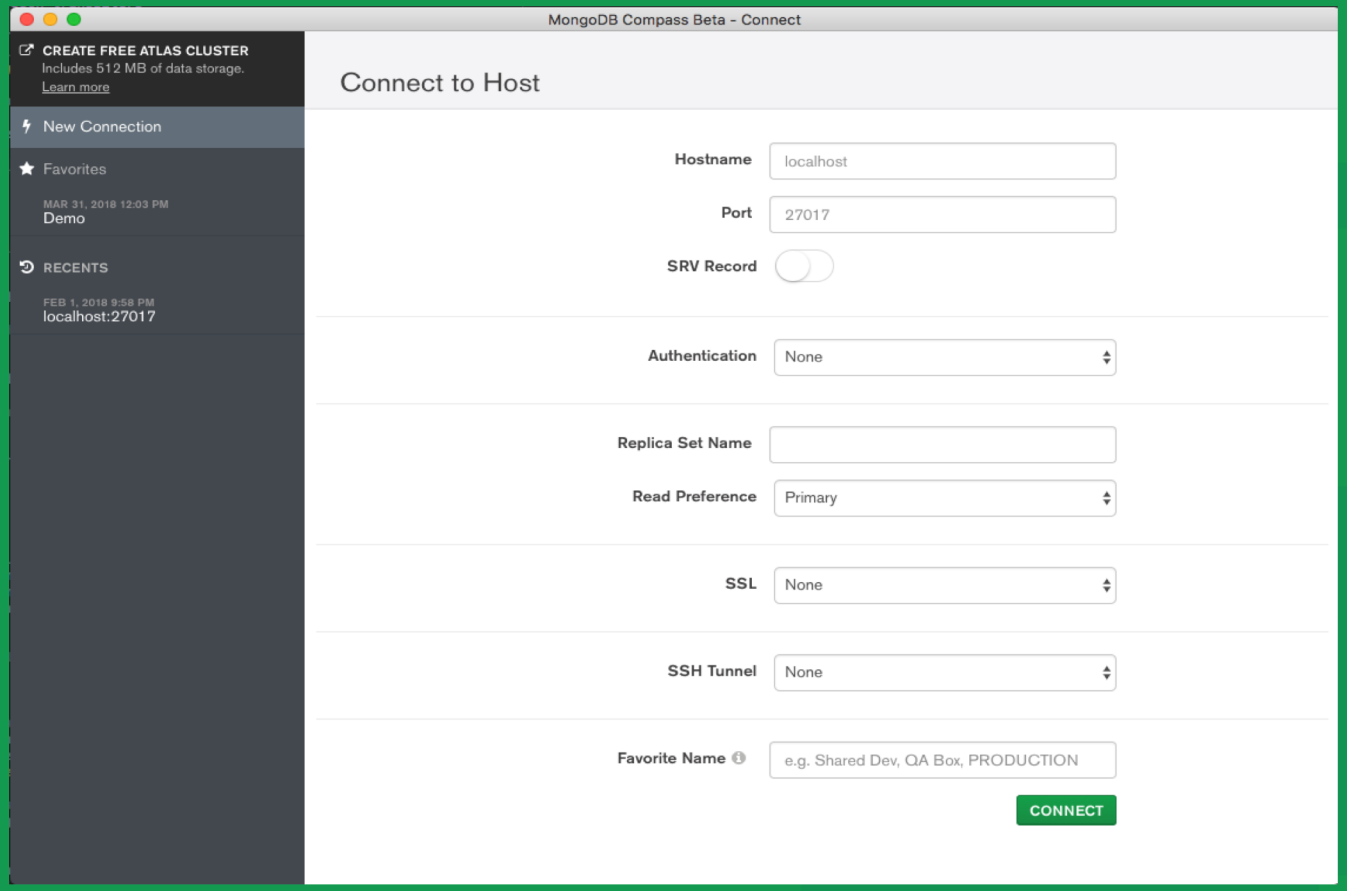
BSON-Transpilers

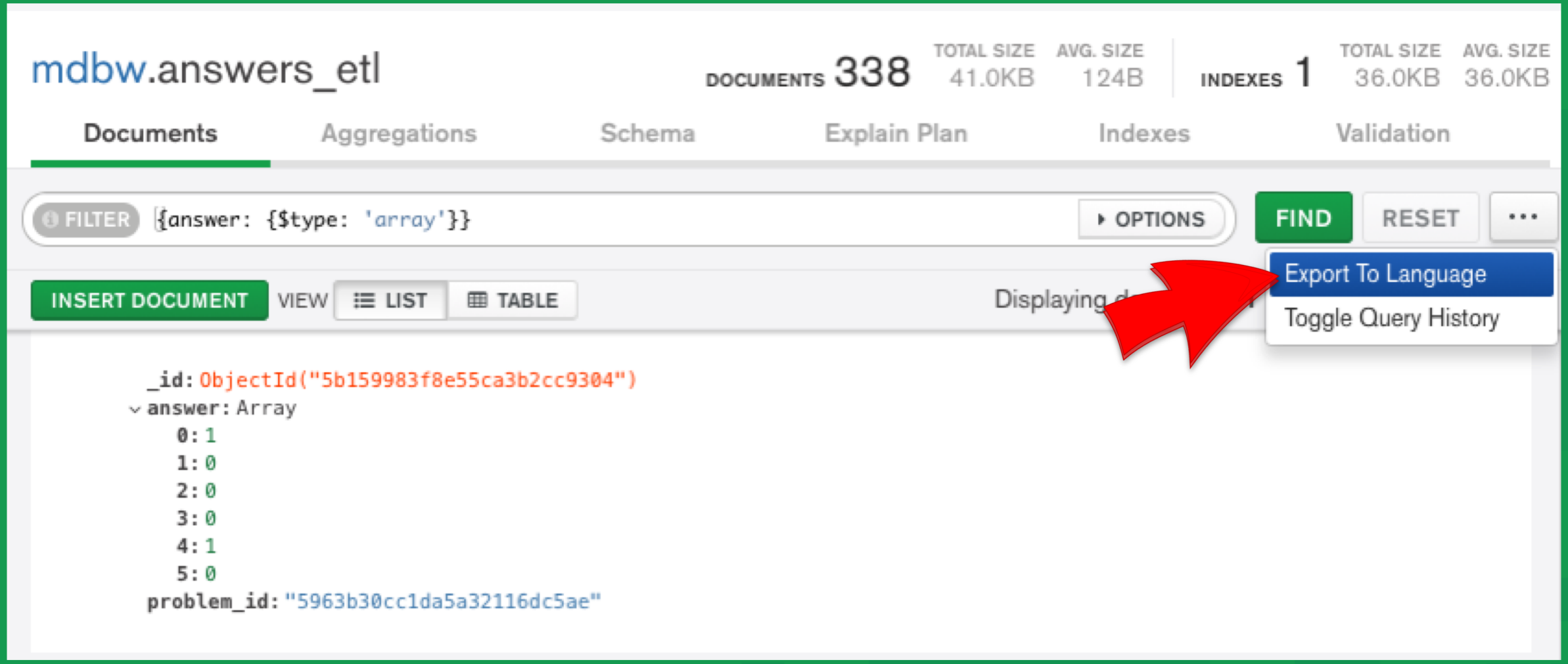
Anna Herlihy
Senior Software Engineer
Stockholm
@annaisworking

1. Feature Requirements
2. Technical Requirements
3. How to contribute!

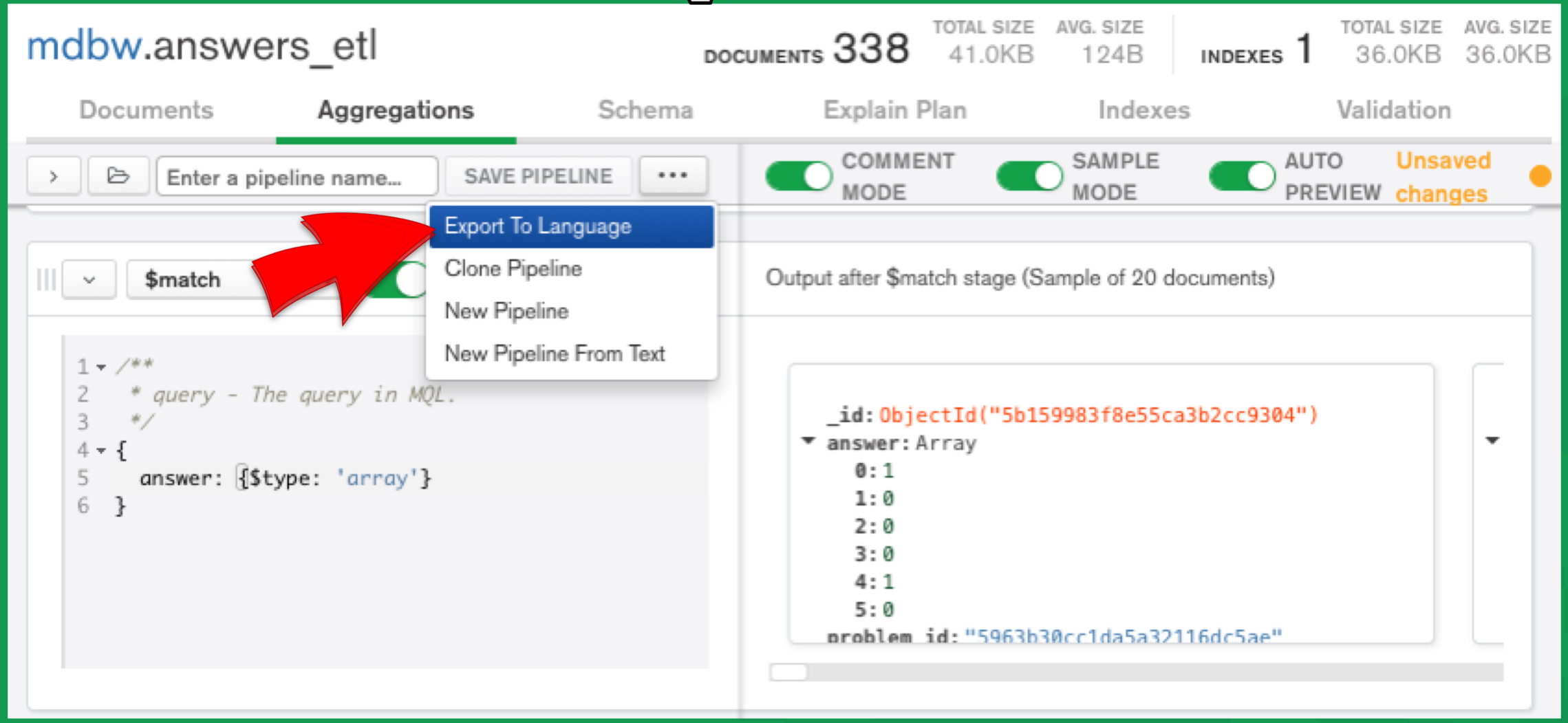
Compass = The UI for MongoDB



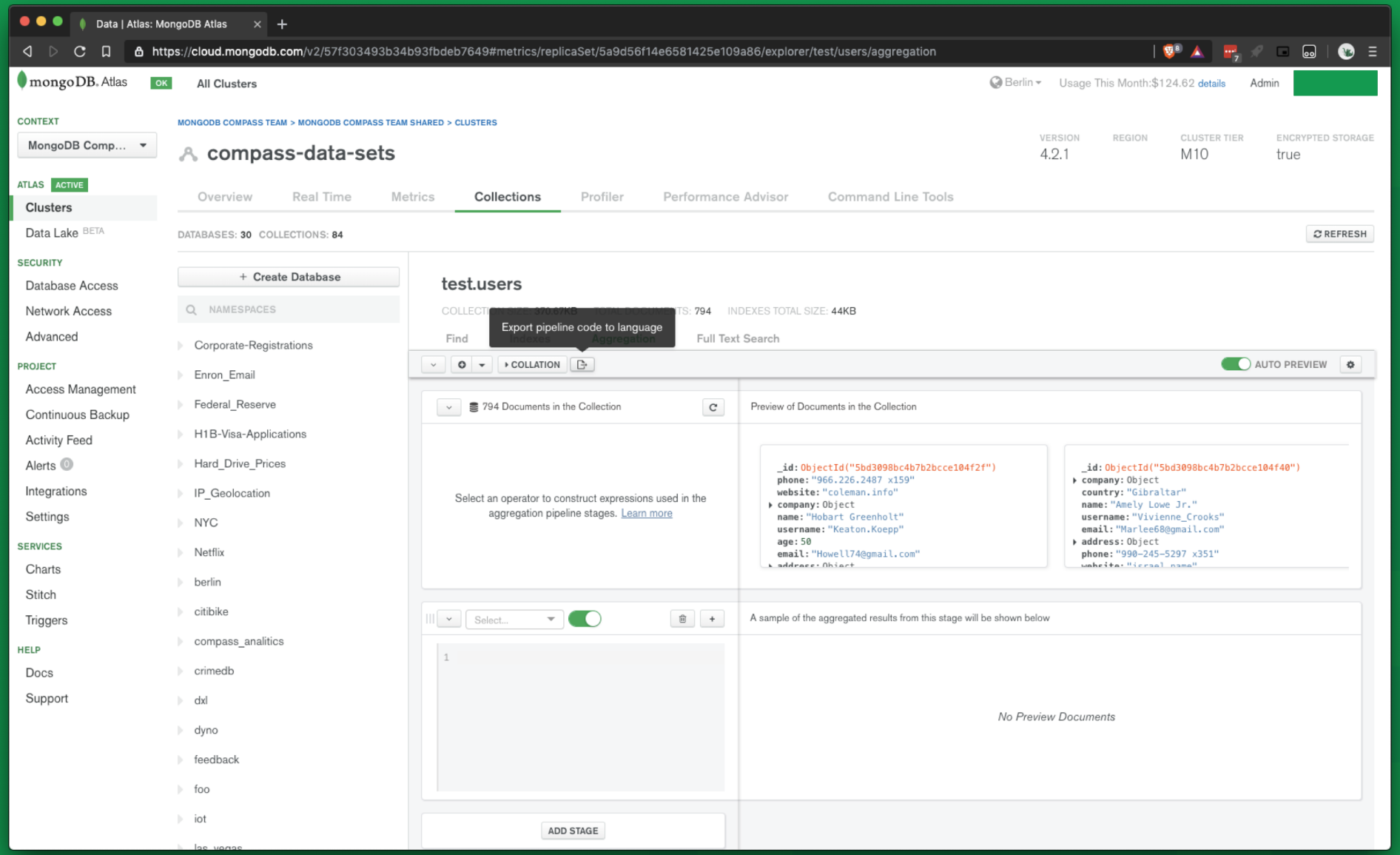
Export To Language (query)



Export To Language (aggregation)



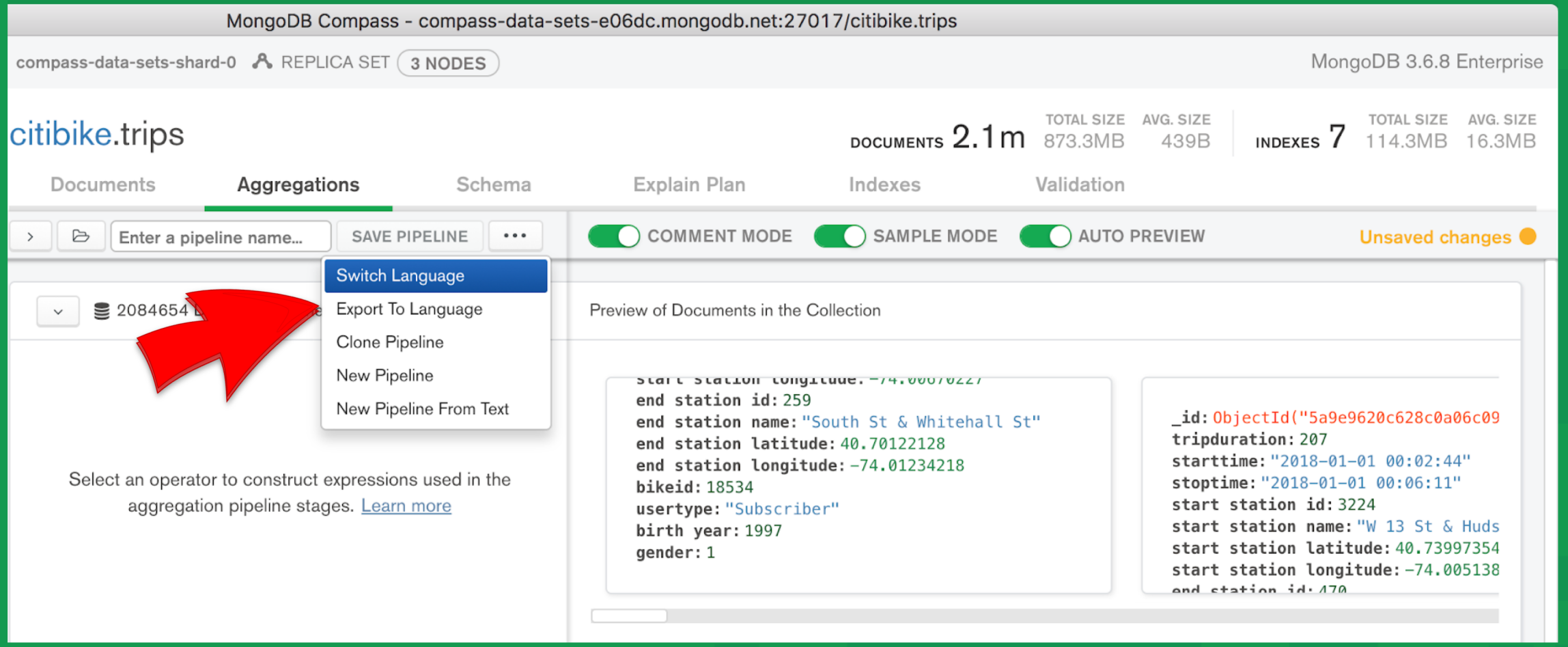
Also used on MongoDB Atlas



But wait...

Wouldn't it be great if you could just write whatever language you want, directly into Compass?

Language-Modes



BSON is a large enough subset to treat the problem as if we're parsing the entire language syntax

Requirements

Accept query or aggregation in any language

Export query or aggregation to any language


Any language
to
any language
translation!

Possible Approaches

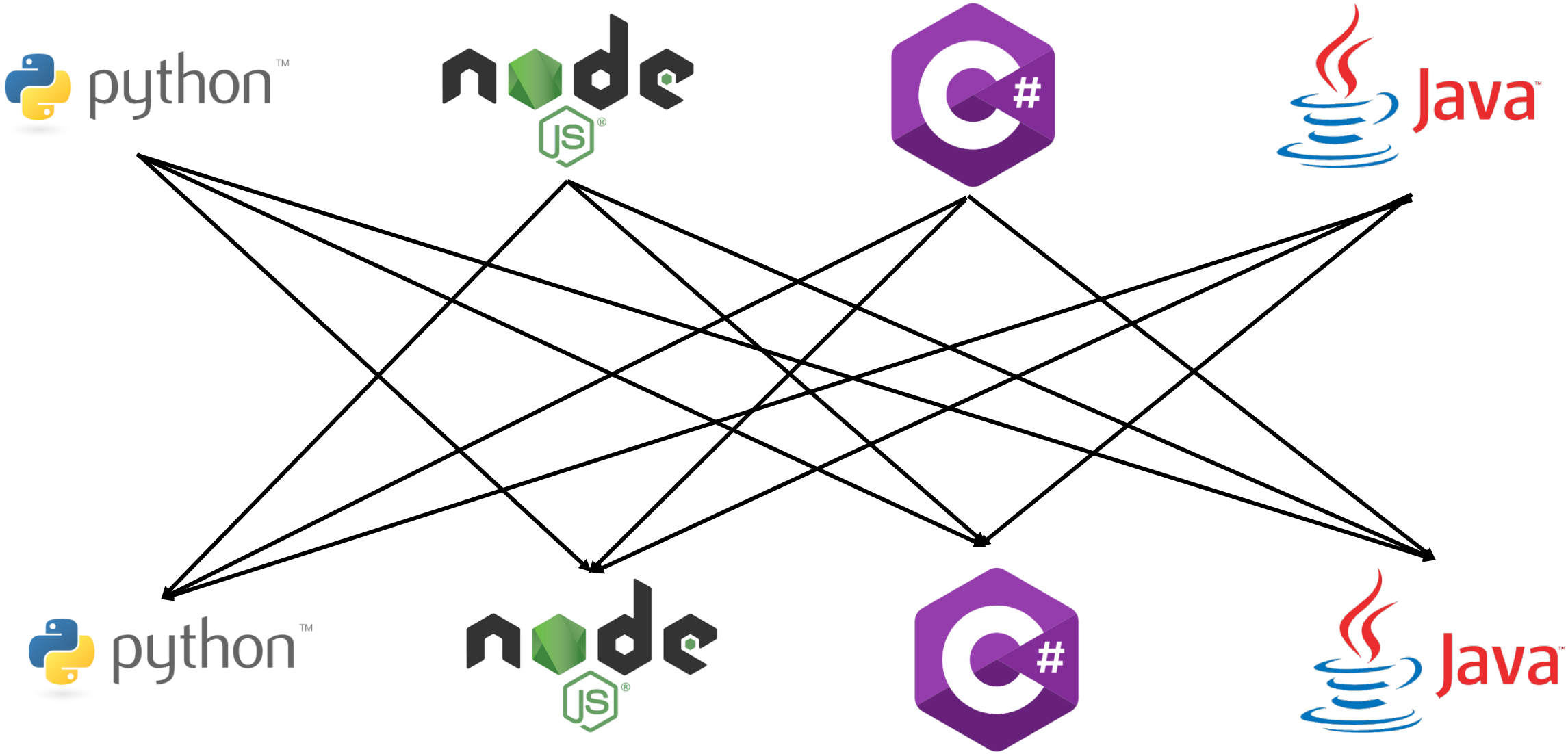

python


node
JS

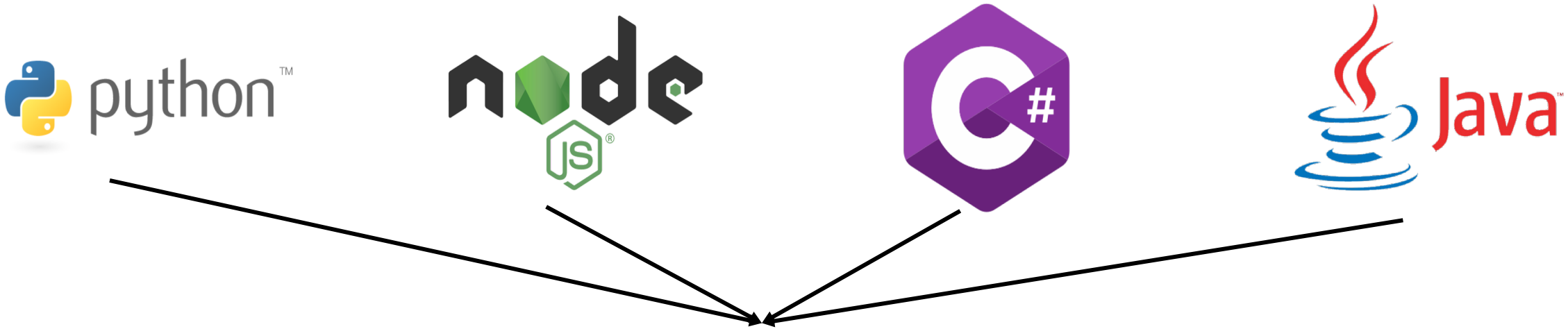

C#


Java

Naïve Approach

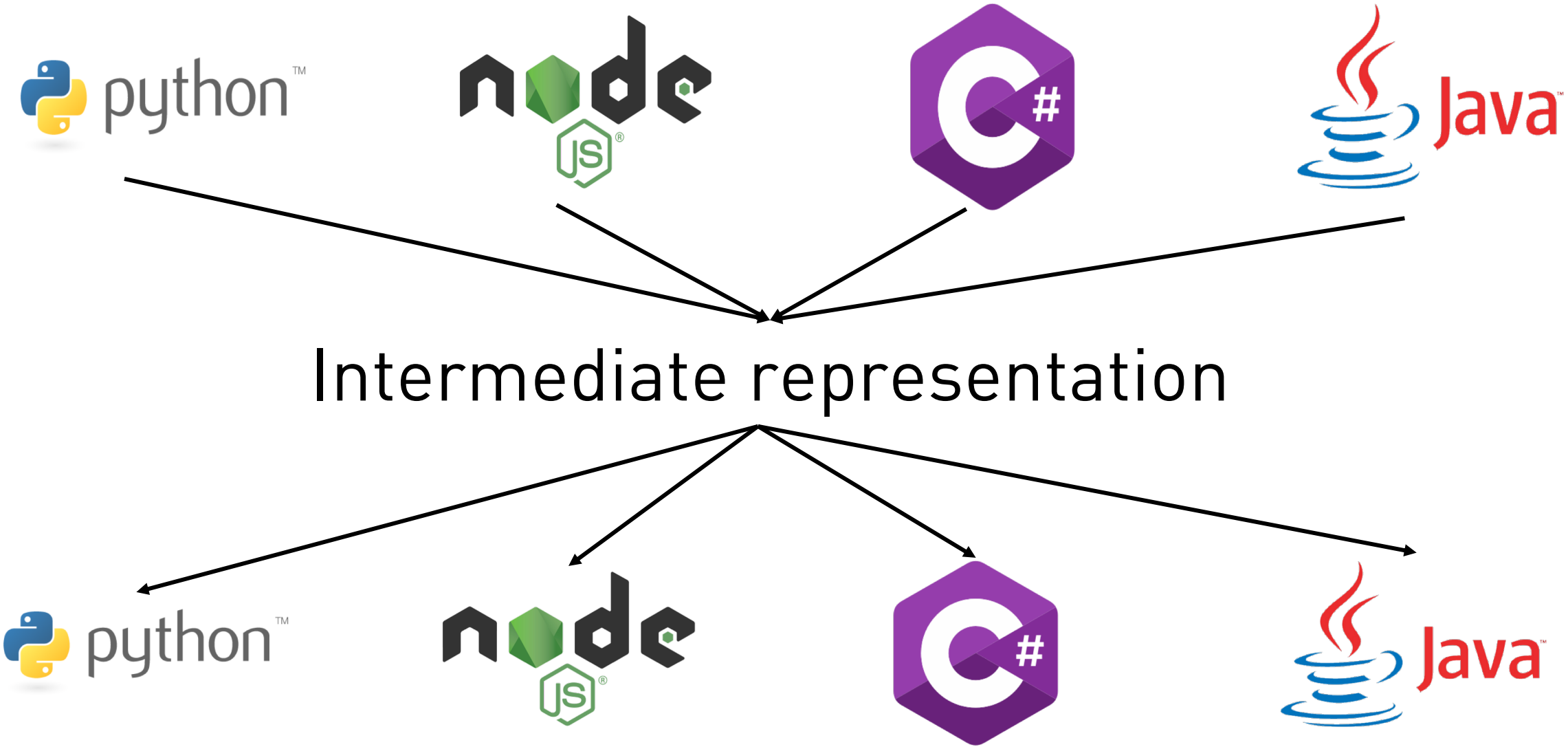


Intermediate Representation (IR)

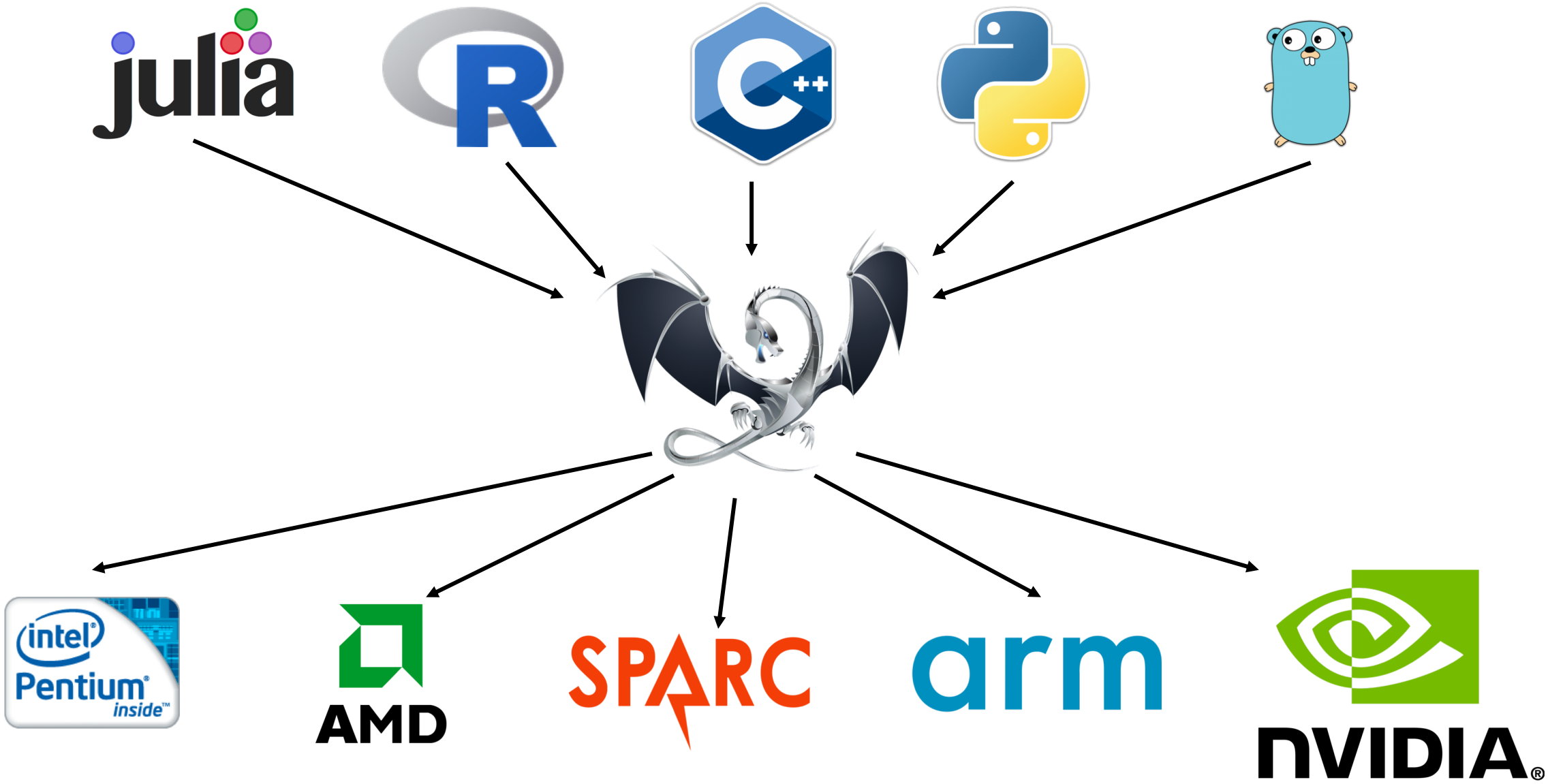


Intermediate representation

Intermediate Representation (IR)



LLVM-IR



Linear Regression in Python

```
def linreg(data=listf100, w=listf100, g=listf100, dims=int):
    dot = 1.0
    c = 0
    while c<dims:
        dot+=data[c]*w[c]
        c+=1
    label = data[dims]
    dot = dot* -label
    c2=0
    while(c2<dims):
        g[c2] += dot*data[c2]
        c2+=1
```

Linear Regression in LLVM-IR

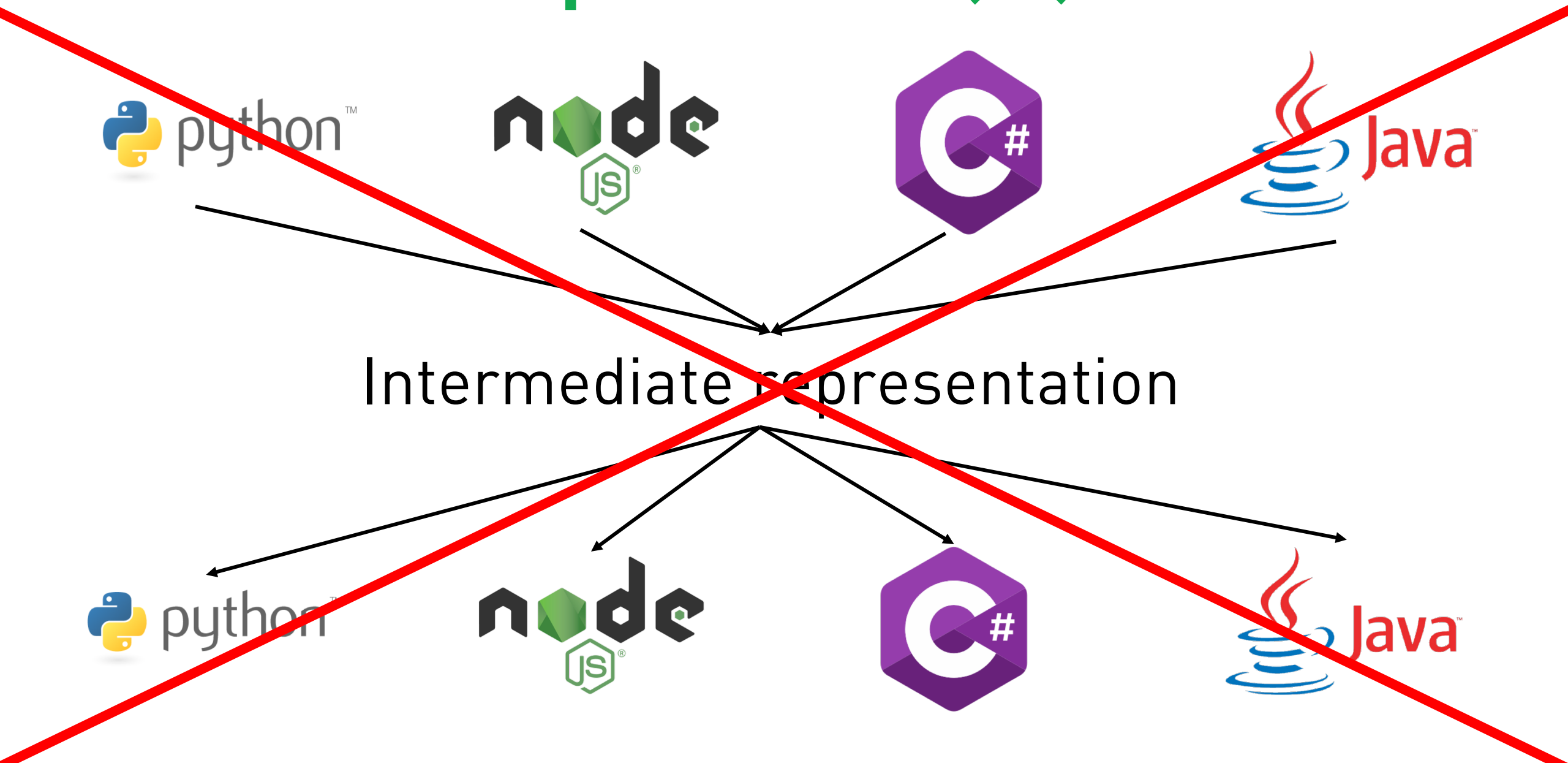
```
define void @linreg([100 x float]* %data, [100 x float]* %w, [100 x float]* %g, i32 %dims) {
entry:
  %tmp48 = alloca [100 x float]*
  store [100 x float]* %data, [100 x float]** %tmp
  %tmp49 = alloca [100 x float]*
  store [100 x float]* %w, [100 x float]** %tmp49
  %tmp50 = alloca [100 x float]*
  store [100 x float]* %g, [100 x float]** %tmp50
  %tmp51 = alloca i32
  store i32 %dims, i32* %tmp51
  %tmp52 = alloca float
  store float 1.000000e+00, float* %tmp52
  %tmp53 = load float* %tmp52
  %dot = alloca float
  store float %tmp53, float* %dot
  %tmp54 = alloca i32
  store i32 8, i32* %tmp54
  %tmp55 = load i32* %tmp54
  %c = alloca i32
  store i32 %tmp55, i32* %c
  br label %start_while

start_while:
  %tmp56 = load i32* %c
  %tmp57 = load i32* %tmp51
  %cmptmp = icmp slt i32 %tmp56, %tmp57
  %booltmp = uitofp i1 %cmptmp to float
  %whilecond = fcmp one float %booltmp, 0.000000e+00
  br i1 %whilecond, label %do_while, label %end_while
```

```
do_while:
  %tmp58 = load [100 x float]** %t
  %tmp59 = load i32* %c
  %0 = getelementptr [100 x float]
  %tmp60 = load float* %0
  %tmp61 = load [100 x float]** %t
  %tmp62 = load i32* %c
  %1 = getelementptr [100 x float]
  %tmp63 = load float* %1
  %tmp64 = fmul float %tmp60, %tmp
  %tmp65 = load float* %dot
  %2 = fadd float %tmp65, %tmp64
  %tmp66 = load float* %dot
  store float %2, float* %dot
  %tmp67 = alloca i32
  store i32 1, i32* %tmp67
  %tmp68 = load i32* %tmp67
  %tmp69 = load i32* %c
  %3 = add i32 %tmp69, %tmp68
  %tmp70 = load i32* %c
  store i32 %3, i32* %c
  br label %start_while
```

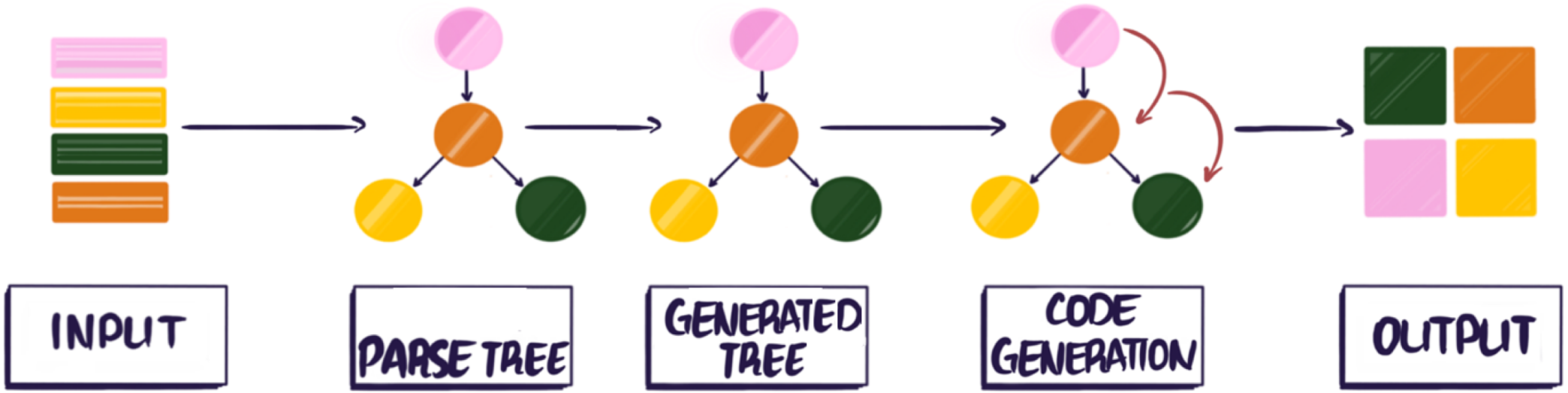
```
end_while:                                        ; preds = %start_while
  %tmp71 = load [100 x float]** %tmp48
  %tmp72 = load i32* %tmp51
  %4 = getelementptr [100 x float]* %tmp71, i32 0, i32 %tmp72
  %tmp73 = load float* %4
  %label = alloca float
  store float %tmp73, float* %label
  %tmp74 = load float* %dot
  %tmp75 = load float* %label
  %tmp76 = fsub float 0.000000e+00, %tmp75
  %tmp77 = fmul float %tmp74, %tmp76
  store float %tmp77, float* %dot
  %tmp78 = alloca i32
  store i32 0, i32* %tmp78
  %tmp79 = load i32* %tmp78
  %c2 = alloca i32
  store i32 %tmp79, i32* %c2
  br label %start_while1
define void @main() {
entry:
  %tmp112 = alloca float
  store float 1.000000e+02, float* %tmp112
  %tmp113 = load float* %tmp112
  %0 = alloca [100 x float], i32 100
  %1 = getelementptr [100 x float]* %0, i32 0, i32 0
  store float 0.000000e+00, float* %1
```

```
%g = alloca [100 x float]*
store [100 x float]* %205,
%tmp124 = load [100 x float
%tmp125 = load [100 x float
%tmp126 = load [100 x float
%tmp127 = alloca i32
store i32 99, i32* %tmp127
%tmp128 = load i32* %tmp127
call void @linreg([100 x fl
ret void
```

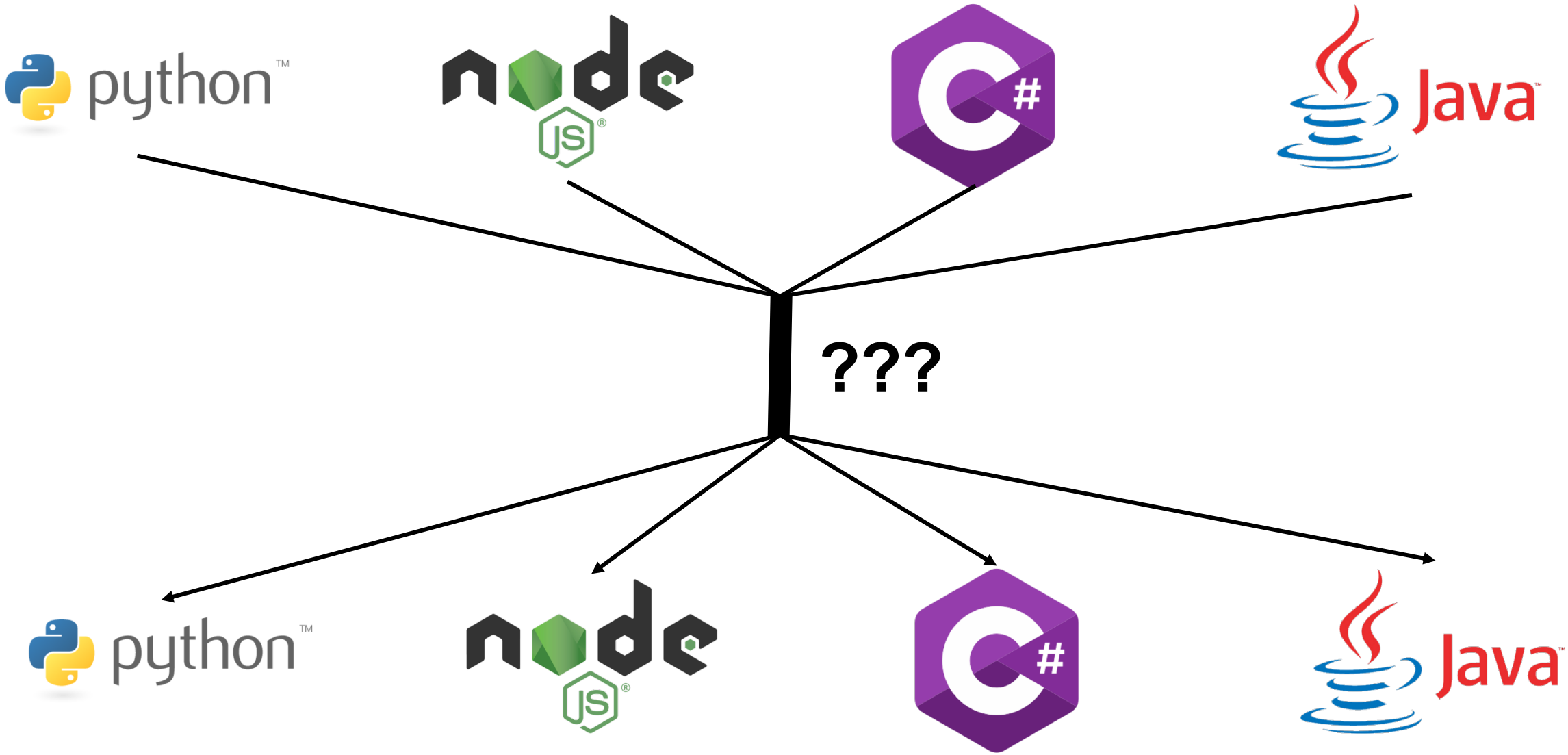
Intermediate Representation (IR)

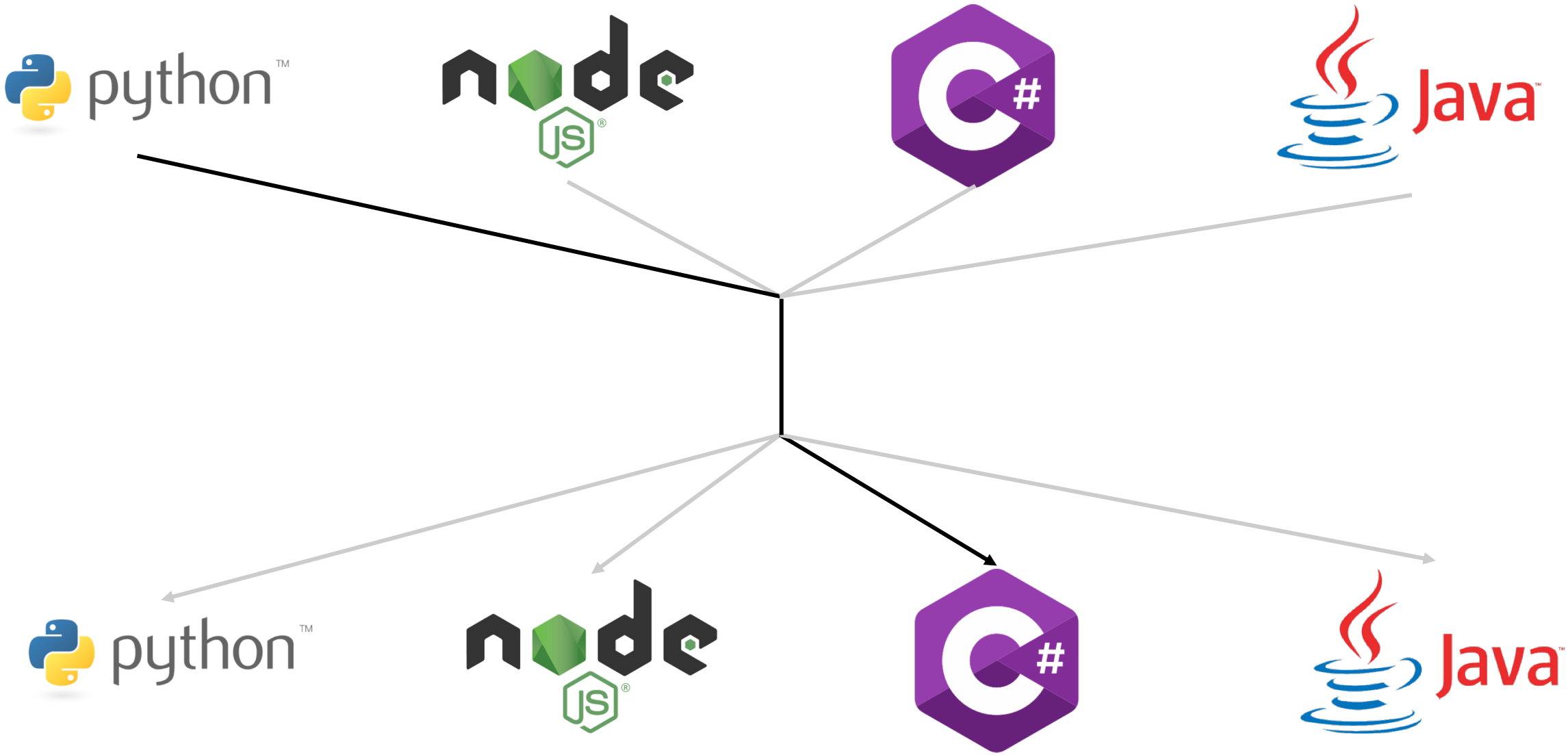


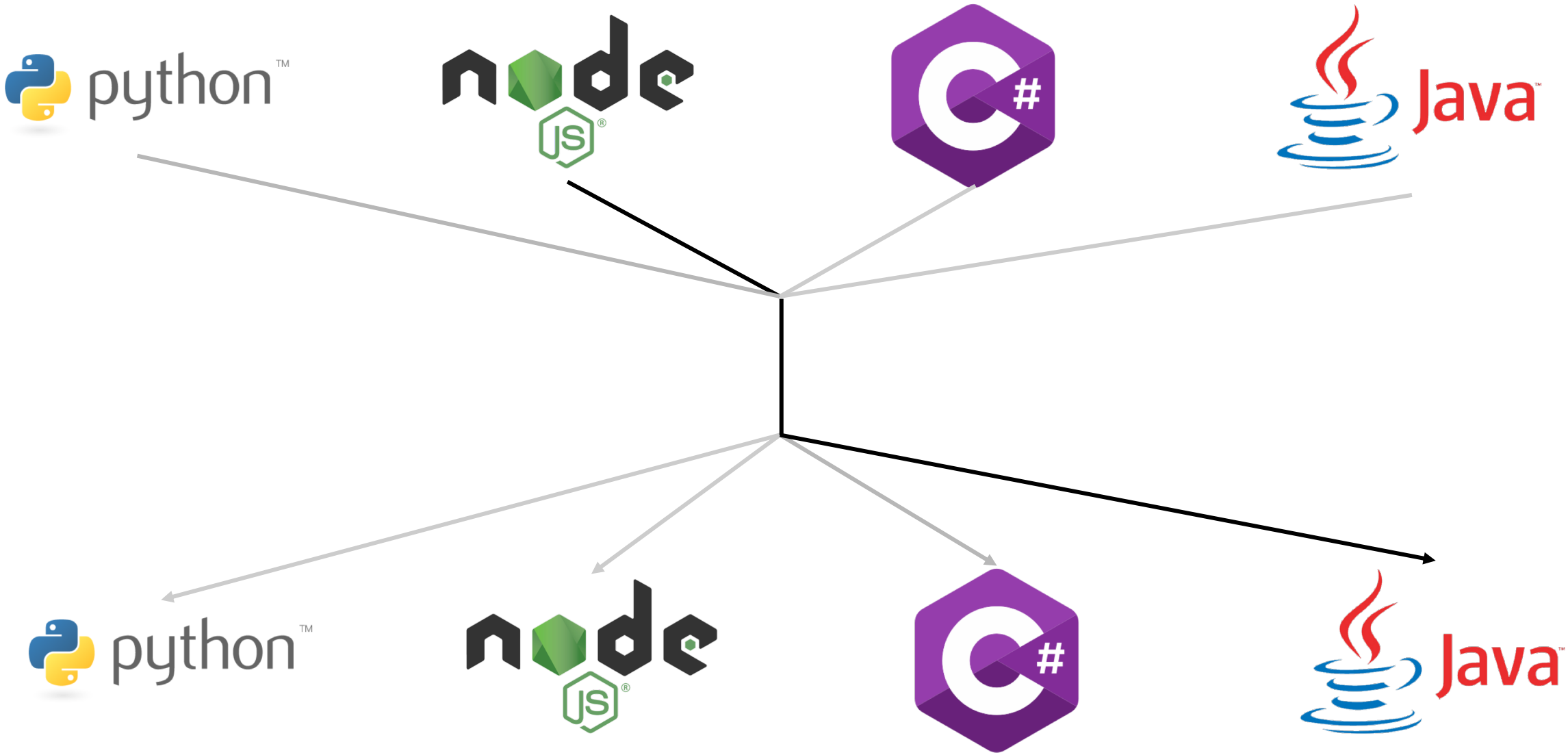
How does Babel work?

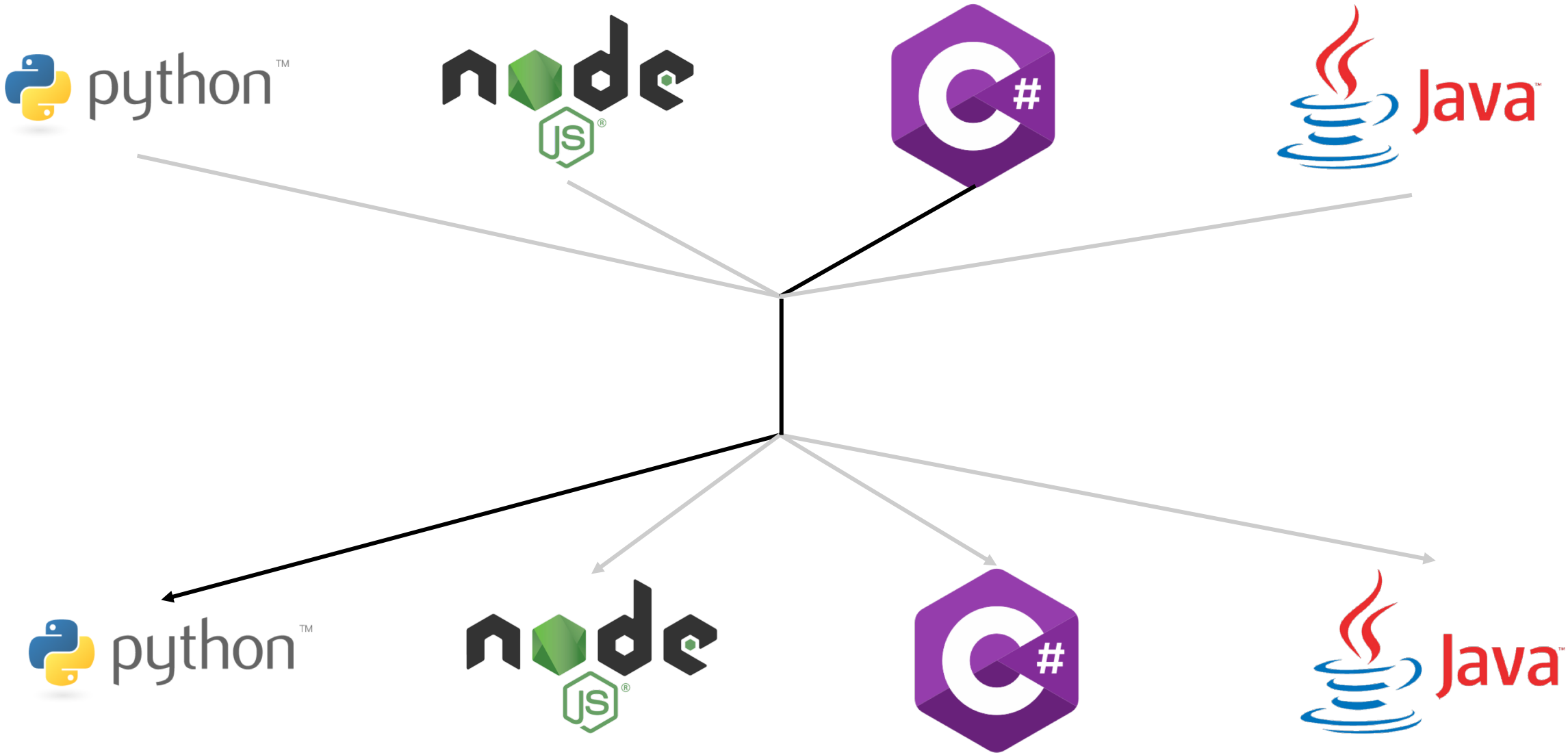


Babel Parser only parses JavaScript ☹


python
node
JS
C#
Java
???
python
node
JS
C#
Java


python
node
JS
C#
Java
python
node
JS
C#
Java


python
node
JS
C#
Java
python
node
JS
C#
Java


python
node
JS
C#
Java
python
node
JS
C#
Java

Complexity

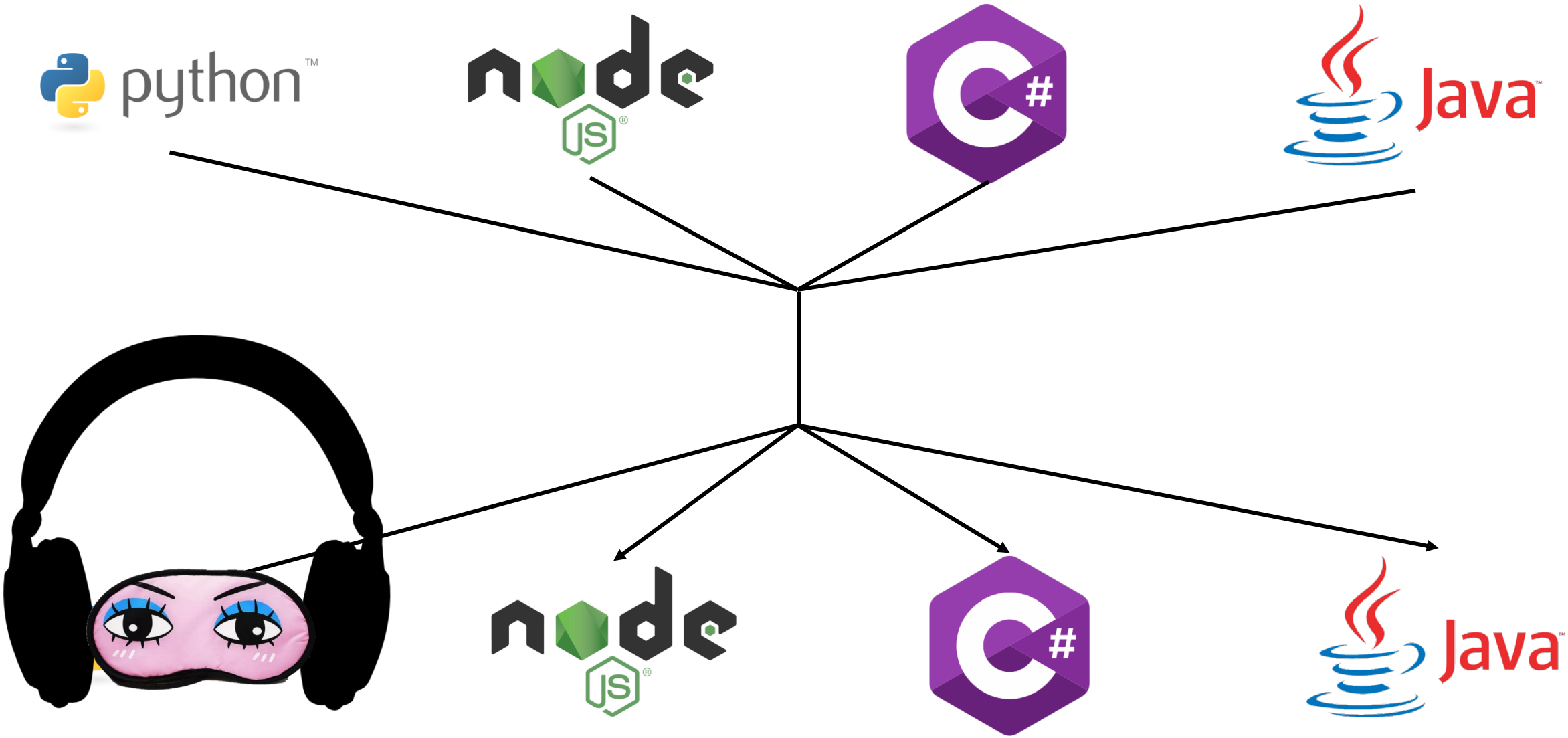
For every input language we support:

1. **We want to only have to do the work once**
2. We want to define the input language without knowing or caring how many output languages exist

We do not want to have to rewrite the translation for every possible combination, that would be $O(n^2)$ and we want $O(n)$

For every input language we support:

1. We want to only have to do the work once
2. **We want to define the input language without knowing or caring how many output languages exist**


python
node
JS
C#
Java
node
JS
C#
Java

For every input language we support:

1. We want to only have to do the work once
2. We want to define the input language without knowing or caring how many output languages exist

Same principle for target languages


R


node
JS


Java


C++


python™


golang


C#


Ruby


php


Scala


Rust


Perl

Distributed Development

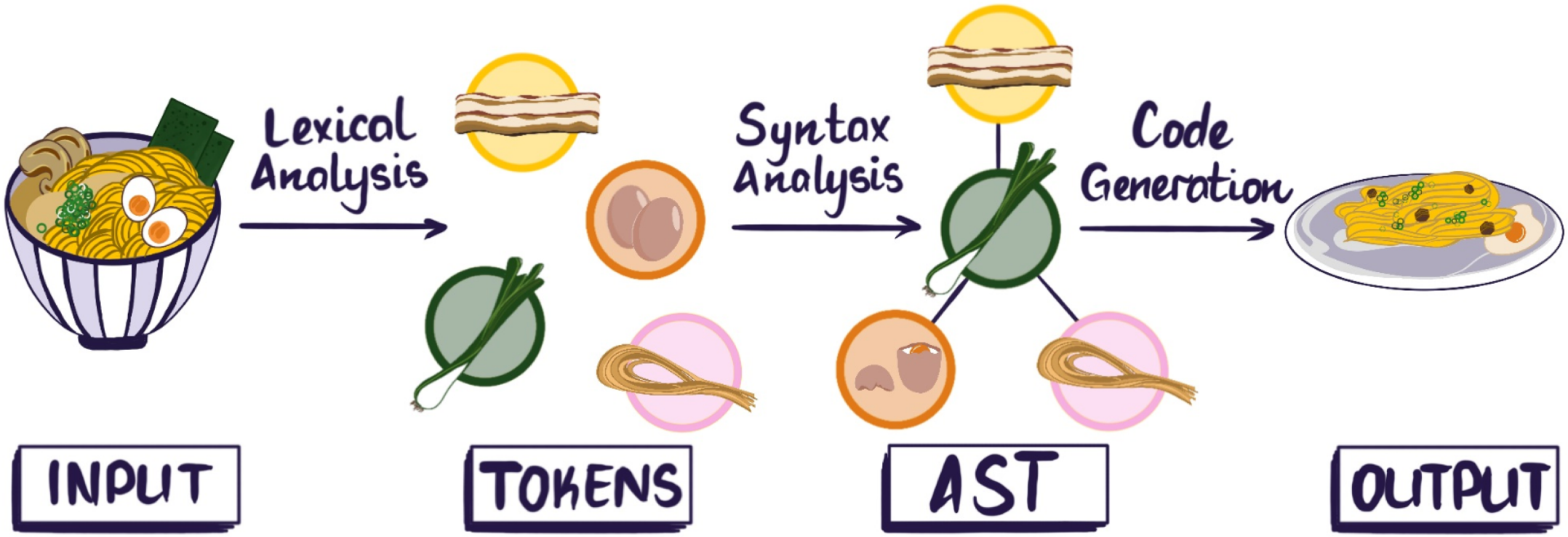
1. Many communities are small + passionate
2. Open source ethos :)
3. How many people are compiler experts?

Summary of Technical Requirements

1. Accept arbitrary # of input languages
2. Generate arbitrary # of output languages
3. Support distributed development
4. Linear-time development cost
5. Web-friendly JavaScript library

How to add your own output language

Compiler 101



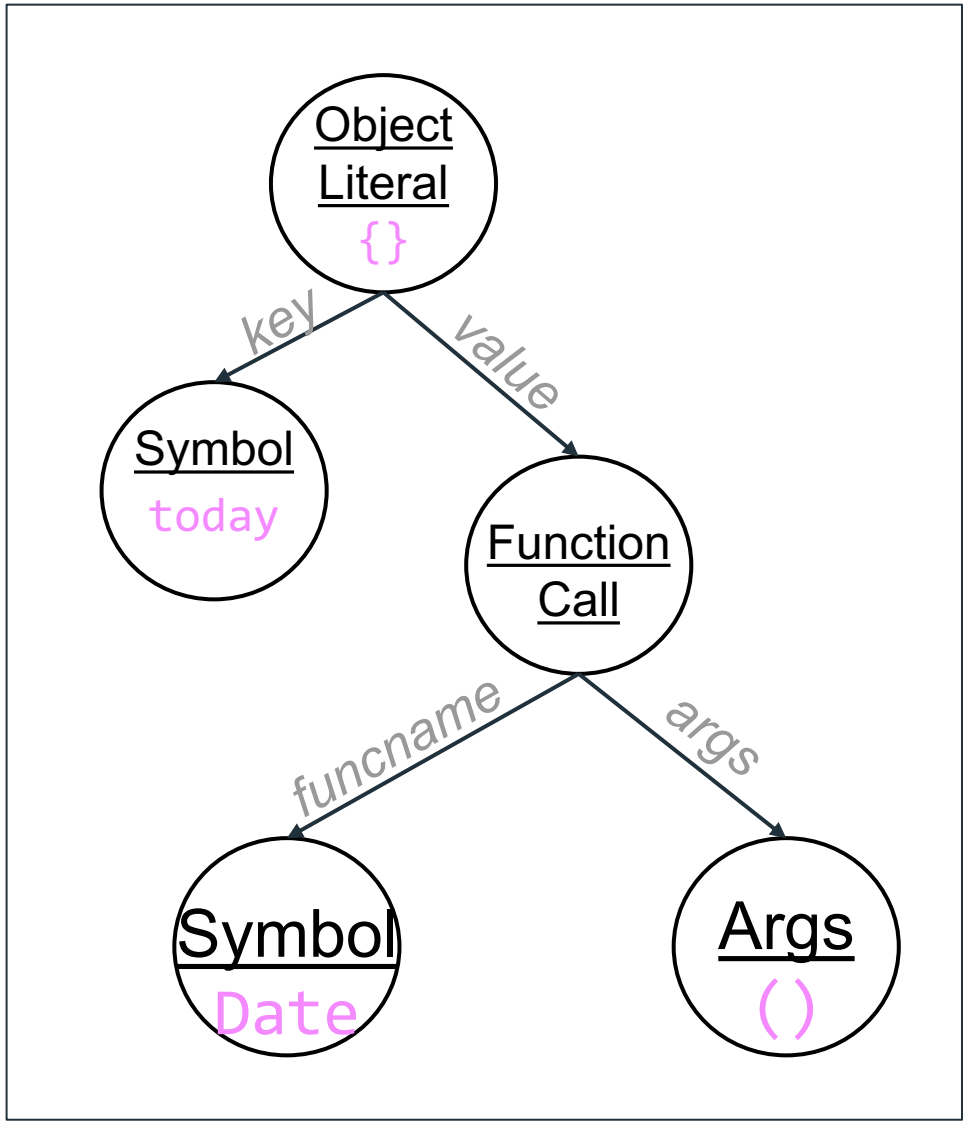
Illustrations by @Irina!

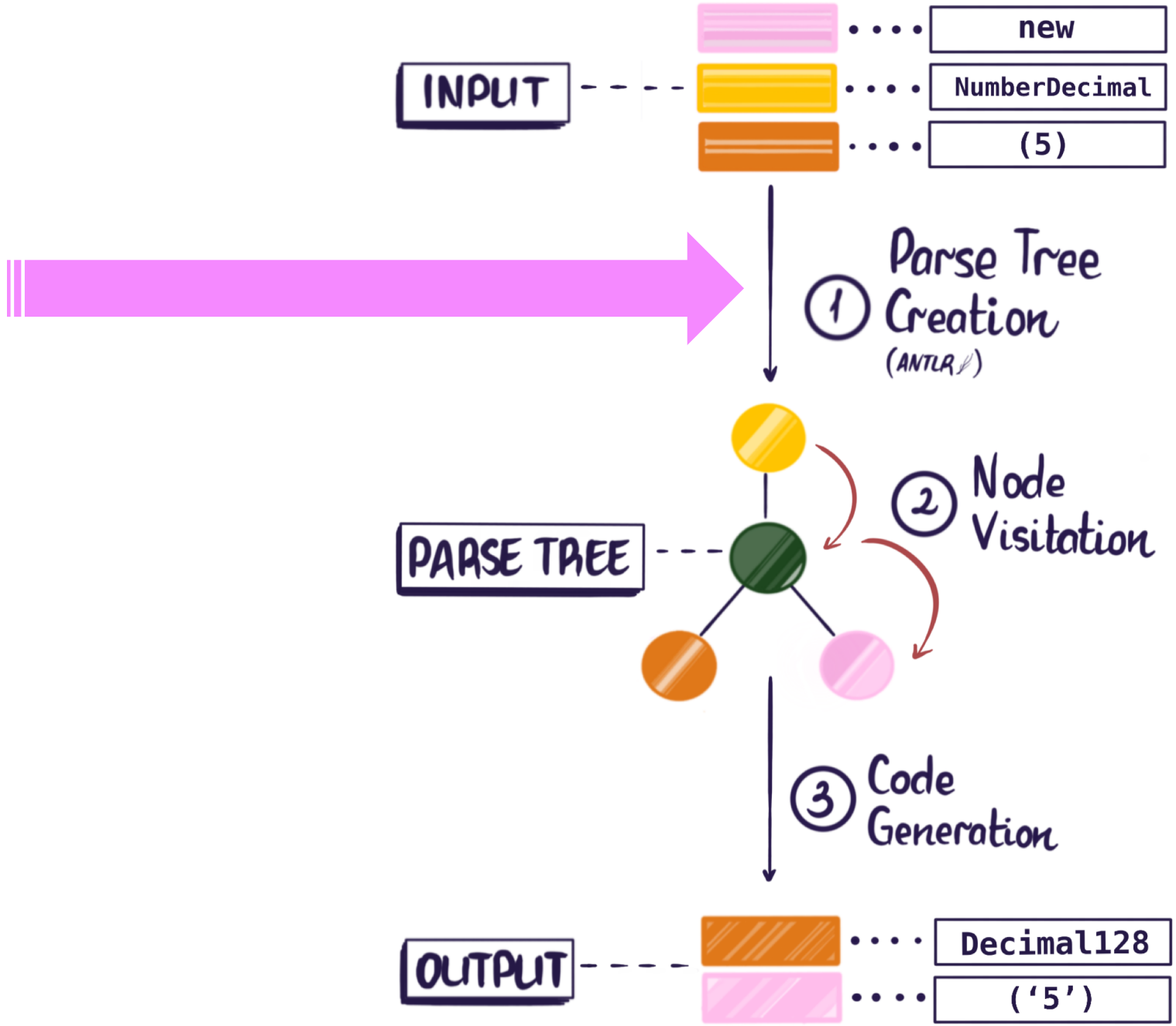
Tree Building

The tree-building stage includes lexing, parsing, syntax analysis, and more!

START

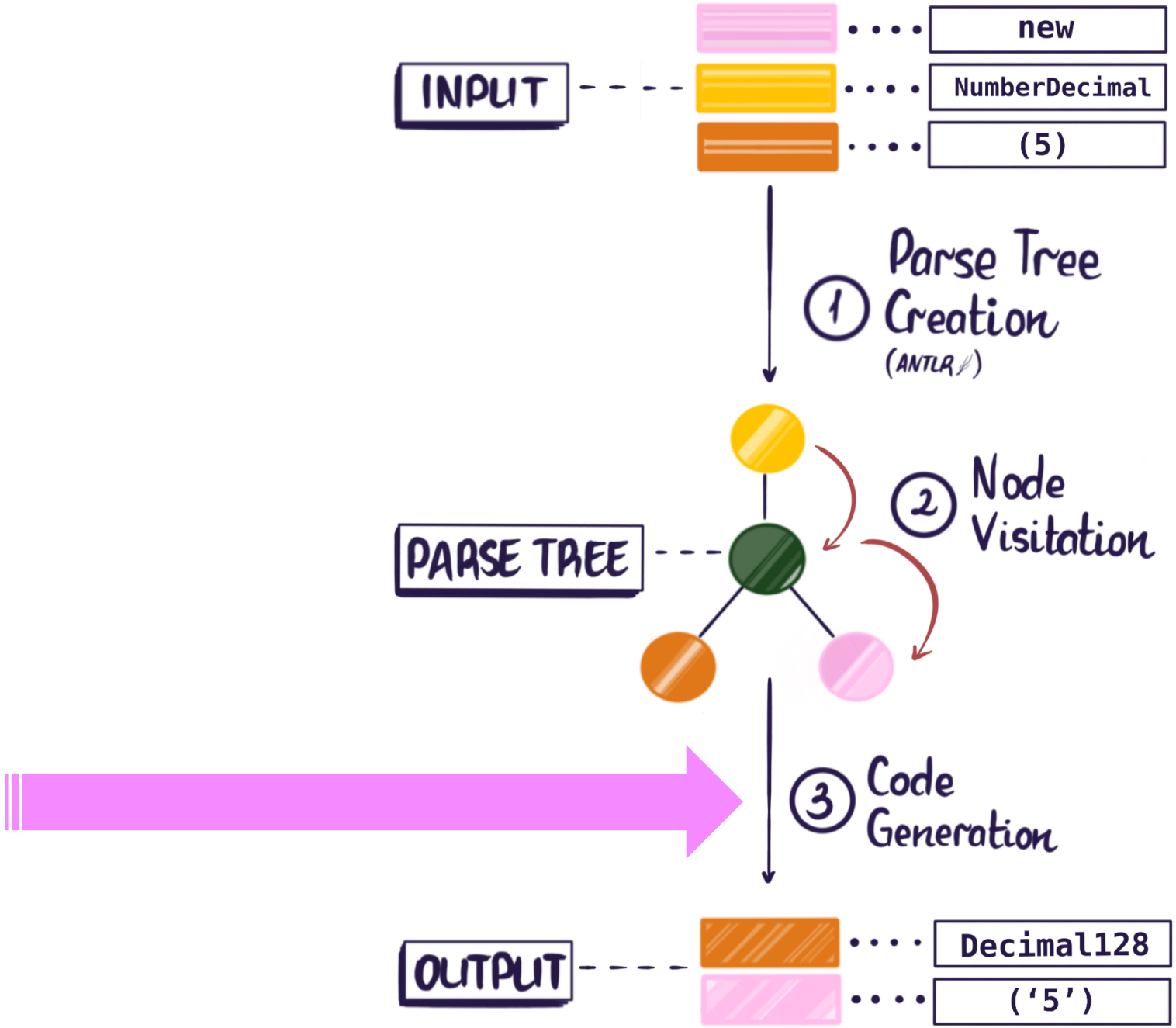
```
{  today: Date() }
```




INPUT
new
NumberDecimal
(5)
1 Parse Tree Creation
(ANTLR)
PARSE TREE
2 Node Visitation
3 Code Generation
OUTPUT
Decimal128
('5')

- Parsers are hard!
- Support for most languages
- We apply ANTLR to our input and we get a tree. *The tree building stage is completely handled!*
- Bonus: ANTLR generates a visitor class!

OUR SAVIOR


INPUT
new
NumberDecimal
(5)
1 Parse Tree Creation
(ANTLR)
PARSE TREE
2 Node Visitation
3 Code Generation
OUTPUT
Decimal128
('5')

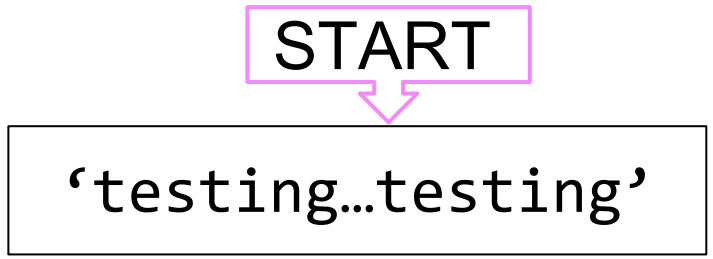
The Visitor Pattern

Visitors traverse trees by "*visiting*" each node

For each type of node, the visitor calls the corresponding function.

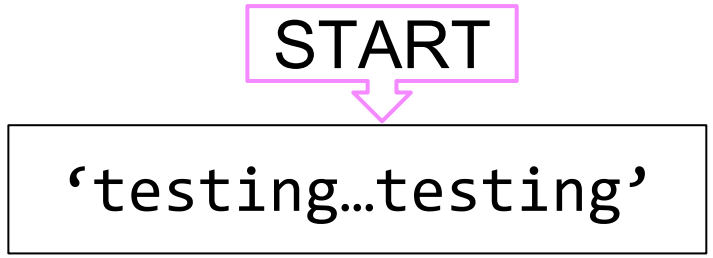
➢ When the visitor sees a node that is a "string" type, it will call the **`visitString`** method and expect the generated code to be returned.

Extremely Simple Visitor Example

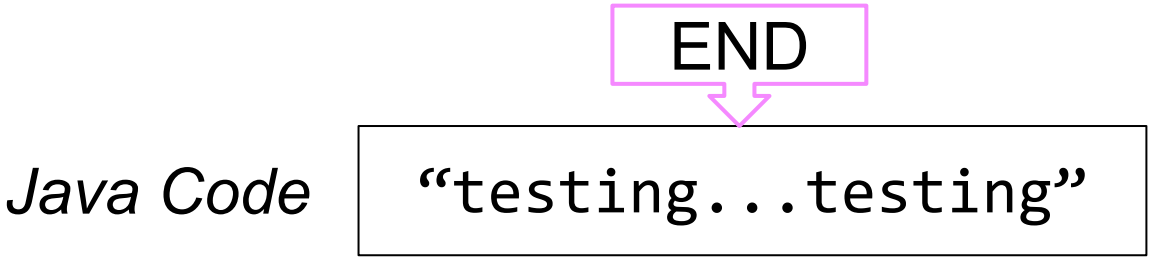


JavaScript Code

Extremely Simple Visitor Example

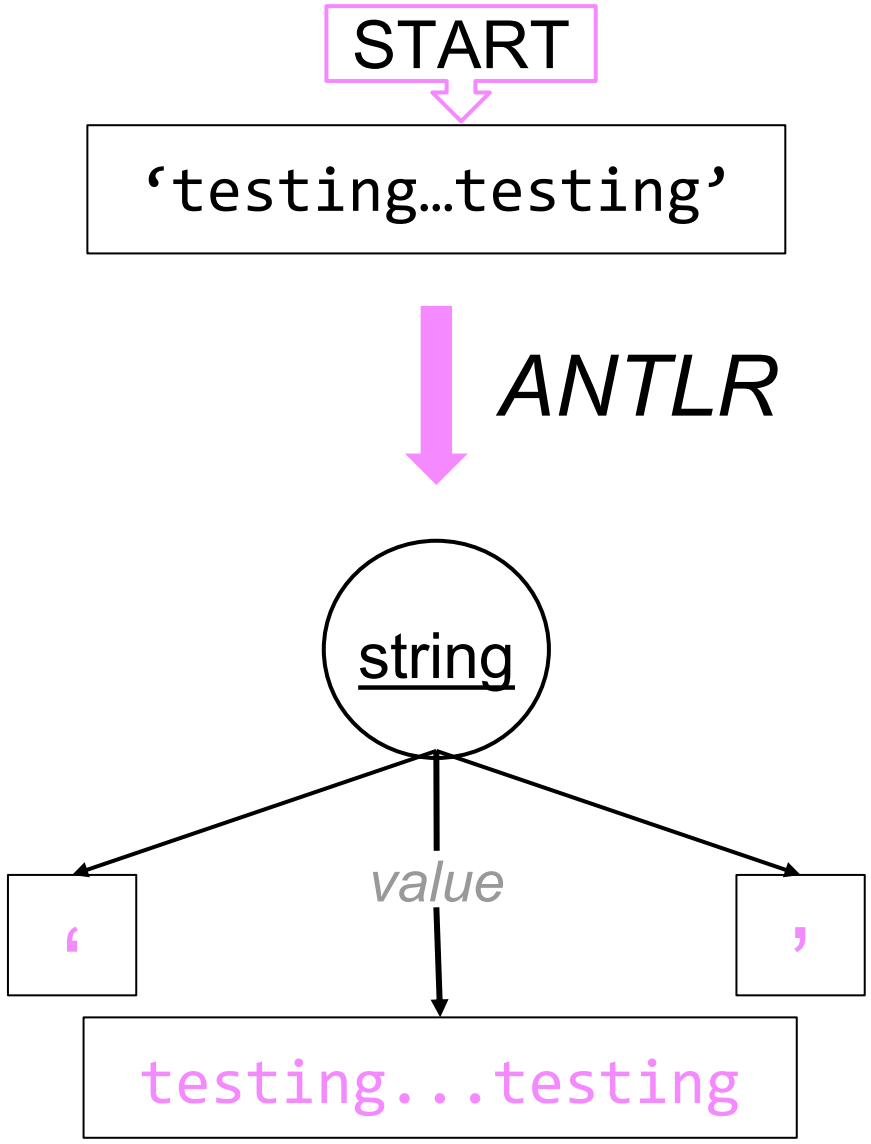


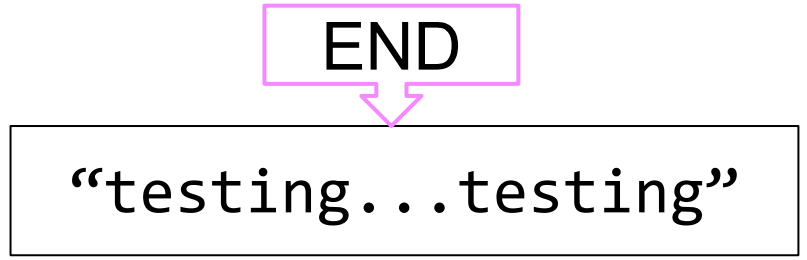
JavaScript Code



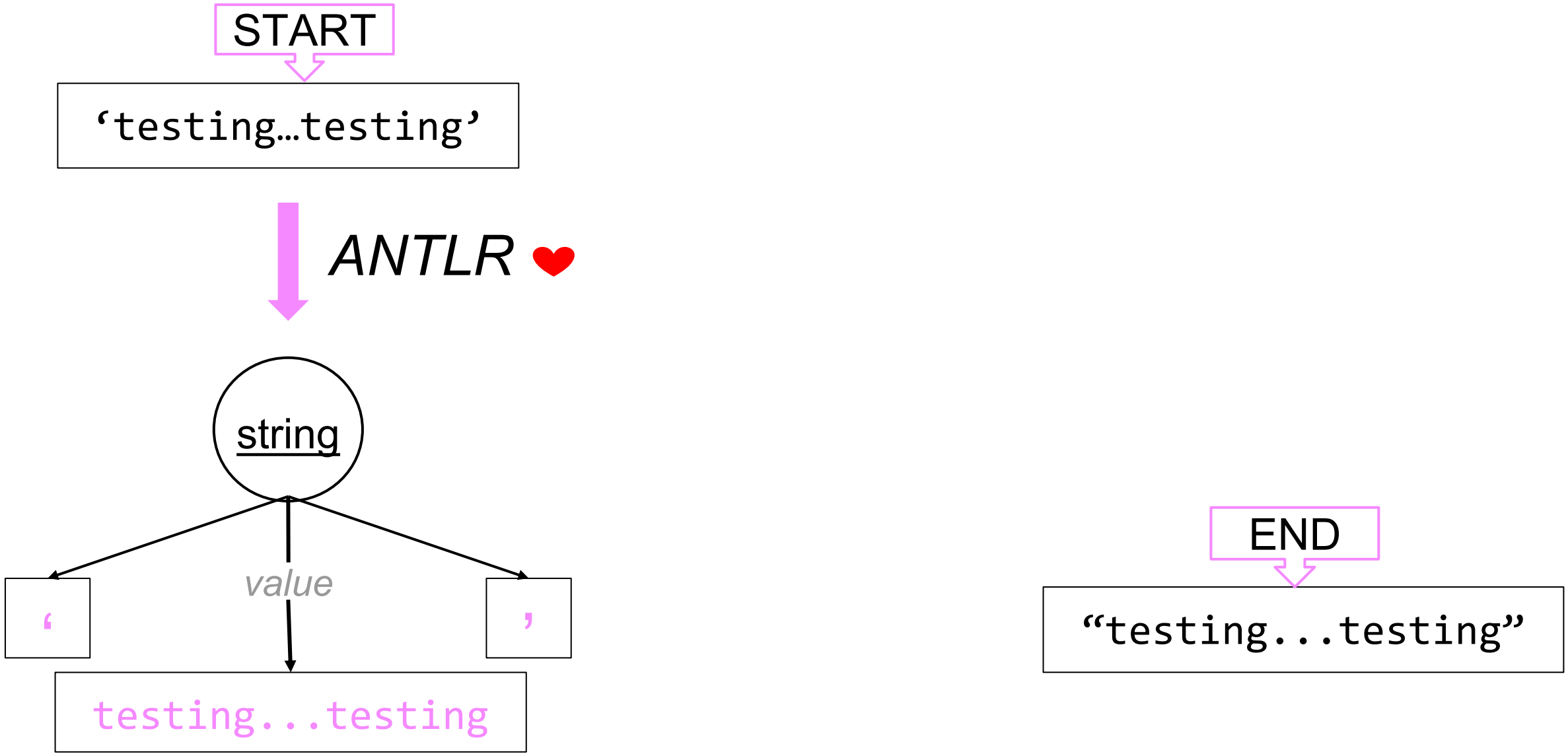
Java Code

Extremely Simple Visitor Example

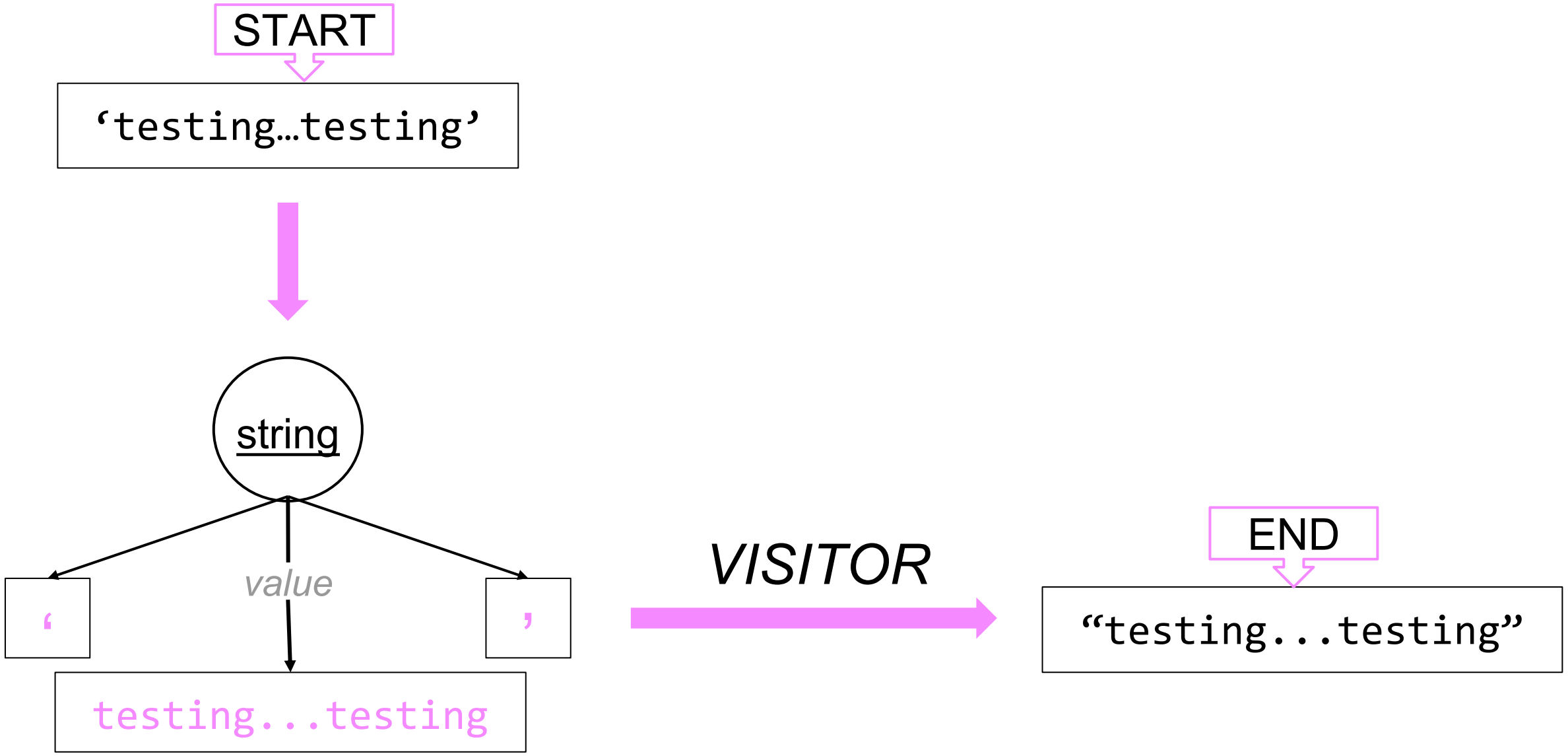




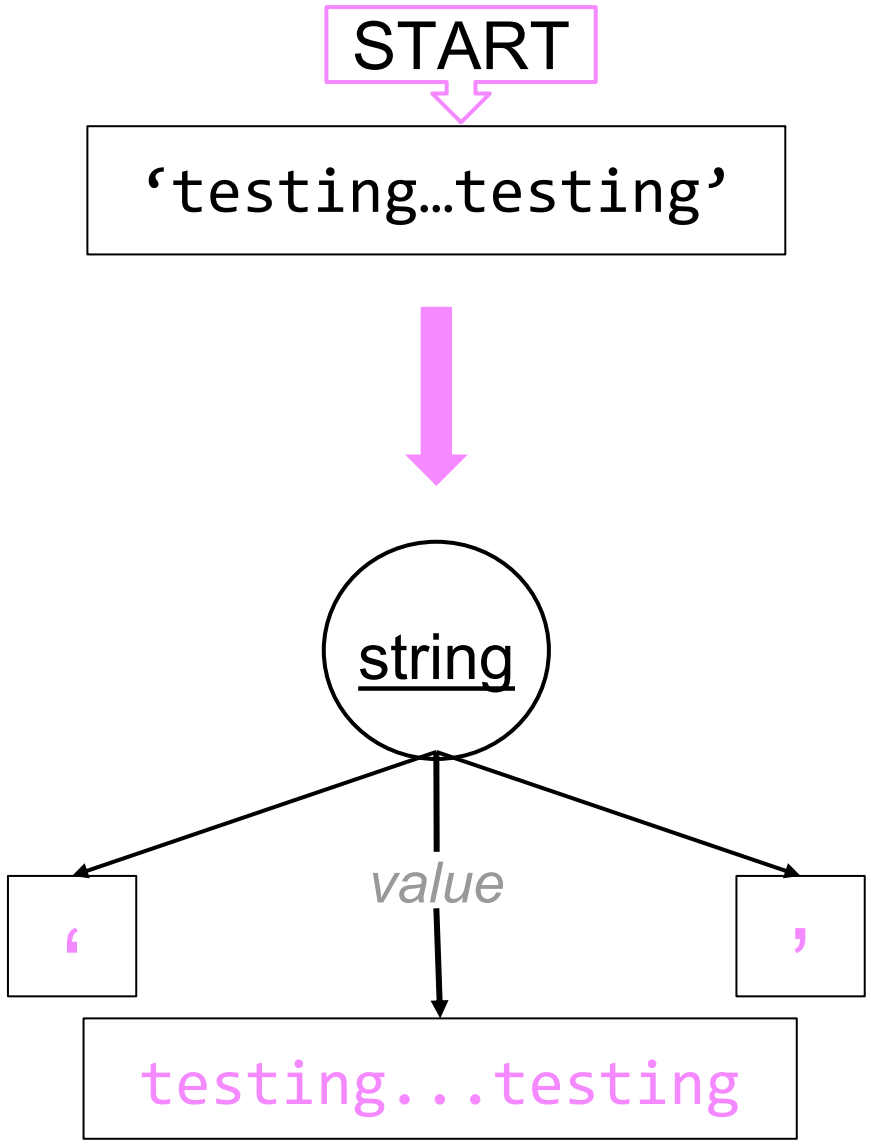
Extremely Simple Visitor Example



Extremely Simple Visitor Example



Extremely Simple Visitor Example

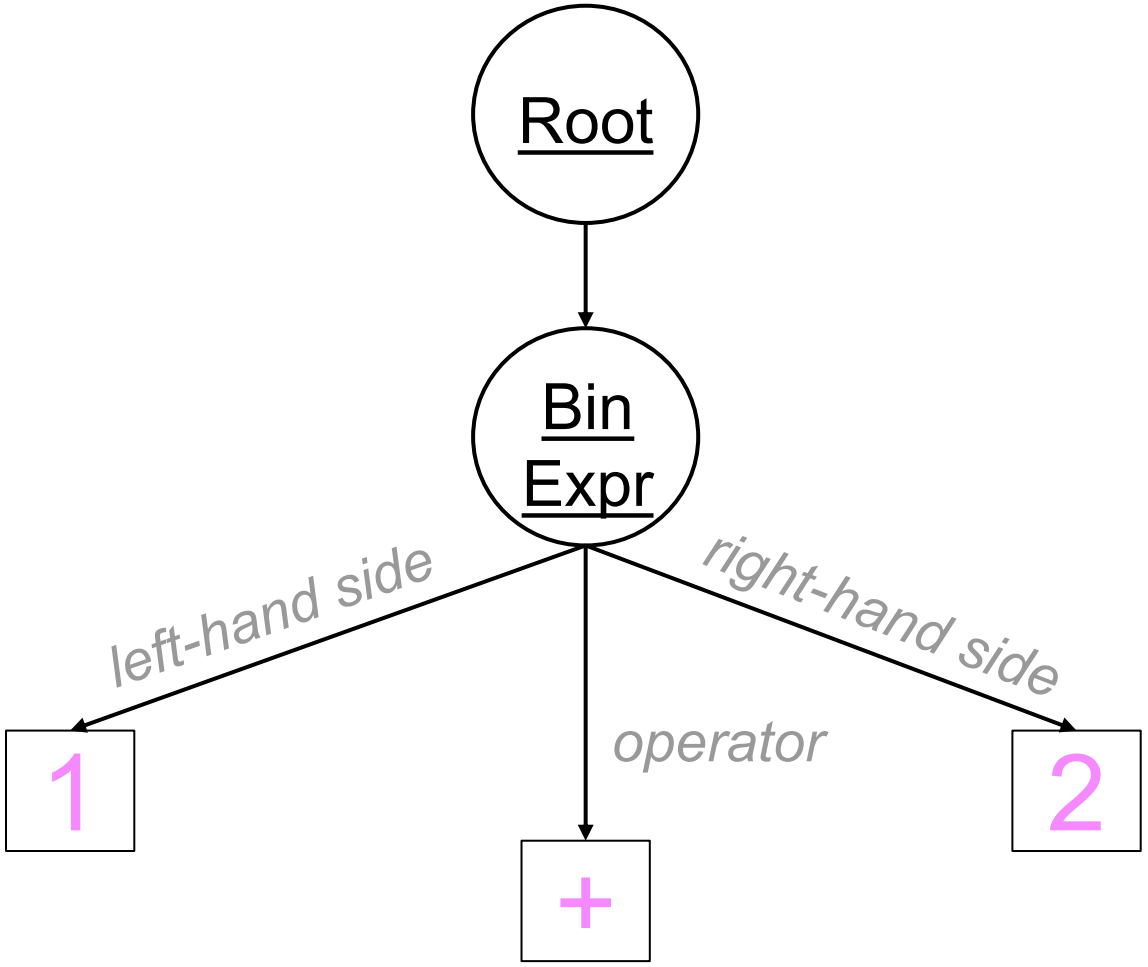



```
class Visitor() extends ANTLRVisitor {
    visitString(node) {
        const val = node.value;
        return ‘“‘ + val + ‘”‘;
    }
}
```

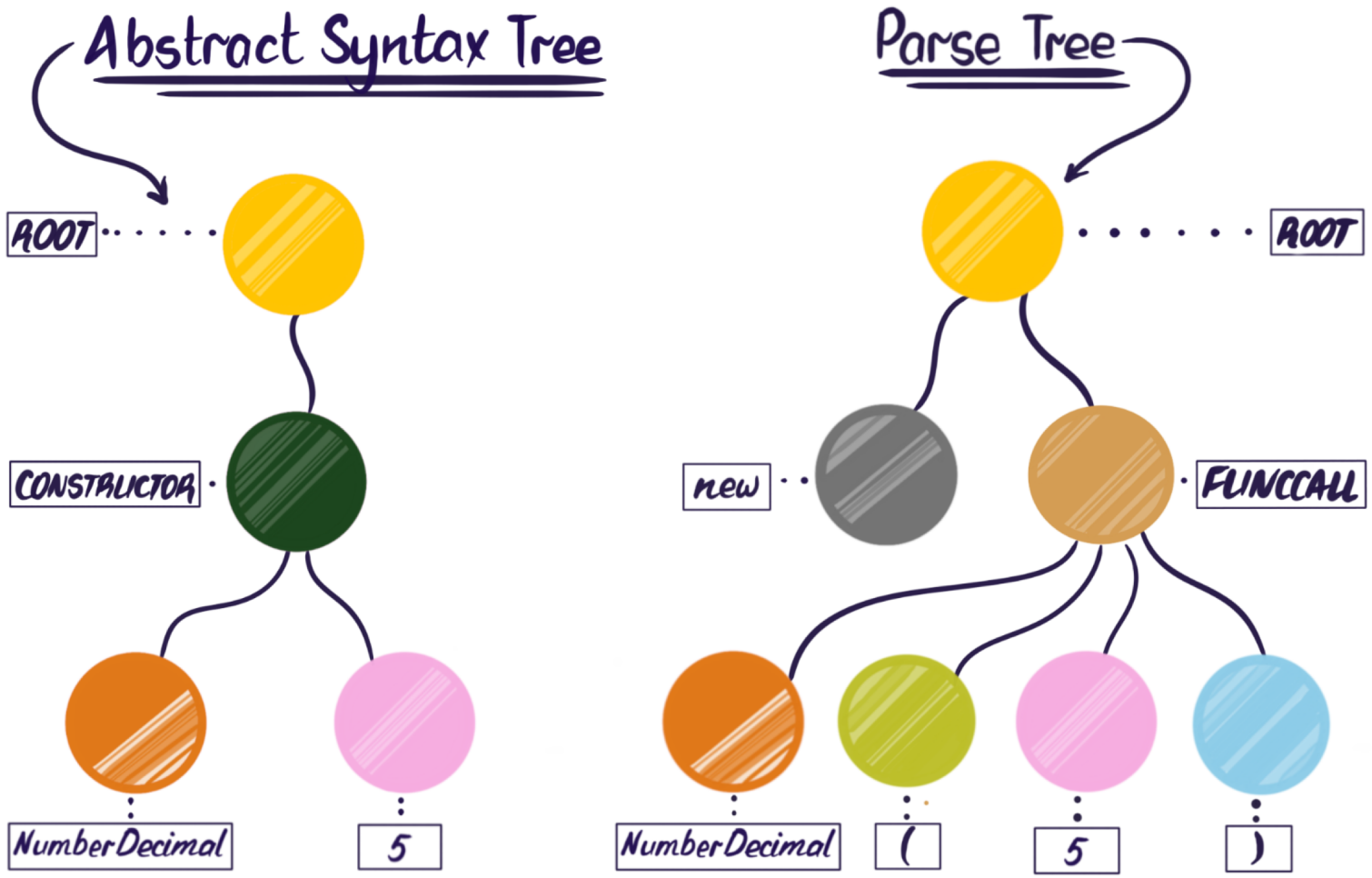
END

“testing...testing”

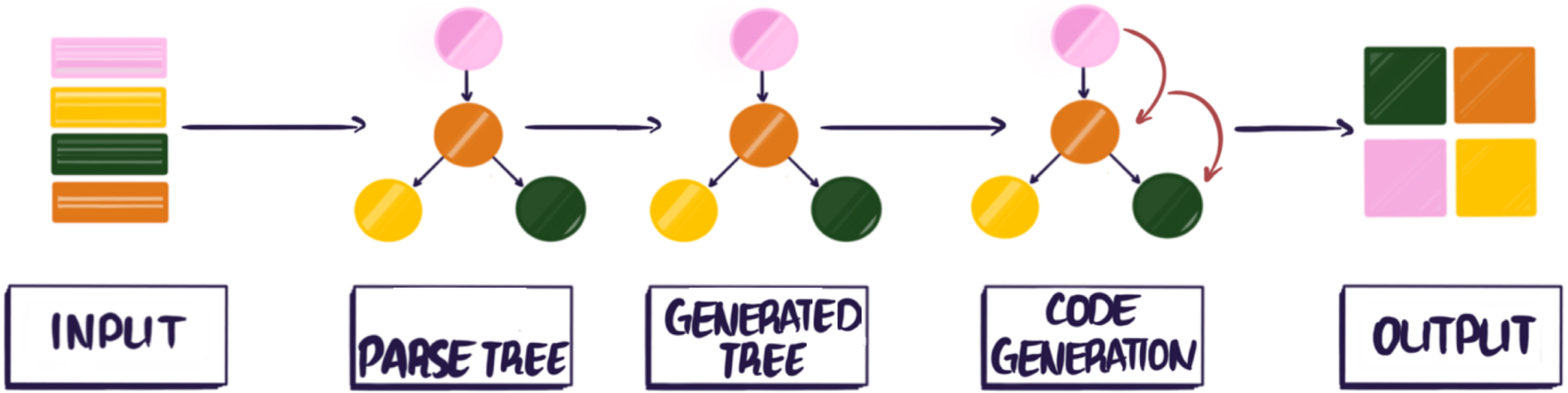
What does ANTLR look like?



Parse Tree vs AST

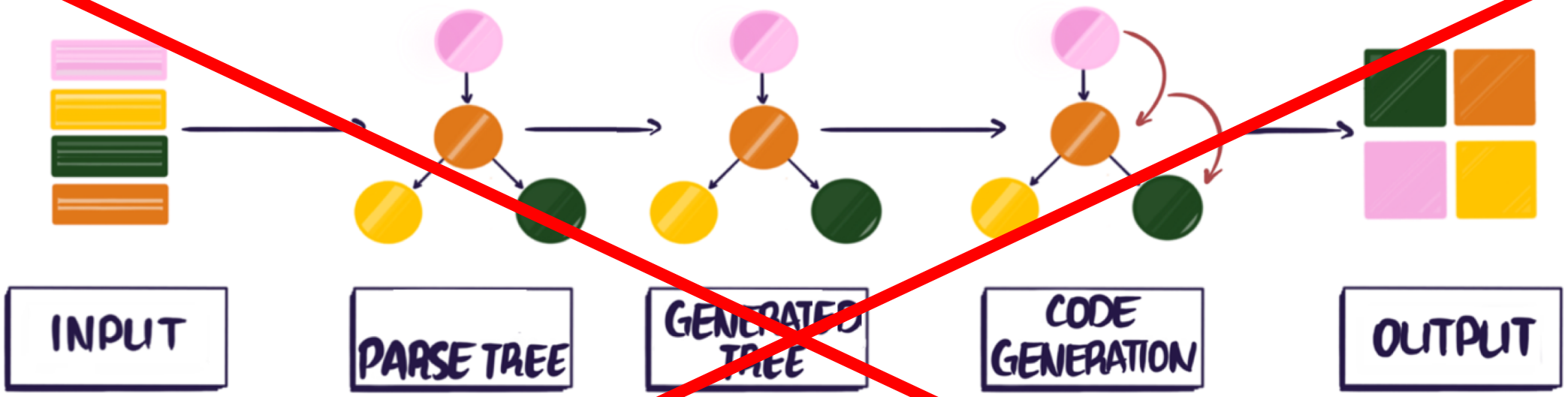


How does Babel work?

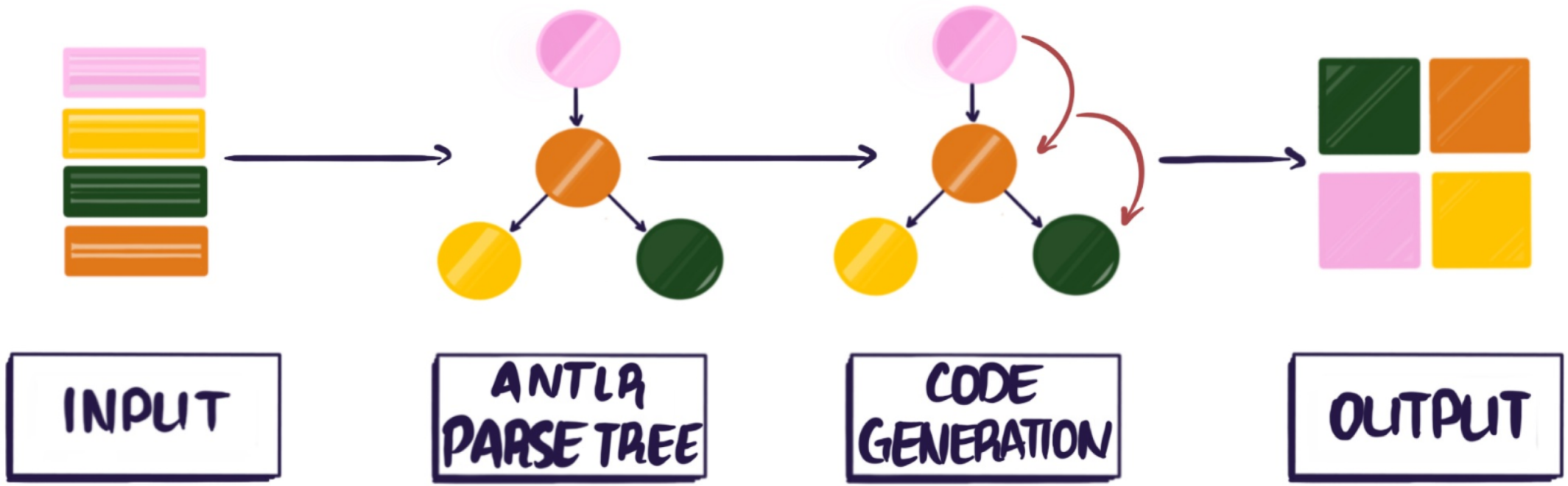


ANTLR trees are read-only ☹

How does Babel work?



ANTLR trees are read-only ☹


INPUT
ANTLR PARSE TREE
CODE GENERATION
OUTPUT

ANTLR-generated Visitor Classes

```
class ECMAScriptVisitor {
  // Visit a parse tree produced by ECMAScriptParser#LogicalOrExpression.
  visitLogicalOrExpression = function(ctx) {
    return this.visitChildren(ctx);
  };
  // Visit a parse tree produced by ECMAScriptParser#LogicalAndExpression.
  visitLogicalAndExpression = function(ctx) {
    return this.visitChildren(ctx);
  };
  // Visit a parse tree produced by ECMAScriptParser#NotExpression.
  visitNotExpression = function(ctx) {
    return this.visitChildren(ctx);
  };

  ...
```

Abstract away our problems (part I)

Visitor classes can only visit trees generated from a single grammar → need one Visitor per input language

- To avoid having the same code in every visitor, abstract the shared code into a super class!
- Each visitor will act as abstraction layer between superclass visitor and grammar-generated nodes.

codegeneration/python/visitor.js

```
const Python3Visitor = require(
  '../lib/antlr/Python3Visitor'
);

class Visitor extends Python3Visitor {
  visitAnd_expr(ctx) {
    return this.generateAndExpression(ctx);
  }
}
```

codegeneration/python/visitor.js

```
const Python3Visitor = require(
  '../lib/antlr/Python3Visitor'
);

class Visitor extends Python3Visitor {
  visitAnd_expr(ctx) {
    return this.generateAndExpression(ctx);
  }
}
```

codegeneration/javascript/visitor.js

```
const ECMAScriptVisitor = require(
  '../lib/antlr/ECMAScriptVisitor'
);

class Visitor extends ECMAScriptVisitor {
  visitLogicalAndExpression(ctx) {
    return this.generateAndExpression(ctx);
  }
}
```

codegeneration/python/visitor.js

```
const Python3Visitor = require(
  '../lib/antlr/Python3Visitor'
);

class Visitor extends Python3Visitor {
  visitAnd_expr(ctx) {
    return this.generateAndExpression(ctx);
  }
}
```

codegeneration/javascript/visitor.js

```
const ECMAScriptVisitor = require(
  '../lib/antlr/ECMAScriptVisitor'
);

class Visitor extends ECMAScriptVisitor {
  visitLogicalAndExpression(ctx) {
    return this.generateAndExpression(ctx);
  }
}
```

codegeneration/code-generation-visitor.js

```
export default (ANTLRVisitor) =>
  class CodeGenerationVisitor extends ANTLRVisitor {
    generateAndExpression(ctx) {
      const lhs = this.visit(ctx.getChildAt(0));
      const rhs = this.visit(ctx.getChildAt(2));

      return `${lhs} && ${rhs}`;
    }
  };
```

codegeneration/python/visitor.js

```
const Python3Visitor = require(
  '../lib/antlr/Python3Visitor'
);

class Visitor extends Python3Visitor {
  visitAnd_expr(ctx) {
    return this.generateAndExpression(ctx);
  }
}
```

codegeneration/javascript/visitor.js

```
const ECMAScriptVisitor = require(
  '../lib/antlr/ECMAScriptVisitor'
);

class Visitor extends ECMAScriptVisitor {
  visitLogicalAndExpression(ctx) {
    return this.generateAndExpression(ctx);
  }
}
```

codegeneration/code-generation-visitor.js

```
export default (ANTLRVisitor) =>
  class CodeGenerationVisitor extends ANTLRVisitor {
    generateAndExpression(ctx) {
      const lhs = this.visit(ctx.getChildAt(0));
      const rhs = this.visit(ctx.getChildAt(2));

      return `${lhs} && ${rhs}`;
    }
  };
```

codegeneration/python/visitor.js

```
const Python3Visitor = require(
  '../lib/antlr/Python3Visitor'
);

class Visitor extends Python3Visitor {
  visitAnd_expr(ctx) {
    return this.generateAndExpression(ctx);
  }
}
```

codegeneration/javascript/visitor.js

```
const ECMAScriptVisitor = require(
  '../lib/antlr/ECMAScriptVisitor'
);

class Visitor extends ECMAScriptVisitor {
  visitLogicalAndExpression(ctx) {
    return this.generateAndExpression(ctx);
  }
}
```

codegeneration/code-generation-visitor.js

```
export default (ANTLRVisitor) =>
  class CodeGenerationVisitor extends ANTLRVisitor {
    generateAndExpression(ctx) {
      const lhs = this.visit(ctx.getChildAt(0));
      const rhs = this.visit(ctx.getChildAt(2));

      return `${lhs} && ${rhs}`;
    }
  };
```

Abstract away our problems (part II)

We now have one visitor per input language

How do we avoid having to specialize each visitor for every combination of languages?

- Define ”Generator” classes that generate code in methods called “emit”
- Treat Visitors as **abstract interfaces**.

codegeneration/python/generator.js

```
export default (Visitor) =>
  class Generator extends Visitor {
    emitAndExpression(lhs, rhs) {
      return `${lhs} and ${rhs}`;
    }
  };
```

codegeneration/javascript/generator.js

```
export default (Visitor) =>
  class Generator extends Visitor {
    emitAndExpression(lhs, rhs) {
      return `${lhs} && ${rhs}`;
    }
  };
```

codegeneration/python/generator.js

```
export default (Visitor) =>
  class Generator extends Visitor {
    emitAndExpression(lhs, rhs) {
      return `${lhs} and ${rhs}`;
    }
  };
```

codegeneration/javascript/generator.js

```
export default (Visitor) =>
  class Generator extends Visitor {
    emitAndExpression(lhs, rhs) {
      return `${lhs} && ${rhs}`;
    }
  };
```

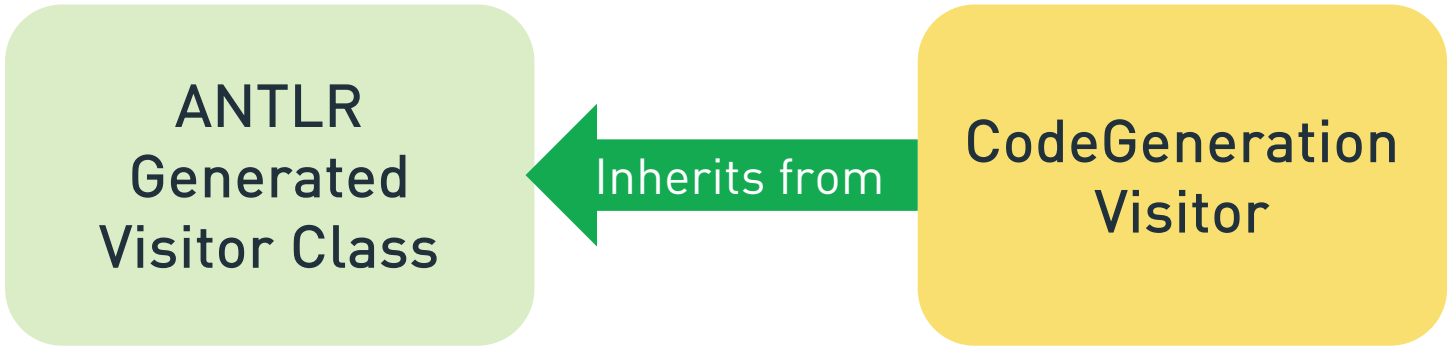
codegeneration/code-generation-visitor.js

```
export default (ANTLRVisitor) =>
  class CodeGenerationVisitor extends ANTLRVisitor {
    generateAndExpression(ctx) {
      const lhs = this.visit(ctx.getChildAt(0));
      const rhs = this.visit(ctx.getChildAt(2));

      return this.emitAndExpression(lhs, rhs);
    }
  };
```

Composable Transpiler

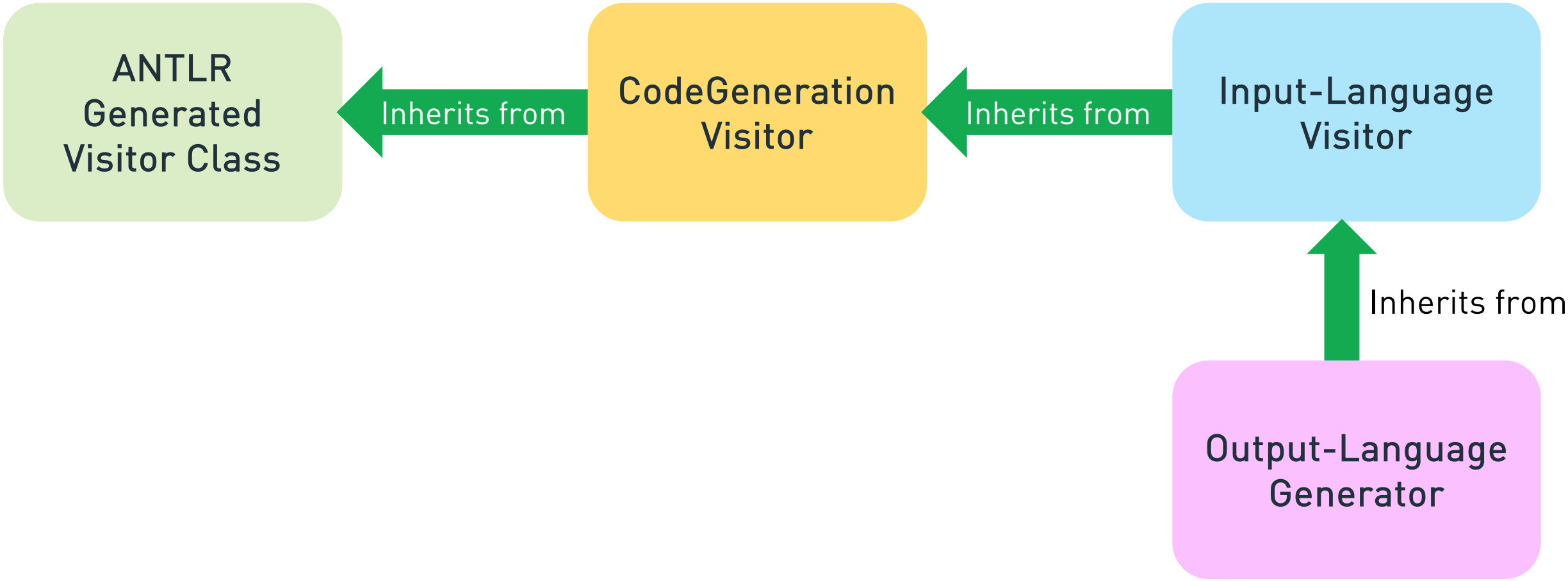
ANTLR
Generated
Visitor Class

Composable Transpiler

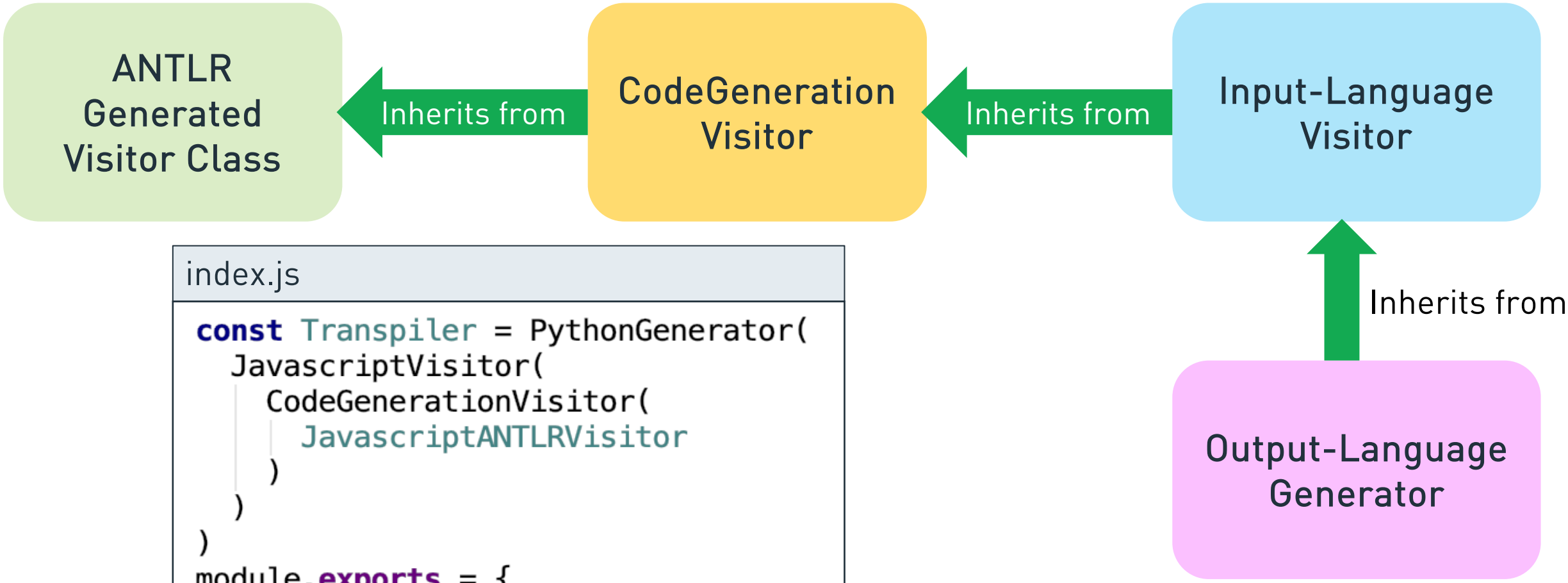


Composable Transpiler

Composable Transpiler



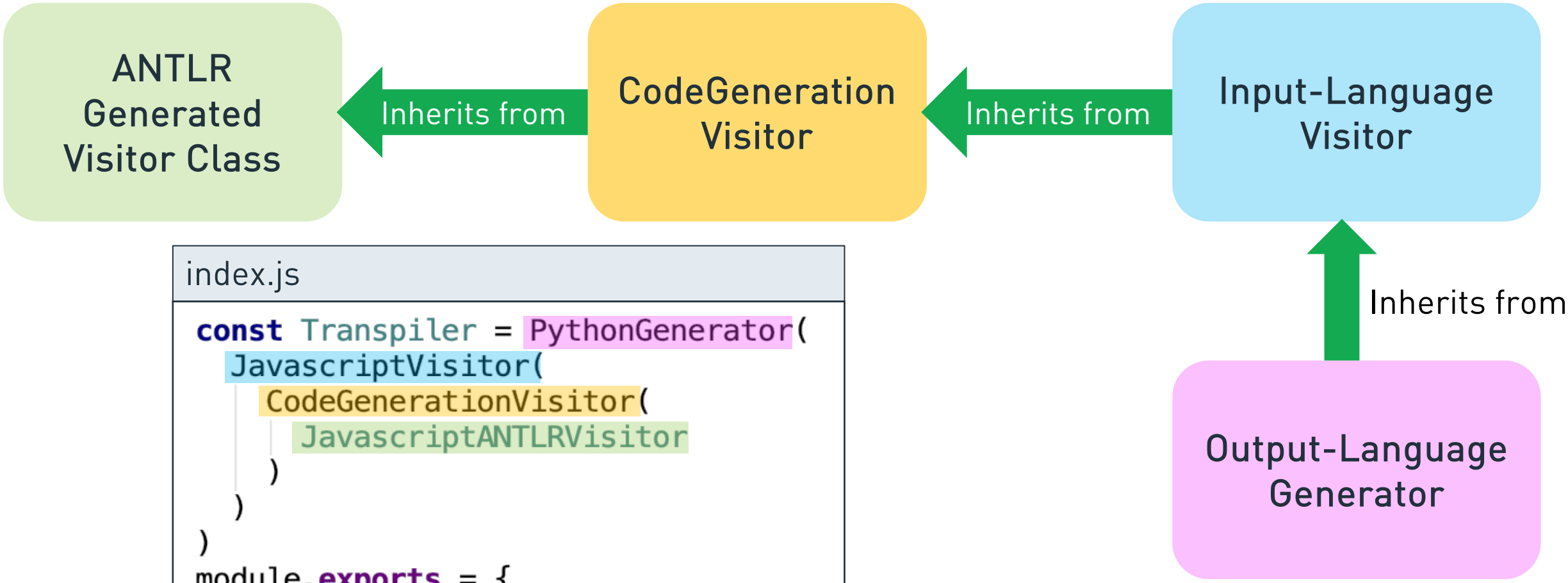
Composable Transpiler



index.js

```
const Transpiler = PythonGenerator(
  JavascriptVisitor(
    CodeGenerationVisitor(
      JavascriptANTLRVisitor
    )
  )
)
module.exports = {
  javascript: {
    python: new Transpiler()
  }
}
```

Composable Transpiler



index.js

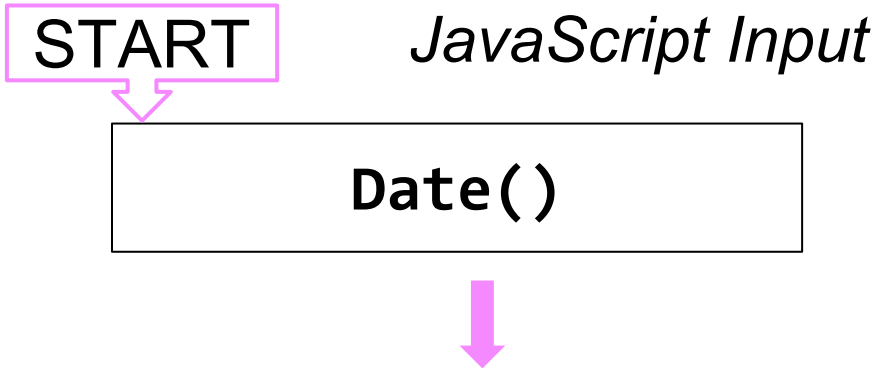
```
const Transpiler = PythonGenerator(
  JavascriptVisitor(
    CodeGenerationVisitor(
      JavascriptANTLRVisitor
    )
  )
)
module.exports = {
  javascript: {
    python: new Transpiler()
  }
}
```

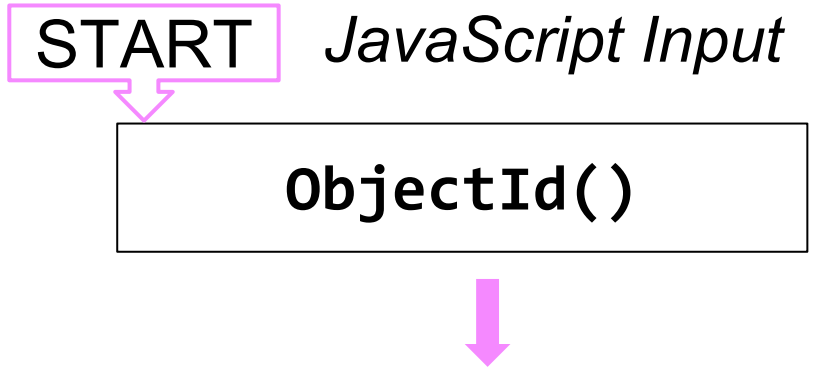
What about functions or variables?

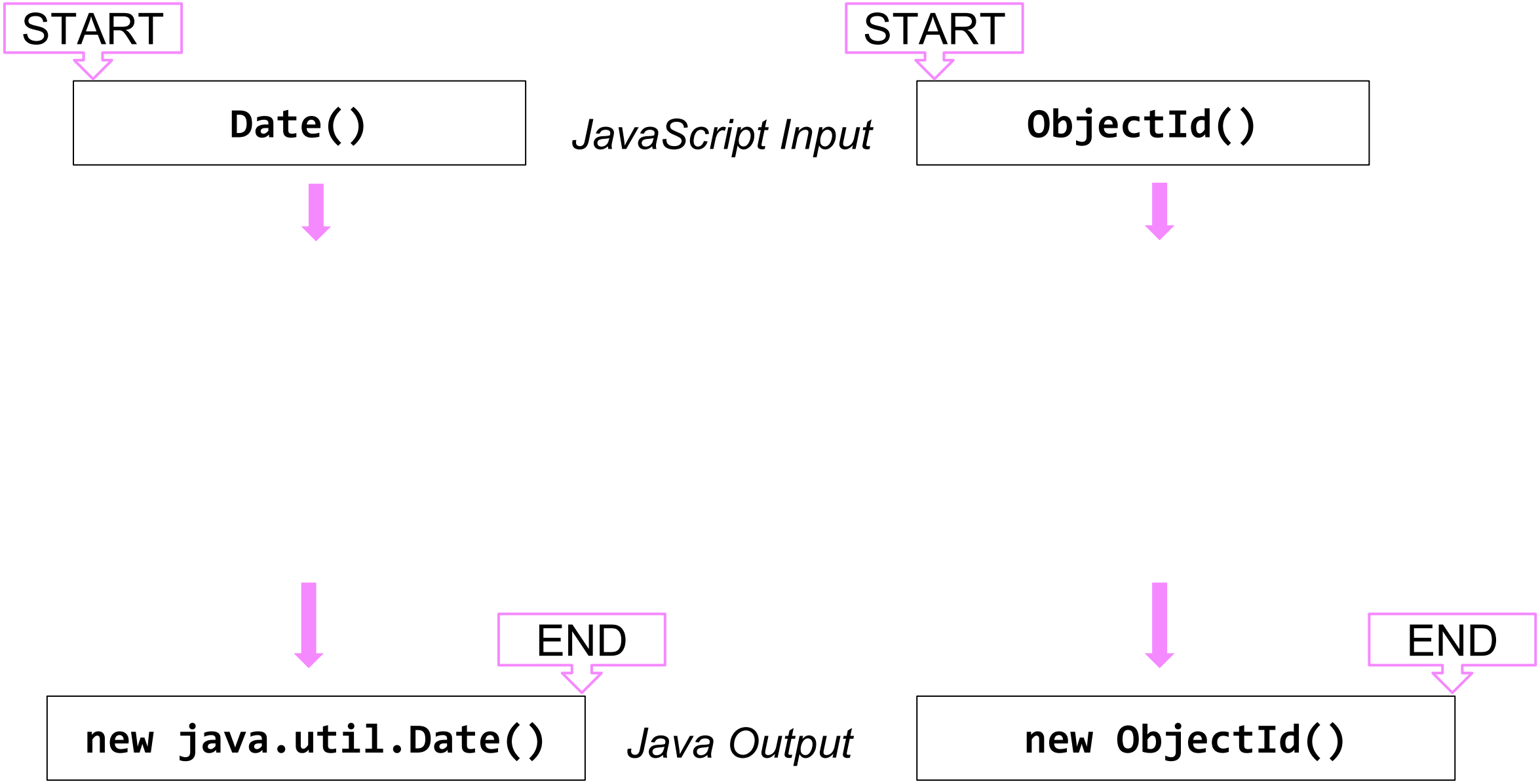
We need to support native language features as well as BSON-specific types.

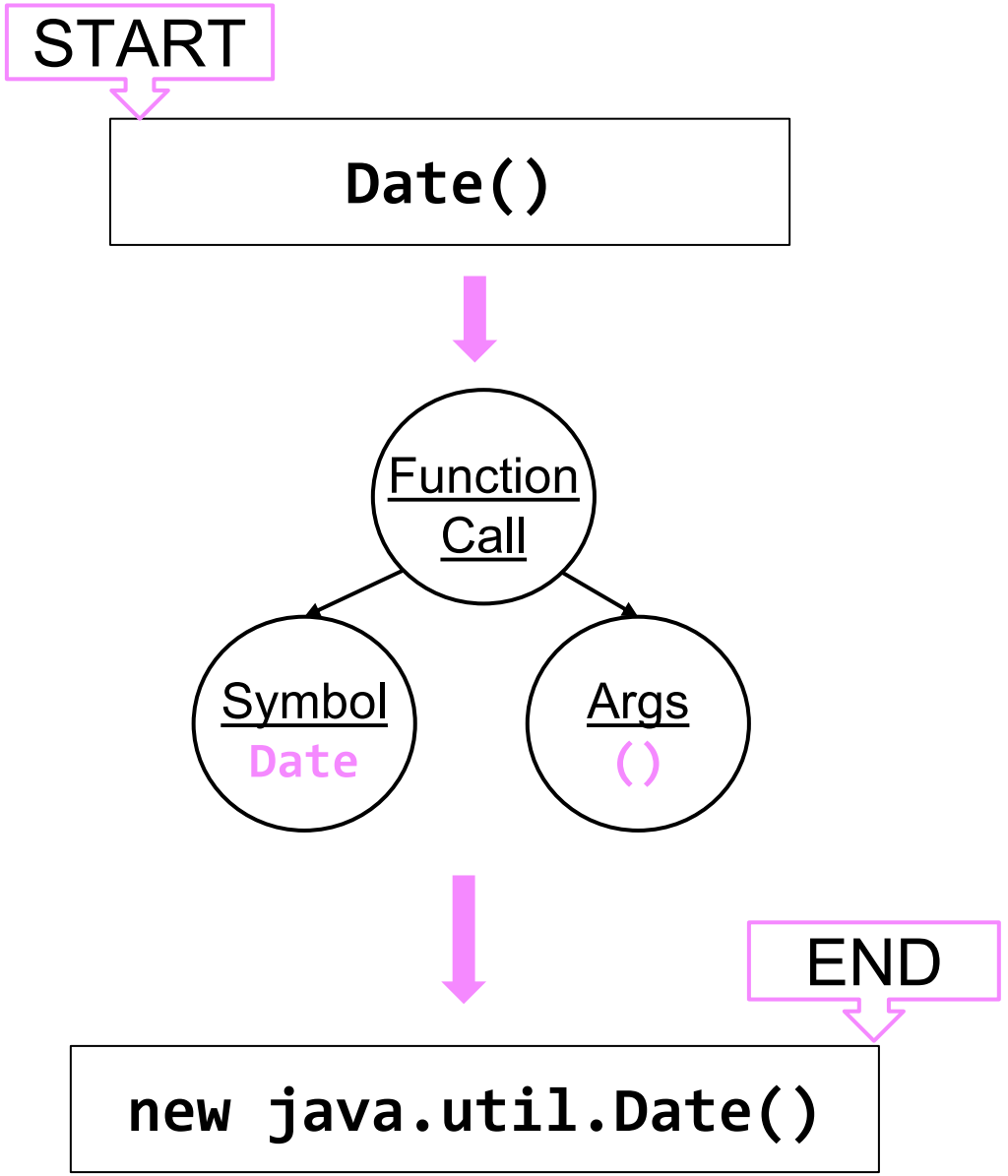
`ObjectId`, **`Date`**, **`Decimal128`**, **`Timestamp`**, etc...

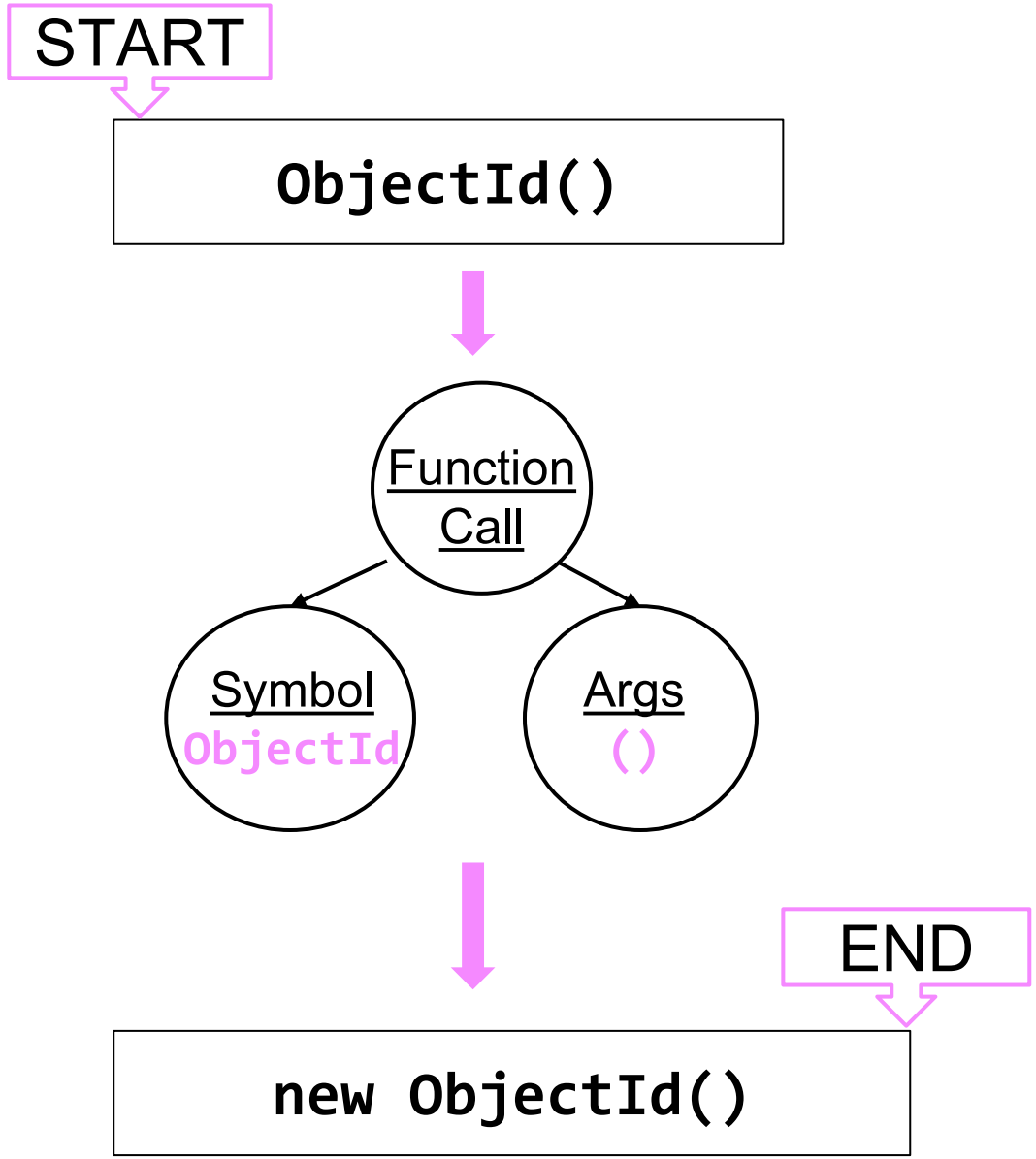
How can we tell the difference between variables?


START
JavaScript Input
Date()

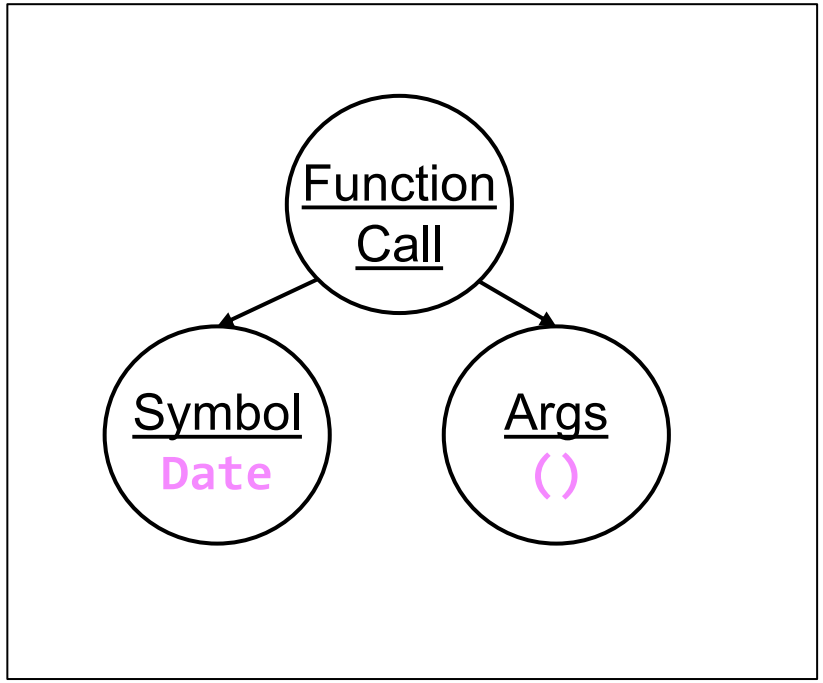

START
JavaScript Input
ObjectId()

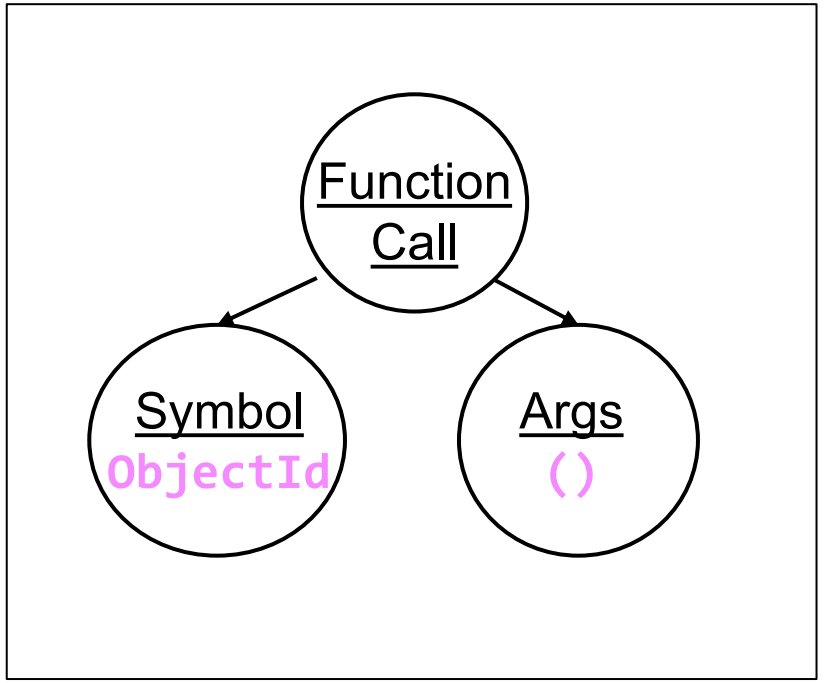

START
Date()
JavaScript Input
START
ObjectId()
END
new java.util.Date()
Java Output
END
new ObjectId()


START
Date()
Function Call
Symbol
Date
Args
()
END
new java.util.Date()


START
ObjectId()
Function Call
Symbol
ObjectId
Args
()
END
new ObjectId()

To the visitor, these two trees look the same...





The same `visit` methods are going to be called for both...


We need a
Symbol Table!

Symbol Table

- Need a place to keep track of all the symbols, i.e. variable or function names.
- When the visitor reaches a Symbol node, it looks it up in the **Symbol Table**
- This is also a convenient place to differentiate between output languages...

Visiting a Symbol

```
visitSymbol(node) {
        const name = node.symbol;
        node.type = this.Symbols[name];
        if (node.type === undefined) {
                throw new ReferenceError(`Symbol ${name} is undefined`);
        }
        return name;
}
```

Visiting a Symbol

```
visitSymbol(node) {
        const name = node.symbol;
        node.type = this.Symbols[name];
        if (node.type === undefined) {
                throw new ReferenceError(`Symbol ${name} is undefined`);
        }
        return name;
}
```

Look up the name of the symbol in the table and “decorate” the node with the results

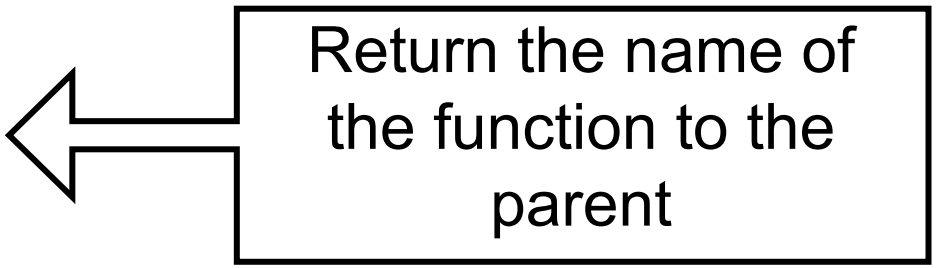
Visiting a Symbol

```
visitSymbol(node) {
        const name = node.symbol;
        node.type = this.Symbols[name];
        if (node.type === undefined) {
                throw new ReferenceError(`Symbol ${name} is undefined`);
        }
        return name;
}
```

If it's not in the table, throw an exception

Visiting a Symbol

```
visitSymbol(node) {
        const name = node.symbol;
        node.type = this.Symbols[name];
        if (node.type === undefined) {
                throw new ReferenceError(`Symbol ${name} is undefined`);
        }
        return name;
}
```



What's in the Symbol Table?

```
ObjectId:
    id: "ObjectId"
    callable: *constructor
    args:
        - [ *StringType, *NumericType, null ]
    type: *ObjectIdType
    attr: {}
    template: *ObjectIdSymbolTemplate
    argsTemplate: *ObjectIdSymbolArgsTemplate
```

What's in the Symbol Table?

```
ObjectId:
    id: "ObjectId"
```

The name of the attribute.

Mostly used for error reporting.

What's in the Symbol Table?

```
ObjectId:
    id: "ObjectId"
    callable: *constructor
```

There are 3 types of symbol:

- *func: a function name
- *constructor: also a function name, but may require a "new"
- *var: a variable. Indicates that the symbol cannot be called.

What's in a Symbol?

```
ObjectId:
    id: "ObjectId"
    callable: *constructor
    args:
        - [ *StringType, *NumericType, null ]
```

If the symbol is callable, this is where the arguments are defined. Each element in the array is a positional argument and contains the list of acceptable types. So ObjectId accepts one string or number argument, or no arguments at all.

What's in a Symbol?

```
ObjectId:
    id: "ObjectId"
    callable: *constructor
    args:
        - [ *StringType, *NumericType, null ]
    type: *ObjectIdType
```

The return type of the function, or if the symbol is a variable, the type of the variable.

What's in a Symbol?

```
ObjectId:
    id: "ObjectId"
    callable: *constructor
    args:
        - [ *StringType, *NumericType, null ]
    type: *ObjectIdType
    attr: {...}
```

Any attributes of the symbol. This is a sub-symbol table, i.e. a mapping of names to symbols. Ex: `ObjectId.fromDate()`

What's in a Symbol?

```
ObjectId:
    id: "ObjectId"
    callable: *constructor
    args:
        - [ *StringType, *NumericType, null ]
    type: *ObjectIdType
    attr: {}
    template: *ObjectIdSymbolTemplate
```

These are functions that accept strings and return strings.

Templates

Simple functions that accept strings and return strings

Responsible for doing the string transformations from one language syntax to another language's syntax

These are specific to the output language and **defined in a separate file that is loaded when the compiler is initialized.**

Each output language has a file
(in YAML)
where the templates are defined.

Symbol File (input language)

```
Double:
    id: "Double"
    code: 104
    callable: *constructor
    args:
        - [ *NumericType, *StringType ]
    type: *DoubleType
    attr: {}
    template: *DoubleSymbolTemplate
```

Template File (output language)

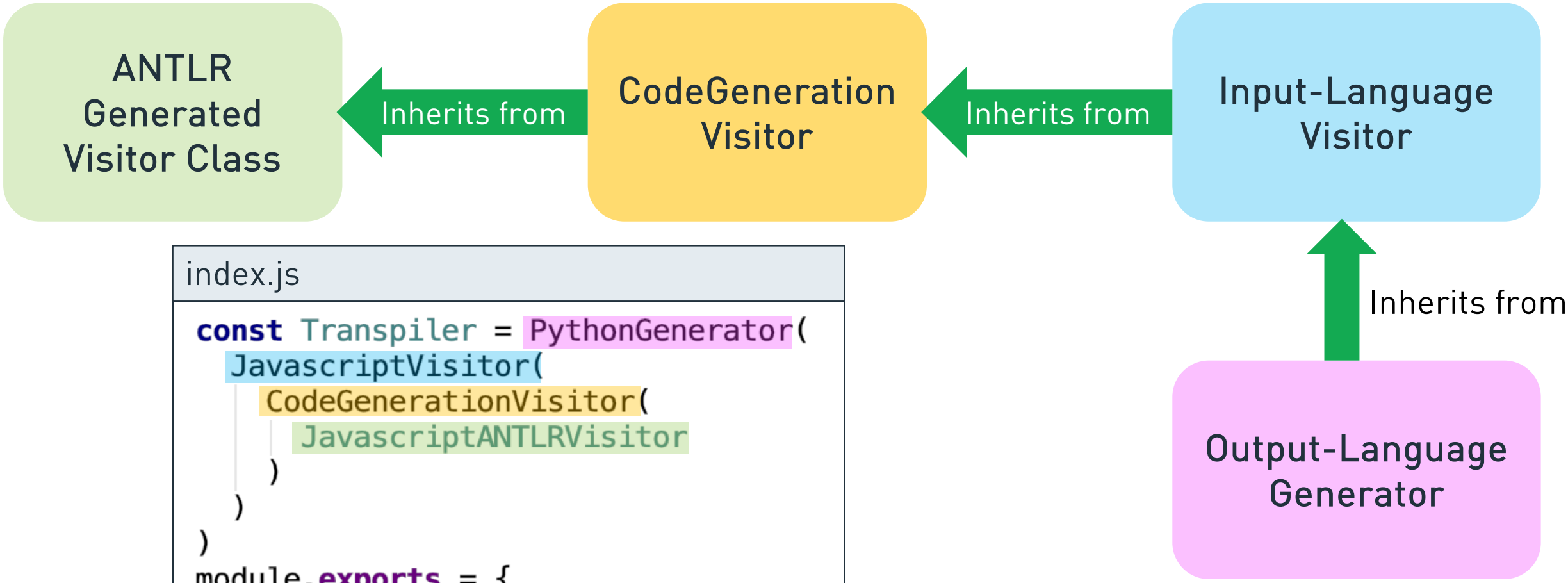
```
DoubleSymbolTemplate: &DoubleSymbolTemplate
    () => {
        return 'float';
    }
```

Symbol File + Template File

=

Symbol Table

Composable Transpiler



index.js

```
const Transpiler = PythonGenerator(
  JavascriptVisitor(
    CodeGenerationVisitor(
      JavascriptANTLRVisitor
    )
  )
)
module.exports = {
  javascript: {
    python: new Transpiler()
  }
}
```

Composable Transpiler

index.js

```
const Symbols = require('javascript/symbols.yaml');
const Templates = require('python/templates.yaml');

const SymbolTable = yaml.load(Symbols + Templates);

const Transpiler = PythonGenerator(
  JavascriptVisitor(
    CodeGenerationVisitor(
      JavascriptANTLRVisitor
    )
  )
)
module.exports = {
  javascript: {
    python: new Transpiler(SymbolTable)
  }
}
```

To Recap

ANTLR creates a tree from the user input

We visit the ANTLR-generated tree using visitors

When the visitor reaches a symbol, it looks up metadata in the Symbol Table.

- The metadata includes **template** functions that specify what code should be generated

So how do I add my own output language to Compass?

Add your own template file!!

All you need to do to add an output language is fill out the templates!

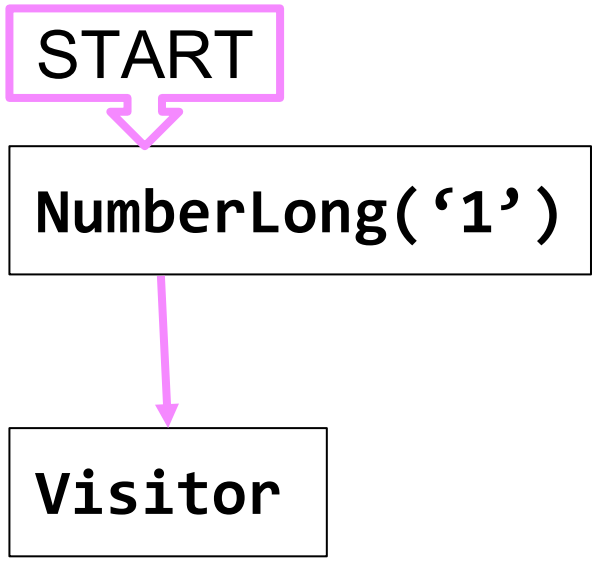
`symbols/sample_template.yaml`

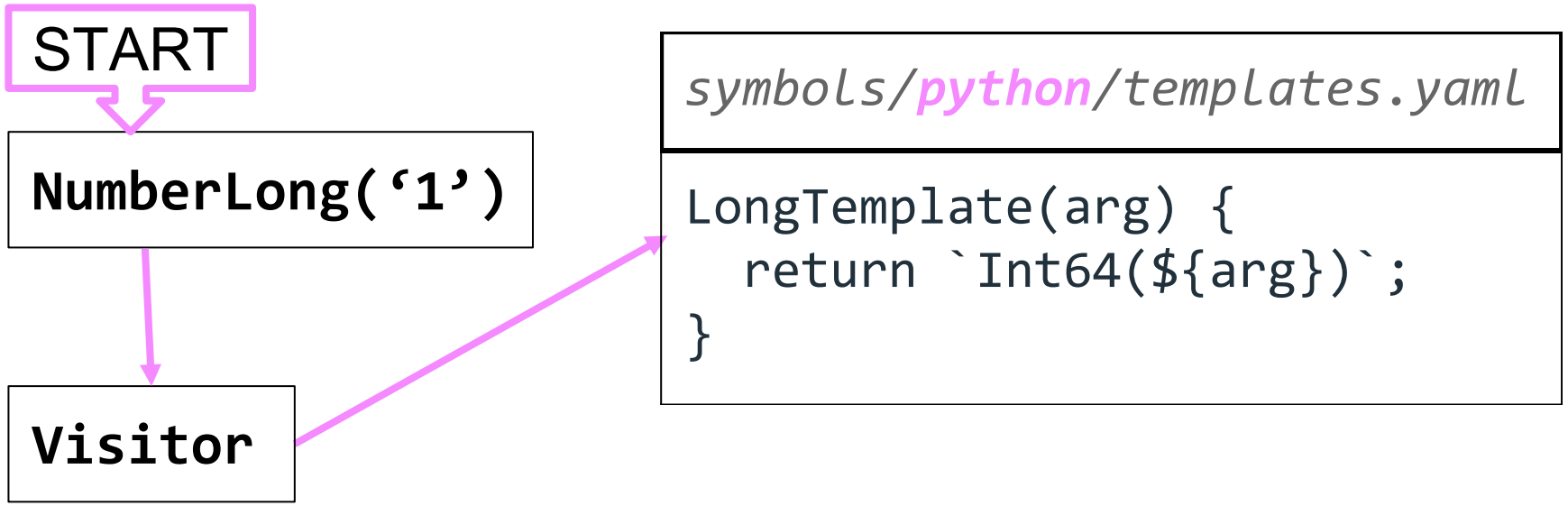
There is a skeleton template file available

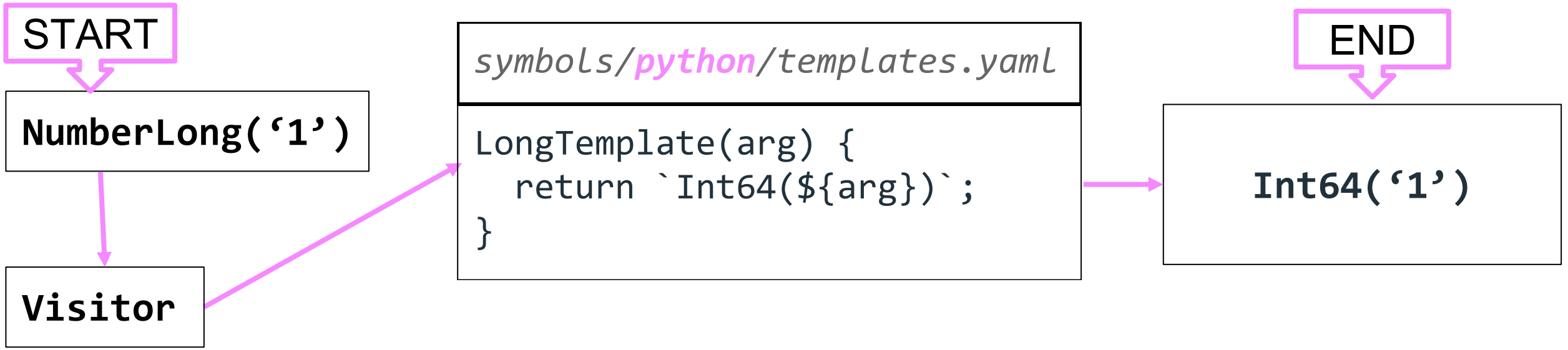
To add a new output language:

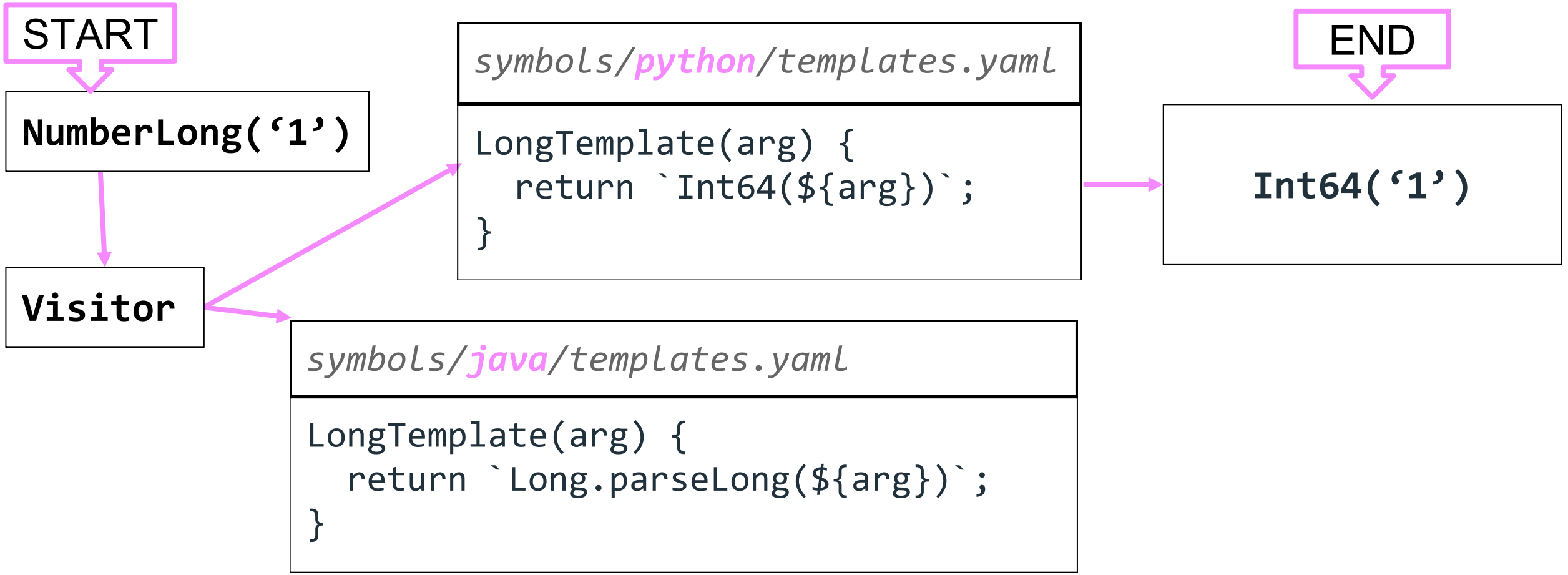
- fill out each template with the correct translation to your language.

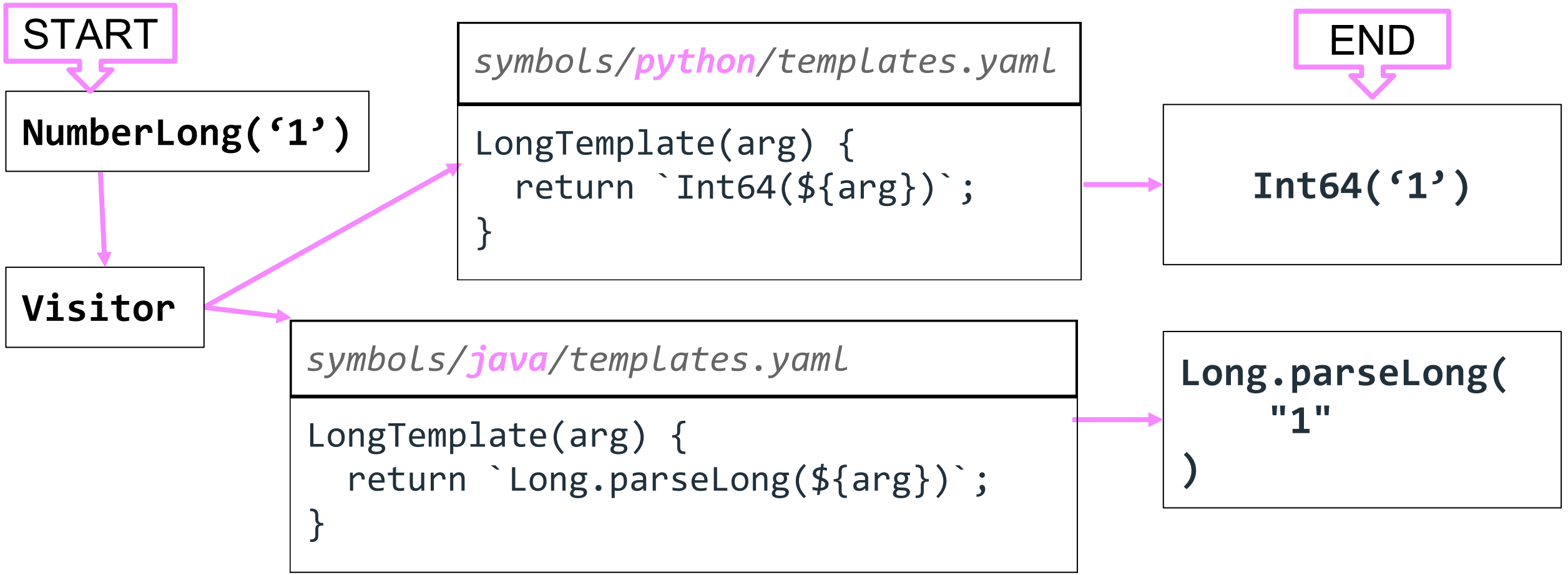
Templates mostly apply to symbols, but there are also templates for literals and other syntax.

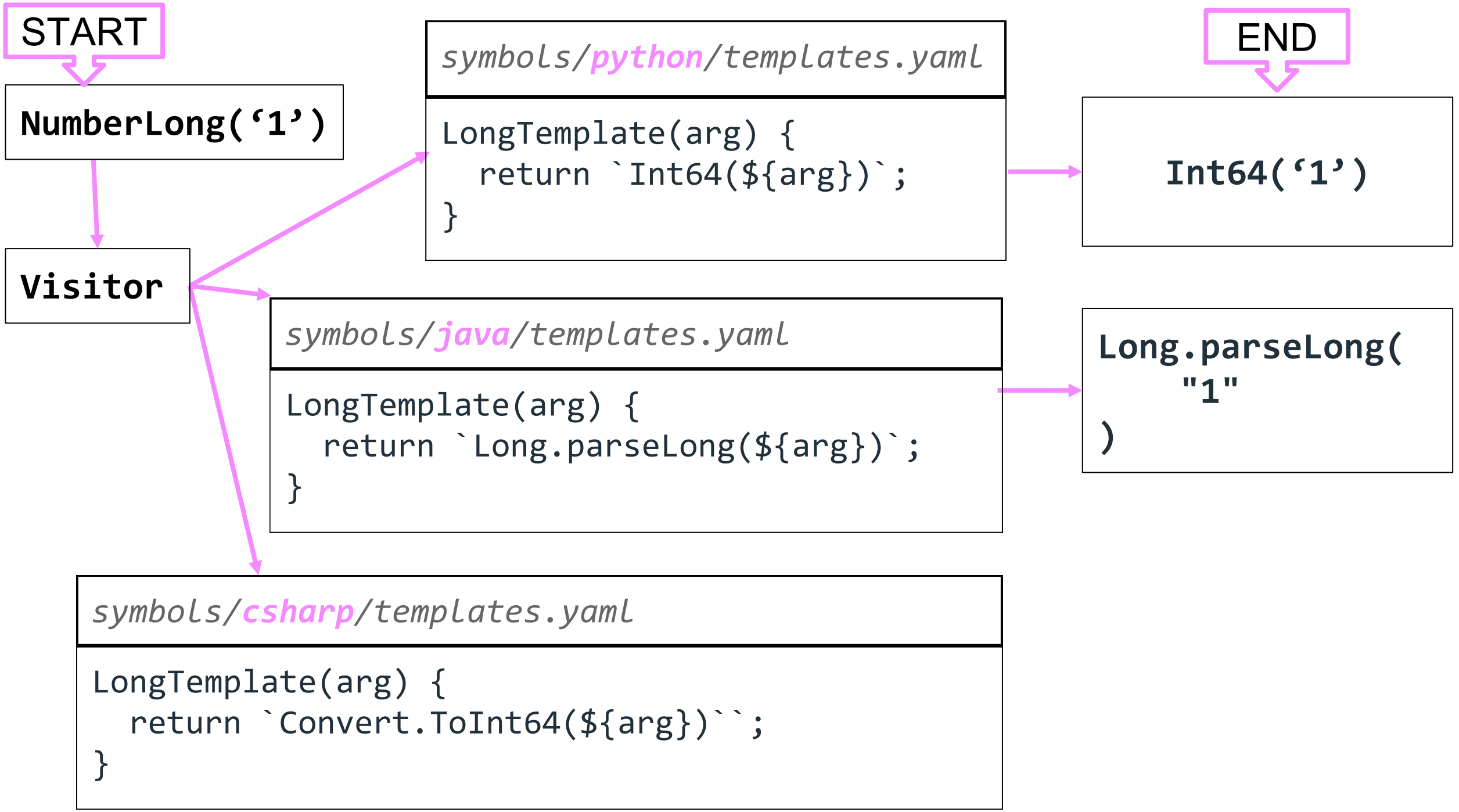

START
NumberLong(‘1’)
Visitor

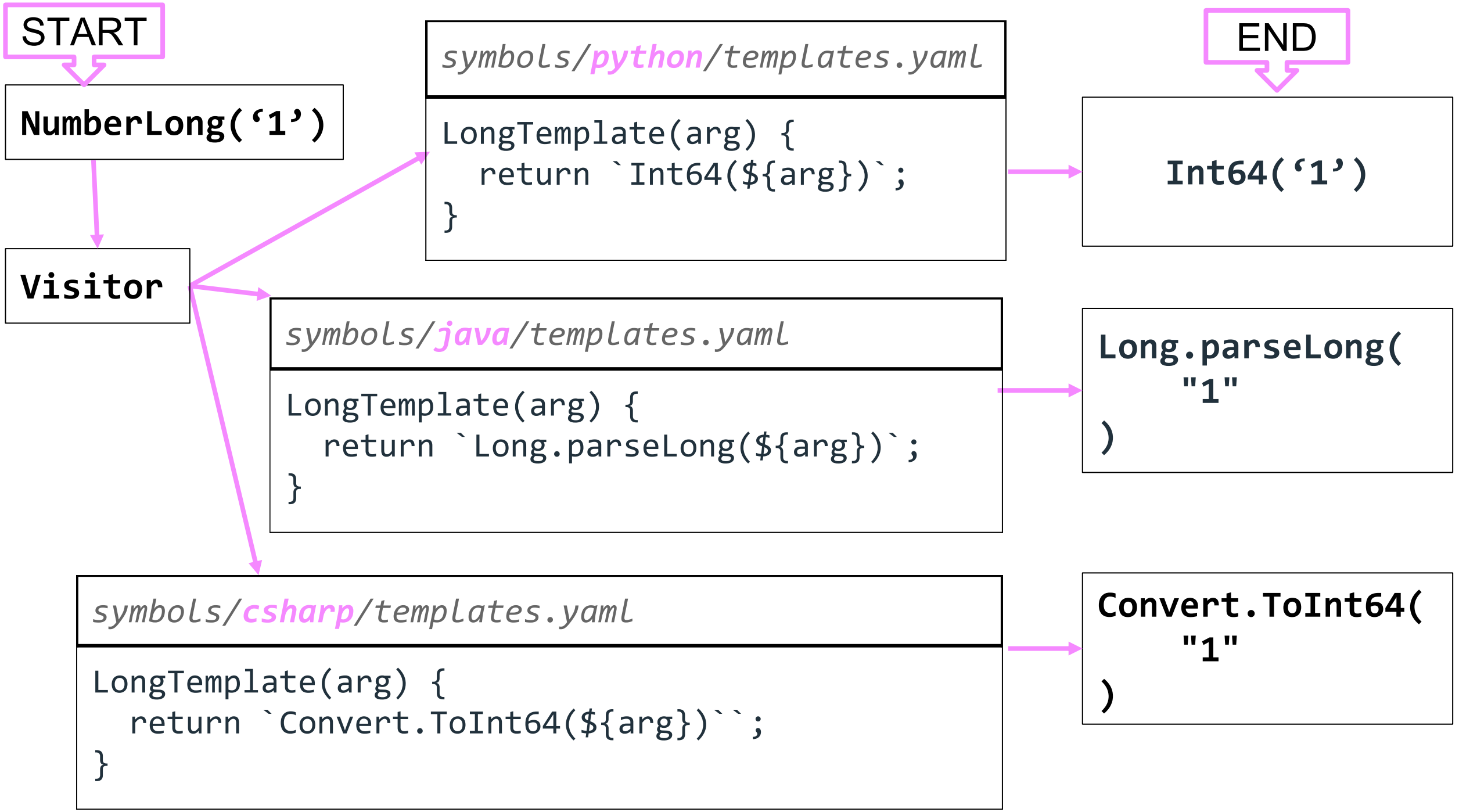

START
NumberLong(‘1’)
Visitor
symbols/python/templates.yaml
LongTemplate(arg) {
return `Int64(${arg})`;
}


START
NumberLong('1')
Visitor
symbols/python/templates.yaml
LongTemplate(arg) {
return `Int64(${arg})`;
}
END
Int64('1')


START
NumberLong('1')
Visitor
symbols/python/templates.yaml
LongTemplate(arg) {
return `Int64(${arg})`;
}
symbols/java/templates.yaml
LongTemplate(arg) {
return `Long.parseLong(${arg})`;
}
END
Int64('1')


START
NumberLong('1')
Visitor
symbols/python/templates.yaml
LongTemplate(arg) {
return `Int64(${arg})`;
}
END
Int64('1')
symbols/java/templates.yaml
LongTemplate(arg) {
return `Long.parseLong(${arg})`;
}
Long.parseLong(
"1"
)


START
NumberLong('1')
Visitor
symbols/python/templates.yaml
LongTemplate(arg) {
return `Int64(${arg})`;
}
END
Int64('1')
symbols/java/templates.yaml
LongTemplate(arg) {
return `Long.parseLong(${arg})`;
}
Long.parseLong(
"1"
)
symbols/csharp/templates.yaml
LongTemplate(arg) {
return `Convert.ToInt64(${arg})``;
}


START
NumberLong('1')
Visitor
symbols/python/templates.yaml
LongTemplate(arg) {
return `Int64(${arg})`;
}
END
Int64('1')
symbols/java/templates.yaml
LongTemplate(arg) {
return `Long.parseLong(${arg})`;
}
Long.parseLong(
"1"
)
symbols/csharp/templates.yaml
LongTemplate(arg) {
return `Convert.ToInt64(${arg})``;
}
Convert.ToInt64(
"1"
)

Expand Templates to Literals

Can apply the same method to literals

Example: Object Literals

- Python: `{'k': 1}`
- JS: `{k: 1}`
- C#: `new BsonDocument()`
- Java: `new Document()`


Go forth and write templates!

Ruby, PHP, Go, R, Rust, C & more!

- Want to add an **output language**?
  - Just fill out a symbol table file!
- Want to write an **input language**?
  - Write a visitor

Expanded the syntax to include driver usage

Export Query To Language

My Pipeline:

```
{
    answer: {
        $type: 'Array'
    }
}
```

Export Pipeline To: PYTHON 3

```
from pymongo import MongoClient
client = MongoClient('mongodb://localhost:27017')
query = {
    "$answer": {
        "$type": "Array"
        }
    }
}
result = client['myDB']['myColl'].find_one(query)

```

☑ Include Import Statements

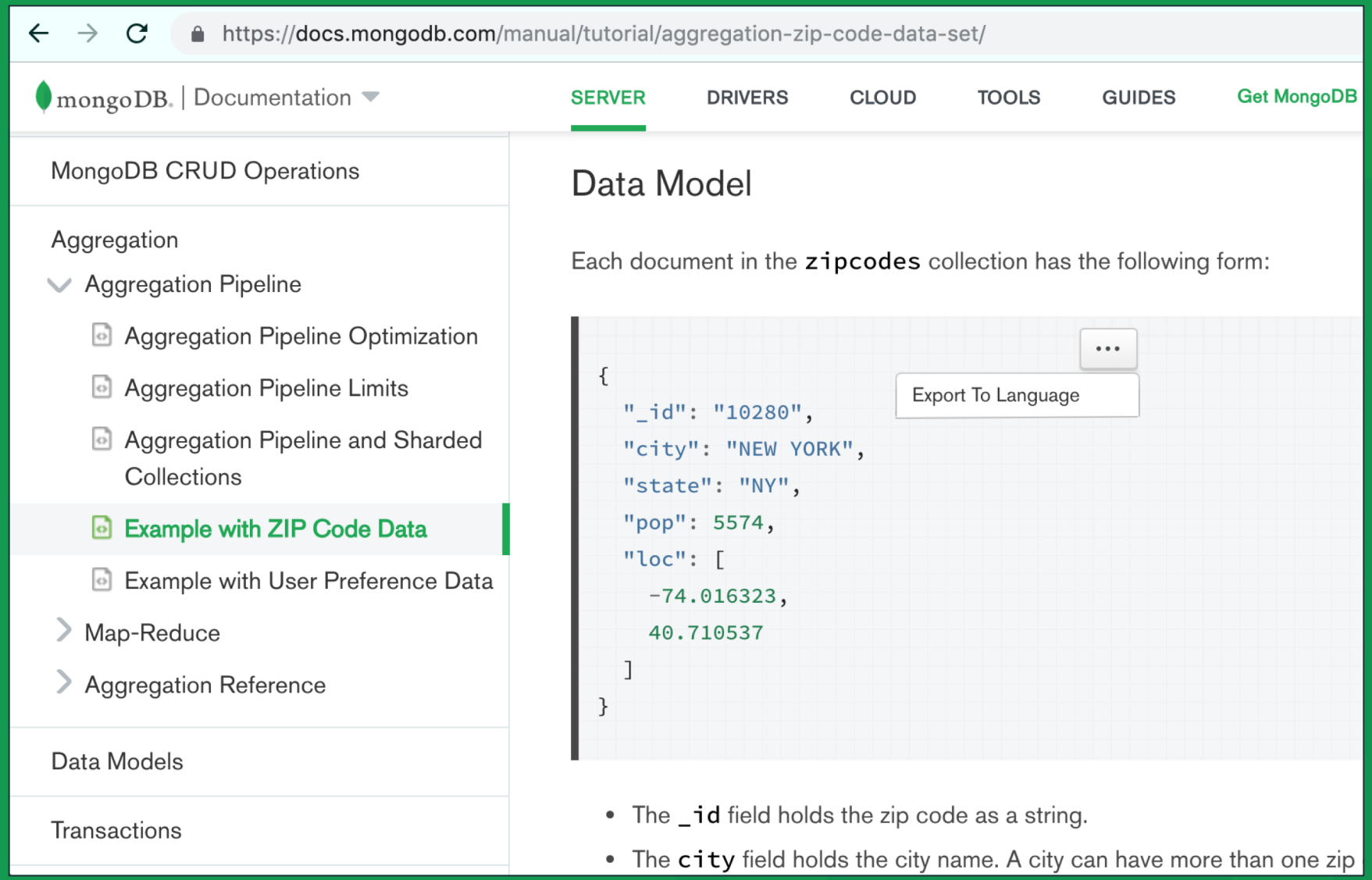
☑ Include Driver Code

CLOSE

Future Features!

We now have a pluggable transpiler from any language BSON to any language BSON....what can we do with it?

Generate examples for MongoDB University



```
mongod  ⌘1    mongo  ⌘2

annaherlihy@auckland:~ $ mongo
MongoDB shell version v3.6.3
connecting to: mongodb://127.0.0.1:27017
MongoDB server version: 3.6.3
Server has startup warnings:
2019-04-09T14:22:23.429+0200 I CONTROL  [initandlisten]
2019-04-09T14:22:23.429+0200 I CONTROL  [initandlisten]
2019-04-09T14:22:23.429+0200 I CONTROL  [initandlisten]
2019-04-09T14:22:23.429+0200 I CONTROL  [initandlisten]
2019-04-09T14:22:23.429+0200 I CONTROL  [initandlisten]
2019-04-09T14:22:23.429+0200 I CONTROL  [initandlisten]
2019-04-09T14:22:23.429+0200 I CONTROL  [initandlisten]
2019-04-09T14:22:23.429+0200 I CONTROL  [initandlisten]
2019-04-09T14:22:23.429+0200 I CONTROL  [initandlisten]
2019-04-09T14:22:23.429+0200 I CONTROL  [initandlisten]
2019-04-09T14:22:23.429+0200 I CONTROL  [initandlisten]
MongoDB Enterprise > use language python
using language python
MongoDB Enterprise > db['myDB']['myColl'].find_one({
...     'answer': {
...         '$type': 'Array'
...     }
... })
```

Put it in front of the shell!

Expand it to support 100% language syntax!

Thanks to the Compass Team!

- ★ Alena Khineika
- ★ Irina Shestak
- ★ Durran Jordan

Thank you!

Everything I said, in much more detail:

github.com/mongodb-js/bson-transpilers

> CONTRIBUTING.md

Questions?

compass@mongodb.com or anna@mongodb.com


MONGODB