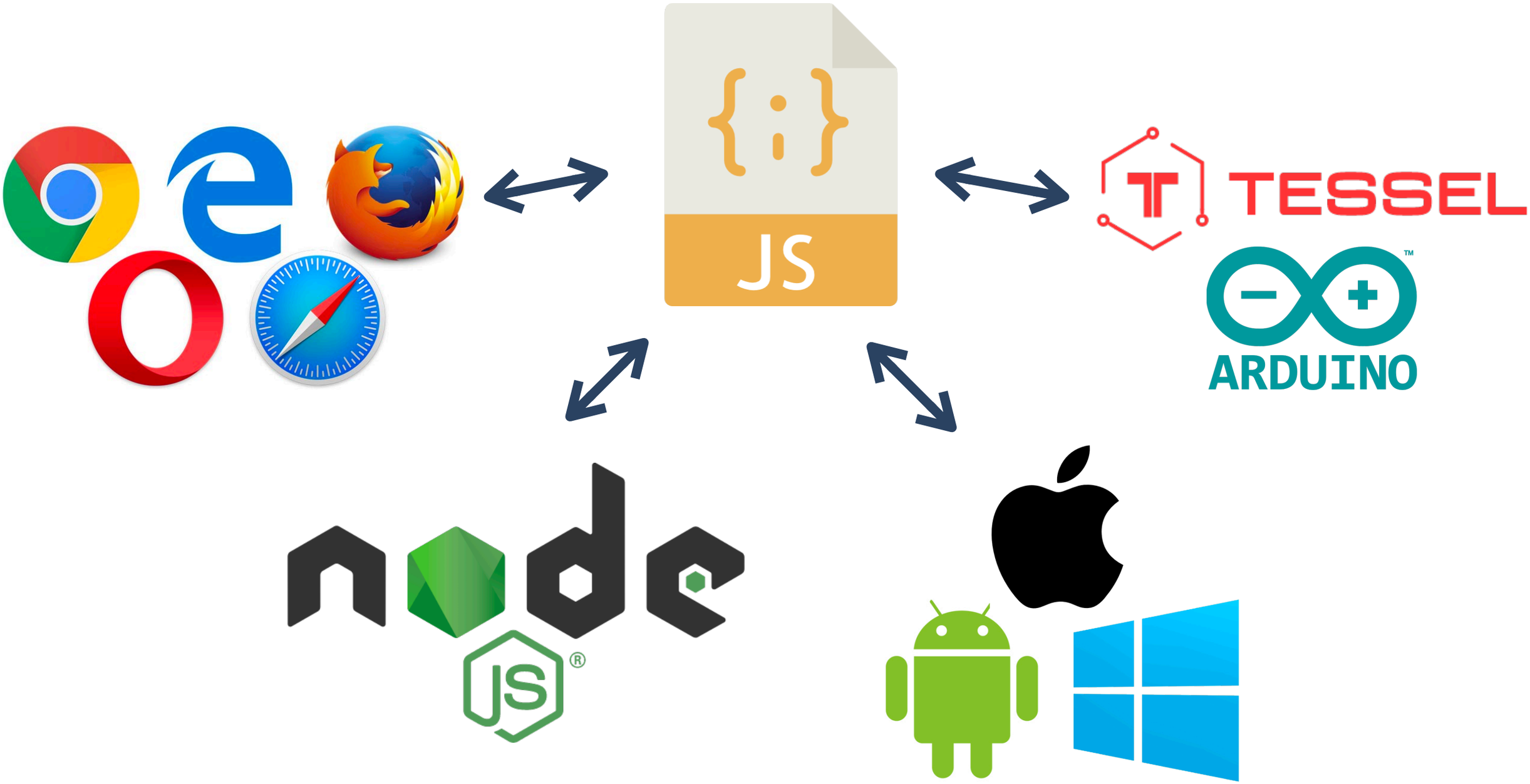
PLRG
Programming Language
Research Group

Update-Tolerant JavaScript Static Analysis for Frequently Released ECMAScript

Jihyeok Park

PLRG @ KAIST
November 26, 2019

JavaScript - Portability



JavaScript - Popularity

> **JavaScript is used as client-side programming language by 95.0%** of all the websites.

https://w3techs.com/technologies/details/cp-javascript

# Ranking	Programming Language	Percentage (Change)
1	JavaScript	20.308% (-0.919%)
2	Python	17.068% (-0.495%)
3	Java	10.366% (+0.362%)
4	Go	8.809% (+0.800%)
5	C++	7.051% (-0.218%)

https://madnight.github.io/githut/#/pull_requests/2019/3

JavaScript - Complex Semantics

```
function f(x) { return x == !x; }
```

Always return `false`?

NO!!

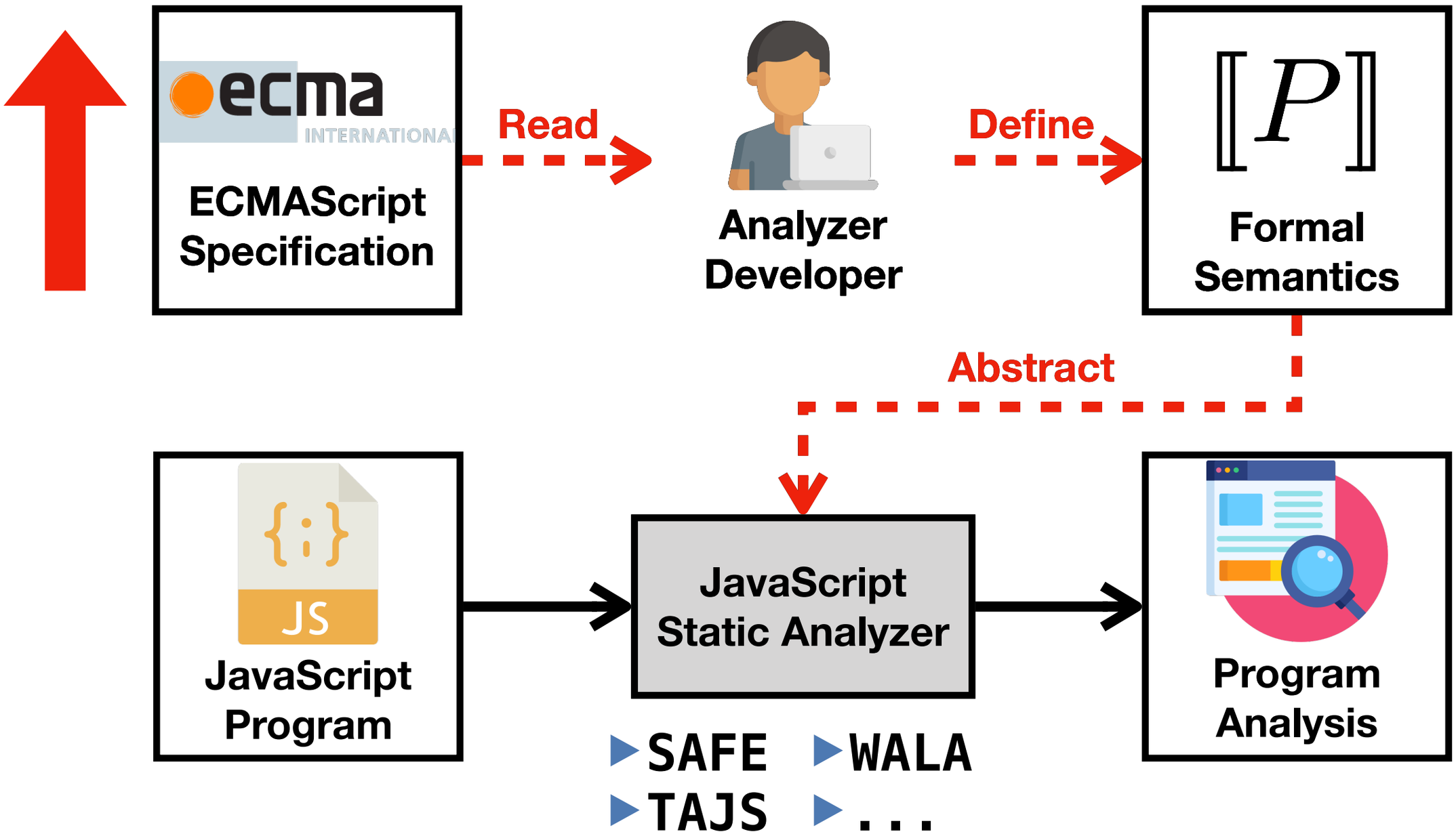
```
f([]) -> [] == ![]
      -> [] == false
      -> +[] == +false
      -> 0 == 0
      -> true
```

ECMAScript

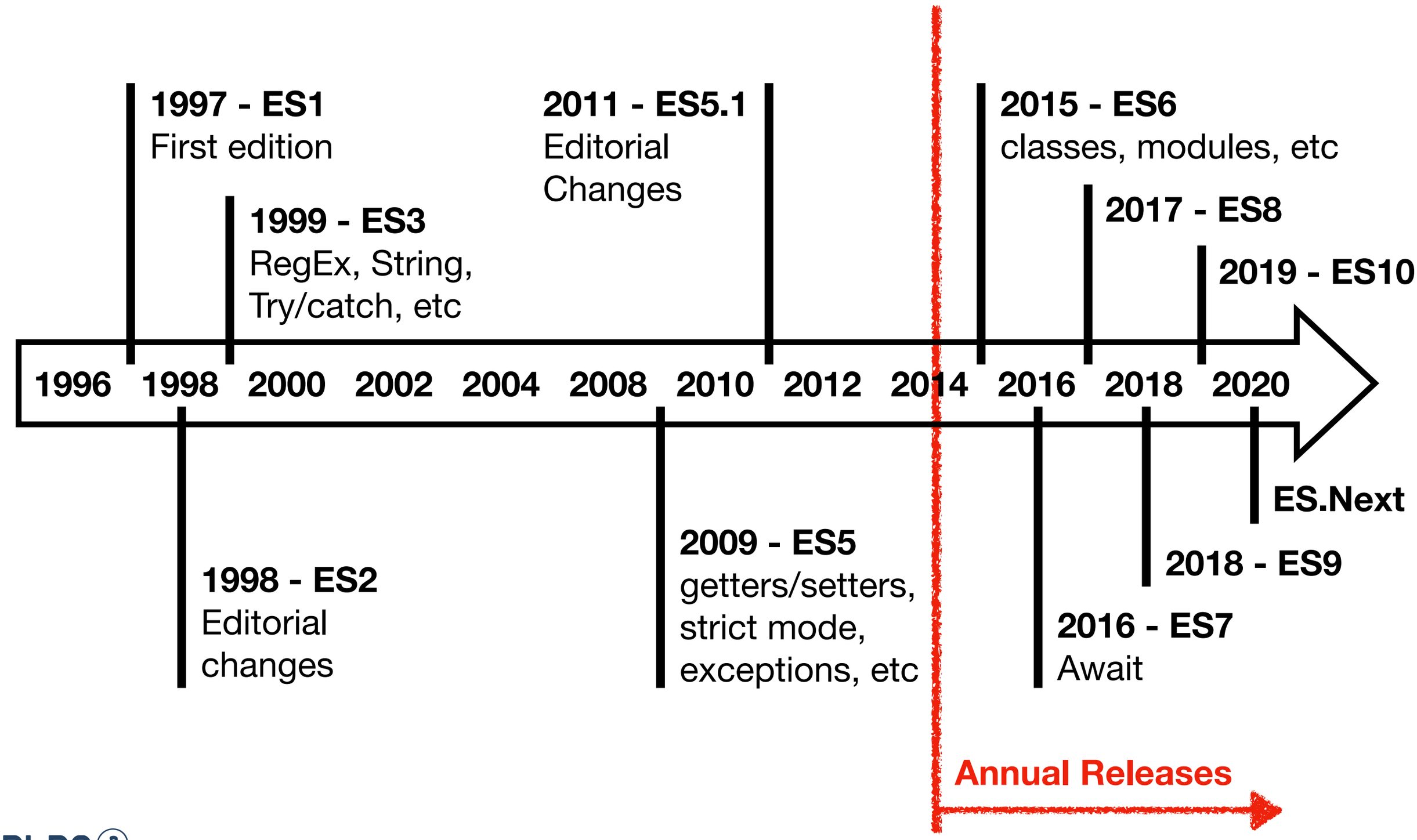
The standard for JavaScript is ECMAScript. As of 2012, all modern browsers fully support ECMAScript 5.1. Older browsers support at least ECMAScript 3. On June 17, 2015, ECMA International published the sixth major version of ECMAScript, which is officially called ECMAScript 2015, and was initially referred to as ECMAScript 6 or ES6.

https://developer.mozilla.org/en-US/docs/Web/JavaScript

Static Analysis for JavaScript



ECMAScript Release Timeline

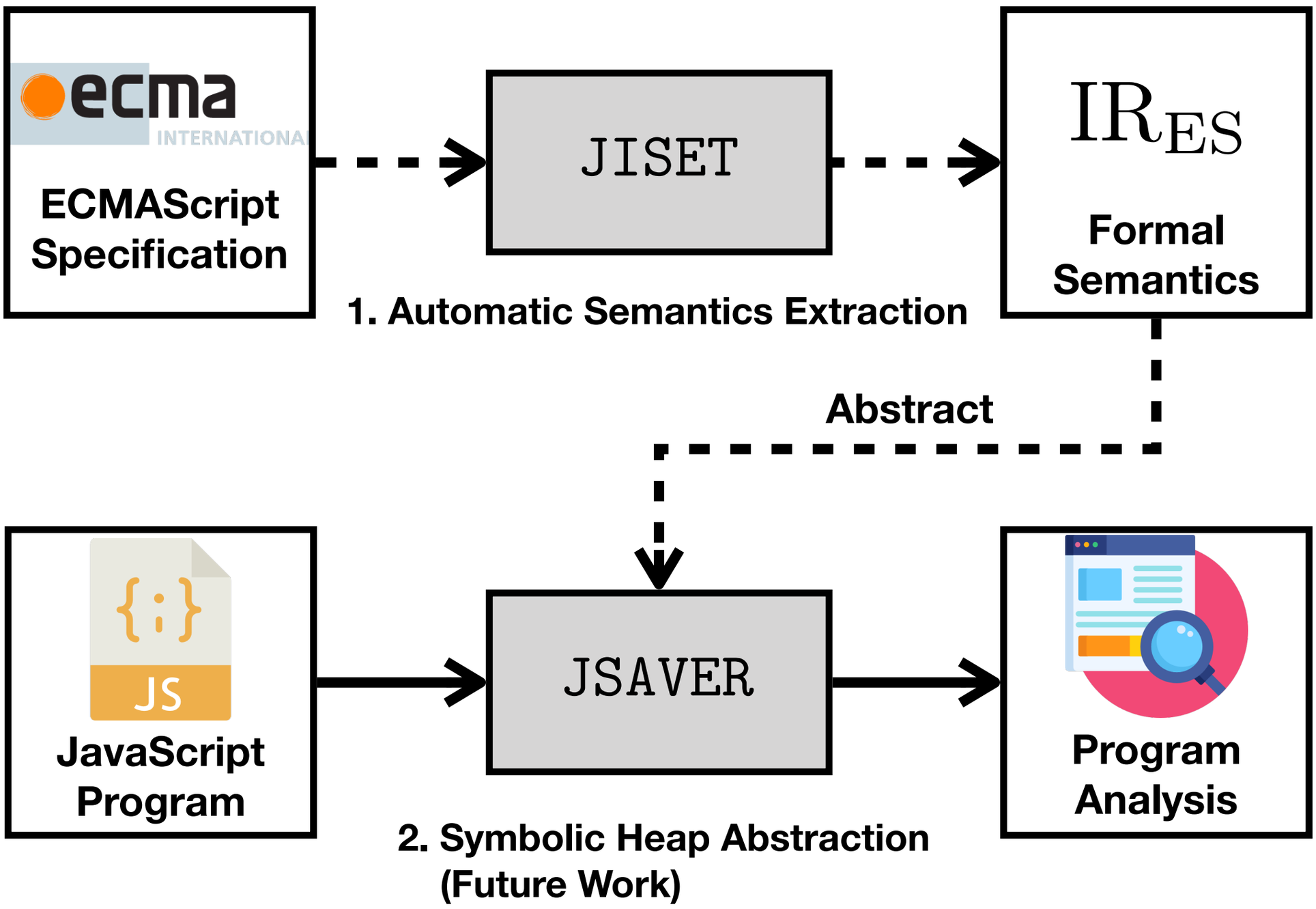


Future Work

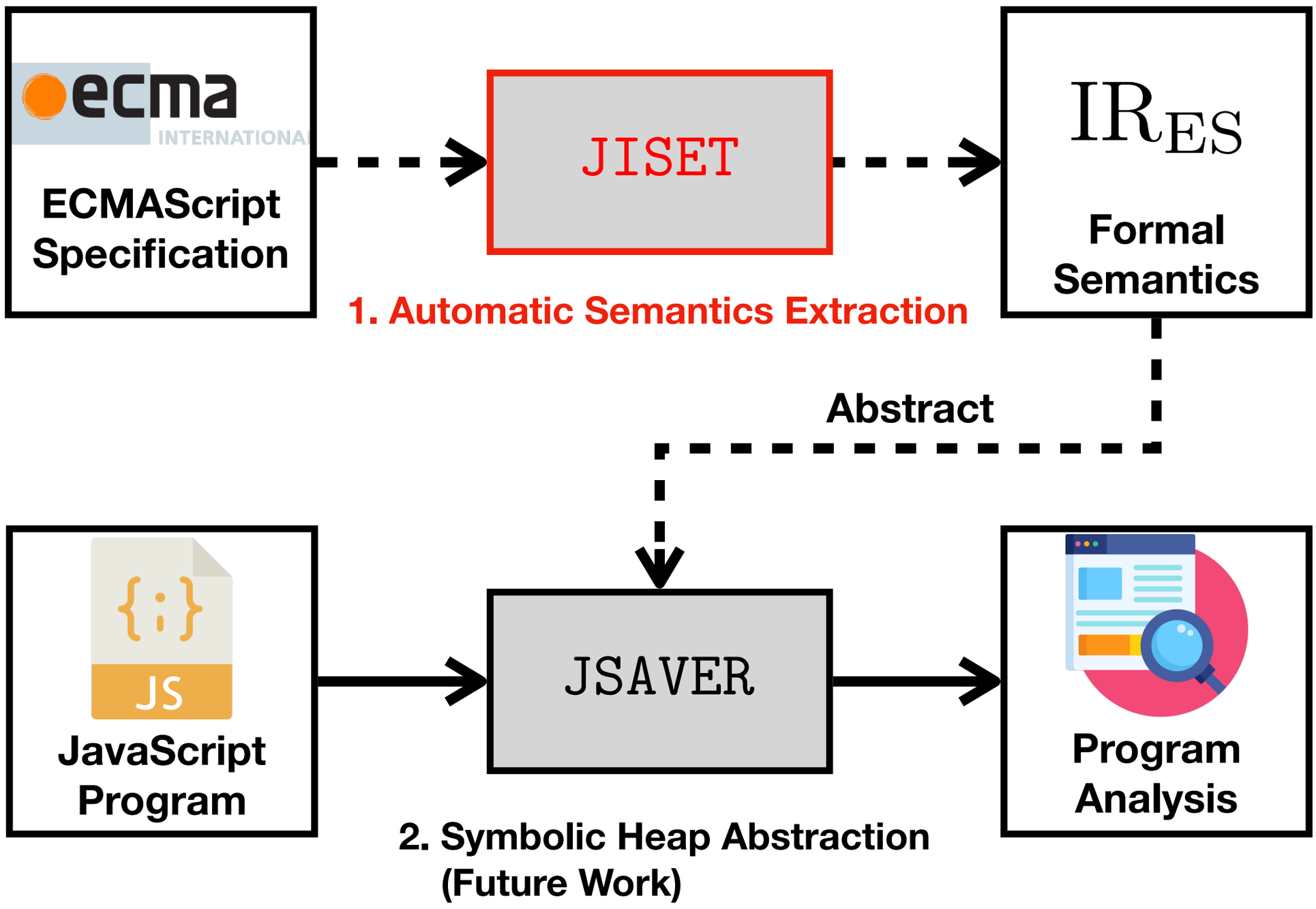
Symbolic Heap Abstraction for Scalability

Update-Tolerant JavaScript Static Analysis for Frequently Released ECMAScript

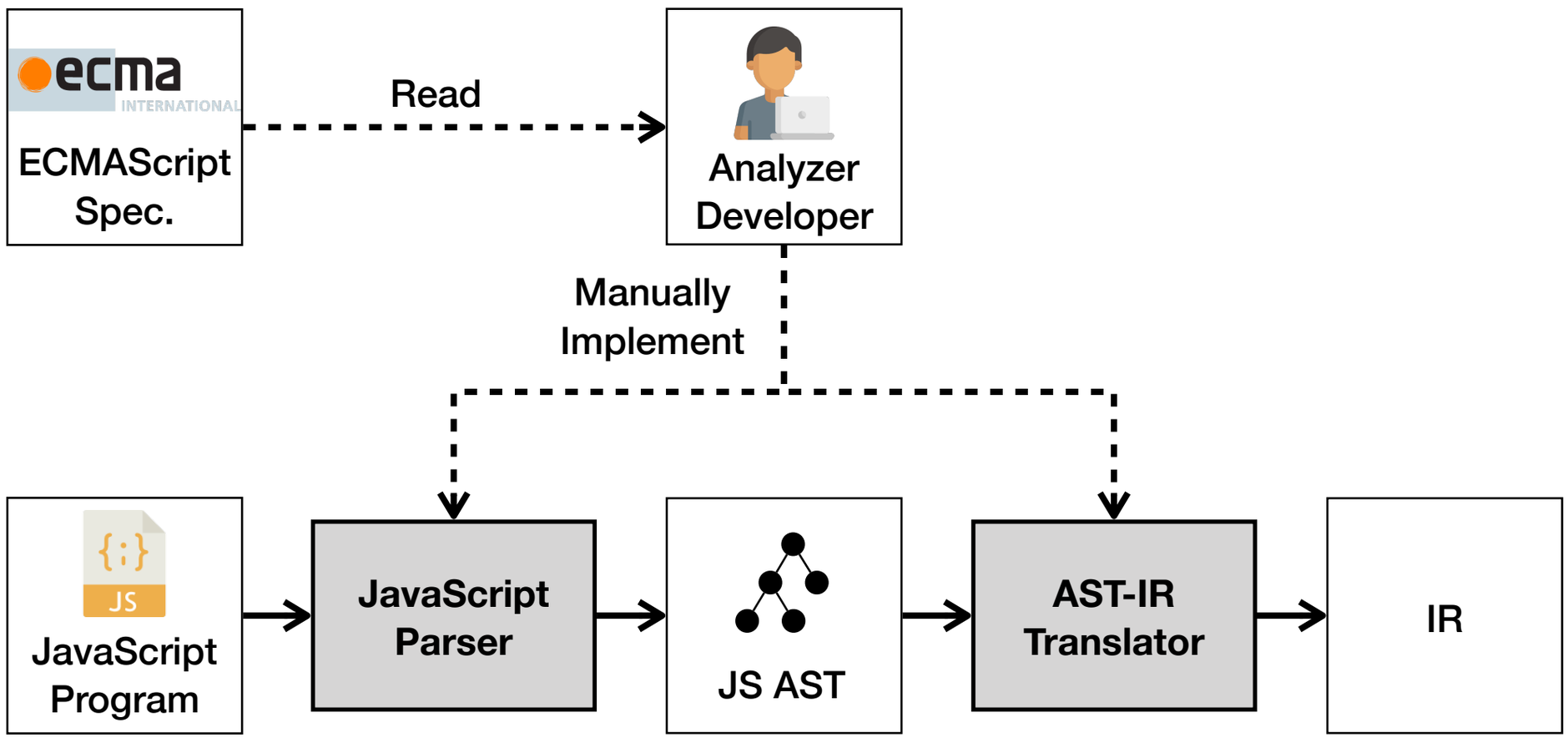
Automatic Semantics Extraction via IR_{ES} (Intermediate Rep. for ES)


ECMAScript Specification
JISET
IR_{ES}
Formal Semantics
1. Automatic Semantics Extraction
Abstract
JavaScript Program
JSAVER
Program Analysis
2. Symbolic Heap Abstraction (Future Work)

JISET: JavaScript IR-based Semantics Extraction Toolchain
(Submitted to ICSE'20)



IR-based Semantics Extraction



IR-based Semantics Extraction

ArrayLiteral[Yield, Await] :

[*Elision*opt]

[*ElementList*[?Yield, ?Await]]

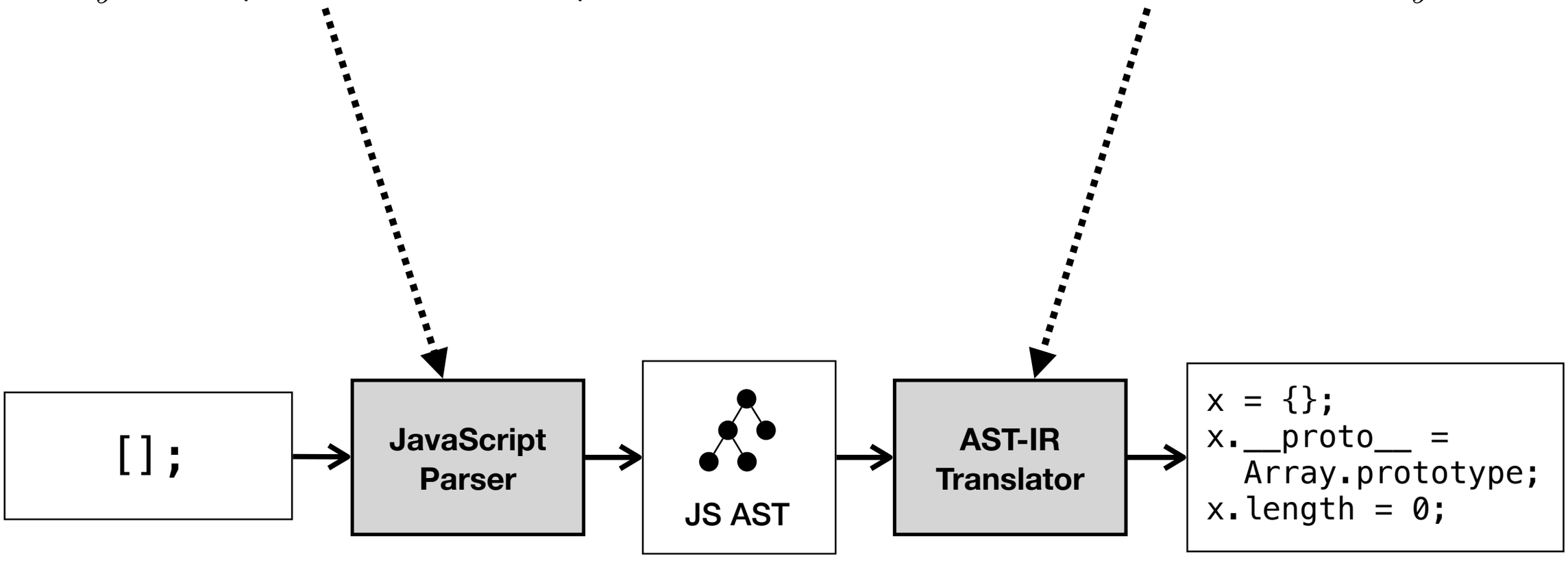
[*ElementList*[?Yield, ?Await] , *Elision*opt]

The *ArrayLiteral* production in ECMAScript 2020

ArrayLiteral : [*Elision*]

1. Let *array* be ! ArrayCreate(0).
2. If *Elision* is present, then
   a. Let *len* be the result of performing ArrayAccumulation for *Elision* with arguments *array* and 0.
   b. ReturnIfAbrupt(*len*).
3. Return *array*.

The semantics of the first alternative for *ArrayLiteral*



Core Idea

ArrayLiteral`[Yield, Await]` :

[*Elision*`opt`]

[*ElementList*`[?Yield, ?Await]`]

[*ElementList*`[?Yield, ?Await]` , *Elision*`opt`]

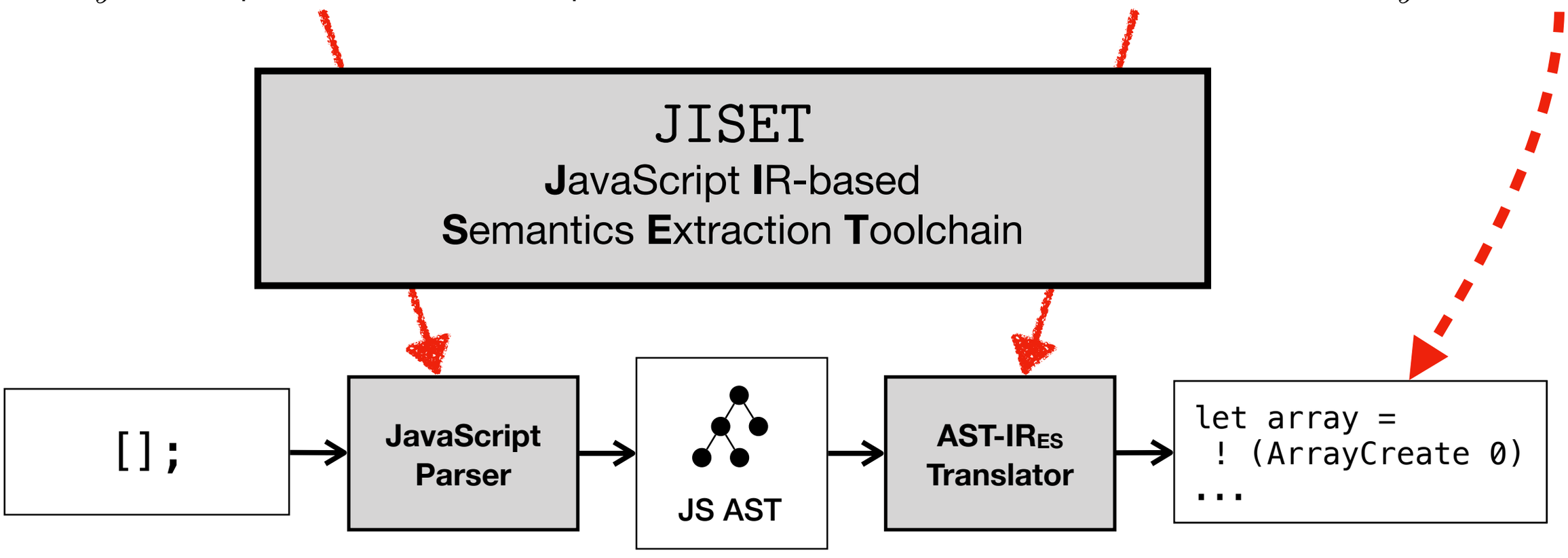
The *ArrayLiteral* production in ECMAScript 2020

ArrayLiteral : [*Elision*]

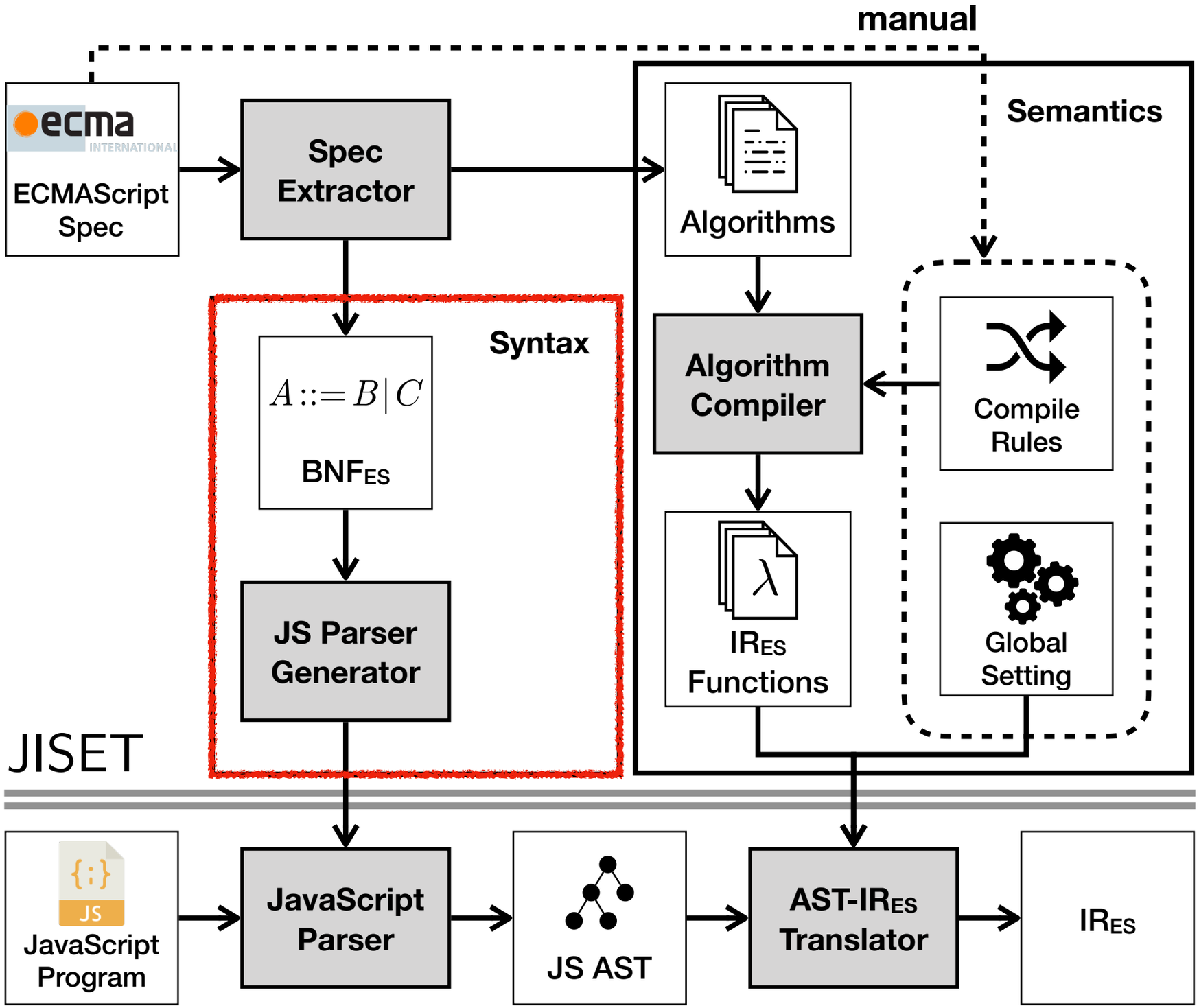
1. Let *array* be ! ArrayCreate(0).
2. If *Elision* is present, then
   a. Let *len* be the result of performing ArrayAccumulation for *Elision* with arguments *array* and 0.
   b. ReturnIfAbrupt(*len*).
3. Return *array*.

The semantics of the first alternative for *ArrayLiteral*

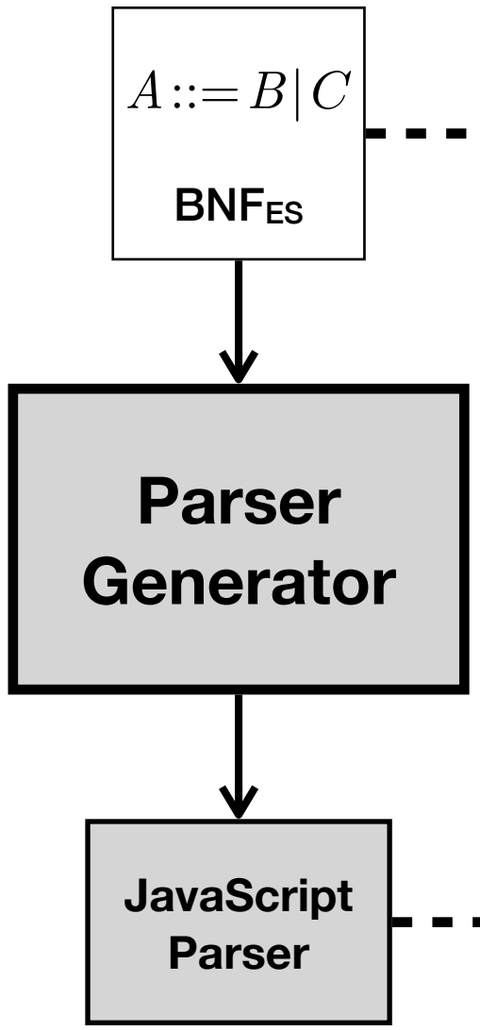
IR$_{ES}$ (Intermediate Rep. for ES)



Overall Structure of JISET



JS Parser Synthesis



$ArrayLiteral_{[Yield,\ Await]}$:

[$Elision_{opt}$]

[$ElementList_{[?Yield,\ ?Await]}$]

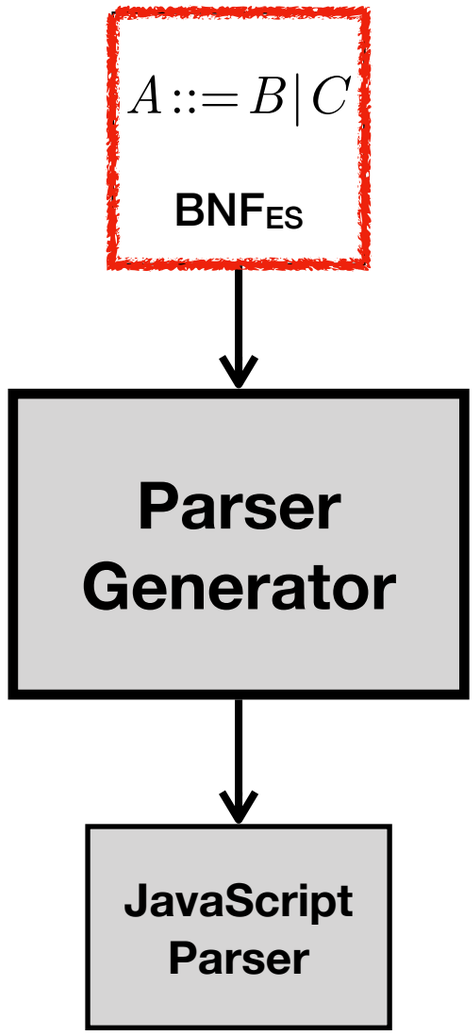
[$ElementList_{[?Yield,\ ?Await]}$, $Elision_{opt}$]

The $ArrayLiteral$ production in ECMAScript 2020

```
type P[T] = List[Boolean] => LAParser[T]
lazy val ArrayLiteral: P[ArrayLiteral] = memo {
  case List(Yield, Await) =>
  "[" ~ opt(Elision) ~ "]"
      ^^ ArrayLiteral0 |
  "[" ~ ElementList(Yield, Await) ~ "]"
      ^^ ArrayLiteral1 |
  "[" ~ ElementList(Yield, Await)
      ~ "," ~ opt(Elision) ~ "]"
      ^^ ArrayLiteral2
}
```

The generated parser for $ArrayLiteral$

BNF_{ES} - BNF for ECMAScript



Productions

$$A(p_1, \cdots, p_k) ::= (c_1 \Rightarrow)^? \alpha_1 \mid \cdots \mid (c_n \Rightarrow)^? \alpha_n$$

where p_i is a boolean parameter.

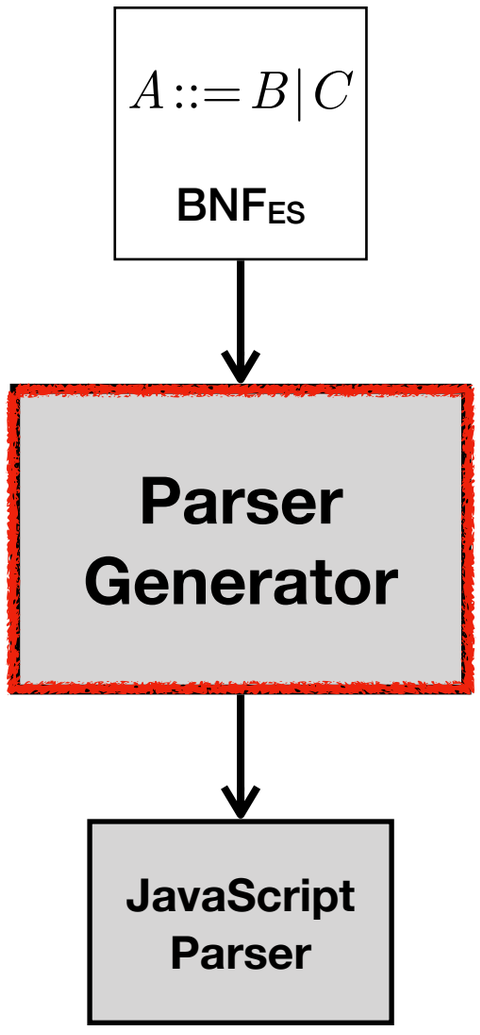
Conditions

$$c ::= p_i \mid !p_i$$

Symbols

Symbol s	Description
ϵ	empty sequnce
`a`	terminal
$A(a_1, \cdots, a_k)$	non-terminal
$s^?$	optional symbol
$+s$	positive lookahead
$-s$	negative lookahead
$s \setminus s'$	exclusion
$\langle \neg \mathtt{LT} \rangle$	no line-terminator

Parser Generator



Parsing Expression Grammar (PEG)

+ Human-Readable Parsers
+ Easy to Support BNFES Features
+ Linear Parsing Time

- ~~Different with BNF (∵ Ordered Choices)~~

+

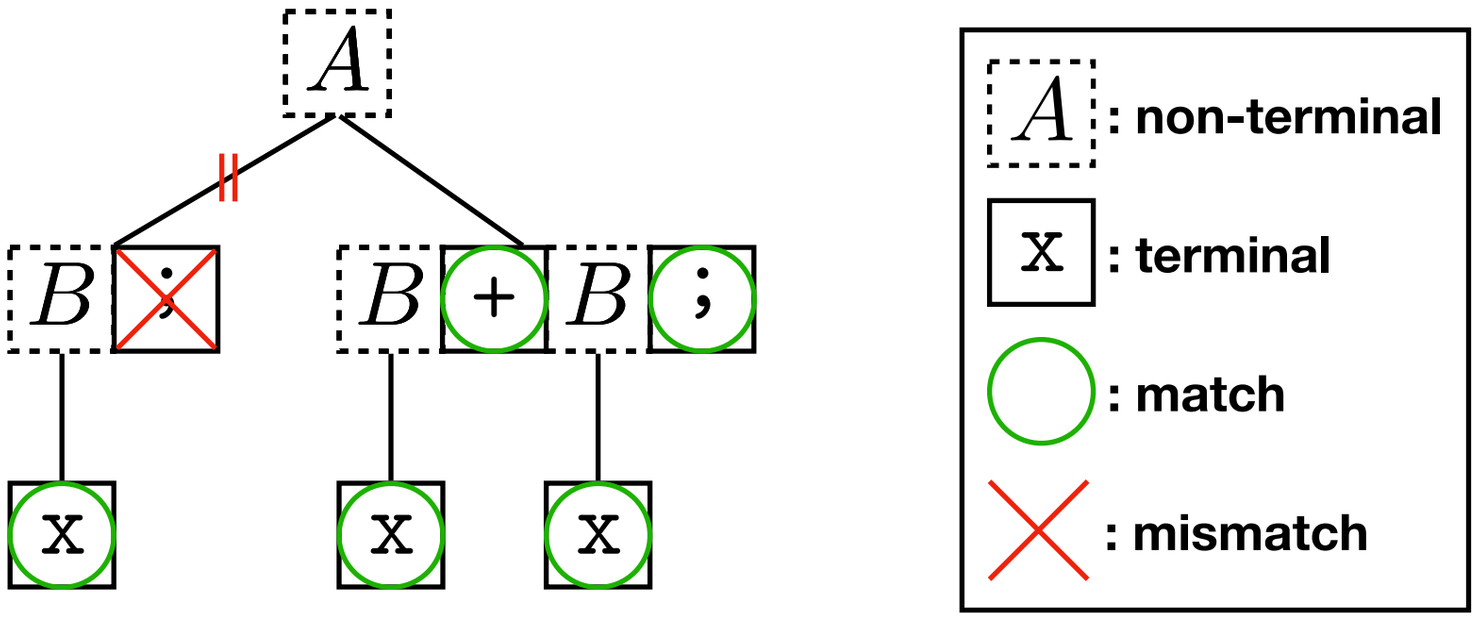
Lookahead Parsing

(POPL'04) Bryan Ford, "Parsing Expression Grammars: A Recognition-based Syntactic Foundation"

(ICFP'02) Bryan Ford, "Packrat parsing: simple, powerful, lazy, linear time, functional pearl"

Parsing Expression Grammar

- **Recursive Descendent Parser** with **Backtracking**

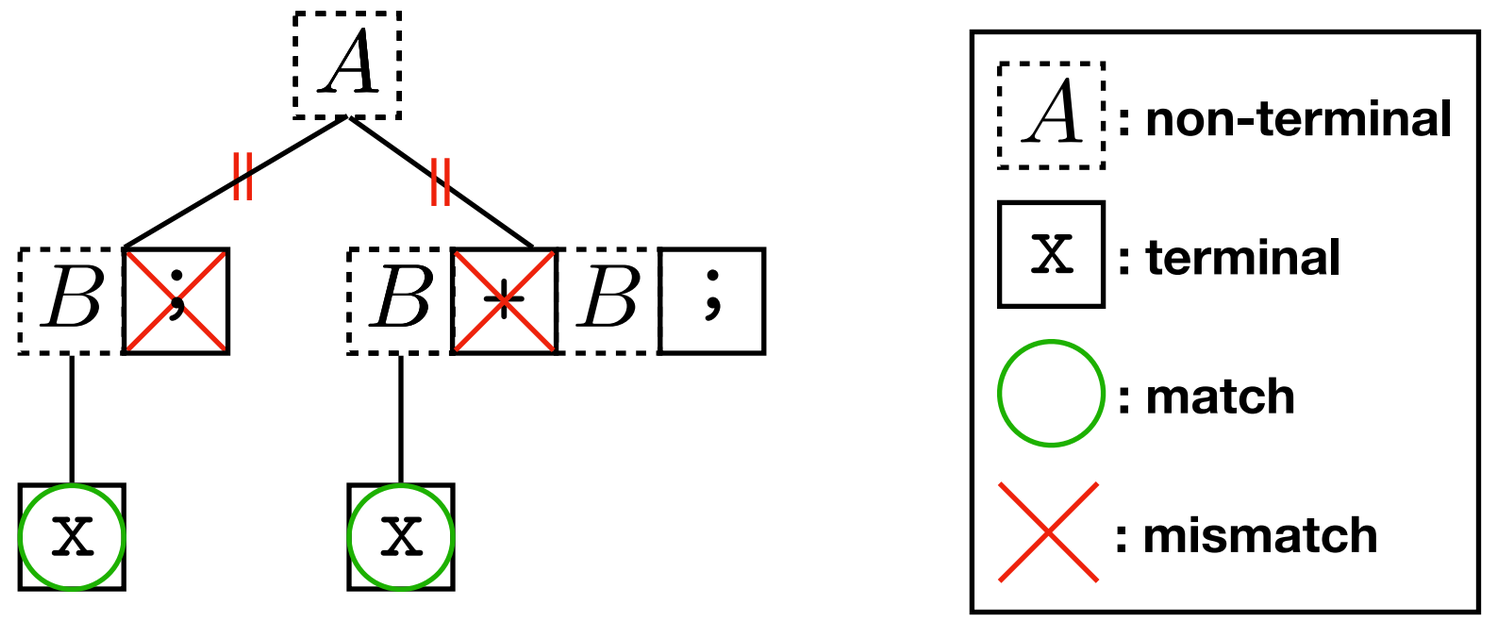


$$A ::= B\,; \;/\; B + B\,;$$
$$B ::= \mathtt{x} \;/\; \mathtt{xy}$$

Input : `x+x;`

Parsing Expression Grammar

- **Ordered Choices**



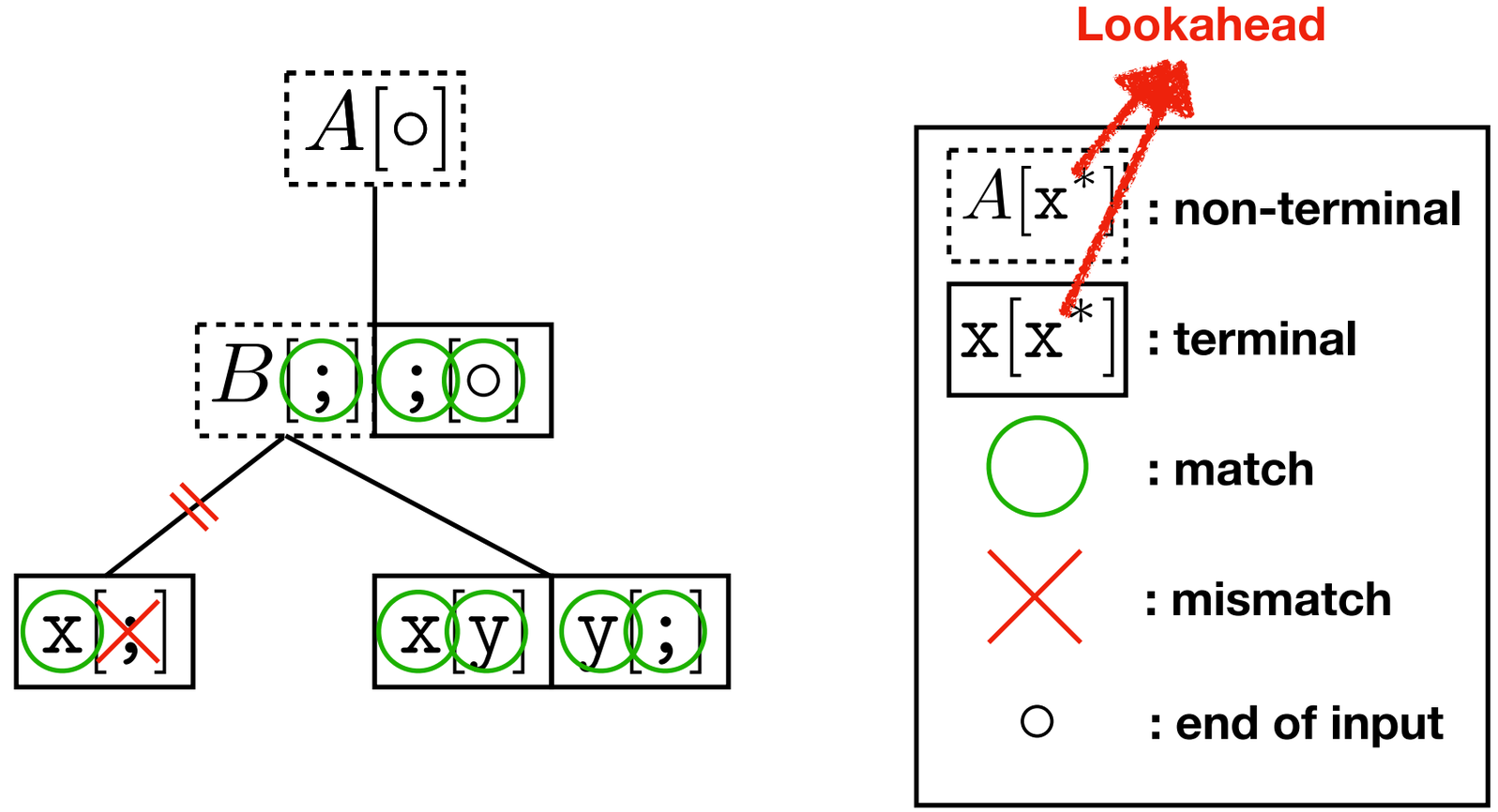
$A ::= B\,; \;/\; B + B\,;$

$B ::= \mathtt{x} \;/\; \mathtt{xy}$ **Always ignored**

Input : xy;

Unable to parse

Lookahead Parsing



$$A ::= B\,; \;/\; B + B\,;$$
$$B ::= \mathtt{x} \;/\; \mathtt{xy}$$

Input : xy;

Lookahead Parsing

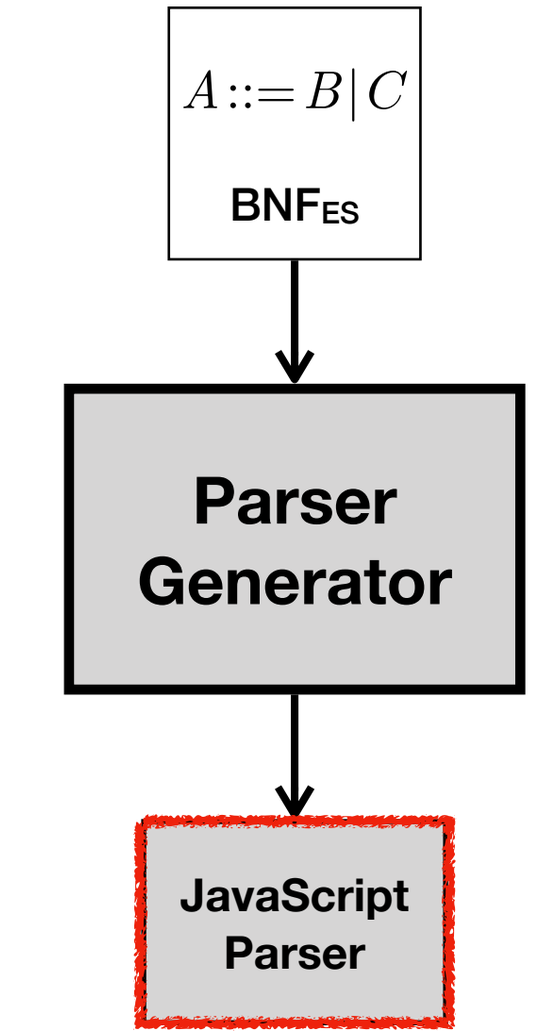
$$
\begin{aligned}
\mathbf{first}_\alpha(s_1 \cdots s_n) &= \mathbf{first}_s(s_1) :+ \mathbf{first}_s(s_2 \cdots s_n) \\
&\quad \text{where } x :+ y = \begin{cases} x \cup y & \text{if } \circ \in x \\ x & \text{otherwise} \end{cases} \\
\mathbf{first}_s(\epsilon) &= \{\circ\} \\
\mathbf{first}_s(\mathsf{a}) &= \{\mathsf{a}\} \\
\mathbf{first}_s(A(a_1, \cdots, a_k)) &= \mathbf{first}_\alpha(\alpha_1) \cup \cdots \cup \mathbf{first}_\alpha(\alpha_n) \\
&\quad \text{where } A(a_1, \cdots, a_k) = \alpha_1 \mid \cdots \mid \alpha_n \\
\mathbf{first}_s(s?) &= \mathbf{first}_s(s) \cup \{\circ\} \\
\mathbf{first}_s(+s) &= \mathbf{first}_s(s) \\
\mathbf{first}_s(-s) &= \{\circ\} \\
\mathbf{first}_s(s \setminus s') &= \mathbf{first}_s(s) \\
\mathbf{first}_s(\langle \neg \mathsf{LT} \rangle) &= \{\circ\}
\end{aligned}
$$

Algorithm for first tokens of BNF$_{ES}$

Algorithm for lookahead parsing

$$
\begin{aligned}
(s_1 \cdots s_n)[L] &= s_1[\mathbf{first}_s(s_2 \cdots s_n) :+ L]\ (s_1 \cdots s_n)[L] \\
\epsilon[L] &= +\mathbf{get}_s(L) \\
\mathsf{a}[L] &= \mathsf{a}\ +\mathbf{get}_s(L) \\
A(a_1, \cdots, a_k)[L] &= \alpha_1[L] \mid \cdots \mid \alpha_n[L] \\
&\quad \text{where } A(a_1, \cdots, a_k) = \alpha_1 \mid \cdots \mid \alpha_n \\
s?[L] &= s[L] \mid \epsilon[L] \\
(\pm s)[L] &= \pm(s[L]) \\
(s \setminus s')[L] &= s[L] \setminus s' \\
\langle \neg \mathsf{LT} \rangle &= \langle \neg \mathsf{LT} \rangle\ +\mathbf{get}_s(L)
\end{aligned}
$$

Implementation



$ArrayLiteral_{[Yield, Await]}$:

[$Elision_{opt}$]

[$ElementList_{[?Yield, ?Await]}$]

[$ElementList_{[?Yield, ?Await]}$, $Elision_{opt}$]

The $ArrayLiteral$ production in ECMAScript 2020

```
type P[T] = List[Boolean] => LAParser[T]
lazy val ArrayLiteral: P[ArrayLiteral] = memo {
  case List(Yield, Await) =>
  "[" ~ opt(Elision) ~ "]"
      ^^ ArrayLiteral0 |
  "[" ~ ElementList(Yield, Await) ~ "]"
      ^^ ArrayLiteral1 |
  "[" ~ ElementList(Yield, Await)
      ~ "," ~ opt(Elision) ~ "]"
      ^^ ArrayLiteral2
}
```

The generated parser for $ArrayLiteral$

Evaluation - Syntax

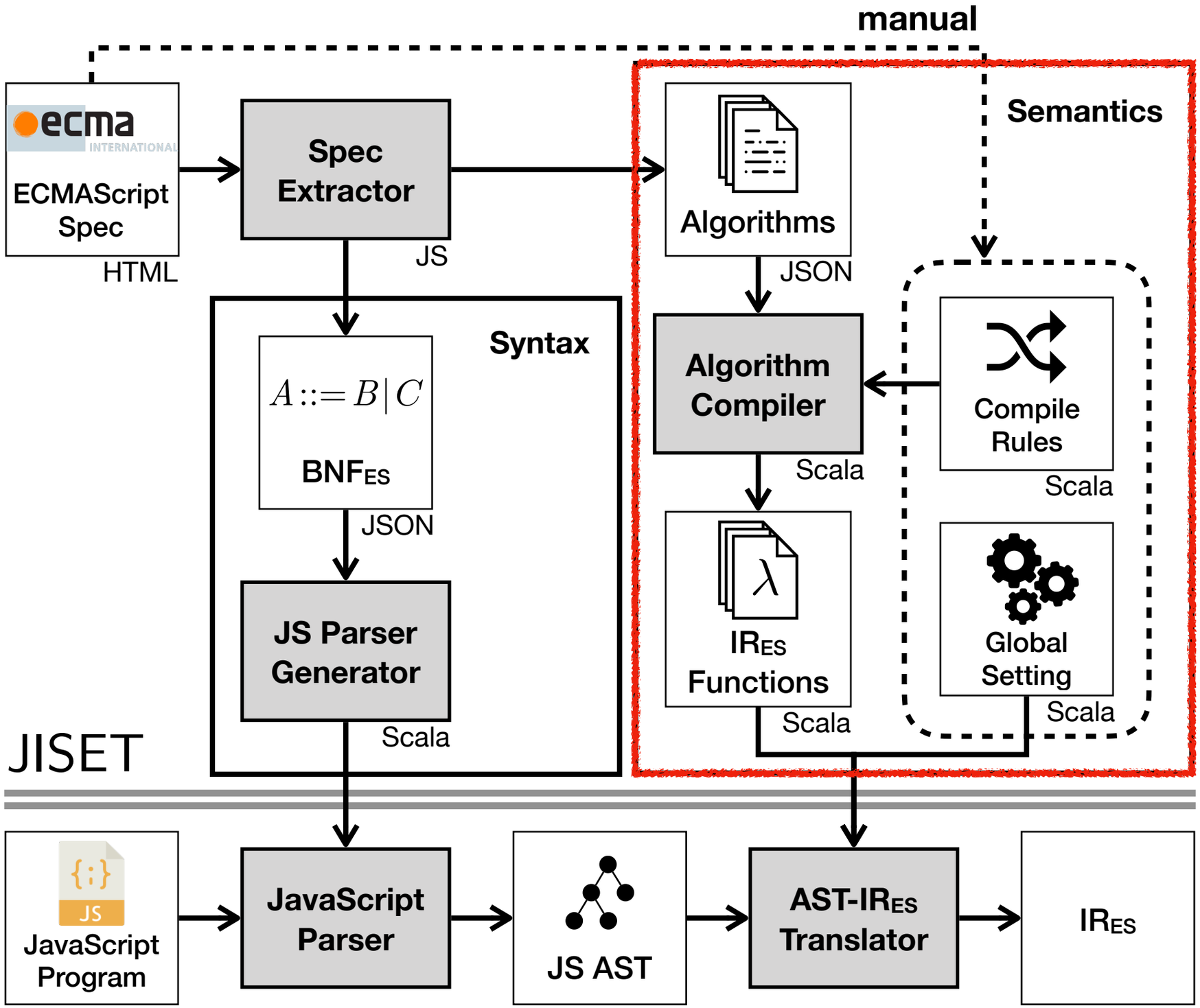
All Success!!

Version	2016	2017	2018	2019	2020	avg.
# Lexical prod.	78	78	78	81	81	79.2
# Syntactic prod.	157	167	167	174	175	168

Test with JS programs in Test262

Old version	2016	2017	2018	2019	avg.
New version	2017	2018	2019	2020	
Δ # Lexical prod.	3	5	6	0	4.7
Δ # Syntactic prod.	140	15	8	2	55

Overview of JISET



JS Semantics Extraction

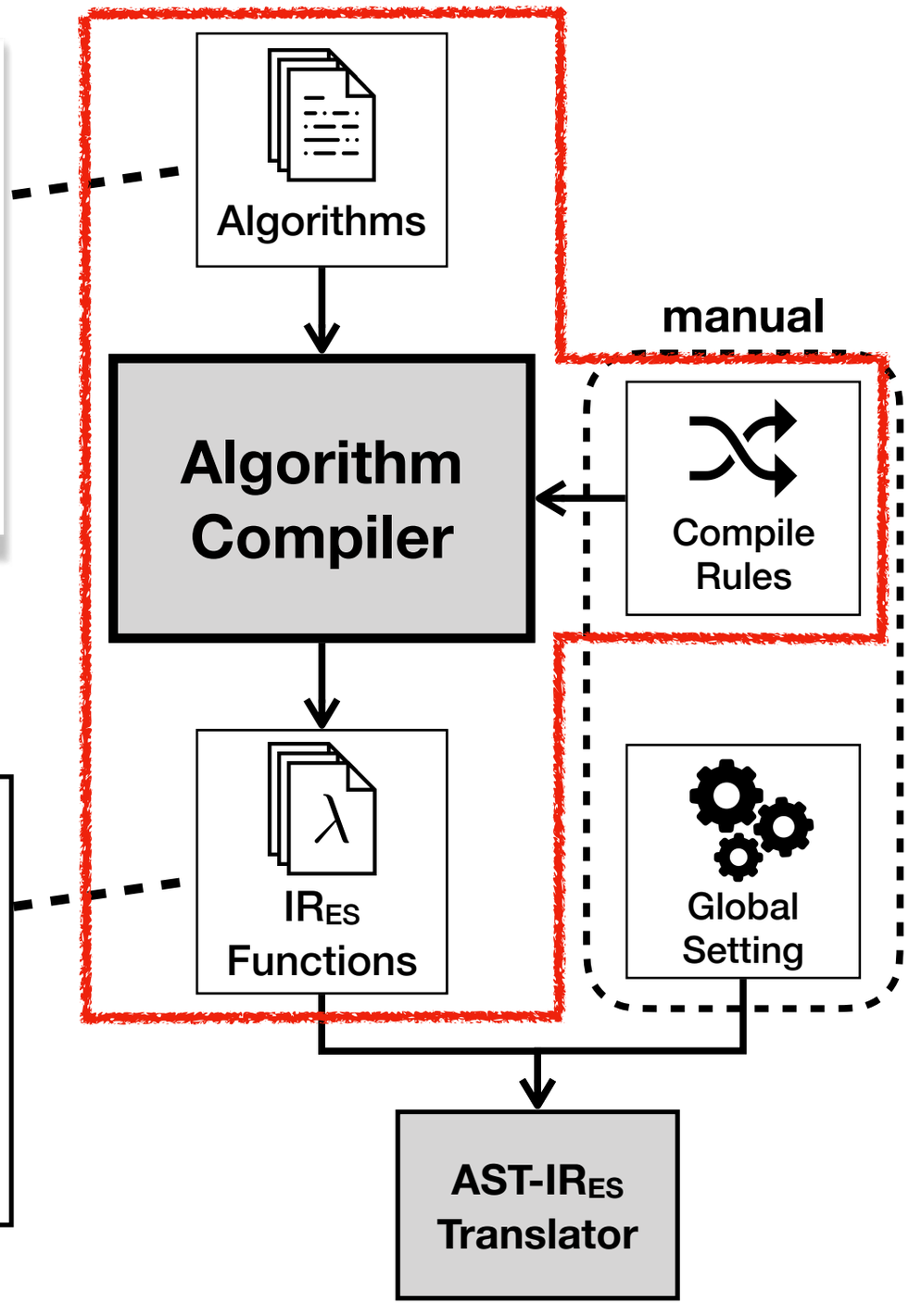
ArrayLiteral : **[** *Elision* **]**

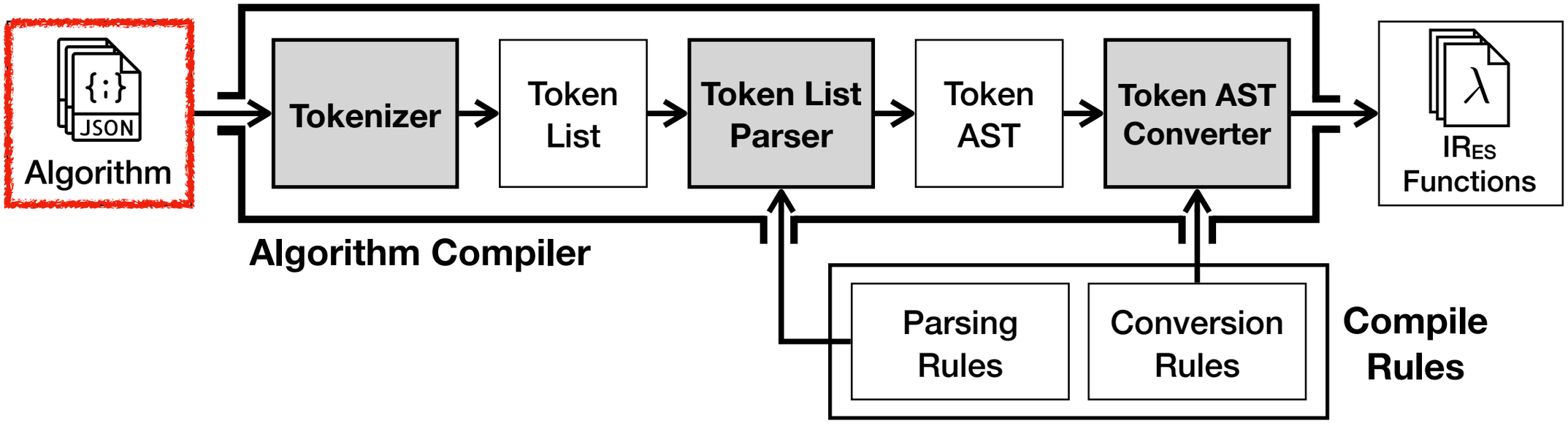
1. Let *array* be ! ArrayCreate(0).
2. If *Elision* is present, then
    a. Let *len* be the result of performing ArrayAccumulation for *Elision* with arguments *array* and 0.
    b. ReturnIfAbrupt(*len*).
3. Return *array*.

The semantics of the first alternative for $ArrayLiteral$

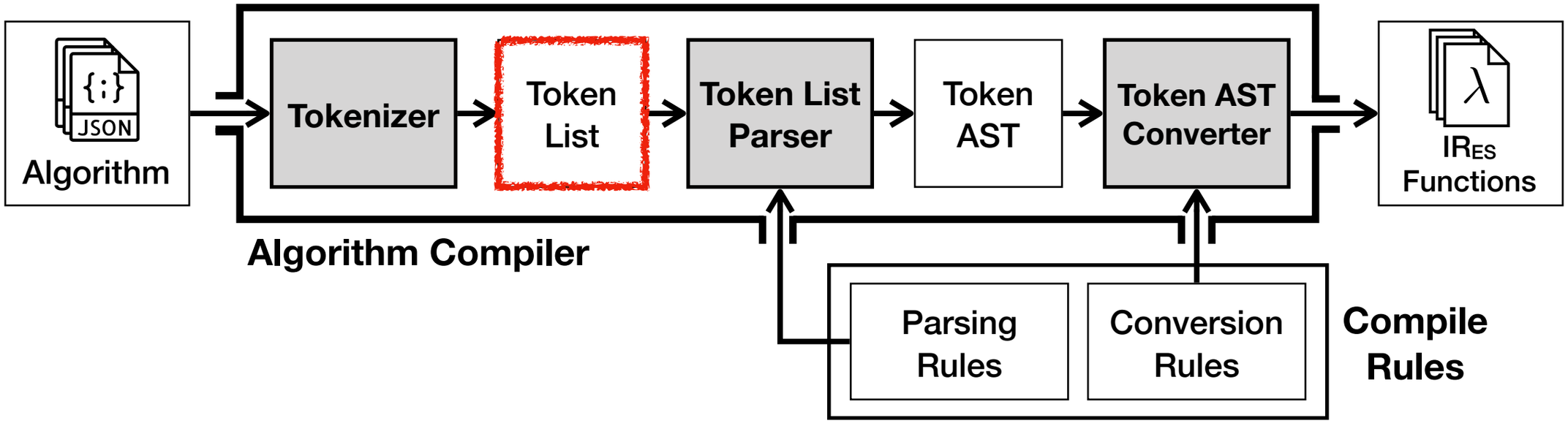
```
"ArrayLiteral0.Evaluation" (Elision) => {
  let array = ! (ArrayCreate 0)
  if (! (= Elision absent)) {
    let len = (Elision.ArrayAccumulation array 0)
    ? len
  }
  return array
}
```

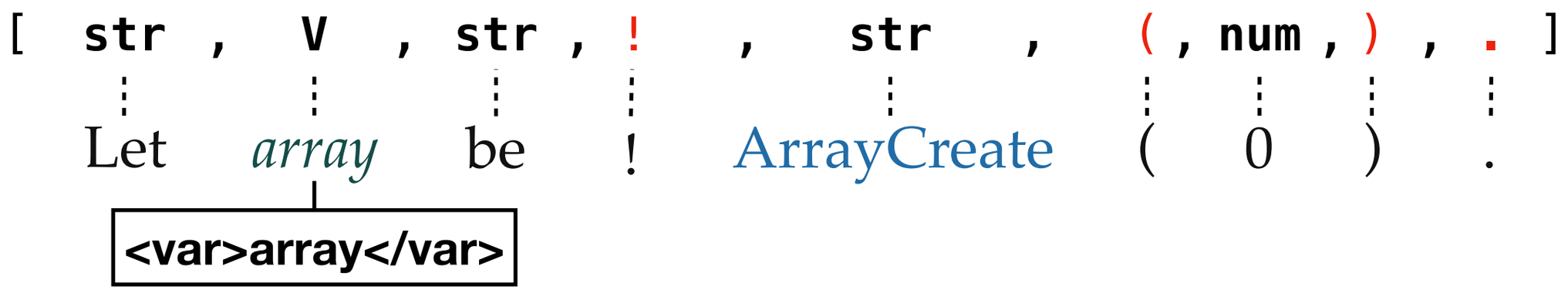
An IR$_{ES}$ function of the first alternative for $ArrayLiteral$

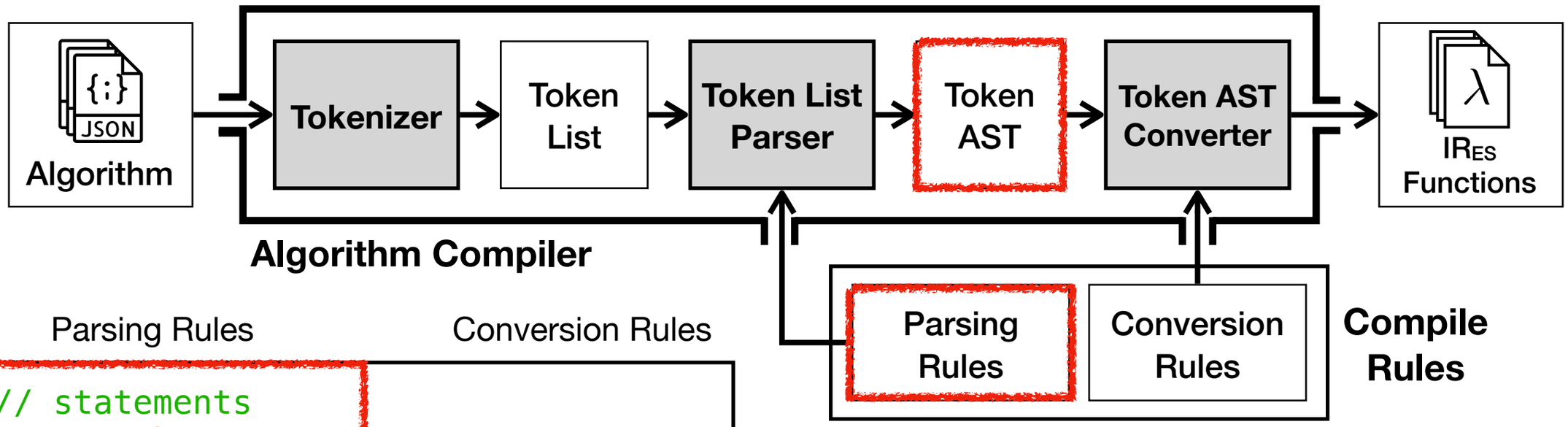




Let *array* be ! ArrayCreate (0) .


JSON
Algorithm
Tokenizer
Token List
Token List Parser
Token AST
Token AST Converter
λ
IRES Functions
Algorithm Compiler
Parsing Rules
Conversion Rules
Compile Rules


[str , V , str , ! , str , (, num ,) , .]
Let array be ! ArrayCreate (0) .
<var>array</var>



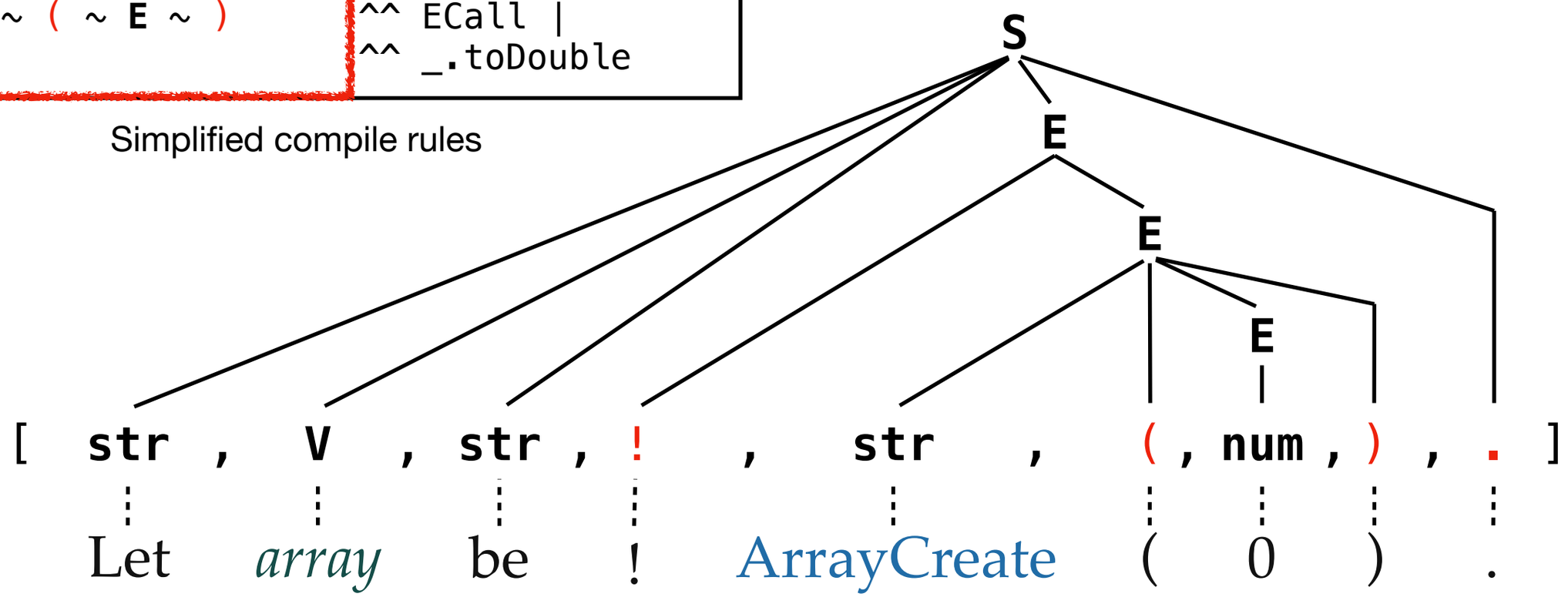
Parsing Rules

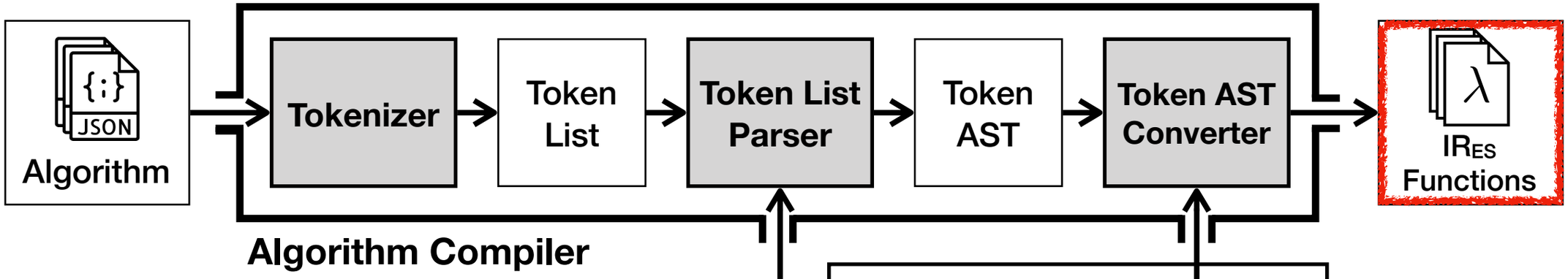
```
S = // statements
  Let ~ V ~ be ~ E ~ .

E = // expressions
  ! E
  str ~ ( ~ E ~ )
  num
```

Conversion Rules

```
^^ ILet


^^ EAbruptCheck |
^^ ECall |
^^ _.toDouble
```

Simplified compile rules





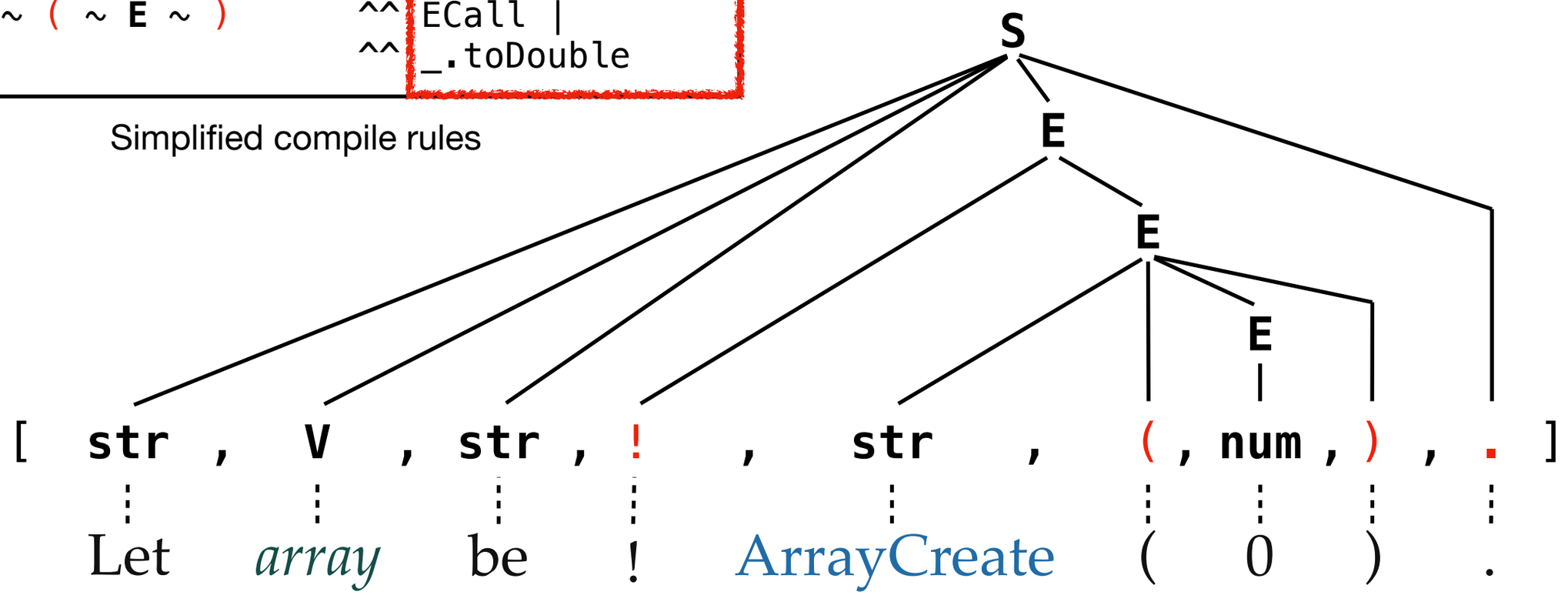
Parsing Rules

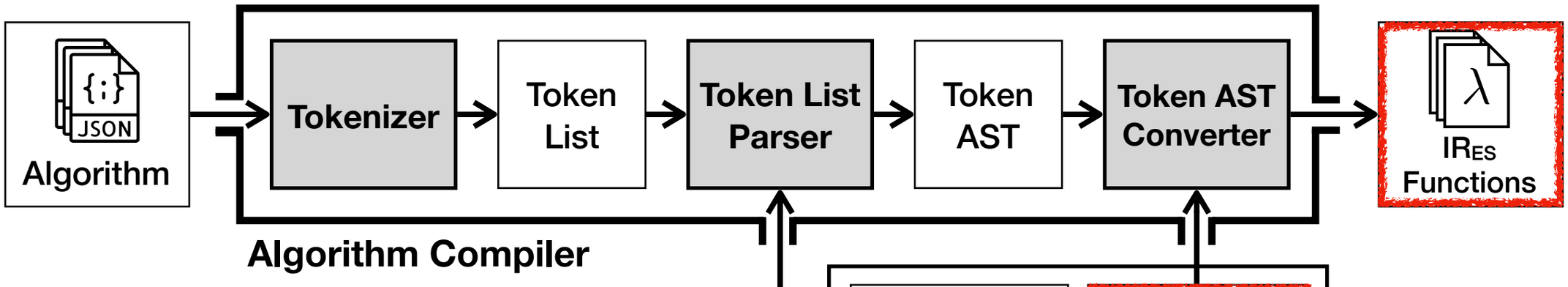
```
S = // statements
  Let ~ V ~ be ~ E ~ . ^^
E = // expressions
  ! E                  ^^
  str ~ ( ~ E ~ )      ^^
  num                  ^^
```

Conversion Rules

```
ILet

EAbruptCheck |
ECall |
_.toDouble
```

Simplified compile rules





Parsing Rules

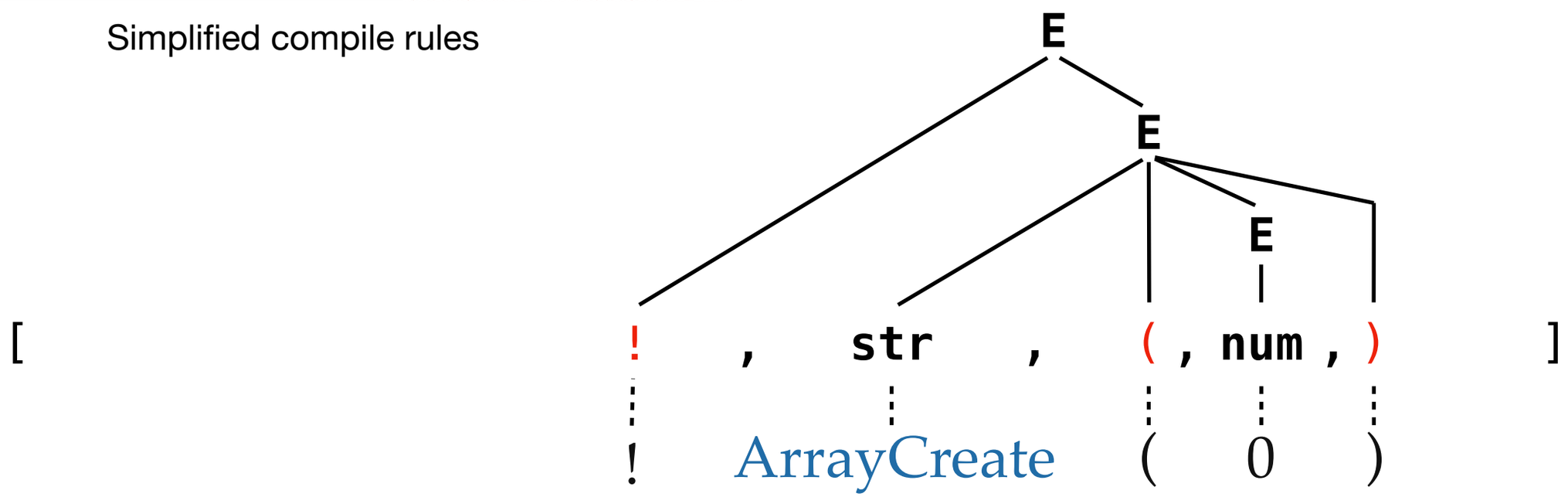
```
S = // statements
  Let ~ V ~ be ~ E ~ . ^^

E = // expressions
  ! E                  ^^
  str ~ ( ~ E ~ )      ^^
  num                  ^^
```

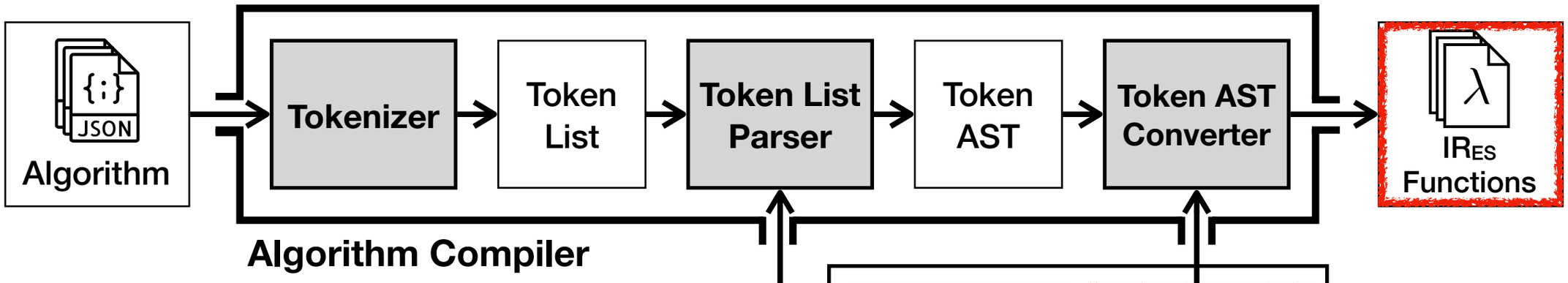
Conversion Rules

```
ILet

EAbruptCheck |
ECall |
_.toDouble
```

```
ILet(array,
                                    )
```

Simplified compile rules





Parsing Rules

```
S = // statements
  Let ~ V ~ be ~ E ~ . ^^

E = // expressions
  ! E                  ^^
  str ~ ( ~ E ~ )      ^^
  num                  ^^
```

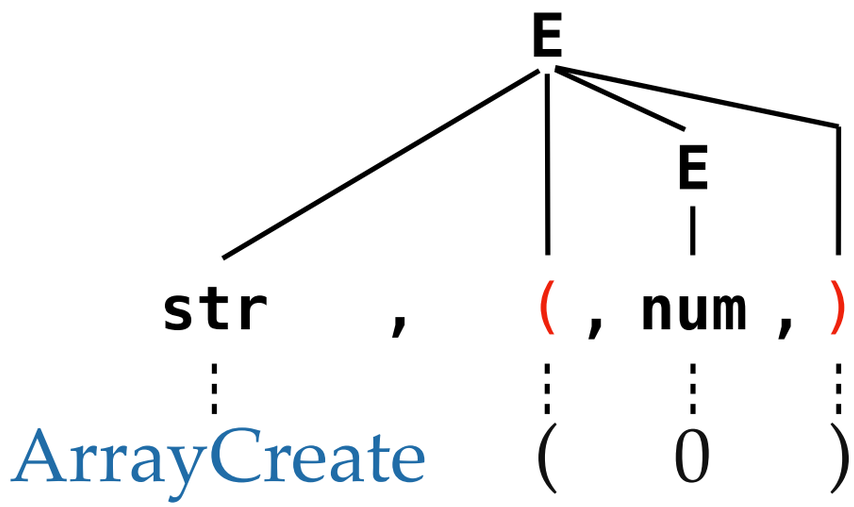
Conversion Rules

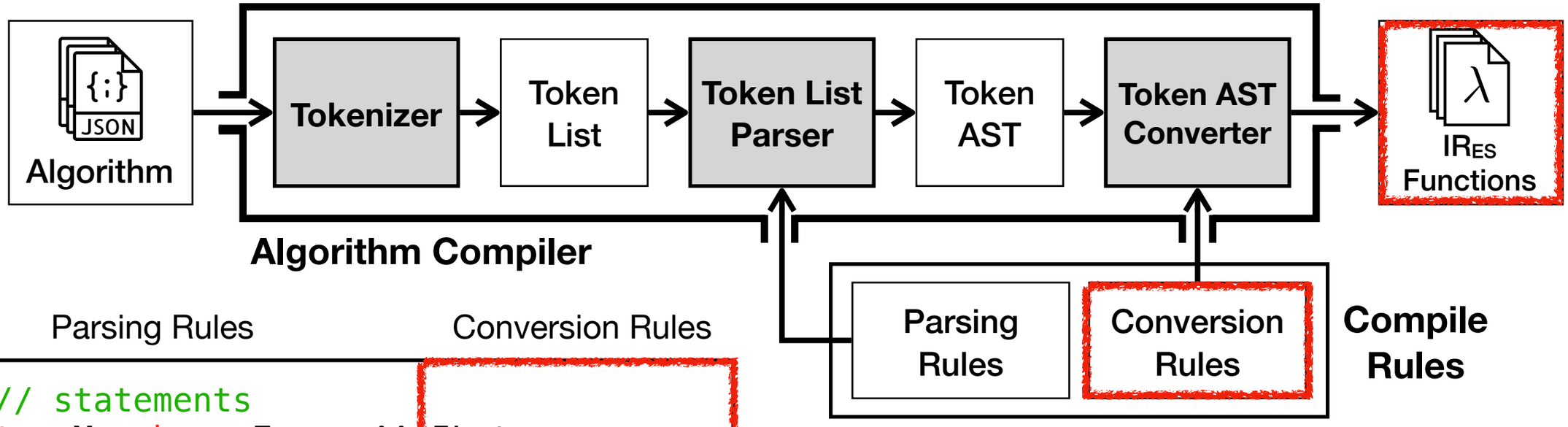
```
ILet

EAbruptCheck |
ECall |
_.toDouble
```

```
ILet(array, EAbruptCheck(
                        ))
```

Simplified compile rules

```
[                                                    ]
```





Parsing Rules

```
S = // statements
  Let ~ V ~ be ~ E ~ . ^^
```

Conversion Rules

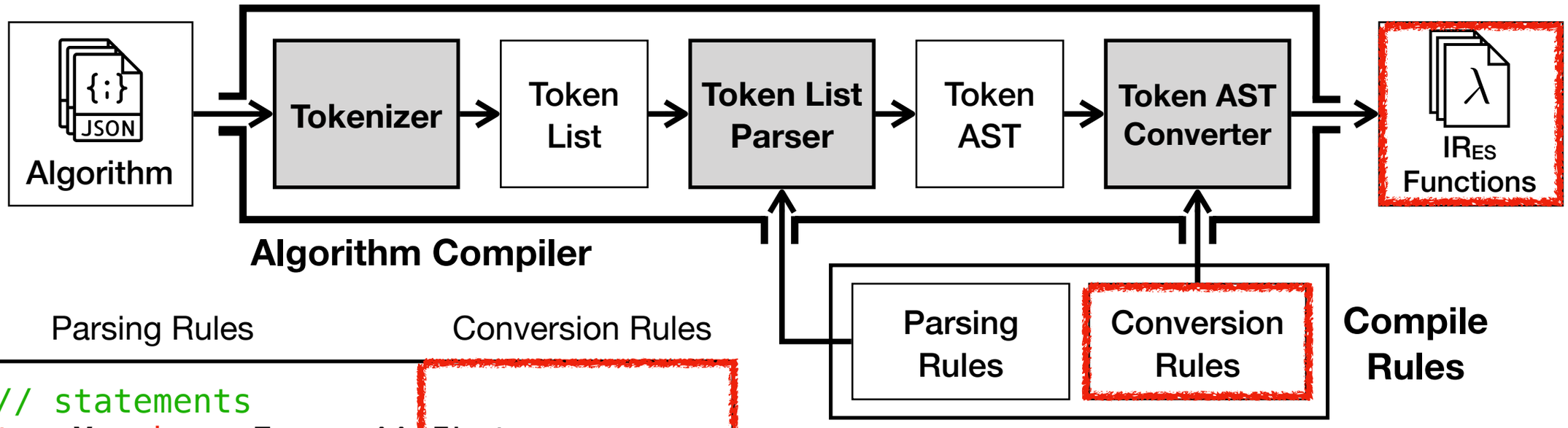
```
ILet
```

```
E = // expressions
  ! E                    ^^ EAbruptCheck |
  str ~ ( ~ E ~ )        ^^ ECall |
  num                    ^^ _.toDouble
```

```
ILet(array, EAbruptCheck(
  ECall("ArrayCreate",  )))
```

Simplified compile rules

```
[                                    E
                                     |
                                    num
                                     ⋮
                                     0                ]
```



Parsing Rules

```
S = // statements
  Let ~ V ~ be ~ E ~ . ^^
```

Conversion Rules

```
ILet
```

```
E = // expressions
  ! E                  ^^ EAbruptCheck |
  str ~ ( ~ E ~ )      ^^ ECall |
  num                  ^^ _.toDouble
```

```
ILet(array, EAbruptCheck(
  ECall("ArrayCreate", 0)))
```

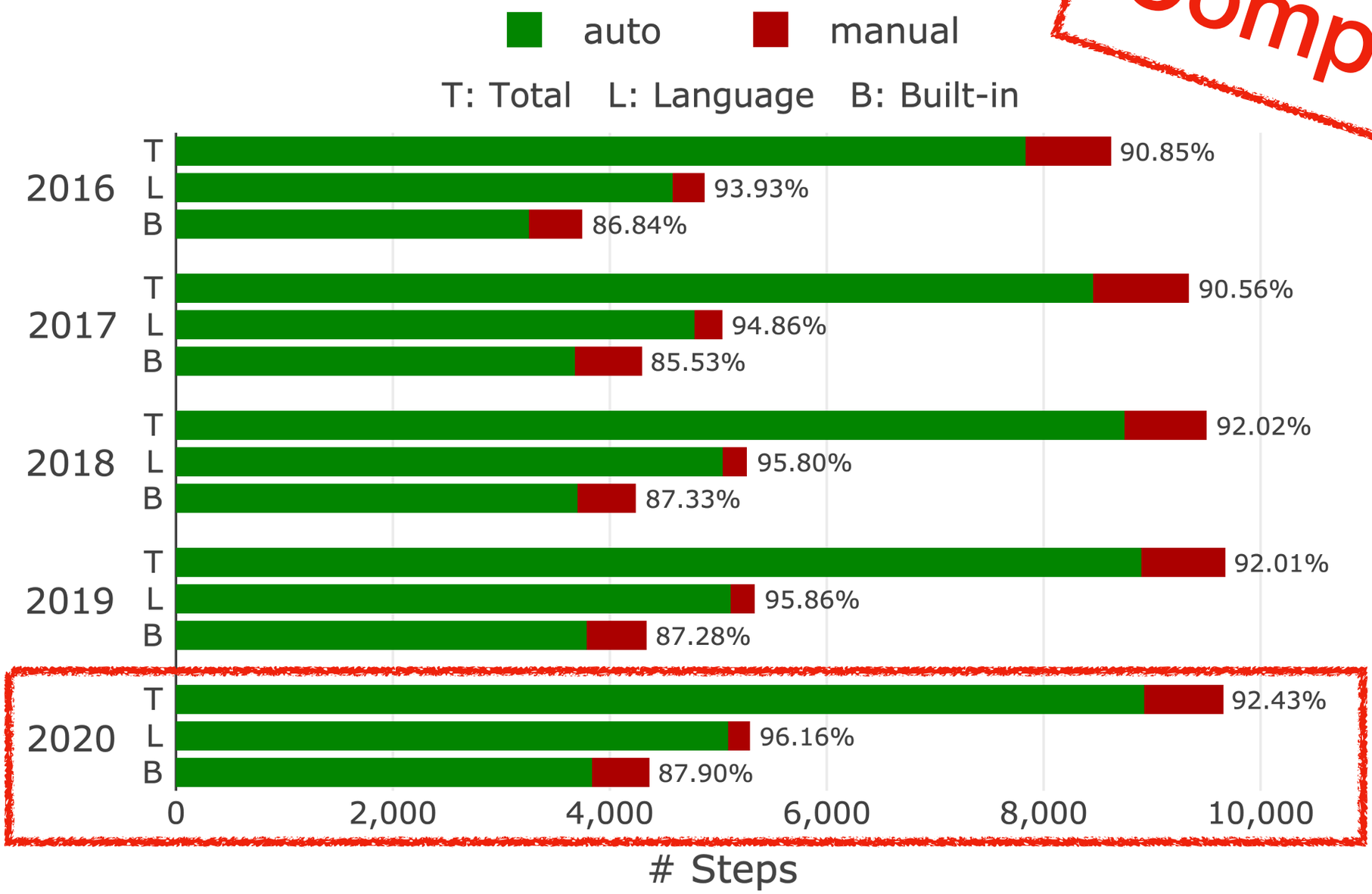
Simplified compile rules

```
let array = ! (ArrayCreate 0)
```

Evaluation - Semantics

Name	Stmt	Expr	Cond	Value	Ty	Ref	SecNo	Total
# Rules	17	16	8	11	33	7	21	113

≥ 90% Compiled



Evaluation - Semantics

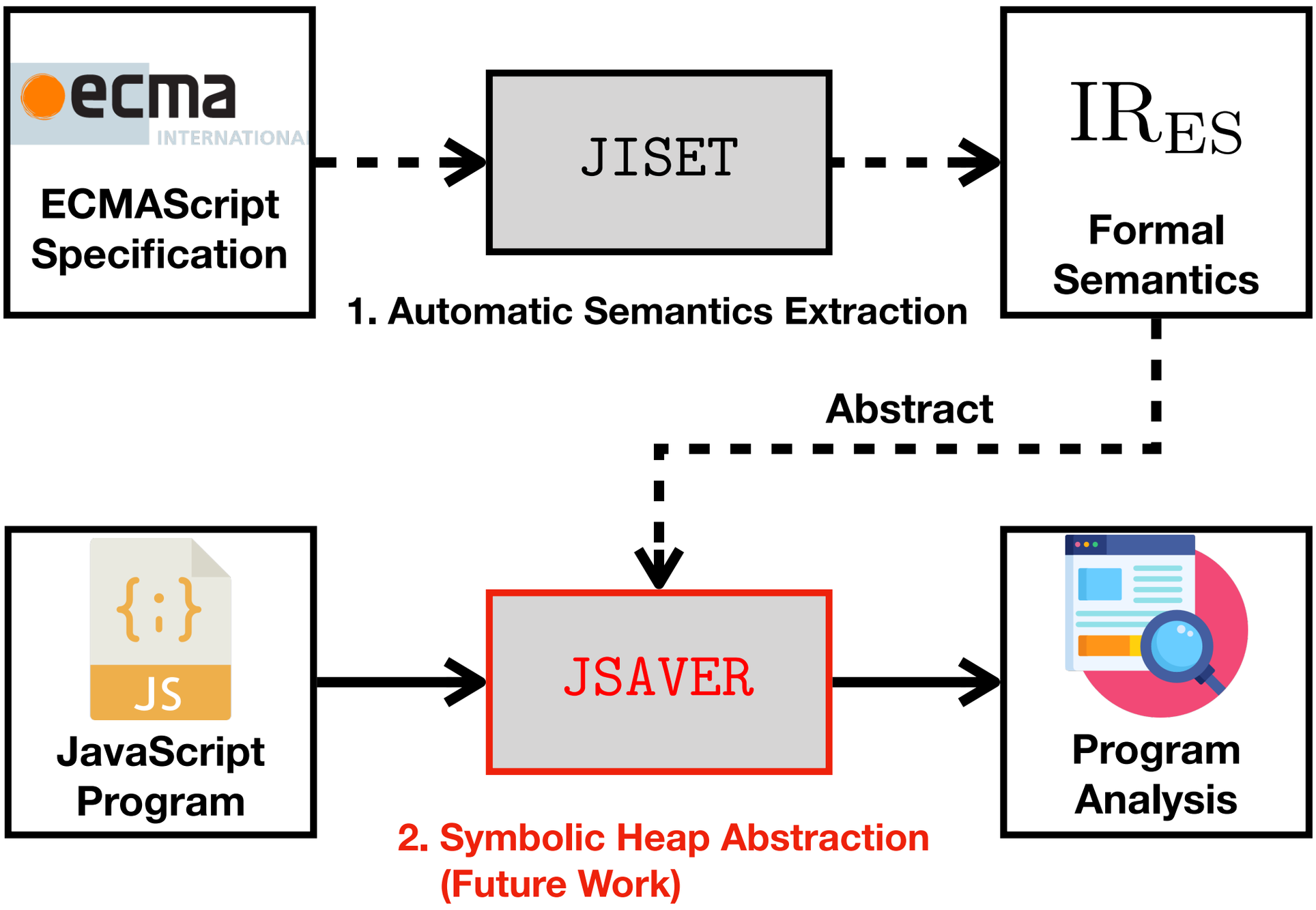
- **Test262** - Official ECMAScript test suite

All Passed

All Test262 Tests	**36,794**
Annexes/Internationalization	1,774
In-progress features	5,895
ECMAScript 2020 Tests	**29,125**
Non-strict mode tests	1,136
Module tests	918
Early error tests	1,316
Inessential built-in object tests	6,473
Applicable Tests	**19,282**
Passed tests	19,220
Failed tests	62

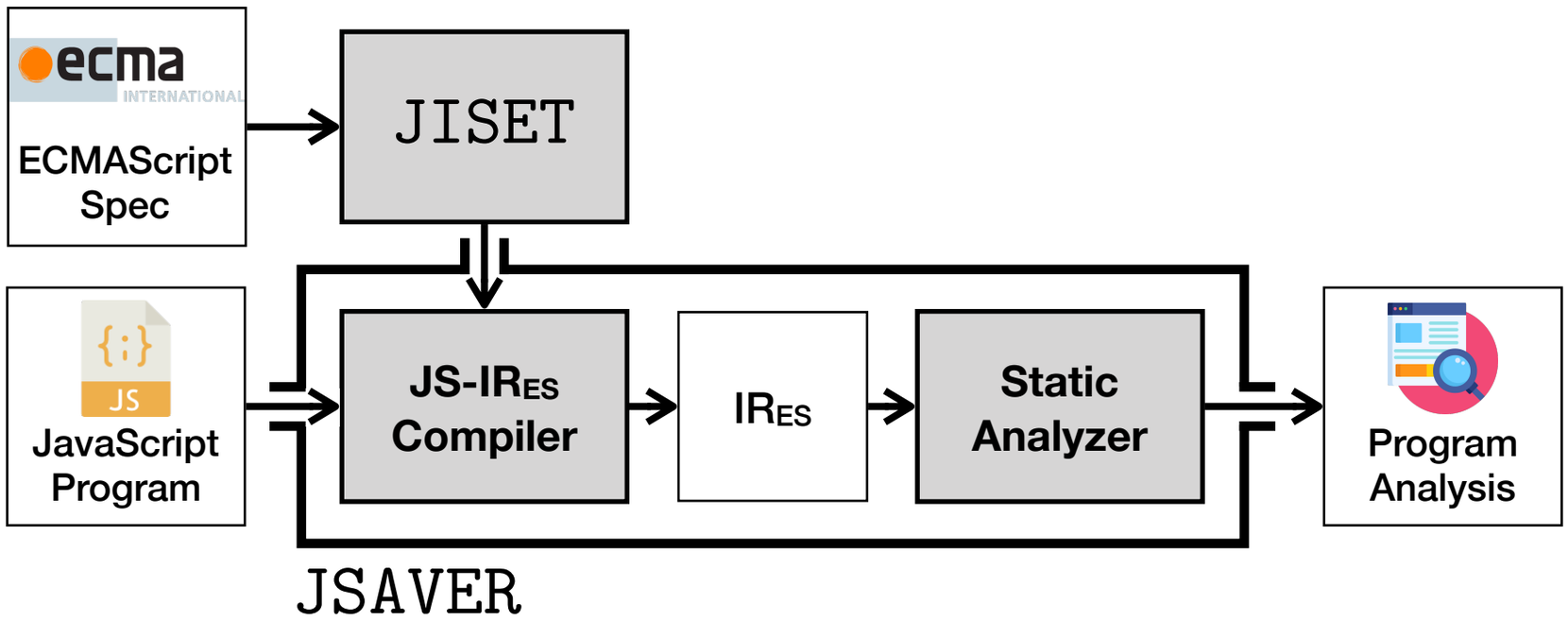
- 5 Wrong
- 54 Impl. Dep
- 3 Timeout (20 min.)

https://github.com/tc39/test262

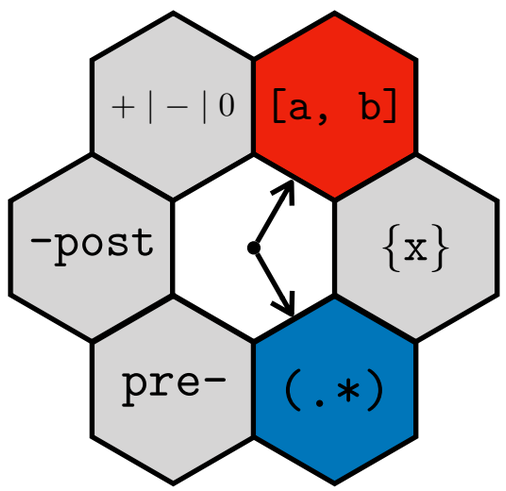

ECMAScript Specification
JISET
IR_{ES}
Formal Semantics
1. Automatic Semantics Extraction
Abstract
JavaScript Program
JSAVER
Program Analysis
2. Symbolic Heap Abstraction (Future Work)

JSAVER

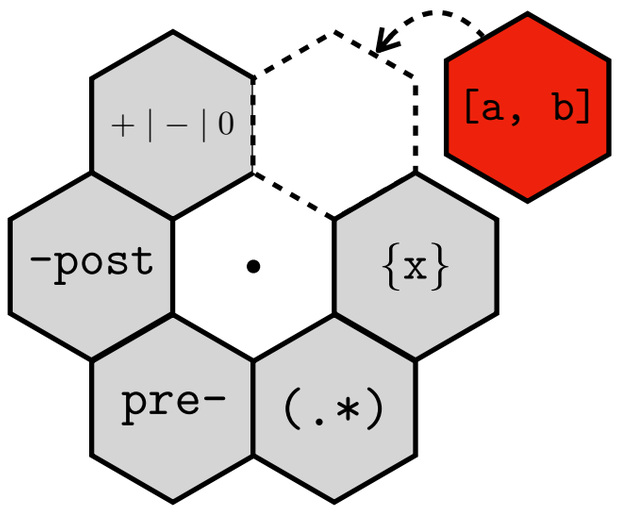
- **J**avaScript **S**tatic **A**nalyzer **V**ia **E**CMAScript **R**epresentations



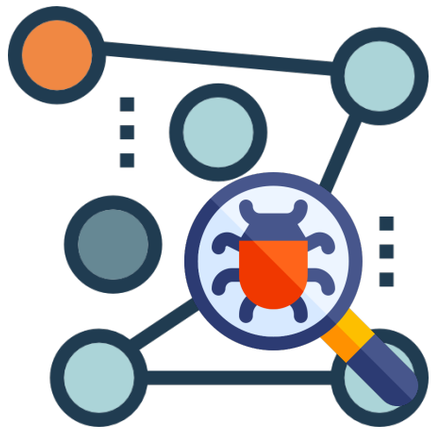
Usability of JSAVER



Pluggability



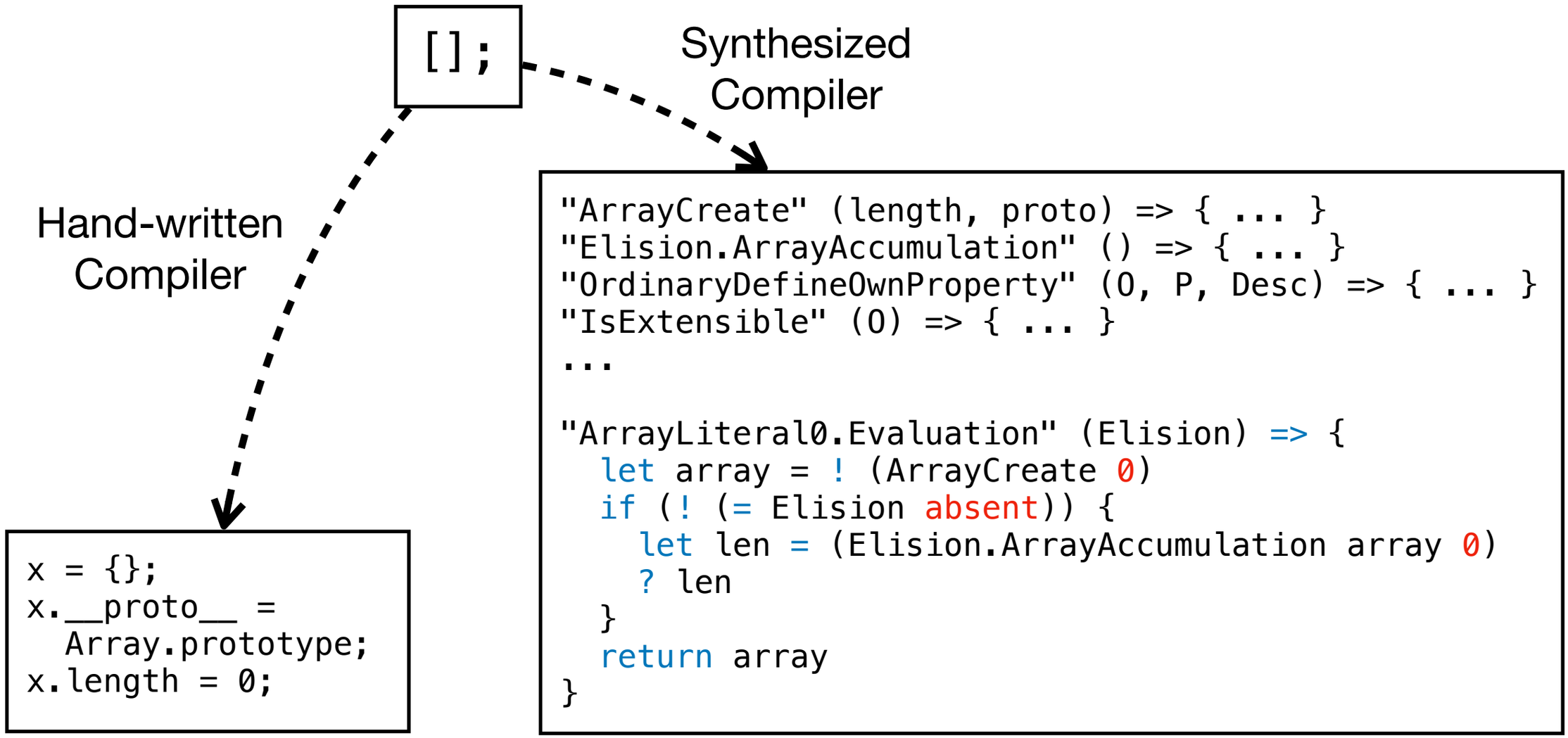
Extensibility



Debuggability

Analysis of JavaScript Web Applications Using SAFE 2.0
(Published in ICSE'17 Demonstrations Track)

Scalability of JSAVER



IR $\mathrm{IR}_{\mathrm{ES}}$

$$||\mathrm{IR}|| \ll ||\mathrm{IR}_{\mathrm{ES}}||$$

Dynamic Features in JavaScript

- Open objects

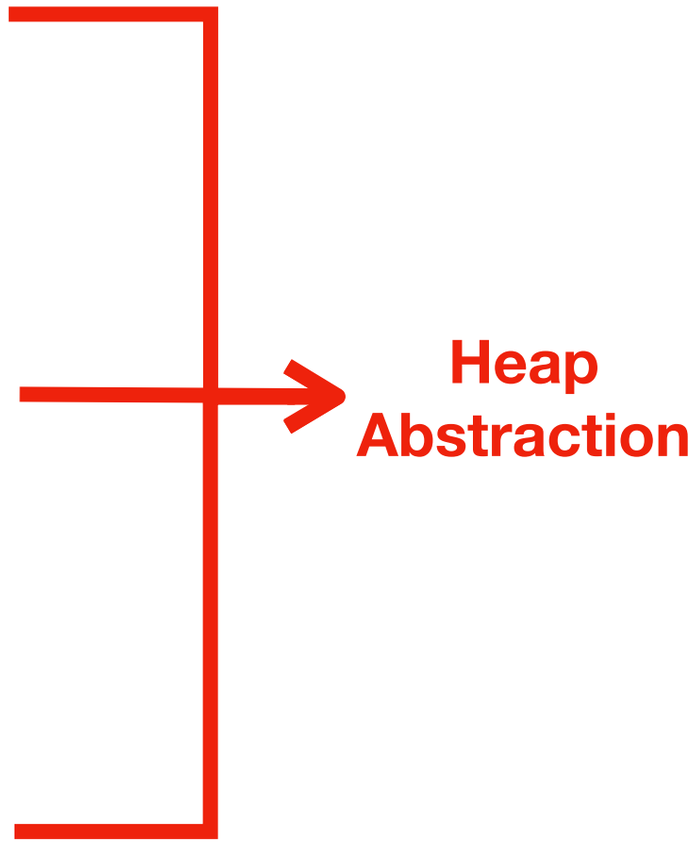
```
let o = { p: 1 };
o.q = 2; delete o.p;
```

- First-class functions

```
let f = (g) => g(42);
f(x => x+1); f(x => x*2);
```

- First-class property names

```
let o = { x7: 42 };
o['x' + (3 + 4)];
```



Allocation-Site Abstraction

Only Weak Updates

```
function f(n) {
  return {p: n + 7}; // a0
}
let x = f(0);
let y = f(1);
let z = f(2);
z.p = 42;
```

An example code

$\mathbb{E}^\sharp$

```
n   -> {0,1,2}
RET -> a0
```

$\mathbb{H}^\sharp$

```
a0 -> {p: 7,8,9}
```

Abstract heap for f

$\mathbb{E}^\sharp$

```
x -> a0
y -> a0
z -> a0
```

$\mathbb{H}^\sharp$

```
a0 -> {p: 7,8,9,42}
```

Global abstract heap

Recency Abstraction

Strong Updates for Recent Objects

```
function f(n) {
  return {p: n + 7}; // a0
}
let x = f(0);
let y = f(1);
let z = f(2);
z.p = 42;
```

An example code

$\mathbb{E}^\sharp$

```
n   -> {0,1,2}
RET -> a0:r
```

$\mathbb{H}^\sharp$

```
a0:o -> {p: 7,8,9}
a0:r -> {p: 7,8,9}
```

Abstract heap for f

$\mathbb{E}^\sharp$

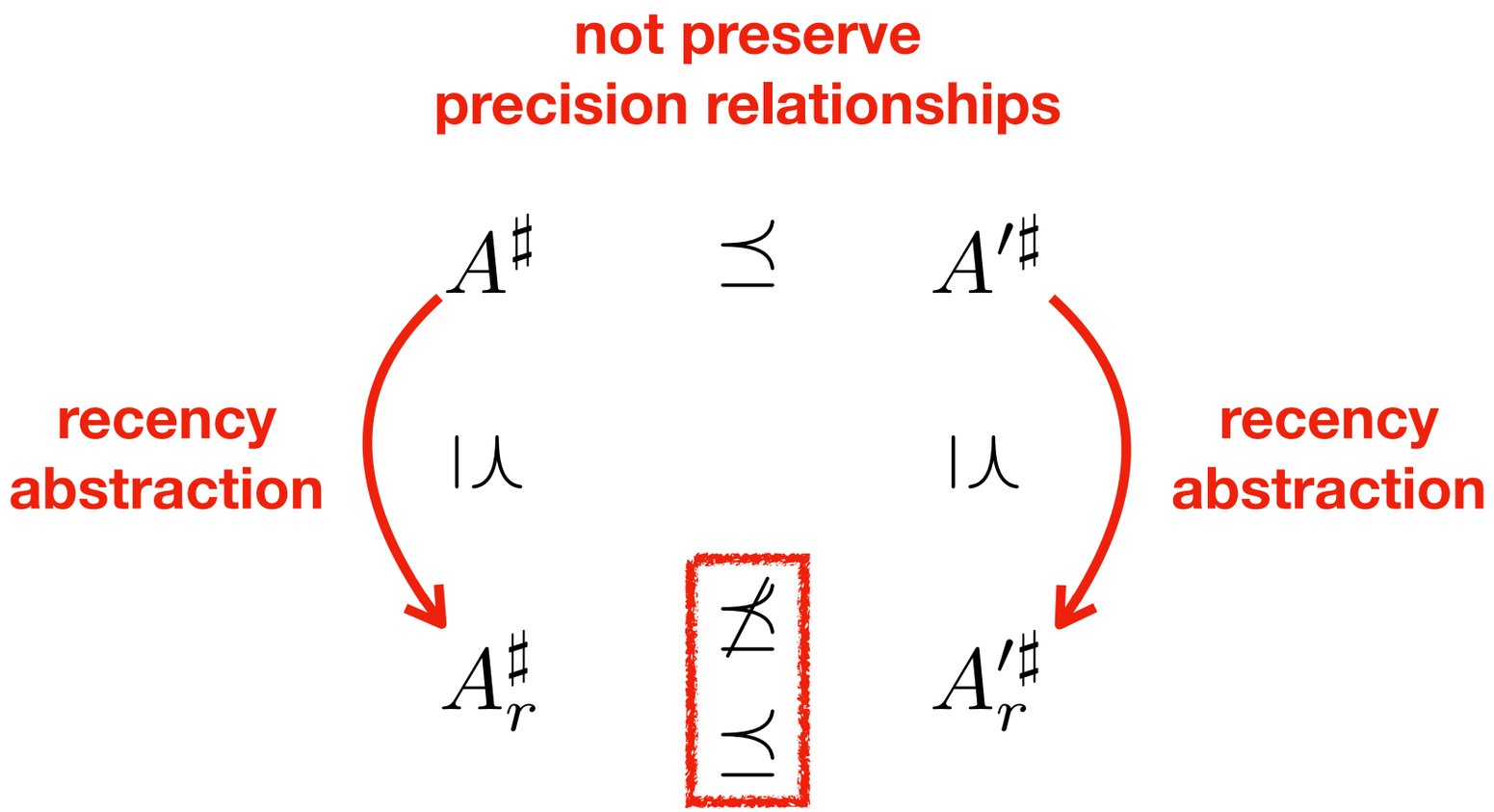
```
x -> a0:o
y -> a0:o
z -> a0:r
```

$\mathbb{H}^\sharp$

```
a0:o -> {p: 7,8,9}
a0:r -> {p: 42}
```

Global abstract heap

Recency Abstraction



Revisiting Recency Abstraction for JavaScript:
Towards an Intuitive, Compositional, and Efficient Heap Abstraction
(Published in SOAP'17)

Symbolic Heap Abstraction

```
function f(n) {
  return {p: n + 7}; // a0
}
let x = f(0); // c0
let y = f(1); // c1
let z = f(2); // c2
z.p = 42;
```

An example code

Ω

```
ωn -> {0,1,2}
```

$\mathbb{E}^\sharp$

```
n   -> ωn
RET -> a0:⊥
```

$\mathbb{H}^\sharp$

```
a0:⊥ -> {p: ωn + 7}
```

Abstract heap for f

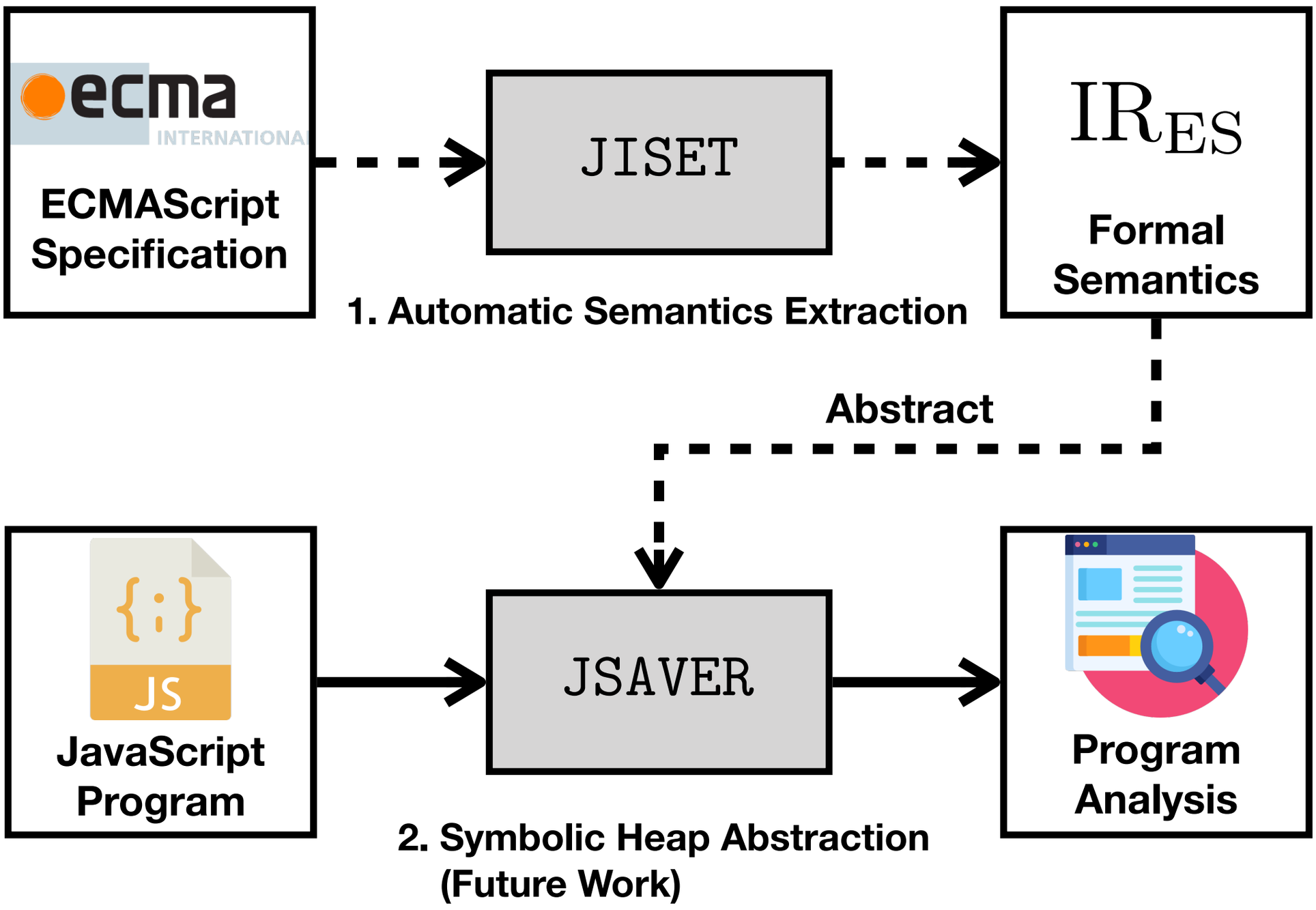
$\mathbb{E}^\sharp$

```
x -> a0:c0
y -> a0:c1
z -> a0:c2
```

$\mathbb{H}^\sharp$

```
a0:c0 -> {p: 7}
a0:c1 -> {p: 8}
a0:c2 -> {p: 42}
```

Global abstract heap


ECMAScript Specification
JISET
IR_{ES}
Formal Semantics
1. Automatic Semantics Extraction
Abstract
JavaScript Program
JSAVER
Program Analysis
2. Symbolic Heap Abstraction (Future Work)

Backup Slides

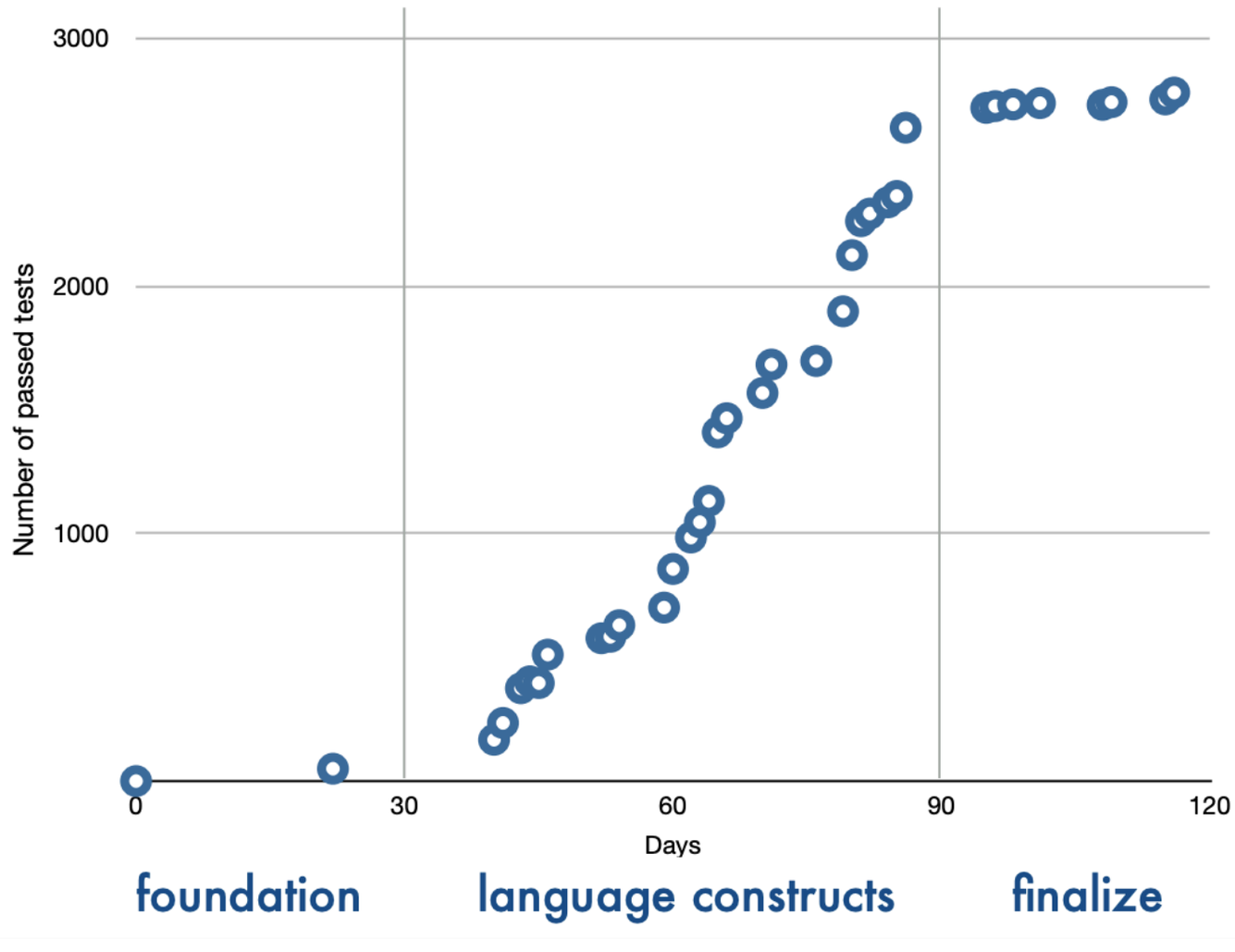
Size of ECMAScript Spec.

Edition	Date	# Pages
1	1997/06	110
2	1998/06	117
3	1999/12	188
5	2009/12	252
5.1	2011/06	258
6	2015/06	566
7	2016/06	586
8	2017/06	885
9	2018/06	805
10	2019/06	764

Development cost of KJS

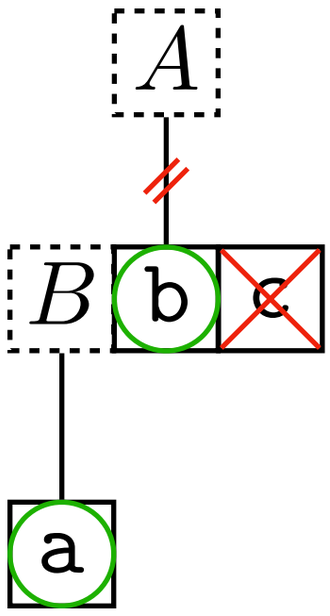
Took only *four months* by a first year PhD student.

semantic rules: 1,370



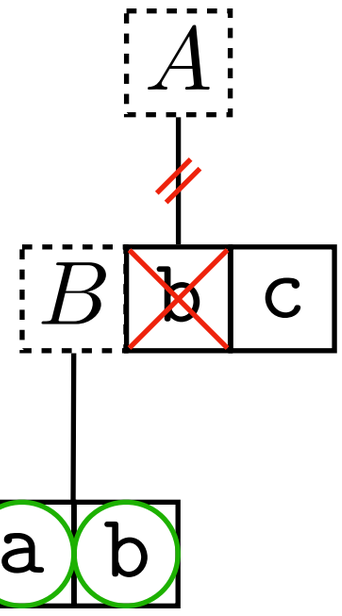
Parsing Expression Grammar

- **Re-orderings are not always solutions**



$A ::= B\ \texttt{bc}$
$B ::= \texttt{a} \mid \texttt{ab}$

abbc



$A ::= B\ \texttt{bc}$
$B ::= \texttt{ab} \mid \texttt{a}$

abc

Timeout Test in Test262

```
var x = [0, 1, 2];
x[4294967294] = 4294967294;
x.length = 2;

alert(x[0] === 0);
alert(x[1] === 1);
alert(x[2] === undefined);
alert(x[4294967294] === undefined);
```

test262/test/built-ins/Array/length/S15.4.5.2_A3_T4.js