Soundness and Completeness of the Idris2 JSON Parser

Overview

We formalize the parser as a computation on strings and give a rigorous, but still informal (non-mechanized), proof of soundness and completeness with respect to the JSON grammar used by the parser. We assume the correctness of primitive parser operations and treat them as axioms.

Definitions

Parser semantics. A parser for values of type A is a partial function

$$p : \Sigma^* \to \text{Maybe}(A \times \Sigma^*),$$

implemented as $\mathsf{runParser}(p, s)$. If $\mathsf{runParser}(p, s) = \text{Just}(x, r)$ we say p succeeds on input s, consumes a prefix of s, yields x, and leaves remainder r.

Axioms (model assumptions). We assume the following about strings, options, primitive parsers, and the basic combinators. These are treated as axioms rather than proved from any language implementation details.

1. **Strings.** Σ^* is the set of finite strings over an alphabet Σ with standard concatenation.
2. **Options.** Maybe(A) has constructors Just(a) and Nothing.
3. **Primitive parsers.**
   (a) $\mathsf{char}(c)$ succeeds iff the next character is c, consuming exactly one character.
   (b) $\mathsf{stringExact}(t)$ succeeds iff the input starts with t, consuming exactly $|t|$ characters.
   (c) $\mathsf{satisfy}(P)$ succeeds iff the next character satisfies P, consuming exactly one character.
4. **Combinators.**
   (a) **Sequencing.** $\mathsf{runParser}(p;q, s) = \text{Just}(y, r')$ iff there exists x, r such that $\mathsf{runParser}(p, s) = \text{Just}(x, r)$ and $\mathsf{runParser}(q, r) = \text{Just}(y, r')$.
   (b) **Choice.** $\mathsf{runParser}(p < | > q, s) = \text{Just}(x, r)$ iff either $\mathsf{runParser}(p, s) = \text{Just}(x, r)$ or ($\mathsf{runParser}(p, s) = \text{Nothing}$ and $\mathsf{runParser}(q, s) = \text{Just}(x, r)$).
   (c) **Repetition.** $\mathsf{runParser}(\mathsf{many}(p), s) = \text{Just}(xs, r)$ iff $s = s_0 r$ where s_0 is a concatenation of zero or more strings each accepted by p in sequence and $\mathsf{runParser}(p, r) = \text{Nothing}$ (maximal consumption); $\mathsf{some}(p)$ is the non-empty case.
   (d) **Separated lists.** $\mathsf{runParser}(\mathsf{sepBy}(p, \mathsf{sep}), s) = \text{Just}(xs, r)$ iff either $\mathsf{runParser}(p, s) = \text{Nothing}$ and $xs = []$ with $r = s$, or $s = s_0 r$ where s_0 is a p-item followed by zero or more repetitions of sep then p, and $\mathsf{runParser}(\mathsf{sep}\,; p, r) = \text{Nothing}$ (maximal consumption).

We only use many and sepBy with parsers that do not accept the empty string, so the maximality conditions above are well-defined.

5. **Whitespace.** Let $W = \mathrm{WS}^*$ (all whitespace strings). The parser ws always succeeds, consuming the longest prefix $w \in W$ and leaving the remaining suffix.

6. **Lexemes.**

   (a) keyword(t) succeeds iff the input starts with t, and in that case consumes t followed by the longest prefix in W.

   (b) comma succeeds iff the input starts with `","`, and in that case consumes it followed by the longest prefix in W.

   (c) stringLiteral succeeds iff the input starts with stringCore, and in that case consumes it followed by the longest prefix in W, yielding the decoded string.

   (d) numberLiteral succeeds iff the input starts with numberCore, and in that case consumes it followed by the longest prefix in W, yielding the matched numeric lexeme.

7. **String character class.** stringChar recognizes exactly the JSON string character class (unescaped non-control characters or valid escape sequences) and yields the decoded character.

Top-level acceptance.

$$\mathsf{parse}(p, s) = \mathrm{Just}(x) \iff \mathsf{runParser}(p, s) = \mathrm{Just}(x, r) \wedge r \in W$$

Grammar (whitespace-extended). Let $\mathcal{G}$ be the grammar used by the parser. We separate core tokens from lexemes that include trailing whitespace:

value	$\to$	W (object | array | string | number | true | false | null)
object	$\to$	`"{"` W members? `"}"` W
members	$\to$	member (comma member)*
member	$\to$	string `":"` W value
array	$\to$	`"["` W elements? `"]"` W
elements	$\to$	value (comma value)*
string	$\to$	stringCore W
number	$\to$	numberCore W
true	$\to$	`"true"` W
false	$\to$	`"false"` W
null	$\to$	`"null"` W
comma	$\to$	`","` W
stringCore	$\to$	`"` char* `"`
numberCore	$\to$	`"-"`? int frac? exp?
int	$\to$	`"0"` | digit1to9 digit*
frac	$\to$	`"."` digit$^+$
exp	$\to$	(`"e"` | `"E"`) (`"+"` | `"-"`)? digit$^+$
digit	$\to$	`"0"` | `"1"` | `"2"` | `"3"` | `"4"` | `"5"` | `"6"` | `"7"` | `"8"` | `"9"`
digit1to9	$\to$	`"1"` | `"2"` | `"3"` | `"4"` | `"5"` | `"6"` | `"7"` | `"8"` | `"9"`
char	$\to$	any unescaped non-control | escape sequence

Language semantics. For each nonterminal X we define its language $L(X) \subseteq \Sigma^*$ as follows:

1. If $X \to t_1 t_2 \cdots t_n$ is a production, then any concatenation $s = s_1 s_2 \cdots s_n$ with $s_i = t_i$ for terminals and $s_i \in L(t_i)$ for nonterminals belongs to $L(X)$.
2. If $X \to Y \mid Z$, then $L(X) = L(Y) \cup L(Z)$.
3. If a rule contains R^* or R^+, then $L(R^*)$ is the set of all finite concatenations of strings in $L(R)$ (including empty), and $L(R^+)$ is the set of all non-empty such concatenations.
4. If a rule contains $R?$, then $L(R?) = L(R) \cup \{\epsilon\}$.

We treat W as a distinguished nonterminal with language WS^*, as given by the axioms.

Parser specification (algorithm). We reason about the following parser equations, which describe the algorithm independently of any implementation. We omit AST construction because it does not affect input consumption.

1. jsonNull parses keyword("`null`").
2. jsonBool parses keyword("`true`") $<|>$ keyword("`false`").
3. jsonNumber parses numberLiteral.
4. jsonString parses stringLiteral.
5. objectField parses stringLiteral ; char("`:`") ; ws ; jsonValue.
6. jsonArray parses char("`[`") ; ws ; sepBy(jsonValue, comma) ; char("`]`") ; ws.
7. jsonObject parses char("`{`") ; ws ; sepBy(objectField, comma) ; char("`}`") ; ws.
8. jsonValue parses ws ; (jsonNull $<|>$ jsonBool $<|>$ jsonNumber $<|>$ jsonString $<|>$ jsonArray $<|>$ jsonObject).

Soundness

Theorem (Soundness). If parse(jsonValue, s) = Just(v), then $s \in L$(value).

Proof. We prove soundness for the mutually recursive parsers jsonValue, jsonArray, jsonObject, and objectField by mutual induction on their definitions, using the axioms for primitives, lexemes, and combinators.

Lemma S1 (lexeme soundness). By Axioms 5 and 6, ws, keyword, comma, stringLiteral, and numberLiteral consume the longest prefixes in the languages of W, true/false/null, comma, string, and number respectively. By the parser specification, jsonNull, jsonBool, jsonNumber, and jsonString consume the same languages.

Lemma S2 (object field soundness). objectField consumes a string, then "`:`", then W, then a value. By Lemma S1, Axiom 3, and Axiom 4 (sequencing), and the mutual induction hypothesis that jsonValue is sound for value, its consumed prefix is in L(member).

Lemma S3 (array soundness). jsonArray consumes `"["`W, then a sepBy list of value separated by comma, then `"]"`W. By Lemma S1, Axiom 4 (sequencing and sepBy), and the mutual induction hypothesis for jsonValue, its consumed prefix is in $L(\mathsf{array})$.

Lemma S4 (object soundness). jsonObject consumes `"{"`W, then a sepBy list of member separated by comma, then `"}"`W. By Lemmas S1 and S2 and Axiom 4, its consumed prefix is in $L(\mathsf{object})$.

Lemma S5 (value soundness). jsonValue consumes W then chooses among parsers for object, array, string, number, true, false, or null. By Lemmas S1, S3, S4 and Axiom 4 (choice), any success yields a prefix in $L(\mathsf{value})$.

The lemmas are mutually consistent, so by mutual induction the soundness of jsonValue holds. With top-level acceptance, the theorem follows; any residual $r \in W$ can be appended to a string in $L(\mathsf{value})$ without leaving the language because W is closed under concatenation. □

Completeness

Theorem (Completeness). If $s \in L(\mathsf{value})$, then there exists v such that $\mathsf{parse}(\mathsf{jsonValue}, s) = \mathrm{Just}(v)$.

Proof. We prove completeness for the same mutually recursive parsers by mutual induction on the grammar, using the axioms.

Lemma C1 (lexeme completeness). By Axioms 5 and 6, ws, keyword, comma, stringLiteral, and numberLiteral consume the longest prefixes in the languages of W, true/false/null, comma, string, and number respectively. By the parser specification, jsonNull, jsonBool, jsonNumber, and jsonString consume the same languages.

Lemma C2 (object field completeness). If $s \in L(\mathsf{member})$, then by the grammar s is a concatenation of a string, `":"`, W, and a value. By Lemma C1 and Axiom 4 (sequencing), and the mutual induction hypothesis that jsonValue is complete for value, objectField succeeds on s.

Lemma C3 (array completeness). If $s \in L(\mathsf{array})$, then s matches `"["`W, then a possibly empty list of value separated by comma, then `"]"`W. By Lemma C1, Axiom 4 (sepBy), and the mutual induction hypothesis for jsonValue, jsonArray succeeds on s.

Lemma C4 (object completeness). If $s \in L(\mathsf{object})$, then s matches `"{"`W, then a possibly empty list of member separated by comma, then `"}"`W. By Lemmas C1 and C2 and Axiom 4, jsonObject succeeds on s.

Lemma C5 (value completeness). If $s \in L(\mathsf{value})$, then by the grammar s is in W followed by one of the alternatives. By Lemmas C1–C4 and Axiom 4 (choice), jsonValue succeeds on s; left bias does not prevent success because choice succeeds whenever at least one branch succeeds.

The lemmas are mutually consistent, so by mutual induction the completeness of jsonValue holds. Because the grammar for each alternative already includes trailing W and ws consumes the leading whitespace prefix, jsonValue consumes the entire input s when $s \in L(\mathsf{value})$, leaving $r = \epsilon \in W$. With top-level acceptance, the theorem follows. □

Notes

This document presents a rigorous, axiomatic proof in mathematical prose; it is not mechanically verified. A machine-checked proof would require a formalization of these axioms in a proof assistant.