A Formalization of the ABNF Notation and a Verified Parser of ABNF Grammars

Alessandro Coglio

Kestrel Institute, Palo Alto, California, USA
coglio@kestrel.edu
http://www.kestrel.edu/~coglio

Abstract. Augmented Backus-Naur Form (ABNF) is a standardized formal grammar notation used in several Internet syntax specifications. This paper describes (i) a formalization of the syntax and semantics of the ABNF notation and (ii) a verified parser that turns ABNF grammar text into a formal representation usable in declarative specifications of correct parsing of ABNF-specified languages. This work has been developed in the ACL2 theorem prover.

Keywords: ABNF · Parsing · Verification

1 Problem, Contribution, and Outlook

Augmented Backus-Naur Form (ABNF) is a standardized formal grammar notation [9,18] used in several Internet syntax specifications, e.g. HTTP [11], URI [6], and JSON [8]. Since inadequate parsing may enable security exploits such as HTTP request smuggling [19], formally verified parsers of ABNF-specified languages are of interest. It is important to ensure that the formal specifications against which the parsers are verified are faithful to the ABNF grammars.

The work described in this paper contributes to this goal by providing:

1. A formalization of the syntax and semantics of the ABNF notation.
2. A verified parser that turns ABNF grammar text (e.g. the grammar of HTTP) into a formal representation usable in declarative specifications of correct parsing (e.g. correct HTTP parsing).

This work has been developed in the ACL2 theorem prover [14]. The development is available [24, books/kestrel/abnf], is thoroughly documented [25, abnf], and includes examples of use of the parser on several Internet grammars such as HTTP, URI, and JSON. It also includes a collection of operations to compose ABNF grammars and to check properties of them, but this paper does not describe these operations. Some of the excerpts of the development shown in this paper are slightly simplified for brevity.

Future work includes the development of verified parsers for ABNF-specified languages such as JSON and HTTP, and of a generator of verified parsers from ABNF grammars.

2 Background

2.1 ABNF

ABNF adds conveniences and makes slight modifications to Backus-Naur Form (BNF) [3], without going beyond context-free grammars.

Instead of BNF's angle-bracket notation for nonterminals, ABNF uses case-insensitive names consisting of letters, digits, and dashes, e.g. `HTTP-message` and `IPv6address`. ABNF includes an angle-bracket notation for prose descriptions, e.g. `<host, see [RFC3986], Section 3.2.2>`, usable as last resort in the definiens of a nonterminal.

While BNF allows arbitrary terminals, ABNF uses only natural numbers as terminals, and denotes them via: (i) binary, decimal, or hexadecimal sequences, e.g. `%b1.11.1010`, `%d1.3.10`, and `%x.1.3.a` all denote the string '1 3 10'; (ii) binary, decimal, or hexadecimal ranges, e.g. `%x30-39` denotes any string 'n' with $48 \leq n \leq 57$ (an ASCII digit); (iii) case-sensitive ASCII strings, e.g. `%s"Ab"` denotes the string '65 98'; and (iv) case-insensitive ASCII strings, e.g. `%i"ab"`, or just `"ab"`, denotes any string among '65 66', '65 98', '97 66', and '97 98'. ABNF terminals in suitable sets represent ASCII or Unicode characters.

ABNF allows repetition prefixes *n*`*`*m*, where *n* and *m* are natural numbers in decimal notation; if absent, *n* defaults to 0, and *m* defaults to infinity. For example, `1*4HEXDIG` denotes one to four `HEXDIG`s, `*3DIGIT` denotes up to three `DIGIT`s, and `1*OCTET` denotes one or more `OCTET`s. A single *n* prefix abbreviates *n*`*`*n*, e.g. `3DIGIT` denotes three `DIGIT`s.

Instead of BNF's `|`, ABNF uses `/` to separate alternatives. Repetition prefixes have precedence over juxtapositions, which have precedence over `/`. Round brackets group things and override the aforementioned precedence rules, e.g. `*(WSP / CRLF WSP)` denotes strings obtained by repeating, zero or more times, either (i) a `WSP` or (ii) a `CRLF` followed by a `WSP`. Square brackets also group things but make them optional, e.g. `[":" port]` is equivalent to `0*1(":" port)`.

Instead of BNF's `::=`, ABNF uses `=` to define nonterminals, and `=/` to incrementally add alternatives to previously defined nonterminals. For example, the rule `BIT = "0" / "1"` is equivalent to `BIT = "0"` followed by `BIT =/ "1"`.

The syntax of ABNF itself is formally specified in ABNF [9, Section 4], after the syntax and semantics of ABNF are informally specified in natural language [9, Sections 1–3]. The syntax rules of ABNF prescribe the ASCII codes allowed for white space (spaces and horizontal tabs), line endings (carriage returns followed by line feeds), and comments (semicolons to line endings).

2.2 ACL2

ACL2 is a general-purpose interactive theorem prover based on an untyped first-order logic of total functions that is an extension of a purely functional subset of Common Lisp [15]. Predicates are functions and formulas are terms; they are false when their value is `nil`, and true when their value is `t` or anything non-`nil`.

```
(defun fact (n)
  (if (zp n) 1 (* n (fact (- n 1)))))

(defthm above
  (implies (natp n) (>= (fact n) n)))
```

```
(defchoose below (b) (n)
  (and (natp b) (< b (fact n))))

(defun-sk between (n)
  (exists (m)
    (and (natp m) (< (below n) m) (< m (fact n)))))
```

Fig. 1. Some simple examples of ACL2 functions and theorems

The ACL2 syntax is consistent with Lisp. A function application is a parenthesized list consisting of the function's name followed by the arguments, e.g. $x + 2 \times f(y)$ is written `(+ x (* 2 (f y)))`. Names of constants start and end with `*`, e.g. `*limit*`. Comments extend from semicolons to line endings (like ABNF, incidentally).

The user interacts with ACL2 by submitting a sequence of theorems, function definitions, etc. ACL2 attempts to prove theorems automatically, via algorithms similar to NQTHM [7], most notably simplification and induction. The user guides these proof attempts mainly by (i) proving lemmas for use by specific proof algorithms (e.g. rewrite rules for the simplifier) and (ii) supplying theorem-specific 'hints' (e.g. to case-split on certain conditions).

The factorial function can be defined like `fact` in Fig. 1, where `zp` tests if `n` is 0 or not a natural number. Thus `fact` treats arguments that are not natural numbers as 0. ACL2 functions often handle arguments of the wrong type by explicitly or implicitly coercing them to the right type—since the logic is untyped, in ACL2 a 'type' is just any subset of the universe of values.

To preserve logical consistency, recursive function definitions must be proved to terminate via a measure of the arguments that decreases in each recursive call according to a well-founded relation. For `fact`, ACL2 automatically finds a measure and proves that it decreases according to a standard well-founded relation, but sometimes the user has to supply a measure.

A theorem saying that `fact` is above its argument can be introduced like `above` in Fig. 1, where `natp` tests if `n` is a natural number. ACL2 proves this theorem automatically (if a standard arithmetic library [24, books/arithmetic] is loaded), finding and using an appropriate induction rule—the one derived from the recursive definition of `fact`, in this case.

Besides the discouraged ability to introduce arbitrary axioms, ACL2 provides logical-consistency-preserving mechanisms to axiomatize new functions, such as indefinite description functions. A function constrained to be strictly below `fact` can be introduced like `below` in Fig. 1, where `b` is the variable bound by the indefinite description. This introduces the logically conservative axiom that, for every `n`, `(below n)` is a natural number less than `(fact n)`, if any exists—otherwise, `(below n)` is unconstrained.

ACL2's Lisp-like macro mechanism provides the ability to extend the language with new constructs defined in terms of existing constructs. For instance, despite the lack of built-in quantification in the logic, functions with top-level quantifiers can be introduced. The existence of a value strictly between `fact` and `below` can be expressed by a predicate like `between` in Fig. 1, where `defun-sk`

```
rulelist = 1*( rule / (*c-wsp c-nl) )
rule = rulename defined-as elements c-nl
defined-as = *c-wsp ("=" / "=/") *c-wsp
elements = alternation *c-wsp
alternation = concatenation *(*c-wsp "/" *c-wsp concatenation)
concatenation = repetition *(1*c-wsp repetition)
repetition = [repeat] element
element = rulename / group / option / char-val / num-val / prose-val
group = "(" *c-wsp alternation *c-wsp ")"
option = "[" *c-wsp alternation *c-wsp "]"
num-val = "%" (bin-val / dec-val / hex-val)
bin-val = "b" 1*BIT [ 1*("." 1*BIT) / ("-" 1*BIT) ]
dec-val = "d" 1*DIGIT [ 1*("." 1*DIGIT) / ("-" 1*DIGIT) ]
hex-val = "x" 1*HEXDIG [ 1*("." 1*HEXDIG) / ("-" 1*HEXDIG) ]
c-wsp = WSP / (c-nl WSP)
c-nl = comment / CRLF
CRLF = CR LF ; carriage return and line feed
WSP = SP / HTAB ; space or horizontal tab
```

Fig. 2. Some rules of the ABNF grammar of ABNF

is a macro defined in terms of `defchoose` and `defun`, following a well-known construction [2].

3 ABNF Formalization

3.1 Abstract Syntax

The formalization starts by defining an abstract syntax of ABNF, based on the ABNF rules that define the concrete syntax of ABNF.[1] To ease validation by inspection, this abstract syntax closely follows the structure of the concrete syntax (as exemplified below) and abstracts away only essentially lexical details (e.g. white space, comments, and defaults of repetition prefixes). ACL2's FTY macro library for introducing structured recursive types [23] is used to define the abstract syntactic entities of ABNF—11 types in total.

For example, the concrete syntax of numeric terminal notations such as `%d1.3.10` and `%x30-39` is defined in ABNF by `num-val` in Fig. 2, where the definitions of `BIT`, `DIGIT`, and `HEXDIG` are not shown but should be obvious from the names. In ACL2, the corresponding abstract syntax is formalized by `num-val` in Fig. 3, where: `fty::deftagsum` introduces a tagged sum type (disjoint union); `num-val` is the name of the type; `:direct` tags direct notations such as `%d1.3.10` whose only component `get` is a list of natural numbers (type `nat-list`, whose recognizer is `nat-listp`) such as `(1 3 10)`; and `:range` tags range notations such as `%x30-39` whose components `min` and `max` are natural numbers (type `nat`, whose recognizer is `natp`) such as `48` and `57`. This type definition introduces: a recognizer `num-val-p` for the type; constructors `num-val-direct` and `num-val-range`; destructors `num-val-direct->get`, `num-val-range->min`, and `num-val-range->max`; and several theorems about these functions. Compared to the concrete syntax rule `num-val` in Fig. 2, the abstract syntax type `num-val`

[1] The meta circularity of the definition of the concrete syntax of ABNF in ABNF is broken by the human in the loop, who defines an abstract syntax of ABNF in ACL2.

```
(fty::deftagsum num-val
  (:direct ((get nat-list)))
  (:range ((min nat) (max nat))))

(fty::defprod rule
  ((name rulename)
   (incremental bool)
   (definiens alternation)))

(fty::deflist rulelist
  :elt-type rule)
```

```
(fty::deftypes alt/conc/rep/elem
  (fty::deflist alternation :elt-type concatenation)
  (fty::deflist concatenation :elt-type repetition)
  (fty::defprod repetition
    ((range repeat-range) (element element)))
  (fty::deftagsum element
    (:rulename ((get rulename)))
    (:group ((get alternation)))
    (:option ((get alternation)))
    (:char-val ((get char-val)))
    (:num-val ((get num-val)))
    (:prose-val ((get prose-val)))))
```

Fig. 3. Some excerpts of the abstract syntax of ABNF formalized in ACL2

in Fig. 3 abstracts the binary, decimal, or hexadecimal notations to their natural number values.

As another example, the concrete syntax of rule definientia is defined in ABNF by `alternation` and mutually recursive companions in Fig. 2. In ACL2, the corresponding abstract syntax is formalized by `alternation` and mutually recursive companions in Fig. 3, where: `fty::deftypes` introduces mutually recursive types; `fty::deflist` introduces a type of lists over the element type that appears after `:elt-type`; `fty::defprod` introduces a product type similar to a `fty::deftagsum` summand; `repeat-range` is a type for repetition prefixes such as `1*` or `3*6`; `rulename` is a type for rule names; `char-val` is a type for string terminal notations such as `%s"Ab"`; and `prose-val` is a type for prose notations such as `<host, see [RFC3986], Section 3.2.2>`. These type definitions introduce recognizers, constructors, destructors, and theorems analogous to the ones for `num-val` above. Compared to the concrete syntax rules `alternation` and companions in Fig. 2, the abstract syntax types `alternation` and companions in Fig. 3 abstract away comments, white space, and line endings.

As a third example, the concrete syntax of grammars (i.e. lists of rules) is defined in ABNF by `rulelist` in Fig. 2. In ACL2, the corresponding abstract syntax is formalized by `rulelist` in Fig. 3, where the `incremental` component of `rule` is a boolean that says whether the rule is incremental (`=/`) or not (`=`). Compared to the concrete syntax rules `rulelist` and `rule`, the abstract syntax types `rulelist` and `rule` abstract away comments, white space, and line endings.

The syntactic structure of ABNF grammars is more complex than the syntactic structure of plain context-free grammars. ABNF rule definientia are expressions built out of various terminal notations (direct and range numeric notations, etc.) and operators (alternation, concatenation, repetition, etc.), while plain context-free rule definientia are sequences of symbols.

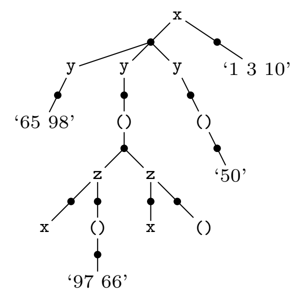
3.2 Semantics

An ABNF grammar describes how a sequence of natural numbers (terminals) can be organized in tree structures according to the grammar's rules. Thus, the semantics of the abstract syntactic entities is formalized via matching relations with trees. The notion of language generated by a grammar is derived from that.

```
(fty::deftypes trees
  (fty::deftagsum tree
    (:leafterm ((get nat-list)))
    (:leafrule ((get rulename)))
    (:nonleaf ((rulename? maybe-rulename) (branches tree-list-list))))
  (fty::deflist tree-list :elt-type tree)
  (fty::deflist tree-list-list :elt-type tree-list))

(defun tree-match-num-val-p (tree num-val)
  (and (tree-case tree :leafterm)
       (let ((nats (tree-leafterm->get tree)))
         (num-val-case num-val
                       :direct (equal nats num-val.get)
                       :range (and (equal (len nats) 1)
                                   (<= num-val.min (car nats))
                                   (<= (car nats) num-val.max))))))

(mutual-recursion
  (defun tree-list-list-match-alternation-p (treess alt rules) ...)
  (defun tree-list-list-match-concatenation-p (treess conc rules) ...)
  (defun tree-list-match-repetition-p (trees rep rules) ...)
  (defun tree-list-match-element-p (trees elem rules) ...)
  (defun tree-match-element-p (tree elem rules)
    (element-case elem
     :rulename (tree-case tree
                :leafterm nil
                :leafrule (equal tree.get elem.get)
                :nonleaf (and (equal tree.rulename? elem.get)
                              (let ((alt (lookup-rulename elem.get rules)))
                                (tree-list-list-match-alternation-p
                                 tree.branches alt rules))))
     :group ...
     :option ...
     :char-val ...
     :num-val (tree-match-num-val-p tree elem.get)
     :prose-val t)))

(defun parse-treep (tree string rulename rules)
  (and (treep tree)
       (tree-match-element-p tree (element-rulename rulename) rules)
       (equal (tree->string tree) string)))

(defun-sk languagep (nats rulenames rules)
  (exists (rulename tree) (and (nat-listp nats)
                               (in rulename rulenames)
                               (parse-treep tree nats rulename rules))))
```

Fig. 4. Some excerpts of the semantics of ABNF formalized in ACL2

Since a single terminal notation like `%d1.3.10` or `%s"Ab"` denotes multiple natural numbers in sequence, it is convenient to use lists (i.e. strings) of natural numbers, instead of individual natural numbers, to label leaves of trees. A rule name (nonterminal) can label the root of a (sub)tree, with branches for one of the concatenations of the alternation that defines the rule name. Since a concatenation is a sequence of repetitions, and each repetition may denote multiple instances of its element, the branches are organized into a list of lists: the outer list matches the list of repetitions that form the concatenation, and each inner list matches the element instances of the corresponding repetition; this organization facilitates the formulation of the matching relations (see below). Rule names can also label leaves, to represent the tree structure of strings that include nonterminals. Round-bracketed groups and square-bracketed options are like anonymous rules: roots of (sub)trees that match groups and options are not labeled by rule names, but have lists of lists of branches for concatenations tree:



rules:

```
x = *y %d1.3.10
y = %s"Ab" / (2z / %x30-39)
z = x ["ab"]
```

string:

'65 98 x 97 66 x 50 1 3 10'

Fig. 5. An example of a tree for a string, given some rules

```
(defchoose parse-http (result) (string)
  (if (string-parsablep string *http-message* *http-grammar*)
      (and (parse-treep result string *http-message* *http-grammar*)
           (disambiguatep result))
    (equal result *error*)))
```

Fig. 6. A sketch of a declarative specification of an HTTP parser

from the alternations inside the brackets, in the same way as named rules; additionally, a square-bracketed option is allowed to have an empty list of lists of branches, to represent the absence of the option.

Formally, (lists of (lists of)) trees are recursively defined by `tree` and mutually recursive companions in Fig. 4, where `maybe-rulename` is a type consisting of rule names and `nil`—the latter is used for roots not labeled by rule names. A function `tree->string` (whose definition is not shown here) collects the natural numbers and rule names at the leaves of a tree, from left to right, into a string (i.e. list).

Trees can be visualized as in Fig. 5. Leaves are labeled by lists of natural numbers or rule names. Roots of (sub)trees are labeled by rule names or, for groups and options, by `()` (which is another way to write `nil` in ACL2). Lines with joints represent lists of lists of branches.

A tree matches a direct numeric terminal notation iff it is a leaf labeled by the same list of natural numbers; a tree matches a range numeric terminal notation iff it is a leaf labeled by a list of one natural number in the range. This is formalized by `tree-match-num-val-p` in Fig. 4, where: `(tree-case tree :leafterm)` tests if `tree` is tagged by `:leafterm`; `(num-val-case num-val ...)` performs a case analysis on the tag of `num-val` (Fig. 3) that binds the variables with dots in their names to the corresponding components of the target variable `num-val` (e.g. `num-val.get` is bound to `(num-val-direct->get num-val)`); `len` returns the length of a list; and `car` returns the first value of a list.

Since an element (e.g. a numeric terminal notation) is matched by a tree, a repetition is matched by a list of trees: the length of the list must be within the repetition prefix's range, and each tree of the list must match the repetition's element. Since a concatenation is a list of repetitions, a concatenation is matched by a list of lists of trees: the length of the outer list must equal the length of the concatenation, and each inner list must match the corresponding repetition. Since an alternation denotes one of its concatenations at a time, an alternation is matched by a list of lists of trees, which must match one of the alternation's concatenations. A rule name is matched by either a leaf tree labeled by the rule name, or a non-leaf tree whose root is labeled by the rule name and whose branches match the alternation that defines the rule name. A group is matched by a non-leaf tree whose root is labeled by `()` and whose branches match the alternation inside the group; an option is matched by either a tree in the same way as a group, or by a non-leaf tree whose root is labeled by `()` and with an empty list of lists of trees as branches.

The assertions in the previous paragraph are formalized by `tree-list-list-match-alternation-p` and mutually recursive companions in Fig. 4, where: `mutual-recursion` introduces mutually recursive `defun`s; `(element-case elem ...)` and `(tree-case tree ...)` perform case analyses on the tags of `elem` (Fig. 3) and `tree` (Fig. 4), analogously to `num-val-case` as explained above; and `lookup-rulename` collects, from the rules of a grammar, all the alternatives that define a rule name. The termination of these mutually recursive functions is proved via a lexicographic measure consisting of the size of the trees followed by the size of the abstract syntactic entities.

A prose notation is matched by any tree, as far as the ABNF semantics alone is concerned. Predicates on trees, external to ABNF grammars, can be used to define the meaning of specific prose notations, and conjoined with the tree matching predicates to specify parsing requirements.[2] Some grammars use prose notations to refer to rules from other grammars, e.g. the HTTP grammar uses prose notations to refer to rules from the URI grammar (an example is in Sect. 2.1): the grammar composition operations briefly mentioned in Sect. 1 replace these prose notations with the referenced rules, resulting in a combined grammar without prose notations.

Given the rules of a grammar and a rule name, a parse tree for a string is a tree that matches the rule name and that has the string at the leaves, as formalized by `parse-treep` in Fig. 4. Given the rules of a grammar and a set of rule names, a string of the language generated by the rules starting from the rule names is a list of natural numbers at the leaves of some parse tree whose root is one of the rule names, as formalized by `languagep` in Fig. 4, where `in` tests set membership; since ABNF grammars do not have an explicit notion of start nonterminal, the start nonterminals of interest are specified by the second argument of `languagep`.

The `parse-treep` predicate can be used to write declarative specifications of correct parsing of ABNF-specified languages. For instance, a (non-executable) HTTP parser can be specified by something like `parse-http` in Fig. 6, where: `*http-grammar*` is a constant of type `rulelist` (Fig. 3) representing the rules of

[2] Future work includes exploring mechanisms to "plug" such external predicates into the ABNF semantics.

```
(def-rule-const *group*
  (/_ "(" (*_ *c-wsp*) *alternation* (*_ *c-wsp*) ")"))

(def-rule-const *num-val*
  (/_ "%" (!_ (/_ *bin-val*) (/_ *dec-val*) (/_ *hex-val*))))
```

Fig. 7. Some excerpts of the concrete syntax of ABNF formalized in ACL2

the ABNF grammar of HTTP; `*http-message*` is a constant of type `rulename` (Fig. 3) representing the top-level rule name `HTTP-message`; `string-parsablep` holds iff there exists a parse tree for the string; the predicate `disambiguatep` states disambiguating restrictions (since the grammar of HTTP is ambiguous); and `*error*` is a constant representing an error, distinct from trees. The function `parse-http` returns concrete syntax trees, because grammars do not specify abstract syntax; a practical HTTP parser can be specified as the composition of `parse-http` followed by a suitable HTTP syntax abstraction function (analogous to the ABNF syntax abstraction functions described in Sect. 3.3).

The semantics of ABNF grammars is more complex than the semantics of plain context-free grammars. ABNF parse trees have branches organized as lists of lists and have roots of non-leaf trees possibly labeled by `()`, while parse trees of plain context-free grammars have branches organized as lists and have roots of non-leaf trees always labeled by nonterminals. Accordingly, the ABNF tree matching relations are more complex than the tree matching relations of plain context-free grammars.

3.3 Concrete Syntax

The concrete syntax of ABNF is formalized in ACL2 using the rules of the ABNF grammar of ABNF, but "written in abstract syntax" because the concrete syntax is not available before it is formalized. This safely captures the meta circularity.

Since the FTY constructors of the abstract syntax are verbose, some specially crafted and named functions and macros are defined, and used to write abstract syntactic entities in a way that looks more like concrete syntax, easing not only their writing, but also their validation by inspection. For example, the rules `group` and `num-val` (Fig. 2) are written in abstract syntax as shown in Fig. 7.

After transcribing the 40 rules of the ABNF grammar of ABNF to this form, a constant `*abnf-grammar*` consisting of their list is defined. Since grammars, i.e. values of type `rulelist` (Fig. 3), are endowed with semantics (Sect. 3.2), this constant provides a formalization of the concrete syntax of ABNF in ACL2.

The link between the concrete and abstract syntax of ABNF is formalized by 51 executable ACL2 functions that map parse trees to their corresponding abstract syntactic entities: these are abstraction functions, which distill the abstract syntactic information from the concrete syntactic information. For example, a function `abstract-num-val` (whose definition is not shown here) maps a tree that matches the rule name `num-val` (Fig. 2) to a value of type `num-val` (Fig. 3). This function calls other abstraction functions on its subtrees, e.g. a function `abstract-*bit` (whose definition is not shown here) that maps a list

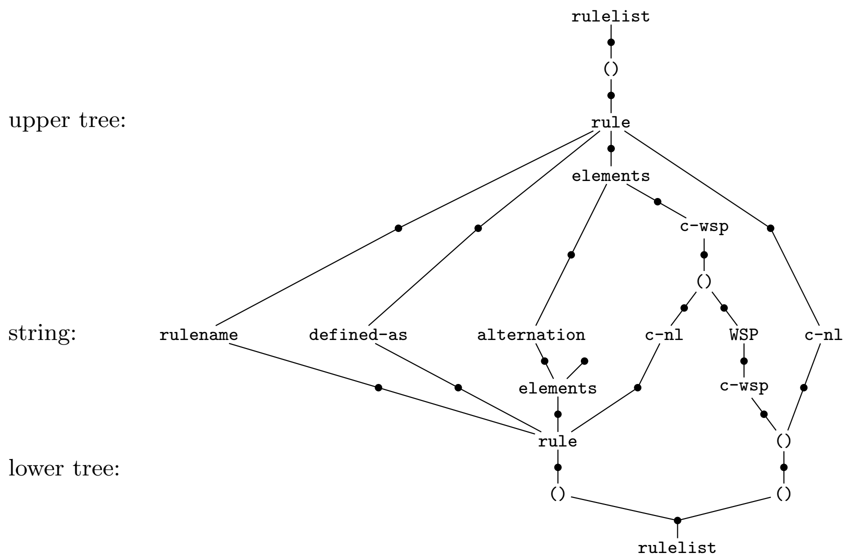


Fig. 8. An example showing the ambiguity of the ABNF grammar of ABNF

of trees that matches `*BIT` to the big endian value of their bits. The top-level abstraction function `abstract-rulelist` (whose definition is not shown here) maps a tree that matches the rule name `rulelist` (Fig. 2) to the corresponding value of type `rulelist` (Fig. 3)—a grammar.

4 ABNF Grammar Parser

When specifying correct parsing of an ABNF-specified language as sketched in Fig. 6, a constant like `*http-grammar*` can be built by manually transcribing the grammar, as done for `*abnf-grammar*` (Sect. 3.3). A better alternative is to perform this transcription automatically, by running (i) the grammar parser described in the rest of this section, which produces a parse tree that matches `rulelist`, followed by (ii) `abstract-rulelist` (Sect. 3.3) on the resulting parse tree.

Since the grammar parser is verified as described below, and this automatic transcription process operates on the actual grammar text (e.g. copied and pasted from an Internet standard document), the resulting formal parsing specification is faithful to the grammar.

Running this process on the ABNF grammar of ABNF produces the same value as the manually built `*abnf-grammar*`. This provides a validation.

4.1 Implementation

The ABNF grammar of ABNF is ambiguous, as shown by the two different parse trees for the same string (of nonterminals, for brevity) in Fig. 8: the first `c-nl` can either end a `rule` (lower tree) or form, with the `WSP`, a `c-wsp` under `elements` (upper tree). The ambiguity only affects where certain comments, white space, and line endings go in the parse trees; it does not affect the abstract syntax, and thus the semantics, of ABNF. The parser resolves the ambiguity by always parsing as many consecutive `c-wsp`s as possible, as in the upper tree in Fig. 8.[3]

Aside from this ambiguity, the ABNF grammar of ABNF is mostly LL(1), with some LL(2) and LL($*$) parts [1,21]. The parser is implemented as a recursive descent with backtracking. Backtracking is expected to be limited in reasonable grammars. Indeed, the parser runs very quickly on all the example grammars included in the development—fractions of a second, including file reading, which is adequate for the expected use of the parser outlined above.

The parser consists of 85 executable ACL2 functions. There is a parsing function for each rule, and parsing functions for certain groups, options, repetitions, and terminal notations. ACL2's Seq macro library for stream processing [25, `seq`] is used to define these functions in a more readable way. Each function takes a list of natural numbers to parse as input, and, consistently with Seq, returns (i) an indication of success (`nil`, i.e. no error) or failure (an error message, which is never `nil`), (ii) a (list of) parse tree(s) if successful, and (iii) the remaining natural numbers in the input.

For example, the parsing function for `CRLF` (Fig. 2) is `parse-crlf` in Fig. 9, where: first `parse-cr` parses a carriage return, yielding a `CR` parse tree that is assigned to `tree-cr`; then `parse-lf` parses a line feed, yielding an `LF` parse tree that is assigned to `tree-lf`; and finally `return` returns (i) `nil` (success), (ii) a `CRLF` parse tree with the two subtrees, and (iii) the remaining input after the carriage return and line feed. If `parse-cr` or `parse-lf` fails, `parse-crlf` fails. The threading of the input and the propagation of the failures is handled by the `seq` macro behind the scenes.

As another example, the parsing function for `WSP` (Fig. 2) is `parse-wsp` in Fig. 9, where: `parse-sp` attempts to parse a space, returning a `WSP` parse tree with a `SP` subtree if successful; otherwise `parse-htab` attempts to parse a horizontal tab, returning a `WSP` parse tree with a `HTAB` subtree if successful. If both `parse-sp` and `parse-htab` fail, `parse-wsp` fails. The backtracking is handled by the `seq-backtrack` macro behind the scenes.

As a third example, the parsing function for `*BIT` is `parse-*bit` in Fig. 9, which uses `parse-bit` to parse as many bits as possible, eventually returning the corresponding list of `BIT` parse trees; `cons` adds an element to the front of a list. The termination of `parse-*bit` is proved by the decrease of the length of the input. This function never fails: when no bits can be parsed, the empty list `nil` of parse trees is returned.

The parsing functions for `alternation` and mutually recursive companions (Fig. 2) are mutually recursive (their definitions are not shown here). Their termination is proved via a lexicographic measure consisting of the size of the input followed by a linear ordering of these functions—the length of the input alone is

[3] Future work includes exploring the formulation of an unambiguous ABNF grammar of ABNF that provably defines the same language as the current ambiguous one.

```
(defun parse-crlf (input)
  (seq input
       (tree-cr := (parse-cr input))
       (tree-lf := (parse-lf input))
       (return (tree-nonleaf *crlf* (list (list tree-cr) (list tree-lf))))))

(defun parse-wsp (input)
  (seq-backtrack input
                 ((tree := (parse-sp input))
                  (return (tree-nonleaf *wsp* (list (list tree)))))
                 ((tree := (parse-htab input))
                  (return (tree-nonleaf *wsp* (list (list tree)))))))

(defun parse-*bit (input)
  (seq-backtrack input
                 ((tree := (parse-bit input))
                  (trees := (parse-*bit input))
                  (return (cons tree trees)))
                 ((return nil))))

(defun parse-grammar (input)
  (b* (((mv error? tree? rest) (parse-rulelist input)))
    (cond (error? nil) (rest nil) (t tree?))))
```

Fig. 9. Some excerpts of the ABNF grammar parser in ACL2

```
(defthm parse-treep-of-parse-grammar
  (implies (and (nat-listp input)
                (parse-grammar input))
           (parse-treep (parse-grammar input) input *rulelist* *abnf-grammar*)))

(defthm input-decomposition-of-parse-crlf
  (implies (and (nat-listp input)
                (not (mv-nth 0 (parse-crlf input))))
           (equal (append (tree->string (mv-nth 1 (parse-crlf input)))
                          (mv-nth 2 (parse-crlf input)))
                  input)))

(defthm tree-match-of-parse-crlf
  (implies (and (nat-listp input)
                (not (mv-nth 0 (parse-crlf input))))
           (tree-match-element-p (mv-nth 1 (parse-crlf input))
                                 (element-rulename *crlf*)
                                 *abnf-grammar*)))
```

Fig. 10. Some excerpts of the ABNF grammar parser soundness proof in ACL2

insufficient to prove termination, because some (e.g. `parse-alternation`) call others (e.g. `parse-concatenation`) on the same input.

The top-level parsing function is `parse-grammar` in Fig. 9, where `b*` binds the results of `parse-rulelist` to the three variables in the triple `(mv ...)`. The function checks that there is no remaining input, returning just the parse tree if successful (or `nil`, i.e. no parse tree, if a failure occurs). There is also a wrapper function `parse-grammar-from-file` (whose definition is not shown here) that takes a file name as input and calls `parse-grammar` on the file's content.

4.2 Verification

The correctness of the parser consists of:
- Soundness: the parser recognizes only ABNF grammars.
- Completeness: the parser recognizes all ABNF grammars (almost; see below).

```
(defthm parse-grammar-when-tree-match
  (implies (and (treep tree)
                (tree-match-element-p tree (element-rulename *rulelist*) *abnf-grammar*)
                (tree-terminatedp tree)
                (tree-rulelist-restriction-p tree))
           (equal (parse-grammar (tree->string tree)) tree)))

(defthm parse-wsp-when-tree-match
  (implies (and (treep tree)
                (nat-listp rest-input)
                (tree-match-element-p tree (element-rulename *wsp*) *abnf-grammar*)
                (tree-terminatedp tree))
           (equal (parse-wsp (append (tree->string tree) rest-input))
                  (mv nil tree rest-input))))

(defthm parse-*bit-when-tree-list-match
  (implies (and (tree-listp trees)
                (nat-listp rest-input)
                (tree-list-match-repetition-p trees (*_ *bit*) *abnf-grammar*)
                (tree-list-terminatedp trees)
                (mv-nth 0 (parse-bit rest-input)))
           (equal (parse-*bit (append (tree-list->string trees) rest-input))
                  (mv nil trees rest-input))))

(defthm fail-sp-when-match-htab
  (implies (and (tree-match-element-p tree (element-rulename *htab*) *abnf-grammar*)
                (tree-terminatedp tree))
           (mv-nth 0 (parse-sp (append (tree->string tree) rest-input)))))

(defthm constraints-from-parse-sp
  (implies (not (mv-nth 0 (parse-sp input)))
           (equal (car input) 32)))

(defthm constraints-from-tree-match-htab
  (implies (and (tree-match-element-p tree (element-rulename *htab*) *abnf-grammar*)
                (tree-terminatedp tree))
           (equal (car (tree->string tree)) 9)))

(defun-sk pred-alternation (input)
  (forall (tree rest-input)
    (implies (and (treep tree)
                  (nat-listp rest-input)
                  (tree-match-element-p tree (element-rulename *alternation*) *abnf-grammar*)
                  (tree-terminatedp tree)
                  ... ; 8 parsing failure hypotheses on rest-input
                  (equal input (append (tree->string tree) rest-input)))
             (equal (parse-alternation (append (tree->string tree) rest-input))
                    (mv nil tree rest-input)))))

(defthm parse-alternation-when-tree-match-lemma
  (pred-alternation input))
```

Fig. 11. Some excerpts of the ABNF grammar parser completeness proof in ACL2

More precisely, the parser not only recognizes ABNF grammars, but also returns the corresponding parse trees, as elaborated below.

The main soundness theorem is `parse-treep-of-parse-grammar` in Fig. 10, where `*rulelist*` represents the rule name `rulelist` (Fig. 2). Semi-formally, the theorem says:

`input` is a list of natural numbers $\wedge$
`(parse-grammar input)` $\neq$ `nil` $\Longrightarrow$
`(parse-grammar input)` is a parse tree
  with `rulelist` at the root and `input` at the leaves

That is, if `parse-grammar` (Fig. 9) succeeds, it returns a parse tree that organizes the input into the tree structure of a grammar (i.e. a list of rules).

This main soundness theorem is proved via two theorems for each of the parsing functions that return triples:

- Input decomposition: if the function succeeds, the string at the leaves of the returned parse tree(s) consists of the natural numbers parsed from the input, and the function also returns the remaining natural numbers in the input.
- Tree matching: if the function succeeds, the returned parse tree/trees is/are consistent with the syntactic entity that the function is intended to parse.

For example, the input decomposition theorem of `parse-crlf` (Fig. 9) is `input-decomposition-of-parse-crlf` in Fig. 10, where `mv-nth` extracts the components (zero-indexed) of the triple returned by `parse-crlf`. The theorem says that if `parse-crlf` succeeds (i.e. its first result is `nil`, not an error), joining the string at the leaves of the returned tree with the returned remaining input yields the original input.

Each input decomposition theorem is proved by expanding the parsing function and using the input decomposition theorems of the called parsing functions as rewrite rules. For instance, in `input-decomposition-of-parse-crlf` (Fig. 10), expanding `parse-crlf` turns the `(append ...)` into one involving `parse-cr` and `parse-lf`, making their input decomposition theorems applicable.

As another example, the tree matching theorem of `parse-crlf` (Fig. 9) is `tree-match-of-parse-crlf` in Fig. 10. The theorem says that if `parse-crlf` succeeds (formulated in the same way as in the input decomposition theorem), the returned parse tree matches `CRLF`—which `parse-crlf` is intended to parse.

Each tree matching theorem is proved by expanding the parsing function and the tree matching predicate, and using the tree matching theorems of the called functions as rewrite rules. For instance, in the `tree-match-of-parse-crlf`, expanding `parse-crlf` and `tree-match-element-p` turns the conclusion into the assertion that the subtrees match `CR` and `LF` when `parse-cr` and `parse-lf` succeed, making their tree matching theorems applicable.

The input decomposition and tree matching theorems of the recursive parsing functions (e.g. `parse-*bit` in Fig. 9) are proved by induction on their recursive definitions.

The main soundness theorem, `parse-treep-of-parse-grammar` (Fig. 10), is proved from the input decomposition and tree matching theorems of `parse-rulelist`, and the fact that `parse-grammar` fails if there is remaining input.

Since the ABNF grammar of ABNF is ambiguous (Fig. 8) but the parser returns a single parse tree at a time, completeness is not provable. But it is provable relatively to trees consistent with how the parser resolves the ambiguity. A predicate `tree-rulelist-restriction-p` formalizes these restrictions on trees: each `(*c-wsp c-nl)` subtree, except the one (if any) that starts a `rulelist`, must not start with `WSP`.

The main completeness theorem is `parse-grammar-when-tree-match` in Fig. 11, where `tree-terminatedp` tests if a tree is terminated, i.e. if the string at its leaves has only natural numbers and no rule names. Semi-formally, the theorem says:

`tree` is a tree $\wedge$
`tree` matches `rulelist` $\wedge$
`tree` has no rule names at the leaves $\wedge$
`tree` satisfies the disambiguating restrictions $\Longrightarrow$
`(parse-grammar (tree->string tree))` $=$ `tree`

That is, if a terminated tree matches a `rulelist` (i.e. it is a concrete syntactic representation of a grammar) and is consistent with how the parser resolves the ambiguity, `parse-grammar` succeeds on the string at the leaves of the tree and returns the tree.

This main completeness theorem is proved via an auxiliary completeness theorem for each of the parsing functions that return triples. The formulation of these auxiliary theorems is analogous to the main one, but with additional complexities: in the conclusions, the parsing functions are applied to the string at the leaves of the tree(s) joined with some remaining input; this makes these theorems usable as rewrite rules, and enables the addition of certain critical hypotheses to these theorems.

For example, the completeness theorem of `parse-wsp` (Fig. 9) is `parse-wsp-when-tree-match` in Fig. 11. As another example, the completeness theorem of `parse-*bit` (Fig. 9) is `parse-*bit-when-tree-list-match` in Fig. 11. The hypothesis that `parse-bit` fails on `rest-input` is critical: without it, `parse-*bit` might parse another bit from `rest-input`, and return a longer list of trees than `trees`.

Each auxiliary completeness theorem is proved by expanding the parsing function and the tree matching predicate, using the completeness theorems of the called functions as rewrite rules, and also using, as needed, certain disambiguation theorems.

The need and nature of these disambiguation theorems, in simple form, are illustrated by considering the proof of the completeness theorem of `parse-wsp`. The hypothesis that `tree` matches `WSP` expands to two cases:

1. The subtree matches the `SP` alternative of `WSP`. In this case, the completeness theorem of `parse-sp` applies, `parse-sp` succeeds returning the subtree, and `parse-wsp` succeeds returning `tree`.
2. The subtree matches the `HTAB` alternative of `WSP`. For the completeness theorem of `parse-htab` to apply, `parse-sp` must be shown to fail so that `parse-wsp` reduces to `parse-htab` and the proof proceeds as in the `SP` case.[4]

The theorem saying that `parse-sp` fails on the string at the leaves of a terminated tree matching `HTAB` is `fail-sp-when-match-htab` in Fig. 11. This theorem is proved via two theorems saying that `parse-sp` and `HTAB` induce incompatible constraints on the same value at the start of the input: the two theorems are `constraints-from-parse-sp` and `constraints-from-tree-match-htab` in Fig. 11. The incompatible constraints are that `parse-sp` requires the ASCII code 32, while `HTAB` requires the ASCII code 9.

[4] Even though the roles of `SP` and `HTAB` are "symmetric" in the rule `WSP` in Fig. 2, the function `parse-wsp` in Fig. 9 "asymmetrically" tries to parse `SP` before `HTAB`.

There are 26 parsing constraint theorems similar to the one for `parse-sp`, and 49 tree matching constraint theorems similar to the one for `HTAB`. There are 87 disambiguation theorems similar to `fail-sp-when-match-htab` (Fig. 11): they say that certain parsing functions fail when trees match certain syntactic entities, effectively showing that the parser can disambiguate all the alternatives in the ABNF grammar of ABNF, including deciding when to stop parsing unbounded repetitions. The disambiguation theorems are used to prove not only some completeness theorems, but also other disambiguation theorems. Some disambiguation theorems critically include parsing failure hypotheses similarly to the completeness theorem of `parse-*bit` (Fig. 11). Many disambiguation theorems show incompatible constraints just on the first one or two natural numbers in the input, corresponding to LL(1) and LL(2) parts of the grammar. But for LL($*$) parts of the grammar, the disambiguation theorems show incompatible constraints on natural numbers that follow unbounded prefixes of the input; to "go past" these prefixes in the proofs of these disambiguation theorems, certain completeness theorems are used in turn.

Since the auxiliary completeness theorems call the parsing functions not on variables but on `(append ...)` terms, induction on the recursive parsing functions is not readily applicable [7, Chapt. 15]. For the singly recursive functions like `parse-*bit`, induction on the list of trees is used. For the mutually recursive functions like `parse-alternation`, an analogous induction on the (lists of (lists of)) trees seems unwieldy due to the number (10) of mutually recursive parsing functions. Instead, the desired completeness assertions are packaged into predicates like `pred-alternation` in Fig. 11, where the tree and remaining input are universally quantified and a new variable `input` is equated to the argument of the parsing function. Given these predicates, theorems like `parse-alternation-when-tree-match-lemma` in Fig. 11 are proved by induction on the recursive parsing functions (now applicable to the variable `input`), from which the desired completeness theorems readily follow.

The main completeness theorem, `parse-grammar-when-tree-match` (Fig. 11), is proved from the auxiliary completeness theorem of `parse-rulelist` and the fact that the absence of remaining input fulfills the parsing failure hypotheses on the remaining input.

All the theorems and proofs overviewed in this subsection are discussed in much greater detail in the documentation of the development [25, abnf]. Even the short overview above should convey that the completeness proof is considerably more laborious than the soundness proof, perhaps because the completeness proof must show that the parser can reconstruct any parse tree from its string at the leaves, while the soundness proof must show that the parser can just construct one appropriate parse tree when it succeeds.

5 Related Work

The author is not aware of other formalizations of the ABNF notation. There are formalizations of regular expressions [10], plain context-free grammars [4], and parsing expression grammars [16]. As explained at the end of Sect. 3.1 and Sect. 3.2, the syntax and semantics of ABNF are more complex than those of plain context-free grammars (and of regular expressions). The syntax of parsing expression grammars has some similarities with ABNF, but their semantics is operational, in terms of parsing steps, in contrast with ABNF's tree matching semantics. The referenced works formalize abstract syntax of the grammar notations, but not concrete syntax; in contrast, the work described in this paper formalizes both abstract and concrete syntax of ABNF, using the former to define the latter as faithfully to the meta circularity [9,18] as allowed by the theorem prover's define-before-use constraints, and validating the definition via the verified ABNF grammar parser as mentioned just before Sect. 4.1.

The author is not aware of other verified parsers of ABNF grammars. There are verified parsers of other languages [27,17]. Due to ABNF's role in Internet syntax specifications, a verified parser of ABNF grammars has a practical significance. There are verified generators of parsers, generators of verified parsers, verified parser interpreters, and verified parser validators [20,5,16,13,22,12]. Since they are based on different grammar notations from ABNF, using these tools for the verified parsing of ABNF grammars would require a trusted translation from the ABNF grammar of ABNF to the tools' grammar notations; in contrast, the verification of the parser described in this paper is based directly on the formalized ABNF notation. APG [26] is an ABNF parser generator, but it does not include or generate formal proofs.

Acknowledgements

This work was supported by DARPA under Contract No. FA8750-15-C-0007.

References

1. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Pearson, 2nd edn. (2007)
2. Avigad, J., Zach, R.: The epsilon calculus. In: Zalta, E.N. (ed.) The Stanford Encyclopedia of Philosophy. Metaphysics Research Lab, Stanford University, summer 2016 edn. (2016), https://plato.stanford.edu/archives/sum2016/entries/epsilon-calculus/
3. Backus, J.W., Bauer, F.L., Green, J., Katz, C., McCarthy, J., Perlis, A.J., Rutishauser, H., Samelson, K., Vauquois, B., Wegstein, J.H., van Wijngaarden, A., Woodger, M.: Report on the algorithmic language ALGOL 60. Communications of the ACM **3**(5), 299–314 (May 1960)
4. Barthwal, A.: A formalisation of the theory of context-free languages in higher-order logic. Ph.D. thesis, The Australian National University (2010)
5. Barthwal, A., Norrish, M.: Verified, executable parsing. In: Proc. 18th European Symposium on Programming (ESOP). pp. 160–174 (2009)
6. Berners-Lee, T., Fielding, R., Masinter, L.: Uniform Resource Identifier (URI): Generic syntax. Request for Comments (RFC) 3986 (Jan 2005)
7. Boyer, R.S., Moore, J.S.: A Computational Logic. Academic Press (1979)

8. Bray, T.: The JavaScript Object Notation (JSON) data interchange format. Request for Comments (RFC) 7159 (Mar 2014)
9. Crocker, D., Overell, P.: Augmented BNF for syntax specifications: ABNF. Request for Comments (RFC) 5234 (Jan 2008)
10. Doczkal, C., Kaiser, J.O., Smolka, G.: A constructive theory of regular languages in Coq. In: Proc. 3rd International Conference on Certified Programs and Proofs (CPP). pp. 82–97. No. 8307 in Lecture Notes in Computer Science (2013)
11. Fielding, R., Reschke, J.: Hypertext Transfer Protocol (HTTP/1.1): Message syntax and routing. Request for Comments (RFC) 7230 (Jun 2014)
12. Gross, J.S.: An Extensible Framework for Synthesizing Efficient, Verified Parsers. Master's thesis, Massachusetts Institute of Technology (2015)
13. Jourdan, J.H., Pottier, F., Leroy, X.: Validating $LR(1)$ parsers. In: Proc. 21st European Symposium on Programming (ESOP). pp. 397–416. No. 7211 in Lecture Notes in Computer Science (2012)
14. Kaufmann, M., Moore, J.S.: The ACL2 theorem prover: Web page, http://www.cs.utexas.edu/users/moore/acl2
15. Kaufmann, M., Moore, J.S.: A precise description of the ACL2 logic. Tech. rep., Department of Computer Sciences, University of Texas at Austin (1998)
16. Koprowski, A., Binsztok, H.: TRX: A formally verified parser interpreter. Logical Methods in Computer Science **7**(2), 1–26 (2011)
17. Kumar, R., Myreen, M.O., Norrish, M., Owens, S.: CakeML: A verified implementation of ML. In: Proc. 41st ACM SIGPLAN Symposium on Principles of Programming Languages (POPL). pp. 179–191 (2014)
18. Kyzivat, P.: Case-sensitive string support in ABNF. Request for Comments (RFC) 7405 (Dec 2014)
19. Linhart, C., Klein, A., Heled, R., Orrin, S.: HTTP request smuggling. White paper, Watchfire (2005)
20. Nipkow, T.: Verified lexical analysis. In: Proc. 11th International Conference on Theorem Proving in Higher-Order Logics (TPHOL). pp. 1–15 (1998)
21. Parr, T., Fisher, K.: *LL(*)*: The foundation of the ANTLR parser generator. In: Proc. 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). pp. 425–436 (2011)
22. Ridge, T.: Simple, functional, sound and complete combinator parsing for all context-free grammars, using an oracle. In: Proc. 7th International Conference on Software Language Engineering (SLE). pp. 261–281. No. 8706 in Lecture Notes in Computer Science (2014)
23. Swords, S., Davis, J.: Fix your types. In: Proc. 13th International Workshop on the ACL2 Theorem Prover and Its Applications (2015)
24. The ACL2 Community: The ACL2 theorem prover and community books: Source code, http://github.com/acl2/acl2
25. The ACL2 Community: The ACL2 theorem prover and community books: User manual, http://www.cs.utexas.edu/~moore/acl2/manuals/current/manual
26. Thomas, L.D.: APG: ABNF Parser Generator, http://www.coasttocoastresearch.com
27. Wisnesky, R., Malecha, G., Morrisett, G.: Certified web services in Ynot. In: Proc. 5th International Workshop on Automated Specification and Verification of Web Systems (WWV). pp. 5–19 (2009)