Journal of Computer and System Sciences 77 (2011) 393–421

Contents lists available at ScienceDirect

Journal of Computer and System Sciences

www.elsevier.com/locate/jcss

On the complexity of regular-grammars with integer attributes

M. Manna[a], F. Scarcello[b], N. Leone[a,*]

[a] Department of Mathematics, University of Calabria, 87036 Rende (CS), Italy
[b] Department of Electronics, Computer Science and Systems, University of Calabria, 87036 Rende (CS), Italy

ARTICLE INFO

Article history:
Received 21 August 2009
Received in revised form 17 May 2010
Available online 27 May 2010

Keywords:
Attribute grammars
Computational complexity
Models of computation

ABSTRACT

Regular grammars with attributes overcome some limitations of classical regular grammars, sensibly enhancing their expressiveness. However, the addition of attributes increases the complexity of this formalism leading to intractability in the general case. In this paper, we consider regular grammars with attributes ranging over integers, providing an in-depth complexity analysis. We identify relevant fragments of tractable attribute grammars, where complexity and expressiveness are well balanced. In particular, we study the complexity of the classical problem of deciding whether a string belongs to the language generated by any attribute grammar from a given class $\mathcal{C}$ (call it $\text{PARSE}_{[\mathcal{C}]}$). We consider deterministic and ambiguous regular grammars, attributes specified by arithmetic expressions over $\{|\ |, +, -, \div, \%, *\}$, and a possible restriction on the attributes composition (that we call *strict* composition). Deterministic regular grammars with attributes computed by arithmetic expressions over $\{|\ |, +, -, \div, \%\}$ are **P**-complete. If the way to compose expressions is strict, they can be parsed in **L**, and they remain tractable even if multiplication is allowed. Problem $\text{PARSE}_{[\mathcal{C}]}$ becomes **NP**-complete for general regular grammars over $\{|\ |, +, -, \div, \%, *\}$ with strict composition and for grammars over $\{|\ |, +, -, \div, \%\}$ with unrestricted attribute composition. Finally, we show that even in the most general case the problem is in polynomial space.

© 2010 Elsevier Inc. All rights reserved.

1. Introduction

1.1. Motivation

The problem of *identifying*, *extracting* and storing information from unstructured documents is widely recognized as a main issue in the field of *information and knowledge management* and has been extensively studied in the literature (see, for instance, [13,16,33,31,17]). Most existing approaches use regular expressions as a convenient mean for *extracting information from text automatically*. Regular grammars, indeed, offer a simple and declarative way to specify patterns to be extracted, and are suitable for efficient evaluation (recognizing whether a string belongs to a regular language is feasible in linear-time). However, regular grammars have a *limited expressiveness*, which is not sufficient for powerful information extraction tasks. There are simple extraction patterns, like, for instance, $a^n b^n$, that are relevant to information extraction but cannot be expressed by a regular grammar. To express such patterns, regular grammars can be enhanced by attributes, storing information at each application of a production rule (a single attribute acting as a counter is sufficient to express $a^n b^n$ through a regular grammar).

In fact, our interest in attribute grammars is strongly motivated by their high potential for information extraction. We have employed them in HıLεX, an advanced system for ontology-based information extraction [49,48] used in real-world

* Corresponding author.
E-mail addresses: manna@mat.unical.com (M. Manna), scarcello@deis.unical.it (F. Scarcello), leone@mat.unical.com (N. Leone).

0022-0000/$ – see front matter © 2010 Elsevier Inc. All rights reserved.
doi:10.1016/j.jcss.2010.05.006

applications. Unfortunately, the complexity of grammars with attributes is sensibly harder than in the attribute-free case; for instance, the addition of string attributes to regular grammars leads to **Exptime**-hardness even in the simple case of deterministic grammars using only two attributes, if (string) concatenation of attributes is allowed [12]. Thus, a careful complexity analysis, leading to the identification of tractable cases of attribute grammars where complexity and expressiveness are well balanced, is called for, and will be carried out in this paper.

It is worthwhile noting that, even if our interest in attribute grammars came from their usage in Information Extraction, attribute grammars are employed in many other domains, ranging from databases to logic programming (see Section 1.5 below). Thus, a precise characterization of their complexity can be profitably used in a wide range of applications.

1.2. The framework: Integer attribute grammars

The attribute grammar formalism is a declarative language introduced by Knuth as a mechanism for specifying the semantics of context-free languages [27]. An attribute grammar $\mathcal{AG}$ is an extension of a context-free grammar $\mathcal{G}$ with a number of attributes which are updated at any application of a production rule (by some suitable functions). Moreover, the applicability of a production rule is conditioned by the truth of some predicates over attribute values. The language generated by an attribute grammar $\mathcal{AG}$ consists of all strings that have a legal parse tree in $\mathcal{G}$ where all attribute values are related in the prescribed way (that is, they satisfy the predicates). Thus, a parse tree of the original context-free grammar may not be a legal parse tree of the attribute grammar, and the language accepted by an attribute grammar is in general a subset of the corresponding context-free language.

In this paper, we consider regular grammars with attributes ranging over integers, also called *integer attribute grammars* or IRGs, for short. An IRG is a quadruple $\mathcal{AG} = \langle \mathcal{G}, Attr, Func, Pred \rangle$ where

- $\mathcal{G}$ is a regular grammar;
- *Attr* is a set of integer attributes associated with the nonterminals of $\mathcal{G}$;
- *Func* is a set of *functions* associated with the production rules of $\mathcal{G}$, assigning values to attributes by means of arithmetic expressions over $\{|\ |, +, -, \div, \%, *\}$[1];
- *Pred* is a set of *predicates*, associated with the production rules of $\mathcal{G}$, checking values of attributes. A predicate is a Boolean combination of comparison predicates of the form $E_1 \odot E_2$, where $\odot$ is a standard comparison operator in $\{<, \leqslant, =, >, \geqslant, \neq\}$, and E_1 and E_2 are arithmetic expressions over $\{|\ |, +, -, \div, \%, *\}$.

A *valid parse tree* for $\mathcal{AG}$ is a parse tree of $\mathcal{G}$ labeled by attributes (computed according to the functions of the corresponding productions of $\mathcal{G}$) such that all predicates evaluate to true. The language generated by $\mathcal{AG}$, denoted by $\mathcal{L}(\mathcal{AG})$, is the set of all strings derived by some valid parse tree for $\mathcal{AG}$.

There are many real world tasks easily described by means of regular grammars with (integer) attributes. Let us briefly introduce syntax and semantics of IRGs through two simple examples.

The first example below shows an IRG, that we call $\mathcal{AG}_{XML}$, for "discovering" XML fragments having a correct nesting of at least k levels. To simplify the presentation we suppose that the input XML string is valid (no further condition is to be checked on the string). Moreover, since we are not interested in tag matching and the like, we assume that a terminal '\$' delimits the XML document, and a terminal 'a' represents any character other than '\$', '<', '/', '>'. It is important to notice that, unlike typical approaches based on the Document Object Model[2] that materialize a tree of polynomial size for performing this task, $\mathcal{AG}_{XML}$-parsing can be performed (deterministically) in logarithmic space, as we will see later.

The (left) regular attribute grammar shown in Fig. 1 allows to verify whether the input XML document has a nesting level of at least **k** or not. This can be done by using only two attributes: x and n.

p_0 : TEXT $\to$ '\$'	$x := 0,$	$n := 0$
p_1 : TEXT $\to$ TEXT 'a'	$x := x,$	$n := n$
p_2 : TAG $\to$ TEXT '<'	$x := x,$	$n := n + 1$
p_3 : END_TAG $\to$ TAG '/'	$x := x,$	$n := n - 2$
p_4 : TAG_NAME $\to$ TAG 'a'	$x := x,$	$n := n$
p_5 : TAG_NAME $\to$ END_TAG 'a'	$x := x,$	$n := n$
p_6 : TAG_NAME $\to$ TAG_NAME 'a'	$x := x,$	$n := n$
p_7 : TEXT $\to$ TAG_NAME '>'	$x := (n = \mathbf{k})\ ?\ 1 : x,$	$n := n$
p_8 : XML $\to$ TEXT '\$'	$(x = 1) \wedge (n = 0)$	

Fig. 1. The $\mathcal{AG}_{XML}$ grammar recognizing XML fragments of nesting $\geqslant k$.

[1] The symbols in the set $\{|\ |, +, -, \div, \%, *\}$ denote the operators *absolute value*, *addition*, *subtraction*, *integer division* with truncation, *modulo reduction* (or remainder from integer division), and *multiplication*, respectively. Note that we only consider the integer division, and thus any arithmetic expression over these operators returns an integer value.

[2] The *Document Object Model* (DOM) is an application programming interface (API) for HTML and XML documents. It defines the logical structure of documents and the way a document is accessed and manipulated [55].

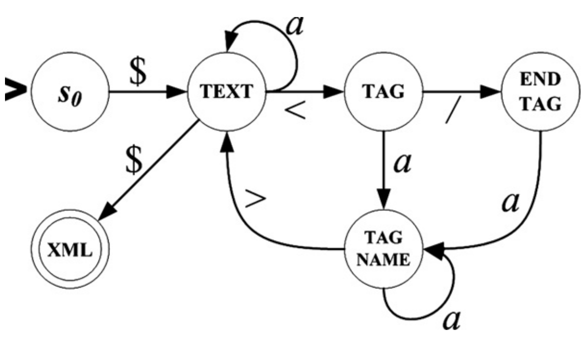


Fig. 2. A DFA for the regular grammar underlying $\mathcal{AG}_{XML}$.

The functions defining the evaluations of attributes x and n are specified by suitable assignment instructions. For instance, production rule p_3 carries out the same value for x and decreases n by 2.

Note that predicate $n = k$ is used in the function determining the x value for p_7; while p_8 applies only if predicate $x = 1 \wedge n = 0$ is true (p_8 is the only production rule which is conditioned by the truth of a predicate; all other productions are unconditionally applied as for standard grammars, but they set up the attribute values).

The attribute n counts the number of open-tags while the attribute x is set to 1 (production p_7) if the value of n reaches $\mathbf{k}$ ($x := \underline{(n = \mathbf{k})\ \mathbf{?}\ 1\ \mathbf{:}\ x}$). When the whole document has been parsed, only predicate $(x = 1) \wedge (n = 0)$ (production p_8) needs to be evaluated. If the value of n is 0, the tag-nesting is correct and we have to check only whether the nesting level is above the desired threshold. This is done by looking at the value of x: if it is 1, then the XML document has a nesting level of at least $\mathbf{k}$, otherwise, the input string has a nesting level strictly lower than $\mathbf{k}$. Fig. 2 shows the deterministic finite-state automaton associated with the regular grammar underlying $\mathcal{AG}_{XML}$.

The second example involves possible extensions of languages for expressing XML schemas (e.g., DTD or XML-Schema). We consider a widespread abstraction of XML documents and DTDs focusing on document structure [45]. An XML document is a *finite ordered, labeled, unranked tree t*. For any node x of t, its label $\mathsf{label}(x)$ belongs to the alphabet Σ of the *element names*. Moreover, $\mathsf{labcdrn}(x)$ denotes the string of the labels of the children of x in the left-to-right order. A DTD over Σ is a triple $d = \langle \Sigma, \pi, s\rangle$ where π is a function that maps each symbol in Σ to a regular language over Σ, and $s \in \Sigma$ is the *start symbol*. The language $\mathcal{L}(d)$ of d is a set of trees such that for each tree $t \in \mathcal{L}(d)$, $\mathsf{root}(t) = s$, and for every node x with $\mathsf{label}(x) = a$, $\mathsf{labcdrn}(x) \in \pi(a)$. An example of DTD is $d_1 = \langle \{s_1, a, b\}, \pi_1, s_1\rangle$ with $\pi_1(s_1) = \{a^n b^m\colon\ n, m > 0\}$ and $\pi_1(a) = \pi_1(b) = \{\varepsilon\}$. An XML document satisfying d_1 might be:

```
<s1>
  <a> first a </a>
  <a> second a </a>
  <b> unique b </b>
</s1>
```

Now, assume we would like to have a different kind of legal trees, e.g., trees where $\pi_1(s_1) = \{a^n b^n\colon\ n > 0\}$. Then, we could not express such a schema according to the above classical definition of DTD, because $\pi_1(s_1)$ is not a regular language. However, it turns out that this schema can be easily expressed if we use integer attribute grammars, instead of standard regular grammars. Indeed, Example 4 in Section 3 shows a simple kind of IRG that allows us to express any language of the form $\{a_1^n \ldots a_j^n\colon\ n > 0,\ j > 1\}$. It is thus quite natural to think of a more expressive notion of DTD, where π maps symbols in Σ to IRG-languages over Σ. As this paper shows, for many interesting classes of IRGs (like the above one) we get a higher expressive power than regular grammars without loosing efficiency. In particular, validating such XML documents would still be feasible in polynomial time, or even less.

1.3. Main problems studied

We face the classical problem of deciding whether a string belongs to the language generated by an integer attribute grammar $\mathcal{AG}$. To characterize precisely its complexity, and to identify possible islands of tractability, we consider languages generated by different classes of grammars and, for any such a class $\mathcal{C}$, we study its corresponding class of parsing problems, denoted by $\textsc{parse}_{[\mathcal{C}]}$. In particular, we consider deterministic and ambiguous regular grammars, and we also distinguish IRGs according to the set of operators (from $\{|\ |, +, -, \div, \%, *\}$) allowed to occur in the arithmetic expressions, as well as according to possible restrictions on attributes compositions. In this respect, we consider both grammars with general productions over the considered set of functions, called *free* grammars, and the so-called *strict* grammars where, in any production p, each p-attribute "contributes" at most once to the computation of new attribute values.[3] For example, $\mathcal{AG}_{XML}$ is a strict grammar. Note that the latter syntactic condition reduces the possible growth of attribute values.

1.4. Overview of results

The analysis carried out in this paper provides many results on the complexity of $\textsc{parse}_{[\mathcal{C}]}$. Most of them are positive results, in that we are able to identify tractable classes of IRGs with rather natural restrictions, which keep a good expressive power without loosing in efficiency. We also provide hardness results that are useful to identify the source of complexity for more general IRGs. In summary, we show that

- $\textsc{parse}_{[\mathcal{C}]}$ is tractable for the following classes of integer attribute grammars:
  - (i) Deterministic regular strict grammars with (attributes specified by) arithmetic expressions over $\{|\ |, +, -, \div, \%\}$;
  - (ii) General (possibly ambiguous) regular strict grammars with arithmetic expressions over $\{|\ |, +, -, \div, \%\}$;
  - (iii) Deterministic regular grammars with arithmetic expressions over $\{|\ |, +, -, \div, \%\}$, without the strict-restriction;
  - (iv) Deterministic regular strict grammars, without any restriction on the arithmetic operators ($*$ may occur in arithmetic expressions).

  In particular, $\textsc{parse}_{[\mathcal{C}]}$ is
  - **L**-complete in case (i);
  - **NL**-complete in case (ii);
  - **P**-complete in cases (iii) and (iv).
- $\textsc{parse}_{[\mathcal{C}]}$ is **NP**-complete for general regular grammars over operators $\{|\ |, +, -, \div, \%\}$ (without restrictions on attribute composition);
- $\textsc{parse}_{[\mathcal{C}]}$ is **NP**-complete for general regular grammars over the set $\{|\ |, +, -, \div, \%, *\}$, under strict composition;
- $\textsc{parse}_{[\mathcal{C}]}$ is still in **PSPACE** in the most general case of any regular grammar with any arithmetic expression over $\{|\ |, +, -, \div, \%, *\}$. For this case, we miss a completeness result, and thus whether or not some IRG may yield a **PSPACE**-hard language remains an open question.

It is worthwhile noting that all **P**-completeness and **L**-completeness results we derived for deterministic regular grammars hold also for both left-regular and right-regular deterministic grammars.

1.5. Related work

As far as related work is concerned, attribute grammars, defined by Knuth for specifying and implementing the (static) semantic aspects of programming languages [27], have been a subject of intensive research, both from a conceptual and from a practical point of view. The first full implementation for parsing attribute grammars was produced by Fang [14], under the supervision of Knuth. Lewis et al. [35] defined two subclasses of attribute grammars, named *S-attributed grammars* and *L-attributed grammars*, for one-pass compilation. Kennedy and Warren [26] presented a method of constructing, for a given attribute grammar, a recursive procedure which performs the specified semantic evaluation. A wide class of attribute grammars (e.g., *absolutely noncircular attribute grammars* and *ordered attribute grammars*) can be handled, and the resulting evaluators are efficient. Kastens et al. [25], Farrow [15], and Koskimies et al. [30] developed systems (called GAG, LINGUIST-86, and HLP84, respectively) with high-level input languages, resulting in powerful language processing tools.

The versatility of attribute grammars is confirmed by the large number of application areas where these grammars are employed, such as *logic programming* [10,11], *databases* [47], *natural language interfaces* [2], and *pattern recognition* [57]. More recently, it has been shown they are useful also for XML, and in particular for *query languages* and *query processing* (see, for instance, [39,41,40,29]). Extensive reviews of attribute grammar theory, implementation, systems, and applications are given in, e.g., [9,8,28,1,43].

Although complexity issues about regular grammars (without attributes) have been thoroughly studied [23], expressiveness and complexity of regular grammars *with* attributes have not been analyzed in depth. A number of results on the expressiveness of attribute grammars has been derived by Efremidis et al. [12], where the authors define two interesting classes of string attribute grammars $\mathcal{A}^p$ and $\mathcal{A}^p_s$, which are shown to be the same as the complexity class **EXP** and "roughly the same" as the complexity class **NP**, respectively.

To the best of our knowledge, the present paper is the first work studying the complexity of regular grammars with integer-domain attributes.

[3] More precisely, we define an IRG to be *strict* if all of its productions are *strict*, where a production p is *strict* if the labeled dependency graph built from the functions in $Func(p)$ is deterministic. That is, every attribute has at most one outgoing arc, or two mutually exclusive outgoing arcs (i.e., arcs whose labels are the logical complements of each other). See Section 4.1 for the formal definition.

1.6. Structure of the paper

The sequel of the paper is organized as follows. Section 2 collects some preliminaries about formal grammars and languages. Section 3 defines our notion of integer attribute grammar with bounded resources, and gives its semantics and an example. Section 4 gives a detailed overview of the complexity results for parsing various classes of attribute grammars. Section 5 shows how to build a Turing machine for computing integer attribute grammars, which is useful for proving complexity upper-bounds. The comprehensive computational complexity analysis is carried out in Section 6. Finally, Section 7 draws our conclusions. Some basic notions on languages and complexity classes which are used in the paper are recalled in Appendix A.

2. Preliminaries on Chomsky languages

Let W be an *alphabet* (finite and nonempty set of symbols). A *string* w (also, *word*) over W is a finite sequence of symbols from W. The number of symbols in w is its *length* and it is denoted by $\|w\|$, while $w[j]$ denotes the jth symbol of w ($0 \leqslant j < \|w\|$). Let ε be the *empty string* ($\|\varepsilon\| = 0$) and W^+ denote all strings of finite length over W, then $\langle W^*, \circ, \varepsilon\rangle$ is a free monoid where $W^* = W^+ \cup \{\varepsilon\}$ and $\circ$ is the binary relation of *concatenation* on W^*. A set L of strings from W^* is a *language* over W, with Λ denoting the empty language. Let L_1, L_2 be two languages. Then, language L_1L_2 contains all strings of the form w_1w_2, with $w_1 \in L_1$ and $w_2 \in L_2$.

2.1. Context-free grammars and languages

A *context-free grammar* (CFG) $\mathcal{G} = \langle \Sigma, N, S, \Pi\rangle$ consists of an alphabet of *terminals* Σ, an alphabet of *nonterminals* N, a *start symbol* $S \in N$, and a finite set of *productions* $\Pi \subseteq N \times (\Sigma \cup N)^*$. A production $p \in \Pi$ is said to be an ε-production if $p \in (N \times \{\varepsilon\})$. Usually, a pair $\langle A, \alpha\rangle \in \Pi$ is written as $A \to \alpha$. Let $\Rightarrow^+_{\mathcal{G}}$ be the binary relation of *derivation* [23] with respect to $\mathcal{G}$.

The language of grammar $\mathcal{G}$ is the set

$$\mathcal{L}(\mathcal{G}) = \{x \in \Sigma^*\colon\ S \Rightarrow^+_{\mathcal{G}} x\}.$$

A language L is a *context-free language* (CFL) if there exists a CFG $\mathcal{G}$ such that $L = \mathcal{L}(\mathcal{G})$.

A context-free grammar $\mathcal{G}$ is *cycle-free* (or non-cyclic) if there is no derivation of the form $A \Rightarrow^+_{\mathcal{G}} A$ for some nonterminal A.

Given a context-free grammar $\mathcal{G} = \langle \Sigma, N, S, \Pi\rangle$, a *parse tree* t of $\mathcal{G}$ is a tree such that: (i) the *root node* $\rho(t)$ is labeled by the start symbol S; (ii) each *leaf node* has label in $\Sigma \cup \{\varepsilon\}$; (iii) each *internal node* is labeled by a nonterminal symbol; (iv) if $A \in N$ is a non-leaf node in t with children $\alpha_1, \ldots, \alpha_h$, listed from left to right, then $A \to \alpha_1 \ldots \alpha_h$ is a production in Π. By concatenating the leaves of t from left to right we obtain the derived string $x(t)$ of terminals, which is called the *yield* of the parse tree. The language generated by $\mathcal{G}$ can be also defined as:

$$\mathcal{L}(\mathcal{G}) = \{x(t)\colon\ t \text{ is a parse tree of } \mathcal{G}\}.$$

The *size* of a parse tree [38] is the number of its non-leaf nodes, while the *height* measures the longest path from the root to some leaf. A tree consisting of a single node (the root) has *height* $= 0$. If $\mathcal{G}$ is cycle-free and t is a parse tree of $\mathcal{G}$ such that $\|x(t)\| = n$, then both $size(t) \leqslant |N| * (2n - 1)$ and $height(t) \leqslant |N| * n$ hold.

A context-free grammar $\mathcal{G}$ is unambiguous if it does not have two different parse trees for any string. A context-free language is unambiguous if it can be generated by an unambiguous context-free grammar. Context-free grammars and languages are ambiguous if they are not unambiguous.

Example 1. Let $L_1 = \{a^nb^nc^n\colon\ n > 0\}$ be a language on the alphabet $\Sigma = \{a, b, c\}$. It is well known that L_1 is not a context-free language because there exists no context-free grammar for it.

Example 2. Let $L_2 = \{a^nb^n\colon\ n > 0\}$ be a language over $\Sigma = \{a, b\}$. For L_2 there exists a (unambiguous) context-free grammar $\mathcal{G}$ such that $\mathcal{L}(\mathcal{G}) = L_2$. The productions of $\mathcal{G}$ are:

$$p_0 : S \to aA,$$

$$p_1 : A \to Sb,$$

$$p_2 : A \to b.$$

2.2. Regular grammars and languages

A *regular grammar* is a context-free grammar $\mathcal{G} = \langle \Sigma, N, S, \Pi \rangle$ where either $\Pi \subseteq N \times (\Sigma \cup (N \circ \Sigma) \cup \{\varepsilon\})$ or $\Pi \subseteq N \times (\Sigma \cup (\Sigma \circ N) \cup \{\varepsilon\})$ holds. Grammar $\mathcal{G}$ is said to be *left-regular* or *right-regular*, respectively. By construction, any regular grammar $\mathcal{G}$ is non-cyclic, so the size of any parse tree t of $\mathcal{G}$ is either $\|x(t)\| + 1$ or $\|x(t)\|$ (depending on whether ε appears in t or not).

Regular grammars can also be ambiguous. Consider the grammar $\mathcal{G}_a$ having the following productions:

$$p_0 : S \to aS,$$
$$p_1 : S \to aA,$$
$$p_2 : S \to b,$$
$$p_3 : A \to aS,$$
$$p_4 : A \to aA,$$
$$p_5 : A \to b.$$

It is easy to see that $\mathcal{L}(\mathcal{G}_a) = \{a^n b\colon\ n \geqslant 0\}$, but every string w of length n can be derived by 2^n different parse trees. However, it is well known that every regular language is unambiguous [54]. That is, for every regular language L, there exists an unambiguous regular grammar $\mathcal{G}$ such that $L = \mathcal{L}(\mathcal{G})$.

Example 3. Let L_3 be the above mentioned language $\{a^n b\colon\ n \geqslant 0\}$ over $\Sigma = \{a, b\}$. The following unambiguous regular grammar generates L_3:

$$p_0 : S \to aS,$$
$$p_1 : S \to b.$$

2.3. Regular grammars and finite automata

2.3.1. NFA automata

A *non-deterministic finite automaton* (*NFA*) is a quintuple [34] $M = \langle K, \Sigma, \Delta, s_0, F \rangle$ where

- K is a finite set of *states*;
- Σ is an alphabet of *terminals* disjoint from K;
- $s_0 \in K$ is the *initial state*;
- $F \subseteq K$ is the set of *final states*;
- Δ, the *transition relation*, is a subset of $K \times (\Sigma \cup \{\varepsilon\}) \times K$. Each triple $(q, u, q') \in \Delta$ is called a *transition* of M.

If M is in state $q \in K$ and the symbol under the input-cursor is $a \in \Sigma$, then M may move to state $q' \in K$ only if $(q, u, q') \in \Delta$ and u is either a or ε. Sometimes the notation $q' \in \Delta(q, u)$ is more convenient. Notice that if $u = \varepsilon$, so the input-cursor does not change its position when M moves to q'.

A *configuration* of M is an element of $K \times \Sigma^*$. Let (q, w) be a configuration, then w is the unread part of the input. Consider now the binary relation $\mapsto_M$ between two configurations. Relation $(q, w) \mapsto_M (q', w')$ holds if and only if there exists $u \in \Sigma \cup \{\varepsilon\}$ such that $w = uw'$ and $q' \in \Delta(q, u)$.

Let $\mapsto_M^*$ be the reflexive, transitive closure of $\mapsto_M$. The language generated by M is defined as:

$$\mathcal{L}(M) = \bigl\{ w \in \Sigma^* : (s_0, w) \mapsto_M^* (q, \varepsilon),\ q \in F \bigr\}.$$

For any regular grammar $\mathcal{G} = \langle \Sigma, N, S, \Pi \rangle$ there exists a non-deterministic finite automaton $M(\mathcal{G}) = \langle K, \Sigma, \Delta, s_0, F \rangle$, *naturally isomorphic* to $\mathcal{G}$, such that $\mathcal{L}(\mathcal{G}) = \mathcal{L}(M)$. If $\mathcal{G}$ is left-regular, the automaton $M(\mathcal{G})$ is defined as follows:

- $K = N \cup \{s_0\}$;
- $B \in \Delta(A, a)$ if and only if $B \to Aa$ belongs to Π;
- $B \in \Delta(s_0, \alpha)$ if and only if $\alpha \in \Sigma \cup \{\varepsilon\}$ and $B \to \alpha$ belongs to Π;
- $F = \{S\}$.

If $\mathcal{G}$ is right-regular, the automaton $M(\mathcal{G})$ is defined as follows:

- $K = N \cup F$, with $F = \{s_f\}$;
- $B \in \Delta(A, a)$ if and only if $A \to aB$ belongs to Π;
- $s_f \in \Delta(A, \alpha)$ if and only if $\alpha \in \Sigma \cup \{\varepsilon\}$ and $A \to \alpha$ belongs to Π;
- $s_0 = S$.

2.3.2. DFA automata

A *deterministic finite automaton* (*DFA*) is a finite-machine $M = \langle K, \Sigma, \delta, s_0, F\rangle$ behaving like an *NFA* where the *transition function* $\delta : K \times \Sigma \to K$ is a restriction of Δ. A configuration of M and the language $\mathcal{L}(M)$ are defined exactly as in the *NFA* case.

Given a regular grammar $\mathcal{G} = \langle \Sigma, N, S, \Pi\rangle$, if $M(\mathcal{G})$ is deterministic, also $\mathcal{G}$ is said to be *deterministic*. In this case, $\mathcal{G}$ is *unambiguous* as well. For instance, the grammar shown in Example 3 is deterministic.

Deterministic regular grammars are a proper subset of unambiguous regular grammars [42]. For completeness, we next recall two classical results about regular languages [34]:

Proposition 2.1. *For each non-deterministic finite automaton, there is an equivalent deterministic finite automaton.*

Proposition 2.2. *A language is said to be regular if and only if it is accepted by a finite automaton.*

3. Integer attribute regular grammars

An integer-attribute regular grammar (short: IRG) $\mathcal{AG}$ consists of an *attribute system* associated with the underlying regular grammar $\mathcal{G}$. Each nonterminal A of $\mathcal{G}$ has a set $Attr(A)$ of symbols named (*synthesized*) *attributes* of A.[4]

Given a set $Oper \subseteq \{|\ |, +, -, \div, \%, *\}$ of arithmetic operators, an *integer-attribute regular grammar* over *Oper* is, formally, a quadruple

$$\mathcal{AG} = \langle \mathcal{G}, Attr, Func, Pred\rangle$$

where

- $\mathcal{G} = \langle \Sigma, N, S, \Pi\rangle$ is a *regular grammar*, called the underlying grammar of the attribute regular grammar $\mathcal{AG}$.
- *Attr* maps every nonterminal symbol to its (*synthesized*) *attributes*, that is, for each nonterminal symbol $A \in N$, the set $Attr(A)$ contains the attributes of A. Each attribute is defined on the *integer* domain.
- *Func* is a set of *semantic rules* for assigning values to attributes through *arithmetic expressions* on the operators specified in the set *Oper*. For each production $p \in \Pi$, say $\alpha_0 \to \alpha_1\alpha_2$ (where, either exactly one of α_1, α_2 is a nonterminal symbol, or at least one of α_1, α_2 is ε), we denote by $Func(p) \subseteq Func$ the set of rules of p computing attribute values. Any rule in $Func(p)$ maps values of certain attributes (possibly none) of the nonterminal (if any) α_i ($i \in \{1, 2\}$) into the value of some attribute of α_0, say a. Such a rule has the form $a := E(a_1, \ldots, a_k)$, where E is an arithmetic expression on the set of attributes $a_1, \ldots, a_k$, denoted by $ExpAttr(E)$, and a_j is an attribute of α_i for any $j \in \{1, \ldots, k\}$. If p does not contain any nonterminal symbol, then E is simply a constant value. We allow also rules with *arithmetic-if* expressions of the form $a := \underline{(Cond)\ \mathbf{?}\ E_T\ \mathbf{:}\ E_F}$, where (i) E_T and E_F are arithmetic expressions over *Oper*, and (ii) *Cond* is a Boolean combination of comparison predicates. A *comparison predicate* is of the form $E_1 \odot E_2$, with $\odot \in \{<, \leqslant, =, >, \geqslant, \neq\}$, and with E_1 and E_2 being arithmetic expressions over *Oper*. The value of an arithmetic-if statement is either E_T, if *Cond* is true, or E_F, otherwise.
- *Pred* is a set of *semantic rules* for checking that, in any production, values of attributes satisfy certain desired conditions. For each production $p \in \Pi$, we denote by $Pred(p) \in Pred$ its predicate, that is, a Boolean combination of comparison predicates (as specified above) involving attributes occurring in p.

A *valid parse tree* for $\mathcal{AG}$ is a parse tree of $\mathcal{G}$ labeled by the grammar attributes such that all labels of its (internal) nodes satisfy the following conditions: all attribute values are computed according to the rules of the corresponding productions of $\mathcal{G}$, and all the predicates evaluate to true. The language generated by $\mathcal{AG}$, denoted by $\mathcal{L}(\mathcal{AG})$, is the set of all strings derived by some valid parse tree for $\mathcal{AG}$.

Example 4. We describe an integer attribute grammar for deriving the strings of language $L_1 = \{a^n b^n c^n\colon n > 0\}$ on the alphabet $\Sigma = \{a, b, c\}$, a well-known *context-sensitive* language. Let $\mathcal{AG}_1 = \{\mathcal{G}_1, Attr, Func, Pred\}$ be an integer attribute grammar, where $\mathcal{G}_1 = \langle \Sigma', N, S, \Pi\rangle$ is the underlying grammar on $\Sigma' = \{a, b, c, \$\}$, $N = \{A, B, C, S\}$, S is the starting symbol, the set of attributes are $Attr(S) = \emptyset$, $Attr(A) = \{h\}$, $Attr(B) = Attr(C) = \{h, n\}$, and the productions (including attributes expressions and predicates) are shown in Fig. 3. Note that, to improve readability, we use sometimes a subscript with the attribute, to specify the nonterminal symbol it refers to (e.g., n_C is the attribute n associated with C).

For instance, consider the string *aabbcc*\$. A valid parse-tree for $\mathcal{AG}_1$, yielding this string, is shown in Fig. 4.

Note the typical fish-spine shape of this tree. Recall that all parse trees of (integer attribute) regular grammars have this form.

[4] Classically, in attribute grammars the set of attributes is partitioned in two subsets, the *synthesized* and the *inherited* attributes. For the sake of simplicity, in our framework we do not consider *inherited* attributes. A study of the more general case where both kinds of attributes may occur will be subject for future work.

$p_6 : A \to a$	$h := 1$		
$p_5 : A \to Aa$	$h := h + 1$		
$p_4 : B \to Ab$	$h_B := 1,$	$n_B := h_A$	
$p_3 : B \to Bb$	$h := h + 1,$	$n := n$	
$p_2 : C \to Bc$	$h_C := 1,$	$n_C := h_B,$	$n_B = h_B$
$p_1 : C \to Cc$	$h := h + 1,$	$n := n$	
$p_0 : S \to C\$$			$n_C = h_C$

Fig. 3. The $\mathcal{AG}_1$ grammar.

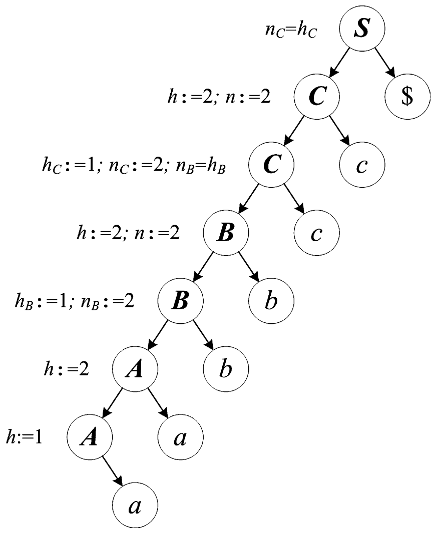


Fig. 4. Parse tree for string *aabbcc*\$ obtained from grammar $\mathcal{AG}_1$.

It is easy to see that the valid parse trees of $\mathcal{AG}_1$ yield precisely the language L_1. Indeed, we separately count the number of a's, b's and c's respectively in the productions (p_5, p_6), (p_3, p_4) and (p_1, p_2) thanks to the attribute h. A rule in p_4 assigns the number of a's, stored in h_A, to the attribute n of B. Then, this value will be compared with the number of b's stored in h_B through rule p_2, and assigned to n_C. Eventually, this value will be compared with the number of c's stored in h_C, through the predicate in rule p_0. Therefore, only parse trees such that the numbers of a, b, and c are the same will be valid. Moreover, by construction of the regular grammar $\mathcal{G}_1$ of $\mathcal{AG}_1$, these parse trees only yield strings where we have first a sequence of a, then a sequence of b, and finally a sequence of c. It follows that, for any string w on the alphabet $\Sigma = \{a, b, c\}$, $w \in L_1$ if and only if $w\$ \in \mathcal{L}(\mathcal{AG}_1)$. In the following sections, we shall prove that such a language can be parsed in deterministic logspace.

Observe that the above grammar can be easily generalized to yield any language of the form $\{a_1^n \dots a_j^n\colon\ n > 0,\ j > 3\}$, for which we need the same kind of attributes (of the form h, n), and exactly $2j + 1$ productions. All of the new productions have the form of p_1 and p_2. Again, parsing these languages is feasible in **L**.

Remark. It is worthwhile noting that the previously shown languages L_1, L_2, and L_3 belong to some very expressive language classes (see Appendix A.4). In particular, $L_1 \in$ **CSL**, $L_2 \in$ **CFL**, and $L_3 \in$ **REG**. Despite of the generality of their classes (recall that **CSL** = **NSPACE**(n)), these languages are all expressible by the simplest kinds of our attribute grammars (σ-*DLReg*$^A_\otimes$ and σ-*DRReg*$^A_\otimes$) defined in the next section. As shown in Section 6, both these classes of attribute grammars are very efficiently computable (parsing is feasible in logarithmic space). This fact is a further confirmation of the usefulness of attributes grammars: they allow us to deal very efficiently (in logspace) with languages requiring computationally more expensive formalisms to be expressed by standard (attribute-free) grammars.

4. Overview of the complexity results

In this section, we give an overview of our analysis of the complexity of IRGs. As we shall show, when attributes are used, classical results on the equivalence of the different kinds of regular grammars are no longer valid. Therefore, we study different classes of regular grammars with integer attributes. More precisely, for any considered class $\mathcal{C}$ of IRGs, we study its corresponding class of parsing problems PARSE$_{[\mathcal{C}]}$ for the languages defined by grammars in $\mathcal{C}$. Therefore, when we speak of the computational complexity of PARSE$_{[\mathcal{C}]}$, in fact we refer to the complexity of parsing any of the hardest languages over all languages definable by some grammar in $\mathcal{C}$.

4.1. Restricted classes of IRGs

We evaluate the complexity of $\textsc{parse}_{[\mathcal{C}]}$ by varying the possible classes $\mathcal{C}$ according to three parameters concerning the structure of the underlying regular grammars, the operators that are allowed in the arithmetic expressions computing the attribute values (i.e., the operators occurring in *Pred* or in *Func*), and the way how attributes can be composed. In particular, we consider the following restrictions for any integer attribute grammar $\mathcal{AG}$ belonging to the considered class $\mathcal{C}$:

- *Grammar.* We distinguish among three kinds of grammars underlying $\mathcal{AG}$, depending on whether they are *deterministic left-regular* (*DLReg*), *deterministic right-regular* (*DRReg*) or simply *regular* (*Reg*). Accordingly, $DLReg^A$, $DRReg^A$, and Reg^A denote, respectively, the class of attribute grammars where the underlying grammar is deterministic left-regular, deterministic right-regular, or regular.
- *Operators.* We consider arithmetic expressions either over the full set of operators $\{|\ |, +, -, \div, \%, *\}$ or over $\{|\ |, +, -, \div, \%\}$. In the latter case, where the multiplication operator is not allowed, the restricted classes of attribute grammars are denoted by $DLReg^A_\otimes$, $DRReg^A_\otimes$, and $Reg^A_\otimes$. Of course, when operator $*$ is allowed to occur in the expressions, the size of an attribute may grow up much more faster than in the $*$-free case. For instance, Example 5 shows an IRG with arithmetic expressions employing the $*$-operator, where the value of some attributes equals $2^{(2^{n-1})}$.
- *Composition.* We say that a production p of $\mathcal{AG}$ is *strict* if each attribute in the right-hand side of p gives a single contribution to the computation of values in all the arithmetic expressions occurring in $Func(p)$. In particular, observe that an attribute b appearing both in the then and in the else branches of an arithmetic-if statement, gives one contribution. The same holds if b occurs in such different branches belonging to different arithmetic-if statements, as long as they have the same if-condition. If an attribute occurs only in the condition of arithmetic-if statements, then its contribution is zero. All other occurrences in the right-hand side of arithmetic expressions count for one contribution. More formally, consider the labeled (*directed*) *dependency graph* $G_p = \langle Nodes_p, Arcs_p \rangle$ built, from p, as follows:
  1. $Nodes_p = Attr$;
  2. $(a', \top, a) \in Arcs_p$ if $a := E$ is in $Func(p)$ and $a' \in AttrExp(E)$;
  3. $(a', Cond, a) \in Arcs_p$ if $a := \underline{(Cond)\ \mathbf{?}\ E_T\ \mathbf{:}\ E_F}$ is in $Func(p)$ and $a' \in AttrExp(E_T)$;
  4. $(a', \neg Cond, a) \in Arcs_p$ if $a := \overline{(Cond)\ \mathbf{?}\ E_T\ \mathbf{:}\ E_F}$ is in $Func(p)$ and $a' \in AttrExp(E_F)$.

  We say that p is *strict* if G_p is *deterministic*, that is, if each node has at most one outgoing arc, or exactly two mutually exclusive outcoming arcs (i.e., arcs whose labels are the logical complements of each other). The grammar $\mathcal{AG}$ is *strict* if all of its productions are strict. The fact that all grammars of a class have this restriction is denoted by the prefix σ-, e.g., σ-Reg^A denotes all IRGs (possibly non-deterministic) with the strict-composition restriction, but with no restriction on the arithmetic operators.

For instance, observe that the grammar in Example 4 is a strict deterministic left-regular grammar with $*$-free arithmetic expressions, and thus it belongs to σ-$DLReg^A_\otimes$.

Note that the three parameters above correspond with orthogonal sources of complexity for IRGs: the first one directly emerges from the use of non-determinism, the second one follows from the different nature of multiplicative and additive operators, and the last one concerns the possible ways of combining attribute values in production rules.[5]

4.2. The complexity of IRGs

Table 1 summarizes the complexity results we derived for $\textsc{parse}_{[\mathcal{C}]}$ under all combinations of the restrictions specified in Section 4.1, contrasted with the corresponding known results on regular grammars without attributes (column 0; see Appendix A.4 for more details). Each row refers to a class of attribute grammars ($DLReg^A$, $DRReg^A$, and Reg^A) where the related regular grammars are deterministic left-regular, deterministic right-regular and (just) regular, respectively.

Each column refers to a pair of restrictions *Operators*/*Composition*. For instance, cell (B,3) refers to strict deterministic right-regular grammars with attributes specified by expressions over $\{|\ |, +, -, \div, \%, *\}$.

The results in Table 1 show many interesting tractable cases. In particular, tractability is guaranteed whenever the grammar $\mathcal{AG}$ satisfies (at least) one of the following conditions: (i) the grammar is deterministic and attributes are specified by

Table 1
Complexity results.

		[0] *Reg*	[1] σ-$Reg^A_\otimes$	[2] $Reg^A_\otimes$	[3] σ-Reg^A	[4] Reg^A
[A]	*DLReg*	$\mathbf{NC}^1$-complete	**L**-complete	**P**-complete	**P**-complete	**PSPACE**
[B]	*DRReg*	$\mathbf{NC}^1$-complete	**L**-complete	**P**-complete	**P**-complete	**PSPACE**
[C]	*Reg*	$\mathbf{NC}^1$-complete	**NL**-complete	**NP**-complete	**NP**-complete	**PSPACE**

[5] Note that the notion of strict grammar is inspired by "strongly polynomial attribute grammar" in [12]. However, the former is a syntactic restriction, while the latter is a semantic one.

any arithmetic expression over $\{|\ |, +, -, \div, \%\}$ (id est, operator $*$ is disallowed)—all cells in the square (A1,B2) in Table 1; (ii) the grammar is strict (possibly ambiguous) and operator $*$ is disallowed—cell (C,1); (iii) the grammar is strict and deterministic (all operators in $\{|\ |, +, -, \div, \%, *\}$ are allowed)—cells (A,3) and (B,3). Importantly, note that the complexity remains in **NL** if the grammar is strict—cell (C,1), and it decreases even to **L** if, in addition, it is deterministic—cell (A,1).

Unlike string attribute grammars, we are able to show that, even for the most general class $\mathcal{C}$ of integer attribute grammars (any regular grammar without any restriction on the arithmetic operators in $\{|\ |, +, -, \div, \%, *\}$ or the attribute composition), $\text{PARSE}_{[\mathcal{C}]}$ remains in **PSPACE**—cell (C,4). Moreover, the complexity goes down to **NP**, if the $*$ operator is forbidden in the attribute expressions, or the grammar is strict (cells (C,3) and (C,2)).

As a final remark, from our proofs it can be easily shown that the complexity of the classes of grammars in the first two lines of Table 1 does not change even if the regular grammar underlying $\mathcal{AG}$ is only required to be unambiguous, and thus not necessarily deterministic.

5. Recognizing IRGs

As shown in Section 2.3, there exists a natural isomorphism mapping a regular grammar $\mathcal{G}$ to a finite state automaton $M(\mathcal{G})$. In this section, we show that there exists a similar kind of natural general algorithm for solving any integer attribute grammar. In particular, we define a mapping from any IRG $\mathcal{AG}$ to a Turing machine that recognizes the language $\mathcal{L}(\mathcal{AG})$. Moreover, in the subsequent section we show that this *general* machine is almost optimal. That is, for any considered class of IRGs, but for the general unrestricted case, the running time of such Turing machines matches the computational complexity of the class of languages defined by those grammars.

5.1. IRGs and Turing machines

Formally, for any integer attribute grammar $\mathcal{AG} = \langle \mathcal{G}, Attr, Func, Pred\rangle$ where $\mathcal{G} = \langle \Sigma, N, S, \Pi\rangle$, we define a multi-tape Turing machine, say *TM*($\mathcal{AG}$), which decides $\mathcal{L}(\mathcal{AG})$. Machine *TM*($\mathcal{AG}$) is endowed with an input tape holding the input string, say w, of length n and with two work-tapes, say $curr_x$ and $prev_x$, for each attribute x in *Attr*. We denote by $curr_x(j)$ the value of x depending on the first j input-symbols of *inp* already parsed ($1 \leqslant j \leqslant n$). Equivalently, $prev_x(j)$ is the value of x depending on the first $j-1$ input-symbols of w, and obviously, $prev_x(j) = curr_x(j-1)$ holds for any $j \geqslant 1$. By default, $curr_x(0) = prev_x(0) = 0$. The Turing machine has also two extra work-tapes: *rul*, which is used for computing functions or evaluating predicates, and *pos*, which is used for saving useful (encoding of) positions of the input-cursor. Each of these auxiliary work-tapes (but *rul*) is used for storing integer values only, and thus we shall often use the name of the tape, say wt, to denote the integer value encoded by this tape.

The transition relation (or function, for deterministic machines) of *TM*($\mathcal{AG}$) is determined by the finite automaton for parsing the regular grammar $\mathcal{G}$, suitably enriched with routines for processing integer attributes and evaluating logical predicates.

5.1.1. DLReg^A attribute grammars

For these grammars, the parsing of the basic grammar is based on the automaton $M(\mathcal{G}) = \langle K, \Sigma, \delta, s_0, F\rangle$, as defined in Section 2.3. In particular, in the machine $TM(\mathcal{AG}) = \langle K', \Sigma, \delta', s_0\rangle$, $K \subset K'$ and δ' is a (completely specified) function extending the (partial) function δ of $M(\mathcal{G})$. Note that one transition from A to B in $M(\mathcal{G})$ corresponds to a number of transitions in *TM*($\mathcal{AG}$). These transitions, without moving the input-cursor, compute attribute values and update the attributes accordingly, then evaluate the predicate, and finally succeed if it is satisfied. More precisely, suppose *TM*($\mathcal{AG}$) is in state $A \in K$, $a \in \Sigma$ is the current symbol of w, and p is the (unique) production related to (A, a), then *TM*($\mathcal{AG}$) computes the functions in *Func*(p) (updating the corresponding attributes), and leads to state $\delta(A, a) = B$ if the valuation of the predicate *Pred*(p) is true. As a special case, if $A = s_0$, then production p has the form $B \to a$; otherwise, p has the form $B \to Aa$. Moreover, if Π includes a production like $B \to \varepsilon$, say p, then *TM*($\mathcal{AG}$) must compute *Func*(p) and evaluate *Pred*(p) before the first symbol of w has been read.

If either $\delta(A, a)$ is not defined or the valuation of *Pred*(p) fails, then *TM*($\mathcal{AG}$) rejects w. Otherwise, if the input is completely read and a final state in F is reached, then w is accepted.

For instance, Fig. 5 depicts the Turing machine associated with the grammar $\mathcal{AG}_1$ in Example 4. First of all, observe that, if we ignore the rhombuses and the boxes inside the *finite control*, we obtain exactly $M(\mathcal{G}_1)$, the deterministic finite automaton for $\mathcal{AG}_1$. The picture snaps the machine while it is evaluating the predicate in the lower-side rhombus, and the input-cursor is hovering over \$. As $prev_n = prev_h$, then the machine will move to the final state S. Between two different states of the original finite state machine there are two or three boxes. The first one always contains all the functions to be computed. The second one (if any) evaluate possible predicates of a production, and the last one sets the value of $prev_x$ to $curr_x$ for each attribute x. This is performed at the end of each macro-step. If some predicate is not satisfied, then the machine goes to the ***reject*** state. Finally, notice that tape *pos* is not used for $\mathcal{AG}_1$ (it is useful for right-regular grammars, see below), and that the value of *rul* is determined by the algorithm performing the computation of attribute values (not detailed here).

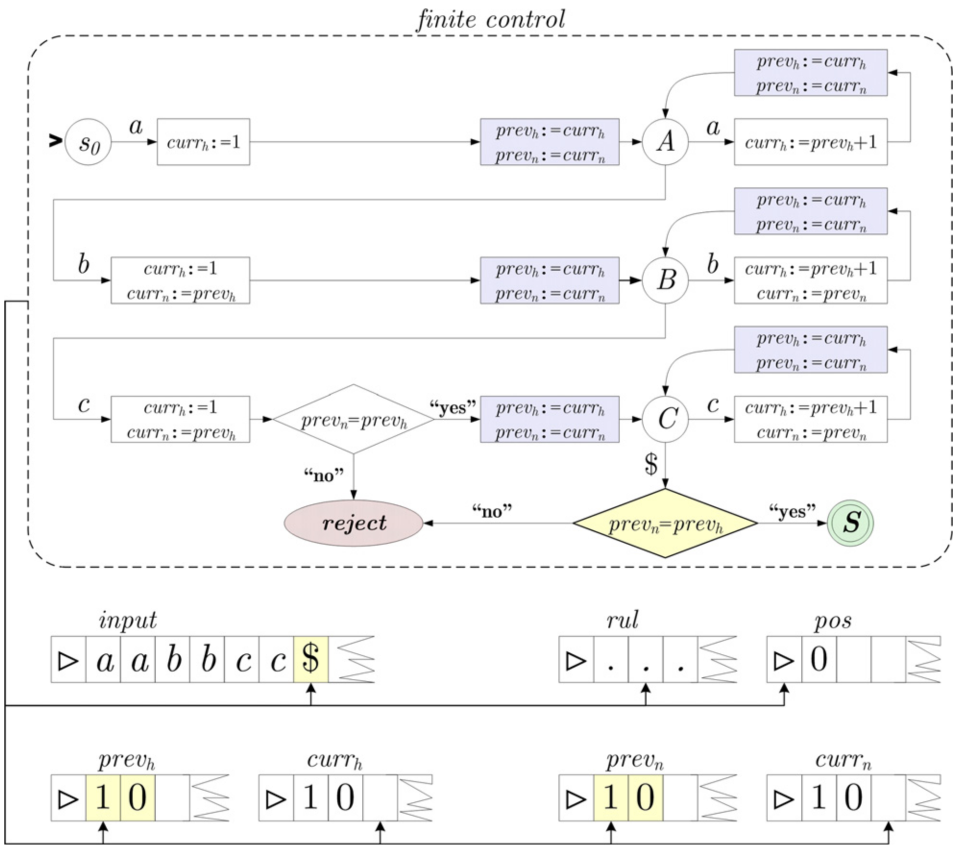


Fig. 5. The Turing machine $TM(\mathcal{AG}_1)$ for the grammar $\mathcal{AG}_1$ in Example 4.

$p_6 : B \to c$	$h := 0$		
$p_5 : B \to bB$	$h := h + 1$		
$p_4 : A \to bB$	$h_A := 0,$	$n_A := h_B + 1$	
$p_3 : A \to aA$	$h := h + 1,$	$n := n$	
$p_2 : S \to aA$			$h_A + 1 = n_A$
$p_1 : S \to bB$			$h_B > 2$

Fig. 6. The $\mathcal{AG}_2$ grammar.

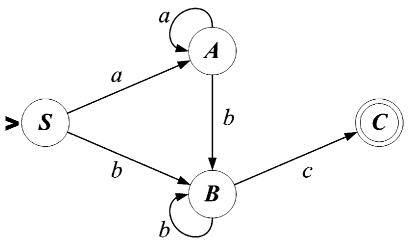


Fig. 7. The DFA for the regular grammar underlying $\mathcal{AG}_2$.

5.1.2. *DRReg*^{A} attribute grammars

The case of right-regular grammars requires a quite different approach. Consider the following attribute grammar $\mathcal{AG}_2$ (see Fig. 6) belonging to *DRReg*^{A} and defining the language $\{a^n b^n c\colon\ n > 0\} \cup \{b^n c\colon\ n > 3\}$.

The grammar $\mathcal{G}$ underlying $\mathcal{AG}_2$ is deterministic right-regular (see Fig. 7), and thus unambiguous. Fig. 8 shows the parse trees for $\mathcal{AG}_2$ on strings *aabbc* and *bbbbc*, respectively.

Note that a machine based on $M(\mathcal{G})$ such as $TM(\mathcal{AG})$ (described above), which parses w from left to right, would require an extra work-tape of polynomial space to decide whether w belongs to $\mathcal{L}(\mathcal{AG})$, because functions and predicates could not be directly computed. In fact, imagine we are parsing the example string *aabbc* as above, and the machine is at state A

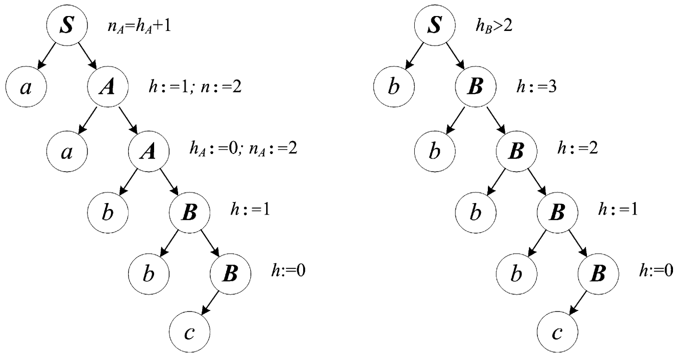


Fig. 8. Parse trees for $\mathcal{AG}_2$ on strings *aabbc* and *bbbbc*.

with the symbol $b \in \Sigma$ under the input-cursor. Note that $\delta(A, b) = B$, because of the production p of the form $A \to bB$ in $\mathcal{G}$. Then, such a $TM(\mathcal{AG})$ should compute $Func(p)$ and evaluate $Pred(p)$, but here the values of the attributes of B depend on the whole substring of w not yet parsed. Therefore, the machine has to "stack" the sequence of functions and predicates (generally linear in $\|w\|$) that may be computed when w has been parsed, until there are enough elements to evaluate the current predicate.

The above drawback of the usual left-to-right string parsing is not acceptable for our purposes, because we are interested in logspace membership results for classes of $DRReg^A$ grammars. We thus resort to a different approach, based on the reverse parsing direction.

Let $\mathcal{AG} = \langle \mathcal{G}, Attr, Func, Pred \rangle$ be any right-regular integer attribute grammar. We define the automaton $M^r(\mathcal{G}) = \langle K, \Sigma, \delta^r, s_0^r, \{s_0\} \rangle$ as the "reverse" of $M(\mathcal{G})$, in that $A \in \delta^r(B, b)$ if and only if $\delta(A, b) = B$. Observe that $M^r(\mathcal{G})$ may be non-deterministic, even if $M(\mathcal{G})$ is deterministic (as in the example above).

The Turing machine $TM(\mathcal{AG}) = \langle K', \Sigma, \delta', s_0^r \rangle$ for $\mathcal{AG}$ is based on $M^r(\mathcal{G})$ and parses the input string w from right to left. Thus, at the beginning of its computation on w, $TM(\mathcal{AG})$ moves to the end of w. Let B be the current state of $TM(\mathcal{AG})$ and b be the symbol in position j under the input-cursor. If $\|\delta^r(B, b)\| > 1$ (non-deterministic choice of the reverse automaton), then $TM(\mathcal{AG})$ performs the following steps: (i) save the value of j in the work-tape *pos*; (ii) move to the first symbol of w; (iii) execute the deterministic automaton $M(\mathcal{G})$ until the input-cursor is again in position j: say $A \in \delta^r(B, b)$ the current state; (iv) execute the functions and the predicate related to either the production $p : A \to b$ or $p : A \to bB$ depending on whether $B = s_0^r$ or not; (v) continue only if $Pred(p)$ is true. As a special case, notice that if Π also includes a production p of the form $A \to \varepsilon$, then $TM(\mathcal{AG})$ must initially compute $Func(p)$ and evaluate $Pred(p)$ before the last symbol of w (recall that $TM(\mathcal{AG})$ works from right to left) has been read. Note that, compared with the machine for grammars in $DLReg^A$, we use here an additional work-tape for saving the value of j at step (i).

5.1.3. Reg^A attribute grammars

We construct a multi-tape Turing machine as shown for the two cases above. However, recall that in this general case the automaton $M(\mathcal{G})$ may be non-deterministic, and thus $TM(\mathcal{AG})$ may be non-deterministic, as well. Again, when $\mathcal{G}$ is right-regular, then $TM(\mathcal{AG})$ works from right to left (extending $M^r(\mathcal{G})$).

5.2. On the properties of $TM(\mathcal{AG})$

We next prove that the above defined Turing machines correctly parse the various classes of IRGs we consider in this paper, and we point out some interesting properties of these machines that will be exploited in the main results of the subsequent section.

Proposition 5.1. *Let $\mathcal{AG}$ be an IRG, then $TM(\mathcal{AG})$ decides $\mathcal{L}(\mathcal{AG})$.*

Proof. Let $\mathcal{AG} = \langle \mathcal{G}, Attr, Func, Pred \rangle$ be a $DLReg^A$ attribute grammar and w be a string to be checked for membership in $\mathcal{L}(\mathcal{AG})$. By construction, $TM(\mathcal{AG})$ parses string w in the same way as $M(\mathcal{G})$, but it additionally computes the values of the attribute functions and evaluates the predicate.

If $w \notin \mathcal{L}(\mathcal{G})$, then $TM(\mathcal{AG})$ rejects the input by construction. Otherwise, if $w \in \mathcal{L}(\mathcal{G})$, consider the (unique) parse tree t of $\mathcal{G}$ yielding w. By definition of parse tree for $\mathcal{G}$, each non-leaf node v in t refers to a production of $\mathcal{G}$, say $p : B \to \beta$, such that v has label B. Since $\mathcal{G}$ is regular, v has at most one child labeled with a nonterminal symbol occurring in β. Moreover, as $\mathcal{G}$ is left-regular, this child (if there exists) is also left-regular. Now recall that $TM(\mathcal{AG})$ (following $M(\mathcal{G})$) parses w from left to right. This means that t is "scanned" from the leftmost (and also deepest) non-leaf node to the rightmost one (the root of t). So the value of any attribute of B in v depends on the only attributes of at most another non-leaf node already parsed. Hence, $TM(\mathcal{AG})$ correctly uses (as described above) just two memory-tapes for each attribute x, one for keeping the current value of x and another one for storing the value computed at the previous step. Finally, if some predicate in $Pred(p)$ fails, $TM(\mathcal{AG})$ immediately halts and rejects w, according to the definition of valid parse tree for $\mathcal{AG}$. Otherwise, if the end of the string is reached, then all the predicates in t have been successfully evaluated. It follows that $TM(\mathcal{AG})$ accepts w if, and only if, it belongs to $\mathcal{L}(\mathcal{AG})$.

Let $\mathcal{AG} = \langle \mathcal{G}, Attr, Func, Pred\rangle$ be a $DRReg^A$ grammar and w be a string to be checked for membership in $\mathcal{L}(\mathcal{AG})$. If w does not belong to $\mathcal{L}(\mathcal{G})$, it is clearly rejected by $TM(\mathcal{AG})$. Otherwise, let t be the (unique) parse tree of $\mathcal{G}$ yielding w. Here, when we are analyzing some character at position j, that is, we are considering some node v of t, unlike the previous case, the attribute values of the terminal symbol B at v depend on at least one non-leaf node of v dealing with characters to the right of v. For this reason, the machine is guided by $M^r(\mathcal{G})$ and the string w is parsed from right to left. Indeed, this way, at each step j, there is information enough to compute all attributes and to evaluate the predicate. However, since $M^r(\mathcal{G})$ is non-deterministic in general, the "direct" automaton $M(\mathcal{G})$ is exploited to decide, by starting on the left of the string, which choice should be taken at step j. More precisely, $M(\mathcal{G})$ is executed until the character at position j is reached. By storing information on the state of $M(\mathcal{G})$ before the step that eventually leads to position j, it is possible to remove the non-determinism from $M^r(\mathcal{G})$. Indeed, its choice from j to $j-1$ may be guided by the knowledge of the state of $M^r(\mathcal{G})$ at position $j-1$. It follows that, at each step, $TM(\mathcal{AG})$ may require the execution of $M(\mathcal{G})$ on w, and thus performs $\mathcal{O}(\|w\|^2)$ steps to parse w, plus the cost for computing the attribute values and evaluating the logical predicates.

Finally, let $\mathcal{AG} = \langle \mathcal{G}, Attr, Func, Pred\rangle$ be a Reg^A grammar. In this case, the proof follows the same line of reasoning as in the previous cases, but here the machine $TM(\mathcal{AG})$ is inherently non-deterministic, because it is non-deterministic even the basic parser $M(\mathcal{G})$ it is based on. $\square$

Thus, the general algorithm encoded by the Turing machine $TM(\mathcal{AG})$ is able to parse all kinds of IRGs considered in this paper. Moreover, from the above proof, it easily follows that for both left-regular and right-regular integer grammars such a $TM(\mathcal{AG})$ is a deterministic Turing machine. Just observe that all operators and predicates dealing with the integer attributes may be implemented as deterministic routines.

Proposition 5.2. *For any attribute grammar $\mathcal{AG}$ belonging to either class $DLReg^A$ or $DRReg^A$, $TM(\mathcal{AG})$ is deterministic.*

Moreover, we next point out that the number of calls to the sub-routines dealing with integer attributes is polynomially bounded by the size of the input string.

Proposition 5.3. *For any attribute grammar $\mathcal{AG}$, $TM(\mathcal{AG})$ performs a polynomial number of calls to the routines computing attribute values and evaluating attribute predicates.*

Proof. Let $\mathcal{AG} = \langle \mathcal{G}, Attr, Func, Pred\rangle$ be an attribute grammar and w be a string of length n to be checked for membership in $\mathcal{L}(\mathcal{AG})$.

Note that the number of calls performed by $TM(\mathcal{AG})$ for computing attribute values and evaluating attribute predicates is polynomially related to the number of non-leaf nodes (recall that leaf nodes are labeled with terminal symbols which do not have attributes) in any parse tree t of $\mathcal{G}$ yielding w. However, for every regular grammar, and hence for $\mathcal{G}$, every parse tree has precisely n non-leaf nodes. $\square$

Proposition 5.4. *For any attribute grammar $\mathcal{AG}$ belonging to the class σ-$Reg^A_{\otimes}$, $TM(\mathcal{AG})$ operates within space $\mathcal{O}(\log n)$.*

Proof. Let $x_1, \ldots, x_k$ be the attributes of $\mathcal{AG} = \langle \mathcal{G}, Attr, Func, Pred\rangle$ and w be any input string of length n. In order to determine an upper-bound for the space employed by the Turing machine $TM(\mathcal{AG})$ for dealing with the integer attributes, we assume w.l.o.g. that $\mathcal{AG}$ contains only expressions over positive integers using the one operator $+$ (because this is the case leading to the maximum growth of the attribute-values in $\{|\ |, +, -, \div, \%\}$). Recall that, at each step j ($1 \leqslant j \leqslant n$), the value of any integer attribute x computed until step $j-1$ is stored in the work-tape $prev_x$, and it is denoted by $prev_x(j)$. Moreover, the work-tape $curr_x$ is (possibly) used to compute the new value $curr_x(j)$ for x, according to the production rules at hands.

Let h and c be the maximum number of terms and the maximum value occurring in any production rule of $\mathcal{AG}$, respectively. Let j be the current step of the machine while parsing the input string, that is, assume we are focusing on the jth symbol of w. Since $\mathcal{AG}$ is strict, at each step, every attribute may contribute to the computation of one attribute only. Thus, at step j let $\mathcal{S} = S_{x_1}, \ldots, S_{x_k}$ be a partition of the subset of $Attr$ containing those attributes that contribute to attribute computations at step j, with S_{x_i} denoting those attributes that effectively contribute to the computation of x_i ($1 \leqslant i \leqslant k$). Therefore, we have

$$curr_{x_i}(j) = \sum_{y \in S_{x_i}} prev_y(j) + c_1 + \cdots + c_s$$

where $c_1, \ldots, c_s$ are the constant values occurring in such a production rule. It follows that

$$curr_{x_i}(j) \leqslant \sum_{y \in S_{x_i}} curr_y(j-1) + hc.$$

Since $\mathcal{S}$ is a partition of a subset of the attributes, by computing the summation over all the k attributes, we also get

$$\sum_{x \in Attr} curr_x(j) \leqslant \sum_{x \in Attr} \left(\sum_{y \in S_x} curr_y(j-1) + hc \right) \leqslant \sum_{x \in Attr} curr_x(j-1) + khc.$$

Clearly, for each $z \in Attr$ at step n,

$$curr_z(n) \leqslant \sum_{x \in Attr} curr_x(j-1) + hc.$$

By exploiting the above relationship from step $j-1$ down to 0, we get

$$curr_z(n) \leqslant \sum_{x \in Attr} curr_x(j-2) + khc + hc \leqslant \sum_{x \in Attr} curr_x(j-3) + 2khc + hc \leqslant \cdots.$$

By recalling that $curr_z(0) = 0$ for each $z \in Attr$, it is easy to obtain

$$curr_z(n) \leqslant (n-1)khc + hc.$$

Note that this upper-bound value can be represented in $\mathcal{O}(\log n)$. Moreover, since we have only a constant number (k) of attributes, in fact all of them may be encoded and evaluated in $\mathcal{O}(\log n)$.

Also, observe that arithmetic expressions over $\{|\ |, +, -, \div, \%\}$ are (widely) linear space computable (see Appendix A.3), and that the logical expression to be evaluated for any production predicate has constant size, and hence involve a constant number of such arithmetic expressions. It follows that *rul* needs $\mathcal{O}(\log n)$ space, too. Therefore, the application of any function or predicate of $\mathcal{AG}$ (on attributes of size $\mathcal{O}(\log n)$) is still feasible in space $\mathcal{O}(\log n)$.

Finally, observe that the work-tape *pos* is used to store a counter, and thus requires size $\mathcal{O}(\log n)$, as well, and that $TM(\mathcal{AG})$ has no further memory requirements. □

It is easy to see that, whenever the strict requirement is not fulfilled or the multiplication operator $*$ is allowed to occur in arithmetic expressions, the above $\mathcal{O}(\log n)$ space upper-bound cannot be guaranteed. However, we next show that $TM(\mathcal{AG})$ can always be implemented in such a way that polynomial space is enough for parsing any IRG.

Proposition 5.5. *For any attribute grammar $\mathcal{AG}$ belonging to the class σ-Reg^A, $TM(\mathcal{AG})$ operates within space $\mathcal{O}(n)$.*

Proof. We follow the same line of reasoning as in the proof of Proposition 5.4, but replacing the operator $+$ with the operator $*$. Therefore, at any step $1 \leqslant j \leqslant n$ we have that

$$\prod_{x \in Attr} curr_x(j) \leqslant c^{hk} \prod_{x \in Attr} curr_x(j-1),$$

and then, for each $z \in Attr$,

$$curr_z(n) \leqslant c^{hk(n-1)} c^h.$$

It follows that these integer values may be represented and evaluated in space $\mathcal{O}(n)$. □

Proposition 5.6. *For any attribute grammar $\mathcal{AG}$ belonging to the class $Reg^A_{\otimes}$, $TM(\mathcal{AG})$ operates within space $\mathcal{O}(n)$.*

Proof. Let $x_1, \ldots, x_k$ be the attributes of $\mathcal{AG} = \langle \mathcal{G}, Attr, Func, Pred \rangle$ and w be any input string of length n. Here, as for Proposition 5.4, we next consider the only (worst-case) operator $+$.

Let h and c be the maximum number of terms and the maximum value occurring in any production rule of $\mathcal{AG}$, respectively. Let j be the current step of the machine while parsing the input string, that is, assume we are focusing on the jth symbol of w. Since $\mathcal{AG}$ is not strict, at each step, the value of an attribute computed at step $j-1$ can contribute more than once to the computation of an attribute at step j. Thus, at the current step, $\mathcal{S} = S_{x_1}, \ldots, S_{x_k}$ is not a partition any more (it could even happen that any $S_{x_i} \equiv Attr$). Moreover, an attribute in S_{x_i} may contribute more than once (and at most h times) to the computation of x_i ($1 \leqslant i \leqslant k$). Therefore, we have

$$curr_{x_i}(j) \leqslant \sum_{y \in S_{x_i}} h \cdot curr_y(j-1) + hc \leqslant h\left(\sum_{x \in Attr} curr_x(j-1) + c \right).$$

Then, by computing the summation over all the k attributes, we also get

$$\sum_{x\in Attr} curr_x(j) \leqslant kh\Big(\sum_{x\in Attr} curr_x(j-1)+c\Big)$$

and clearly, for each $z \in Attr$ at step $j = n$,

$$curr_z(n) \leqslant h\Big(\sum_{x\in Attr} curr_x(j-1)+c\Big).$$

By exploiting the above relationship from step $j-1$ down to 0, we get

$$curr_z(n) \leqslant kh^2\Big(\sum_{x\in Attr} curr_x(j-2)+c\Big) + hc \leqslant k^2h^3\Big(\sum_{x\in Attr} curr_x(j-3)+c\Big) + kh^2c + hc\cdots.$$

By recalling that $curr_z(0) = 0$ for each $z \in Attr$, it is easy to obtain

$$curr_z(n) \leqslant c\sum_{i=1}^{n} k^{j-1}h^j = ch\frac{(hk)^n-1}{hk-1}.$$

Note that this upper-bound value can be represented in $\mathcal{O}(n)$. Moreover, since we have only a constant number (k) of attributes, in fact all of them may be encoded and evaluated in $\mathcal{O}(n)$. $\square$

The case of general integer expression, where $*$ may occur in the production rules, and any attribute may contribute many times to the computation of one or more attributes seems much more difficult, as shown in the following example.

Example 5. Consider the following IRG:

$$\begin{array}{llll} p_3: A \to a, & h := 2, & k := 2, & \\ p_2: A \to aA, & h := h*k, & k := h*k, & \\ p_1: S \to \$A, & & & h_A = k_A. \end{array}$$

It is very easy to see that the value of both attributes h and k grows up to $2^{(2^{n-1})}$, which has an exponential space representation.

Nevertheless, we next show that, differently from similar problems like the parsing of string attribute grammars with the concatenation operator [12], we may avoid here the curse of exponentiality. To this end, we have to implement the algorithm encoded by $TM(\mathcal{AG})$ on a suitable non-deterministic polynomial-time *Random Access Machine* (*RAM*) that can be simulated, in its turn, on a polynomial-space Turing machine [21]—see Appendix A.5.

Proposition 5.7. PARSE$_{[Reg^A]}$ *is in* **PSPACE**.

Proof. Let $\mathcal{AG} = \langle \mathcal{G}, Attr, Func, Pred\rangle$ be any IRG in Reg^A, and assume for the sake of simplicity that $\mathcal{G} = \langle \Sigma, N, S, \Pi\rangle$ is left-regular. We first describe a non-deterministic RAM $R(\mathcal{AG})$ that performs the same task as the Turing machine $TM(\mathcal{AG})$ in a polynomial number of steps, by simulating any $TM(\mathcal{AG})$ computation.

Without loss of generality, fix the alphabet $\Sigma = \{0, 1\}$. Let w be any input string of length n for $TM(\mathcal{AG})$. Then, the input for $R(\mathcal{AG})$ will be the integer **w**, whose string representation is 1 followed by the reverse w^R of string w, that is, 2^n plus the integer value of w^R. For instance, if $w = 1100$ then **w** is the number 19, or 10011, in binary representation. By definition of RAM, its input **w** is stored in the register R_0, when the computation starts. The other registers of $R(\mathcal{AG})$ are the following: a register R_{state} containing (the integer-encoding of) the current state of the simulated $M(\mathcal{G})$; two registers, say $R_{curr(x)}$ and $R_{prev(x)}$, for each attribute x in $Attr$, containing the same values as the corresponding work-tapes of $M(\mathcal{G})$; and an additional register R_{symb} that holds, at every step, the symbol currently read by the input-tape head of $M(\mathcal{G})$, i.e., more precisely, either the number 0 or 1, depending on such a current symbol.

The machine $R(\mathcal{AG})$ performs n macro-iterations. Each iteration begins by executing the pair of instructions $R_{symb} \leftarrow R_0 - (R_0 \div 2)*2$ and $R_0 \leftarrow R_0 \div 2$ in such a way that, at iteration j, the register R_{symb} holds (the integer whose string representation is) $w[j]$, while R_0 encodes the remaining part of the string to be parsed. Moreover, suppose that, at the beginning of the current macro-iteration, $R_{state} = A$ and $R_{sym} = a$. Then, $R(\mathcal{AG})$ jumps to one of the instructions labeled by aA, dealing with some production of the form $p : B \to aA$, by mimicking $TM(\mathcal{AG})$. The instructions of $R(\mathcal{AG})$ starting at this label act as follows: (i) the functions contained in $Func(p)$ are computed and the values of attributes are updated, accordingly; (ii) the register R_{state} is set to B; and (iii) a conditional jump is performed, leading $R(\mathcal{AG})$ to the next macro-iteration if the predicate $Pred(p)$ is evaluated true, and to the **reject** instruction otherwise. Clearly, the number of instructions executed at any of these n macro-steps is bounded by a constant, and thus $R(\mathcal{AG})$ requires $O(n)$ steps to end its computation. Moreover, it accepts **w** if, and only if, $TM(\mathcal{AG})$ accepts w (instruction **accept**).

Since **w** can be (trivially) computed in deterministic polynomial time from w, the statement follows by recalling that such a non-deterministic polynomial-time RAM $R(\mathcal{AG})$ can be simulated on a polynomial-space Turing machine [21].[6]

For completeness, note that we may proceed with the same line of reasoning to simulate the Turing machine associated with a right-regular grammar, but in this case there is no need to reverse the string representation of w to build **w**. □

6. The complexity of parsing IRGs

The above Proposition 5.7 shows that, in the general case, parsing any language generated by regular grammars with integer attributes is feasible in **PSPACE**. In this section, we study the complexity of parsing IRGs under various kinds of restrictions, looking for possible tractable classes. All these results are completeness results, whose proofs of membership are greatly simplified by the properties of the Turing machines described in the previous section, but whose proofs of hardness are sometimes quite involved. Thus, for the sake of presentation and for giving more insights on the gadgets to be used with attribute grammars, we first describe some IRGs that encode classical complete problems for different complexity classes, that will be later used in the complexity proofs.

6.1. Some useful grammars

The first problems that we consider are graph problems. Thus, to solve these problems by using IRGs, we next fix a suitable string encoding of graphs. Without loss of generality for our results we assume hereafter that graphs have no self-loops, that is, they have no arc of the form (i, i).

Any graph $G = \langle V, E\rangle$ (directed or undirected) can be represented in terms of its *adjacency matrix*, which can be linearized as a string in $\{0, 1\}^*$ of length $|V|^2$ by first arranging the first row of the matrix, then the second row, and so on. For our purposes, it is useful to consider matrices where rows and columns are separated by two special symbols, say # and ⋄. Given any graph G, we denote by $\kappa(G)$ this string. For example, if $G = \langle\{1, 2, 3\}, \{(1, 2), (2, 3)\}\rangle$ we have $\kappa(G) = 0 \diamond 1 \diamond 0 \ \# \ 0 \diamond 0 \diamond 1 \ \# \ 0 \diamond 0 \diamond 0$.

Moreover, note that we are interested in regular grammars to be parsed in one string scanning, and in graph problems that, in general, require multiple accesses to the input graph to be solved. Therefore, we find useful to consider input strings where $\kappa(G)$ is replicated m times, with $m \geqslant |V| - 1$. More precisely, for any graph G, we define the string

$$w(G) = v(G)\ \big(\#\ \kappa(G)\big)^m\ \$,$$

where # is a separator, \$ is the string terminator, and $v(G)$ is a string in $\{0, 1\}^*$ that encodes the number of nodes of G, in that its number of 1s is equal to $|V|$.

Of course, building the encoding $w(G)$ from a graph G or, more precisely, from its natural encoding $\kappa(G)$ is an *easy* problem. However, we shall later discuss in some detail the precise complexity of this construction, because this issue is relevant to prove membership in (low) complexity classes.

6.1.1. Grammar RAG

The first IRG showed in this section is called RAG. We will pay attention to it because of its interesting properties strictly related to the problem REACHABILITY: given a directed graph G, decide whether there is a path in G from node 1 to node n. We show that G is a "yes" instance of REACHABILITY if and only if $w(G) \in \mathcal{L}(\mathsf{RAG})$. The regular grammar underling RAG, henceforth called RG, defines the language

$$\mathcal{L}(\mathsf{RG}) = \big\{(0|1)^h\#(0|1)\big((\#|\diamond)(0|1)\big)^k\$\text{: } h > 0,\ k \geqslant 0\big\},$$

which of course includes the string encoding $w(G)$ of every graph G.

Fig. 9 shows an automaton recognizing this language and isomorphic to the regular grammar described below.

We use this figure to give an intuition of how the IRG works, describing how the integer attributes are employed state-by-state. At state A, we count the number of nodes of the graph, that is the number of 1s in the substring $v(G)$ of $w(G)$,

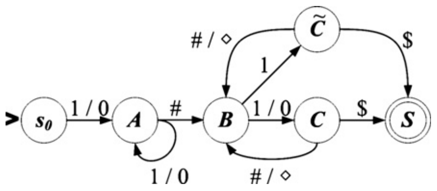


Fig. 9. The *NFA* associated with the grammar underlying RAG.

[6] The RAMs considered in this paper are those called powerful RAMs, or PRAMs, in [21].

and store this number in the attribute ν. Then, we proceed to the scan of the m copies of the adjacency matrix $\kappa(G)$ of the graph. We maintain attributes i and j ranging over rows and columns, respectively, of the (current instance of the) adjacency matrix. Moreover, at any step, the attribute c stores the current node, that is the last node of the graph that we have visited. Finally, the attribute r encodes the result of the procedure. Indeed, it will be assigned 1 if we have reached the target node n, and 0 otherwise.

Note that the automaton is non-deterministic. In fact, when we are scanning the adjacency matrix and we are at state B, by reading a 1 in the matrix, we may go either to state C or to state $\widetilde{C}$. If $i = c$, i.e. we are scanning the row with the arcs starting from the current node c, this non-deterministic step is used to encode the choice of following some arc (i, j) or not. More precisely, going to $\widetilde{C}$ means ignoring that arc, while going to C means following the arc. In the latter case, j becomes the new current node, that is the assignment $c := j$ should be executed for updating the value of attribute c.

Formally, attribute grammar RAG is defined as follows:

$$\begin{array}{ll}
p_0 : A \to 0, & \nu := 0, \\
p_1 : A \to A0, & \nu := \nu, \\
p_2 : A \to 1, & \nu := 1, \\
p_3 : A \to A1, & \nu := \nu + 1, \\
p_4 : B \to A\#, & \nu_B := \nu_A,\ c_B := 1,\ i_B := 1,\ j_B := 1,\ r_B := \underline{(\nu_A = 1)\ \mathbf{?}\ 1\ \mathbf{:}\ 0}, \\
p_5 : B \to C\#, & \nu_B := \nu_C,\ c_B := c_C,\ i_B := \underline{(i_C = \nu_C)\ \mathbf{?}\ 1\ \mathbf{:}\ i_C + 1},\ j_B := 1,\ r_B := r_C, \\
p_6 : B \to C\diamond, & \nu_B := \nu_C,\ c_B := c_C,\ i_B := i_C,\ j_B := j_C + 1,\ r_B := r_C, \\
p_7 : C \to B1, & \nu_C := \nu_B,\ c_C := \underline{(i_B = c_B)\ \mathbf{?}\ j_B\ \mathbf{:}\ c_B},\ i_C := i_B,\ j_C := \underline{(i_B = c_B)\ \mathbf{?}\ 0\ \mathbf{:}\ j_B}, \\
 & r_C := \underline{(i_B = c_B \wedge j_B = \nu_B)\ \mathbf{?}\ 1\ \mathbf{:}\ r_B}, \\
p_8 : C \to B0, & \nu_C := \nu_B,\ c_C := c_B,\ i_C := i_B,\ j_C := j_B,\ r_C := r_B, \\
p_9 : S \to C\$, & r_C = 1, \\
p_{10} : B \to \widetilde{C}\#, & \nu_B := \nu_{\widetilde{C}},\ c_B := c_{\widetilde{C}},\ i_B := \underline{(i_{\widetilde{C}} = \nu_{\widetilde{C}})\ \mathbf{?}\ 1\ \mathbf{:}\ i_{\widetilde{C}} + 1},\ j_B := 1,\ r_B := r_{\widetilde{C}}, \\
p_{11} : B \to \widetilde{C}\diamond, & \nu_B := \nu_{\widetilde{C}},\ c_B := c_{\widetilde{C}},\ i_B := i_{\widetilde{C}},\ j_B := j_{\widetilde{C}} + 1,\ r_B := r_{\widetilde{C}}, \\
p_{12} : \widetilde{C} \to B1, & \nu_{\widetilde{C}} := \nu_B,\ c_{\widetilde{C}} := c_B,\ i_{\widetilde{C}} := i_B,\ j_{\widetilde{C}} := j_B,\ r_{\widetilde{C}} := r_B, \\
p_{13} : S \to \widetilde{C}\$, & r_{\widetilde{C}} = 1.
\end{array}$$

Note that RAG is strict and does not use the $*$ operator. Thus, this grammar belongs to $\sigma\text{-}Reg^A_\otimes$. We next give a more detailed analysis of RAG, by focusing on the role played by integer attributes: Attribute ν first counts (productions p_0–p_3), and then keeps (productions p_4–p_8 and p_{10}–p_{12}) the number n of nodes in G. Attribute i changes its value when the current symbol is #. It is initially set to 1 (productions p_4). Going on, by the expression $i := \underline{(i = \nu)\ \mathbf{?}\ 1\ \mathbf{:}\ i + 1}$, it is incremented until it reaches n and then it is reset to the initial value 1 (production p_5). Also the value of j is initially set to 1 (productions p_4), yet it changes when the current symbol is $\diamond$ (productions p_6). Moreover, when the symbol under the input-cursor is 1 (production p_7), then the pair of values i, j encodes an arc (i, j) of the graph G. In this case, if $i = c$ holds (j is connected to the current node c), then the edge can be added to the path, c changes its value to j (see expression $c := \underline{(i = c)\ \mathbf{?}\ j\ \mathbf{:}\ c}$), and j is set to the constant value 0 ($j := \underline{(i = c)\ \mathbf{?}\ 0\ \mathbf{:}\ j}$). This last point deserves some further attention. Indeed, by definition something should be assigned to attribute j. However, we want this IRG be strict, and j is already used in a branch of the previous arithmetic-if statement in this production. It follows that, at least in the first branch, j cannot be used again, otherwise it would give a double contribution to this production. On the other hand, observe that this value 0 cannot lead to any trouble, because in this production $i = c$ entails c is set to j. Then, since $(i, j) \in E$ entails $i \neq j$ (there are no self-loops), the condition $i = c$ will be never true again while scanning the current row of the adjacency matrix (i.e., until i will be changed in a subsequent execution of production p_5). Then, j will be set to 1, and the scanning of a new row will start correctly. Eventually, productions p_4 and p_7 set attribute r to 1 if n is reached and 0 if not. Please, notice that n can be reached before $w(G)$ has been read completely, and it is possible to find a new edge $(i = c = \nu, j)$ that extends the path. In this case, the expression $r := \underline{(i = c \wedge j = \nu)\ \mathbf{?}\ 1\ \mathbf{:}\ r}$ does not modify r, which keeps its previous value. Note that the condition of this arithmetic-if could be written equivalently $(|i - c| + |j - \nu| = 0)$, in a more arithmetic-oriented style. We use both forms in this paper, choosing each time the one that seems (to us) more readable.

As outlined above, production p_{12}, differently from production p_7, "skips" unconditionally the current edge (i, j). In fact, (i, j) is left also if the current path ends with the node i when $c = i$. The productions p_{10}, p_{11} and p_{13} are like the productions p_5, p_6 and p_9, where nonterminal $\widetilde{C}$ is used instead of C.

At this point, only predicate $r = 1$ (production p_9) needs to be evaluated. We claim that predicate $r = 1$ holds true if and only if node n is reached. Indeed, by construction, r is set to 1 only if node n is reached, that is only if the last reached node is some i, and there is an outgoing arc (i, j), with $j = n$ (i.e., $j = \nu$). The reverse holds as well, because if node n is reachable from node 1 there is a sequence of at most $n-1$ arcs leading from 1 to n, and thus the non-deterministic automaton may correctly chose such arcs (going to state C rather than to $\widetilde{C}$), while scanning the $n-1$ copies of the adjacency matrix of G. It follows that $w(G) \in \mathcal{L}(\mathsf{RAG})$ if and only if there is a path from node 1 to node n in G.

6.1.1.1. Grammar HPAG The second grammar shown in this section is related to problem HAMILTON-PATH. It is designed in such a way that given a digraph G, there is a path in G that visits each node exactly once if and only if $w(G) \in \mathcal{L}(\mathsf{HPAG})$.

The productions of HPAG are the following:

$$
\begin{array}{ll}
p_0 : A \to \varepsilon, & \nu := 1, \\
p_1 : A \to A1, & \nu := 2 * \nu, \\
p_2 : A \to A0, & \nu := \nu, \\
p_3 : B \to A1, & s_B := 1,\ \nu_B := 2 * \nu_A,\ h_B := 1, \\
p_4 : B \to B1, & s := 2 * s,\ \nu := 2 * \nu,\ h := 2 * h, \\
p_5 : B \to B0, & s := s,\ \nu := \nu,\ h := h, \\
p_6 : C \to B\#, & c_C := s_B,\ i_C := 1,\ j_C := 1,\ k_C := 1,\ \nu_C := \nu_B - 1,\ h_C := h_B, \\
p_7 : C \to D\#, & c_C := c_D,\ i_C := \underline{(2 * i_D - 1 = \nu_D)\ \mathbf{?}\ 1\ \mathbf{:}\ 2 * i_D}, \\
 & j_C := 1,\ k_C := 1,\ \nu_C := \nu_D,\ h_C := h_D, \\
p_8 : C \to \widetilde{D}\#, & c_C := c_{\widetilde{D}},\ i_C := \underline{(2 * i_{\widetilde{D}} - 1 = \nu_{\widetilde{D}})\ \mathbf{?}\ 1\ \mathbf{:}\ 2 * i_{\widetilde{D}}}, \\
 & j_C := 1,\ k_C := 1,\ \nu_C := \nu_{\widetilde{D}},\ h_C := h_{\widetilde{D}}, \\
p_9 : C \to D\diamond, & c_C := c_D,\ i_C := i_D,\ j_C := 2 * j_D,\ k_C := 2 * k_D,\ \nu_C := \nu_D,\ h_C := h_D, \\
p_{10} : C \to \widetilde{D}\diamond, & c_C := c_{\widetilde{D}},\ i_C := i_{\widetilde{D}},\ j_C := 2 * j_{\widetilde{D}},\ k_C := 2 * k_{\widetilde{D}},\ \nu_C := \nu_{\widetilde{D}},\ h_C := h_{\widetilde{D}}, \\
p_{11} : D \to C1, & c_D := \underline{(i_C = c_C \wedge h_C \div j_C \% 2 = 0)\ \mathbf{?}\ j_C\ \mathbf{:}\ c_C},\ i_D := i_C, \\
 & j_D := \underline{(i_C = c_C \wedge h_C \div j_C \% 2 = 0)\ \mathbf{?}\ 1\ \mathbf{:}\ j_C}, \\
 & k_D := \underline{(i_C = c_C \wedge h_C \div k_C \% 2 = 0)\ \mathbf{?}\ 1\ \mathbf{:}\ k_C},\ \nu_D := \nu_C, \\
 & h_D := \underline{(i_C = c_C \wedge h_C \div k_C \% 2 = 0)\ \mathbf{?}\ h_C + k_C\ \mathbf{:}\ h_C}, \\
p_{12} : D \to C0, & c_D := c_C,\ i_D := i_C,\ j_D := j_C,\ k_D := k_C,\ \nu_D := \nu_C,\ h_D := h_C, \\
p_{13} : \widetilde{D} \to C1, & c_{\widetilde{D}} := c_C,\ i_{\widetilde{D}} := i_C,\ j_{\widetilde{D}} := j_C,\ k_{\widetilde{D}} := k_C,\ \nu_{\widetilde{D}} := \nu_C,\ h_{\widetilde{D}} := h_C, \\
p_{14} : S \to D\$ \mid \widetilde{D}\$, & \nu_D = h_D \mid \nu_{\widetilde{D}} = h_{\widetilde{D}}.
\end{array}
$$

Note that HPAG is strict and uses the $*$ operator. Thus, this grammar belongs to $\sigma\text{-}Reg^A$.

As for the previous case, this grammar is associated with a non-deterministic automaton, shown in Fig. 10. Here, the non-deterministic choice of following an arc (i, j) or not is performed by going to state D or $\widetilde{D}$, respectively. Note that, in this case, we have to store in some way all the vertices that we have visited, in order to avoid going twice in the same node and to check whether eventually we have visited all nodes of the graph. For this purpose, we use integer attributes as bit-vector sets, where the ith bit (starting from the right) is associated with node i of the graph G. For instance, the set of nodes $\{1, 3, 4\}$ is encoded with the integer 1101_b, where the subscript b denotes the binary representation. Similarly, a single number (node) 4 is encoded with the integer 1000_b. If i is such a mask encoding, we shall denote by $\hat{\imath}$ the corresponding number (node of the graph), that is, $i = 2^{\hat{\imath}-1}$ holds. Therefore, if an integer h encodes a set of nodes, to add the node $\hat{\imath}$ to this set, we just write $h := h + i$. Also, checking whether $\hat{\imath} \in h$ can be performed by checking whether $(h \div i)\%2$ is 0 or not.

We next give a detailed analysis of HPAG, by focusing on the role played by integer attributes. As for the grammar RAG, while scanning the current copy of the adjacency matrix of G, attributes i and j hold the current row number $\hat{\imath}$ and column number $\hat{\jmath}$, respectively. We also maintain a copy k of j, to be used in the production p_{11} in place of j, to avoid multiple

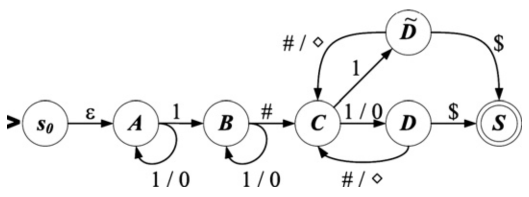


Fig. 10. The *NFA* associated with the grammar underlying HPAG.

contributions of this attribute. Otherwise, the strict-composition restriction of HPAG would be violated. Also, attribute s holds the start node $\hat{s}$ of the path (chosen non-deterministically through productions p_3 and p_4), while attribute c holds the last visited node, and the attribute h stores all visited nodes, at any step of the string parsing. Moreover, ν encodes the set of all n nodes of the graph, that is, $\nu = 11\cdots 1_b = 2^n - 1$.

More in detail, in the first productions (p_0–p_5) the encoding ν of the set of nodes is computed, the starting node s is "guessed" (by going to state B), h is initialized to the singleton $\{s\}$ (in fact, it is set to the same value), and the current node c is initialized to s. Moreover, both i and j are initialized to 1. Then, the scanning of the adjacency matrix starts. Of course, incrementing a row (column) index is here performed by doubling the attribute i (j). Note that, in the productions p_7 and p_8, the row index i may be reset to 1 if we have found the symbol # and the last row $\hat{n}$ has been scanned, i.e., if $i = 2^{n-1}$. This is checked through the expression $2 * i - 1 = \nu$, because recall that $\nu = 2^n - 1$, by construction. If this test fails we just proceed with the next row, by setting $i = 2 * i$ and $j = 1$.

During the computation, whenever $c = i$ and we found a 1 at position j, i.e., $(\hat{\imath}, \hat{\jmath})$ is the edge of G under consideration, then we have two possibilities: (i) add (production p_{11}) j to the set of reached nodes, after checking that $h \div j\%2 = 0$ (node $\hat{\jmath}$ is not yet in the path); or (ii) ignore (production p_{13}) edge $(\hat{\imath}, \hat{\jmath})$. In case (i), we change the value of c to j, we update the path from h to $h + k$ (recall that attribute k maintains a copy of j), and in addition we set j (and k) to some constant in order to meet the strict-composition restriction of the grammar (see the discussion above for grammar RAG). Eventually, we evaluate the (one) predicate of HPAG $\nu = h$ (production p_{14}).

We claim that the latter check is true, and thus $w(G)$ belongs to the language, if and only if G has a Hamiltonian path. Indeed, if $h = \nu$ then the set (encoded by) h contains all nodes, by construction. Moreover, the above described check in production p_{11} prevents from going more than once on the same node. It follows that the only way to visit all nodes is through a Hamiltonian path. On the other hand, if there is a Hamiltonian path in G, then clearly its $n - 1$ arcs can be traversed correctly by choosing the right applications of production p_{11}, while scanning the $n - 1$ copies of $\kappa(G)$.

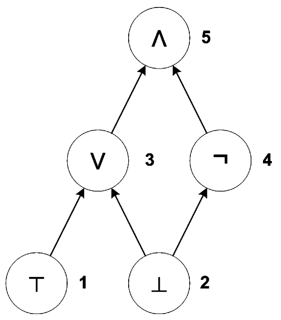
6.1.2. Grammar CVAG

The last notable IRG we describe in this section is related to the circuit-evaluation problem. Without loss of generality we consider in this paper circuits whose gates have fan-in two, at most. Recall that a (variable-free) Boolean circuit $C = \langle V, E, g\rangle$ is a directed acyclic graph where every node p has a label $g(p)$ that specifies its sort, that is, either a Boolean value in $\{\top, \bot\}$ (*true* or *false*, resp.), or a logical gate in $\{\wedge, \vee, \neg\}$. Since it is a DAG, we assume w.l.o.g. that nodes are numbered in such a way that every node has incoming arcs only from nodes assigned with lower numbers. As a consequence, the output gate of the Boolean circuit is the node with the highest number $n = |V|$. Given a node (gate) p ($p \leqslant n$), define $T(p)$ to be the usual Boolean evaluation function for the (sub)circuit of C consisting of all gates $i \leqslant p$. Then, the evaluation $T(C)$ of the circuit C is the Boolean value $T(n)$.

A natural encoding for such a circuit C is a matrix where, at coordinates (j, i) (row, column), with $i < j \leqslant n$, there is a 1 if there is an arc (i, j) in C, that is, if i is incoming in j. There are no further 1s. In other words, this part of the matrix is the transpose of the standard graph adjacency-matrix. And, for the above assumption, it is inferior-triangular. Moreover, there is an additional column $n + 1$ in this matrix, whose cell $(j, n + 1)$ contains the label of the circuit gate j. Fig. 11 shows a Boolean circuit and its matrix encoding. Then, for the circuit C, we denote by $\kappa(C)$ the row-by-row string encoding of its matrix, and by $w(C)$ the string \$ $\kappa(C)$ \$.

The grammar CVAG described below is designed in such a way that the value $T(C)$ of C is true if and only if $w(C) \in \mathcal{L}$(CVAG). The productions of CVAG are:

$$p_0 : A \to \$, \quad i := 1,\ j := 1,\ t := 0,\ c := 0,$$
$$p_1 : A \to A0, \quad i := 2 * i,\ j := j,\ t := t,\ c := c,$$
$$p_2 : A \to A1, \quad i := 2 * i,\ j := j,\ t := t,\ c := \underline{(((2 * t * i) \div j)\%2 = 1)\ \textbf{?}\ c + 1\ \textbf{:}\ c},$$



	1	2	3	4	5	
1	0	0	0	0	0	⊤
2	0	0	0	0	0	⊥
3	1	1	0	0	0	∨
4	0	1	0	0	0	¬
5	0	0	1	1	0	∧

Fig. 11. A Circuit and its encoding.

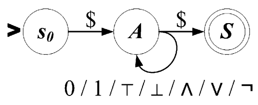


Fig. 12. The *NFA* associated with the grammar underlying CVAG.

$$
\begin{array}{llll}
p_3: A \to A\top, & i := 1,\ j := 2*j,\ t := 2*t+1, & & c := 0,\\
p_4: A \to A\bot, & i := 1,\ j := 2*j,\ t := 2*t, & & c := 0,\\
p_5: A \to A\wedge, & i := 1,\ j := 2*j,\ t := \underline{(c=2)\ \mathbf{?}\ 2*t+1\ \mathbf{:}\ 2*t}, & & c := 0,\\
p_6: A \to A\vee, & i := 1,\ j := 2*j,\ t := \underline{(c \geqslant 1)\ \mathbf{?}\ 2*t+1\ \mathbf{:}\ 2*t}, & & c := 0,\\
p_7: A \to A\neg, & i := 1,\ j := 2*j,\ t := \underline{(c=0)\ \mathbf{?}\ 2*t+1\ \mathbf{:}\ 2*t}, & & c := 0,\\
p_8: S \to A\$, & (t_A\%2) = 1. & &
\end{array}
$$

Note that Grammar CVAG is strict. Its arithmetic expressions employ the $*$ operator, and the underlying grammar is deterministic left-regular. Then, CVAG belongs to σ-*DLReg*^{A}. See the automaton shown in Fig. 12.

Let us explain how this IRG works, starting with a simple overview: After attributes initialization (in p_0), each row of the matrix is scanned through productions p_1 and p_2, which maintain suitable information on the evaluation of incoming gates. Then, productions p_3–p_7 evaluate the current gate. Eventually the comparison predicate in production p_8 decides whether C evaluates to true or not, depending on the evaluation of gate n.

The integer attributes i and j encode the column number $\hat{\imath}$ and the row number $\hat{\jmath}$ of the matrix, with the same technique used in the previous grammar HPAG. That is, for any matrix index $\hat{x} \leqslant n$, $x = 2^{\hat{x}-1}$. E.g., if $\hat{x} = 4$ then $x = 1000_b$. The attribute c is just a counter that stores the number of those gates p incoming in the current gate such that $T(p) = \top$. The most important attribute here is t that, when the current gate is $\hat{\jmath}$, encodes the evaluation $T(p)$ of every gate $p < \hat{\jmath}$ and eventually $T(\hat{\jmath})$, after productions p_3–p_7. To this end, the integer attribute t encodes a bit-vector set with these values, where the leftmost bit corresponds to the first gate of C, and so on. Note that this order is reversed with respect to the order of bit-vector sets employed for grammar HPAG. E.g., $t = 1011_b$ means here that only gate 2 evaluates to *false*, that is, $T(1) = \top$, $T(2) = \bot$, $T(3) = \top$, and $T(4) = \top$.

Now, let us analyze some implementation details of CVAG. In production p_2 we deal with some arc $(\hat{\imath}, \hat{\jmath})$ of C, where $\hat{\jmath}$ is the current gate, that is, we are scanning its corresponding row j in the circuit matrix. At this point, we have to check whether $T(\hat{\imath}) = \top$, because in this case the counter c should be incremented to encode the fact that gate $\hat{\jmath}$ has one more incoming gate $(\hat{\imath})$ that evaluates to *true*. This check is performed in the condition $((2*t*i) \div j)\%2 = 1$ of the arithmetic-if statement of this production. Indeed, the number $(2*t*i) \div j$ is the sub-vector of t where the bit associated with $\hat{\imath}$ is in the rightmost (parity) position. To see that, observe that $2*t*i$ gives a number where t is shifted to the left of $\hat{\imath}$ bits, and the subsequent division by j performs a shift to the right of $\hat{\jmath} - 1$ bits. Recall that, at this stage, t consists of $\hat{\jmath} - 1$ bits. Then, after the above computations, we get a number with $\hat{\jmath} - 1 + \hat{\imath} - (\hat{\jmath} - 1) = \hat{\imath}$ bits, which restores the situation when the bit for $\hat{\imath}$ has been added in the rightmost position. For instance, let $j = 10000_b$, $i = 100_b$ and $t = 1011_b$, which means that the current gate (row) $\hat{\jmath}$ is 5, that we are dealing with the arc $(3, 5)$ from gate 3, and that all the first four gates, but gate 2, evaluate to *true*. Then, $2*t*i$ is 1011000_b (shift to the left of three positions), and $(2*t*i) \div j = 101_b$ (shift to the right of four positions). Moreover, the rightmost symbol of 101_b encodes the truth value $T(3)$, because it is precisely the third bit starting from the left of the attribute t. Since $5\%2 = 1$, $T(3) = \top$ and we increment the value of c.

Therefore, when the row ends and we find the label $g(\hat{\jmath})$, by looking at c (in the case of logical gates) we have the information to set in t the value of $T(\hat{\jmath})$ (productions p_3–p_7). E.g., consider production p_5 where the gate symbol is $\wedge$: in this case, $T(\hat{\jmath})$ is *true* iff $c = 2$, that is, all the two incoming gates evaluate to *true*. Thus, if $c = 2$, we add the $\hat{\jmath}$th bit to t and set it to 1 by performing $t := 2*t+1$; otherwise t gets the value $2*t$, with the new bit set to 0.

Eventually, when the whole input string has been scanned, the rightmost bit of t encodes $T(n)$, and thus $T(C)$. In fact $w(G)$ is accepted iff $(t\%2) = 1$ (predicate in production p_8), that is, iff $T(C)$ evaluates to *true*.

6.2. Completeness results

In this section, we give all the proofs of our complexity analysis of IRGs. Hardness proofs are mainly based on the above constructions. We assume the reader is familiar with the computational complexity theory. However, for the sake of completeness, we report in Appendix A the main notions relevant to this paper, such as complexity classes, uniform circuits families, and so on.

Theorem 6.1. PARSE$_{[\sigma\text{-}Reg^A_\otimes]}$ *is* **NL**-*complete.*

Proof. (Membership) Let $\mathcal{AG}$ be an IRG in σ-*Reg*$^A_\otimes$, and *TM*($\mathcal{AG}$) be the non-deterministic Turing machine associated with it that recognizes $\mathcal{L}(\mathcal{AG})$. Then, the result immediately follows from Proposition 5.4, saying that *TM*($\mathcal{AG}$) works in space $\mathcal{O}(\log n)$.

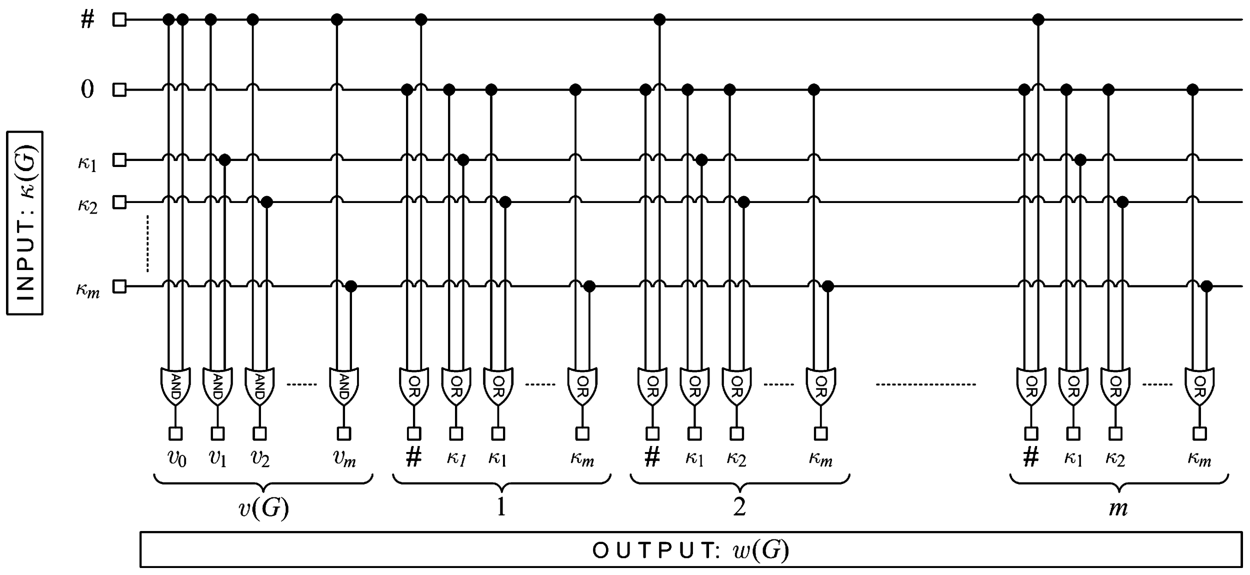


Fig. 13. The mth circuit of the $\mathbf{NC}^0$ family COPY.

(Hardness) Recall that REACHABILITY is the following **NL**-complete problem: given a directed graph $G = \langle V, E\rangle$ with $n = |V|$, decide whether there is a path in G from node 1 to node n.

We show that REACHABILITY $\leqslant_m^{\mathbf{NC}^0} \mathcal{L}(\mathsf{RAG})$, where RAG is the σ-$Reg^A_\otimes$ IRG described above in Section 6.1. The reduction is given by a **DLOGTIME**-uniform bounded fan-in Boolean circuit family of polynomial-size depth-one circuits, that we call COPY, containing only $\vee$ gates of fan-in (at most) two.[7]

Let $G = \langle V, E\rangle$ be a directed graph encoded by its linearized adjacency matrix $\kappa(G)$, and let $n = |V|$ and $m = |\kappa(G)|$. The mth circuit of the family COPY, shown in Fig. 13, outputs the string $w(G) = v(G)\ (\#\ \kappa(G))^m\ \$$. Thus, the circuit essentially computes m copies of $\kappa(G)$, besides the first string $v(G)$ encoding the cardinality of V. In particular, the leftmost part of COPY outputs $v(G)$, as it outputs a string of length $|\kappa(G)| + 1$ in such a way that, for each $i \in \{1,\dots,m\}$, $v(G)[i] = 1$ if $\kappa(G)[i] = \#$ and $v(G)[i] = 0$ otherwise; also $v(G)[0] = 1$. Clearly, $v(G)$ contains exactly n 1s, as the separator symbol # occurs $n-1$ times in $\kappa(G)$. The rest of the circuit simply replicates m times bit-by-bit the input string $\kappa(G)$, by executing the logical $\vee$ of every bit with the constant value 0.

Note that such a circuit has depth 1. Moreover, the family COPY is **DLOGTIME**-uniform, because the connections of its mth circuit can be recognized in **DLOGTIME** with respect to 1^m. Indeed, there are only two kinds of gates, whose connections can be easily determined. For instance, consider the circuit in Fig. 13 and assume that gates are numbered from left to write. Then, suppose we have to decide whether the input wire (bit) κ_j is connected to an OR gate O_r. Observe that the bit encoding of r has the form $r_a \cdot r_b$, where r_a encodes the number of the block O_r belongs to (i.e., either 0 for $v(G)$ or a number in $[1\dots m]$ identifying a copied instance), and r_b is the gate number within the r_ath block, and thus it ranges from 0 to m. Then, it is sufficient to check that $r_a > 0$ (the gate is of type OR) and $j = r_b$, which takes $O(\log m)$ time, as required (as encoding these numbers requires at most $O(\log m)$ bits, of course).

We conclude by recalling that, as shown in Section 6.1, G has a path from node 1 to node n if and only if $w(G) \in \mathcal{L}(\mathsf{RAG})$. $\square$

Theorem 6.2. PARSE$_{[\sigma\text{-}DLReg^A_\otimes]}$ *is* **L**-*complete.*

Proof. (Membership) Let $\mathcal{AG}$ be an IRG in σ-$DLReg^A_\otimes$, and $TM(\mathcal{AG})$ be the Turing machine associated with it that recognizes $\mathcal{L}(\mathcal{AG})$. Thus, the result follows from Proposition 5.2 and Proposition 5.4 saying that $TM(\mathcal{AG})$ is deterministic and works in space $\mathcal{O}(\log n)$, respectively.

(Hardness) Recall that DET-REACHABILITY is the following **L**-complete problem: given a *deterministic* digraph $G = (V, E)$ with $n = |V|$, decide whether there is a path in G from node 1 to node n. Moreover, let DLRAG be the σ-$DLReg^A_\otimes$ IRG obtained from the IRG RAG (see Section 6.1) by removing the productions p_{10}–p_{13}. We show that DET-REACHABILITY $\leqslant_m^{\mathbf{NC}^0} \mathcal{L}(\mathsf{DLRAG})$ holds.

[7] Formally, Boolean circuits compute only functions from $\{0,1\}^*$ to $\{0,1\}^*$, while our alphabet is $\{0,1,\#,\diamond,\$\}$. However, of course any symbol $w(G)$ can be encoded as a 3 bit string. Thus, for the sake of simplicity, we shall describe circuits as if they cover the whole alphabet. See Appendix A.3 for more information on **DLOGTIME**-uniform circuits.

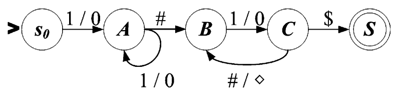


Fig. 14. The *DFA* associated with the grammar underlying DLRAG.

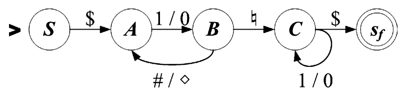


Fig. 15. The *DFA* associated with the grammar underlying DRRAG.

First, note that DLRAG is in σ-$DLReg^A_\otimes$ because its underlying grammar is deterministic (see Fig. 14), unlike grammar RAG, which is in σ-$Reg^A_\otimes$.

We claim that predicate $r = 1$ (production p_9) holds true if and only if node n is reached. Indeed, since the graph G is deterministic, at each step only one node can be chosen. With respect to the original grammar RAG, this means that here we do not need production p_{12} that "skips" unconditionally the current edge (i, j), as well as we do not need productions p_{10}, p_{11} and p_{12}, which only mimic what productions p_4, p_6 and p_9 do. Then, from the same reasoning as RAG, it follows that r is set to 1 only if node n is reached. The reverse holds as well, because if node n is reachable from node 1, then there is a *unique* sequence of at most $n - 1$ arcs leading from 1 to n, and thus the *deterministic* automaton may correctly follow such arcs and go to state C, while scanning the $n - 1$ copies of the adjacency matrix of G. Therefore, $w(G) \in \mathcal{L}(\text{DLRAG})$ if and only if there is a path from node 1 to node n in the deterministic graph G. Finally, note that $w(G)$ can be computed from the adjacency matrix encoding of G through the same uniform family COPY as in Theorem 6.1. □

Theorem 6.3. PARSE$_{[\sigma\text{-}DRReg^A_\otimes]}$ *is* **L**-*complete.*

Proof. (Membership) Same proof as Theorem 6.2.

(Hardness) We show that DET-REACHABILITY $\leqslant_m^{\mathbf{NC^0}}$ $\mathcal{L}(\text{DRRAG})$, where DRRAG is the following σ-$DRReg^A_\otimes$ grammar:

$$\begin{array}{ll}
p_0: S \to \$A, & r_A = 1, \\
p_1: A \to 0B, & \nu_A := \nu_B,\ c_A := c_B,\ i_A := i_B,\ j_A := j_B,\ r_A := r_B, \\
p_2: A \to 1B, & \nu_A := \nu_B,\ c_A := \underline{(i_B = c_B)\ \mathbf{?}\ j_B\ \mathbf{:}\ c_B},\ i_A := i_B,\ j_A := \underline{(i_B = c_B)\ \mathbf{?}\ 1\ \mathbf{:}\ j_B}, \\
 & r_A := \underline{(i_B = c_B \wedge j_B = \nu_B)\ \mathbf{?}\ 1\ \mathbf{:}\ r_B}, \\
p_3: B \to \#A, & \nu_B := \nu_A,\ c_B := c_A,\ i_B := \underline{(i_A = \nu_A)\ \mathbf{?}\ 1\ \mathbf{:}\ i_A + 1},\ j_B := 1,\ r_B := r_A, \\
p_4: B \to \diamond A, & \nu_B := \nu_A,\ c_B := c_A,\ i_B := i_A,\ j_B := j_A + 1,\ r_B := r_A, \\
p_5: B \to \natural C, & \nu_B := \nu_C,\ c_B := 1,\ i_B := 1,\ j_B := 1,\ r_B := \underline{(\nu_C = 1)\ \mathbf{?}\ 1\ \mathbf{:}\ 0}, \\
p_6: C \to 1C, & \nu := \nu + 1, \\
p_7: C \to 0C, & \nu := \nu, \\
p_8: C \to \$, & \nu := 0.
\end{array}$$

Let $G = (V, E)$ be a deterministic digraph encoded by its linearized adjacency matrix $\kappa(G)$, and let $n = |V|$ and $m = |\kappa(G)|$. Define a **DLOGTIME**-uniform family COPY^R of circuits such that the mth circuit of the family outputs the string

$$w^R(G) = \$\, \kappa^R(G)\, \left(\#\, \kappa^R(G)\right)^{m-1}\, \natural\, \nu(G)\, \$$$

where $\kappa^R(G)$ is the reverse encoding of G, the symbol \$ is used as string terminator, and ♮ is a new symbol (used instead of #) to distinguish the string encoding of the number of nodes from the adjacent matrices of G. Note that this circuit family is a simple modification of the above depth-1 family COPY: even in this case the mth circuit just computes m copies of $\kappa(G)$ (but here the wire connections are such that copies are obtained in reverse order), besides the leading string $\nu(G)$.

It is easy to see that, by parsing the input string $w^R(G)$ from right to left for membership in $\mathcal{L}(\text{DRRAG})$, we precisely mimic what we do for $\mathcal{L}(\text{DLRAG})$ on the string $w(G)$. That is, there is a path from 1 to n in G if, and only if, $w^R(G) \in \mathcal{L}(\text{DRRAG})$. Moreover, observe that DRRAG is in σ-$DRReg^A_\otimes$ as it is strict and makes only use of arithmetic expressions over $\{+\}$. Fig. 15 shows the deterministic automaton associated with its underlying regular grammar. □

Theorem 6.4. PARSE$_{[\sigma\text{-}Reg^A]}$ *and* PARSE$_{[Reg^A_\otimes]}$ *are* **NP**-*complete.*

Proof. (Membership) Let $\mathcal{AG}$ be an IRG in σ-Reg^A (resp., $Reg^A_\otimes$) and $TM(\mathcal{AG})$ be the non-deterministic Turing machine associated with it that recognizes $\mathcal{L}(\mathcal{AG})$. From Proposition 5.3, this machine ends after a polynomial number of calls to arithmetic expressions evaluations, which take at most polynomial-time, in their turn. Indeed, there is no trouble in numbers-size explosion here, because from Proposition 5.5 (resp., Proposition 5.6) all such arithmetic operations require only space $\mathcal{O}(n)$ on the work-tape of $TM(\mathcal{AG})$.

(Hardness) Recall that HAMILTON-PATH is the **NP**-complete problem defined as follows: Given a digraph G, decide whether there is a path that visits each node exactly once. In Section 6.1, we have described a σ-Reg^A IRG HPAG such that there is a path in G that visits each node exactly once if, and only if, its associated string $w(G) \in \mathcal{L}(\mathsf{HPAG})$, where $w(G) = v(G)\ (\#\ \kappa(G))^m\ \$$. Moreover, we have seen that such a string can be obtained by using the uniform family COPY of $\mathbf{NC}^0$ Boolean circuits. It follows that HAMILTON-PATH $\leqslant_m^{\mathbf{NC^0}} \mathcal{L}(\mathsf{HPAG})$.

Now, consider the IRG $\mathsf{HPAG}_\otimes$ obtained from HPAG by replacing each expression of the form $2 * x$ (where x is any attribute) with $x + x$. The new grammar $\mathsf{HPAG}_\otimes$ contains productions with attributes contributing more than once to the computation of attribute values, and thus it is not a strict grammar. However, the $*$ operator is not used in arithmetic expressions, and thus it belongs to $Reg_\otimes^A$, and it is clearly equivalent to HPAG. Therefore, HAMILTON-PATH $\leqslant_m^{\mathbf{NC^0}} \mathcal{L}(\mathsf{HPAG}_\otimes)$, too. □

Theorem 6.5. *PARSE$_{[\sigma\text{-}DLReg^A]}$ and PARSE$_{[\sigma\text{-}DRReg^A]}$, as well as PARSE$_{[DLReg_\otimes^A]}$ and PARSE$_{[DRReg_\otimes^A]}$ are **P**-complete.*

Proof. (Membership) Let $\mathcal{AG}$ be an IRG in σ-$DLReg^A$ or in σ-$DRReg^A$ (resp., in PARSE$_{[DLReg_\otimes^A]}$ or in PARSE$_{[DRReg_\otimes^A]}$), and $TM(\mathcal{AG})$ be the Turing machine associated with it that recognizes $\mathcal{L}(\mathcal{AG})$. Since the regular grammar underlying $\mathcal{AG}$ is deterministic, from Proposition 5.2 $TM(\mathcal{AG})$ is a deterministic machine, and from Proposition 5.3 it ends after a polynomial number of calls to arithmetic expressions evaluations, which take at most polynomial-time, in their turn. Indeed, as observed in the previous theorem, there is no trouble in numbers-size explosion here, because from Proposition 5.5 (resp., Proposition 5.6) all such arithmetic operations require only space $\mathcal{O}(n)$ on the work-tape of $TM(\mathcal{AG})$.

(Hardness) Recall that CIRCUIT-VALUE is the **P**-complete problem defined as follows: given a digraph $C = \langle V, E\rangle$ representing a variable-free Boolean circuit, decide whether the value (at the output gate) of circuit C is 1. In Section 6.1, we have described a σ-$DLReg^A$ IRG CVAG such that its associated string $w(C) = \$\ \kappa(C)\ \$$ belongs to $\mathcal{L}(\mathsf{CVAG})$ if, and only if, the value of circuit C is 1, where $\kappa(C)$ is the linearized matrix string that encodes C. Then, clearly enough, we get CIRCUIT-VALUE $\leqslant_m^{\mathbf{NC^0}} \mathcal{L}(\mathsf{CVAG})$. Indeed, the transformation from C's encoding to $w(C)$ is trivial, because we have to add just a symbol \$ at the beginning and at the end of the circuit encoding $\kappa(C)$. Moreover, as far as PARSE$_{[\sigma\text{-}DRReg^A]}$ is concerned, we can simply modify grammar CVAG to obtain an equivalent IRG in σ-$DRReg^A$ (see, for instance, Theorem 6.3).

Now, consider the IRG $\mathsf{CVAG}_\otimes$ obtained from CVAG by replacing each expression of the form $2 * x$ (where x is any attribute) with $x + x$. As observed for a similar case in the proof of the previous theorem, the new grammar $\mathsf{CVAG}_\otimes$ contains productions with attributes contributing more than once to the computation of attribute values, and thus it is not a strict grammar. However, the $*$ operator is not used in arithmetic expressions, and thus it belongs to $DLReg_\otimes^A$, and it is clearly equivalent to CVAG. Therefore, CIRCUIT-VALUE $\leqslant_m^{\mathbf{NC^0}} \mathcal{L}(\mathsf{CVAG}_\otimes)$, too. Finally, this hardness result holds even for $DRReg_\otimes^A$ grammars. Just apply the above modification to remove $*$ occurrences in arithmetic expressions to the **P**-hard σ-$DRReg^A$ IRG version of CVAG mentioned in the first part of this proof. □

7. Conclusions

We have analyzed the complexity of the languages generated by integer attribute grammars. Since attribute grammars can easily generate exponential-size attribute values, we have looked for tractable cases. We have proposed some non-severe restrictions on the grammars and their attributes that avoid the exponential explosion of attribute sizes, and that allow us to parse efficiently the languages generated by such grammars. Finally, we have shown that even in the most general case the parsing process remains in polynomial space.

In fact, many tractability results are given for (possibly non-deterministic) space-based classes. Thus, for the sake of completeness, we observe that the following good (time) complexities may easily be obtained in a realistic RAM model, by exploiting the properties described in this paper and state-of-the-art results about arithmetic operations[8]:

- σ-$DLReg_\otimes^A$ and σ-$DRReg_\otimes^A$ can be evaluated in time $\mathcal{O}(n \log n \log\log n \log\log\log n)$;
- σ-$DLReg^A$, σ-$DRReg^A$, $DLReg_\otimes^A$, and $DRReg_\otimes^A$ can be evaluated in time $\mathcal{O}(n^2 \log n \log\log n)$.

A thorough analysis of the most general cases without restrictions on operators and their compositions, or with non-deterministic grammars, will be a subject of future work.

[8] Given two n-digit numbers, both *addition* and *subtraction* can be done in linear time; *multiplication* can be done in time $\mathcal{O}(n \log n \log\log n)$ (see [52,18]) which is very close to the lower bound $\Omega(n \log n)$; *integer division* with truncation and *modulo reduction* have the same time complexity of multiplication, by the well-known *Newton–Raphson method*.

Acknowledgments

The authors would like to thank Eric Allender and Mikhail N. Vyalyi for their useful hints about arithmetic circuits and RAMs.

Appendix A. On languages and complexity classes

This section recalls the definitions of the complexity classes relevant to the present work (see, for instance, [23,34,44]) and related notions, and provides the notations used in the paper.

A.1. Turing machines and automata

Informally, a *Turing machine* is a basic device able to read from and write on semi-infinite tapes, whose contents may be locally accessed and changed in a computation. In particular, a *finite automaton* (or *finite-state machine*) may be considered like a very simple Turing machine able only to read from a finite tape. Since we heavily exploit these models of computation in the proofs, to make the paper self-contained and to set the formal notations we next briefly recall these notions.

A.1.1. Multi-tape deterministic Turing machines
Formally, a *multi-tape deterministic Turing machine* (*k-DTM*) is defined as a quadruple $M = \langle K, \Sigma, \delta, s_0\rangle$, where

- K is a finite set of *states*;
- Σ is the *alphabet* of M ($\Sigma \cap K = \emptyset$);
- $\delta : K \times (\Sigma \cup \{\sqcup\})^k \to (K \cup \{\mathsf{y}, \mathsf{n}\}) \times ((\Sigma \cup \{\sqcup\}) \times D)^k$ is the (total) *transition function* of M, where $k \geqslant 1$ is the number of tapes of M, $\sqcup$ is the *blank symbol*, $\{\mathsf{y}, \mathsf{n}\}$ are the *final states*, and $D = \{-1, 0, +1\}$ denotes *motion directions*;
- $s_0 \in K$ is the *initial state*.

A *configuration* of M is an element of $(K \cup \{\mathsf{y}, \mathsf{n}\}) \times ((\Sigma \cup \{\sqcup\})^* \times D)^k$ on which is defined the binary relation $\mapsto_M$. In particular, if M is in state q and a_i ($1 \leqslant i \leqslant k$) is the symbol in position $\imath_i$ under cursor i, then we say that the relation $(q, w_1, \imath_1, \ldots, w_k, \imath_k) \mapsto_M (q', w'_1, \imath'_1, \ldots, w'_k, \imath'_k)$ holds if

1. $\delta(q, a_1, \ldots, a_k) = (q', a'_1, d_1, \ldots, a'_k, d_k)$;
2. $d_i \neq -1$ whenever $\imath_i = 0$;
3. $\imath'_i = \imath_i + d_i$ and $d_i \in D$;
4. $w'_i[j] = w_i[j]$ for any $j \neq \imath_i$;
5. $w_i[\imath_i] = a_i$ and $w'_i[\imath_i] = a'_i$;
6. $a'_1 = a_1$, or equivalently $w'_1 = w_1$.

Let $\mapsto^*_M$ be the reflexive, transitive closure of $\mapsto_M$. The language generated by M is defined as:

$$\mathcal{L}(M) = \{w \in \Sigma^* : (s_0, w, 0, \varepsilon, 0, \ldots, \varepsilon, 0) \mapsto^*_M (\mathsf{y}, w, \imath_1, w_2, \imath_2, \ldots, w_k, \imath_k)\}.$$

A.1.2. Multi-tape non-deterministic Turing machines
Formally, a *multi-tape non-deterministic Turing machine* (*k-NTM*) is a quadruple $M = \langle K, \Sigma, \Delta, s_0\rangle$, where

- K, Σ and s_0 have been defined in *k-DTMs*.
- $\Delta \subseteq K \times (\Sigma \cup \{\sqcup\})^k \times (K \cup \{\mathsf{y}, \mathsf{n}\}) \times ((\Sigma \cup \{\sqcup\}) \times D)^k$ is the *transition relation* of M.

A *configuration* of M is an element of $(K \cup \{\mathsf{y}, \mathsf{n}\}) \times ((\Sigma \cup \{\sqcup\})^* \times D)^k$ on which is defined the binary relation $\mapsto_M$. In particular, if M is in state q and a_i ($1 \leqslant i \leqslant k$) is the symbol in position $\imath_i$ under cursor i, then the relation $(q, w_1, \imath_1, \ldots, w_k, \imath_k) \mapsto_M (q', w'_1, \imath'_1, \ldots, w'_k, \imath'_k)$ holds only if the conditions form 2 to 6 defined for (*k-DTM*)s hold, and $(q, a_1, \ldots, a_k, q', a'_1, d_1, \ldots, a'_k, d_k) \in \Delta$ or equivalently, $(q', a'_1, d_1, \ldots, a'_k, d_k) \in \Delta(q, a_1, \ldots, a_k)$. The definition of the language generated by M does not change.

A.2. Time and space complexity classes

Given a Turing machine M on alphabet Σ, and a string $w \in \Sigma^*$. Let $t_M(w)$ and $s_M(w)$ denote, respectively, the *time complexity* and the *space complexity* of M for the input w, i.e., the number of executed steps and the number of used tape cells to decide whether $w \in \mathcal{L}(M)$. Then, language $\mathcal{L}(M)$ is in:

- **L** if $s_M(w) \leqslant \log(p(\|w\|))$ for all $w \in \Sigma^*$;
- **P** if $t_M(w) \leqslant p(\|w\|)$ for all $w \in \Sigma^*$;

- **PSPACE** if $s_M(w) \leqslant p(\|w\|)$ for all $w \in \Sigma^*$;
- **EXP** if $t_M(w) \leqslant 2^{p(\|w\|)}$ for all $w \in \Sigma^*$;

where $p : \mathbb{N} \mapsto \mathbb{N}$ is a polynomial function.

Classes **L** and **P** have the non-deterministic counterparts **NL** and **NP** if M is non-deterministic. The known relationships among these classes are: $\mathbf{L} \subseteq \mathbf{NL} \subseteq \mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{EXP}$ and $\mathbf{P} \subset \mathbf{EXP}$. We shall henceforth consider also generalizations of the above classes, namely, complexity classes of the form $\mathbf{DTIME}(f(n))$ (deterministic time), $\mathbf{DSPACE}(f(n))$ (deterministic space), where f is any proper function. They contain exactly those languages that can be decided by a Turing machine M in such a way that for any input string w with $\|w\| = n$, $t_M(w)$ (or $s_M(w)$, respectively) is $\mathcal{O}(f(n))$. Their non-deterministic counterparts are $\mathbf{NTIME}(f(n))$ and $\mathbf{NSPACE}(f(n))$. For instance, we have that:

- $\mathbf{L} = \mathbf{DSPACE}(\log n)$;
- $\mathbf{P} = \mathbf{DTIME}(n^{\mathcal{O}(1)}) = \bigcup_{k>0} \mathbf{DTIME}(n^k)$;
- $\mathbf{PSPACE} = \mathbf{DSPACE}(n^{\mathcal{O}(1)}) = \bigcup_{k>0} \mathbf{DSPACE}(n^k)$;
- $\mathbf{EXP} = \mathbf{DTIME}(2^{n^{\mathcal{O}(1)}}) = \bigcup_{k>0} \mathbf{DTIME}(2^{n^k})$.

Some fundamental relationships [44] are just recalled below. Let $f(n)$ be a proper complexity function. Then,

- $\mathbf{DTIME}(f(n)) \subset \mathbf{DTIME}(f(n) \log^2 f(n))$;
- $\mathbf{DSPACE}(f(n)) \subset \mathbf{DSPACE}(f(n) \log f(n))$;
- $\mathbf{DSPACE}(f(n)) \subseteq \mathbf{NSPACE}(f(n))$;
- $\mathbf{DTIME}(f(n)) \subseteq \mathbf{NTIME}(f(n))$;
- $\mathbf{NTIME}(f(n)) \subseteq \mathbf{DSPACE}(f(n))$;
- $\mathbf{NSPACE}(f(n)) \subseteq \mathbf{DTIME}(k^{\log n + f(n)})$;
- $\mathbf{NSPACE}(f(n)) \subseteq \mathbf{DSPACE}([f(n)]^2)$;
- $\mathbf{DTIME}(f(n)) \subseteq \mathbf{DSPACE}(\frac{f(n)}{\log f(n)})$.

A.3. Circuit complexity

A *transducer circuit* is a finite directed acyclic graph[9] $C = \langle V, E \rangle$ with $n \geqslant 0$ *input nodes* (each of in-degree 0) labeled $x_1, \ldots, x_n$, and $m \geqslant 1$ *output nodes* (of out-degree 0). If $m = 1$, we say that C is a *Boolean circuit*, because it computes a *Boolean function*. The edges of the graph are called *wires*. Every non-input node v is called a *gate*, and has an associated *gate function* $g_v : \{0,1\}^r \to \{0,1\}$, that takes as many arguments as there are wires coming into v (r is the in-degree of v). The value of the gate is transmitted along each wire that goes out of v. The *size* of C is the number of its nodes, while the *depth* is the length of the longest path in the circuit. *Fan-in* (*fan-out*) of C is the maximum in-degree (out-degree) of any gate in C. Since C is cycle-free, then every Boolean assignment $x \in \{0,1\}^n$ of values to the input nodes determines a unique value for each gate and wire, and the value of the output gates is the *output* (or *value*) $C(x)$ of the circuit. When C is a Boolean circuit, we say that C *accepts* x if, and only if, $C(x) = 1$.

A *variable-free circuit* is a circuit C where the n input nodes have a fixed Boolean assignment $x \in \{0,1\}^n$.

A *circuit family* $\{C_n\}$ consists of a sequence of circuits $\{C_1, C_2, \ldots, C_s\}$, where each C_n has n input nodes. The language accepted by a family $\{C_n\}$ of Boolean circuits is

$$\mathcal{L}(\{C_n\}) = \{x \mid x \in \{0,1\}^+,\ C_{|x|}(x) = 1\}.$$

The *size complexity* of family $\{C_n\}$ is the function $z(n)$ giving the number of nodes in C_n. The *depth complexity* is the function $d(n)$ giving the depth of C_n. Family $\{C_n\}$ is said to be *uniform* if circuit C_n can be constructed for any input size n. In particular, $\{C_n\}$ is called **P**-uniform (respectively, **L**-uniform) if circuit C_n can be constructed from the string 1^n in polynomial time (resp., $\mathcal{O}(\log n)$ space). For reductions among problems belonging to very low complexity classes (below **L**), **DLOGTIME**-uniform circuit families are typically employed. For such families, the circuit member C_n should be recognized in **DLOGTIME**, that is, there exists a deterministic Turing machine $T_{\{C_n\}}$ that decides in $\mathcal{O}(\log n)$ time the language

$$\mathcal{L}(\{C_n\}) = \{\langle c, i, j, n \rangle \mid C_n[i] = c \text{ and } j \text{ is an input for gate } C_n[i],\ n \geqslant i > 0\}.$$

Intuitively, the Turing machine has to answer in deterministic logarithmic time the question “Does node i in the circuit C_n have gate-type c and node (or wire) j among its inputs?” Note that, as in the other above mentioned uniform classes of circuits, the cost is measured w.r.t. n, which is the size of string 1^n. If the input of $T_{\{C_n\}}$ only contains the (standard) binary encodings of numbers i, j, and n (the gate-type c is a constant), then the time-complexity of $T_{\{C_n\}}$ is in fact linear with respect to its actual input size.

[9] Since C is acyclic, then all edges can be assumed of the form (i, j), where $i < j$.

In the following, unless otherwise stated, "uniform" means **DLOGTIME**-uniform. Given two complexity functions $z(n)$ and $d(n)$, two circuit complexity classes are defined:

- class $\mathbf{SIZE}(z(n))$ includes all languages accepted by **DLOGTIME**-uniform bounded fan-in circuit families whose size complexity is at most $z(n)$;
- class $\mathbf{DEPTH}(d(n))$ includes all languages accepted by **DLOGTIME**-uniform bounded fan-in circuit families whose depth complexity is at most $d(n)$.

For all $k \geqslant 0$, the complexity class $\mathbf{NC}^k$ [3] is the class of languages accepted by **DLOGTIME**-uniform bounded fan-in circuit families of polynomial size and $\mathcal{O}(\log^k n)$ depth. In other words, $\mathbf{NC}^k \subseteq \mathbf{SIZE}(n^{\mathcal{O}(1)}) \cap \mathbf{DEPTH}(\mathcal{O}(\log^k n))$. In particular, the case $k = 0$ gives constant-depth circuit families. In general, $\mathbf{NC}^0$ is not studied as a language class since the output gate can only depend on a constant number of input bits (such circuits cannot even compute the logical OR of n input bits) but it is interesting as a function class.

If $d(n) \geqslant \log n$, it is known that if a circuit C is in $\mathbf{DEPTH}(d(n))$ then it belongs to $\mathbf{DSPACE}(d(n))$, too [5]. This proves, for instance, that $\mathbf{NC}^1 \subseteq \mathbf{L}$. We also recall that the following chain of inclusions holds [56]:

$$\mathbf{NC}^0 \subset \mathbf{NC}^1 \subseteq \mathbf{L} \subseteq \mathbf{NL} \subseteq \mathbf{NC}^2.$$

The parallel complexity of basic arithmetic operations has been closely investigated since the 1960's. Typically, the input size is measured in terms of the binary encoding of the integer at hands. In particular, it is well known [58] that, given two integers x and y whose binary notations require at most n bits each, then the sum $x + y$, the difference $x - y$, and the product $x * y$ can be done in **L**-uniform $\mathbf{NC}^1$, i.e. by logspace computable Boolean circuit families of $\mathcal{O}(\log n)$ depth and with $n^{\mathcal{O}(1)}$ Boolean gates. Only recently, the same result was found also for nonnegative integer division [7].

Concerning %, observe that $x\%y$ is nothing else but $x - (x \div y) * y$, and that by composing the corresponding circuits we remain in $\mathbf{NC}^1$.

Finally, a recent result on Integer Division [22] shows that this problem is in the class **DLOGTIME**-uniform $\mathbf{TC}^0$, which is a subclass of **DLOGTIME**-uniform $\mathbf{NC}^1$.

In particular, from these results it follows that all the above mentioned integer operators may be evaluated in (deterministic) logspace, which is what we need in this paper.

A.4. Relationships between languages and complexity classes

Let **REG**, **DCFL**, **CFL** and **CSL** be respectively the complexity classes of all the *regular languages*, *deterministic context-free languages*,[10] *context-free languages* and *context-sensitive languages*.

Fig. 16 shows a synthesis of the main complexity results about these classes.

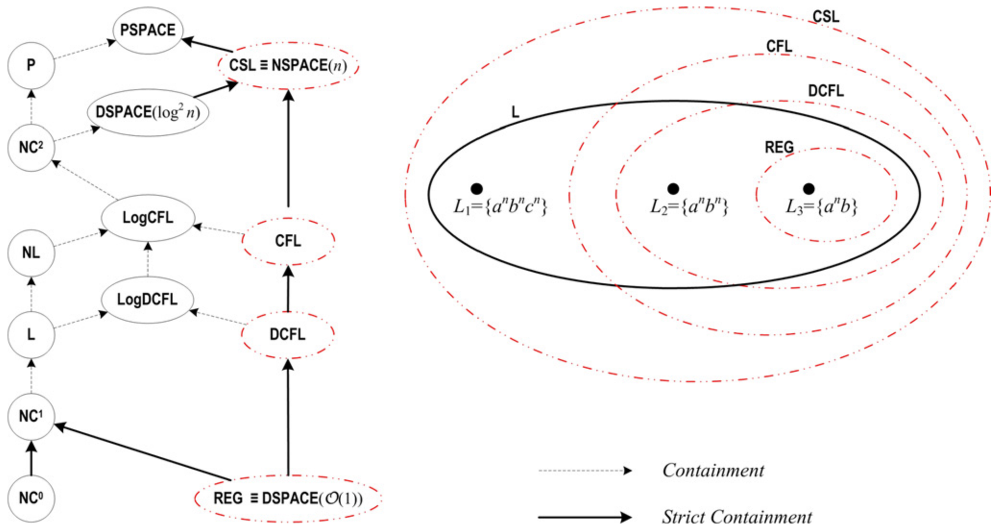


Fig. 16. Language complexity diagram.

[10] A context-free grammar is said to be *deterministic* (*DCFG*), if it can be parsed by a *deterministic pushdown automaton* [37].

In particular:

- $\mathbf{REG} \equiv \mathbf{DSPACE}(\mathcal{O}(1))$ [46,53];
- $\mathbf{CSL} \equiv \mathbf{NSPACE}(n)$ [32];
- $\mathbf{CFL} \subseteq \mathbf{DSPACE}(\log^2 n)$ [36];
- $\mathbf{REG} \subset \mathbf{DCFL} \subset \mathbf{CFL} \subset \mathbf{CSL}$ by the well-known *Chomsky Hierarchy* [19,6];
- $\mathbf{L} \subseteq \mathbf{LOGDCFL} \subseteq \mathbf{LOGCFL}$ and $\mathbf{NL} \subseteq \mathbf{LOGCFL}$ and $\mathbf{DCFL} \subseteq \mathbf{LOGDCFL}$ and $\mathbf{CFL} \subseteq \mathbf{LOGCFL}$. Moreover there is a **DCFL** language being log-tape complete for the family **LOGDCFL** [56], and a **CFL** language, called *Greibach's hardest context-free language* [20], complete for **LOGCFL**;
- $\mathbf{CFL} \subseteq \mathbf{NC}^2$ [50];
- $\mathbf{LOGCFL} \subseteq \mathbf{NC}^2$ [51];
- Every regular language is computable by $\mathbf{NC}^1$ circuits of linear size, namely $\mathbf{REG} \subseteq \mathbf{NC}^1$, and there are regular languages being complete for the class $\mathbf{NC}^1$ as well [4];
- Some other subclasses of **CFL** are in $\mathbf{NC}^1$ [24]. Then $\mathbf{REG} \subset \mathbf{NC}^1$.

A.5. Random access machines

As defined in [21], a powerful RAM acceptor (we just say RAM, hereafter) is a (finite) set of registers $R_0, R_1, \ldots, R_s$ each of which is capable of storing a nonnegative integer in binary representation, together with a finite *program* of (possibly labeled) *instructions* chosen from the following set:

$R_i \leftarrow R_j\ (=k)$	(assignment)
$R_i \leftarrow R_{R_j}$	(indirect addressing)
$R_i \leftarrow R_j + R_k$	(sum)
$R_i \leftarrow R_j \dot{-} R_k$	(proper subtraction)
$R_i \leftarrow R_j * R_k$	(product)
$R_i \leftarrow R_j \div R_k$	(integer division)
$R_i \leftarrow R_j$ bool R_k	(Boolean operations)
if R_i comp R_j label$_1$ **else** label$_2$	(conditional jump)
accept	
reject	

We recall that *proper subtraction* is the operation from $\mathbb{N} \times \mathbb{N}$ to $\mathbb{N}$ defined by $a \dot{-} b = \max\{0, a - b\}$. If no two labels are the same, we say that the program is deterministic, otherwise it is non-deterministic. We call a RAM model deterministic if we consider only deterministic programs from the instruction set. Moreover:

- comp is a comparison operator from $\{<, \leqslant, =, >, \geqslant, \neq\}$;
- bool may be any binary Boolean operation, e.g. $\wedge, \vee$, *eor*, *nand*, etc.

The computation of a RAM starts by putting the input in register R_0, setting all registers to 0, and executing the first instruction of the RAM program. Instructions are executed in sequence until a conditional jump is encountered, then one of the instructions with label *label*$_1$ is executed if the condition is satisfied, and one of the instructions with label *label*$_2$ is executed otherwise. Execution stops when either an **accept** or a **reject** instruction is met. A string $x \in \{0, 1\}^*$ is accepted by the RAM is there is a finite computation ending with the execution of an **accept** instruction. The complexity measures defined for RAMs are:

The class **PTIME-RAM** (resp., **NPTIME-RAM**) is the class of languages L for which there is a deterministic (resp., non-deterministic) machine that accepts L within a polynomial number of steps. The following remarkable result by Hartmanis and Simon [21] states that such polynomial-time powerful RAMs in fact characterize the class polynomial space.

Proposition A.1. ***PSPACE* $\equiv$ *PTIME-RAM* $\equiv$ *NPTIME-RAM*.**

References

[1] H. Alblas, B. Melichar (Eds.), SAGA'91: Proceedings of the International Summer School on Attribute Grammars, Applications and Systems, Prague, Czechoslovakia, June 4–13, 1991, Lecture Notes in Comput. Sci., vol. 545, Springer, Berlin, Heidelberg, 1991.

[2] Z. Alexin, T. Gyimóthy, T. Horváth, K. Fábricz, Attribute grammar specification for a natural language understanding interface, in: Proceedings of the International Conference WAGA on Attribute Grammars and Their Applications, Paris, France, September 19–21, 1990, in: Lecture Notes in Comput. Sci., vol. 461, Springer, Berlin, Heidelberg, 1990, pp. 313–326.

[3] E. Allender, M.C. Loui, K.W. Regan, Complexity Classes, CRC Press, 1998, pp. 27.1–27.23.

[4] D.A. Barrington, Bounded-width polynomial-size branching programs recognize exactly those languages in $\mathbf{NC}^1$, in: STOC'86 – Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing, Berkeley, CA, United States, May 28–30, 1986, ACM, New York, NY, USA, 1986, pp. 1–5.

[5] A. Borodin, On relating time and space to size and depth, SIAM J. Comput. 6 (4) (1977) 733–744.
[6] F.-J. Brandenburg, On one-way auxiliary pushdown automata, in: Proceedings of the 3rd GI Conference Darmstadt on Theoretical Computer Science, March 28–30, 1977, in: Lecture Notes in Comput. Sci., vol. 48, Springer, Berlin, Heidelberg, 1977, pp. 132–144.
[7] A. Chiu, G. Davida, B. Litow, Division in logspace-uniform $\mathbf{NC}^1$, RAIRO Theor. Inform. Appl. 35 (3) (2001) 259–275.
[8] P. Deransart, M. Jourdan (Eds.), WAGA'90: Proceedings of the International Conference on Attribute Grammars and Their Applications, Paris, France, September 19–21, 1990, Lecture Notes in Comput. Sci., vol. 461, Springer, Berlin, Heidelberg, 1990.
[9] P. Deransart, M. Jourdan, B. Lorho, Attribute Grammars – Definitions, Systems and Bibliography, Lecture Notes in Comput. Sci., vol. 323, Springer, Berlin, Heidelberg, 1988.
[10] P. Deransart, J. Maluszynski, Relating logic programs and attribute grammars, J. Log. Program. 2 (2) (1985) 119–155.
[11] P. Deransart, J. Maluszynski, A Grammatical View of Logic Programming, MIT Press, Cambridge, MA, USA, 1993.
[12] S. Efremidis, C.H. Papadimitriou, M. Sideri, Complexity characterizations of attribute grammar languages, Inform. and Comput. 78 (3) (1988) 178–186.
[13] L. Eikvil, Information Extraction from World Wide Web – A Survey, Tech. Rep. 945, Norwegian Computing Center, 1999.
[14] I. Fang, FOLDS – A declarative semantic formal language definition system, Tech. Rep. STAN-CS-72-239, Computer Science Department, Stanford University, Stanford, Calif., 1972.
[15] R. Farrow, LINGUIST-86: Yet another translator writing system based on attribute grammars, in: SIGPLAN'82 – Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction, Boston, MA, United States, June 23–25, 1982, ACM, New York, NY, USA, 1982, pp. 160–171.
[16] R. Feldman, Y. Aumann, M. Finkelstein-Landau, E. Hurvitz, Y. Regev, A. Yaroshevich, A comparative study of information extraction strategies, in: Proceedings of the Third International Conference, CICLing 2002 on Computational Linguistics and Intelligent Text Processing, Mexico City, Mexico, February 17–23, 2002, in: Lecture Notes in Comput. Sci., vol. 2276, Springer, Berlin, Heidelberg, 2002, pp. 21–34.
[17] R. Feldman, B. Rosenfeld, M. Fresko, TEG—a hybrid approach to information extraction, Knowl. Inf. Syst. 9 (1) (2006) 1–18.
[18] M. Furer, Faster integer multiplication, SIAM J. Comput. 39 (3) (2009) 979–1005.
[19] S. Ginsburg, S.A. Greibach, Deterministic context free languages, in: SWCT'65 – Conference Record of the 6th Annual Symposium on Switching Circuit Theory and Logical Design, 1965, IEEE Computer Society, Los Alamitos, CA, USA, 1965, pp. 203–220.
[20] S.A. Greibach, The hardest context-free language, SIAM J. Comput. 2 (4) (1973) 304–310.
[21] J. Hartmanis, J. Simon, On the power of multiplication in random access machines, in: SWAT'74 – Proceedings of the 15th IEEE Annual Symposium on Switching and Automata Theory, University of New Orleans, USA, 14–16 October, 1974, IEEE Computer Society, Los Alamitos, CA, USA, 1974, pp. 13–23.
[22] W. Hesse, E. Allender, D.A.M. Barrington, Uniform constant-depth threshold circuits for division and iterated multiplication, J. Comput. Syst. Sci. 65 (4) (2002) 695–716.
[23] J.E. Hopcroft, R. Motwani, J.D. Ullman, Introduction to Automata Theory, Languages, and Computation, 3/E, Addison–Wesley, 2007.
[24] O.H. Ibarra, T. Jiang, B. Ravikumar, J.H. Chang, On some languages in $\mathbf{NC}^1$ (Extended abstract), in: VLSI Algorithms and Architectures (AWOC'88) – Proceedings of the 3rd Aegean Workshop on Computing, Corfu, Greece, June 28–July 1, 1988, in: Lecture Notes in Comput. Sci., vol. 319, Springer, Berlin, Heidelberg, 1988, pp. 64–73.
[25] U. Kastens, B. Hutt, E. Zimmermann, GAG: A Practical Compiler Generator, Lecture Notes in Comput. Sci., vol. 141, Springer, Berlin, Heidelberg, 1982.
[26] K. Kennedy, S.K. Warren, Automatic generation of efficient evaluators for attribute grammars, in: POPL'76 – Proceedings of the 3rd ACM SIGACT-SIGPLAN Symposium on Principles on Programming Languages, Atlanta, Georgia, January 19–21, 1976, ACM, New York, NY, USA, 1976, pp. 32–49.
[27] D.E. Knuth, Semantics of context-free languages, Theory Comput. Syst. 2 (2) (1968) 127–145.
[28] D.E. Knuth, The genesis of attribute grammars, in: WAGA'90 – Proceedings of the International Conference on Attribute Grammars and Their Applications, Paris, France, September 19–21, 1990, in: Lecture Notes in Comput. Sci., vol. 461, Springer, Berlin, Heidelberg, 1990, pp. 1–12.
[29] C. Koch, S. Scherzinger, Attribute grammars for scalable query processing on XML streams, VLDB J. 16 (3) (2007) 317–342.
[30] K. Koskimies, O. Nurmi, J. Pakki, The design of a language processor generator, Software: Practice and Experience 18 (2) (1988) 107–135.
[31] S. Kuhlins, R. Tredwell, Toolkits for generating wrappers – A survey of software toolkits for automated data extraction from web sites, in: Objects, Components, Architectures, Services, and Applications for a Networked World – International Conference NetObjectDays, NODe 2002 Erfurt, Revised Papers, Germany, October 7–10, 2002, in: Lecture Notes in Comput. Sci., vol. 2591, Springer, Berlin, Heidelberg, 2003, pp. 184–198.
[32] S.-Y. Kuroda, Classes of languages and linear-bounded automata, Inform. and Control 7 (2) (1964) 207–223.
[33] A.H.F. Laender, B.A. Ribeiro-Neto, A.S. da Silva, J.S. Teixeira, A brief survey of web data extraction tools, SIGMOD Rec. 31 (2) (2002) 84–93.
[34] H.R. Lewis, C.H. Papadimitriou, Elements of the Theory of Computation, Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.
[35] P.M. Lewis, D.J. Rosenkrantz, R.E. Stearns, Attributed translations, J. Comput. System Sci. 9 (3) (1974) 279–307.
[36] P.M. Lewis II, R.E. Stearns, J. Hartmanis, Memory bounds for recognition of context-free and context-sensitive languages, in: SWCT'65 – Conference Record of the 6th Annual Symposium on Switching Circuit Theory and Logical Design, 1965, IEEE Computer Society, Los Alamitos, CA, USA, 1965, pp. 191–202.
[37] S.J. Løvborg, Declarative Programming and Natural Language, http://www2.imm.dtu.dk/pubdb/p.php?5388, 2007.
[38] A.A. Muchnik, One application of real-valued interpretation of formal power series, Theoret. Comput. Sci. 290 (3) (2003) 1931–1946.
[39] F. Neven, Extensions of attribute grammars for structured document queries, in: Research Issues in Structured and Semistructured Database Programming (DBPL'99) – Revised Papers of the 7th International Workshop on Database Programming Languages, Kinloch Rannoch, UK, September 1–3, 1999, in: Lecture Notes in Comput. Sci., vol. 1949, Springer, Berlin, Heidelberg, 2000, pp. 99–117.
[40] F. Neven, Attribute grammars for unranked trees as a query language for structured documents, J. Comput. System Sci. 70 (2) (2005) 221–257.
[41] F. Neven, J. Van Den Bussche, Expressiveness of structured document query languages based on attribute grammars, J. ACM 49 (1) (2002) 56–100.
[42] T.J. Ostrand, M.C. Paull, E.J. Weyuker, Parsing regular grammars with finite lookahead, Acta Inform. 16 (2) (1981) 125–138.
[43] J. Paakki, Attribute grammar paradigms—a high-level methodology in language implementation, ACM Comput. Surv. 27 (2) (1995) 196–255.
[44] C.M. Papadimitriou, Computational Complexity, Addison–Wesley, Reading, MA, 1994.
[45] Y. Papakonstantinou, V. Vianu, DTD inference for views of XML data, in: PODS'00 – Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, Dallas, TX, United States, May 14–19, 2000, ACM, New York, NY, USA, 2000, pp. 35–46.
[46] M.O. Rabin, Two-way finite automata, in: Proceedings of the Summer Institute of Symbolic Logic, Cornell University, 1957, Communications Research Division, Institute for Defense Analyses, Princeton, NJ, 1960, pp. 366–369.
[47] D. Ridjanovic, M.L. Brodie, Defining database dynamics with attribute grammars, Inform. Process. Lett. 14 (3) (1982) 132–138.
[48] M. Ruffolo, M. Manna, HιLεX: A system for semantic information extraction from web documents, in: Enterprise Information Systems, 8th International Conference, ICEIS 2006, Revised Selected Papers, Paphos, Cyprus, May 23–27, 2006, in: Lecture Notes in Business Information Processing, vol. 3, Springer, Berlin, Heidelberg, 2008, pp. 194–209.
[49] M. Ruffolo, M. Manna, L. Gallucci, N. Leone, D. Saccà, A logic-based tool for semantic information extraction, in: Proceedings of the 10th European Conference, JELIA 2006 on Logics in Artificial Intelligence, Liverpool, UK, September 13–15, 2006, in: Lecture Notes in Comput. Sci., vol. 4160, Springer, Berlin, Heidelberg, 2006, pp. 506–510.
[50] W.L. Ruzzo, On uniform circuit complexity, in: FOCS'79 – Proceedings of the 20th IEEE Annual Symposium on Foundations of Computer Science, October 29–31, 1979, IEEE Computer Society, Los Alamitos, CA, USA, 1979, pp. 312–318.
[51] W.L. Ruzzo, Tree-size bounded alternation, J. Comput. Syst. Sci. 21 (2) (1980) 218–235.

[52] A. Schönhage, V. Strassen, Schnelle Multiplikation großer Zahlen, Computing 7 (1971) 281–292.
[53] J.C. Shepherdson, The reduction of two-way automata to one-way automata, IBM J. Research and Development 3 (2) (1959) 198–200.
[54] D.A. Simovici, R. Tenney, Theory of Formal Languages with Applications, World Scientific, Singapore, 1999.
[55] J. Stenback, A. Heninger, Document Object Model (DOM) Level 3 Load and Save Specification, W3C Recommendation, http://www.w3.org/TR/DOM-Level-3-LS/, April 2004.
[56] I.H. Sudborough, On the tape complexity of deterministic context-free languages, J. ACM 25 (3) (1978) 405–414.
[57] P. Trahanias, E. Skordalakis, Syntactic pattern recognition of the ECG, IEEE Trans. Pattern Analysis and Machine Intelligence 12 (7) (1990) 648–657.
[58] I. Wegener, The Complexity of Boolean Functions, John Wiley & Sons, Inc., New York, NY, USA, 1987.