CDMTCS Research Report Series

Indexed Grammars, ET0L Systems and Programming Languages — A Tribute to Alexandru Mateescu —

Radu Nicolescu
Department of Computer Science,
The University of Auckland,
Auckland, New Zealand

Dragoş Vaida
Department of Computer Science,
University of Bucharest,
Bucharest, Romania

CDMTCS-464
July 2014

Centre for Discrete Mathematics and
Theoretical Computer Science

Indexed Grammars, ET0L Systems and Programming Languages — A Tribute to Alexandru Mateescu — *

Radu Nicolescu
Department of Computer Science, The University of Auckland,
Auckland, New Zealand, r.nicolescu@auckland.ac.nz

Dragoş Vaida
Department of Computer Science, University of Bucharest,
Bucharest, Romania, dragos.vaida@clicknet.ro

14 July, 2014

Abstract

We revise and extend a couple of earlier incompletely published papers regarding the competence limits of formal systems in modelling the full syntax of programing languages. We show that the full syntax of mainstream programming languages (e.g. similar to Pascal or CAML) and of schema based XML documents cannot be modelled by either ET0L systems or indexed grammars. We raise a few open questions related to ET0L languages and two powerful but less known classes of languages: iterative languages and generalised Ogden-like languages.

Keywords: programming languages, formal syntax, static semantics, formal grammars and languages, context-free grammars and languages, ET0L systems and languages, indexed grammars and languages, iterative languages, generalised Ogden-like languages, Pascal-like languages, CAML-like languages, XSD schemas, valid XML documents.

1 Introduction

Classical results showed that *full syntax* (which includes *phrase structure* and *static semantics*) of typical programming languages cannot be modelled by simple generative systems, such as context-free grammars [2, 10]. The well-known limits of such models, together with the practical and theoretical interest of this modelling problem, have

*Gh. Paun, G. Rozenberg, A. Salomaa, eds.: "Discrete Mathematics and Computer Science. Papers in Memoriam Alexandru Mateescu (1952-2005)", The Publ. House of the Romanian Academy, Bucharest, 2015 (*in press*).

triggered the quest for more powerful generative systems, such as matrix languages, languages which verify certain pumping-type conditions, mildly context-sensitive languages, *indexed grammars* [1] or *ET0L systems* [8, 16].

Marcus [18] and Păun [7, 25] highlight the early results on these topics obtained by the Romanian computational linguistics school. Briefly, many context-free extensions have been shown inadequate; however, indexed grammars and ET0L systems still proved elusive.

Next, Vaida [30] and Vaida and Mateescu [33] discussed the enhanced *competence* of the indexed grammars: arguably, indexed languages form the largest "natural" class between context-free and context-sensitive languages. They mentioned that (at that time) finding a programming language construct which was beyond the limits of indexed grammars was still an *open* problem. However, they also cited a negative result due to Hayashi [13], regarding the competence limits of indexed grammars, suggesting that this could be used to provide an answer.

Then, using Hayashi's results, Nicolescu [22] and Nicolescu and Vaida [23] solved this open problem, showing that FORTRAN and other similar algorithmic languages allow constructions which cannot be modelled by either indexed grammars or ET0L systems. The first solution involves the required correspondence, in number, order and type, between arguments used in function calls with parameters appearing in their definition; but other possible constructions were also mentioned. Essentially, the proofs used constructions which can be *recursively nested*. These results were subsequently cited, but never formally published (except few bits).

Revisiting and extending these unpublished papers, we now show that the syntax and static semantics of Pascal, CAML, XML and other similar languages allow constructs which lie outside the competence limits of even sophisticated generative systems such as indexed grammars and ET0L systems.

The Pascal and CAML results are based on our earlier results [22, 23]; however, our XML results appear to be *novel*, despite the considerable practical interest raised by XML and XML schemas and the related extensive theoretical research, see e.g. [3–6, 15, 21].

In the following sections, we identify several syntactic constructions, based on *recursive nesting*, which exceed the competence limits of these two major context-free extensions: *indexed grammars* and *ET0L systems*. We then take a look at a few solved or still open problems, related to the classes of *iterative languages* and *generalised Ogden-like languages*, introduced by Mateescu and Vaida [20, 32].

Finally, we suggest that the contextual grammar model, introduced by Marcus [17] and further developed by the Romanian school on computational linguistics [19,27], could offer additional insight into some specific programming language constructs.

2 Background

Here, we use the term "Pascal-like" as a wide umbrella, covering imperative languages which, like Pascal, have a context-free phrase structure and a static semantics which defines clear rules on:

- the correspondence of arguments used in function calls with parameters used in corresponding function definitions, in number, order and types;
- the correspondence of integer indices used in array expressions with corresponding array declarations, in number;
- optionally, support for associative arrays or indexed properties, where the correspondence between array expressions and corresponding definitions extends to number, order and types of indices.

With this convention, our definition for Pascal-like language covers many mainstream languages, e.g. Pascal and its derivatives, C and its derivatives, managed languages such as Java and C#.

By CAML-like languages we understand an umbrella concept which covers an important segment of the functional languages derived from ML: with main members CAML, OCAML and F#.

In both cases. we consider an *ideal* scenario, when there are no bounds on the size of programs and of their elements, such as identifiers, parameter lists, array dimensionality.

By XML we understand valid W3C schema (XSD) based XML documents. We consider that, besides restricting the unconstrained anyType phrase structure, schemas roughly correspond to the declaration sections of Pascal programs. Therefore, we consider that schemas form an integral part of the documents which use them; this can be physically realized either by using inline schemas or concatenating each document with all its required schemas. Our constructions use only simple declarations allowed by both existing schema definitions: (i) the older, but still widely used, XSD 1.0 (2001, 2004), and (ii) the newer XSD 1.1 (2012); however, we note that more counter-examples can be created under XSD 1.1 rules. Again, we consider an *ideal* scenario, with no bounds on schema and document size or on their types and elements.

Under these assumptions, we shows that all these three language families, i.e. Pascal-like languages, CAML-like languages and the XML language, contain *recursively nested* constructs which map to the famous not indexed language: $\{(aw)^{|w|} \mid w \in T^*\}$ [13].

3 Indexed grammars and languages

In this section we briefly recall a few basic definitions and concepts on indexed grammars and languages, as required for the following sections. Here we use a notation inspired by the modern definitions used by Hopcroft and Ullman [14], which are equivalent to — but easier to use than — the original definitions used by Aho [1] and Hayashi [13] (for a quick introduction, see also the Wikipedia entry [35]).

Definition 3.1. An *indexed grammar* is a 5-tuple $G = (N, T, F, P, S)$ where:

- N is finite alphabet of *non-terminal* (*variable*) symbols
- T is finite alphabet of *terminal* symbols
- F is a finite set of *index* symbols (or indices)

- P is a finite set of *production rules* (briefly *productions* or *rules*), further classified into three different *types*, as discussed below
- $S \in N$ is the start symbol

An *indexed non-terminal* is an element of NF^*, typically written as $A[\sigma]$, where $A \in N, \sigma \in F^*$; in this context, σ is the *stack attached* to A. A *sentential form* α is an element of $(NF^* \cup T)^*$, typically written as $\alpha = X_1[\theta_1]X_1[\theta_2]\dots X_n[\theta_n]$, where $n \in \mathbb{N}$ and $(X_i \in N, \theta_i \in F^*) \vee (X_i \in T, \theta_i = \lambda), \forall i \in [1, n]$.

The *attachment* operation is extended from non-terminals to sentential forms, by distributing stacks over all non-terminals. Formally, for α defined as above and $\sigma \in NF^*$, $\alpha[\sigma] = X_1[\tau_1]X_1[\tau_2]\dots X_n[\tau_n]$, where $(X_i \in N, \tau_i = \theta_i\sigma) \vee (X_i \in T, \tau_i = \lambda), \forall i \in [1, n]$. For example, $(aB[fg]cD[]e)[h] = aB[fgh]cD[h]e$.

The three types of productions are defined in following way:

1. A production of *type 1* is an element of $N \times (N \cup T)^*$, often written as $A[..] \to \alpha[..]$, where $A \in N, \alpha \in (N \cup T)^*$.
2. A production of *type 2* (also known as a *push* rule) is an element of $N \times NF$, often written as $A[..] \to B[f..]$, where $A, B \in N, f \in F$.
3. A production of *type 3* (also known as a *pop* rule) is an element of $NF \times (N \cup T)^*$, often written as $A[f..] \to \alpha[..]$, where $A \in N, f \in F, \alpha \in (N \cup T)^*$.

Given a sentential form $\beta A[\phi]\gamma$, with $A \in N, \phi \in F^*, \beta, \gamma \in (NF^* \cup T)^*$, *direct derivations*, denoted by $\Rightarrow$, are defined separately for each type of production:

1. If $A[..] \to \alpha[..]$ is a type 1 production, then $\beta A[\phi]\gamma \Rightarrow \beta\alpha[\phi]\gamma$.
2. If $A[..] \to B[f..]$ is a type 2 (push) production, then $\beta A[\phi]\gamma \Rightarrow \beta B[f\phi]\gamma$.
3. If $A[f..] \to \alpha[..]$ is a type 3 (pop) production, then $\beta A[f\phi]\gamma \Rightarrow \beta\alpha[\phi]\gamma$.

Direct derivations are extended, in the usual way, to *transitive and reflexive* closures, denoted by $\Rightarrow^*$.

Definition 3.2. The language $L \subset T^*$ generated by the indexed grammar G is called an *indexed language* and is defined by: $L = L(G) = \{w \in T^* \mid S[] \Rightarrow^* w\}$.

Indexed languages form a "natural" class in an extended Chomski's hierarchy, larger than context-free but smaller than context-sensitive and are precisely the languages recognised by *nested stack automata* [1] (and by other equivalent formalisms), but we do not follow this discussion here.

Example 3.3. The language $\{a^n b^n c^n \mid n \in \mathbb{N}_+\}$ is a classical simple indexed language which is not context-free and can be generated by the following indexed grammar:

$$\begin{array}{lll} S[..] \to T[g..] & A[f..] \to (aA)[..] & A[g..] \to a[..] \\ T[..] \to T[f..] & B[f..] \to (bB)[..] & B[g..] \to b[..] \\ T[..] \to (ABC)[..] & C[f..] \to (cC)[..] & C[g..] \to c[..] \end{array}$$

For example, the string $a^2b^2c^2$ can be derived in the following way:

$$\begin{array}{lll} S[] & \Rightarrow & T[g] \Rightarrow T[fg] \Rightarrow (ABC)[fg] = A[fg]B[fg]C[fg] \\ & \Rightarrow^3 & (aA)[g](bB)[g](cC)[g] = aA[g]bB[g]cC[g] \Rightarrow^3 aa[]bb[]cc[] \\ & = & a^2b^2c^2. \end{array}$$

Remark 3.4. Applying the attachment definition for sentential forms, several rules of the above sample grammar are typically written more succinctly, e.g.:

$$\begin{array}{ll} T[..] \to (ABC)[..] & T[..] \to A[..]B[..]C[..] \\ A[f..] \to (aA)[..] & A[f..] \to aA[..] \\ A[g..] \to a[..] & A[g..] \to a \end{array}$$

Thus, the above derivation can also be written more succinctly:

$$\begin{array}{lll} S[] & \Rightarrow & T[g] \Rightarrow T[fg] \Rightarrow A[fg]B[fg]C[fg] \\ & \Rightarrow^3 & aA[g]bB[g]cC[g] \Rightarrow^3 a^2b^2c^2. \end{array}$$

Remark 3.5. Combining rules of type 1 and type 2 (push), one can obtain derivations that simulate the more complex *push* rules of the original definition proposed by Aho [1]. For example, the hypothetical rule $A \mapsto aB[fg..]bC[h..]c$ can simulated by the following rules (where $\bar{B}$, $\hat{B}$, and $\bar{C}$ are ad-hoc symbols): $A[..] \to a\bar{B}[..]b\bar{C}[..]c$, $\bar{B}[..] \to \hat{B}[g..]$, $\hat{B}[..] \to B[f..]$, $\bar{C}[..] \to C[h..]$.

To prove that a language is not indexed, one can use Hayashi's *pumping lemma* [13]. This lemma is quite complex and rather unwieldy for practical purposes. Instead, in the following sections, we will use one of its remarkable consequences and the fact that indexed languages are closed under several fundamental operations (the same result can also be obtained with the ulterior and easier to use Gilman's shrinking lemma [11]).

Fact 3.6. [13] For any alphabet T and symbol a, where $a \notin T \neq \Phi$, the language $L_{a,T} = \{(aw)^{|w|} \mid w \in T^*\}$ is not indexed.

Fact 3.7. As shown by Aho [1], the indexed languages form a *full abstract family of languages* (full AFL), and are therefore closed under any finite applications of the following operations: unions, concatenations, Kleene closures, intersections with regular sets, left and right quotients by regular sets, substitutions, homomorphisms, inverse homomorphisms, direct and inverse generalised sequential machine (GSM) mappings.

4 Pascal-like languages are not indexed

Let us consider three languages, L_0^P, L_1^P, L_2^P, defined as follows.

Let L_0^P be the set of all *syntactically correct* Pascal programs, according to its context-free phrase rules *and* its static syntax.

Intuitively, L_1^P highlights a valid scenario with recursively nested function calls. Let $L_1^P = \{Q_n^{\pi_1,\pi_2,\ldots,\pi_n} \mid n \in \mathbb{N}_+\}$, where each $Q_n^{\pi_1,\pi_2,\ldots,\pi_n}$ is the Pascal-like code shown in

```
program P;
  function F (π1, π2, ..., πn: integer): integer;
  begin F := 0 end;
  var V: integer;
begin
  V := F(                          {∗outer call to F, with n arguments∗}
        F(1, 1, ..., 1),           {∗1-st inner call to F, with n all 1 arguments∗}
        F(1, 1, ..., 1),           {∗2-nd inner call to F, with n all 1 arguments∗}
        ...,
        F(1, 1, ..., 1));          {∗n-th inner call to F, with n all 1 arguments∗}
  WriteLn(V);
end.
```

Figure 1: $Q_n^{\pi_1,\pi_2,...,\pi_n}$, a typical element in L_1^P.

Fig. 1, where $\pi_1, \pi_2, ..., \pi_n \in p^+$ are *pairwise distinct*, ellipses (...) are meta-syntactic symbols, and markers $\{* \ *\}$ delimit meta-comments, which are (true) assertions and do not appear in the actual code.

Intuitively, L_2^P is relaxed version of L_1^P, without any constraints on the size of the argument lists used in the recursively nested calls. Let $L_2^P = \{R_{n,m,k_1,k_2,...,k_m}^{\pi_1,\pi_2,...,\pi_n} \mid n, m, k_1, k_2, \ldots, k_m \in \mathbb{N}_+\}$, where each $R_{n,m,k_1,k_2,...,k_m}^{\pi_1,\pi_2,...,\pi_n}$ is the Pascal-like code shown in Fig. 2, where $\pi_1, \pi_2, ..., \pi_n \in p^+$, ellipses (...) are meta-syntactic symbols, and markers $\{* \ *\}$ delimit meta-comments, which are (true) assertions and do not appear in the actual code.

```
program P;
  function F (π1, π2, ..., πn: integer): integer;
  begin F := 0 end;
  var V: integer;
begin
  V := F(                          {∗outer call to F, with m arguments∗}
        F(1, 1, ..., 1),           {∗1-st inner call to F, with k1 all 1 arguments∗}
        F(1, 1, ..., 1),           {∗2-nd inner call to F, with k2 all 1 arguments∗}
        ...,
        F(1, 1, ..., 1));          {∗m-th inner call to F, with km all 1 arguments∗}
  WriteLn(V);
end.
```

Figure 2: $R_{n,m,k_1,k_2,...,k_m}^{\pi_1,\pi_2,...,\pi_n}$, a typical element in L_2^P.

Lemma 4.1. $L_1^P \subset L_0^P$ (i.e. all items in L_1^P are correct Pascal programs).

Proof. Each $Q_n^{\pi_1,\pi_2,...,\pi_n} \in L_1^P$ follows Pascal's phrase structure. Additionally, it is semantically correct, because: (i) parameters are given by distinct identifiers; (ii) each of the n inner calls to F has exactly n integer arguments; and (iii) the outer call to F has exactly n integer arguments. □

Lemma 4.2. $L_1^P \subset L_2^P$.

Proof. Straightforward, as each $Q_n^{\pi_1,\pi_2,\ldots,\pi_n} \in L_1^P$ appears as $R_{n,n,n,n,\ldots,n}^{\pi_1,\pi_2,\ldots,\pi_n} \in L_2^P$. □

Lemma 4.3. $L_1^P \subset L_0^P \cap L_2^P$.

Proof. Straightforward, from Lemmas 4.1 and 4.2. □

Lemma 4.4. $L_2^P \cap L_0^P \subset L_1^P$.

Proof. Let us consider an item $\rho = R_{n,m,k_1,k_2,\ldots,k_m}^{\pi_1,\pi_2,\ldots,\pi_n} \in L_2^P$ and assume that it also appears in L_0^P, i.e. it is semantically correct. Therefore: (i) parameter identifiers must be pairwise distinct; (ii) each of the m inner calls to F must have n arguments, thus $k_1 = k_2 = \cdots = k_m = n$; and (iii) the outer call to F must have n arguments, thus $m = n$. Consequently, $\rho = Q_n^{\pi_1,\pi_2,\ldots,\pi_n} \in L_1^P$. □

Lemma 4.5. $L_1^P = L_0^P \cap L_2^P$.

Proof. Straightforward, from Lemmas 4.3 and 4.4. □

Lemma 4.6. L_2^P is regular.

Proof. Briefly, all repetitive fragments which appear in elements of L_2^P can be independently generated by Kleene closures. More formally, $L_2^P = L_3^P$, where L_3^P is the language generated by the regular expression of Fig. 3, where brackets
([ ]) are meta-syntactic grouping symbols. □

```
program P;
  function F ([p+, ]*p+: integer): integer;
  begin F := 0 end;
  var V: integer;
begin
  V := F(
[         F(1, 1, ..., 1),]*
         F(1, 1, ..., 1));
  WriteLn(V);
end.
```

Figure 3: The regular expression defining the language L_3^P.

Theorem 4.7. The set of syntactically correct Pascal programs, L_0^P, is not an indexed language.

Proof. By contradiction, let us assume that L_0^P is an indexed language. By Lemmas 4.5 and 4.6, L_1^P is the intersection of an indexed language, L_0^P, with a regular language, L_2^P. By Fact 3.7, L_1^P must be an indexed language.

Consider now a homomorphism, h, which maps 'F' to a, '1' to b, and all other symbols to λ. Applying h to L_1^P, we obtain $L_1' = h(L_1^P) = \{aaa(ab^n)^n \mid n \in \mathbb{N}_+\}$. By Fact 3.7, L_1' must be an indexed language.

Finally, we define a simple GSM mapping, g, which translates L_1' into the language $g(L_1') = L_{a,\{b\}} = \{(aw)^{|w|} \mid w \in b^*\}$ of Fact 3.6 (this process is straightforward, so we leave details out). Alternatively, we can take the left-quotient by the regular set $\{aaa\}$, to obtain the same result. By Fact 3.7, L_1' must also be an indexed language, but it is not. Therefore, our initial assumption was invalid, and L_0^P cannot be an indexed language. □

Remark 4.8. Theorem 4.7 solely relies on the following closure operators on indexed languages: intersection with regular sets, λ-homomorphisms, and left-quotient with regular sets or GSM mappings. Therefore, a similar result can be obtained in a straightforward way for any family of languages which: (i) does not contain $\{(aw)^{|w|} \mid w \in T^*\}$; and (ii) is closed under the same operations, in particular if it is a full AFL.

Remark 4.9. Nicolescu and Vaida [22, 23] present a list of several other programming constructs which cannot be modelled by indexed grammars. For example, one can also start with an n-dimensional array, initialize it properly, and then access it via a nested construction, as in the sample code of Fig 4. We leave the details as an exercise.

```
program P;
  var A: array [1..1, 1..1, ..., 1..1] of integer;
  var V: integer;
begin
  A[1, 1, ..., 1] := 1;
  V := A[
        A[1, 1, ..., 1],
        A[1, 1, ..., 1],
        ...,
        A[1, 1, ..., 1]];
  WriteLn(V);
end.
```

Figure 4: Array-based alternative example (Pascal).

Remark 4.10. Note that the proof Theorem 4.7 maps correct Pascal into a $L_{a,T} = \{(aw)^{|w|} \mid w \in T^*\}$ language, where T is a singleton set, $T = \{b\}$. If needed (for various other reasons), Nicolescu and Vaida [22,23] indicate how to map correct Pascal into more complex members of the $L_{a,T}$ family, where T contains any number of letters, $s \geq 1$, by using explicit conversions between otherwise incompatible types.

Figure 5 shows such a stronger version of the earlier introduced $Q_n^{\pi_1,\pi_2,\ldots,\pi_n}$, for $s = 2$. We redefine function F to use an arbitrary mix of integers and reals, which must be strictly followed by all function calls, i.e. we set the additional constraints $(\alpha_i = \textbf{integer} \wedge \phi_i = \textbf{trunc} \wedge \pi_i = \textbf{trunc}(1.0)) \vee (\alpha_i = \textbf{real} \wedge \phi_i = \lambda \wedge \pi_i = 1.0)$. We recall that Pascal

requires explicit real to integer conversions and we leave the details as an exercise for the interested reader.

Similar results can be obtained with associative arrays, if the language supports this; e.g. C# offers associative arrays as indexed properties.

```
program P;
  function F (π1 : α1 , π2 : α2 , ... , πn : αn ): real;
  begin F := 0.0 end;
  var V: real;
begin
  V := F(
        φ1(F(π1 , π2 , ... , πn )) ,
        φ2(F(π1 , π2 , ... , πn )) ,
        ... ,
        φn(F(π1 , π2 , ... , πn )));
  WriteLn(V);
end.
```

Figure 5: A stronger version of $Q_n^{\pi_1,\pi_2,\ldots,\pi_n}$, with additional constraints on parameters.

5 CAML-like languages are not indexed

In this section we extend our quest from imperative to functional languages. Specifically, we discuss recursively nested constructs, similar to those used in Section 4, which are beyond the competence limits of indexed grammars and appear in the CAML family of functional languages. We restrict our attention to F#, which is a .NET integrated variant of OCAML; however, the F# constructs can be straightforwardly transferred to the other major members of the ML family, OCAML and CAML itself, by only minor syntactic changes.

Figure 6 shows a typical element of set L_1^F, i.e. of an F# version of the L_1^P language defined in Section 4 (with similar context constraints, which are not repeated here).

```
let F (π1:int) (π2:int) ... (πn:int) = 0
let V:int =
  F (F 1 1 ... 1)  // 1
    (F 1 1 ... 1)  // 2
    ...
    (F 1 1 ... 1)  // n
printfn "%A" V
```

Figure 6: Typical element of language L_1^F (F#).

The above snippet is a syntactically correct F# program and maps to the same non-indexed language as in Section 4, i.e. $\{(aw)^n \mid w \in T^*, n = |w|\}$. Therefore, we obtain the following theorem:

Theorem 5.1. The set of syntactically correct CAML programs is not an indexed language.

As suggested in Fig. 7, the same result can also be obtained by starting with an F# analogue of the nested array construct mentioned in Remark 4.9.

```
let A =  // n repetitions
  Array.create 1 (Array.create 1 (... (Array.create 1 0) ...))
let V:int =
  A
    .[A.[0].[0] ... .[0]]  // 1
    .[A.[0].[0] ... .[0]]  // 2
    ...
    .[A.[0].[0] ... .[0]]  // n
printfn "%d" V
```

Figure 7: Array-based alternative example (F#).

Interestingly, if we drop the explicit type of result V, then V's static type is determined by the context and the array expression can be reduced from n lines to *any* number of lines $m \leq n$. As shown in Fig. 8, the last print statement can be kept, if we replace the integer %d placeholder by the *generic* %A placeholder.

```
let A =  // n repetitions
  Array.create 1 (Array.create 1 (... (Array.create 1 0) ...))
let V =
  A
    .[A.[0].[0] ... .[0]]  // 1
    .[A.[0].[0] ... .[0]]  // 2
    ...
    .[A.[0].[0] ... .[0]]  // m ≤ n !
printfn "%A" V
```

Figure 8: A more interesting array-based example (F#).

In the end, we obtain a potentially more interesting language, $\{(aw)^m \mid w \in T^*, m \in [0, |w|]\}$.

We note that the construction used in Remark 4.10 can be repeated here, to obtain the same counter-example based on s letters. We leave this as an exercise for the interested reader.

6 XML is not indexed

In this section, we continue our quest and we discuss nested constructs, similar to those used in Section 4, which appear in valid XSD based XML documents and are beyond the competence limits of indexed grammars.

Figures 9 and 10 show a typical element of set L_1^X, i.e. of an XML version of the L_1^P language defined in Section 4. This element is presented as the concatenation of *two* parts: (i) Fig. 9 shows the essential fragment of an XML schema definition (either XSD 1.0 or XSD 1.1); and (ii) Fig. 10 shows the essential fragment of the document instance; for brevity, here we skip the verbose XSD and XML prologues and epilogues; however, we give full codes in the Appendix.

```
<xs:complexType name="RecType">
  <xs:sequence>
    <xs:element name="R" type="RecType" nillable="true"/>
    <xs:element name="R" type="RecType" nillable="true"/>
    ...
    <xs:element name="R" type="RecType" nillable="true"/>
  </xs:sequence>
</xs:complexType>
<xs:element name="R" type="RecType"/>
```

Figure 9: Typical element of L_1^X: the essential part of its XML schema definition.

```
<R>
  <R xsi:nil="true"/>
  <R xsi:nil="true"/>
  ...
  <R xsi:nil="true"/>
</R>
<R>
  <R xsi:nil="true"/>
  <R xsi:nil="true"/>
  ...
  <R xsi:nil="true"/>
</R>
...
<R>
  <R xsi:nil="true"/>
  <R xsi:nil="true"/>
  ...
  <R xsi:nil="true"/>
</R>
```

Figure 10: Typical element of L_1^X: the essential part of its XML document instance.

Adding the omitted standard prologues and epilogues (cf. Appendix), the above snippet becomes a syntactically valid XML document and can be straightforwardly mapped to the same non-indexed language as in Section 4, i.e. to $\{(aw)^n \mid w \in T^*, n = |w|\}$. Therefore, we obtain the following theorem:

Theorem 6.1. The set of syntactically valid XML documents is not an indexed language.

Here we have used a simple nillable recursive complex type. Other constructs that lead to the same non-indexed language (or even more complex variants) can be based on more sophisticated schema ingredients, such as:

1. key constraints, i.e. unique, key, and keyref schema elements (available in both XSD 1.0 and XSD 1.1)
2. assertions, i.e. assert schema elements (only available in XSD 1.1)

However, because of space constraints, we leave these as an exercise.

7 ET0L systems and languages

In this section we briefly recall a few basic definitions and concepts on ET0L systems and languages, as required for discussing the results.

Definition 7.1. An *ET0L* system is a 4-tuple $G = (N, T, P, \omega)$ where:

- N is finite alphabet of *non-terminal* (*variable*) symbols
- T is finite alphabet of *terminal* symbols
- P is a finite set of *tables* (or *substitutions*), further discussed below
- $\omega \in (N \cup T)^*$ is the start sequence

Each table $\Pi \in P$ is a finite left-total relation on $(N \cup T) \times (N \cup T)^*$, i.e. (i) Π is a finite set of pairs $(X, \alpha) \in (N \cup T) \times (N \cup T)^*$, called *production rules* (briefly *productions* or *rules*), typically written as $X \to \alpha$; and (ii) for each symbol $X \in (N \cup T)$, there is an $\alpha \in (N \cup T)^*$ such that $X \to \alpha \in \Pi$.

Direct derivations between sentential forms are denoted, as usually, by $\Rightarrow$ and are defined in a total parallel mode. Let us consider $\alpha, \beta \in (N \cup T)^*$ and $k \in \mathbb{N}$, such that $\alpha = X_1 X_2 \dots X_k$, with $X_i \in (N \cup T), \forall i \in [1, k]$. Then:

$$\alpha \Rightarrow \beta \iff \exists \Pi \in P, \forall i \in [1, k], \exists X_i \to \gamma_i \in \Pi : \beta = \gamma_1 \gamma_2 \dots \gamma_k.$$

Direct derivations are extended in the usual way to *transitive and reflexive* closures, denoted by $\Rightarrow^*$.

Definition 7.2. The language $L \subset T^*$ generated by the ET0L system G is called an *ET0L language* and is defined by: $L = L(G) = \{w \in T^* \mid \omega \Rightarrow^* w\}$.

Example 7.3. The language $\{a^n b^n c^n \mid n \in \mathbb{N}_+\}$ is a classical simple ET0L language which is not context-free and can be generated by the ET0L system $G = (\{A, B, C\}, \{a, b, c\}, \{\Pi_1, \Pi_2\}, ABC)$, where:

$$\Pi_1 = \{A \to aA, B \to bB, C \to cC, a \to a, b \to b, c \to c\}$$
$$\Pi_2 = \{A \to a, B \to b, C \to c, a \to a, b \to b, c \to c\}$$

For example, the string $a^2b^2c^2$ can be derived in the following way:

$$ABC \Rightarrow aAbBcC \Rightarrow aabbcc = a^2b^2c^2.$$

ET0L languages form an interesting class which is strictly larger than context-free, but strictly smaller than indexed [9], and a fortiori smaller than context-sensitive. There are several specific ways to prove that a given language is not ET0L, such as: (i) the inclusion of the ET0L class in the indexed or context-sensitive class [9]; (ii) the "rare/frequent" theorem [8]; or (iii) an Ogden lemma for ET0L languages [28].

Fact 7.4. Using either of these, one can prove that the language $L_{a,T} = \{(aw)^{|w|} \mid w \in T^*\}$, defined in Fact 3.6, is not ET0L either.

Fact 7.5. As shown by Rozenberg [29], the ET0L languages form an AFL, therefore, this class enjoys all closure properties mentioned in Fact 3.7.

7.1 Pascal-like languages, CAML-like languages and XML are not ET0L

By way of Facts 7.4 and 7.5 and Remark 4.8, one can adapt Theorems 4.7, 5.1 and 6.1 to conclude that:

Theorem 7.6. The set of syntactically correct Pascal programs, the set of syntactically correct CAML programs and the set of syntactically valid XML documents are not ET0L languages.

However, this result is not totally satisfactory, because we relied on a counter-example language, $\{(aw)^{|w|} \mid w \in T^*\}$, which is not indexed (i.e. not even in a strictly larger class). Can one find tighter counter-examples, based on languages which are in the non-void border between ET0L and indexed? We formulate the following open question:

Open question 7.7. Are there programming language constructs which map to a language which is indexed but not ET0L?

8 Iterative and GOC-like languages

In this section we follow the earlier works of Mateescu and Vaida [20, 30–33], which propose two novel classes of grammar-less languages – the iterative languages and the GOC-like languages – and discuss their relation to programming languages.

The family of *iterative languages* is defined solely by the verification of a language-based constraint, which is more general than the conclusion of the classical Ogden's iteration theorem [24] and similar to Greibach's strong iteration condition [12].

Let $\delta(z)$ denote the number of distinguished positions in a word z, and $\epsilon(z)$ the number of excluded positions in z.

Definition 8.1. [20] A language L is n-iterative if there exist integers $n \geq 2, p \geq 1, q \geq 1$, such that any word $z \in L$, with $\delta(z) \geq p$, has a factorisation $z = x_1y_1x_2y_2 \ldots\ x_ny_nx_{n+1}$, satisfying:

- $z_m = x_1y_1^m x_2y_2^m \ldots x_ny_n^m x_{n+1} \in L, \forall m \in \mathbb{N}$;
- $\exists i \in \mathbb{N}, 2 \leq i \leq n$, with
  - $\delta(y_{i-1}x_iy_i) \leq q$;
  - $(\delta(x_{i-1}), \delta(y_{i-1}), \delta(x_i) \geq 1)$ or $(\delta(x_i), \delta(y_i), \delta(x_{i+1}) \geq 1)$.

Next, Mateescu and Vaida [20] proved that the family of iterative languages is *closed* under several fundamental operations.

Theorem 8.2. [20] The family of iterative languages is closed under union and concatenation.

Proof. (Sketch) Let us consider two iterative languages, L_1 and L_2, where each L_h is an n_h-iterative language with constants p_h and q_h, for $h = 1, 2$. Then:

- The union, $L_1 \cup L_2$, is n-iterative, for $n = \max(n_1, n_2)$, $p = \max(p_1, p_2)$, $q = \max(q_1, q_2)$.
- The product, $L_1 \cdot L_2$, is n-iterative, for $n = \max(n_1, n_2)$, $p = p_1 + p_2 - 1$, $q = \max(q_1, q_2)$.

□

Theorem 8.3. [20] The family of iterative languages is closed under Kleene closure (star).

Proof. (Sketch) Let us consider L, an n-iterative language with constants p and q. Then, its Kleene closure, L^*, is n-iterative, for $p' = 2p$, $q' = \max(2(p-1), q)$. □

Theorem 8.4. [20] The family of iterative languages is closed under substitution.

Proof. (Sketch) Let us consider:

- L_0, an n_0-iterative language with constants p_0 and q_0, over a size k alphabet $T = \{a_1, a_2, \ldots, a_k\}$;
- $\mathcal{L} = \{L_h \mid h \in [1, k]\}$, a family of iterative languages, where each L_h is an n_h-iterative language with constants p_h and q_h, for $h \in [1, k]$;
- a function, $f : T \to \mathcal{L}$, $f(a_h) = L_h$, $h \in [1, k]$.

Then, the substitution, $f(L_0)$, is n-iterative, for:

- $n = \max\{n_h \mid h \in [0, k]\}$;
- $p = p_0 \cdot \max\{p_h \mid h \in [1, k]\}$;
- $q = \max\{(p_h - 1)q_0, q_h \mid h \in [1, k]\}$.

□

Theorem 8.5. [20] The family of iterative languages is closed under homomorphic replication (as defined by Greibach [12]).

Further, Mateescu and Vaida [20] defined a *generalised Ogden condition* (GOC) and proved that the family GOC-like languages transcend the classical Chomski hierarchy.

Definition 8.6. [20] A language L is GOC-like if $\exists k \in \mathbb{N}$, such that any word $z \in L$, with $\delta(z) > k^{\epsilon(z)+1}$, has a factorisation $z = uvwxy$, satisfying:

- $\delta(vwx) \leq k^{\epsilon(vwx)+1}$;
- $d(uvw) + \delta(wxy) \geq 1 \wedge \epsilon(vx) = 0$;
- $z_m = uv^m wx^m y \in L, \forall m \in \mathbb{N}$.

Theorem 8.7. [20] There exists properly context-sensitive, properly recursive, properly recursively enumerable and properly non-recursively enumerable languages which are GOC-like.

The following results [20] indicate the position of these two novel classes against the well-known family of n-locally linear languages [34].

Theorem 8.8. [20] The class of iterative languages is included in the class of locally linear languages; more precisely: each n-iterative language is n-locally linear.

Theorem 8.9. [20] The class of GOC-like languages is included in the class of locally linear languages; more precisely: each GOC-like language is 2-locally linear.

Based on the above inclusions, Theorems 8.8 and 8.9, the following result [20] shows that neither of these classes can fully characterise all programming languages constructs.

Theorem 8.10. [20] Pascal-like programming languages are not locally linear.

```
program P:
  procedure V^k;
  begin end;
begin
   V^k;  V^k;  ...  ;  V^k
end.
```

Figure 11: Typical format of Pascal programs used in the proof of Th. 8.10.

Proof. (Sketch) The original proof [20] uses PL/I. Here we show an analogous construct in Pascal, based on the family of syntactically correct Pascal programs shown in Fig. 11, for $k \in \mathbb{N}_+$.

The proof is direct, without relying on any closure properties. □

Theorem 8.11. [20] Pascal-like programming languages are neither interative nor GOC-like.

Remark 8.12. [20] The families of simple matrix languages and of equal matrix languages are both homomorphic replications of locally linear languages. By Theorem 8.10, it follows that programming languages are not members of these families either. For a similar result, see Păun [26].

Open question 8.13. We list a few remaining open problems, related to the families of interative and GOC-like languages:

- Is the inclusion of Theorem 8.8 strict? If yes, can we find programming language constructs that map to their difference?
- Is the inclusion of Theorem 8.9 strict? If yes, can we find programming language constructs that map to their difference?
- What is the relation between 2-iterative and GOC-like languages? If these are not equal, can we find programming language constructs that map to their difference(s)?

9 Conclusions

We revisit our earlier unpublished results regarding the competence limits of indexed grammars and ET0L systems as models for the full syntax of programing languages. We show that usual programing languages are neither indexed nor ET0L. We extend these results by showing a novel result: that schema based XML documents are neither indexed nor ET0L. We revisit our earlier less published results and show that programing languages are not iterative, not GOC-like, and not locally linear. Finally, we formulate a few open questions.

Acknowledgments. We are indebted to the anonymous reviewers for their valuable comments and suggestions.

References

[1] A. V. Aho. Indexed grammars - an extension of context-free grammars. *J. ACM*, 15(4):647–671, 1968.

[2] A. V. Aho and J. D. Ullman. *The Theory of Parsing, Translation, and Compiling*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1972.

[3] M. Arenas, W. Fan, and L. Libkin. What's hard about XML schema constraints? In A. Hameurlain, R. Cicchetti, and R. Traunmüller, editors, *DEXA*, volume 2453 of *Lecture Notes in Computer Science*, pages 269–278. Springer, 2002.

[4] M. Arenas, W. Fan, and L. Libkin. Consistency of XML specifications. In L. E. Bertossi, A. Hunter, and T. Schaub, editors, *Inconsistency Tolerance*, volume 3300 of *Lecture Notes in Computer Science*, pages 15–41. Springer, 2005.

[5] J. Berstel and L. Boasson. Formal properties of XML grammars and languages. *Acta Inf.*, 38(9):649–671, 2002.

[6] A. Brüggemann-Klein and D. Wood. Balanced context-free grammars, hedge grammars and pushdown caterpillar automata. In *Extreme Markup Languages®*, 2004.

[7] C. Cîşlaru and G. Păun. Classes of languages with Bar-Hillel, Perles and Shamir's property. *Math. Soc. Sci. Math. Roumanie*, 18(66):273–278, 1975.

[8] A. Ehrenfeucht and G. Rozenberg. On proving that certain languages are not ET0L. *Acta Inf.*, 6:407–415, 1976.

[9] A. Ehrenfeucht, G. Rozenberg, and S. Skyum. A relationship between ET0L and EDT0L languages. *Theoretical Computer Science*, 1(4):325 – 330, 1976.

[10] R. W. Floyd. On the nonexistence of a phrase structure grammar for ALGOL 60. *Commun. ACM*, 5(9):483–484, Sept. 1962.

[11] R. H. Gilman. A shrinking lemma for indexed languages. *Theor. Comput. Sci.*, 163(1&2):277–281, 1996.

[12] S. A. Greibach. The strong independence of substitution and homomorphic replication. *ITA*, 12(3), 1978.

[13] T. Hayashi. On derivation trees of indexed grammars — an extension of the uvwxyz theorem. *Publ. RIMS, Kyoto Univ.*, 9(1):61–92, 1973.

[14] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[15] D. Lee and W. W. Chu. Comparative analysis of six XML schema languages. *SIGMOD Record*, 29(3):76–87, 2000.

[16] A. Lindenmayer and G. Rozenberg. Developmental systems and languages. In *Proceedings of the Fourth Annual ACM Symposium on Theory of Computing*, STOC '72, pages 214–221, New York, NY, USA, 1972. ACM.

[17] S. Marcus. Contextual grammars. *Revue Roumaine de Mathématiques Pures et Appliquées*, 14(12):1525–1534, 1969.

[18] S. Marcus. Linguistics for programming languages. *Rev. Roum. Ling. – Cahiers Ling. Theor. Appl.*, 16(1):29–38, 1979.

[19] S. Marcus. *Words and Languages Everywhere.* Polimetrica International Scientific Publisher, Milano, Italy, 2007.

[20] A. Mateescu and D. Vaida. *Structuri Matematice Discrete. Applicatii (Discrete Mathematical Structures. Applications).* Editura Academiei, Bucharest, Romania, 1989.

[21] M. Murata, D. Lee, M. Mani, and K. Kawaguchi. Taxonomy of XML schema languages using formal language theory. *ACM Trans. Internet Techn.*, 5(4):660–704, 2005.

[22] R. Nicolescu. *Limbaje Formale şi Limbaje de Programare (Formal Languages and Programming Languages).* PhD thesis, University of Bucharest, Faculty of Mathematics, 1985.

[23] R. Nicolescu and D. Vaida. Limbaje de programare, limbaje indexate şi limbaje ET0L (Programming languages, indexed languages and ET0L languages). In *Lucrările celui de al V-lea Colocviu de Informatică (Proceedings of the V-th Colloquium on Informatics)*, INFO-IAŞI'85, pages 220–229, Iaşi, Romania, 1985. University A.I.Cuza.

[24] W. F. Ogden. A helpful result for proving inherent ambiguity. *Mathematical Systems Theory*, 2(3):191–194, 1968.

[25] G. Păun. Asupra proprietăţii Bar-Hillel, Perles şi Shamir (Romanian). *Stud. Cerc. Matem.*, 28(3):303–309, 1976.

[26] G. Păun. *Matrix Grammars (Romanian).* Editura Ştiinţifică şi Enciclopedică, Bucharest, Romania, 1981.

[27] G. Păun. *Contextual Grammars (Romanian).* The Publishing House of the Romanian Academy, Bucharest, Romania, 1982.

[28] M. Rabkin. Ogden's lemma for ET0L languages. In *LATA*, pages 458–467, 2012.

[29] G. Rozenberg. Extension of tabled 0L-systems and languages. *International Journal of Parallel Programming*, 2(4):311–336, 1973.

[30] D. Vaida. Condiţii de iterare de tipul Ogden şi aplicaţii la limbajele de programare (Ogden-like iteration conditions and applications to programming languages). In *Lucrările celui de al III-lea Colocviu de Informatică (Proceedings of the III-th Colloquium on Informatics)*, INFO-IAŞI'83, pages 551–587, Iaşi, Romania, 1983. University A.I.Cuza.

[31] D. Vaida. Observaţie asupra limbajelor tare iterative (Remark concerning strong iterative languages). In *Lucrările celui de al V-lea Colocviu de Informatică (Proceedings of the V-th Colloquium on Informatics)*, INFO-IAŞI'85, pages 210–219, Iaşi, Romania, 1985. University A.I.Cuza.

[32] D. Vaida. Iteration conditions of W. Ogden's type and applications to programming languages (II). In *Developments in Language Theory*, pages 44–50, 1993.

[33] D. Vaida and A. Mateescu. *Limbaje Formale si Tehnici de Compilare — Capitole Speciale de Limbaje Formale*. Universitatea din Bucureşti, Bucharest, Romania, 1984.

[34] A. P. J. van der Walt. Locally linear families of languages. *Information and Control*, 32(1):27–32, 1976.

[35] Wikipedia. Indexed grammar — Wikipedia, the free encyclopedia, 2014. [Online; accessed 4-May-2014] `http://en.wikipedia.org/w/index.php?title=Indexed_grammar&oldid=604802542`.

A Appendix

Figures 12 and 13 (respectively) show the full XSD 1.0 and XML code of a typical element of language L_1^X, defined in Section 6.

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema elementFormDefault="qualified"
    targetNamespace="http://recursion.org/"
    xmlns="http://recursion.org/"
    xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:complexType name="RecType">
    <xs:sequence>
      <xs:element name="R" type="RecType" nillable="true"/>
      <xs:element name="R" type="RecType" nillable="true"/>
      ...
      <xs:element name="R" type="RecType" nillable="true"/>
    </xs:sequence>
  </xs:complexType>

  <xs:element name="R" type="RecType"/>
</xs:schema>
```

Figure 12: Full XSD schema definition (see Sec. 6).

```
<?xml version="1.0"?>
<R  xmlns="http://recursion.org/"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" >

  <R>
    <R xsi:nil="true"/>
    <R xsi:nil="true"/>
    ...
    <R xsi:nil="true"/>
  </R>
  <R>
    <R xsi:nil="true"/>
    <R xsi:nil="true"/>
    ...
    <R xsi:nil="true"/>
  </R>
  ...
  <R>
    <R xsi:nil="true"/>
    <R xsi:nil="true"/>
    ...
    <R xsi:nil="true"/>
  </R>
</R>
```

Figure 13: Full XML document instance (see Sec. 6).