Metaprogramming

Writing programs that manipulate programs as data

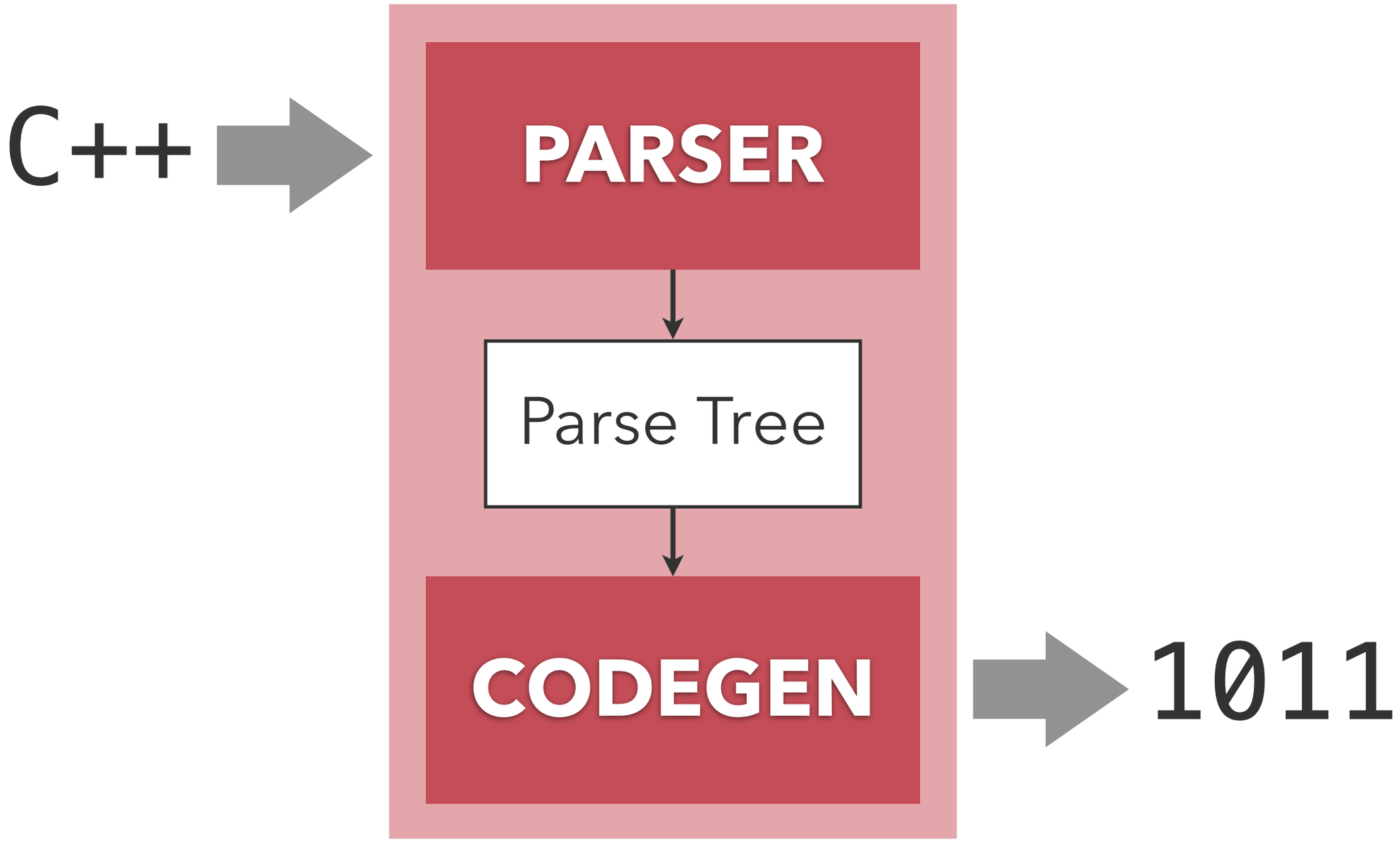
Static Metaprogramming

Writing programs that manipulate code

COMPILER


Code
COMPILER
Code


C++
COMPILER
1011

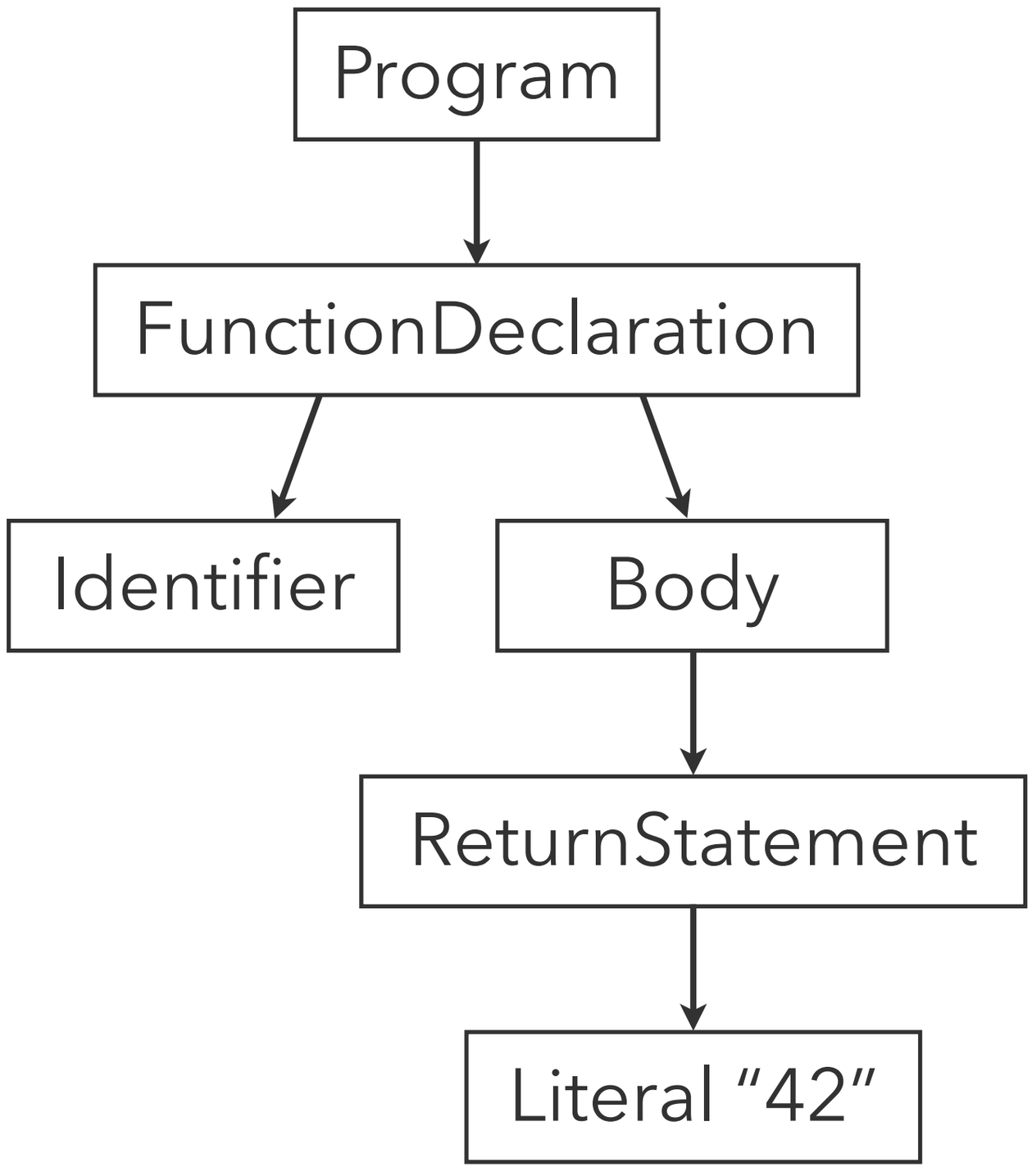

C++
PARSER
Parse Tree
CODEGEN
1011

Esprima

A high-performance JavaScript parser written in JavaScript.

Also: estraverse & escodegen

```
var esprima = require('esprima');

esprima.parse("                 \
function getAnswer() {  \
  return 42;                    \
}                               \
");
```


Program
FunctionDeclaration
Identifier
Body
ReturnStatement
Literal “42”

```
{
    "type": "Program",
    "body": [
        {
            "type": "FunctionDeclaration",
            "id": {
                "type": "Identifier",
                "name": "getAnswer"
            },
            "params": [],
            "defaults": [],
            "body": {
                "type": "BlockStatement",
                "body": [
                    {
                        "type": "ReturnStatement",
                        "argument": {
                            "type": "Literal",
                            "value": 42,
                            "raw": "42"
                        }
                    }
                ]
            },
        }
    ]
}
```

```
function checkStyle(code, filename) {
  var ast = esprima.parse(code, parseOptions);
  var errors = [];
  estraverse.traverse(ast, {
    enter: function(node, parent) {
      if (node.type === 'VariableDeclaration')
        checkVariableNames(node, errors);
    }
  });
  return formatErrors(code, errors, filename);
}

function checkVariableNames(node, errors) {
  _.each(node.declarations, function(decl) {
    if (decl.id.name.indexOf('_') >= 0) {
      return errors.push({
        location: decl.loc,
        message: 'Use camelCase, not hacker_style!'
      });
    }
  });
}
```

```
function checkStyle(code, filename) {
  var ast = esprima.parse(code, parseOptions);
  var errors = [];
  estraverse.traverse(ast, {
    enter: function(node, parent) {
      if (node.type === 'VariableDeclaration')
        checkVariableNames(node, errors);
    }
  });
  return formatErrors(code, errors, filename);
}

function checkVariableNames(node, errors) {
  _.each(node.declarations, function(decl) {
    if (decl.id.name.indexOf('_') >= 0) {
      return errors.push({
        location: decl.loc,
        message: 'Use camelCase, not hacker_style!'
      });
    }
  });
}
```

```
function checkStyle(code, filename) {
  var ast = esprima.parse(code, parseOptions);
  var errors = [];
  estraverse.traverse(ast, {
    enter: function(node, parent) {
      if (node.type === 'VariableDeclaration')
        checkVariableNames(node, errors);
    }
  });
  return formatErrors(code, errors, filename);
}

function checkVariableNames(node, errors) {
  _.each(node.declarations, function(decl) {
    if (decl.id.name.indexOf('_') >= 0) {
      return errors.push({
        location: decl.loc,
        message: 'Use camelCase, not hacker_style!'
      });
    }
  });
}
```

```
function checkStyle(code, filename) {
  var ast = esprima.parse(code, parseOptions);
  var errors = [];
  estraverse.traverse(ast, {
    enter: function(node, parent) {
      if (node.type === 'VariableDeclaration')
        checkVariableNames(node, errors);
    }
  });
  return formatErrors(code, errors, filename);
}
```

```
function checkVariableNames(node, errors) {
  _.each(node.declarations, function(decl) {
    if (decl.id.name.indexOf('_') >= 0) {
      return errors.push({
        location: decl.loc,
        message: 'Use camelCase, not hacker_style!'
      });
    }
  });
}
```

```
function checkStyle(code, filename) {
  var ast = esprima.parse(code, parseOptions);
  var errors = [];
  estraverse.traverse(ast, {
    enter: function(node, parent) {
      if (node.type === 'VariableDeclaration')
        checkVariableNames(node, errors);
    }
  });
  return formatErrors(code, errors, filename);
}

function checkVariableNames(node, errors) {
  _.each(node.declarations, function(decl) {
    if (decl.id.name.indexOf('_') >= 0) {
      return errors.push({
        location: decl.loc,
        message: 'Use camelCase, not hacker_style!'
      });
    }
  });
}
```

```
var foo = bar;
var this_is_bad = 3;
function blah() {
  return function x() {
    var oops_another_one;
  }
}
```

```
var foo = bar;
var this_is_bad = 3;
function blah() {
  return function x() {
    var oops_another_one;
  }
}
```

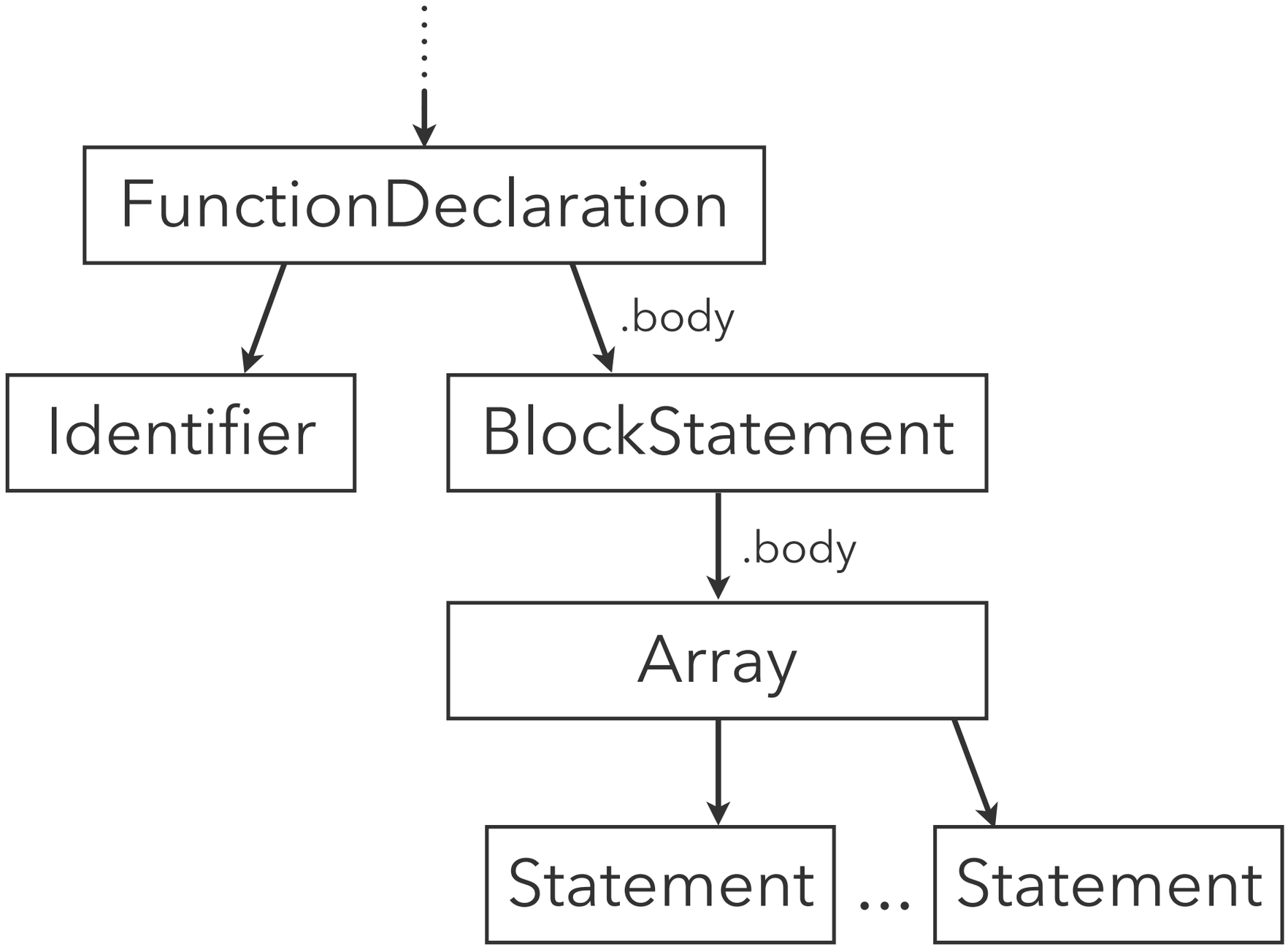
```
[ 'Line 1, column 34: Use camelCase for variable names, not
hacker_style.',
  'Line 1, column 119: Use camelCase for variable names, not
hacker_style.' ]
```

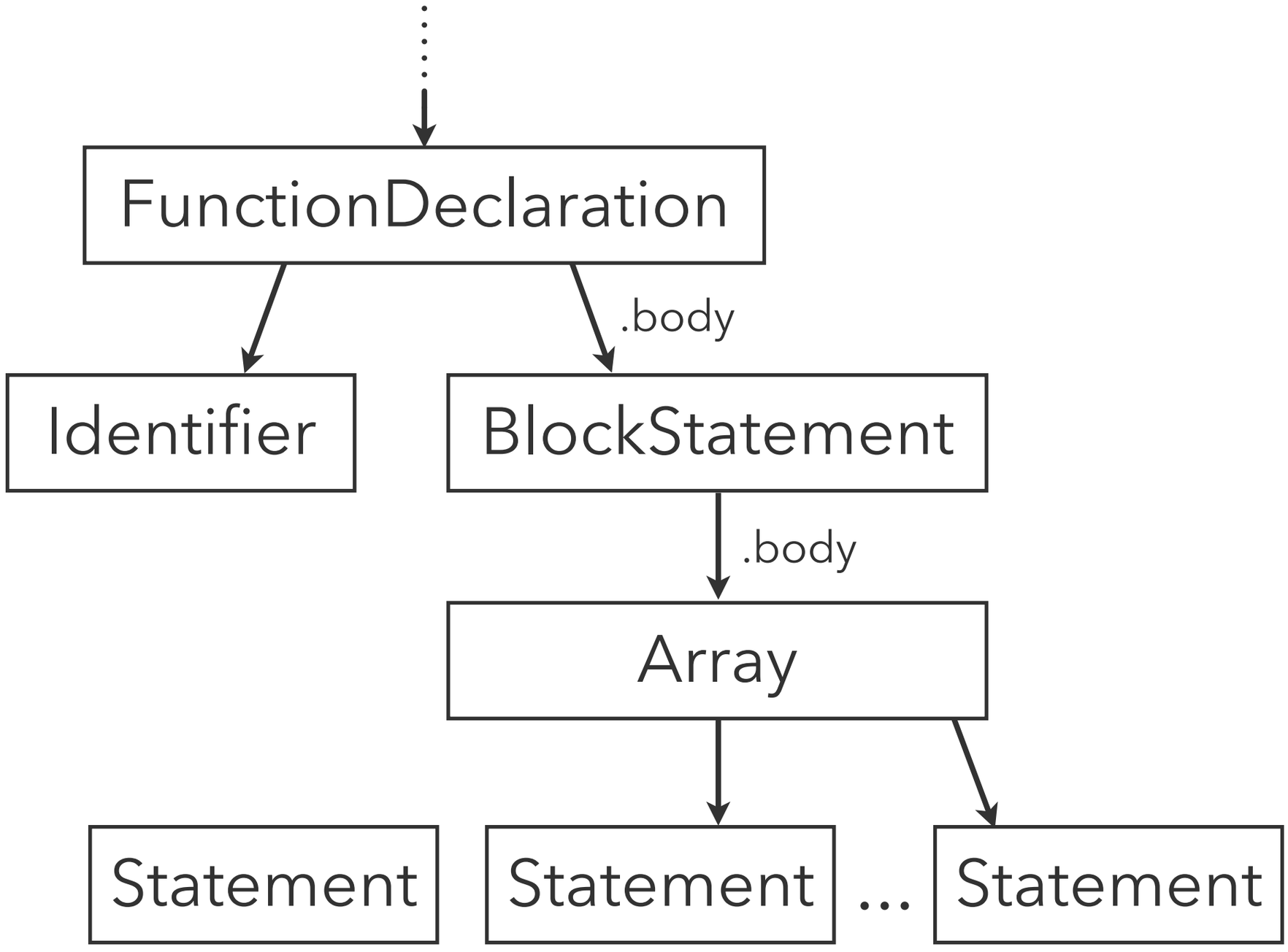
```
function addLogging(code) {
  var ast = esprima.parse(code);
  estraverse.traverse(ast, {
    enter: function(node, parent) {
      if (node.type === 'FunctionDeclaration'
          || node.type === 'FunctionExpression') {
        addBeforeCode(node);
      }
    }
  });
  return escodegen.generate(ast);
}

function addBeforeCode(node) {
  var name = node.id ? node.id.name : '<anonymous function>';
  var beforeCode = "console.log('Entering " + name + "()');";
  var beforeNodes = esprima.parse(beforeCode).body;
  node.body.body = beforeNodes.concat(node.body.body);
}
```

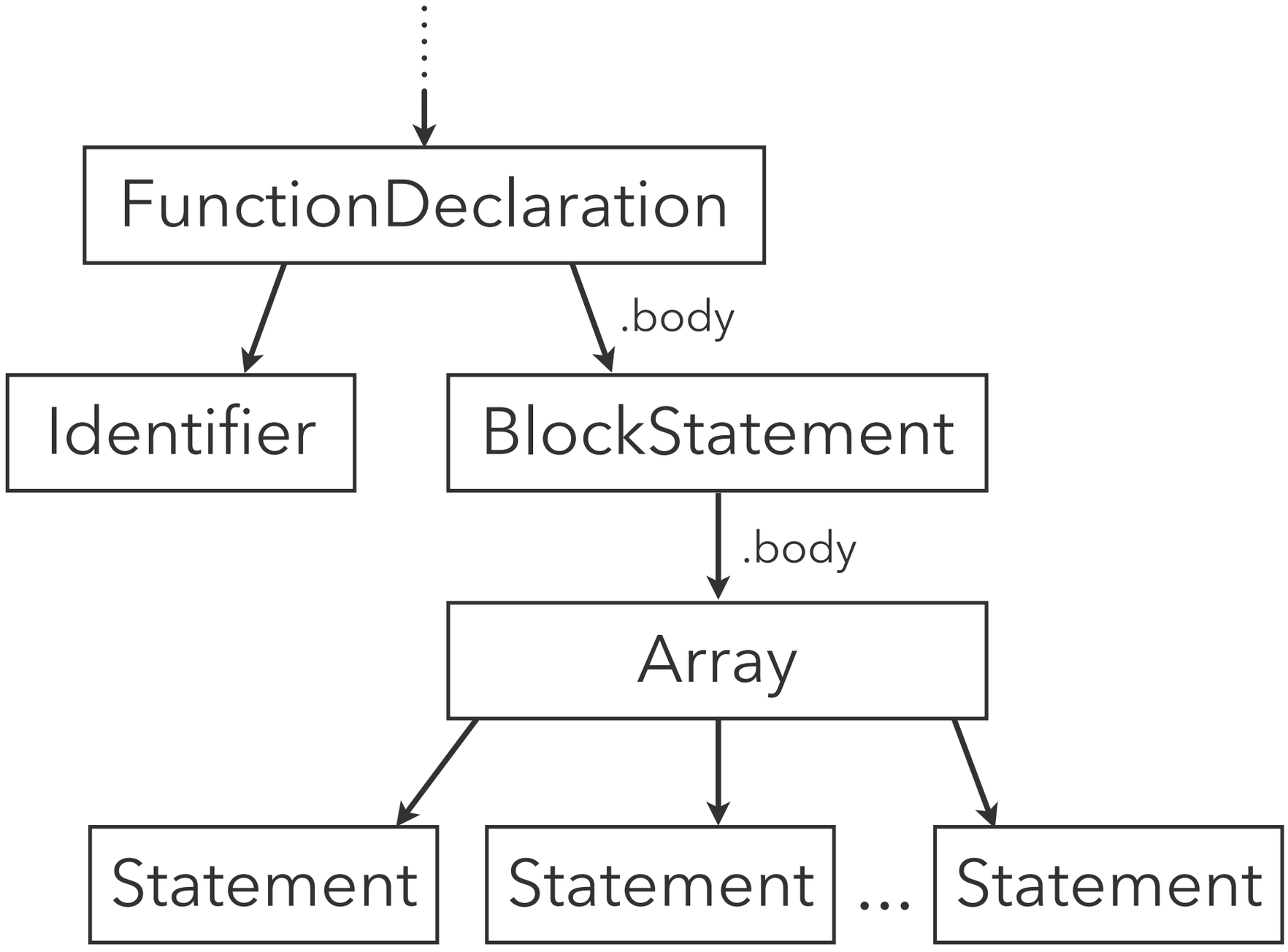
```
function addLogging(code) {
  var ast = esprima.parse(code);
  estraverse.traverse(ast, {
    enter: function(node, parent) {
      if (node.type === 'FunctionDeclaration'
          || node.type === 'FunctionExpression') {
        addBeforeCode(node);
      }
    }
  });
  return escodegen.generate(ast);
}

function addBeforeCode(node) {
  var name = node.id ? node.id.name : '<anonymous function>';
  var beforeCode = "console.log('Entering " + name + "()');";
  var beforeNodes = esprima.parse(beforeCode).body;
  node.body.body = beforeNodes.concat(node.body.body);
}
```

```
function addLogging(code) {
  var ast = esprima.parse(code);
  estraverse.traverse(ast, {
    enter: function(node, parent) {
      if (node.type === 'FunctionDeclaration'
          || node.type === 'FunctionExpression') {
        addBeforeCode(node);
      }
    }
  });
  return escodegen.generate(ast);
}

function addBeforeCode(node) {
  var name = node.id ? node.id.name : '<anonymous function>';
  var beforeCode = "console.log('Entering " + name + "()');";
  var beforeNodes = esprima.parse(beforeCode).body;
  node.body.body = beforeNodes.concat(node.body.body);
}
```

```
function addLogging(code) {
  var ast = esprima.parse(code);
  estraverse.traverse(ast, {
    enter: function(node, parent) {
      if (node.type === 'FunctionDeclaration'
          || node.type === 'FunctionExpression') {
        addBeforeCode(node);
      }
    }
  });
  return escodegen.generate(ast);
}

function addBeforeCode(node) {
  var name = node.id ? node.id.name : '<anonymous function>';
  var beforeCode = "console.log('Entering " + name + "()');";
  var beforeNodes = esprima.parse(beforeCode).body;
  node.body.body = beforeNodes.concat(node.body.body);
}
```


FunctionDeclaration
Identifier
.body
BlockStatement
.body
Array
Statement
...
Statement


FunctionDeclaration
.body
Identifier
BlockStatement
.body
Array
Statement
Statement
...
Statement


FunctionDeclaration
.body
Identifier
BlockStatement
.body
Array
Statement
Statement
...
Statement

```
function addLogging(code) {
  var ast = esprima.parse(code);
  estraverse.traverse(ast, {
    enter: function(node, parent) {
      if (node.type === 'FunctionDeclaration'
          || node.type === 'FunctionExpression') {
        addBeforeCode(node);
      }
    }
  });
  return escodegen.generate(ast);
}

function addBeforeCode(node) {
  var name = node.id ? node.id.name : '<anonymous function>';
  var beforeCode = "console.log('Entering " + name + "()');";
  var beforeNodes = esprima.parse(beforeCode).body;
  node.body.body = beforeNodes.concat(node.body.body);
}
```

```
function addLogging(code) {
  var ast = esprima.parse(code);
  estraverse.traverse(ast, {
    enter: function(node, parent) {
      if (node.type === 'FunctionDeclaration'
          || node.type === 'FunctionExpression') {
        addBeforeCode(node);
      }
    }
  });
  return escodegen.generate(ast);
}

function addBeforeCode(node) {
  var name = node.id ? node.id.name : '<anonymous function>';
  var beforeCode = "console.log('Entering " + name + "()');";
  var beforeNodes = esprima.parse(beforeCode).body;
  node.body.body = beforeNodes.concat(node.body.body);
}
```

```
addLogging("                    \
  function foo(a, b) {          \
    var x = 'blah';             \
    var y = (function () {      \
      return 3;                 \
    })();                       \
  }                             \
  foo(1, 'wut', 3);             \
");
```

```
addLogging("                    \
  function foo(a, b) {          \
    var x = 'blah';             \
    var y = (function () {      \
      return 3;                 \
    })();                       \
  }                             \
  foo(1, 'wut', 3);             \
");
```

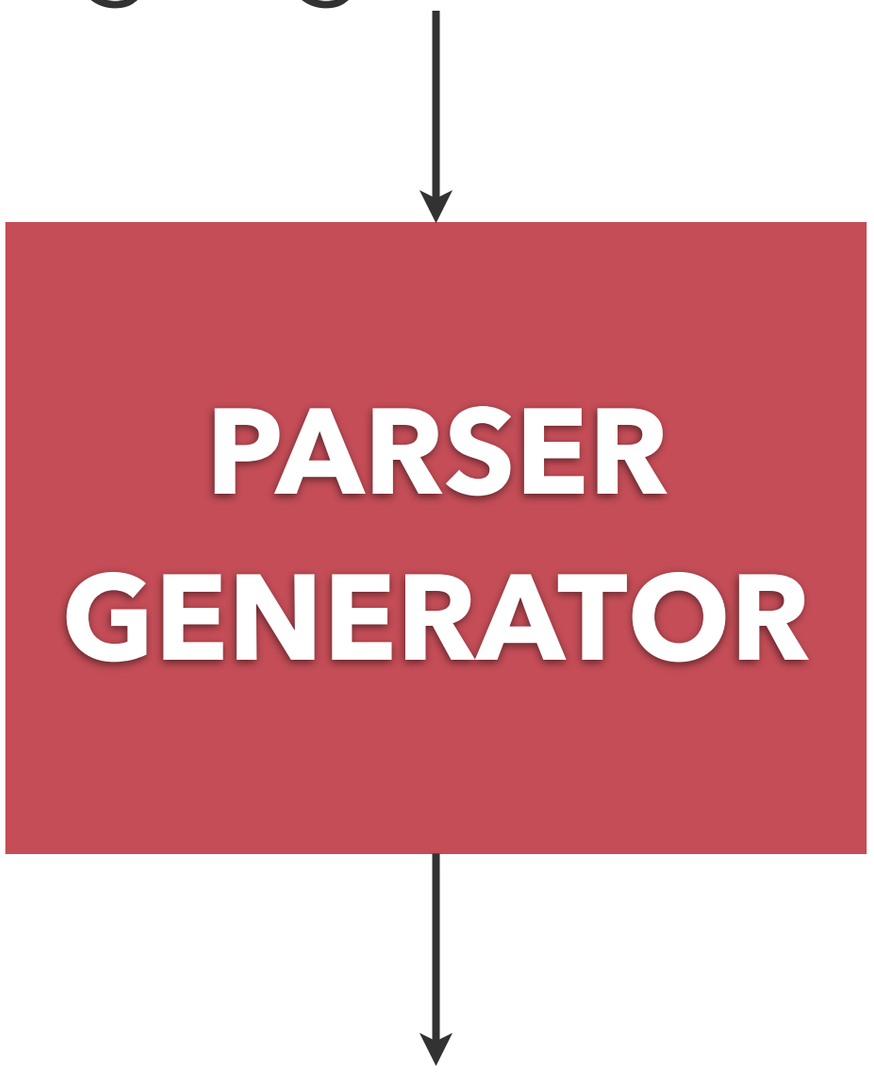
```
function foo(a, b) {
    console.log('Entering foo()');
    var x = 'blah';
    var y = function () {
            console.log('Entering <anonymous function>()');
            return 3;
        }();
}
foo(1, 'wut', 3);
```

```
addLogging("                    \
  function foo(a, b) {          \
    var x = 'blah';             \
    var y = (function () {      \
      return 3;                 \
    })();                       \
  }                             \
  foo(1, 'wut', 3);             \
");
```

```
function foo(a, b) {
    console.log('Entering foo()');
    var x = 'blah';
    var y = function () {
            console.log('Entering <anonymous function>()');
            return 3;
        }();
}
foo(1, 'wut', 3);
```

Parser Generators

Language Grammar



Parser

PEG.js

Parser Generator for JavaScript

PEG.js is a simple parser generator for JavaScript that produces fast parsers with excellent error reporting. You can use it to process complex data or computer languages and build transformers, interpreters, compilers and other tools easily.

Try PEG.js online

— or —

```
npm install pegjs
```

— or —

Download browser version

- **PEG.js — minified**
- **PEG.js — development**

FOLLOW ME ON twitter

Features

- Simple and expressive grammar syntax
- Integrates both lexical and syntactical analysis
- Parsers have excellent error reporting out of the box
- Based on parsing expression grammar formalism — more powerful than traditional LL(*k*) and LR(*k*) parsers
- Usable from your browser, from the command line, or via JavaScript API

Formal Grammars

Context-Free Grammar

expr → expr [-+] term | term

term → term [*/] factor | factor

factor → '(' expr ')' | number

number → [0-9]+

Backus-Naur Form

expr ::= expr [-+] term | term

term ::= term [*/] factor | factor

factor ::= '(' expr ')' | number

number ::= [0-9]+

EBNF

expr = expr [-+] term | term

term = term [*/] factor | factor

factor = '(' expr ')' | number

number = [0-9]+

CFG: Unordered choice

```
expr = expr [-+] term | term

term = term [*/] factor | factor

factor = '(' expr ')' | number

number = [0-9]+
```

PEG: Ordered Choice

```
expr = expr [-+] term / term

term = term [*/] factor / factor

factor = '(' expr ')' / number

number = [0-9]+
```

```
var PEG = require('pegjs');

var parser = PEG.buildParser("       \
  expr = expr [-+] term / term       \
  term = term [*/] factor / factor   \
  factor = '(' expr ')' / number     \
  number = [0-9]+                    \
");
parser.parse('1+10');
```

```
var PEG = require('pegjs');

var parser = PEG.buildParser("       \
  expr = expr [-+] term / term       \
  term = term [*/] factor / factor   \
  factor = '(' expr ')' / number     \
  number = [0-9]+                    \
");
parser.parse('1+10');
```

```
~/node_modules/pegjs/lib/peg.js:3316
            throw new PEG.GrammarError(
                  ^
PEG.GrammarError: Left recursion detected for rule "expr".
```

Left Recursion

```
expr = expr [-+] term / term

term = term [*/] factor / factor

factor = '(' expr ')' / number

number = [0-9]+
```

Left Recursion

```
expr = term ([-+] term)*

term = factor ([*/] factor)*

factor = '(' expr ')' / number

number = [0-9]+
```

```
var PEG = require('pegjs');

var parser = PEG.buildParser("       \
  expr = term ([-+] term)*           \
  term = factor ([*/] factor)*       \
  factor = '(' expr ')' / number     \
  number = [0-9]+                    \
");
parser.parse('1+10');
```

```
var PEG = require('pegjs');

var parser = PEG.buildParser("        \
  expr = term ([-+] term)*            \
  term = factor ([*/] factor)*        \
  factor = '(' expr ')' / number      \
  number = [0-9]+                     \
");
parser.parse('1+10');
```

```
[[["1"],[]],["+",[["1","0"],[]]]]
```

Semantic Actions

```
expr
  = term ([-+] term)*

term
  = factor ([*/] factor)*

factor
  = '(' expr ')' / number

number
  = [0-9]+
```

Semantic Actions

```
expr
  = term ([-+] term)*

term
  = factor ([*/] factor)*

factor
  = '(' expr ')' / number

number
  = digits:[0-9]+
```

Semantic Actions

```
expr
  = term ([-+] term)*

term
  = factor ([*/] factor)*

factor
  = '(' expr ')' / number

number
  = digits:[0-9]+ { return digits.join(''); }
```

Semantic Actions

```
expr
  = term ([-+] term)*

term
  = factor ([*/] factor)*

factor
  = '(' expr ')' / number
```

```
[["1",[]],[["+",["10",[]]]]]
```

Semantic Actions

```
expr
  = term ([-+] term)*

term
  = factor ([*/] factor)*

factor
  = '(' expr ')' / number
```

```
[["1",[]],["+",["10",[]]]]
```

```
expr
  = term ([-+] term)*

term
  = factor ([*/] factor)*

factor
  = '(' expr ')' / number

number
  = digits:[0-9]+ { return digits.join(''); }
```

```
{
  function Number(digits) {
    this.nodeType = 'Number';
    this.value = digits.join('');
  }
  ...
}

expr
  = term ([-+] term)*

term
  = factor ([*/] factor)*

factor
  = '(' expr ')' / number

number
  = digits:[0-9]+ { return digits.join(''); }
```

```
{
  function Number(digits) {
    this.nodeType = 'Number';
    this.value = digits.join('');
  }
  ...
}

expr
  = term ([-+] term)*

term
  = factor ([*/] factor)*

factor
  = '(' expr ')' / number

number
  = digits:[0-9]+ { return digits.join(''); }
```

```
{
  function Number(digits) {
    this.nodeType = 'Number';
    this.value = digits.join('');
  }
  ...
}

expr
  = term ([-+] term)*

term
  = factor ([*/] factor)*

factor
  = '(' expr ')' / number

number
  = digits:[0-9]+ { return new Number(digits); }
```

An AltJS Language in 5 minutes

```
expr
  = term ([-+] term)*

term
  = factor ([*/] factor)*

factor
  = '(' expr ')' / number

number
  = digits:[0-9]+ { return digits.join(''); }
```

```
expr
  = term ([-+] term)*
```

```
expr
  = term ([-+] term)*
  / decl
```

```
expr
  = term ([-+] term)*
  / decl

decl = ident ' := ' expr
```

```
expr
  = term ([-+] term)*
  / decl

decl = ident ' := ' expr

ident = (digit / letter / '_')+
```

```
expr
  = term ([-+] term)*
  / decl

decl = ident ' := ' expr

ident = (digit / letter / '_')+

digit = [0-9]
```

```
expr
  = term ([-+] term)*
  / decl

decl = ident ' := ' expr

ident = (digit / letter / '_')+

digit = [0-9]

letter = [a-zA-Z]
```

```
program = expr? ('.' [ \\n]* expr)*

expr
  = term ([-+] term)*
  / decl

decl = ident ' := ' expr

ident = (digit / letter / '_')+

digit = [0-9]

letter = [a-zA-Z]
```

```
program = expr? ('.' [ \\n]* expr)*

expr
  = term ([-+] term)*
  / decl

decl = ident ' := ' expr

ident = (digit / letter / '_')+

digit = [0-9]

letter = [a-zA-Z]
```

```
> parser.parse('x := 2+5. y := 3')
```

```
program = expr? ('.' [ \\n]* expr)*

expr
  = term ([-+] term)*
  / decl

decl = ident ' := ' expr

ident = (digit / letter / '_')+

digit = [0-9]

letter = [a-zA-Z]
```

```
> parser.parse('x := 2+5. y := 3')
[[["x"]," := ",[["2",[]],[["+",["5",[]]]]],[[".",[],
[["y"]," := ",[["3",[]],[]]]]]
```

```
program = expr? ('.' [ \\n]* expr)*

expr
  = term ([-+] term)*
  / decl

decl = ident ' := ' expr

ident = (digit / letter / '_')+

digit = [0-9]

letter = [a-zA-Z]
```

```
program = expr? ('.' [ \\n]* expr)*

expr
  = term ([-+] term)*
  / decl

decl = id:ident ' := ' e:expr
       { return 'var ' + id + ' = ' + e + ';'; }

ident = (digit / letter / '_')+

digit = [0-9]

letter = [a-zA-Z]
```

```
program = expr? ('.' [ \\n]* expr)*

expr
  = term ([-+] term)*
  / decl

decl = id:ident ' := ' e:expr
       { return 'var ' + id + ' = ' + e + ';'; }

ident = (digit / letter / '_')+

digit = [0-9]

letter = [a-zA-Z]
```

```
program = expr? ('.' [ \\n]* expr)*

expr
  = term ([-+] term)*
  / decl

decl = id:ident ' := ' e:expr
       { return 'var ' + id + ' = ' + e + ';'; }

ident = (digit / letter / '_')+

digit = [0-9]

letter = [a-zA-Z]
```

```
program = expr? ('.' [ \\n]* expr)*

expr
  = term ([-+] term)*
  / decl

decl = id:ident ' := ' e:expr
       { return 'var ' + id + ' = ' + e + ';'; }

ident = (digit / letter / '_')+

digit = [0-9]

letter = [a-zA-Z]
```

```
> parser.parse('x := 2+5. y := 3')
```

```
program = expr? ('.' [ \\n]* expr)*

expr
  = term ([-+] term)*
  / decl

decl = id:ident ' := ' e:expr
       { return 'var ' + id + ' = ' + e + ';'; }

ident = (digit / letter / '_')+

digit = [0-9]

letter = [a-zA-Z]
```

```
> parser.parse('x := 2+5. y := 3')
["var x = 2,,+,5,;",[[".",[],"var y = 3,,;"]]]
```

```
program = expr? ('.' [ \\n]* expr)*

expr
  = term ([-+] term)*
  / decl

decl = id:ident ' := ' e:expr
       { return 'var ' + id + ' = ' + e + ';'; }

ident = (digit / letter / '_')+

digit = [0-9]

letter = [a-zA-Z]
```

```
program = expr? ('.' [ \\n]* expr)*

expr
  = t:term rest:([-+] term)*
    { return flatten(t.concat(rest)); }
  / decl

decl = id:ident ' := ' e:expr
       { return 'var ' + id + ' = ' + e.join('') + ';'; }

ident = (digit / letter / '_')+

digit = [0-9]

letter = [a-zA-Z]
```

```
program = expr? ('.' [ \\n]* expr)*

expr
  = t:term rest:([-+] term)*
    { return flatten(t.concat(rest)); }
  / decl

decl = id:ident ' := ' e:expr
       { return 'var ' + id + ' = ' + e.join('') + ';'; }

ident = (digit / letter / '_')+

digit = [0-9]

letter = [a-zA-Z]
```

```
> parser.parse('x := 2+5. y := 3')
```

```
program = expr? ('.' [ \\n]* expr)*

expr
  = t:term rest:([-+] term)*
    { return flatten(t.concat(rest)); }
  / decl

decl = id:ident ' := ' e:expr
       { return 'var ' + id + ' = ' + e.join('') + ';'; }

ident = (digit / letter / '_')+

digit = [0-9]

letter = [a-zA-Z]
```

```
> parser.parse('x := 2+5. y := 3')
["var x = 2+5;",[[".",[],"var y = 3;"]]]
```

```
program = expr? ('.' [ \\n]* expr)*

expr
  = t:term rest:([-+] term)*
    { return flatten(t.concat(rest)); }
  / decl

decl = id:ident ' := ' e:expr
       { return 'var ' + id + ' = ' + e.join('') + ';'; }

ident = (digit / letter / '_')+

digit = [0-9]

letter = [a-zA-Z]
```

```
program = e:expr? rest:(('.' [\\n ]* e:expr){ return e; })*
          { return [e].concat(rest).join('\n'); }

expr
  = t:term rest:([-+] term)*
    { return flatten(t.concat(rest)); }
  / decl

decl = id:ident ' := ' e:expr
       { return 'var ' + id + ' = ' + e.join('') + ';'; }

ident = (digit / letter / '_')+

digit = [0-9]

letter = [a-zA-Z]
```

```
program = e:expr? rest:(('.' [\\n ]* e:expr){ return e; })*
          { return [e].concat(rest).join('\n'); }

expr
  = t:term rest:([-+] term)*
    { return flatten(t.concat(rest)); }
  / decl

decl = id:ident ' := ' e:expr
       { return 'var ' + id + ' = ' + e.join('') + ';'; }

 ident = (digit / letter / '_')+

 digit = [0-9]
```

```
> parser.parse('x := 2+5. y := 3')
```

```
program = e:expr? rest:(('.' [\\n ]* e:expr){ return e; })*
          { return [e].concat(rest).join('\n'); }

expr
  = t:term rest:([-+] term)*
    { return flatten(t.concat(rest)); }
  / decl

decl = id:ident ' := ' e:expr
       { return 'var ' + id + ' = ' + e.join('') + ';'; }

ident = (digit / letter / '_')+

digit = [0-9]
```

```
> parser.parse('x := 2+5. y := 3')
var x = 2+5;
var y = 3;
```