# BITES instead of FIRST for Parsing Expression Grammar*

Roman R. Redziejowski

**Abstract**

In an earlier paper, the author adapted to Parsing Expression Grammars (PEGs) the properties FIRST and FOLLOW used in the construction of predictive top-down parsers. The purpose was to obtain warnings for possible "language hiding". It turned out that FIRST does not work well with lookahead expressions. To repair this, it is replaced here by a property named BITES that is a set of input strings instead of terminals.

## 1 Introduction

Parsing Expression Grammar (PEG), introduced by Ford in [2], is a way to define recursive-descent parsers with limited backtracking. Backtracking lifts the $LL(1)$ restriction usually imposed by top-down parsers; however, the language accepted by the parser is not always obvious. One of the problems is the phenomenon of "language hiding", which is a consequence of the backtracking being limited. The phenomenon, loosely called "prefix capture" in [4], was discussed in more detail in [5]. In [7], Schmitz formulated a sufficient condition for absence of language hiding, called "general semi-disjointness". Unfortunately, its mechanical verification is not simple. There exists a much stronger sufficient condition, which is the PEG's counterpart of $LL(1)$, namely that different alternatives in a choice expression start in a different way. If we could point out violations of this condition, we could then check by individual analysis if language hiding actually takes place. In [6], trying to formulate the condition more precisely, the author adapted to the world of PEGs the properties FIRST and FOLLOW traditionally used to check for $LL(1)$ property. In the original form, FIRST is the set of possible first letters in a string derived by a symbol, and FOLLOW is the set of letters that may follow a string derived by the symbol. (See, for example, Section 4.4 of [1], or Section 6.2 of [8].) In the modified form, FIRST became the set of terminals that an expression may call to start its processing; FOLLOW became the set of expressions that may be called after a given expression.

Unfortunately, FIRST does not work well with lookahead expressions, which call a terminal to check that the input *does not* start with it. In an experiment with a real-life grammar, this gave many false alarms. This paper is an attempt to correct that fault. For a better handling of lookahead expressions, we replace FIRST by BITES: a set of input strings that an expression may "bite", that is, make some progress towards successfully consuming a nonempty string.

While the original purpose of adapting FIRST and FOLLOW was to get a better understanding of PEGs, Mizushima, Yaeda and Yamaguchi [3] found yet another application: to cut down the memory needed for packrat parsing. It seems that using BITES may improve also their result.

## 2 Parsing Expression Grammar

We recall that Parsing Expression Grammar is a set of *parsing expressions*, specified by rules of the form $A = e$, where $e$ is a parsing expression and $A$ is the name given to it. Parsing expressions are procedures for parsing strings, written in a special language. The expressions can call each other recursively, thus forming together a recursive-descent parser.

In general, parsing expression is applied to a character string (the "input") at a position indicated by some "cursor". It tries to recognize a portion of the string ahead of the cursor. If it succeeds, it "consumes" the recognized portion by advancing the cursor and reports success; otherwise, it reports

*Appeared in *Fundamenta Informaticae 109*, 3 (2011) 323–337.

failure and does not consume anything (does not advance the cursor). Some expressions can succeed without advancing the cursor. We say in this case that the expression consumed empty string.

The possible forms of parsing expressions are listed below.

| | |
|---|---|
| `"`$s$`"` | (*String Literal*) If the input ahead starts with the string $s$, consume that string and report success. Otherwise report failure. |
| `[`$s$`]` | (*Character Class*) If the input ahead starts with any character from the string $s$, consume that character and report success. Otherwise report failure. |
| `[`$c_1$`-`$c_2$`]` | (*Character Range*) If the input ahead starts with any character from the range $c_1$–$c_2$, consume that character and report success. Otherwise report failure. |
| `_` | (*Any Character*) If there is a character ahead, consume that character and report success. Otherwise (that is, at the end of input) report failure. |
| $e$`?` | (*Option*) Call expression $e$ and report success (even if $e$ failed). |
| $e$`*` | (*Star*) Call expression $e$ repeatedly as long as it succeeds.<br>Report success (even if $e$ did not succeed a single time). |
| `!`$e$ | (*Lookahead*) Call expression $e$ and then reset the cursor as it was before the invocation of $e$. Report success if $e$ failed or failure if $e$ succeeded. |
| $e_1 \dots e_n$ | (*Sequence*) Call expressions $e_1, \dots, e_n$ in this order. If all succeeded, report success. If any of them failed, reset the cursor as it was before the invocation of $e_1$ and report failure. |
| $e_1$ `/` $\dots$ `/` $e_2$ | (*Choice*) Call expressions $e_1, \dots, e_n$ in this order, until one of them succeeds, and report success. If none of them succeeded, report failure. |

The first four act directly on the input. They are analogous to terminals of a classical context-free grammar, and are in the following referred to as "terminals". The remaining five call other expressions to do their job. The expressions $e$ and $e_i$ appearing in the table may be specified either explicitly or by name. Parentheses may be used to indicate the order of applying the operators. In the absence of parentheses, operators appearing higher in the table have precedence.

Backtracking takes place in the sequence expression. If $e_1, \dots, e_j$ succeed and consume some input, and then $e_{j+1}$ fails, the cursor is moved back to where it was before trying $e_1$. If this sequence was called as an alternative in a choice expression, this latter has an opportunity to try next alternative (if any) on the same input. However, the opportunity to try another alternative is limited: once $e_i$ in the choice succeeded, the parser will never return and try $e_j$ with $j > i$ if the parse fails later on.

# 3 Some notation

Let $\Sigma$ be the input alphabet. The set of all strings of letters from $\Sigma$ is denoted by $\Sigma^*$, and the set of all nonempty strings by $\Sigma^+$. The empty string is denoted by $\varepsilon$. The complement of $X \subseteq \Sigma^*$ with respect to $\Sigma^*$ is denoted by $\overline{X}$. For $w \in \Sigma^*$ and $p \geq 1$, we write $w[p{\rightarrow}]$ to mean the tail of $w$ starting at position $p$. Using the standard notation for concatenation product, we write $X\Sigma^*$ to mean the set of strings $xy$ where $x \in X$ and $y \in \Sigma^*$.

In the following, we consider a fixed grammar. The set of all its expressions is denoted by $\mathcal{G}$. All sub-expressions are treated as separate members of $\mathcal{G}$. When we speak of "an expression", we always mean a member of $\mathcal{G}$. Some of the expressions must be terminals; their set is denoted by $\mathcal{T} \subseteq \mathcal{G}$.

For an expression $e$ and $w \in \Sigma^*$, we write $e(w) = x$ to mean that $e$ applied to input $w$ consumes $x$. We write $e(w) = \varphi$ to mean that $e$ fails when applied to $w$. We define:

$$\mathcal{M}(e) = \{w \in \Sigma^* \,|\, e(w) \neq \varphi\}\,. \tag{1}$$

In [2], $\mathcal{M}(e)$ was called the "match set" of $e$. We define the language of $e$ as:

$$\mathcal{L}(e) = \{x \in \Sigma^* \,|\, e(xy) = x \;\text{ for some }\; y \in \Sigma^*\}\,. \tag{2}$$

Note that $x \in \mathcal{L}(e)$ does not necessarily mean $e(xy) = x$ for an arbitrary string $y$. However,

$$x \in \mathcal{L}(t) \;\Rightarrow\; t(xy) = x \;\text{ for all }\; t \in \mathcal{T} \;\text{ and }\; y \in \Sigma^*. \tag{3}$$

A relation $R$ on $\mathcal{G}$ is a subset of $\mathcal{G} \times \mathcal{G}$. As usual, we write $R(e)$ to mean the set of all $e'$ such that $(e, e') \in R$. The transitive and reflexive closure of $R$ is denoted by $R^*$, and the product of relations $R$ and $S$ by $R \times S$.

# 4 Nullability, fallibility, and well-formedness

We shall need three properties of expressions defined in [2]. We call them here *nul* (nullable), *adv* (may advance), and *fail* (may fail). We treat them as subsets of $\mathcal{G}$ that contain expressions with the respective property. The following is shown to hold for all $e \in \mathcal{G}$:

If $e(x) = \varepsilon$ for some $x \in \Sigma^*$ then $e \in \mathit{nul}$;
If $e(x) \in \Sigma^+$ for some $x \in \Sigma^*$ then $e \in \mathit{adv}$;
If $e(x) = \varphi$ for some $x \in \Sigma^*$ then $e \in \mathit{fail}$.

Note that these implications can not, in general, be reversed.

The evaluation of an expression $e$ may never terminate, because of infinite recursion or iteration. Such $e$ may have none of the above properties, but may have some of them (if it terminates for some input strings).

Termination is handled in [2] by another property of an expression, denoted by WF (well-formed). It is shown that if all expressions are well-formed, the evaluation of every expression always terminates. The grammar is then said to be well-formed. A well-formed grammar is not left-recursive and does not contain any expression $e^*$ with $e \in \mathit{nul}$. In the following, we assume $\mathcal{G}$ to be well-formed.

# 5 Auxiliary properties

We shall need two more auxiliary properties: SUCC and NULL.

**Definition 1.** For $e \in \mathcal{G}$, define $\mathrm{SUCC}(e)$ as the subset of $\Sigma^*$ such that:

$\mathrm{SUCC}(t) = \mathcal{L}(t)\Sigma^*$ for $t \in \mathcal{T}$;
$\mathrm{SUCC}(e_1/\ldots/e_n) = \mathrm{SUCC}(e_1) \cup \cdots \cup \mathrm{SUCC}(e_n)$;
$\mathrm{SUCC}(e) = \varnothing$ otherwise.

Because we assumed $\mathcal{G}$ to be well-formed, and thus not left-recursive, this definition is not recursive, and the value of $\mathrm{SUCC}(e_1/\ldots/e_n)$ can always be expressed in terms of the other two cases.

**Proposition 1.** *For any $e \in \mathcal{G}$ and $w \in \Sigma^*$, $w \in \mathrm{SUCC}(e) \;\Rightarrow\; e(w) \neq \varphi$.*

(Proof is found in the Appendix.)

**Definition 2.** For $e \in \mathcal{G}$, define $\mathrm{NULL}(e)$ as the subset of $\Sigma^*$ such that:

$\mathrm{NULL}(t) = \varnothing$ for $t \in \mathcal{T}$;
$\mathrm{NULL}(!e) = \overline{\mathrm{SUCC}(e)}$;
$\mathrm{NULL}(e^*) = \Sigma^*$;
$\mathrm{NULL}(e?) = \Sigma^*$;
$\mathrm{NULL}(e_1 \ldots e_n) = \varnothing$ if $e_1 \ldots e_n \notin \mathit{nul}$;
$\mathrm{NULL}(e_1 \ldots e_n) = \mathrm{NULL}(e_1) \cap \cdots \cap \mathrm{NULL}(e_n)$ if $e_1 \ldots e_n \in \mathit{nul}$;
$\mathrm{NULL}(e_1/\ldots/e_n) = \mathrm{NULL}(e_1) \cup \cdots \cup \mathrm{NULL}(e_n)$.

Again, one can verify that this definition is not recursive for a well-formed $\mathcal{G}$, so the values of NULL for $e_1 \ldots e_n$ and $e_1/\ldots/e_n$ are eventually reduced to those defined directly.

**Proposition 2.** *For any $e \in \mathcal{G}$ and $w \in \Sigma^*$, $e(w) = \varepsilon \;\Rightarrow\; w \in \mathrm{NULL}(e)$.*

(Proof is found in the Appendix.)

The main use of NULL will be to identify inputs on which a lookahead expression *may* succeed. Thus, for example, NULL(`!"abc"`) = $\overline{\texttt{"abc"}\Sigma^*}$ means that `!"abc"` may only succeed (by consuming $\varepsilon$) on inputs that do not start with *abc*. The definition of NULL(`!`$e$) depends on SUCC($e$): a set of inputs on which $e$ *must* succeed. We cannot use here the match set $\mathcal{M}(e)$ because its evaluation is in general quite intractable. In fact, the purpose of the whole exercise reported here is to get some grasp of it. For this reason, we choose SUCC($e$) to be a subset of $\mathcal{M}(e)$ that can be easily obtained, namely $\mathcal{M}(e)$ if $e$ represents a set of terminals, or $\varnothing$ otherwise. The consequence is that NULL(`!`$e$) = $\Sigma^*$ for a more complex $e$: we cannot say anything about the inputs on which `!`$e$ succeeds.

# 6 Property BITES

We shall say that an expression $e$ applied to a string $w$ "bites" that string to mean that it makes some progress towards consuming a part of that string. More precisely: at least one terminal called in the process otherwise than via a lookup expression consumes a nonempty string. We are going to define BITES($e$) as the set of inputs that $e$ may bite.

**Definition 3.** For $e \in \mathcal{G}$, define BITES($e$) as the subset of $\Sigma^*$ such that:
BITES($t$) = $\mathcal{L}(t)\Sigma^*$ for $t \in \mathcal{T}$;
BITES(`!`$e$) = $\varnothing$;
BITES($e^*$) = BITES($e$);
BITES($e$`?`) = BITES($e$);
BITES($e_1 \dots e_n$) = BITES($e_1$) $\cup \bigcup_{1<i\leq n}\big(\text{NULL}(e_1 \dots e_{i-1}) \cap \text{BITES}(e_i)\big)$;
BITES($e_1/\dots/e_n$) = BITES($e_1$) $\cup \dots \cup$ BITES($e_n$).

Because NULL($e_1 \dots e_{i-1}$) = $\varnothing$ if $e_1 \dots e_{i-1} \notin$ *nul*, the union in the definition of BITES($e_1 \dots e_n$) can be restricted to those $i$ for which $e_1 \dots e_{i-1} \in$ *nul*. Using this observation, one can verify that the above definition is not recursive for a well-formed $\mathcal{G}$.

**Proposition 3.** *For any $e \in \mathcal{G}$ and $w \in \Sigma^*$, if $e$ bites $w$ then $w \in$ BITES($e$).*

(Proof is found in the Appendix.)

As an illustration, we show the computation of BITES for two expressions:

BITES(`"abc"[a-z]`$^*$) = `"abc"`$\Sigma^* \cup (\varnothing \cap$ `[a-z]`$) =$ `"abc"`$\Sigma^*$.

BITES(`(!"abc")[a-z]`$^*$) = $\varnothing \cup (\overline{\texttt{"abc"}\Sigma^*} \cap$ `[a-z]`$\Sigma^*) = \overline{\texttt{"abc"}\Sigma^*} \cap$ `[a-z]`$\Sigma^*$.

One can see from these examples how BITES expresses information about "first terminals". Because succeeding on an input requires biting it, the first expression may only succeed on inputs starting with *abc*, and the second on inputs starting with a letter, but not with *abc*.

# 7 Application to choice expression

**Definition 4.** We say that the choice $e_1/\dots/e_n$ is *disjoint* to mean that
BITES($e_i$) $\cap$ BITES($e_j$) = $\varnothing$ for $1 \leq i, j \leq n$, $i \neq j$.

**Proposition 4.** *At most one alternative of a disjoint choice can bite a given input string.*

*Proof.* Suppose each of $e_i$, $e_j$, $i \neq j$, in a disjoint choice $e_1/\dots/e_n$ bites string $w$. According to Proposition 3, $w \in$ BITES($e_i$) and $w \in$ BITES($e_j$), which means BITES($e_i$) $\cap$ BITES($e_j$) $\neq \varnothing$. □

The "language hiding" mentioned in the Introduction occurs when a greedy alternative of a choice expression prevents a later alternative from being applied to inputs that it could otherwise accept. Classical examples are `("a"/"aa")"b"`, `("aa"/"a")"ab"`, which both fail on input *aab*, but would succeed if the second alternative was tried.

Let us define "language hiding" more precisely as the situation where $e_i$ in $e_1/\dots/e_n$ succeeds on some input but parsing fails later on; a later alternative $e_j$ would also succeed on that input, to end in a successful parse.

**Proposition 5.** *Language hiding does not occur in non-nullable disjoint choice.*

*Proof.* From $e_1/\ldots/e_n \notin nul$ follows $e_i \notin nul$ for $1 \leq i \leq n$. That means each of the successful $e_i$ and $e_j$ must bite the same input, which is not possible according to Proposition 4. □

With the same reasoning as above, we obtain:

**Proposition 6.** *If one alternative of a non-nullable disjoint choice fails after biting some input, all subsequent alternatives will fail on that input.*

In other words, we can exit and indicate failure as soon as an alternative failed after consuming at least one terminal. One may view it as predictive parsing for PEG. Mizushima, Yaeda and Yamaguchi [3] used this observation to drastically improve memory utilization in packrat parsers. By inserting a "cut" operator after the first terminal, they instruct the parser to dispose of backtrack information as soon as it becomes unnecessary.

# 8 Property NEXTBITES

We are now going to extend our results to the nullable case. Language hiding in a nullable choice is exemplified by `("a"/"c"?)"a"`, which fails on input *ab*, but would succeed if `"c"?` was tried. Note that this does not happen in `("a"/"c"?)"b"`, where `"a"` does not hide `"c"?`. This indicates that we need to look at what happens after the choice expression.

We also need to consider another cause of language hiding, namely the star operator. For example, `"ab"` in the expression `("a")`$^*$ `("ab"/"c")` will never succeed; it is hidden by the greedy `("a")`$^*$. However, nothing will be hidden in `("a")`$^*$ `("b"/"c")`. Again, we need to look at what happens after the expression.

In order to look at what happens after an expression, we introduce the property NEXTBITES. It is a set of strings that the parser may bite after return from a given expression. This set will depend on whether the expression succeeded or failed.

NEXTBITES applies to an expression in the context of parsing process. We consider a process that begins by calling some "starting expression" $S$ with cursor at the beginning of an input string $w$. We say that expression $e$ was "called at position $p$" to mean that it was called with cursor at position $p$ in $w$. If $e$ succeeds, it may move the cursor to some position $q \geq p$. We say then that $e$ "succeeded at $q$". (We have $q = p$ if $e$ consumed empty string.) If $e$ fails, we say that $e$ "failed at $p$". We say that $e$ "terminated at $p$" to mean that $e$ either succeeded at $p$ or failed at $p$. If the expression that terminated is $S$, the parsing process ends at $p$. Otherwise the expression returns to its caller $E$. The caller $E$ may then call one or more expressions that consume $\varepsilon$, but eventually one of these must happen:
(a) $E$ terminates at $p$.
(b) $E$ calls at $p$ an expression $e'$ that bites the input.
(c) $E$ backtracks.
In case (a) we say that $e$ is "last in" $E$. In case (b) we say that $e'$ "bites the input after" $e$.

Let $\mathcal{R} = \{\texttt{succ},\texttt{fail}\}$ represent the possible results (success, failure) reported by a terminating expression. In the Appendix, we show how to define relations $\text{Last} \subseteq (\mathcal{G},\mathcal{R}) \times (\mathcal{G},\mathcal{R})$ and $\text{Next} \subseteq (\mathcal{G},\mathcal{R}) \times \mathcal{G})$ having these properties:

**Proposition 7.** *If expression $e$ that terminated with result $r$ is last in $E$ that terminated with result $R$ then $(e,r) \in \text{Last}(E,R)$.*

**Proposition 8.** *If $e'$ bites the input after $e$ that terminated with result $r$ then $e' \in \text{Next}(e,r)$.*

There can be a chain of expressions $E_n, E_{n-1}, \ldots, E_2, E_1, e$, calling each other in this order, such that $e$ is last in $E_1$, $E_1$ is last in $E_2$, etc., and $E_n$ calls an expression $e'$ that bites the input after $E_{n-1}$. We say also in this case that $e'$ bites the input (indirectly) after $e$.

**Definition 5.** For $e \in \mathcal{G}$ and $r \in \mathcal{R}$ define

$$\text{NEXTBITES}(e, r) = \bigcup_{e' \in \text{NEXT}(e,r)} \text{BITES}(e')\,,$$

where $\text{NEXT} = \text{Last}^* \times \text{Next}$.

**Proposition 9.** *Suppose an expression $e$ terminates at $p$ with result $r$, and $e'$ bites the input (directly or indirectly) after $e$. Then $w[p\rightarrow] \in \text{NEXTBITES}(e, r)$.*

(Proof is found in the Appendix.)

# 9 Choice expression revisited

We restrict our definition of language hiding in nullable choice to the cases where the parser does not terminate at, or backtrack from, the position where the choice succeeded by consuming null string.

**Proposition 10.** *Language hiding does not occur in a disjoint choice $E = e_1/\ldots/e_n$ satisfying these conditions:*

$$e_i \notin \mathit{nul} \;\; \mathit{for} \;\; 1 \le i < n\,; \tag{4}$$

$$e_n \in \mathit{nul} \;\Rightarrow\; \text{BITES}(e_i) \cap \text{NEXTBITES}(E, \texttt{succ}) = \varnothing \;\; \mathit{for} \;\; 1 \le i < n\,. \tag{5}$$

*Proof.* Assume that language hiding occurs in $E$: alternative $e_i$ with $i < n$ succeeds, but parse fails later on; alternative $e_j$ with $j > i$ applied to the same input results in a successful parse.
According to (4), $e_i$ consumes a nonempty string. If $e_j$ also consumes a nonempty string, both $e_i$ and $e_j$ bite the same input, which is not possible according to Proposition 4.
Suppose now that $e_j$ consumes $\varepsilon$, and thus terminates at the position $p$ where $E$ was called. According to (4), we have $j = n$ and $e_n \in \mathit{nul}$. If the parser does not terminate at $p$ or backtrack from $p$, but continues to a successful parse, it must eventually call at $p$ an expression $e$ that consumes a nonempty string. This $e$ bites the input after $E$, so we have $w[p\rightarrow] \in \text{NEXTBITES}(E, \texttt{succ})$ by Proposition 9. As $e_i$ consumed a nonempty string when applied to $w$ at position $p$, we have also $w[p\rightarrow] \in \text{BITES}(e_i)$, which is not possible according to (5). □

The following extends Proposition 6 to the nullable case.

**Proposition 11.** *Let $E = e_1/\ldots/e_n$ be a disjoint choice satisfying these conditions:*

$$E \in \mathit{nul} \;\Rightarrow\; \text{BITES}(e_i) \cap \text{NEXTBITES}(E, \texttt{succ}) = \varnothing \;\; \mathit{for} \;\; 1 \le i < n\,; \tag{6}$$

$$E \in \mathit{fail} \;\Rightarrow\; \text{BITES}(e_i) \cap \text{NEXTBITES}(E, \texttt{fail}) = \varnothing \;\; \mathit{for} \;\; 1 \le i \le n\,. \tag{7}$$

*After its alternative $e_i$ failed after biting the input, no other expression, either a part of $E$, or after $E$, can succeed on that input.*

*Proof.* Suppose $E$ is called at position $p$; its alternative $e_i$ bites the input and fails. Suppose that $e$ bites the input after $E$.

According to Proposition 4, no other alternative of $E$ can bite the input after $e_i$. That means $E$ either fails at $p$, or succeeds at $p$ because an alternative $e_j$ with $j > i$ consumes $\varepsilon$. We consider these two cases, remembering that $w[p\rightarrow] \in \text{BITES}(e_i)$.
Case 1: $E$ succeeded. As $e_j \in \mathit{nul}$, we have $E \in \mathit{nul}$, and from $j > i$ follows $i < n$. According to Proposition 9, $w[p\rightarrow] \in \text{NEXTBITES}(E, \texttt{succ})$, which contradicts (6).
Case 2: $E$ failed. We have $E \in \mathit{fail}$. According to Proposition 9, $w[p\rightarrow] \in \text{NEXTBITES}(E, \texttt{fail})$, which contradicts (7). □

In other words, once an alternative of $E$ failed after succeeding with its first terminal, the parse will not progress past the point of call of $E$, except for temporary advance by lookahead expressions – until the parser possibly terminates, or backtracks and takes a completely new approach.

# 10 Application to star expression

Let us define language hiding in a star expression as the situation where $e^*$ succeeds after $n > 1$ calls to $e$, and the subsequent parse fails. But, calling $e$ only $m < n$ times would result in a successful parse. As in the case of choice expression, we restrict this definition to the case where the parser neither terminates nor backtracks after the $m$-th call.

**Proposition 12.** *Language hiding does not occur in expression $E = e^*$ satisfying this condition:*

$$\text{BITES}(e) \cap \text{NEXTBITES}(E, \texttt{succ}) = \varnothing\,. \tag{8}$$

*Proof.* Suppose language hiding does occur: $E = e^*$ succeeds after $n > 1$ calls to $e$; stopping $e^*$ after $m < n$ calls results in a successful parse. Let $p$ be the position where the $m$-th call terminated. If now the parser neither terminates nor backtracks, the successful parse requires that some expression $e'$ is called at $p$ and consumes a nonempty string. This $e'$ bites the input after successful $E$, so we have, form Proposition 9, $w[p\rightarrow] \in \text{NEXTBITES}(E, \texttt{succ})$. As $\mathcal{G}$ is well-formed, we have $e \notin \mathit{nul}$, so the $(m+1)$-th call to $e$ also consumed a nonempty string starting at $p$. We have thus $w[p\rightarrow] \in \text{BITES}(e)$, which is impossible according to (8). □

The following is analogous to Proposition 11:

**Proposition 13.** *If the expression $e$ in $E = e^*$ satisfying (8) fails after biting its input, no expression called after $E$ can succeed on that input.*

*Proof.* Suppose the $n$-th call to $e$ (where $n \geq 1$) fails at $p$ after biting the input, so $E$ succeeds at $p$ after $n-1$ calls to $e$. We have $w[p\rightarrow] \in \text{BITES}(e)$. Suppose that $e'$ bites the input after $E$. According to Proposition 9, we have $w[p\rightarrow] \in \text{NEXTBITES}(E, \texttt{succ})$. But, this is not possible according to (8). □

In other words, once $e$ failed after succeeding with its first terminal, the parse will not progress past the point of success of $E$, except for temporary advance by lookahead expressions – until the parser possibly terminates, or backtracks and takes a completely new approach.

# 11 What works and what does not

We have seen in Section 6 how BITES encodes information about "first terminals". Thus, for example, $\text{BITES}(e) = \overline{\texttt{"abc"}\Sigma^*} \cap \texttt{[a-z]}\Sigma^*$ is a concise way of saying that strings consumed by $e$ may start with a letter, but not with *abc*. The following two examples show how this encoding is better than one using the FIRST sets.

In the choice `"abc"[a-z]*/(!"abc")[a-z]*`, the first alternative consumes strings of letters starting with *abc*, and the other strings of letters *not* starting with *abc*. There is no language hiding here, but the alternatives are not disjoint according to [6]. The reason is that the FIRST sets for the two alternatives are, respectively, {`"abc"`} and {`"abc"`,`[a-z]`}, which are not disjoint. An automatic checker using this condition suggests further examination. Meanwhile, the BITES sets of the two alternatives are $\texttt{"abc"}\Sigma^*$ and $\overline{\texttt{"abc"}\Sigma^*} \cap \texttt{[a-z]}\Sigma^*$. They are disjoint, and Proposition 5 confirms the absence of hiding.

Another example is the frequently used construction of the form `(!"a"_)*"a"`, meaning "consume everything up to *a*". Using the approach from [6], an automatic checker suggests that `(!"a"_)` may hide `"a"`, because FIRST`(!"a"_)` = {`"a"`,`_`}. Here we have $\text{BITES}(\texttt{(!"a"\_)}) = \overline{\texttt{"a"}\Sigma^*} \cap \Sigma^+$ and $\text{NEXTBITES}(\texttt{(!"a"\_)}^*, \texttt{succ}) = \texttt{"a"}\Sigma^*$, which satisfies (8) and confirms the absence of hiding.

The new approach thus eliminates some artifacts produced by the use of FIRST. Unfortunately, it still does not work in many cases. One example is the typical definition of names that are distinct from reserved keywords:

```
name = !key [a-z][a-z]*
key  = "down"![a-z]
```

The `name` is supposed to be any string of letters other than *down*, which is expressed by starting its definition with `!key`. The `![a-z]` at the end of `!key` is needed to allow `name`s starting with *down*. Unfortunately, it gives $\text{SUCC}(\texttt{key}) = \varnothing$ and $\text{BITES}(\texttt{name}) = \texttt{[a-z]}\Sigma^*$, which is not disjoint with

BITES(`key`) = `"down"`$\Sigma^*$. We could repair this by defining SUCC(`key`) = `"down"`$\overline{\texttt{[a-z]}}\Sigma^*$. This suggests a possible improvement: define SUCC in the form $R\Sigma^*$ where $R$ is a regular expression on (languages of) terminals extended with Boolean operations.

Another example is `(!A_)`$^*$ `A` where `A` is not a set of terminals. We have then SUCC(`A`) = $\varnothing$ and BITES`(!A_)`$= \Sigma^+$, which does not satisfy (8). Again, this might be repaired by extending the definition of SUCC.

# 12 Final comments

The new approach gives an improvement with respect to [6], but is more difficult to implement. Relations FIRST and FOLLOW used in [6] are sets of terminals that are easily computed using bit sets and bit matrices. This is still true for relation NEXT used here, but BITES and NEXTBITES are set of strings described by regular expressions with Boolean operators. The emptiness problem for such expressions is decidable, but standard procedures are cumbersome with a large alphabet. It may be possible to find some shortcuts using the observation that the sets are Boolean combinations of sets of the form $\mathcal{L}(t)\Sigma^*$ for $t \in \mathcal{T}$.

As indicated in Section 5, SUCC($e$) is a subset of $\mathcal{M}(e)$. One can easily see that BITES($e$) is a superset of $\mathcal{M}(e)$. We can view BITES($e$) and SUCC($e$) as approximations of $\mathcal{M}(e)$ "from above" and "from below" based on initial terminal calls. Using more initial terminals, as suggested above, would improve these approximations and language hiding diagnostics. On the other hand, the "predictive parsing" application from [3] (see comment to Proposition 6) requires that we rely on as few terminals as possible, so we can quickly exclude impossible alternatives.

# A Appendix

## A.1 Proof of Proposition 1

By structural induction: assume the Proposition is true for each expression on the right-hand side in Definition 1, and consider any $E \in \mathcal{G}$.

- Case $E = t \in \mathcal{T}$: Take any $w = xy$ where $x \in \mathcal{L}(t)$, $y \in \Sigma^*$. According to (3), $t(w) = x \neq \varphi$.
- Case $E = e_1/\ldots/e_n$: Take any $w \in \mathrm{SUCC}(e_1) \cup \cdots \cup \mathrm{SUCC}(e_n)$, That means $w \in \mathrm{SUCC}(e_i)$ for at least one $i$, and by induction hypothesis, $e_i$ succeeds on $w$.
- Case "otherwise": The left-hand side of the implication is false for every $w$.

## A.2 Proof of Proposition 2

By structural induction: assume the Proposition is true for each expression on the right-hand side in Definition 2, and consider any $E \in \mathcal{G}$.

- Case $E \in \mathcal{T}$: As a terminal never consumes $\varepsilon$, the left-hand side of the implication is false.
- Case $E =$ `!`$e$: Suppose $E(w) = \varepsilon$. It means that $e(w) = \varphi$. By Proposition 1, $w \notin \mathrm{SUCC}(e)$.
- Case $E = e^*$ and $E = e?$: The right-hand side of the implication is true for each $w$.
- Case $E = e_1 \ldots e_n \notin \mathit{nul}$: We have $E(w) \neq \varepsilon$; the left-hand side of the implication is false.
- Case $E = e_1 \ldots e_n \in \mathit{nul}$: Suppose $E(w) = \varepsilon$. That means each of the expressions $e_i$ consumed $\varepsilon$. By induction hypothesis, $w \in \mathrm{NULL}(e_i)$ for each of them.
- Case $E = e_1/\ldots/e_n$: Suppose $E(w) = \varepsilon$. That means one of the expressions $e_i$, consumed $\varepsilon$. By induction hypothesis, $w \in \mathrm{NULL}(e_i)$ for that $e_i$.

## A.3 Proof of Proposition 3

By structural induction: assume the Proposition is true for each expression on the right-hand side in Definition 3, and consider any $E \in \mathcal{G}$.

- Case $E \in \mathcal{T}$: Suppose $E \in \mathcal{T}$ bites $w$. That means it consumes a nonempty string, that is, $w = xy$ where $x \in \mathcal{L}(E)$ and $y \in \Sigma^*$.
- Case $!e$: By definition, $E$ does not bite input, so the left-hand side of the implication is false.
- Case $E = e^*$ and $E = e?$: Suppose $E$ did bite $w$. That could only happen by $e$ biting $w$. By induction hypothesis, $w \in \text{BITES}(e)$.
- Case $E = e_1 \dots e_n$: Suppose $E$ did bite $w$. That means either $e_1$ bit $w$, or $e_1 \dots e_{i-1}$ consumed $\varepsilon$, after which the $e_i$ bit $w$, where $1 < i \leq n$. By induction hypothesis and Proposition 2, we have $w \in \text{BITES}(e_1)$, respectively $w \in \text{BITES}(e_i)$ and $w \in \text{NULL}(e_1 \dots e_{i-1})$.
- Case $E = e_1 / \dots / e_n$: Suppose $E$ did bite $w$. That means one of $e_1, \dots, e_n$ bit $w$. Let this expression be $e_i$. By induction hypothesis, we have $w \in \text{BITES}(e_i)$.

## A.4 Proof of Proposition 9

We note first that NEXT it is a relation in $((\mathcal{G}, \mathcal{R}) \times \mathcal{G})$. Suppose $e$ terminated at $p$ with result $r$ and $e'$ bit the input after $e$.

Suppose $e'$ was called directly by the caller of $e$. According to Proposition 8, we have $((e, r), e') \in \text{Next}$. Because $\text{Last}^*$ contains identity relation, we have $\text{Next} \subseteq \text{Last}^* \times \text{Next}$, so $e' \in \text{NEXT}(e, r)$.

Suppose $e'$ was called via the described chain of $n > 1$ expressions $E_i$ that terminated with result $r_i$, respectively. According to Proposition 8, we have $((e, r), (E_1, r_1)) \in \text{Last}$, and $((E_i, r_i), (E_{i+1}, r_{i+1})) \in \text{Last}$ for $1 \leq i < n - 1$. According to Proposition 8, we have $((E_{i-1}, r_{i-1}), e') \in \text{Next}$, so $((e, r), e') \in \text{Last} * \times \text{Next}$, that is, $e' \in \text{NEXT}(e, r)$.

Expression $e'$ biting the input means $w[p \rightarrow] \in \text{BITES}(e')$. As $e' \in \text{NEXT}(e, r)$, we have $\text{BITES}(e') \subseteq \text{NEXTBITES}(e, r)$ according to Definition 5.

## A.5 Construction of relation Last

We decide if a pair $((e, r), (E, R))$ belongs to Last separately for all four combinations of $r, R$.

### A.5.1 Case $r = \texttt{succ},\ R = \texttt{succ}$.

The pair $((e, \texttt{succ}), (E, \texttt{succ}))$ belonging to Last means that $e$ and $E$ appear in the following scenario:
(1) Expression $E$ calls expression $e$.
(2) $e$ succeeds at $p$, returns to $E$.
(3) $E$ succeeds at $p$.

An expression $E$ calling $e$ can appear in these contexts:

- $E = e?$. If $e$ succeeds at $p$, so does $E$. The scenario implies only $e \in \mathit{adv} \cup \mathit{nul}$.
- $E = e^*$. The scenario is not possible in this case: $E$ succeeds only after $e$ failed.
- $E = !e$. The scenario is not possible in this case: if $e$ succeeds, $E$ fails.
- $e = e_i$ in $E = e_1 \dots e_n$. If $E$ called $e_i$, all $e_j$ for $j < i$ must have succeeded. If $E$ succeeded at $p$, all of $e_j$ for $j > i$ must have consumed $\varepsilon$. This implies $\forall_{j<=i}(e_j \in \mathit{adv} \cup \mathit{nul}) \ \wedge \ \forall_{j>i}(e_j \in \mathit{nul})$.
- $e = e_i$ in $E = e_1 / \dots / e_n$. If $E$ called $e_i$, all of $e_j$ with $j < i$ must have failed. If $e$ succeeds at $p$, so does $E$. This implies $e_i \in \mathit{adv} \cup \mathit{nul} \ \wedge \ \forall_{j<i}\,(e_j \in \mathit{fail})$.

Thus, $((e, \texttt{succ}), (E, \texttt{succ})) \in \text{Last}$ if and only if any of the following holds:

- $e$ appears in $E = e?$ and $e \in \mathit{adv} \cup \mathit{nul}$;
- $e = e_i$ in $E = e_1 / \dots / e_n$ and $e_i \in \mathit{adv} \cup \mathit{nul} \ \wedge \ \forall_{j<i}\,(e_j \in \mathit{fail})$;
- $e = e_i$ in $E = e_1 \dots e_n$ and $\forall_{j<=i}(e_j \in \mathit{adv} \cup \mathit{nul}) \ \wedge \ \forall_{j>i}(e_j \in \mathit{nul})$.

### A.5.2 Case $r = \texttt{succ},\ R = \texttt{fail}$.

The pair $((e, \texttt{succ}), (E, \texttt{fail}))$ belonging to Last means that $e$ and $E$ appear in the following scenario:
(1) Expression $E$ calls expression $e$.
(2) $e$ succeeds at $p$, returns to $E$.
(3) $E$ fails at $p$.

An expression $E$ calling $e$ can appear in these contexts:
- $E = e^*$ or $E = e?$. The scenario is not possible in this case: $E$ always succeeds.
- $E = \texttt{!}e$. If $e$ succeeds, $E$ fails. If $E$ failed at $p$, it must have been called at $p$. That means $e$ was called at $p$. To succeed at $p$, it must have consumed $\varepsilon$. This implies $e \in \mathit{nul}$.
- $e = e_i$ in $E = e_1 \dots e_n$. If $e$ succeeded and $E$ failed, some $e_j$ with $j > i$ must have failed. For $e_j$ (and also $e_i$) to be called, all $e_k$ with $k < j$ must have succeeded. $E$ failing at $p$ means that it has been called at $p$ and backtracking returned the cursor there. Hence, $e_i$ succeeding at $p$ means that all of $e_1, \dots, e_i$ must have consumed null.
  This implies $\exists_{j>i}(e_j \in \mathit{fail} \wedge \forall_{k<j}(e_k \in \mathit{adv} \cup \mathit{nul}))$ and $\forall_{k<=i}(e_k \in \mathit{nul})$.
- $e = e_i$ in $E = e_1 / \dots / e_n$. The scenario is not possible in this case: if $e$ succeeds, so does $E$.

Thus, $((e, \texttt{succ}), (E, \texttt{fail})) \in$ Last if and only if any of the following holds:
- $e$ appears in $E = \texttt{!}\,e$ and $e \in \mathit{nul}$;
- $e = e_i$ in $E = e_1 \dots e_n$ and $\exists_{j>i}(e_j \in \mathit{fail} \wedge \forall_{k<j}(e_k \in \mathit{adv} \cup \mathit{nul})) \ \wedge\ \forall_{k<=i}(e_k \in \mathit{nul})$.

### A.5.3 Case $r = \texttt{fail},\ R = \texttt{succ}$.

The pair $((e, \texttt{fail}), (E, \texttt{succ}))$ belonging to Last means that $e$ and $E$ appear in the following scenario:
(1) Expression $E$ calls expression $e$.
(2) $e$ fails at $p$, returns to $E$.
(3) $E$ succeeds at $p$.

An expression $E$ calling $e$ can appear in these contexts:
- $E = e^*$ or $E = e?$. After $e$ failing at $p$, $E$ succeeds at $p$. The scenario implies only $e \in \mathit{fail}$.
- $E = \texttt{!}e$. If $e$ failed, $E$ succeeds. But, $E$ always consumes $\varepsilon$. If it succeeded at $p$, it must have been called at $p$. That means $e$ was called at $p$, which agrees with its failing at $p$. The scenario implies only $e \in \mathit{fail}$.
- $e = e_i$ in $E = e_1 \dots e_n$. The scenario is not possible in this case: if $e_i$ fails, so does $E$.
- $e = e_i$ in $E = e_1 / \dots / e_n$. If $E$ succeeded, some $e_j$ with $j > i$ must have succeeded. If $E$ called $e_j$, all of $e_k$ with $k < j$ must have failed. If $e_i$ failed at $p$, it was called at $p$, which means $E$ was also called at $p$. For $E$ to succeed at $p$, $e_j$ must have also succeed at $p$, which means $e_j$ must have consumed $\varepsilon$. This implies $\exists_{j>i}(e_j \in \mathit{nul} \wedge \forall_{k<j}(e_k \in \mathit{fail}))$.

Thus, $((e, \texttt{fail}), (E, \texttt{succ})) \in$ Last if and only if any of the following holds:
- $e$ appears in $E = e?$ and $e \in \mathit{fail}$;
- $e$ appears in $E = e^*$ and $e \in \mathit{fail}$;
- $e$ appears in $E = \texttt{!}e$ and $e \in \mathit{fail}$;
- $e = e_i$ in $E = e_1 / \dots / e_n$ and $\exists_{j>i}(e_j \in \mathit{nul} \ \wedge\ \forall_{k<j}(e_k \in \mathit{fail}))$.

### A.5.4 Case $r = \texttt{fail},\ R = \texttt{fail}$.

The pair $((e, \texttt{fail}), (E, \texttt{fail}))$ belonging to Last means that $e$ and $E$ appear in the following scenario:
(1) Expression $E$ calls expression $e$.
(2) $e$ fails at $p$, returns to $E$.
(3) $E$ fails at $p$.

An expression $E$ calling $e$ can appear in these contexts:
- $E = e^*$ or $E = e?$. The scenario is not possible in this case: $E$ never fails.
- $E = \texttt{!}e$. The scenario is not possible in this case: if $e$ fails, $E$ succeeds.
- $e = e_i$ in $E = e_1 \dots e_n$. That $E$ failed at $p$ means it was called at $p$. That $e_i$ failed at $p$ means it was also called at $p$. If $e_i$ was called at $p$, all $e_j$ with $j < i$ must have consumed $\varepsilon$. This implies $e_i \in \mathit{fail} \ \wedge\ \forall_{j<i}(e_j \in \mathit{nul})$.

- $e = e_i$ in $E = e_1/\ldots/e_n$. That $E$ failed at $p$ means it was called at $p$, and so was $e_i$, which can only fail at $p$. That $E$ failed means all alternatives failed, and implies $\forall_i(e_i \in \mathit{fail})$.

Thus, $((e, \texttt{fail}), (E, \texttt{fail})) \in$ Last if and only if any of the following holds:
- $e = e_i$ in $E = e_1 \ldots e_n$ and $e_i \in \mathit{fail} \wedge \forall_{j<i}(e_j \in \mathit{nul})$;
- $e = e_i$ in $E = e_1/\ldots/e_n$ and $\forall_i(e_i \in \mathit{fail})$.

## A.6 Construction of relation Next

We decide if a pair $((e, r), e')$ belongs to Next separately for each value of $r$.

### A.6.1 Case $r = \texttt{succ}$

The pair $((e, \texttt{succ}), e')$ belonging to Next means that $e$ and $e'$ appear in the following scenario:
(1) Expression $E$ calls expression $e$.
(2) $e$ succeeds at $p$, returns to $E$.
(3) $E$ calls $e'$ at $p$.
(4) $e'$ bites the input.

An expression $E$ calling $e'$ after return of $e$ can appear in these contexts:
- $e' = e$ in $E = e^*$. $E$ always calls $e$ again after successful $e$. The scenario implies $e \in \mathit{adv}$.
- $e = e_i$, $e' = e_{i+1}$ in $E = e_1 \ldots e_n$. For $E$ to call $e' = e_{i+1}$, $e_{i+1}$ must exist, and all of $e_1, \ldots, e_{i-1}$ must have succeeded. As $e_i$ also succeeded, this implies $i < n \wedge e_{i+1} \in \mathit{adv} \wedge \forall_{j<=i}(e_j \in \mathit{adv} \cup \mathit{nul})$.
- $e = e_i$, $e' = e_{i+1}$ in $E = e_1/\ldots/e_n$. The scenario is not possible in this case: $E$ does not make any more calls after successful $e_i$.

Thus, $((e, \texttt{succ}), e') \in$ Next if and only if any of the following holds:
- $e$ appears in $E = e^*$, $e' = e$ and $e \in \mathit{adv}$;
- $e = e_i$, $e' = e_{i+1}$ in $E = e_1 \ldots e_n$ and $i < n \wedge e_{i+1} \in \mathit{adv} \wedge \forall_{j<=i}(e_j \in \mathit{adv} \cup \mathit{nul})$.

### A.6.2 Case $r = \texttt{fail}$

The pair $((e, \texttt{fail}), e')$ belonging to Next means that $e$ and $e'$ appear in the following scenario:
(1) Expression $E$ calls expression $e$.
(2) $e$ fails at $p$, returns to $E$.
(3) $E$ calls $e'$ at $p$ as the next sub-expression.
(4) $e'$ bites the input.

An expression $E$ calling $e'$ after return of $e$ can appear in these contexts:
- $e' = e$ in $E = e^*$. The scenario is not possible in this case: $E$ does not make any more calls after $e_i$ failed.
- $e = e_i$, $e' = e_{i+1}$ in $E = e_1 \ldots e_n$. The scenario is not possible in this case: $E$ does not make any more calls after $e_i$ failed.
- $e = e_i$, $e' = e_{i+1}$ in $E = e_1/\ldots/e_n$. For $E$ to call $e' = e_{i+1}$, $e_{i+1}$ must exist, and all of $e_1, \ldots, e_{i-1}$ must have failed. As $e_i$ also failed, this implies $i < n \wedge e_{i+1} \in \mathit{adv} \wedge \forall_{j<=i}(e_j \in \mathit{fail})$.

Thus, $((e, \texttt{fail}), e') \in$ Next if and only if the following holds:
- $e = e_i$, $e' = e_{i+1}$ in $E = e_1/\ldots/e_n$ and $i < n \wedge e_{i+1} \in \mathit{adv} \wedge \forall_{j<=i}(e_j \in \mathit{fail})$.

# References


[1] Aho, A. V., Sethi, R., Ullman, J. D.: *Compilers. Principles, Techniques, and Tools*, Addison-Wesley, 1987.

[2] Ford, B.: Parsing Expression Grammars: A Recognition-Based Syntactic Foundation, *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004* (N. D. Jones, X. Leroy, Eds.), ACM, Venice, Italy, 14–16 January 2004.

[3] Mizushima, K., Maeda, A., Yamaguchi, Y.: Packrat parsers can handle practical grammars in mostly constant space, *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE'10, Toronto, Ontario, Canada, June 5-6, 2010* (S. Lerner, A. Rountev, Eds.), ACM, 2010.

[4] Redziejowski, R. R.: Parsing Expression Grammar as a Primitive Recursive-Descent Parser with Backtracking, *Fundamenta Informaticae*, **79**(3–4), 2007, 513–524.

[5] Redziejowski, R. R.: Some Aspects of Parsing Expression Grammar, *Fundamenta Informaticae*, **85**(1–4), 2008, 441–454.

[6] Redziejowski, R. R.: Applying Classical Concepts to Parsing Expression Grammar, *Fundamenta Informaticae*, **93**(1–3), 2009, 325–336.

[7] Schmitz, S.: *Modular Syntax Demands Verification*, Technical Report I3S/RR-2006-32-FR, Laboratoire I3S, Université de Nice - Sophia Antipolis, October 2006,
`http://www.i3s.unice.fr/~mh/RR/2006/RR-06.32-S.SCHMITZ.pdf`.

[8] Tremblay, J.-P., Sorenson, P. G.: *The Theory and Practice of Compiler Writing*, McGraw-Hill, 1985.