2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)

ppt:https://plrg.korea.ac.kr/assets/data/slides/2020/ase20-slides.pdf

JISET: JavaScript IR-based Semantics Extraction Toolchain

js基于IR的语义提取工具链

Jihyeok Park
KAIST, South Korea
jhpark0223@kaist.ac.kr

Jihee Park
KAIST, South Korea
j31d0@kaist.ac.kr

Seungmin An
KAIST, South Korea
h2oche@kaist.ac.kr

Sukyoung Ryu
KAIST, South Korea
sryu.cs@kaist.ac.kr

ABSTRACT

JavaScript was initially designed for client-side programming in web browsers, but its engine is now embedded in various kinds of host software. Despite the popularity, since the JavaScript semantics is complex especially due to its dynamic nature, understanding and reasoning about JavaScript programs are challenging tasks. Thus, researchers have proposed several attempts to define the formal semantics of JavaScript based on ECMAScript, the official JavaScript specification. However, the existing approaches are manual, labor-intensive, and error-prone and all of their formal semantics target ECMAScript 5.1 (ES5.1, 2011) or its former versions. Therefore, they are not suitable for understanding *modern JavaScript* language features introduced since ECMAScript 6 (ES6, 2015). Moreover, ECMAScript has been annually updated since ES6, which already made five releases after ES5.1.

To alleviate the problem, we propose JISET, a JavaScript IR-based Semantics Extraction Toolchain. It is the first tool that *automatically synthesizes* parsers and AST-IR translators directly from a given language specification, ECMAScript. For syntax, we develop a parser generation technique with *lookahead parsing* for BNF$_{ES}$, a variant of the extended BNF used in ECMAScript. For semantics, JISET synthesizes AST-IR translators using *forward compatible* rule-based compilation. *Compile rules* describe how to convert each step of abstract algorithms written in a structured natural language into IR$_{ES}$, an Intermediate Representation that we designed for ECMAScript. For the four most recent ECMAScript versions, JISET automatically synthesized parsers for all versions, and compiled 95.03% of the algorithm steps on average. After we complete the missing parts manually, the extracted core semantics of the latest ECMAScript (ES10, 2019) passed all 18,064 applicable tests. Using this *first formal semantics of modern JavaScript*, we found nine specification errors in ES10, which were all confirmed by the Ecma Technical Committee 39. Furthermore, we showed that JISET is *forward compatible* by applying it to nine feature proposals ready for inclusion in the next ECMAScript, which let us find three errors in the BigInt proposal.

从一个给定的语言规范中自动合成解析器和AST-IR的翻译器

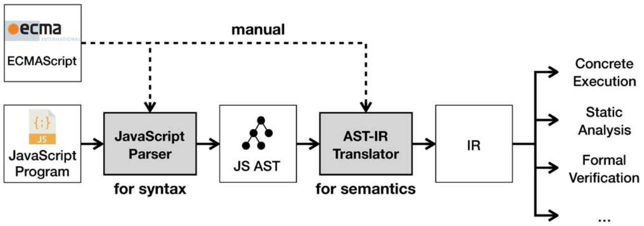


Figure 1: Existing approaches: Manually built parsers and AST-IR translators for JavaScript IR-based semantics

KEYWORDS

JavaScript, mechanized formal semantics, program synthesis

1 INTRODUCTION

JavaScript is one of the most widely used programming languages not only for client-side but also for server-side programming [5, 6] and even for small embedded systems [3, 9]. It is the top-ranked language used in active GitHub repositories[1], and #7 in the TIOBE Programming Community index[2]. According to W3Techs[3], 95.0% of websites use JavaScript as their client-side programming language.

Despite its popularity, JavaScript developers often suffer from its intricate semantics, which may cause unexpected behaviors. For example, the following function may seem to always return `false`:

复杂语义→意想不到的行为

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

Unfortunately, it returns `true` when its argument is an empty array `[]`. To correctly understand and reason about such a complex behavior, the formal semantics of JavaScript is necessary.

不太理解这个例子体现出形式化语言的必要性在哪里？（笑哭）

Researchers have defined various JavaScript formal semantics [17, 18, 21, 25] suitable for static analysis [19, 20, 23, 29] and formal verification [17] by referring to ECMAScript. ECMAScript is the official specification that describes the JavaScript syntax using a variant of the extended BNF (EBNF) notation, and its semantics using abstract algorithms written in English in a clear and structured manner. *IR-based semantics extraction* is a traditional way to define the formal semantics of a language by building a compiler front-end that takes programs and produces their Intermediate Representations (IRs) to indirectly represent the semantics of the given programs. As illustrated in Figure 1, a compiler front-end consists of a parser that constructs Abstract Syntax Trees (ASTs) of given JavaScript programs, and an AST-IR translator that converts ASTs to their own IRs. It helps researchers focus on IRs without worrying about

ECMAScript是一JS的官方规范

ASE '20, September 21–25, 2020, Virtual Event, Australia
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-6768-4/20/09…$15.00
https://doi.org/10.1145/3324884.3416632

[1] https://githut.info/
[2] https://www.tiobe.com/tiobe-index/
[3] https://w3techs.com/technologies/details/cp-javascript/all/all

diverse and enormous features of JavaScript in developing new techniques for static analysis and formal verification.

目前针对Js的基于IR的语义提取的工作中，全是手动构建解析器、翻译器

表明传统的方法很耗时（构建翻译器）

（1）手动方法似乎没有足够的可扩展性
（2）ES6到ES10没有正式的语义

However, to the best of our knowledge, all existing approaches to JavaScript IR-based semantics extraction *manually* build parsers and translators. Although manually building them was reasonable until ECMAScript 5.1 (ES5.1, 2011) [1], it is too tedious, labor-intensive, and error-prone to deal with the large size of *modern JavaScript* since ECMAScript 6 (ES6, 2015) [2]. ES6 introduced numerous significantly new features such as lexical binding via `let`, the spread `...` operator, classes, the `for-of` operator, the `async` functions, and generators. For example, consider KJS [25], one of formal semantics of ES5.1 defined on top of $\mathbb{K}$, which is a framework for defining language semantics. According to an author of KJS, it took *four months* to implement an AST-IR translator for 1,370 steps out of 2,932 steps in 368 abstract algorithms [24]. However, the most recent version of ECMAScript (ES10, 2019) [7] has 2,026 abstract algorithms consisting of 10,101 steps. Thus, the manual approaches do not seem to be scalable enough to build an AST-IR translator for modern JavaScript, and indeed no formal semantics exists for ES6 to ES10.

js的语法和语义每年更新1次，大量新的规范不断涌现进来。（导致手工提取语义更加复杂低效）

Moreover, JavaScript syntax and semantics are annually updated. Until ES5.1, JavaScript was a stable language because the specification was rarely updated. However, the Ecma Technical Committee 39 (TC39) [8] decided to release the specification annually in late 2014. After this official announcement, several syntax features and roughly 1,000 to 3,000 steps of abstract algorithms have been modified or newly added in the specification every year. To handle these frequent and massive updates of ECMAScript, the manual approaches require researchers to manually update parsers and AST-IR translators, which incurs tremendous efforts.

提出了一种技术，可以直接从具有前向兼容性的ECMAScript中自动合成解析器和AST-IR翻译器的方法

这里列举了一些合成解析器和翻译器的挑战

To alleviate this problem, we propose a technique to *automatically synthesize* parsers and AST-IR translators directly from ECMAScript with *forward compatibility*. There are several technical challenges in synthesizing parsers and translators. For syntax, ECMAScript utilizes its own variant of EBNF with parametric nonterminals, conditional alternatives, and various special terminal symbols. Thus, no existing parser generation technique is directly applicable for this variant. Moreover, JavaScript provides automatic semicolon insertion in its parsing algorithm with several complex rules, not in a lexer. For semantics, abstract algorithms in ECMAScript are written in English. Besides, a general and forward compatible representation of abstract algorithms is necessary to support future versions of ECMAScript.

Our contribution is JISET, a JavaScript IR-based Semantics Extraction Toolchain:

第一个从语言规范中提取基于IR语义的工具

- JISET **is the *first tool that automatically extract* IR-based semantics from a language specification, ECMAScript.** For syntax, we formally introduce a variant of EBNF, BNF$_{\text{ES}}$, and propose a parser generation technique with lookahead parsing for BNF$_{\text{ES}}$, which supports automatic semicolon insertion. For semantics, we propose semi-automatic synthesis of AST-IR translators assisted by compile rules. Compile rules describe how to convert each step of abstract algorithms into our intermediate representation IR$_{\text{ES}}$ designed for ECMAScript. We evaluated JISET with the four most recent ECMAScript versions (ES7 to ES10). JISET automatically generated parsers for all versions, and automatically compiled 95.03% of the steps in abstract algorithms on average.

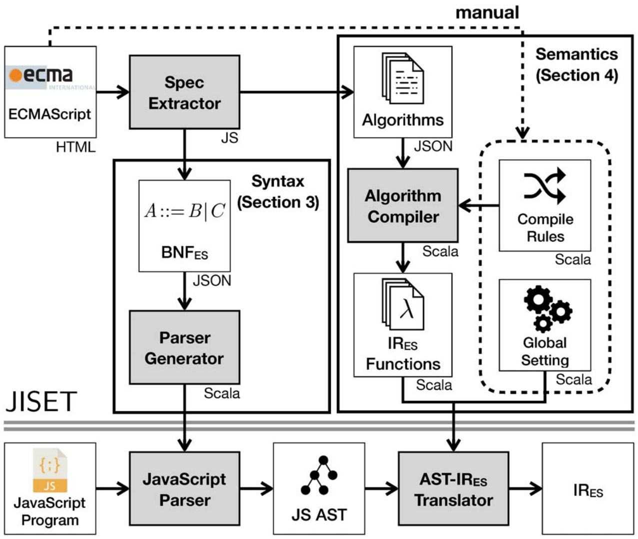


Figure 2: Overall structure of JISET**: Automatically synthesized** JavaScript Parser **and** AST-IR$_{\text{ES}}$ Translator **for JavaScript IR-based semantics**

- JISET **bridges gaps between the specification written in a natural language and tests.** To evaluate the correctness of JISET, we checked the extracted semantics with the official test conformance suite, Test262 [10]. By manually completing missing parts of the AST-IR translator for the latest ECMAScript (ES10, 2019), we defined the *first IR-based formal semantics of modern JavaScript*. It failed for 1,709 tests because of specification errors in ES10. Using the tests, we found eight specification errors, three of which had not been reported before. They were all confirmed by TC39 and will be fixed in the next release. After fixing them, the formal semantics passed all 18,064 applicable tests.
- JISET **is also *forward compatible* with new language features proposed for future ECMAScript specifications.** We evaluated the forward compatibility of JISET by applying it to all nine proposals that are ready for inclusion in the next ECMAScript (ES11, 2020). It automatically synthesized parsers and compiled 560 out of 595 algorithm steps for all the proposals. After completion of the missing parts, we found three specifications errors in BigInt proposal by executing the corresponding tests in Test262. After fixing them, the extracted semantics passed all applicable ES10 tests and 303 new applicable tests.

2 OVERVIEW

In this section, we introduce the overall structure of JISET depicted in Figure 2. Compared to the existing approaches shown in Figure 1, our tool automatically synthesizes JavaScript Parser and AST-IR$_{\text{ES}}$ Translator directly from ECMAScript. The motivation of this work is twofold: 1) ECMAScript is written in a well-organized style, and 2) the writing style is converged since ES7 in 2016. We explain how JISET utilizes such common patterns in the writing style to

ArrayLiteral$_{\texttt{[Yield, Await]}}$ **:**

[*Elision*$_{\texttt{opt}}$ **]**

[*ElementList*$_{\texttt{[?Yield, ?Await]}}$ **]**

[*ElementList*$_{\texttt{[?Yield, ?Await]}}$ **,** *Elision*$_{\texttt{opt}}$ **]**

(a) *ArrayLiteral* production in ES10

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

(b) Generated parser for the *ArrayLiteral* production

Figure 3: *ArrayLiteral* production in ES10 and its parser

synthesize JavaScript Parser and AST-IR$_{\text{ES}}$ Translator using the syntax and semantics in JSON format extracted from ECMAScript by Spec Extractor.

Syntax. ECMAScript provides the lexical and syntactic grammars in Appendix A using a variant of EBNF for ECMAScript. We dub it BNF$_{\text{ES}}$ and formally define it in Section 3. Our Spec Extractor reads the grammars written in BNF$_{\text{ES}}$ and converts them into JSON files. For example, Figure 3(a) shows the *ArrayLiteral* production in ES10. It takes two boolean parameters Yield and Await and has three alternatives. The first alternative consists of three symbols: two terminal symbols `[` and `]`, and one non-terminal symbol *Elision*$_{\text{opt}}$. The opt subscript denotes that it is optional. In the second and third alternatives, *ElementList*$_{\text{[?Yield, ?Await]}}$ denotes a parametric non-terminal symbol *ElementList* with the parameters Yield and Await of *ArrayLiteral* as its two arguments. The prefix ? of a symbol denotes that the symbol is passed as an argument.

To generate JavaScript Parser from a given BNF$_{\text{ES}}$ grammar, we construct Parser Generator in Scala. It synthesizes a JavaScript parser according to the given BNF$_{\text{ES}}$, and the generated parser is defined with Scala parser combinators [12]. Moreover, in order to parse BNF$_{\text{ES}}$ grammars correctly and efficiently, we propose *lookahead parsers*, which keep track of lookaheads, sets of possible next tokens. With lookahead parsing, generated parsers now have one-to-one mapping to their corresponding grammar productions, improving readability. For example, Figure 3(b) shows the generated parser for the *ArrayLiteral* production in Figure 3(a). Each parser has the `List[Boolean] => LAParser[T]` type because each production in BNF$_{\text{ES}}$ is parametric with boolean values. The `memo` is a memoization function for pairs of boolean parameters and resulting parsers for performance optimization. The value `ArrayLiteral` corresponds to the *ArrayLiteral* production. In the parser, each string literal such as `"["` or `"]"` denotes a parser for a terminal symbol. The `opt` helper function creates optional parsers. The parametric non-terminal *ElementList* with arguments Yield and Await is represented as a function call `ElementList(Yield, Await)`. The ~ operator combines two parsers and the `^^` operator describes how to construct ASTs. When the left-hand side of `^^` is matched, its right-hand side shows a corresponding AST constructor, where the name of each constructor has a number denoting the order among alternatives. For example, the `ArrayLiteral0` constructor corresponds to the first alternative of the *ArrayLiteral* production.

ArrayLiteral **:** **[** *ElementList* **,** *Elision*$_{\texttt{opt}}$ **]**

1. Let *array* be ! ArrayCreate(0).
2. Let *len* be the result of performing ArrayAccumulation for *ElementList* with arguments *array* and 0.
3. ReturnIfAbrupt(*len*).
4. Let *padding* be the ElisionWidth of *Elision*; if *Elision* is not present, use the numeric value zero.
5. Perform Set(*array*, `"length"`, ToUint32(*padding* + *len*), **false**).
6. NOTE: The above Set cannot fail because of the nature of the object returned by ArrayCreate.
7. Return *array*.

(a) Evaluation abstract algorithm for the third alternative

```
ArrayLiteral[2].Evaluation (ElementList, Elision) => {
  let array = ! (ArrayCreate 0)
  let len = (ElementList.ArrayAccumulation array 0)
  ? len
  if (= Elision absent) let padding = 0
  else let padding = Elision.ElisionWidth
  (Set array "length" (ToUint32 (+ padding len)) false)
  return array
}
```

(b) The generated IR$_{\text{ES}}$ function

Figure 4: Evaluation abstract algorithm for the third alternative of *ArrayLiteral* in ES10 and its generated IR$_{\text{ES}}$ function

Semantics. ECMAScript describes the language semantics as abstract algorithms in English. While they are written in a natural language, the writing style is well-organized with ordered steps and tagged tokens. Spec Extractor reads abstract algorithms with HTML tags and converts them into JSON files. For example, Figure 4(a) presents the **Evaluation** abstract algorithm of the third alternative of the *ArrayLiteral* production in ES10, and it has seven steps. In the HTML files describing the abstract algorithms, each non-terminal symbol (e.g. *ElementList*), local variable (e.g. *array*), code (e.g. `"length"`), or value (e.g. **false**) has the `<nt>`, `<var>`, `<code>`, or `<emu-val>` tag, respectively.

To translate such abstract algorithms into representations suitable for manipulation, we define IR$_{\text{ES}}$, a specialized intermediate representation for ECMAScript. Then, we develop Algorithm Compiler in Scala using Scala parser combinators again to convert given abstract algorithms to IR$_{\text{ES}}$ functions. It also takes Compile Rules as another input, which has two parts: parsing rules and conversion rules. They are *manually* specified for ECMAScript; we explain them in detail in Section 4. Thus, each abstract algorithm is converted into a function written in IR$_{\text{ES}}$ via Algorithm Compiler. For example, Figure 4(b) presents the generated IR$_{\text{ES}}$ function for the **Evaluation** abstract algorithm shown in Figure 4(a). The `ArrayLiteral[2].Evaluation` function takes two parameters for two non-terminal symbols: `ElementList` and `Elision`. The parameter `Elision` has a special value `absent` when the non-terminal symbol *Elision*$_{\text{opt}}$ is not present. Thus, we convert the condition in step 4, "if *Elision* is not present," into the equality check with `absent`: `if (= Elision absent)`. Codes are represented as string values and values are represented as corresponding IR$_{\text{ES}}$ values. For instance, the code `"length"` and the value **false** are converted into the string value `"length"` and the boolean value `false`, respectively.

Finally, JISET constructs AST-IR_{ES} Translator with the given IR_{ES} functions and *manually* specified Global Setting, which has minor but necessary information to evaluate JavaScript programs described in ECMAScript such as the structure of the standard built-in objects and ECMAScript data types. Putting them all together, we can translate a given JavaScript program into IR_{ES} via generated JavaScript Parser and AST-IR_{ES} Translator by JISET. Even though JISET is not fully automatic because of Compile Rules and Global Setting, it could dramatically reduce the efforts to building parsers and translators from scratch.

In the remainder of this paper, we explain the details of how to automatically generate parsers (Section 3) and how to compile abstract algorithms (Section 4). After evaluating JISET (Section 5), we discuss related work (Section 6), and conclude (Section 7).

3 PARSER GENERATOR

In this section, we explain how to automatically generate JavaScript parsers from a given ECMAScript.

3.1 BNF_{ES}: Grammar for ECMAScript

ECMAScript describes the JavaScript syntax using a variant of the extended BNF. We formally define the notation and dub it BNF_{ES}. It consists of a number of *productions* with the following form:

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

The left-hand side of ::= represents a parametric non-terminal A with multiple boolean parameters $p_1, \cdots, p_k$. If a non-terminal takes no parameter, parentheses are omitted for brevity. A production has multiple alternatives separated by | with optional conditions. A condition c is either a boolean parameter p or its negation $!p$. An alternative α is a sequence of symbols, where a symbol s is one of the following:

- ϵ: the empty sequence, which passes without any conditions
- $\texttt{a}$: a terminal, which is any token
- $A(a_1, \cdots, a_k)$: a non-terminal, which takes multiple arguments where each argument a_i is either a boolean value `#t` or `#f`, or a parameter p_i
- s?: option, which is the same with $s \mid \epsilon$
- $+s$ $(-s)$: positive (negative) lookahead, which checks whether s succeeds (fails) and *never consumes any input*
- $s \setminus s'$: exclusion, which first checks whether s succeeds and then checks whether the parsing result does not correspond to s'
- $\langle\neg\texttt{LT}\rangle$: no line-terminator, which is a special symbol that restricts the white spaces between two different symbols

For example, consider the following production:

$$A(p) ::= p \Rightarrow \texttt{a} \mid !p \Rightarrow \texttt{b} \mid \texttt{c}$$

Then, $A(\texttt{\#t})$ means $\texttt{a} \mid \texttt{c}$ and $A(\texttt{\#f})$ means $\texttt{b} \mid \texttt{c}$.

3.2 Lookahead Parsing

To support BNF_{ES} correctly, we extend PEG-based parser generation techniques with lookahead parsing.

Background: Parsing Expression Grammar. Most parser generators target context-free languages with specific parsing algorithms for Context-Free Grammar (CFG): JavaCC with LL(k) [13], Bison with GLR [30], and ANTLR with ALL(*) [26]. However, they are not directly applicable for the ECMAScript syntax because ECMAScript lexical and syntactic grammars require context-sensitive lexers and parsers:

- **Context-sensitive tokens:** ECMAScript tokens are context-sensitive because of JavaScript regular expressions and template strings. For example, `/x/g` could be a single regular expression token or four tokens that represent division by variables `x` and `g` depending on enclosing contexts. Thus, lexers should be evaluated during parsing not before parsing.
- **Context-sensitive BNF_{ES} symbols:** BNF_{ES} supports context-sensitive symbols, which are positive (negative) lookahead $+s$ $(-s)$, exclusion $s \setminus s'$, and no line-terminator $\langle\neg\texttt{LT}\rangle$. They are highly expressive and they can even represent the classic non-context-free language $\{a^n b^n c^n : n \geq 1\}$ with the following productions:

$$\begin{array}{ll} S ::= +(X\ \texttt{c})\ A\ Y & X ::= \texttt{a}\ X?\ \texttt{b} \\ A ::= \texttt{a}\ A? & Y ::= \texttt{b}\ Y?\ \texttt{c} \end{array}$$

However, it is not trivial to support such BNF_{ES} symbols in CFG-based parser generators.

Unlike CFG-based parser generators, parser generators based on *Parsing Expression Grammar (PEG)* [16] can easily resolve these problems. PEGs are defined with a top-down (LL-style) recursive descent parser with *backtracking*. It visits each alternative of a production in order and backtracks to its previous production when parsing fails. PEG-based parser generators treat lexers as parsers, thus we can use appropriate lexers depending on parsing contexts. Moreover, PEGs support *and-predicate* (&) and *not-predicate* (!) operators that denote the same meaning of the positive and negative lookahead symbols in BNF_{ES}, respectively. Therefore, we can easily support context-sensitive tokens and BNF_{ES} symbols in PEG-based parser generators.

Problem: Prioritized Choices. While PEG-based parser generators support the context-sensitivity, PEGs have one fundamental difference with BNF_{ES}: *prioritized choices*. PEGs use the prioritized choice operator '/' instead of the unordered pipe operator '|' in BNF_{ES}; even when multiple alternatives are applicable, PEGs always pick the first successful alternative. For example, consider the following BNF_{ES}:

$$\begin{array}{l} S ::= E + E \\ E ::= \texttt{x} \mid \texttt{x.p} \end{array} \tag{1}$$

As expected, this grammar accepts the string `x+x.p`. However, the following PEG:

$$\begin{array}{l} S ::= E + E \\ E ::= \texttt{x} \;/\; \texttt{x.p} \end{array} \tag{2}$$

does not accept the same string `x+x.p`. Because the first alternative `x` of E is chosen whenever an input string starts with `x`, the second alternative `x.p` of E is always unreachable. A simple solution to accept the string is just to change the order of alternatives of E like $E ::= \texttt{x.p} \;/\; \texttt{x}$.

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

Figure 5: Over-approximated first tokens of BNF$_{\text{ES}}$ symbols

Unfortunately, simple reordering is not a general solution for all cases. Consider the following BNF$_{\text{ES}}$:

$$
\begin{aligned}
S &::= A\ \mathsf{b}\\
A &::= \mathsf{a} \mid \mathsf{ab}
\end{aligned}
\tag{3}
$$

It accepts both strings `ab` and `abb`. However, the following PEG:

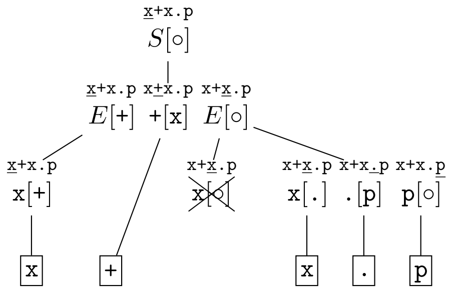
$$
\begin{aligned}
S &::= A\ \mathsf{b}\\
A &::= \mathsf{a} \ / \ \mathsf{ab}
\end{aligned}
\tag{4}
$$

accepts only `ab`, and another PEG with reordered productions as follows:

$$
\begin{aligned}
S &::= A\ \mathsf{b}\\
A &::= \mathsf{ab} \ / \ \mathsf{a}
\end{aligned}
\tag{5}
$$

accepts only `abb`.

Solution: Lookahead Tokens. To alleviate the problem, we propose *lookahead parsing*, which is an extended parsing algorithm for PEGs with *lookahead tokens*. The key idea of lookahead parsing is to keep track of the next possible tokens by statically calculating a set of first tokens for each symbol using the algorithm in Figure 5. For example, the following steps explain how to utilize lookahead tokens during parsing of the string `x+x.p` with the PEG in Equation (1):



Each node $s[L]$ denotes a symbol s with a set of lookahead tokens L. The underlined character in the string of each node denotes the current position in the parsing process that follows a pre-order traversal. The parser starts from the starting non-terminal S with the special lookahead ∘, which denotes the end of inputs. Then, it visits the first alternative $E + E$ with the same lookahead ∘. Each symbol is visited with its corresponding lookahead, which is the first tokens of the right next symbol. For example, for the second symbol `+` in $E + E$, the next symbol is E and its first tokens are:

$$
\begin{aligned}
\mathbf{first}_s(E) &= \mathbf{first}_\alpha(\mathsf{x}) \cup \mathbf{first}_\alpha(\mathsf{x.p})\\
&= \mathbf{first}_s(\mathsf{x}) \cup (\mathbf{first}_s(\mathsf{x}) :+ \mathbf{first}_\alpha(\mathsf{.p})) = \{\mathsf{x}\}
\end{aligned}
$$

Thus, the parser visits `+` with the lookahead `x`. The most important point here is the difference between two visits of the non-terminal E in $E + E$. The first visit of E has the lookahead `+` and the actual next character after matching `x` is also `+`. Thus, the first alternative `x` of E is chosen for the first visit. However, in the second visit of E, the lookahead is the end of inputs ∘ but the next character after matching `x` is the dot character (`.`) instead of the end of inputs. Therefore, the second alternative `x.p` is chosen in the second visit and the parser now successfully parses the input `x+x.p`.

We formally define the semantics of lookahead parsers in Figure 6. The helper function $\mathbf{get}_s(L)$ generates a parser by combining all tokens in the lookahead L using prioritized choices. In this case, the order does not change the semantics of lookahead parsers because $\mathbf{get}_s(L)$ just checks the existence of a given token.

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

Figure 6: Formal semantics of lookahead parsers

3.3 Implementation

We implemented the lookahead parsing technique by extending the Scala parser combinators library, which is a Scala library for PEG-based parser generation. We developed Parser Generator to synthesize PEG-based parsers with lookahead parsing for BNF$_{\text{ES}}$.

AST Generation. Parser Generator first automatically synthesizes ASTs as Scala classes from a given BNF$_{\text{ES}}$ grammar. Because the structure of lexical productions do not affect the ECMAScript semantics, we represent lexical non-terminals as string values. For each syntactic production $A(p_1, \cdots, p_k)\ ::=\ (c_1\ \Rightarrow)^{?}\alpha_1\ \mid\ \cdots\ \mid\ (c_n\ \Rightarrow)^{?}\alpha_n$, the generator synthesizes a trait `A` and its multiple subclasses A_i for $0 \le i \le n - 1$ that represent its alternatives. Each class A_i has non-terminals in its corresponding alternative as its fields. For instance, the *ArrayLiteral* production in Figure 3 gets automatically translated to the following Scala classes:

```
trait ArrayLiteral extends AST
case class ArrayLiteral0(x1: Option[Elision])
case class ArrayLiteral1(x1: ElementList)
case class ArrayLiteral2(x1: ElementList, x3: Option[Elision])
```

Parser Generation. The next step is to automatically synthesizes parsers for each production in BNF$_{\text{ES}}$. We extended Scala parser combinators to support lookahead parsing and BNF$_{\text{ES}}$ notations. For example, the synthesized parser from the production

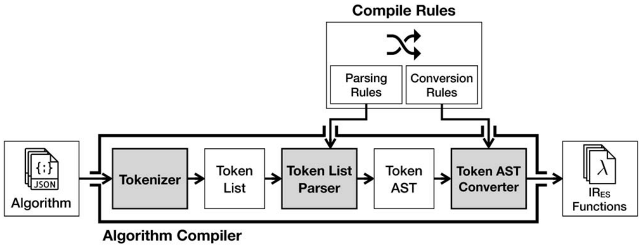


Figure 7: Overall structure of Algorithm Compiler

ArrayLiteral in Figure 3(a) is the one in Figure 3(b). A naïve implementation of lookahead parsing would take exponential time because of backtracking. To reduce it to linear time, we applied the memoization technique introduced in Packrat parsing [15]. Moreover, we also implemented the *growing the seed* technique presented by Warth et al. [32] to support direct and even indirect left recursive productions. It enables the synthesis of parsers without changing the structure of each production in BNF_{ES}.

The synthesized parsers also support the automatic semicolon insertion algorithm, which is one of the most distinctive parsing features in ECMAScript. We extended our parsing algorithm to keep track of the right-most position that fails to be parsed in a given input. In ECMAScript, the token at that position is defined as an *offending token* and the automatic semicolon insertion algorithm is defined with such tokens. The algorithm is simple when we already have the positions of offending tokens. Thus, we just manually supported them by following the rules defined in Section 11.9[4] in ES10. The automatic semicolon insertion rules rarely change; since ES5.1 written in 2011, only one sub-rule was added.

4 ALGORITHM COMPILER 算法编译程序

In this section, we explain Algorithm Compiler that compiles abstract algorithms to IR_{ES} functions as illustrated in Figure 7.

4.1 Tokenizer

Before compiling abstract algorithms, Tokenizer first tokenizes each abstract algorithm into a list of tagged tokens. An algorithm consists of ordered steps, and a step may contain sub-steps as well. For example, the **Evaluation** abstract algorithm in Figure 4(a) has seven steps. Moreover, the tokens of each step have their own HTML tags and each tag has a meaning. We keep such HTML tag information for each token to construct more precise Compile Rules. If an HTML element is just a text without any explicit tags, it is divided into multiple tokens and each token becomes a sequence of alphanumeric characters or a single non-alphanumeric character. For example, in the **Evaluation** algorithm, **`"length"`** is a single token with the HTML tag `<code>` and `Perform Set(` is divided into three text tokens `Perform`, `Set`, and `(`.

Moreover, Tokenizer flattens a structured step to a single token list to handle multi-step statements easily. Some statements in abstract algorithms consist of multiple steps. For example, the `if-then-else` statement often consists of two steps: one for the `then`-branch and the other for the `else`-branch. To treat them as a linear structure, we introduce three special tokens to break down structured algorithms: ↓ denotes the end of a single step, and ↘ and ↙ denote the start and the end of nested steps, respectively. For example, the following left abstract algorithm is tokenized to the right token list.

```
1. A
2. B        ⟹    A ↓ B ↘ C ↓↙↓
  a. C
```

After tokenizing abstract algorithms, Algorithm Compiler compiles token lists into IR_{ES} functions using Token List Parser and Token AST Converter. They depend on Compile Rules and each compile rule consists of a *parsing rule* and a *conversion rule*:

```
val CompileRule = ParsingRule ^^ ConversionRule
```

For each compile rule, its parsing rule describes how to parse a given token list into a structured token AST, and its conversion rule describes how to convert the given token AST structure into an IR_{ES} component. Now, we explain the token list parser and token AST converter with parsing rules and conversion rules, respectively.

4.2 Token List Parser

The token list parser is defined with *parsing rules*. A parsing rule is a basic parsing rule or a composition of multiple parsing rules. The composition `A | B` of two parsing rules `A` and `B` parses an input using both rules and collects the longest matched results. If both rules fail or match the same length of the input, the composition fails.

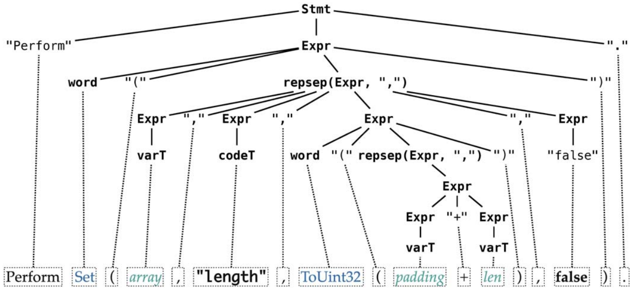
We provide two kinds of basic parsing rules: *tag-based rules* and *content-based rules*. A tag-based rule just checks whether the next token has a given tag. For example, the tag-based parser `varT` and `codeT` check whether the next token has the tag `<var>` and `<code>`, respectively. A content-based parser checks whether the next token is a text token and its content passes a given condition. For example, the string literal `"Perform"` denotes a content-based parser that checks whether the next token is a text token with the content `Perform`. We also define two content-based parsers `word` and `number` that check whether the content of the next token consists of only alphabets or numbers, respectively. In addition, we provide several helper functions such as the optional rule `A?` and the positive(negative) predicate `+A`(`-A`). For instance, the helper function `repsep(A, B)` generates a new parsing rule that denotes zero or more repetition of the parsing rule `A` using another parsing rule `B` as a separator.

Consider the following example parsing rule for the step 5 of the **Evaluation** algorithm in Figure 4(a).

```
// statements
val Stmt = "Perform" ~ Expr ~ "." ^^ ...
// expressions
val Expr =
  // codes       // false literal
  codeT    ^^ ... |  "false"           ^^ ... |
  // variables       // additions
  varT     ^^ ... |  Expr ~ "+" ~ Expr ^^ ... |
  // function calls
  word ~ "(" ~ repsep(Expr, ",") ~ ")" ^^ ...
```

[4] https://www.ecma-international.org/ecma-262/#sec-automatic-semicolon-insertion

We omit the conversion rule for each compile rule for brevity. The `Stmt` compile rule describes how to compile statements with a single parsing rule, and the `Expr` compile rule describes how to compile expressions with five parsing rules. A token parser with the above rules parses the step 5 of **Evaluation** to the following token AST:



4.3 Token AST Converter

Token-AST转换器

Conversion rules describe how to generate an IR$_{ES}$ function for a given token AST. Each conversion rule is defined with its corresponding parsing rule. For basic parsing rules, their conversion rules always return the string values of the contents in parsed tokens. For example, the following conversion rules are the omitted parts in the previous example for the step 5 of **Evaluation**:

```
// statements
val Stmt = ... ^^ { case _ ~ e ~ _ => IExpr(e) }
// expressions
val Expr =
  // codes       // false literal
  ... ^^ EStr | ... ^^ { _ => EBool(false) }           |
  // variables  // additions
  ... ^^ EId  | ... ^^ { case x ~ _ ~ y => EAdd(x, y) } |
  // function calls
  ... ^^ { case x ~ _ ~ y ~ _ => ECall(x, y) }
)
```

The conversion rule of the `Stmt` compile rule uses only the second sub tree and constructs an `IExpr` IR$_{ES}$ instruction. For the second sub-tree, the conversion rule of the fifth `Expr` compile rule is applied. It constructs `ECall` IR$_{ES}$ expression with the string value of the first sub-tree and the sequence of the expressions of the third sub-tree. In this way, the step 5 of **Evaluation** is converted to the following IR$_{ES}$ instruction whose beautified form is the seventh line in Figure 4(b).

```
IExpr(ECall(EId("Set"), List(
  EId(array), EStr("length"), ECall(EId(ToUint32), List(
    EAdd(EId("padding"), EId("len")))), EBool(false))))
```

We define IR$_{ES}$ to represent abstract algorithms as its functions with the following design choices:

- **Dynamic typing:** Because each variable in abstract algorithms is not statically typed, variables do not have their own static types while each value of IR$_{ES}$ has its dynamic type.
- **Imperative style:** IR$_{ES}$ represents algorithm steps as imperative instructions in the sense that each instruction changes the current state consisting of an environment and a heap.

Table 1: General compile rules for ECMAScript

Name	`Stmt`	`Expr`	`Cond`	`Value`	`Ty`	`Ref`
# Rules	21	27	16	11	34	9

- **Higher-order functions with restricted scopes:** In each function of IR$_{ES}$, only global variables, parameters, and its local variables are available, which means that a function closure does not capture its current environment. We use such restricted scopes because they are enough to represent abstract algorithms.
- **Primitive values:** IR$_{ES}$ supports ECMAScript primitive values except "symbols" because symbols can be represented as singleton objects. Also, IR$_{ES}$ provides the unique `absent` value to represent the absence of parameters. For example, when the optional second parameter *Elision* of **Evaluation** in Figure 4(a) is absent, the parameter has the `absent` value.
- **Abstract data types:** IR$_{ES}$ supports only three abstract data types: `Record` for mappings from values to values, `List` for sequential data, and `Symbol` for singleton data. For example, ECMAScript environment records are represented as `Record` from string values to addresses that represent the bindings of the string values.

We define the syntax of IR$_{ES}$ that has 15 kinds of instructions and 26 kinds of expressions with the notation i and e, respectively. We also formally define its operational semantics $\sigma \vdash i \Rightarrow \sigma$ for instructions and $\sigma \vdash e \Rightarrow (v, \sigma)$ for expressions, where σ denotes a state and v denotes a value. For presentation brevity, we omit the formalization of IR$_{ES}$ in this paper and include it in a companion report [14].

4.4 Implementation

We implemented Algorithm Compiler by extending the Packrat parsing [15] library in Scala parser combinators. We modified the meaning of the composition operator (|) to collect all the longest matched results. If a parser detects a step that cannot be parsed or is parsed in multiple ways, it reports the step with parsing results.

Compile Rules. Algorithm Compiler requires compile rules to compile given abstract algorithms to IR$_{ES}$ functions. As already explained in Section 2, we found common patterns in the writing style of abstract algorithms. We manually defined general compile rules to represent such a writing style with six different kinds as summarized in Table 1. The compile rule for statements, `Stmt`, generates IR$_{ES}$ instructions. The `Expr`, `Cond`, and `Value` compile rules generate IR$_{ES}$ expressions, but they represent different contexts in ECMAScript; `Expr` represents a context where any expression can appear, `Cond` denotes a context where any boolean-valued expression can appear, and `Value` represents a context where a fully evaluated value can appear. The `Ty` compile rule denotes type names and generates string primitives used in object constructions. The `Ref` compile rule represents references such as identifier lookup and member accesses of objects, and it generates IR$_{ES}$ references.

Global Setting. AST-IR$_{ES}$ Translator uses global settings consisting of *ECMAScript data types* and *built-in objects*. Unlike compile rules, global settings depend on ECMAScript versions. In this paper, we construct global settings only for the latest ECMAScript, ES10.

Table 2: Syntax coverage: Number of productions in each specification and in each update between adjacent versions, from *all* of which JISET automatically generated parsers

Version	ES7	ES8	ES9	ES10	Average
# Lexical productions	78	78	78	81	78.75
# Syntactic productions	157	167	167	174	166.25

Old version	ES7	ES8	ES9	Average
New version	ES8	ES9	ES10	
Δ # Lexical productions	3	5	6	4.67
Δ # Syntactic productions	140	15	8	54.33

ECMAScript describes data types with some fields and methods. While the methods are like abstract algorithms, their semantics are slightly different from abstract algorithms. They implicitly get their receiver objects as arguments at callsites. To mimic such an implicit behavior, we added a special variable `this` as the first parameter of each method, and passed a receiver object at its callsite by modifying Algorithm Compiler. For example, an Environment Record type has the **DeleteBinding (N)** method. Thus, its corresponding IR_{ES} function has two parameters, the special parameter `this` and a normal parameter `N`, and the method call *DclRec*.DeleteBinding(*N*) in an abstract algorithm is compiled to the IR_{ES} instruction: `(DclRec.DeleteBinding DclRec N)`.

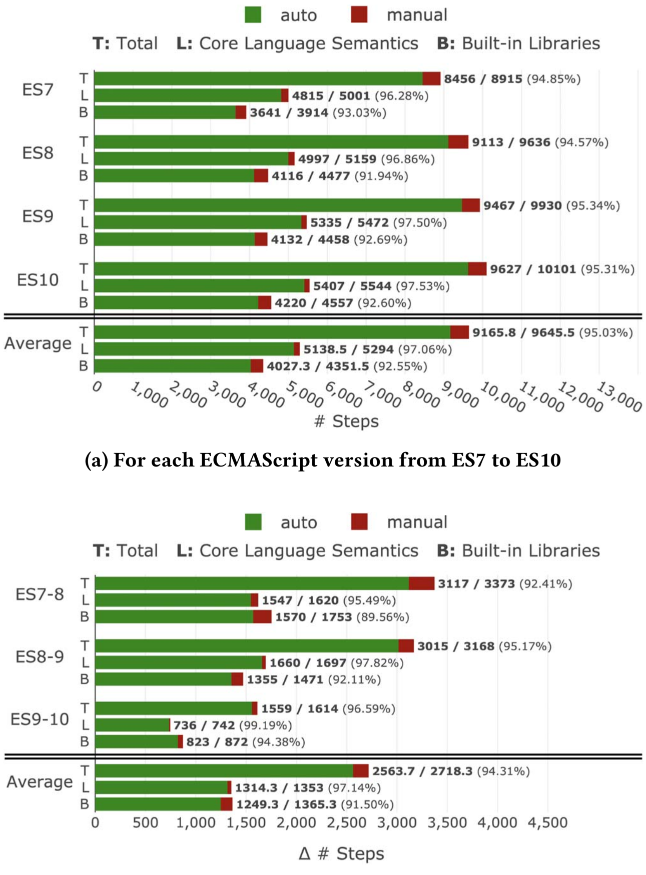
In ECMAScript, built-in objects are pre-defined functions with several built-in functions. For example, `Array` is the constructor of array objects, and its prototype `Array.prototype` has built-in functions for array objects. For instance, `[1,2,3].flat()` calls the `Array.prototype.flat` built-in function with the array `[1,2,3]`. Because built-in functions are also abstract algorithms, each of them is automatically converted to an IR_{ES} function. However, the structures of built-in objects should be manually implemented. Thus, we implemented built-in objects in Scala and connected their properties with the extracted IR_{ES} functions. Some built-in objects that are explicitly referenced in abstract algorithms are intrinsic objects, which have their own aliased names summarized in Table 7[5] of Section 6.1.7.4 Well-Known Intrinsic Objects in ES10. We extracted the alias into Global Setting to utilize it during evaluation.

5 EVALUATION

We developed JISET as an open-source tool[6], and evaluated the tool based on the following research questions:

- RQ1. **Coverage:** How much percentage of the syntax and semantics does JISET automatically extract from ES7 to ES10?
- RQ2. **Correctness:** Does JISET extract an IR-based formal semantics from ECMAScript correctly?
- RQ3. **Forward Compatibility:** Is JISET applicable to language features ready for inclusion in the next ECMAScript (ES11)?

We performed our experiments on a machine equipped with 4.2GHz Quad-Core Intel Core i7 and 64GB of RAM. On the machine, JISET took less than one minute to extract IR-based semantics from a given ECMAScript. 速度比较快



(a) For each ECMAScript version from ES7 to ES10

(b) For each update between adjacent versions

Figure 8: Semantics coverage: Number of algorithm steps in specifications, from which JISET generated the semantics

5.1 Coverage

We evaluated the coverage of JISET in two respects: syntax and semantics. For syntax, we measured how many grammar productions in specifications JISET automatically generated parsers from, and for semantics, we measured how many abstract algorithm steps in specifications it automatically generated the JavaScript semantics from. Because JISET utilizes common patterns in the converged writing style since ES7 as we discussed in Section 2, we evaluated its coverage using the most recent four versions of ECMAScript, ES7 to ES10. We measured the numbers for each ECMAScript version and for each update between adjacent versions.

For syntax, JISET automatically generated parsers for *all* the lexical and syntactic productions. As Table 2 shows, the average numbers of lexical and syntactic productions are 78.75 and 166.25, respectively. Also, the average numbers of annually updated lexical and syntactic productions between adjacent versions are 4.67 and 54.33, respectively.

For semantics, Figure 8 shows that JISET automatically compiled algorithm steps to corresponding IR_{ES} instructions with the success rate of 95.03% on average for each ECMAScript version from ES7 to ES10, and 94.31% for each update between adjacent versions.

[5] https://www.ecma-international.org/ecma-262/10.0/#sec-well-known-intrinsic-objects

[6] https://github.com/kaist-plrg/jiset

Table 3: Specification errors in ES10 and the BigInt proposal ready for inclusion in ES11

Name	Feature	Description	Known	Created	Resolved	Existed	# Fails
ES10-1	Iteration	Missing the `async-iterate` case in the assertion of **ForIn/OfHeadEvaluation**	X	2018-02-16	2020-03-25	768 days	1,116
ES10-2	Condition	Ambiguous grammar production for the dangling `else` problem in *IfStatement*	X	2015-06-01	TBD	TBD	1
ES10-3	String	Wrong use of the = operator in **StringGetOwnProperty**	X	2015-06-01	2020-05-07	1,802 days	7
ES10-4	Completion	Unhandling abrupt completion in **Abstract Equality Comparison**	X	2015-06-01	2020-04-28	1,793 days	9
ES10-5	Completion	Unhandling abrupt completion in **Evaluation** of *EqualityExpression*	O	2015-06-01	2019-05-02	1,431 days	2
ES10-6	Await	Passing a value of wrong type to the second parameter of **PromiseResolve**	O	2019-02-27	2019-04-13	45 days	1,294
ES10-7	Function	No semantics of **IsFunctionDefinition** for `function(...){...}`	O	2015-10-30	2020-01-18	1,541 days	306
ES10-8	Function	No semantics of **ExpectedArgumentCount** for the base case of *FormalParameters*	O	2016-11-02	2020-02-20	1,205 days	81
ES10-9	Iteration	Two semantics of **VarScopedDeclarations** for `for await(var x of e){...}`	O	2018-02-16	2019-10-11	602 days	0
BigInt-1	Expression	Using the wrong variable `oldvalue` instead of `oldValue` in **Evaluation** of *UpdateExpression*	X	2019-09-27	2020-04-23	209 days	533
BigInt-2	Number	Using **ToInt32** instead of **ToUint32** in **Number::unsignedRightShift**	X	2019-09-27	2020-04-23	209 days	2
BigInt-3	Number	Unhandling BigInt values in the **Number** constructor	O	2019-09-27	2019-11-19	53 days	1

Table 4: Test results for Test262

All Test262 Tests	**35,990**
Annexes	1,060
Internationalization	640
In-progress features	5,338
ES10 Tests	**28,952**
Non-strict mode	1,150
Modules	918
Early errors before actual execution	2,288
Inessential built-in objects	6,532
Applicable Tests	**18,064**
Passed tests	16,355
Failed tests	1,709

ECMAScript abstract algorithms describe not only core language semantics but also built-in libraries with various helper functions. Note that built-in libraries are written in more diverse styles than core language semantics due to their own specific functionalities. Therefore, built-in libraries have lower success rates (92.55% for specifications and 91.50% for updates) than core language semantics (97.06% for specifications and 97.14% for updates).

Because JISET automatically extracts the syntax and semantics from specifications and updates with high coverage rates, it reduces efforts not only in developing JavaScript tools from scratch from specifications but also in evolving existing tools for updates.

5.2 Correctness

To evaluate the correctness of JISET, we tested the extracted semantics from the latest ECMAScript (ES10) by executing Test262 as of February 28, 2019, when ES10 was branched out. To focus on the core language semantics of JavaScript, we completed only necessary parts missing from the extracted AST-IR translator. As Figure 8(a) shows, for the abstract algorithms in ES10, 9,627 steps out of 10,101 steps are automatically compiled by Algorithm Compiler. It fully covers 1,783 algorithms out of 2,026 abstract algorithms and 243 algorithms are partially covered. Among the remaining 474 steps, we manually implemented all missing steps for the core language semantics (137 steps) and the essential parts of the built-in libraries (140 steps out of 337). Based on this manual implementation, 146 more abstract algorithms are fully covered. We also manually implemented Global Setting as described in Section 4.4 for the core language features. Note that we do not support minor language features such as the non-strict mode, modules, early errors before actual execution, and inessential built-in objects. Among 35,990 tests in Test262, we filtered out 17,926 tests as summarized in Table 4. To focus on ES10, we excluded 7,038 tests for annexes, internationalization, and in-progress features. We also filted out 10,888 tests that use minor language features. Finally, the extracted semantics took about three hours to evaluate 18,064 applicable tests and failed for 1,709 tests.

We investigated the failed tests and found out that they failed due to specification errors in ES10. Using the failed tests, we discovered nine errors: ES10-1 to ES10-9 in Table 3. Among them, five errors (ES10-5 to ES10-9) were previously reported and fixed in the current draft of the next ECMAScript, and the remaining four errors (ES10-1 to ES10-4) were never reported before. All four errors were confirmed by TC39, and will be fixed in the next ECMAScript, ES11.

The specification error ES10-1 is due to a wrong assertion. While ES9 introduced the `for await` iteration statement with a new *iterationKind* tag, `async-iterate`, the `ForIn/OfHeadEvaluation` algorithm missed the `async-iterate` case in an assertion, which caused 1,120 tests failed. We reported the error and proposed a specification fix to include the `async-iterate` case, and TC39 accepted it on March 25, 2020. Because the error was created on February 16, 2018, it existed for 768 days.

ES10-2 comes from the well-known dangling `else` problem introduced in ALGOL 60 [11]. ES10 describes how to parse it in prose: the `else` statement should be associated with the nearest `if` statement.

Table 5: Proposals that will be included in ES11

Proposal	Δ # Prod.		Δ#Steps	Δ#Tests	# Tests
	Lex.	Syn.			
`matchAll` of `String`	0	0	9/9	5/5	18,064/18,064
`import()`	0	2	38/38	0/0	18,064/18,064
`BigInt`	4	0	298/326	196/207	17,539/18,064
`Promise.allSettled`	0	0	79/85	50/50	18,064/18,064
`globalThis`	0	0	1/1	1/1	18,064/18,064
`for-in` mechanics	0	0	36/37	0/0	18,064/18,064
Optional Chaining	3	3	74/74	19/19	18,064/18,064
Nullish Coalescing Op.	1	4	10/10	21/21	18,064/18,064
`import.meta`	0	2	15/15	0/0	18,064/18,064
Total	8	11	560/595		

Because it is written in prose rather than in the ES10 grammar productions, it caused one failed test. We proposed a fix to revise the ambiguous grammar production, and TC39 confirmed it on April 23, 2020.

ES10-3 is due to a misuse of the = operator for numbers. In abstract algorithms, "$x = y$" denotes equality testing for double-precision 64-bit binary format IEEE 754-2008 values; thus, "+0 = -0" evaluates to true. However, to check whether *index* is exactly the same with -0, **StringGetOwnProperty** used "*index* = -0", which is true even when *index* is +0. It caused seven failed tests. We proposed a fix accepted on May 7, 2020. Thus, ES10-3 existed for 1,802 days.

ES10-4 and ES10-5 happened because ES10 did not handle abrupt completion from function calls. Our proposed fix to ES10-4 was accepted on April 28, 2020, and ES10-5 was resolved on May 2, 2019 after existing for 1,431 days.

ES10-6 is due to incorrect uses of an abstract algorithm. While **PromiseResolve(C, x)** expects a JavaScript object for its second argument, ES10 passed a list of values rather than an object in three invocations of **PromiseResolve**. The wrong invocations were introduced on February 27, 2019 and caused 1,294 tests failed. They were fixed on April 13, 2019 after existing 45 days.

ES10-7 and ES10-8 happened because ES10 missed semantics for some cases. They both existed for more than 1,200 days.

ES10-9 is due to multiple semantics. While no tests in Test262 fail with any of the semantics, we could detect this error via Spec Extractor even before executing the semantics. It is supplementary merit of the mechanization of IR-based semantics extraction.

After resolving the nine specification errors in ES10, we extracted a semantics from the revised specification. The extracted semantics from the revised ES10 passed all 18,064 applicable tests in Test262, which shows that JISET extracts an IR-based formal semantics from ECMAScript correctly. In addition, the evaluation witnesses that JISET can detect specification errors effectively. We could detect not only five previously-known errors but also four new errors. We believe that JISET bridges gaps between ECMAScript written in a natural language and executable tests in Test262.

5.3 Forward Compatibility

We evaluated whether JISET is forward compatible by applying it to the proposals ready for inclusion in ES11. Because ECMAScript is an open-source project, various proposals for new features are available with their own specification changes and tests. A separate repository [4] maintains them in six stages: Stage 0 to Stage 3, Finished, and Inactive. A proposal starts with Stage 0, and the TC39 committee examines proposals in Stage 3. If a proposal is confirmed, the committee changes its stage to Finished and integrates it into the next ECMAScript. Otherwise, its stage becomes Inactive.

We applied JISET to all nine Finished proposals as shown in Table 5. Collectively, the proposals modified eight lexical and 11 syntactic productions, and JISET successfully synthesized parsers for them. The synthesized parsers parse all applicable tests for all proposals. For abstract algorithms, 560 steps out of 595 are automatically converted to corresponding IR_{ES} instructions by Algorithm Compiler without changing Compile Rules. Thus, JISET has the success rate of 94.12% on average for forthcoming proposals.

We checked the extracted semantics from the proposals by implemented missing parts of the AST-IR translator for each proposal and checking the semantics with Test262. All of them passed all applicable tests except the semantics from the `BigInt` proposal. It failed for 11 tests out of 207 applicable tests for the proposal and 525 tests out of 18,064 applicable tests for ES10.

Using the failed tests, we discovered three errors in the BigInt proposal: two new errors (BigInt-1 and BigInt-2) and one known error (BigInt-3) as summarized in Table 3. All of them were confirmed by TC39 and will be fixed in ES11. The proposal added two new types: BigInt as a new type of primitives and Numeric as a unified type of the original Number type and the new BigInt type. Therefore, it not only added new algorithms for BigInt but also modified all existing algorithms for Number values. The error BigInt-1 is due to a misuse of the variable `oldValue` in **Evaluation** of *UpdateExpression*. BigInt-2 breaks the backward compatibility because of misusing **ToInt32** instead of **ToUint32** in unsigned right shift operators. BigInt-3 is due to missing BigInt primitives in the Number constructor. On average, three errors existed for 157 days in the proposal.

After fixing the errors in the proposal, we extracted a semantics from the revised specification. The extracted semantics passed all 207 applicable tests for the proposal and 18,064 applicable tests for ES10. Thus, JISET also correctly extracts an IR-based semantics from future proposals, which implies that it is forward compatible.

6 RELATED WORK

Our technique is closely related to three fields: parser generation, automatic semantics extraction, and formal semantics of JavaScript.

Parser Generation: From Packrat parsing [15] with PEG [16], recursive-descent parsers with backtracking support linear-time parsing. However, it has the fundamental problem of ordered choices: *ab* is silently unmatched with *a* / *ab*. While Generalized LL (GLL) parsing [28] is basically recursive-descent with backtracking that can support general context-free grammars even in the presence of ambiguous grammars, its worst-time complexity is $O(n^3)$ for the input size n and it does not support context-sensitive features. Unlike GLL parsing, our lookahead parsing is applicable for JavaScript parsers with context sensitive features such as positive/negative lookaheads. Moreover, the complexity of lookahead parsing is $O(k \cdot n)$ for the constant number of tokens k. We experimentally showed that it can generate parsers for the most recent four versions of ECMAScript.

解析器生成

RD：顺序选择问题没有解决

GLL：（1）不支持上下文敏感的特性；（2）最差时间复杂度太高

最后采用适合上下文敏感特性的前视解析器

自动语义提取

Automatic Semantics Extraction: The closest work to ours is the formal semantics extraction for x86 [22] and ARM [31]. They utilized complex Natural Language Processing (NLP) and Machine Learning (ML) to extract formal semantics of small-sized low-level assembly languages. Another related work is Zhai et al. [33]'s automatic model generation, which generates Java code from Javadoc comments for API functions. Using NLP techniques and heuristic methods, it produces candidate code and removes wrong ones by testing them with actual implementation. Unlike their approach, we introduce a semi-automatic synthesis using general compile rules that represent common writing patterns of specifications. The extracted semantics by JISET is also executable, which allows to bridge gaps between the specification written in a natural language and executable tests. 其他一些定义js形式化语义的方法

Several approaches defined the formal semantics of JavaScript. Guha et al. [18] defined a core calculus of JavaScript expressing non-core features using desugaring, but its correspondence with ECMAScript is not obvious. KJS [25] and JaVerT [17] defined the JavaScript semantics by manually converting ECMAScript to their own formal languages. KJS mapped ES5.1 in the K framework [27] and JaVerT converts the specification to their own IR. However, they all target only ES5.1 or former and they do not provide any solution for annual updates of ECMAScript. Our approach provides a mechanized framework to synthesize JavaScript parsers and to automatically extract semantics using a rule-based compilation technique, which significantly reduces human efforts.

不足

缺点是没有对应最新版的ECMAScript且依赖于人工。本文方法解决了这两个问题。

7 CONCLUSION

Annual updates of ECMAScript make it difficult to build program analysis or formal verification of JavaScript due to the required human efforts in modeling a moving target. In this paper, we proposed JISET, a tool that *automatically* extracts the syntax and semantics as a parser and an AST-IR translator from ECMAScript written in English. The tool automatically extracts all the syntax and 95.03% of the semantics for the most recent four versions of ECMAScript (ES7 to ES10). We evaluated the correctness of the tool by testing the extracted semantics from ES10 with Test262, the official conformance suite. Using 1,709 failed tests, we found nine specification errors, four of which are newly discovered, confirmed by TC39, and planned to be integrated in ES11. After fixing the errors, the extracted semantics passed all 18,064 applicable tests in Test262. We also showed that JISET is forward compatible by applying it to nine proposals to be included in ES11, which let us find three errors in the BigInt proposal. We believe that JISET can dramatically reduce human efforts in building various JavaScript tools correctly.

这边的彩蛋提到了一致性测试

ACKNOWLEDGEMENTS

This work was supported by National Research Foundation of Korea (NRF) (Grants NRF-2017R1A2B3012020 and 2017M3C4A7068177).

REFERENCES

[1] 2011. Standard ECMA-262 5.1 Edition ECMAScript Language Specification. (2011). Retrieved May 8, 2020 from https://ecma-international.org/ecma-262/5.1/

[2] 2015. Standard ECMA-262 6th Edition ECMAScript 2015 Language Specification. (2015). Retrieved May 8, 2020 from https://ecma-international.org/ecma-262/6.0/

[3] 2019. Espruino - JavaScript for Microcontrollers. (2019). Retrieved May 8, 2020 from https://www.espruino.com/

[4] 2019. GitHub repository for ECMAScript proposals. (2019). Retrieved May 8, 2020 from https://github.com/tc39/proposals

[5] 2019. MEAN.JS - Open-Source Full-Stack Solution for MEAN Applications. (2019). Retrieved May 8, 2020 from https://meanjs.org/

[6] 2019. Node.js - A JavaScript runtime built on Chrome's V8 JavaScript engine. (2019). Retrieved May 8, 2020 from https://nodejs.org/

[7] 2019. Standard ECMA-262 10th Edition ECMAScript 2019 Language Specification. (2019). Retrieved May 8, 2020 from https://ecma-international.org/ecma-262/10.0/

[8] 2019. TC39 – ECMAScript. (2019). Retrieved May 8, 2020 from https://www.ecma-international.org/memento/tc39-m.htm

[9] 2019. Tessel 2 is a robust IoT and robotics development platform. (2019). Retrieved May 8, 2020 from https://tessel.io/

[10] 2019. Test262: ECMAScript Test Suite. (2019). Retrieved May 8, 2020 from https://github.com/tc39/test262

[11] Paul W Abrahams. 1966. A final solution to the dangling else of ALGOL 60 and related languages. *Commun. ACM* 9, 9 (1966), 679–682.

[12] Moors Adriann, Piessens Frank, and Martin Odersky. 2008. *Parser Combinators in Scala.* Technical Report. K.U.Leuven, Leuven, Belgium, Technical Report. http://www.cs.kuleuven.be/publicaties/rapporten/cw/CW491.pdf

[13] Alfred V Aho and Jeffrey D Ullman. 1973. The theory of Parsing. *Translation and Compiling* 1, 1972 (1973).

[14] Seungmin An, Jihyeok Park, and Sukyoung Ryu. 2020. *IR_{ES}: Intermediate Representation for ECMAScript Specifications.* Technical Report. https://drive.google.com/file/d/1chmyQullDPkSfoHk5PvO3r0yDHOQiAcj

[15] Bryan Ford. 2002. Packrat Parsing:: Simple, Powerful, Lazy, Linear Time, Functional Pearl. In *In Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02).* ACM, New York, NY, USA, 36–47. https://doi.org/10.1145/581478.581483

[16] Bryan Ford. 2004. Parsing Expression Grammars: A Recognition-based Syntactic Foundation. In *In Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '04).* ACM, New York, NY, USA, 111–122. https://doi.org/10.1145/964001.964011

[17] José Fragoso Santos, Petar Maksimović, Daiva Naudžiūnienė, Thomas Wood, and Philippa Gardner. 2017. JaVerT: JavaScript Verification Toolchain. *Proc. ACM Program. Lang.* 2, POPL, Article 50 (Dec. 2017), 33 pages. https://doi.org/10.1145/3158138

[18] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. 2010. The Essence of Javascript. In *In Proceedings of the 24th European Conference on Object-oriented Programming (ECOOP'10).* Springer-Verlag, Berlin, Heidelberg, 126–150. http://dl.acm.org/citation.cfm?id=1883978.1883988

[19] Simon Holm Jensen, Anders Møller, and Peter Thiemann. 2009. Type Analysis for JavaScript. In *In Proceedings of the International Symposium on Static Analysis.* 238–255.

[20] Vineeth Kashyap, Kyle Dewey, Ethan A. Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann, and Ben Hardekopf. 2014. JSAI: A Static Analysis Platform for JavaScript. In *In Proceedings of the International Symposium on Foundations of Software Engineering.* 121–132.

[21] Sergio Maffeis, John C. Mitchell, and Ankur Taly. 2008. An Operational Semantics for JavaScript. In *In Proceedings of the Asian Symposium on Programming Languages and Systems (APLAS '08).* 307–325.

[22] HLY Nguyen. 2018. Automatic extraction of x86 formal semantics from its natural language description. *Information Science* (2018).

[23] Changhee Park and Sukyoung Ryu. 2015. Scalable and Precise Static Analysis of JavaScript Applications via Loop-Sensitivity. In *In Proceedings of 29th European Conference on Object-Oriented Programming (ECOOP 2015) (Leibniz International Proceedings in Informatics (LIPIcs)),* John Tang Boyland (Ed.), Vol. 37. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 735–756. https://doi.org/10.4230/LIPIcs.ECOOP.2015.735

[24] Daejun Park. 2015. KJS: A Complete Formal Semantics of JavaScript (Slides). (2015). Retrieved May 8, 2020 from https://daejunpark.github.io/2015-06-16-park-stefanescu-rosu-PLDI.pdf

[25] Daejun Park, Andrei Stefănescu, and Grigore Roşu. 2015. KJS: A Complete Formal Semantics of JavaScript. In *In Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15).* ACM, New York, NY, USA, 346–356. https://doi.org/10.1145/2737924.2737991

[26] Terence Parr, Sam Harwell, and Kathleen Fisher. 2014. Adaptive LL(*) parsing: the power of dynamic analysis. In *In Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14).* ACM, 579–598. https://doi.org/10.1145/2660193.2660202

[27] Grigore Roşu and Traian Florin Şerbănuţă. 2010. An overview of the K semantic framework. *The Journal of Logic and Algebraic Programming* 79, 6 (Aug. 2010), 397–434.

[28] Elizabeth Scott and Adrian Johnstone. 2010. GLL Parsing. In *Electronic Notes in Theoretical Computer Science.* https://doi.org/10.1016/j.entcs.2010.08.041

[29] Manu Sridharan, Julian Dolby, Satish Chandra, Max Schäfer, and Frank Tip. 2012. Correlation Tracking for Points-to Analysis of JavaScript. In *In Proceedings of the European Conference on Object-Oriented Programming.*

[30] Masaru Tomita. 1985. An Efficient Context-Free Parsing Algorithm for Natural Languages.. In *IJCAI*, Vol. 2. 756–764.

[31] Anh V Vu and Mizuhito Ogawa. 2019. Formal semantics extraction from natural language specifications for ARM. In *International Symposium on Formal Methods*. Springer, 465–483.

[32] Alessandro Warth, James R. Douglass, and Todd Millstein. 2008. Packrat Parsers Can Support Left Recursion. In *In Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM '08)*. ACM, New York, NY, USA, 103–110. https://doi.org/10.1145/1328408.1328424

[33] Juan Zhai, Jianjun Huang, Shiqing Ma, Xiangyu Zhang, Lin Tan, Jianhua Zhao, and Feng Qin. 2016. Automatic Model Generation from Documentation for Java API Functions. In *In Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 380–391. https://doi.org/10.1145/2884781.2884881