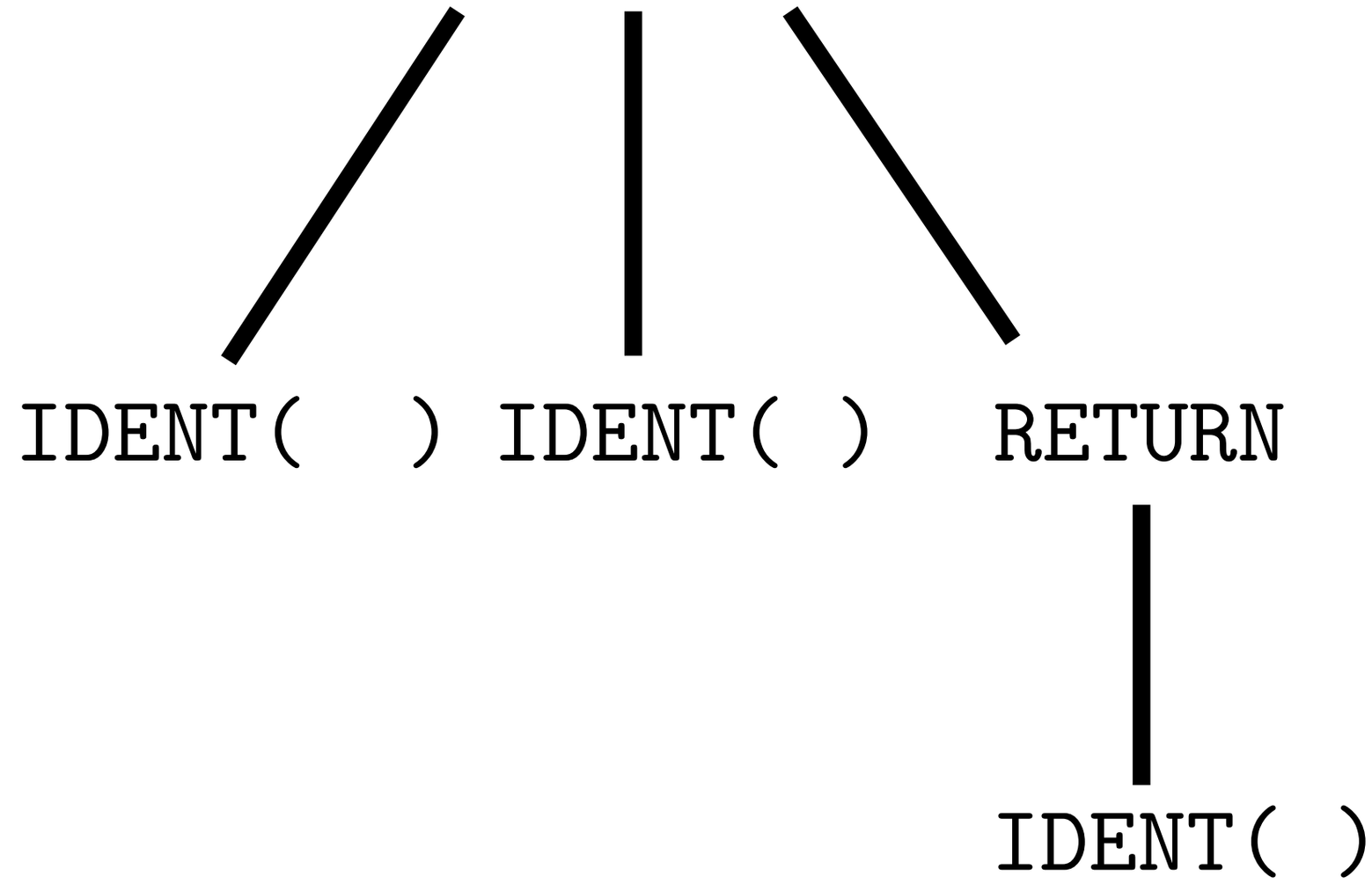
RPAR

LBRACE

SEMI

RBRACE


FUNCTION
IDENT( ) IDENT( ) RETURN
IDENT( )


FUNCTION
IDENT( ) IDENT( ) RETURN
IDENT( )

How do you define a lexer?

Lexical specification

- Token types
- Whitespace
- Keywords
- Operators
- Comments
- Identifiers
- Punctuation

How do you define tokens?

regex

- Identifiers match `[A-Za-z_][A-Za-z0-9_]*`
- Delimiters match `[();]`
- Operators match `[-*+/^=]`
- Integers match `-?([1-9][0-9]*|0)`
- Whitespace is ignored
- Comments match `^#[^\n]*`

regex is a pattern language.

(a family of them, actually)

regex languages

- math
- unix
- pcre

Regexen match strings.

Minimal regex

Pattern forms

Character: *c*

ex: a matches a

Sequence: $p_1 p_2$

ex: `ab` **matches** `ab`

Choice: $p_1 \mid p_2$

ex: `a|b` **matches** `a`

ex: `a|b` **matches** `b`

Repetition: $p*$

ex: a* matches a

ex: a* matches aa

ex: a* matches aaa

ex: a* matches

Subpattern: (p)

ex: `(ab)*` **matches** `ab`

ex: `(ab)*` matches `abab`

ex: `ab*` **matches** `abbbbb`

Sugared regex

Option: p?

ex: a? matches a

ex: a? matches

$$p? = (p|)$$

Strict repetition: $p+$

ex: a+ matches a

ex: a+ !matches

$$p+ = pp*$$

Char set: $[c_1 c_2 \ldots]$

ex: `[ab]` **matches** `a`

ex: `[ab]` **matches** `b`

Inverse char set: [^$c_1 c_2 \ldots$]

Ranged char set: [c_1-c_2]

ex: `[a-c]` **matches** `b`

ex: `[^ab]` **matches** `c`

Warning: `echo $LANG`

Solution: `LANG=C`

Any character: .

ex: . matches a

ex: . matches c

ex: . matches b

Start of line/string: ^

End of line/string: $

Repeat: $p\{n\}$

ex: $p\{3\} = ppp$

Bounded repeat: $p\{n,m\}$

ex: $p\{2,4\} = pp|ppp|pppp$

Irregular expressions

Submatch: (p) and \1

ex: `(.)\1` **matches** `aa`

Shortest match: p+?

Dialects

Dialect: BRE

- `( )` => `\( \)`
- `{ }` => `\{ \}`
- `?` => `\?`
- `+` => `\+`
- `|` => `\|`

Dialect: BRE, ERE, PCRE

Defined char classes

BRE, ERE	PCRE
[[:word:]]	\w
[[:alpha:]]	[A-Za-z]
[[:space:]]	\s
[[:lower:]]	[a-z]
[[:upper:]]	[A-Z]
[[:digit:]]	\d
[[:punct:]]	

Define char classes

BRE, ERE	PCRE
\b	\b
[^[:space:]]	\S
[^[:word:]]	\W
[^[:digit:]]	\D

grep: examples and pitfalls

```
grep foo|bar words
```

```
grep 'foo|bar' words
```

```
grep 'foo\|bar' words
```

```
egrep 'foo|bar' words
```

```
grep '.x.n' words
```

```
grep '^.x.n$' words
```

```
grep '^(.*)\1$' words
```

```
egrep '^(.*)\1$' words
```

```
egrep '98.17.132.45' log
```

```
egrep '\b98\.17\.132\.45\b' log
```

```
egrep '([0-255]\.){3}[0-255]' log
```

```
\d|1?\d\d|2[0-4]\d|25[0-5]
```

```
\d*[02468]
```

```
perl -e 'while (<STDIN>) {
   if (/^1?$|^(11+?)\1+$/)
   { print }
}'
```

Challenge

- Match dates YYYY MM DD
- Account for leap years
  - If multiple of 400: Yes
  - If multiple of 100 but not 400: No
  - If multiple of 4 but not 100: Yes

Challenge

- Match RFC 3696 email addresses
- Never try this with lives at stake

- Identifiers match `[A-Za-z_][A-Za-z0-9_]*`
- Delimiters match `[();]`
- Operators match `[-*+/^=]`
- Integers match `-?([1-9][0-9]*|0)`
- Whitespace is ignored
- Comments match `^#[^\n]*`

Python tokens

Integer literals

```
integer          ::=  decimalinteger |
                      octinteger |
                      hexinteger | bininteger
decimalinteger   ::=  nonzerodigit digit* | "0"+
nonzerodigit     ::=  "1"..."9"
digit            ::=  "0"..."9"
octinteger       ::=  "0" ("o" | "O") octdigit+
hexinteger       ::=  "0" ("x" | "X") hexdigit+
bininteger       ::=  "0" ("b" | "B") bindigit+
octdigit         ::=  "0"..."7"
hexdigit         ::=  digit | "a"..."f" | "A"..."F"
bindigit         ::=  "0" | "1"
```

Floating point

```
floatnumber   ::=  pointfloat | exponentfloat
pointfloat    ::=  [intpart] fraction | intpart "."
exponentfloat ::=  (intpart | pointfloat) exponent
intpart       ::=  digit+
fraction      ::=  "." digit+
exponent      ::=  ("e" | "E") ["+" | "-"] digit+
```

Operators

+	-	*	**	/	//	%
<<	>>	&	|	^	~	
<	>	<=	>=	==	!=	

Questions?