The Romanian Journal of Analytic Philosophy
Vol. VII, 1°, 2013, pp. 5-18

ON THE GRAMMAR OF FIRST-ORDER LOGIC

Mihai GANEA*

Abstract. Determining the computational requirements of formal reasoning is a key task in implementing David Hilbert's foundational program. This paper presents a number of results on the syntactical complexity of some versions of the language of first-order logic (L_{FOL}), some of which are already known. Two versions of L_{FOL} with inefficient (tally) indexing are shown to be context-free, and the sets of their sentences (the sub-language $L_{S\text{-}FOL}$) and a version of L_{FOL} with efficient (positional) indexing are shown to be not context-free. The latter is not even the intersection of a finite number of context-free languages but is in *D-LogSpace* and consequently rudimentary in Smullyan's sense.

Keywords: Hilbert's Program; formal grammar; first-order logic; context-free language; pebble automaton; logarithmic space.

It has been argued that first-order logic (*FOL*) is the fundamental logical theory (see Quine 1986 and Wagner 1987). Essentially this argument is based on the fact that the language of first-order logic (L_{FOL}) has significant expressive power (sufficient for formalizing mathematical reasoning) and admits a natural semantics whose set of logical truths is axiomatizable. While the semantical properties of L_{FOL} have been extensively investigated by model theorists, its syntactical properties have not received the same degree of attention. Nevertheless, the syntax of L_{FOL} is not devoid of philosophical interest. Elsewhere I have argued that the guiding idea of David Hilbert's foundational program is to derive the basic notions and principles of finitistic arithmetic from an analysis of the computational requirements of formalizing mathematical practice (see Ganea 2010). Numbers are to be identified with a special kind of expressions and then their basic theory is to be selected as the optimal description of the syntactical competence with respect to those expressions. Given the central role that *FOL* has in the formalization of

* Mihai GANEA, Department of Philosophy, University of Toronto.
Email: mihai.ganea@utoronto.ca.

mathematics, determining the computational complexity of the various notions and operations presupposed by the syntax of L_{FOL} is an important task in the implementation of Hilbert's Program. A basic problem in this respect is the position of the various versions of L_{FOL} within Chomsky's hierarchy and the original purpose of the present paper was to answer three questions raised by Harold Levin about this issue[1].

A *language* L is a sub-set of the set of finite strings of symbols from a finite alphabet X, $L \subseteq X^*$. If x, y are in X^* then $x^\wedge y$ is the string obtained by the concatenation of x and y, $|x|$ is the length of x (the number of distinct symbol occurrences in it), and x^n is $x^\wedge x^\wedge \ldots ^\wedge x$ (n copies of x concatenated). It is usually assumed that any language L includes the null symbol ε which acts as a neutral element with respect to the operation of concatenation, i.e. $x^\wedge \varepsilon = \varepsilon^\wedge x = x$ for any string of symbols x. When context permits, the concatenation sign will be omitted, i.e. $x^\wedge y$ will be written simply as xy.

A *grammar* G is a quadruple (X, Y, S, P), where X and Y are disjoint finite alphabets, and

(i) X is the *terminal alphabet* for the grammar;

(ii) Y is the *non-terminal alphabet* for the grammar, and its elements are also called *variables*;

(iii) $S \in Y$ is the *start symbol* for the grammar;

(iv) P is the grammar's set of *productions. P* is a finite set of pairs (v, w) with $v \in (X \cup Y)^*$ containing a least one non-terminal symbol, while w is an arbitrary element of $(X \cup Y)^*$. An element (v, w) of P is usually written $v \rightarrow w$.

Let G be a grammar and y, z in $(X \cup Y)^*$. We write $y \Rightarrow z$ and say that y *directly derives* z, if z can be obtained from y by replacing an occurrence in y of the left side of some production by its right side (i.e. G has a production $v \rightarrow u$ such that y can be written as pvr and z as pur with p, r in $(X \cup Y)^*$). $\Rightarrow^*$ is the reflexive transitive closure of $\Rightarrow$.

The language generated by a grammar G, $L(G)$, is the set of terminal strings derivable from its start symbol:

$L(G) = \{w : w \in X^* \text{ and } S \Rightarrow^* w\}$.

A grammar G is i) *regular* if all its productions have the form $A \rightarrow aB$ or $A \rightarrow B$ where A, B are variables and a is a terminal symbol; ii) *context-free* if all its productions have the form $A \rightarrow u$ where A is a variable and u an arbitrary string; iii) *context-sensitive* if all its productions are *length-increasing*, i.e. if $v \rightarrow u \in P$, then $|u| \geq |v|$ and iv) *unrestricted* if no constraint is placed on its productions. A language L is regular, context-free, context-sensitive or recursively enumerable if it can be generated by a regular, context-free, context-sensitive or unrestricted grammar. These kinds of languages form the levels of Chomsky's hierarchy, which are known to be ordered by strict inclusion[2]. Given a formal

[1] The questions I am addressing are formulated in note 5 to Chapter 1 of Levin 1982, pp. 55-56.

[2] The basic notions concerning formal languages are well described throughout the first four parts of Martin 1997 and the Chomsky hierarchy is introduced in §11.5 (pp. 327-330).

language, it is natural to ask about its position within this hierarchy. In particular, is the language of first-order logic (L_{FOL}) context-free?

As it is, the question is not sufficiently precise. We do not have a standard definition of L_{FOL} since some authors prefer the Polish notation for logical operators and some do not; some use distinct symbols for free and bound variables and some do not. Furthermore, many authors take the alphabet of L_{FOL} to be infinite, containing at least infinitely many individual variable symbols, which *prima facie* renders the above question meaningless. These worries can however be easily removed. Instead of assuming infinitely many primitive symbols, one can start with a finite alphabet and generate infinitely many distinct individual variables by indexing. A similar procedure can be used to deal with individual constants, functional constants and relation constants. I will take L_{FOL} to be the language over $X = \{v, c, f, R, a, b, \neg, \wedge, \forall\}$ defined below.

Definition 1.

(i) x is an individual variable if and only if there exists $n > 0$ such that $x = va^n$.

(ii) x is an individual constant if and only if there exists $n > 0$ such that $x = ca^n$.

(iii) x is a functional expression if and only if there exist $n, m > 0$ such that $x = fa^nb^m$.

(iv) x is a relational expression if and only if there exist $n, m > 0$ such that $x = Ra^nb^m$.

(v) x is a term if and only if a) x is an individual variable or b) x is an individual constant or c) there exists a functional expression fa^nb^m and m terms $t_1,\ldots, t_m$ such that $x = fa^nb^m\, t_1 \ldots\, t_m$.

(vi) $x \in L_{FOL}$ if and only if a) there exists a relational expression Ra^nb^m and m terms $t_1,\ldots, t_m$ such that $x = Ra^nb^m\, t_1 \ldots\, t_m$ or b) there exist $u, v \in L_{FOL}$ and an individual variable w such that $x = \neg u$ or $x = \wedge uv$ or $x = \forall wu$.[3]

Definition 2.

Let $G_{L_{FOL}}$ be the context-free grammar with X from Definition 1 as the terminal alphabet, $Y = \{V, C, T, A, B, Z, S\}$ as the non-terminal alphabet, and the following set of productions:

$V \rightarrow vA$, $C \rightarrow cA$;

$T \rightarrow V$; $T \rightarrow C$; $T \rightarrow fAZ$; $A \rightarrow a$; $A \rightarrow aA$; $Z \rightarrow bT$; $Z \rightarrow bZT$;

$S \rightarrow RAZ$; $S \rightarrow \neg S$; $S \rightarrow \wedge SS$; $S \rightarrow \forall VS$.[4]

[3] This definition uses Levin's method of indexing relational expressions given in note 5 to Chapter 1 of his book. There are a number of differences between what Levin takes to be L_{FOL} and the language introduced by Definition 1, regarding the use of parentheses, functional expressions, and the existence of free variables in the formulas of L_{FOL}. These differences are not essential in that the conclusions of this paper hold if the language is modified to include parentheses and exclude functional symbols. The status of the sub-language $L_{S\text{-}FOL}$ of first-order sentences (formulas of L_{FOL} with no free variables) is characterized by Proposition 1 below.

[4] This definition is based on the grammar given by Levin on p. 56 for the set of atomic formulas of his version of L_{FOL}.

Clearly $G_{L_{FOL}}$ generates L_{FOL} as is already implicit in Levin's discussion.

Levin conjectured that the sub-language $L_{S\text{-}FOL}$ consisting in the sentences (closed formulas) of L_{FOL} is *not* context-free[5]. That this is indeed the case can be shown by an application of

Ogden's Lemma.[6] Suppose L is a context-free language. Then there exists an integer k such that if $u \in L$, $|\, u \,| \geq k$, and any k or more positions of u are designated as ‚distinguished', then there are strings x_1, x_2, x_3, x_4 and x_5 satisfying
(i) $u = x_1x_2x_3x_4x_5$;
(ii) x_2x_4 contains at least one distinguished position;
(iii) $x_2x_3x_4$ contains no more than k distinguished positions;
(iv) x_3 contains at least one distinguished position;
(v) for every $l \geq 0$, $x_1x_2^1x_3x_4^1x_5 \in L$.

Proposition 1. $L_{S\text{-}FOL}$ is not context-free.

Proof. Suppose that $L_{S\text{-}FOL}$ is context-free and let k be the constant with the properties given by Ogden's Lemma. Let $u = \forall va^nRabbva^nva^n$ be an element of $L_{S\text{-}FOL}$ such that $n > k$ and suppose that we designate all occurrences of a in the string a^n following the second occurrence of v in u (i.e. the symbols in the index of the second occurrence of the bound variable va^n) as distinguished. It is obvious that, no matter how we choose x_1, x_2, x_3, x_4 and x_5, it is impossible that $x_1x_2^2x_3x_4^2x_5$ is a sentence of L_{FOL}. The point is that at least one of x_2, x_4 must be a proper sub-string of the index of the second variable occurrence in u; doubling it will inevitably alter the index of this variable occurrence. This effect can be partially offset if both x_2, x_4 are not null and doubling them changes the indices of the first two variable occurrences in the same way. But then the third variable occurrence in $x_1x_2^2x_3x_4^2x_5$ will no longer be bound by the quantifier and hence $x_1x_2^2x_3x_4^2x_5 \notin L_{S\text{-}FOL}$, a contradiction.

Levin asks two further questions about other versions of L_{FOL} (see Levin 1982, p. 56). The first is whether its context-free character is affected by changing the order of the indices of the various expressions, i.e. by modifying the clauses *(v-vi)* of Definition 1 thus:

(v') x is a term if and only if a) x is an individual variable or b) x is an individual constant or c) there exists a functional expression fa^nb^m and n terms $t_1,\ldots, t_n$ such that $x = fa^nb^m\ t_1\ldots\ t_n$;

[5] What is a closed formula of L_{FOL}? In order to answer this question it is sufficient to define the notion of quantifier scope for the formulas of L_{FOL}. If the quantifier expression $\forall w$, where w is a variable, occurs in a formula u of L_{FOL} and it is not followed by an occurrence of a, then the unique sub-expression v of u immediately following $\forall w$ in u, such that $v \in L_{FOL}$, is the scope of the occurrence of $\forall w$ in question. The notions of bound occurrence of a variable, bound variable and closed formula (sentence) of L_{FOL} are introduced in the usual way. The latter are the elements of the language $L_{S\text{-}FOL}$.

[6] As stated in Martin 1997, p. 244. Proposition 1 is proved independently by several authors (see van Benthem 1988, p. 123).

(vi') $x \in L'_{FOL}$ if and only if a) there exists a relational expression $Ra^n b^m$ and n terms $t_1,\ldots, t_n$ such that $x = Ra^n b^m\, t_1 \ldots\, t_n$ or b) there exist $u, v \in L'_{FOL}$ and an individual variable w such that $x = \neg u$ or $x = \wedge uv$ or $x = \forall wu$.

In other words, in the new version of the language it is the first index of a functional or relational expression that expresses its arity.

The answer to this first question is that the language L'_{FOL} is still context-free. It is generated by a grammar with the same alphabet as $G_{L_{FOL}}$ and the following set of productions:

$V \rightarrow vA$, $C \rightarrow cA$; $A \rightarrow a$; $A \rightarrow aA$;
$T \rightarrow V$; $T \rightarrow C$; $T \rightarrow faZT$; $Z \rightarrow aZT$; $Z \rightarrow B$; $B \rightarrow b$; $B \rightarrow bB$;
$S \rightarrow RaZT$; $S \rightarrow \neg S$; $S \rightarrow \wedge SS$; $S \rightarrow \forall VS$.

The second question is whether using positional numerals for the indices of expressions changes the context-free character of L_{FOL}. The answer is *yes, it does*. I will use binary numerals just for the arity indices of the functional and relational expressions. The individuating indices of expressions will still be non-null strings from {*a*}*. The indices identifying arity will be non-null strings from {0,1}* such that their first symbol is 1, i.e. non-zero binary numerals. The number expressed by $n = s_k \ldots s_1$ is $\tau(n) = \sum_{i=1}^{k} 2^{i-1} \cdot s_i$.

Definition 3.

(i-ii) just as in Definition 1.

(iii) x is a functional expression if and only if there exist $n > 0$ and a binary numeral m such that $x = fa^n m$.

(iv) x is a relational expression if and only if there exist $n > 0$ and a binary numeral m such that $x = Ra^n m$.

(v) x is a term if and only if a) x is an individual variable or b) x is an individual constant or c) there exist a functional expression $fa^n m$ and $\tau(m)$ terms $t_1,\ldots, t_{\tau(m)}$ such that $x = fa^n m\, t_1 \ldots\, t_{\tau(m)}$.

(vi) $x \in L''_{FOL}$ if and only if a) there exists a relational expression $Ra^n m$ and $\tau(m)$ terms $t_1,\ldots, t_{\tau(m)}$ such that $x = Ra^n b^m\, t_1 \ldots\, t_{\tau(m)}$ or b) there exist $u, v \in L''_{FOL}$ and an individual variable w such that $x = \neg u$ or $x = \wedge uv$ or $x = \forall wu$.

Proposition 2. L''_{FOL} is not context-free.

Proof. Suppose that L''_{FOL} is context-free and let k be the integer with the properties given by Ogden's Lemma relative to L''_{FOL}. Suppose that $u = Ra^n mva \ldots va$ is a string in L''_{FOL} with more than k occurrences of va (i.e. $\tau(m) > k$), that m is a string of 1's, and that all and only the occurrences of v in this string are designated as distinguished. Again it is impossible to find strings x_1, x_2, x_3, x_4 and x_5 satisfying Ogden's Lemma with respect to u. Clearly one of x_2, x_4 must be a proper sub-string of the sequence of $\tau(m)$ occurrences of va in u. Doubling it will increase the number of occurrences of va in $x_1 x_2^2 x_3 x_4^2 x_5$.

It follows that x_2 must be a non-null sub-string of m in order to compensate for this increase in the number of terms following the relational expression. However, doubling any sub-string of m creates a numeral m' such that $\tau(m') > 2\tau(m)$. Consequently in $x_1x_2^2x_3x_4^2x_5$ there cannot be agreement between the arity index of the relational expression and the new number of variable occurrences (which is strictly smaller than $2\tau(m)$ given that x_4 cannot contain all the occurrences of v in u). Hence $x_1x_2^2x_3x_4^2x_5 \notin L''_{FOL}$, a contradiction.

One can also prove that L''_{FOL} is not even the intersection of a finite number of context free languages by using the results in Gorun 1980. Proposition 4 in that paper states that every bounded language that is the intersection of a finite number of context-free languages is *slip*, i.e. its image under the Parikh function is a semi-linear set. Let us introduce these concepts:

Definition 4.

(i) A language L is bounded if there exist words $w_1,\ldots, w_k$, $k \geq 1$, such that $L \subseteq w_1^* \ldots w_2^*$

(ii) A set $A \subseteq \mathbb{N}^k$ is linear if there are vectors $c, p_1,\ldots, p_r$ in $\mathbb{N}^k$ such that $A = \{c + \sum_{i=1}^{r} n_i p_i : n_i \in \mathbb{N}, 1 \leq i \leq r\}$. A subset of $\mathbb{N}^k$ is semi-linear if it is a finite union of linear sets.

(iii) Let $X = \{a_1,\ldots, a_r\}$ be an alphabet. The Parikh function $\mathcal{P}: X^* \rightarrow \mathbb{N}^r$ is defined recursively thus: $\mathcal{P}(a_i)$ is the vector with 0 in all positions apart form the i-th, where it has the component 1; if $u, v \in X^*$, then $\mathcal{P}(uv) = \mathcal{P}(u) + \mathcal{P}(v)$.

Proposition 3. L''_{FOL} is not the intersection of a finite number of context free languages.

Proof. Assume the contrary. Then $L''_{FOL} \cap \{Ra10^m(va)^n : m, n \geq 0\}$ is also the intersection of finitely many context-free languages and a bounded language as well. Its elements are those expressions of $\{Ra10^m(va)^n : m, n \geq 0\}$ with the property that $n = 2^m$. If we give its alphabet the ordering $\langle 0, a, v, R, 1\rangle$, we have that its image under the Parikh function will be the set $\{(m, 2^m + 1, 2^m, 1, 1) : m \geq 0\}$. Clearly this set is semi-linear only if the set $\{(m, 2^m) : m \geq 0\}$ is (which is not): contradiction.

On the positive side it can be shown that L''_{FOL} is in deterministic logarithmic space (*D-LogSpace*), i.e. it can be decided by a deterministic Turing machine with a read-only input tape and a single work-tape, using at most $\lfloor \log_2(x) \rfloor$ squares (cells) of the work-tape for any input x[7].

[7] The two-tape Turing machine is described briefly on p. 45 of Nepomnyaschii 1972 and at greater length in chapter 2 of Szepietowski 1994 (pp. 7-14). The main motivation for using a work tape distinct from the input tape is to study computations that use less space (i.e. number of cells scanned on the work tape) than it is necessary to write the input, such as those in *LogSpace*.

The proof of this fact is based on three observations. First, if in Definition 3 we allow functional expressions, relational expressions, sentential operators and quantifiers to apply to any finite number of the resulting pseudo-terms or pseudo-formulas, respectively, then the resulting language (which will be called pseudo-L''_{FOL}) is recognizable by a so-called deterministic pebble automaton and therefore by a Turing machine operating within *D-LogSpace* (here we rely on Theorem 3.2.2 of Szepietowski 1994, p. 16). Secondly, checking if a pseudo-formula is obtained from correct applications of the sentential operators and quantifiers to possibly incorrect atomic pseudo-formulas can be done by a simple adaptation of a folklore method of verifying the correctness of propositional formulas in Polish notation described in Exercise 1.18 on p. 21 of Mendelson 1997. Thirdly, checking if a pseudo-formula also has correct atomic sub-formulas (and thus is a member of L''_{FOL}) can be done by counting the occurrences of certain symbols in x (namely the occurrences of v not preceded by a quantifier, the occurrences of c and those of f), recording the total obtained in binary notation, subtracting functional arity indices from the total and comparing the total with the first relational arity index encountered in x (after a total is compared with the index of a relational expression then it can be discarded and the counting resumes). All these procedures can be implemented by a deterministic Turing machine operating in *LogSpace*. Let us detail these observations.

Definition 5. (i-iv) just as in Definition 3.

(v) x is a pseudo-term if and only if a) x is an individual variable or b) x is an individual constant or c) there exist a functional expression fa^nm and pseudo-terms $t_1,\ldots, t_k$ such that $x = fa^nm\ t_1\ldots\ t_k$.

(vi) $x \in$ pseudo-L''_{FOL} if and only if a) there exist a relational expression Ra^nm and pseudo-terms $t_1,\ldots, t_k$ such that $x = Ra^nb^m\ t_1\ldots\ t_k$ or b) there exist $u_1,\ldots, u_k \in$ pseudo-L''_{FOL} and an individual variable w such that $x = \neg u_1\ldots u_k$ or $x = \wedge u_1\ldots u_k$ or $x = \forall w u_1\ldots u_k$.

Lemma 1. The language pseudo-L''_{FOL} can be decided by a deterministic pebble automaton and therefore by a deterministic Turing machine operating in *LogSpace*.

Proof. First we should note that the complex pseudo-terms of L''_{FOL} are words with the structure $\varphi_1\psi_1\ \ldots\ \varphi_n\psi_n$ (with $n \geq 1$) where each φ_i is a sequence of functional expressions and every ψ_i is a sequence of primitive terms (variables or constants). A sequence of pseudo-terms is either a complex pseudo-term or a complex pseudo-term prefixed by a sequence of primitive terms. The atomic pseudo-formulas are words with the structure $\rho\sigma$, where ρ is a relational expression and σ is a sequence of pseudo-terms, whereas the complex pseudo-formulas are words with the structure $\alpha_1\beta_1\ldots\alpha_n\beta_n$ (with $n \geq 1$) where each α_i is a sequence of sentential operators and quantified

variables (such a sequence will be called a 'prefix' from now on) and each β_i is a sequence of primitive pseudo-formulas.

Also note that the set of primitive terms, the set of functional expressions, the set of relational expressions and the set of prefixes are regular languages. For example, the set of prefixes is generated by the grammar over the terminal alphabet $\{\neg, \wedge, \forall, v, a\}$ with the following productions:

$$S\to\neg S,\ S\to\neg,\ S\to\wedge S,\ S\to\wedge,\ S\to\forall V,\ V\to vI,\ I\to aI,\ I\to a,\ I\to aS.$$

Regular languages are recognized by very simple machines, the finite automata (see §3.3 of Martin 1997, pp. 79-83). Hence, the regular languages indicated above are recognizable by automata which we will denote by $\mathcal{A}_T$, $\mathcal{A}_F$, $\mathcal{A}_R$ and $\mathcal{A}_P$ respectively.

A pebble automaton is a strengthened version of a finite automaton: it is a machine with a two-way input tape and a reading head that can also place and remove pebbles (markers) numbered from 1 to k on the cells of the tape (these machines are described in §3.2 of Szepietowski 1994, p. 16). Each move of the machine is determined by the current internal state of the machine and by the contents of the currently scanned tape cell (which apart from an input symbol may also include one or more pebbles). A move of the machine will consist in moving the reading head, changing the internal state and (possibly but not necessarily) placing or removing pebbles from the currently scanned cell.

Let us introduce now a series of pebble automata and sketch their operation. The machines that come later in the series rely on the earlier ones and have an increasing number of pebbles.

$\mathcal{A}_1$ is a 2-pebble automaton that recognizes sequences of primitive terms. If an input word does not begin with c or v, then $\mathcal{A}_1$ rejects it. Otherwise it places 1 on the first symbol. If 1 is placed, then $\mathcal{A}_1$ searches the next occurrence of c or v to the right of 1's position and places 2 on the cell left of it. If no such occurrence exists, 2 is placed at the end of the input. After 2 is placed, $\mathcal{A}_1$ returns to 1 and behaves like $\mathcal{A}_T$ on the word between 1 and 2. If $\mathcal{A}_T$ accepts the word between 1 and 2, then 1 is re-placed on the cell to the right of 2. If no such cell exists, then $\mathcal{A}_1$ accepts the input.

Similarly one can define a 2-pebble automaton $\mathcal{A}_2$ that recognizes sequences of functional expressions (relying on $\mathcal{A}_F$).

$\mathcal{A}_3$ is an 8-pebble automaton that recognizes pseudo-terms. If an input word does not begin with f, v or c, then $\mathcal{A}_3$ rejects it. If the input begins with v or c, then $\mathcal{A}_3$ behaves like $\mathcal{A}_T$. If the input begins with f, then 1 is placed on it. If 1 is placed, then $\mathcal{A}_3$ seeks the next occurrence of v or c after it and places 2 on the cell to its left. If no such occurrence exists, then the input is rejected (a pseudo-term must end with a primitive term). After 2 is placed, $\mathcal{A}_3$ returns to 1 and behaves like $\mathcal{A}_2$ on the word between 1 and 2. If the $\mathcal{A}_2$ sub-system accepts the word between 1 and 2 (otherwise $\mathcal{A}_3$ rejects the input), then the

sub-system pebbles (i.e. 3 and 4) are collected and 5 is placed to the right of 2. If 5 is placed, then $\mathcal{A}_3$ seeks the next occurrence of *f* after it and places 6 to the left of that occurrence. If no such occurrence exists, 6 is placed at the end of the input word. If 6 is placed, then $\mathcal{A}_3$ returns to 5 and behaves like $\mathcal{A}_1$ on the word between 5 and 6. If the $\mathcal{A}_1$ sub-system accepts the word between 5 and 6 (otherwise $\mathcal{A}_3$ rejects the input), then its pebbles (i.e. 7 and 8) and pebbles 1 and 2 are collected, and 1 is placed to the right of 6. If that is impossible, then $\mathcal{A}_3$ stops and accepts the input.

One can then define an automaton $\mathcal{A}_4$ that recognizes sequences of pseudo-terms (recall that such a sequence is either a complex pseudo-term or a complex pseudo-term prefixed by a sequence of primitive terms), an automaton $\mathcal{A}_5$ that recognizes atomic pseudo-formulas (using $\mathcal{A}_R$ and $\mathcal{A}_4$), an automaton $\mathcal{A}_6$ that recognizes sequences of atomic pseudo-formulas (using $\mathcal{A}_5$) and finally an automaton $\mathcal{A}_7$ that recognizes pseudo-formulas (using $\mathcal{A}_P$ and $\mathcal{A}_6$). The ideas involved are pretty much the same: pairs of pebbles are used to mark out segments of the input that are recognizable by previously defined automata. In the case of $\mathcal{A}_5$, for example, the input must have the form $\rho\sigma$, where ρ is a relational expression and σ is a sequence of pseudo-terms with the boundary between the two being the earliest occurrence of *f*, *v* or *c* in the input (hence $\mathcal{A}_5$ needs just an extra pebble compared with $\mathcal{A}_4$).

At this point it is possible to invoke Theorem 3.2.2 of Szepietowski 1994, which asserts that deterministic pebble automata can be simulated by deterministic *LogSpace* Turing machines, and conclude that pseudo-L''_{FOL} is indeed in *D-LogSpace*.

Suppose now that we strengthen Definition 5 by adopting clause *(vi.b)* from Definition 3, i.e. we impose that negation and quantification are unary sentential operators and conjunction is a binary sentential operator. Call the resulting language quasi-L''_{FOL}. We have a ready-made *D-LogSpace* procedure for determining whether a word in pseudo-L''_{FOL} satisfies this extra condition, which we might call *molecular integrity*. Adapted to our system of notation, Exercise 1.18 on p. 21 of Mendelson 1997 introduces the Polish variant of the language of propositional logic with alphabet $\{\neg, \wedge, \vee, \supset, \equiv, P, a\}$ and the following formation rules:

(i) Pa^n is a well-formed formula (wff) for every $n \geq 1$. Each such expression is a statement letter.
(ii) If α and β are wffs, then so are $\neg\alpha$, $\wedge\alpha\beta$, $\vee\alpha\beta$, $\supset\alpha\beta$ and $\equiv\alpha\beta$.
(iii) Only expressions obtained by a finite number of applications of clauses (i-ii) are wffs.

Mendelson then offers the following criterion for determining whether an expression α over this alphabet is a wff: count each occurrence of a binary operator as +1, each occurrence of a statement letter as -1 and each occurrence of negation as 0. Then α is a wff if and only if a) the sum of the symbols

in α is -1 and b) the sum of the symbols in any proper initial segment of α is non-negative.

This criterion can be immediately adapted to a word x in pseudo-L''_{FOL}: simply count (from left to right) every occurrence of R as -1, every occurrence of $\neg$ and $\forall$ as 0 (i.e. simply ignore them!) and every occurrence of $\wedge$ as +1. If the total reached for x is -1 and the total for every proper initial segment of x up to the last occurrence of R is non-negative, then x has the property of molecular integrity, i.e. x is in fact an element of quasi-L''_{FOL}. In other words, x is constructed by proper applications of sentential operators and quantification to what might be defective atomic pseudo-formulas, i.e. atomic formulas in which relation symbols or functional symbols are applied to the wrong number of arguments. Clearly Mendelson's criterion for molecular integrity can be implemented by a deterministic Turing machine working in *LogSpace*. After simulating $\mathcal{A}_7$ the machine uses the working tape to record binary numerals for numbers between 0 and the number of occurrences of $\wedge$ in the input (sign can be taken care of by the internal states of the machine). The input is rejected only if the count does not end at -1 or -1 is reached more than once. We can thus state

Lemma 2. The language quasi-L''_{FOL} can be decided by a deterministic Turing machine operating in *LogSpace*.

What about the last property that qualifies a word x for membership in L''_{FOL}, i.e. the fact that functional expressions and relational expressions in x apply to the correct number of arguments (a property we might call *atomic integrity*)? In order to describe the procedure for determining atomic integrity I will introduce a parameter (called 'valence') for sequences of pseudo-terms which gives the number of *active* arguments in such a sequence. Valence is calculated as follows: starting from the right of the sequence, add 1 for every occurrence of v, c or f; when encountering a functional expression, subtract its arity index from the current total. For example, the sequence $faa10\hat{}vaa\hat{}ca\hat{}fa1\hat{}vaaa$ has valence 2: there are 5 occurrences of v, c and f in it, but the combined arities of the two functional symbols are 3. It can be proved by induction that the numerical value in the calculation of the valence for a sequence σ of pseudo-terms never becomes negative if and only if the sequence has a unique decomposition in proper terms (the induction proceeds on the number of occurrences of functional expressions in σ).

Definition 6. Let $\mathcal{S}$ be the set of sequences of pseudo-terms. The valence function $\mathcal{V} : \mathcal{S} \to \mathbb{Z}$ is defined thus:
(i) $\mathcal{V}(\varepsilon) = 0$;
(ii) $\mathcal{V}(va^n\hat{}\sigma) = \mathcal{V}(ca^n\hat{}\sigma) = \mathcal{V}(\sigma) + 1$ for every $\sigma \in \mathcal{S}$ and $n > 0$;
(iii) $\mathcal{V}(fa^nm\hat{}\sigma) = (\mathcal{V}(\sigma) + 1) - \tau(m)$ for every $n > 0$ and binary numeral $m > 0$.

Lemma 3. Let σ be a sequence of pseudo-terms. Then there exists a unique sequence of terms $\lambda = t_1 \ldots t_k$ with $k = \mathcal{V}(\sigma)$ such that $\sigma = \lambda$ if and only if $\mathcal{V}(\sigma) > 0$ and $\mathcal{V}(\mu) > 0$ for every $\mu \in \mathcal{S}$ that is an final segment of σ (i.e. there exists a sequence η of primitive terms and functional symbols such that η^μ = σ).

Proof. For one direction of this equivalence it can be shown by induction on term complexity that $\mathcal{V}(t) = 1$ for every term t and that $\mathcal{V}(\eta) > 0$ for every final segment η in $\mathcal{S}$ of a term t. Suppose there exists a unique sequence of terms $\lambda = t_1 \ldots t_k$ with $k = \mathcal{V}(\sigma)$ such that $\sigma = \lambda$ and that $\mu \in \mathcal{S}$ is a final segment of σ. Then either μ is a final segment of t_k or $\mu = \eta\hat{}t_{j+1}\ldots t_k$ with η a final segment of t_j and $j < k$. In either case $\mathcal{V}(u) > 0$.

The converse can be obtained by induction on the number of function symbols in σ. If σ is just a sequence of primitive terms, then there is nothing to prove:σ admits a unique decomposition in proper terms (and clearly the number of those terms equals $\mathcal{V}(\sigma)$). Assume now that σ has at least one functional expression occurrence. If so, then there is a leftmost such occurrence, i.e. $\sigma = \gamma\hat{}\varphi\hat{}\theta$, where γ is a possibly empty sequence of primitive terms $\gamma = u_1\ldots u_k$ $(k \geq 0)$, φ is a functional expression and $\theta \in \mathcal{S}$. If the valence of every final segment in $\mathcal{S}$ of σ is positive, then the same is true of every final segment of θ and $\mathcal{V}(\theta) > 0$ as well. Since θ has fewer functional expression occurrences than σ, it follows by the induction hypothesis that θ admits a unique decomposition in proper terms $\chi = t_1 \ldots t_m = \theta$ with $\mathcal{V}(\theta) = m$. But φ^χ is also a final segment of σ and hence $\mathcal{V}(\varphi\hat{}\chi) > 0$. It follows that the arity of φ, say n, is smaller than m and therefore φ^χ admits the unique decomposition $\varphi\hat{}\chi = \varphi(t_1,\ldots,t_n)\, t_{n+1}\ldots t_m$, whereas σ admits the unique proper term decomposition $\lambda = u_0\ldots u_k\, \varphi(t_1,\ldots,t_n)\, t_{n+1}\ldots t_m$ with $(k + m + 1) - n$ members. Furthermore, $\mathcal{V}(\varphi\hat{}\chi) = (m - n) + 1$, and hence $\mathcal{V}(\sigma) = \mathcal{V}(\gamma) + \mathcal{V}(\varphi\hat{}\chi) = (k + m + 1) - n$.

Lemma 3 allows us to devise a procedure for determining the atomic integrity of a quasi-formula α which consists in calculating the valence of sequences of pseudo-terms and comparing it to the arity index of the relational expressions encountered moving from right to left in α. If valence ever becomes negative (i.e. we reach a term that requires more arguments than those that are active), then α does not have atomic integrity. When a relational expression is reached, if valence agrees with its arity index, then valence is reset to zero and the calculation restarts. If agreement does not obtain, then α lacks atomic integrity.

This procedure can also be implemented by a deterministic Turing machine operating in *LogSpace*, since the only significant operations it involves are:

a) counting from right to left symbol occurrences (namely those of *v* not preceded by a quantifier, of *c*, and of *f*) in α, with the total reached being inscribed (and modified) as a binary numeral on the work tape;

b) subtracting binary numerals on the input tape (arity indices of functional terms) from the binary numeral on the working tape;

c) comparing the binary numeral on the working tape with binary numerals on the input tape (arity indices of relational expressions).

Only the second operation requires some comment. If the reading head of the machine reaches a binary numeral on the input tape, then it moves left until it detects whether it is a relational or a functional index. In the second case, both machine heads return to the rightmost digit of the binary numerals on the input tape and the work tape (call them the *i*-numeral and the *w*-numeral and their digits the *i*-digits and the *w*-digits respectively). Subtracting the *i*-numeral from the *w*-numeral can be done with the usual 'borrowing' method. While performing the subtraction the machine operates in two modes, *normal* and *indebted*, and the computation starts in normal mode.

In normal mode the machine takes the following actions:

(i) If the scanned *i*-digit is 0, then both heads move left (the scanned *w*-digit is left unchanged) and the machine stays in normal mode.
(ii) If the scanned *i*-digit is 1 and the scanned *w*-digit is 1, then the *w*-digit is changed to 0, both heads move left and the machine stays in normal mode.
(iii) If the scanned *i*-digit is 1 and the scanned *w*-digit is 0, then the *w*-digit is changed to 1, both heads move left and the machine switches to indebted mode.
(iv) If the end of the *w*-numeral is reached before the end of the *i*-numeral, then the input is rejected (the quasi-formula examined lacks atomic integrity). If the end of the *i*-numeral is reached before or simultaneously with the end of the *w*-numeral, then the machine resumes counting occurrences of *v* (if not preceded by a quantifier), *c*, *f*, starting from the *w*-numeral (possibly after erasing all its leading 0-s).

In indebted mode the machine takes the following actions:

(i) If the scanned *w*-digit is 1, then it is changed to 0 and the machine switches back to normal mode.
(ii) If the scanned *w*-digit is 0 and the scanned *i*-digit is 1, then both heads move left (the scanned *w*-digit is left unchanged) and the machine stays in indebted mode.
(iii) If the *w*-digit is 0 and the scanned *i*-digit is 0, then the *w*-digit is changed to 1, both heads move left and the machine stays in indebted mode.
(iv) If the end of the *i*-numeral is reached in indebted mode prior to the end of the *w*-numeral, then the working head moves left changing all consecutive 0-s to 1-s until it reaches the first 1, which it changes to 0 (unless it is the leftmost 1, in which case it is simply erased). If the end of the *w*-numeral is reached in indebted mode (as it would happen when subtracting 11 from 10), then the input is rejected.

Using this method binary subtraction can be performed without any extra space being used on the working tape other than the one required for counting in binary the occurrences of the symbols v (not preceded by $\forall$), c and f in the input. Hence the verification of atomic integrity can also be done deterministically in *LogSpace*. We can therefore state

Proposition 4. L''_{FOL} is in *D-LogSpace*.

All the languages examined here are rudimentary in Smullyan's sense[8]. This can be shown indirectly by using Proposition 4 and the result in Nepomnyaschii 1972 that *D-LogSpace* is included in the class of rudimentary languages.

Acknowledgments - I wish to thank two anonymous referees for comments and criticisms that have helped improve this paper, especially with respect to the proof of Proposition 4.

REFERENCES

van Benthem, J. 1988. Logical syntax. *Theoretical Linguistics*, 14: 119-142.

Ganea, Mihai. 2010. Two (or three) notions of finitism. *The Review of Symbolic Logic* 3 (1): 119-144.

Gorun, I. 1980. On intersections of context-free languages. *Fundamenta Informaticae* 3 (4): 491-496.

Jones, N. D. 1969. Context-free languages and rudimentary attributes. *Mathematical Systems Theory* 3 (2): 102-109.

Levin, G. 1982. *Categorical Grammar and the Logical Form of Quantification*. Bibliopolis.

Martin, J. C. 1997. *Introduction to Languages and the Theory of Computation* 1st ed. McGraw-Hill.

Mendelson, E. 1997. *Introduction to Mathematical Logic* 4th ed. Chapman and Hall.

Nepomnyaschii, V.A. 1972. Rudimentary interpretation of two-tape Turing computation. *Cybernetics* 6: 43-53.

Quine, W. V. 1986. *Philosophy of Logic* 2nd ed. Harvard University Press.

[8] For a definition of rudimentary languages see for instance Jones 1969, p. 104.

Szepietowski, A. 1994. *Turing Machines with Sublogarithmic Space*. Lecture Notes in Computer Science 843. Springer.

Wagner, S. J. 1987. The rationalist conception of logic. *Notre Dame Journal of Symbolic Logic* 28 (1): 3-35.