BNF-Style Notation as It Is Actually Used

BNF-Style Notation as it is Actually Used

D. Quinlan, J. B. Wells, and F. Kamareddine

Heriot-Watt University, Edinburgh

Abstract. The famous BNF grammar notation, as introduced and used in the Algol 60 report, was subsequently followed by numerous notational variants (EBNF, ABNF, RBNF, etc.), and later by a new formal "grammars" metalanguage used for discussing structured objects in Computer Science and Mathematical Logic. We refer to this latter offspring of BNF as *MBNF* (Math-BNF), and to aspects common to MBNF, BNF, and its notational variants as *BNF-style*. MBNF is sometimes called "abstract syntax", but we avoid that name because MBNF is in fact a concrete form and there is a more abstract form. What all BNF-style notations share is the use of production rules like (P) below which state that "every instance of $\circ_i$ for $i \in \{1, \ldots, n\}$ is also an instance of $\bullet$".

$$\bullet ::= \circ_1 \mid \cdots \mid \circ_n \tag{P}$$

However, MBNF is distinct from all variants of BNF in the entities and operations it allows. Instead of strings, MBNF builds arrangements of symbols that we call math-text and allows "syntax" to be built by interleaving MBNF production rules and other mathematical definitions that can contain chunks of math-text. The differences between BNF (or its variant forms) and MBNF have not been clearly presented before. (There is also no clear definition of MBNF anywhere but this is beyond the scope of this paper.)

This paper reviews BNF and some of its variant forms as well as MBNF, highlighting the differences between BNF (including its variant forms) and MBNF. We show via a series of detailed examples that MBNF, while superficially similar to BNF, differs substantially from BNF and its variants in how it is written, the operations it allows, and the sets of entities it defines. We also argue that the entities MBNF handles may extend far outside the scope of rewriting relations on strings and syntax trees derived from such rewriting sequences, which are often used to express the meaning of BNF and its notational variants.

1 Introduction

In this paper we discuss a form of BNF-style notation which is sometimes called abstract syntax, but which we term Math-BNF (hereafter MBNF), because MBNF is in fact a concrete form and there is a more abstract form.[1] MBNF is important for interpreting papers in theoretical computer science. Out of the 30 papers in the ESOP 2012

[1] For example, consider an abstract syntax tree (AST) representing $\lambda x.e$. An AST is a tree where each branch goes to a syntactic evaluation of a metavariable and each node is either a metavariable assignment which contains no further evaluations or a function taking metavariables, which represents an evaluation. In an AST for $\lambda x.e$, we would not be interested that the x and the e are arranged with a dot between them and a λ in front of them. Rather, $(\lambda\Box.\Box)$ would just be a name for a particular function taking two arguments of an appropriate type.

proceedings [30], 19 used MBNF and none used BNF.[2] Section 2 covers existing definitions for BNF as well as some formally defined standards which extend it into BNF variants. It also covers some limitations of BNF and its notational variants, which drive computer scientists to use what we term MBNF, despite the fact that MBNF lacks a formal definition. Section 3 discusses how MBNF differs from BNF and its variants.

First we introduce some notational conventions. Since the text of other documents' meta-levels is part of the object level of this one, we introduce the following notation. We use "boxes" for both `inline` and block text.

> “ Text placed in a quotebox (aside from this one) is quoted directly from another document. ”

Text placed in an undecorated box (aside from this one) is intended to imitate the text of other documents which we may look to deal with, and is usually derived from things written elsewhere, but it is not a direct quote.

2 BNF and its Variants

We give a brief overview of BNF and its more popular variants which remain broad enough to cover how BNF variants normally work, the kinds of entity they work with and the kinds of operations they usually allow. According to Zaytev [35]:
"The grammarware technological space is commonly perceived as mature and drained of any scientific challenge, but provides many unsolved problems for researchers who are active in that field."
While we agree with Zaytev that there are still numerous reasons for a deeper comparison of these relatives to BNF, which might touch on more obscure examples and order them in terms of how quickly they may be used to build syntax or their expressive power, this is not what we aim to provide here. The main purpose of this section is as a preliminary to our examination of MBNF. The idea we intend readers to take away from this section is that, while the notations examined here allow for a great deal of variation between them, they are still more like one another than they are like MBNF.

2.1 How BNF Works

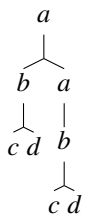
BNF can be thought of as a game: you start with a non-terminal and are then given rules for what you can replace this with. The value of the non-terminal defined by the BNF grammar is just the set of all things you can produce by applying these rules to each non-terminal provided you reach a string entirely composed of terminal symbols after a finite number of steps.

[2] We chose ESOP 2012, but we could equally pick any other conferences. Because the first book we picked contained an abundance of challenging instances of MBNF, our wider searching has mainly been to find even more challenging examples. We will be happy to receive pointers to additional interesting cases. We also checked the POPL 2017 proceedings [11] and found that out of 46 papers using BNF-style notation, not one used notation exactly corresponding to the EBNF [20], ABNF [6] or RBNF [10] standards and only one [16] could possibly be thought of as EBNF or ABNF with variant syntax. Out of the other 45 POPL 2017 papers featuring BNF-style notation, 44 use what we call MBNF.

The rules are called production rules, and normally look like this:

$$\bullet ::= \circ_1 \mid \cdots \mid \circ_n$$

A production rule simply states that the symbol on the left-hand side of the ::= must be replaced by one of the alternatives on the right hand side. For example the non-terminal $\langle a \rangle$ in $\langle a \rangle ::= \langle b \rangle \mid \langle b \rangle\langle a \rangle$ would range over things of the form $\langle b \rangle$, $\langle b \rangle\langle b \rangle$, $\langle b \rangle\langle b \rangle\langle b \rangle$ etc. If we were also given $\langle b \rangle ::= cd$, then it would range over cd, $cdcd$, etc. The alternatives are separated by |. Alternatives usually consist of "non-terminals" and "terminals". Terminals are simply pieces of the final string that are not "non-terminals". They are called terminals because there are no production rules for them: they terminate the production process. We can write a tree (sometimes called an abstract syntax tree) to show how BNF style notation produces syntax as an instance of a non-terminal (where each child node is an instance of its parent). Here is how we would write $cdcd$ as an instance of a given the rules $\langle a \rangle ::= \langle b \rangle \mid \langle b \rangle\langle a \rangle$ and $\langle b \rangle ::= cd$:



Non-terminals are distinguished from terminals either by placing them in triangular brackets or by surrounding terminals by quotes and using either a comma or a space to separate both non-terminals and terminals. The language of $\bullet$ is the set of all things of the form $\circ_i$ for $1 < i < n$. In the example where $a ::= b \mid ba$ and $b ::= cd$ the language of a would be $\{ (cd)^n \mid n \in \mathbb{N} \wedge n > 0 \}$ where $\mathbb{N}$ is the set of natural numbers and $(cd)^n$ denotes something of the form cd concatenated with itself n times.

2.2 Backus and Naur

To illustrate what the original BNF looked like we present an example of BNF as it was used by Backus and BNF as it was used by Naur.

Backus [1, p. 129] used ":≡" to symbolise a production and "$\overline{or}$" to separate production rules. He picked out non-terminals by surrounding them with angle brackets.

> "
> $$\begin{aligned} \langle digit \rangle &:\equiv O \,\overline{or}\, 1 \,\overline{or}\, 2 \,\overline{or}\, 3 \,\overline{or}\, 4 \,\overline{or}\, 5 \,\overline{or}\, 6 \,\overline{or}\, \\ &\quad 7 \,\overline{or}\, 8 \,\overline{or}\, 9 \\ \langle integer \rangle &:\equiv \langle digit \rangle \,\overline{or}\, \langle integer \rangle\langle digit \rangle \end{aligned}$$
> "

Here, $\langle digit \rangle$ ranges over the set of symbols {"0, "1", "2", "3", "4", "5", "6", "7", "8", "9"}. $\langle integer \rangle$ ranges over the set of strings one would use to write the non-negative integers using digits 0 to 9.

Naur [2, p. 3,5] used "::=" instead of ":≡" and "|" instead of "$\overline{or}$". Other than that the grammar is the same.

“
$$\langle digit\rangle ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$
$$\langle unsigned\ integer\rangle ::= \langle digit\rangle \mid \langle unsigned\ integer\rangle\langle digit\rangle$$
”

Here $\langle digit\rangle$ ranges over the same set $\langle digit\rangle$ ranged over in the previous example and $\langle unsigned\ integer\rangle$ ranges over the same set $\langle integer\rangle$ did.

2.3 Extensions to BNF

The following are extensions to BNF, which, unlike MBNF have a formal definition.

EBNF (Extended Backus-Naur Form) adds facilities for dealing with repetition of syntactic rules (braces around repeated text), special sequences (Two ?s around names of special characters), optional choice of syntactic rules (square brackets around optional text) and exceptions to syntactic rules (written $R - E$ where R is a rule and E an exception). Instead of having non-terminal symbols surrounded by angle brackets, terminal symbols are surrounded by single quotes and all symbols are separated by commas. Each line is ended in a semicolon. A full copy of the syntax for EBNF is found in [20].

In EBNF, the terminating decimals *DI* can be written as:

$$DI ::= [\text{`}-\text{'}], D, \{D\}, [\text{`.'}, D, \{D\}];$$
$$D ::= \text{`0'} \mid \text{`1'} \mid \text{`2'} \mid \text{`3'} \mid \text{`4'} \mid \text{`5'} \mid \text{`6'} \mid \text{`7'} \mid \text{`8'} \mid \text{`9'};$$

We read the rule $DI ::= [\text{`}-\text{'}], D, \{D\}, [\text{`.'}, D, \{D\}];$ as giving the following instructions for producing something of the form *DI*: First, begin with an optional minus, $[\text{`}-\text{'}]$, followed by a choice of $D$, $D$, followed by any number of choices of $D$, $\{D\}$, followed by an optional choice of a member of a group, $[\text{`.'}, D, \{D\}]$. This group consists of things produced with the following instructions: begin with a dot, `.', followed by a choice of D, D, followed by any number of choices of D, $\{D\}$.

EBNF without exceptions to syntactic rules is not more powerful than BNF in terms of what sets of strings it can generate, but it is more convenient and the parse trees it generates may look different. The above example is more cumbersome in BNF:

$$\langle DI\rangle ::= \langle PD\rangle \mid -\langle PD\rangle \mid \langle PD\rangle.\langle PD\rangle \mid -\langle PD\rangle.\langle PD\rangle$$
$$\langle PD\rangle ::= \langle D\rangle \mid \langle PD\rangle\langle D\rangle$$
$$D ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

Lemma 1. *Repetition and choice can be written into equivalent BNF productions.*
Proof. Let ε stands for the empty string. We outline the process:

- Convert every repetition { E } to a fresh non-terminal X and add X = ε | X E.
- Convert every option [E] to a fresh non-terminal X and add X = ε | E.
  (We can convert X = A [E] B. to X = A E B | A B.)
- Convert every group (E) to a fresh non-terminal X and add X = E.

⬡

Exceptions to syntactic rules mean EBNF is not context-free.

Lemma 2. *Production rules of the form $R - E$ have no BNF equivalent.*

Proof. Let ε stands for the empty string. First consider intersection

1. The language of $L_1 = \{a^n b^n a^m \mid n, m \geq 0\}$ is generated by:
   $\langle L_1\rangle ::= \langle X\rangle\langle A\rangle \quad \langle X\rangle ::= a\langle X\rangle b \mid \varepsilon \quad \langle A\rangle ::= \langle A\rangle a \mid \varepsilon$
2. The language of $L_2 = \{a^n b^m a^m \mid n, m \geq 0\}$ is generated by: $\langle L_2\rangle ::= \langle A\rangle\langle X\rangle$
3. $L_1 \cap L_2 = \{a^n b^n a^n \mid n \geq 0\}$ is not context free by the pumping lemma [3, p. 110] since, for a given $p \geq 1$ we can choose $n > p$ such that $s = a^n b^n a^n$ is in our language and we cannot pick any substring q of s such that q is of length p, $s = xqy$ for some strings x and y, and $xq^n y \in L_1 \cap L_2$ (since q would have to take in at least one a and should take in the same number of as on the left as on the right).

It follows easily that rules of the form $R - E$ cannot be modelled in BNF, since BNF only generates the context free grammars and $\{a^n b^n a^n \mid n \geq 0\} = L_1 - (L_1 - L_2)$ ⬡

Lemma 2 shows that some things represented in EBNF cannot be represented in BNF.

P.E.G.s (Parsing Expression Grammars) [12] have many of the same facilities as EBNF, but contain an ordered choice operator which indicates parsing preference between options. For example, the EBNF rules $A = a, b \mid a$ and $A = a \mid a, b$ are equivalent, but the P.E.G. rules $A \leftarrow a\,b \;/\; a$ and $A \leftarrow a \;/\; a\,b$ are different. The second alternative in the latter P.E.G. rule will never succeed because the first choice is always taken if the input string to be recognized begins with 'a'. P.E.G. rules also add and, '&', and not, '!', syntactic predicates which match a pattern only if a certain context is present. The expression '&e' attempts to match pattern e, then unconditionally backtracks to the starting point, preserving only the knowledge of whether e succeeded or failed to match. Conversely, the expression '!e' fails if e succeeds, but succeeds if e fails. E.g. `!EndOfLine .` matches any single character so long as the nonterminal EndOfLine does not match starting at the same position. P.E.G. provides us with slightly more power than EBNF. However, understanding P.E.G. rules rests on the user's intuitive understanding of parsing and string recognition - the body of literature for which is very large. In addition there is not a particularly close correspondence between the extra operators provided by P.E.G. and anything in the syntax of MBNF.

ABNF (Augmented Backus-Naur Form) [6] contains no facilities not also included in ENBF (which ISO gives as the standard for BNF itself at the time of writing). We include it here only for completeness.

RBNF (Routing Backus-Naur Form) [10] Contains most of the facilities of EBNF aside from the ability to write exceptions to syntactic rules. RBNF can generate the same syntax as BNF. We include it here only for completeness.

LBNF (Labelled BNF) [13] extends EBNF with functionality for dealing with polymorphic lists of rules. It also provides a few pre-defined sets such as characters, integers, strings, and identifiers. It also provides labels which deal with higher order abstract syntax [26], however this is not intended to be used in LBNF grammars written by hand, but in ones generated from the grammar formalism GF (Grammatical Framework) [28].

Again there is no clear mathematical model of this functionality to aid human understanding; analysing the given functions requires understanding the programs used to compile expressions in the grammar. In addition there is not a particularly close correspondence between the labels provided by LBNF and anything in the syntax of MBNF.

TBNF (Translational Backus-Naur Form) [24] Puts non terminals in the place of internal nodes and terminals in place of external nodes on a tree (which we call an abstract syntax tree or AST). When the resulting syntax is parsed, the AST it creates is traversed. It adds to EBNF additional production arrows as follows:

- **[~>]** Reverse Production Arrow. An arrow preceding the right side of a rule for which you want the nodes to be arranged in reverse order.
- **[=>]** Call A Function. This means to call a function when a rule in the grammar has been recognized. A rule in the grammar may have multiple function calls.
- **[+>]** Make A Node. This means to make a node corresponding to this rule in the grammar. During AST traversal this node will be processed with a built-in default node processing function.
- **[*>]** Make A Node and during AST traversal, call a function with the same name as this node, instead of calling the default node-processing function. This allows customization of the code-generation process.
- **[$1]** Parse Stack Position. This refers to the symbol in parse stack position # 1, the first symbol in the right side of the rule. $n refers to the *n*th position.
- **[..*]** Node Traversal Indicator. Indicates when processing for this node should occur, at top-down, pass-over, or bottom-up time, respectively. *.. indicates top-down only. ..* indicates bottom-up only. *.* indicates top-down and bottom-up.
- **[(...)]** The Arguments. Arguments are used for function calls (=>) and node processing (+>). For node processing the arguments apply to the relative '*' in the Node Traversal Indicator. *-* would require two string arguments.
- **[& 0]** Counter Indicator. When the AST processor enters a node, it increments a counter for the node and puts in on a stack. The '& 0' indicates the current count for the node taken from the stack. A '& 1' means the counter for the parent node on the stack and '& 2' means the counter of the grandparent. This provides a unique number for labels.

TBNF provides a very rich extension to EBNF which is particularly well suited for relating expressions to their abstract syntax trees. However a clear mathematical model of the trees generated by TBNF is not provided alongside its syntax (again these are created in a compiler rather than presented in a form intuitive to human beings). In addition while it covers some of the functionality authors expect when they use MBNF it does not particularly resemble the way in which it is written.

2.4 Limitations of BNF and its Variants

When deciding if a BNF-like syntax can be readily converted into BNF, it is important to note that BNF is a language for building sets of strings and the only notion of equality it deals with easily is string equality. It is possible to derive a notion of tree equality from the parse trees generated by an EBNF grammar, but there is no guarantee that parse trees will be unambiguous. Unless an author writes their grammar with a parser in mind, inferring a sensible parse tree from a set of productions is non-trivial.

BNF can describe exactly the context-free grammars in Chomsky's hierarchy [5]. Non-context-free grammars cannot be written without extending BNF in some way.

3 MBNF

In this section, we highlight ways in which the notation we call MBNF differs from BNF and its variants. We show that MBNF is non-trivially different from existing extensions of BNF and does not deal with the same kinds of entities as these extensions.

3.1 Where BNF and its Variants use Strings, MBNF Uses Math-Text

Parentheses for disambiguation are not needed in MBNF grammars and when an MBNF grammar specifies such parentheses they can often be omitted without any need to explain. E.g. writing $(\lambda x.xx)\lambda y.y$ instead of $((\lambda x.(xx))(\lambda y.y))$. When possible, MBNF takes advantage of the tree-like structure implicit in the layout of symbols on the page when features like superscripting and overbarring are used. E.g. in $\overline{f_x^{n+1} + \overline{y \cdot fj}}$.

Instead of non-terminal symbols, MBNF uses *metavariables*[3], which appear in what we call *math-text* and obey the conventions of mathematical variables. Metavariables are not distinguished from other symbols by annotating them as BNF and its notational variants do, but by math-text features such as font, spacing, or merely tradition.

In addition to arranging symbols from left to right on the page, math-text allows subscripting, superscripting, pre-subscripting, pre-superscripting and placing text above or below other text. It also allows for marking whole segments of text, for example with an overbar (a vinculum). Readers can find more detailed information on how math-text can be laid out in The TeXbook [22], or the Presentation MathML [19] and OpenDocument [21] standards. Here is a nonsense piece of Math-text to show how it may be laid out:

$$ {}^{c}\!\downarrow a' = \check{p}\langle v_x'' \odot a^{2+1}\rangle - \overline{\overline{f_x^n + \overline{y \cdot fj}}} + \sum_{i=0}^{\infty} s_{i\in 1\ldots n} \xrightarrow{a,b,c} b\hat{a} $$

3.2 MBNF Is Aimed Exclusively at Human Readers

MBNF can be used to write all of the grammars BNF and its variants can produce, but it also defines grammars BNF does not. However, unlike BNF, MBNF is meant to be interpreted by humans, not computers, as it has not been adequately formalised yet. Authors may define an MBNF grammar in an article for humans and a separate grammar for use with a parser generator to build a corresponding implementation. MBNF defines entities not intended or expected to be serialized or parsed. Dolan and Mycroft provide a typical example in [7, pp. 61–65]

> “
> $$
> \begin{array}{rcl}
> e & ::= & x \mid \lambda x.e \mid e_1 e_2 \mid \{\ell_1 = e_1, ..., \ell_n = e_n\} \mid e.\ell \mid \mathit{true} \mid \mathit{false} \mid \mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3 \\
> & & \mid \hat{\mathbf{x}} \mid \mathit{let}\ \ \hat{\mathbf{x}} = e_1\ \mathit{in}\ e_2 \\
> \Gamma & ::= & \epsilon \mid \Gamma, x : \tau \mid \Gamma, \hat{\mathbf{x}} : \forall \boldsymbol{\alpha}.\tau \\
> \tau & ::= & \mathit{bool} \mid \tau_1 \to \tau_2 \mid \{\ell_1 : \tau_1, ..., \ell_n : \tau_n\} \mid \alpha \mid \top \mid \bot \mid \tau \sqcup \tau \mid \tau \sqcap \tau \\
> \Delta & ::= & \epsilon \mid \Delta, x : \tau \\
> \Pi & ::= & \epsilon \mid \Pi, \hat{\mathbf{x}} : [\Delta]\tau
> \end{array}
> $$
> ”

In this example Γ, Δ and Π are never intended to be serialised. The authors provide an implementation in OCaml which looks very little like the above syntax.[4]

[3] Here a metavariable is a variable at the meta-level which denotes something at an object-level.
[4] www.cl.cam.ac.uk/ sd601/mlsub/

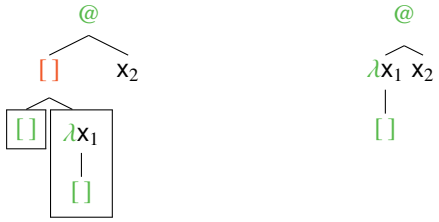
Most MBNF grammars are missing features needed to disambiguate complex terms (e.g. notation for separating metavariables from concrete syntax and from other kinds of evaluated syntax (like ⟨ and ⟩ do in BNF), bracketing (as covered in section 3.1) and notation for declaring operator precedence (for the example above [7], no rules are given for the order in which patterns should be matched). Papers often put complicated uses of the mathematical metalanguage inside MBNF notation (examples of this can be found throughout this section).

3.3 MBNF Allows Powerful Operators Like Context Hole Filling (a.k.a. Tree Splicing)

Chang and Felleisen [4, p 134] present an MBNF grammar defining the λ-term contexts with one hole where the spine[5] is a balanced segment[6] ending in a hole. For explanatory purposes, we write $e@e$ instead of $e\,e$ and add parentheses. Concrete syntax and BNF-style notation are green. Metavariables are blue. Tree-splicing operators are red.

$$e ::= x \mid (\lambda x.e) \mid (e@e)$$
$$A ::= [\,] \mid (A[(\lambda x.A)]@e)$$

One can think of the context hole filling operation ($[\,]$ in $(A[(\lambda x.A)]@e)$) as performing tree splicing operations within the syntax. Here are trees illustrating steps in building syntax trees for A (the operation $\bullet@\bullet$ is higher up the tree than $\bullet[\,\bullet\,]$ because of parsing precedence inherited from math):



These trees show the result of the second rule where each A is $[\,]$ and e is a variable. The tree on the left is the tree corresponding to $A[\lambda x.A]@e$ before the hole filling operation is performed, where the first A is assigned $[\,]$. The tree on the right represents an unparsing of the typical syntax tree for $((\lambda x_1.[\,])@x_2)$. x_1 and x_2 are disambiguated instances of x. A metavariable assigned a value won't appear in the final tree. If it's not a terminal node, $[\,]$ tells us to fill in the leaf in the frame on the left with the tree in the frame on the right. Once performed, $[\,]$ disappears.

We can show that unlike BNF, the "language" of the metavariable/non-terminal A (the set of strings derived from A using roughly the rules of BNF plus hole filling) is not context-free and so MBNF certainly isn't.

[5] The root node is on the spine. If A is applied to B by an application on the spine, the root node of A is on the spine and the root node of B is not. If a node on the spine is an abstraction each of its children is on the spine (i.e., every node appearing on the furthest left hand side of the tree is on the spine).

[6] A balanced segment is one where each application has a matching abstraction and where each application/abstraction pair contains a balanced segment.

For this section we use ToStr informally to mean a function which takes an object that an MBNF metavariable may range over, provided it can be written as a chunk of math-text whose only operation is concatenation and whose only equivalence is syntactic equivalence, and takes them to the fully parenthesised string one might use to refer to them. For example if $O = (\lambda a.[\,])@b$ declares the object O, then $\mathsf{ToStr}(O)$ = "$((\lambda a.[\,])@b)$" where "$((\lambda a.[\,])@b)$" is the symbol "(" concatenated with another "(" concatenated with "λ" etc. and O is the data structure $(\lambda a.[\,])@b$ represents. In order to show that MBNF is not context free this we make use of the pumping lemma for context-free languages [3, p. 110].

Lemma 3. Hole Filling is Not "Context-free" *For A given by the MBNF above, the language given by $\{ x \in \mathsf{String} \mid x = \mathsf{ToStr}(A) \}$ is not context-free.*

Proof Sketch. We need to show that for any given $p \geq 1$ we can produce an $s = \mathsf{ToStr}(A)$ such that no substring of s can be "pumped" (some non-empty part of one or both of its outermost substrings repeated) to give another string in the language $\{ x \in \mathsf{String} \mid x = \mathsf{ToStr}(A) \}$.

Since each A has a balanced segment along the spine we must be expected to keep count of both abstractions and applications. Parentheses must also be balanced. Getting the desired result is as simple as picking an $s = \mathsf{ToStr}(A)$ such that the abstraction at the bottom of the spine of A is more than p abstractions away from the application closest to the bottom of the spine of A and such that A contains no e long enough to be pumped. Since parentheses are balanced, the only possible section we might pick to "pump" is around the hole and since there are p abstractions before we reach an application, there is no way that "pumping" this could give us a balanced segment. ⬡

BNF itself is context free. EBNF [20], ABNF [6] and RBNF [10] don't use tree-splicing or context hole filling.

3.4 MBNF Mixes Mathematical Language With BNF-Style Notation

Germane and Might [14, p. 20] mix BNF-style notation freely with mathematical notation in such a way that the resulting grammar relies upon both sets produced with set theory notation and MBNF production rules which use metavariables defined using mathematical notation:

"
$$
\begin{array}{rcl@{\qquad}rcl}
u \in \mathit{UVar} & = & \text{a set of identifiers} & \mathit{ccall} \in \mathit{CCall} & ::= & (q\, e^*)_\gamma \\
k \in \mathit{CVar} & = & \text{a set of identifiers} & e, f \in \mathit{UExp} & = & \mathit{UVar} + \mathit{ULam} \\
\mathit{lam} \in \mathit{Lam} & = & \mathit{ULam} + \mathit{CLam} & q \in \mathit{CExp} & = & \mathit{CVar} + \mathit{CLam} \\
\mathit{ulam} \in \mathit{ULam} & ::= & (\lambda e(u^* k)\mathit{call}) & \ell \in \mathit{ULab} & = & \text{a set of labels} \\
\mathit{clam} \in \mathit{CLam} & ::= & (\lambda_\gamma(u^*)\mathit{call}) & \gamma \in \mathit{CLab} & = & \text{a set of labels} \\
\mathit{call} \in \mathit{Call} & = & \mathit{UCall} + \mathit{CCall} & \mathit{ucall} \in \mathit{UCall} & ::= & (f e^* q)_\ell
\end{array}
$$
"

The results of math computations are interleaved with MBNF production rules, not just applied after the results of the production rules have been obtained. This grammar uses $\bullet_1 \in \bullet_2$ to mean "$\bullet_2$ is the language of $\bullet_1$" (this is the case in both the MBNF production rules (::=) and the math itself (=)).

In the MBNF above + is used as union and we have the additional requirement that there must exist some procedure for choosing sets fulfilling these constraints such that,

if, for some terms X and Y, $X+Y$ appears in the grammar, then X and Y do not intersect. Here, the requirement is most likely fulfilled by the author following the convention that any arbitrary sets declared separately are disjoint (i.e., $CVar \cap UVar = \emptyset$). However, in order to check that grammars like the one above are correct, we would still need a general procedure for checking that $ULam$ and $CLam$ don't overlap if such a convention is chosen.

BNF, EBNF, ABNF and RBNF don't have a concept of disjoint union and don't allow one to interleave set theoretic operations on the language of a non-terminal with ordinary BNF definitions.

3.5 MBNF Has at Least the Power of Indexed Grammars

Inoe and Taha [18, p. 361] use this MBNF:

> “ $$\mathcal{E}^{\ell,m} \in ECtx^{\ell,m}_{\mathbf{n}} ::= \cdots \mid \langle \mathcal{E}^{\ell+1,m} \rangle \mid \cdots$$ ”

This suggests that MBNF deals with the family of indexed grammars [17, p 389-390], which is yet another reason it's not context-free. The $\ell + 1$ is a calculation that is not intended to be part of the syntax. The production rule above defines an infinite set of metavariables ranging over different sets.

BNF, EBNF, ABNF and RBNF don't allow for indexing.

3.6 MBNF Has a Native Concept of Binding

In Germane and Might [14, p. 20] we found the following:

$$pr \in Pr = \{ulam : ulam \in Ulam, \text{closed}(ulam)\}$$

In order to perform this evaluation of the set $Ulam$ we must recognise which variables are bound.

In addition we need a notion of binding to deal with some of the issues surrounding α-equivalence that often arise when authors start working with the grammar they define as part of a reduction system. Chang and Felleisen [4, p 134] give the following axiom:

$$\hat{A}[A_1[\lambda x.\check{A}[E[x]]]A_2[v]] = \hat{A}[A_1[A_2[\check{A}[E[x]]x := v]]] \qquad \text{where } \hat{A}[\check{A}] \in A$$

Here we are meant to recognise an implicit convention, known as the Barendregt convention, on the terms we are β-reducing over. In this case the Barendregt convention would require that we pick terms from the α-equivalence class of $\hat{A}[A_1[\lambda x.\check{A}[E[x]]]A_2[v]]$ such that no bound variable of $A_1[\lambda x.\check{A}[E[x]]]$ is a free variable in $A_2[v]$ and none of the bound variables in $A_2[v]$ are free variables in $\check{A}[E[x]]$.[7] Since Chang and Felleisen also expect the Church-Rosser property to hold of their reduction relations, terms are identified up to α-equivalence again after performing the reduction and filling the holes.

A notion of binding is not native to BNF, EBNF, ABNF, RBNF or TBNF but must be defined after the grammar.

3.7 MBNF Allows “Arbitrary” Side Conditions on Production Rules

An example of a production rule with a side condition can be found in Chang and Felleisen [4, p 134]:

[7] Actually a slightly weaker condition than the one we give here is probably sufficient for the Barendregt convention to hold, but it would be more complicated to state.

> “ $E = [\,] \mid Ee \mid A[E] \mid \hat{A}[A[\lambda x.\check{A}[E[x]]]E]$ where $\hat{A}[\check{A}] \in A$ ”

It is possible to make side conditions that prevent MBNF rules from having a solution. A definition for MBNF can help in finding conditions on side conditions that ensure MBNF rules actually define something.

Potential Contradictions When Dealing With Side Conditions Side conditions may cause problems in making sure an MBNF is well defined. We offer a set of assumptions about what one may be allowed to do in an MBNF which are separately plausible and unproblematic, but which allow us to create a grammar which is obviously undefined if we use all of them unrestrictedly.

Where we are allowed to use $\in$ we are usually allowed to use $\notin$. The side condition of the MBNF for E has a metavariable that is created by filling a hole in a member of $\hat{A}$ with a member of $\check{A}$. This suggests that we may be allowed to use mathematical productions similar to those used in the production rules themselves to create the metavariables featuring in the side conditions of the production rules. We may conclude that, provided we have a production rule for B, we can have $\lambda x.B$ in one of our side conditions. MBNF also allows us to have production rules that reference themselves, either directly or indirectly. By allowing all these assumptions, we can produce an MBNF which defines a non existent language.

$$\begin{array}{lll} A ::= x \mid \lambda x.A \mid B & & \text{Where } \lambda x.B \notin C \\ B ::= x \mid A \mid \pi x.B & & \\ C ::= A \mid B & & \end{array}$$

There exists $b \in B$ such that b is of the form $\pi x.B$, consider one such b. Suppose that $\lambda x.b \notin C$. Then $b \in A$ and since, for all $a \in A$, $\lambda x.a \in A$ we would have that $\lambda x.b \in A$ and, since $C ::= A \mid B$, $\lambda x.b \in C$. Suppose instead that $\lambda x.b \in C$, then either $\lambda x.b \in A$ or $\lambda x.b \in B$. Every statement in B is either an x, or else it begins with π, or else it is also in A, so $\lambda x.b \in A$. $\lambda x.y \in A$ if and only if $y \in A$, so $b \in A$. Since b is not of the form x or $\lambda x.A$, then b can only be produced by the production rule B, in which case $\lambda x.b \notin C$. So we have that, if $\lambda x.b \in C$ then $\lambda x.b \notin C$ and if $\lambda x.b \notin C$ then $\lambda x.b \in C$.

So there is no set of statements we can produce that satisfies the rules of the grammar. We can't isolate any particular production rule which causes the problem, each rule alone may be fine within the context of a slightly different grammar.

We believe that authors often have some heuristic in mind which allows them to avoid cross reference of the sort in our fictitious example, but do not know of a definition which explicitly says what's allowed.

Neither BNF nor its variants allow arbitrary side conditions on production rules.

3.8 MBNF Can Contain Very Large Infinite Sets as Part of the “Syntax”

Toronto and McCarthy [34, p 297] write:

> “ $e ::= \cdots \mid \langle t_{set}, \{e^{*\kappa}\}\rangle$
> Here $\{e^{*\kappa}\}$ means sets of no more than κ terms from the language of e. ”

> “ ...The language of $v ::= \langle t_{set}, \{v^{*\kappa}\}\rangle$ consists of the encodings of all the hereditarily accessible sets. ”

The author does not state what κ is, but elsewhere in the paper it is an inaccessible cardinal. It seems as though κ is also intended to be an inaccessible cardinal here.

BNF and its notational variants, by contrast, only deal with strings of finite length.

3.9 MBNF Allows Infinitary Operators

Díaz and Núñez [23, p. 539] write an MBNF with an infinitary operator:

> “ $$P ::= \cdots \mid \prod_{i\in I} P_i \mid \cdots$$
>
> ...But, for instance in our language we have the term
>
> $$\prod_{n\in\mathbb{N}} P_n$$
>
> where each P_n is born at time n, and so P is born at time $\omega + 1$.
>
> ...So, to fully formalize the set of valid expressions, we begin by bounding the size of the possible sets of indices I, and that of the set of actions Act by some infinite cardinal κ. The functional governing the right hand side of the equation is clearly monotone, but it is not so obvious whether it has any fixpoint. Fortunately it has. Besides, it is guaranteed that it is reached after (at most) λ iterations, where λ is the smallest regular cardinal bigger than κ. Then, the principle of structural induction is valid and corresponds to the principle of transfinite induction. ”

We may think of infinitary operators as defining trees of infinite breadth (i.e., trees whose internal nodes may have infinitely many direct children), where BNF and its notational variants deal with finite data structures (usually strings).

3.10 MBNF Allows Co-Inductive Definitions

Eberhart, Hirschowitz and Seiller [8, p 94] write:

> “ We consider processes to be infinite terms as generated by the grammar:
>
> $$P, Q ::= \Sigma_{i\in n} G_i \mid (P|Q) \qquad G ::= \overline{a}\langle b\rangle.P \mid a(b).P \mid \nu a.P \mid \tau.P \mid \heartsuit.P$$
>
> up to renaming of bound variables as usual. Such a coinductive definition... ”

We may think of co-inductive definitions as allowing us to define trees of infinite depth (i.e. trees in which paths may pass through infinitely many nodes), where BNF and its notational variants deal with finite data structures.

3.11 MBNF may be Considered up to “Arbitrary” Equivalences

As well as α-equivalence and binding, the objects created by a piece of MBNF may be considered up to various other equivalences, e.g., associativity and composition with a

0 element (as in the π-calculus [25]), equivalence up to the exchanging of labels (as in the λ-calculus with records [27, p. 129]), equivalence up to repetition of elements (as with set-like syntactic objects), and additional equivalences which may be defined by the author. Tobisawa [33, p. 386] defines equivalences $\simeq_s$ & $\simeq_t$:

“
$$\begin{aligned} \mathrm{id}\langle v, d\rangle &:= v^d[\mathrm{id}], \\ ({}^w\!\downarrow(M)\cdot\sigma)\langle v, d\rangle &:= \begin{cases} M & \text{If } v = w \text{ and } d = 0, \\ \sigma\langle v, d - \delta_{vw}\rangle & \text{otherwise,} \end{cases} \\ (\uparrow_w \cdot\sigma)\langle v, d\rangle &:= \sigma\langle v, d + \delta_{vw}\rangle \end{aligned}$$
”

where δ_{vw} is the integer defined by

“
$$\delta_{vw} := \begin{cases} 1 & \text{If } v = w \\ 0 & \text{otherwise.} \end{cases}$$
”

Then $\simeq_s$ and $\simeq_t$ are defined inductively.

“
$$\begin{aligned} \mathrm{id} &\simeq_s \mathrm{id} && \\ \sigma &\simeq_s \tau && \text{if } \sigma\langle v, d\rangle \simeq_t \tau\langle v, d\rangle \text{ for any } v, d \\ v^d[\sigma] &\simeq_t v^d[\tau] && \text{if } \sigma \simeq_s \tau \\ \lambda v.M &\simeq_t \lambda v.N && \text{if } M \simeq_t N \\ M_1 \ @_\ell \ M_2 &\simeq_t N_1 @_\ell N_2 && \text{if } M_1 \simeq_t N_1 \text{ and } M_2 \simeq_t N_2 \end{aligned}$$
”

However in order to work modulo these equivalences, he must also be working modulo arithmetic equivalence on some computations. He must also be working with the implicit assumption that := is a bijective transformation, that maps syntax to syntax preserving equivalence. Let $f : A \mapsto B$ be a function from some subset of the terms defined by the BNF, A, to some other set of terms, B, which are possibly meant to be definable with a different BNF (with side conditions and arithmetic computations), such that $\forall a \in A$; $a := f(a)$ and if $a := b$, then $b = f(a)$. The author wishes us to use a convention where, two terms $a, b \in A$ are equivalent in the paper if and only if $f(a), f(b) \in B$ are equivalent.

A sufficiently general notion of equivalence is not native to BNF and its notational variants but must be defined after the grammar.

4 Related Work

There has already been some work done in the area of defining MBNF, however, to our knowledge, no other authors have highlighted all the issues we have, or presented it as a significant departure from BNF and its notational variants. In fact, MBNF is rarely even given a name to distinguish it from similar notations, on the few occasions authors do refer to it they call it “abstract syntax,” which is misleading. We have had to coin the term MBNF to make it clear what we are talking about.

We take a look at some of the existing work related to the definition of MBNF and talk about why this paper exposes important issues which existing work does not address. Since MBNF has not yet been properly recognised as a notation distinct from

BNF and its extensions, which is in need of a definition, we have chosen papers dealing with a broad set of different problems. Some of this work deals with syntactic structures which are in some way related to the syntactic structures used by MBNF. Some of this work focusses on comparison of BNF-style notations (which may or may not include MBNF), but does not focus on the issues we do. One of the pieces we cite evaluates non-MBNF syntax, but allows some functionality more commonly associated with MBNF and produces something like MBNF as output.

Ott [31] provides a formal language for writing specifications like those written in MBNF. The process of moving from an Ott specification to an MBNF can be performed automatically. However, Ott does not offer support for interpreting MBNF without requiring it to be specified in a theorem-prover friendly format. Ott focuses on translating to Coq 8.3, HOL 4 and Isabelle directly, but offers less support for those seeking a general mathematical intuition. Ott only allows contexts with a single hole, does not allow for hole-filling operations to appear in the clause of a production rule and currently does not support rules being used coinductively. Ott also does not handle the common practice of using mathematical text outside of the MBNF grammar as part of its definition.

Steele [32] covers many of the notational variants of BNF, and some MBNF. However, he is primarily interested in making an initial attempt at documenting computer science meta-notation. He focuses on the differences between CSM and earlier versions, such as BNF, only insofar as they help this goal, and remind us of alternative choices that might have been better than the ones we ended up with. He does not discuss how the underlying mathematical structure of MBNF differs wildly from BNF and its variants.

Grewe et al. [15] discuss the exploration of language specifications with first-order theorem provers. However, they still require the reader to be able to intuitively translate language specifications to a sufficiently formal language first.

Reynolds [29, p. 1-51] has the best attempt at a definition of MBNF which we could find after looking through the books in our collection, which he calls "abstract syntax"[8]. However, he only deals with context-free grammars and usually proceeds by example.

Farmer [9] deals with syntax featuring evaluation and quotation. This resembles the kinds of entity MBNF works with. However, he is more concerned with syntax evaluation as it stands for some algorithmic operations than syntax standing for itself modulo some equivalence and he does not go into how syntax featuring evaluation and quotation is meant to be combined with BNF-style notation.

Zaytsev [35] provides a comprehensive review of BNF variants and a tool for studying mappings between them. He does not deal with MBNF at all nor does he claim to, but his work highlights in detail the similarities between BNF and its extensions touched upon in section 2 as well as other variants we did not have space to cover. A comparison of MBNF to the grammars examined in his work undelines that MBNF is not an extension of BNF with some transformation applied to its syntax.

None of the above authors deal with MBNF grammars with very large infinite things in them or with mixing math and MBNF. They do not even discuss this as a permissible practice, nor the need to treat MBNF as a notation distinct from BNF and its extensions.

[8] As previously noted we avoid that name, because MBNF is a concrete form.

5 Conclusions

While MBNF bears a superficial resemblance to BNF and its variants, we have demonstrated that it is different in the entities it works with and the operations it allows. We conclude neither BNF nor any or its variants are suitable to express the range of modern Computer Science and Mathematical Logic, since MBNF is used frequently in these fields and differs from BNF and its variants on a deep conceptual level. In particular, BNF and its variants deal with strings and the trees detailing string building operations, both of which are finite with a natural ordering. MBNF, on the other hand, deals with the richer syntactic structures used in writing mathematics, which may be infinite rather than finite and which are defined modulo some notion of equivalence, where it is largely irrelevant to understanding the syntax whether or not members of that equivalence class have a canonical order. The operations allowed in BNF and its variants are defined as standard alongside its production rules and belong to the class of string building operations. Rather than define operations in this way, MBNF adds the concept of syntax to the broader category of mathematical entities and the concept of the production rule to a broader class of mathematical operators. These concepts may then be used alongside any concept the author needs from the field of mathematics.

While some work exists which might address some aspects of MBNF, none provides a full definition. We are not aware of another work that highlights all the differences we have here, or that recognises MBNF as a significant departure from BNF, as opposed to merely a syntactic variant, so this is unsurprising. We offer this paper as a reference point for the main issues which authors aiming to define this notation need to overcome.

References

1. J. W. Backus. The syntax and semantics of the proposed international algebraic language of the zurich acm-gamm conference. In *IFIP Congress*, 1959.
2. J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, M. Woodger. Revised report on the algorithm language algol 60. *Commun. ACM*, 6(1), 1963.
3. J. Berstel, A. Lauve, C. Reutenauer, F. V. Saliola. *Combinatorics on words. Christoffel words and repetitions in words.* Providence, RI: American Mathematical Society (AMS), 2009.
4. S. Chang, M. Felleisen. The call-by-need lambda calculus, revisited. In Seidl [30].
5. N. Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2(3), 1956.
6. D. Crocker, Ed., P. Overell. Augmented bnf for syntax specifications: Abnf. Internet Requests for Comments, 2008.
7. S. Dolan, A. Mycroft. Polymorphism, subtyping, and type inference in MLsub. In Fluet [11].
8. C. Eberhart, T. Hirschowitz, T. Seiller. An Intensionally Fully-abstract Sheaf Model for π^*. In L. S. Moss, P. Sobocinski, eds., *6th Conference on Algebra and Coalgebra in Computer Science (CALCO 2015)*, vol. 35 of *Leibniz International Proceedings in Informatics (LIPIcs)*, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.
9. W. M. Farmer. The formalization of syntax-based mathematical algorithms using quotation and evaluation. *CoRR*, abs/1305.6052, 2013.
10. A. Farrel. Routing backus-naur form (rbnf): A syntax used to form encoding rules in various routing protocol specifications. RFC 5511, RFC Editor, 2009.
11. M. Fluet, ed. *POPL'17 :Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, New York, NY, USA, 2017. ACM.

12. B. Ford. Parsing expression grammars: A recognition-based syntactic foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, New York, NY, USA, 2004. ACM.
13. M. Forsberg, A. Ranta. The labelled bnf grammar formalism, 2005.
14. K. Germane, M. Might. A posteriori environment analysis with pushdown delta cfa. In Fluet [11].
15. S. Grewe, S. Erdweg, A. Pacak, M. Raulf, M. Mezini. Exploration of language specifications by compilation to first-order logic. *Sci. Comput. Program.*, 155, 2018.
16. R. Grigore. Java generics are turing complete. In Fluet [11].
17. J. E. Hopcroft, R. Motwani, J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.
18. J. Inoue, W. Taha. Reasoning about multi-stage programs. In Seidl [30].
19. P. D. F. Ion, N. Poppelier, D. Carlisle, R. R. Miner. Mathematical markup language (MathML) version 2.0. W3C recommendation, W3C, 2001. https://www.w3.org/TR/MathML2/chapter3.html.
20. ISO. Information technology – Syntactic metalanguage – Extended BNF. Standard, International Organization for Standardization, Geneva, CH, 1996.
21. ISO. Information technology – Open Document Format for Office Applications (OpenDocument) v1.2 – Part 1: OpenDocument Schema. Standard, International Organization for Standardization, Geneva, CH, 2015.
22. D. E. Knuth. *The TeXbook*. Addison-Wesley Professional, 1986.
23. L. F. Llana-Díaz, M. Núñez. Testing semantics for unbounded nondeterminism. In C. Lengauer, M. Griebl, S. Gorlatch, eds., *Euro-Par'97 Parallel Processing*, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.
24. P. B. Mann. A translational bnf grammar notation (tbnf). *SIGPLAN Not.*, 41(4), 2006.
25. R. Milner, J. Parrow, D. Walker. A calculus of mobile processes, i. *Inf. Comput.*, 100(1), 1992.
26. F. Pfenning, C. Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, New York, NY, USA, 1988. ACM.
27. B. C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002.
28. A. Ranta. Grammatical framework. *J. Funct. Program.*, 14(2), 2004.
29. J. C. Reynolds. *Theories of Programming Languages*. Cambridge University Press, New York, NY, USA, 1st edition, 2009.
30. H. Seidl, ed. *Programming Languages and Systems*. Springer, 2012.
31. P. Sewell, F. Z. Nardelli, S. Owens, G. Peskine, T. Ridge, S. Sarkar, R. Strniša. Ott: Effective tool support for the working semanticist. *SIGPLAN Not.*, 42(9), 2007.
32. G. L. Steele, Jr. It's time for a new old language. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '17, New York, NY, USA, 2017. ACM.
33. K. Tobisawa. A meta lambda calculus with cross-level computation. In *POPL '15*, 2015.
34. N. Toronto, J. McCarthy. Computing in cantor's paradise with λ zfc. In T. Schrijvers, P. Thiemann, eds., *Functional and Logic Programming*, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.
35. V. Zaytsev. The grammar hammer of 2012. *CoRR*, abs/1212.4446, 2012.