Precedence Automata and Languages

Violetta Lonati[1], Dino Mandrioli[2], Matteo Pradella[3]

[1] DSI - Università degli Studi di Milano, via Comelico 39/41, Milano, Italy
lonati@dsi.unimi.it
[2] DEI - Politecnico di Milano, via Ponzio 34/5, Milano, Italy
dino.mandrioli@polimi.it
[3] IEIIT - Consiglio Nazionale delle Ricerche, via Golgi 42, Milano, Italy
matteo.pradella@polimi.it

Abstract. Operator precedence grammars define a classical Boolean and deterministic context-free family (called Floyd languages or FLs). FLs have been shown to strictly include the well-known visibly pushdown languages, and enjoy the same nice closure properties. We introduce here Floyd automata, an equivalent operational formalism for defining FLs. This also permits to extend the class to deal with infinite strings to perform for instance model checking.

Keywords: Operator precedence languages, Deterministic Context-Free languages, Omega languages, Pushdown automata.

1 Introduction

The history of formal language theory has always paired two main and complementary formalisms to define and process –not only formal– languages: grammars or syntaxes and abstract machines or automata. The power and the complementary benefits of these two formalisms are so evident and well-known that it is certainly superfluous to remind them here. Also universally known are the conceptual relevance and practical impact of the family of context-free languages and the corresponding grammars paired with pushdown automata.

Among the many subfamilies that have been introduced throughout the last decades with various goals, operator precedence grammars, herewith renamed Floyd grammars (FGs) in honor of their inventor [9], represent a pioneering model mainly aimed at deterministic –and therefore efficient– parsing. Visibly pushdown languages (VPLs) are a much more recent subfamily of (deterministic) context-free languages introduced in the seminal paper [1] with the goal of extending the typical closure properties of regular languages to larger families of languages accepted by infinite-state machines; a major practical result is the possibility of extending such powerful verification technique as model checking beyond the scope of finite state machines. Along the usual tradition, VPLs have been characterized both in terms of abstract machines, the visibly pushdown automata (VPAs), and by means of a suitable subclass of context-free grammars.

Rather surprisingly, instead, investigation of the basic –and nice, indeed– properties of FGs has been suspended, probably as a consequence of the advent of other, more general, parsing techniques, such as LR parsing [10]. Although FGs generate obviously a subclass of deterministic CF languages and therefore can be parsed by any deterministic pushdown machine, typically a shift-reduce one [10], we are not aware of a family of automata that perfectly matches the generative power of this class of grammars. On the other hand, operator precedence parsers are still used today, thanks to their elegant simplicity and efficiency. For instance, they are present in Parrot, Perl 6's virtual machine, as part of the Parser Grammar Engine (PGE); in GCC's C and C++ hand-coded parsers, for managing arithmetic expressions.[4]

Quite recently we realized strong relations between these two seemingly unrelated families of languages; precisely we showed that: VPLs are a proper subclass of languages defined by FGs (i.e. Floyd Languages, or FLs in short), and coincide with those languages that can be generated by FGs characterized by a well precise shape of operator precedence matrix (OPM). The inclusion relation is effective in that a FG can be algorithmically derived form a VPA and conversely a VPA can be obtained by a FG whose OPM satisfies the restriction [5].

FLs enjoy all typical closure properties of regular languages that motivated the study of VPLs and other related families [3,12,4]. Precisely, closure w.r.t. Boolean operations was proved a long time ago in [7], whereas closure under concatenation, Kleene star, and other typical algebraic operations has been investigated only recently under the novel interest ignited by the above remark [6]. Thus, the old-fashioned FLs turned out to be the largest known class of deterministic context-free languages that enjoy closure under all traditional language operations. Another reason why, in our opinion, FLs are far from obsolete and uninteresting in these days is that, unlike most other deterministic languages of practical use, they can be parsed not necessarily left-to-right, thus offering interesting opportunities, e.g., to exploit parallelism and incrementality [10].

In this paper we provide another missing tile of the "old and new puzzle", namely we introduce a novel class of stack-based automata perfectly carved on the generation mechanism of FGs, which too we name in honor of Robert Floyd. Not surprisingly they inherit some features of VPAs (mainly a clear separation between push and pop operations) and maintain some typical behavior of shift-reduce parsing algorithms; however, they also exhibit some distinguishing features and imply some non-trivial technicalities to derive them automatically from FGs and conversely.

The availability of a precise family of automata allows to apply to FLs the now familiar ω-extension –a further extension of Kleene $*$ operation–, i.e., the definition of languages of infinite strings and the various criteria for their acceptance or rejection by recognizing devices. ω-languages are now more and more important to deal with never-ending computations such as operating systems, web-services, embedded applications, etc. Thus, we also introduce the ω-version of FLs and we show their potential in terms of modeling the behavior of some realistic systems.

The paper is structured as follows: Section 2 recalls basic definitions on Floyd's grammars; Section 3 introduces Floyd automata (FAs) and shows that, as well as FSMs and VPAs, but unlike pushdown automata, their deterministic version is not less powerful than the nondeterministic counterpart; Section 4 provides effective constructions to derive a FA from a FG and conversely; Section 5 extends the definition of FLs to sets of infinite strings by applying to FAs the well-known concepts of ω-behavior and acceptance; finally Section 6 draws some conclusions.

[4] The interested reader may find more information at *http://gcc.gnu.org*, and *http://www.parrot.org*, respectively.

2 Preliminaries

Let Σ be an alphabet. The empty string is denoted ε. A *context-free* (CF) grammar is a 4-tuple $G = (N, \Sigma, P, S)$, where N is the nonterminal alphabet, P the rule (or production) set, and S the axiom. An *empty rule* has ε as the right hand side (r.h.s.). A *renaming rule* has one nonterminal as r.h.s. A grammar is *reduced* if every rule can be used to generate some string in Σ^*. It is *invertible* if no two rules have identical r.h.s.

The following naming convention will be adopted, unless otherwise specified: lowercase Latin letters $a, b, \ldots$ denote terminal characters; uppercase Latin letters $A, B, \ldots$ denote nonterminal characters; letters $u, v, \ldots$ denote terminal strings; and Greek letters $\alpha, \ldots, \omega$ denote strings over $\Sigma \cup N$. The strings may be empty, unless stated otherwise.

A rule is in *operator form* if its r.h.s has no adjacent nonterminals; an *operator grammar* (OG) contains just such rules. Any CF grammar admits an equivalent OG, which can be also assumed to be invertible [11,13].

The coming definitions for operator precedence grammars [9], here renamed *Floyd Grammars* (FG), are from [7]. We refer the reader unfamiliar with precedence grammars and parsing techniques to [10], that contains an easily readable, practical description of FGs.

For an OG G and a nonterminal A, the *left and right terminal sets* are

$$\mathcal{L}_G(A) = \{a \in \Sigma \mid A \overset{*}{\Rightarrow} Ba\alpha\} \qquad \mathcal{R}_G(A) = \{a \in \Sigma \mid A \overset{*}{\Rightarrow} \alpha a B\}$$

where $B \in N \cup \{\varepsilon\}$ and $\Rightarrow$ denotes the derivation relation. The grammar name G will be omitted unless necessary to prevent confusion.

R. Floyd took inspiration from the traditional notion of precedence between arithmetic operators in order to define a broad class of languages, such that the shape of the derivation tree is solely determined by a binary relation between terminals that are consecutive, or become consecutive after a bottom-up reduction step.

For an OG G, let α, β range over $(N \cup \Sigma)^*$ and $a, b \in \Sigma$. Three binary operator precedence (OP) relations are defined:

$$\begin{aligned} \text{equal in precedence:} \quad & a \doteq b \iff \exists A \to \alpha a B b \beta, B \in N \cup \{\varepsilon\} \\ \text{takes precedence:} \quad & a \gtrdot b \iff \exists A \to \alpha D b \beta, D \in N \text{ and } a \in \mathcal{R}_G(D) \\ \text{yields precedence:} \quad & a \lessdot b \iff \exists A \to \alpha a D \beta, D \in N \text{ and } b \in \mathcal{L}_G(D) \end{aligned} \tag{1}$$

For an OG G, the *operator precedence matrix* (OPM) $M = OPM(G)$ is a $|\Sigma| \times |\Sigma|$ array that with each ordered pair (a, b) associates the set M_{ab} of OP relations holding between a and b.

Definition 1. *G is an* operator precedence *or* Floyd grammar *(FG) if, and only if, $M = OPM(G)$ is a conflict-free matrix, i.e., $\forall a, b$, $|M_{ab}| \leq 1$.*

Example 1. Arithmetic expressions with prioritized operators, a classical construct, are presented in a simple variant without parentheses, together with its OPM.

$$\begin{aligned} & S \to E, \\ & E \to E + T \mid T \times a \mid a, \\ & T \to T \times a \mid a \end{aligned} \qquad \begin{array}{c|ccc} & a & + & \times \\ \hline a & & \gtrdot & \gtrdot \\ + & \lessdot & \gtrdot & \lessdot \\ \times & \doteq & & \end{array}$$

The equal in precedence relations of a FG alphabet are connected with an important parameter of the grammar, namely the length of the right hand sides of the rules. Clearly, a rule $A \to A_1 a_1 \dots A_t a_t A_{t+1}$, where each A_i is a possibly missing nonterminal, is associated with relations $a_1 \doteq a_2 \doteq \dots \doteq a_t$. If the $\doteq$ relation is cyclic, there is no finite bound on the length of the r.h.s of a production. Otherwise the length is bounded by $2 \cdot c + 1$, where $c \geq 1$ is the length of the longest $\doteq$-chain. In this paper, for the sake of simplicity and brevity we assume that all precedence matrices are $\doteq$-cycle free. In the case of FGs this prevents the risk of r.h.s of unbounded length [7], in the case of FAs we will see that it avoids a priori the risk of an unbounded sequence of push operations onto the stack matched by only one pop operation. The hypothesis of $\doteq$-cycle freedom could be replaced by weaker ones, such as a bound on r.h.s, as it happens with FGs, at the price of heavier notation, constructions, and proofs.

Definition 2. *A FG is in Fischer normal form [8] if it is invertible, the axiom S does not occur in the r.h.s. of any rule, no empty rule exists except possibly $S \to \varepsilon$, the other rules having S as l.h.s are renaming, and no other renaming rules exist.*

OPMs play a fundamental role in deterministic parsing of FGs. Thus in the view of defining automata to parse FLs we pair them with the alphabet somewhat mimicking VPL's approach where the terminal alphabet is partitioned into calls, returns, and internals [2]. To this goal, we use a special symbol # not in Σ to mark the beginning and the end of any string. This is consistent with the typical operator parsing technique that requires the lookback and lookahead of one character to determine the precedence relation [10]. The precedence relation in the OPM are extended to include # in the normal way.

Definition 3. *An* operator precedence alphabet *is a pair (Σ, M) where Σ is an alphabet and M is a conflict-free operator precedence matrix, i.e. a $|\Sigma \cup \{\#\}|^2$ array that with each ordered pair (a, b) associates at most one of the operator precedence relations: $\doteq$, $\lessdot$ or $\gtrdot$.*

For $u, v \in \Sigma^*$ we write $u \lessdot v$ if $u = xa$ and $v = by$ with $a \lessdot b$. Similarly for the other precedence relations.

3 Floyd automata

Definition 4. *A nondeterministic* precedence automaton *(or Floyd automaton) is given by a tuple: $\mathcal{A} = \langle \Sigma, M, Q, I, F, \delta \rangle$ where:*

- *(Σ, M) is a precedence alphabet,*
- *Q is a set of states (disjoint from Σ),*
- *$I \subseteq Q$ is a set of initial states,*
- *$F \subseteq Q$ is a set of final states,*
- *$\delta : Q \times (\Sigma \cup Q) \to 2^Q$ is the transition function.*

The transition function can be seen as the union of two disjoint functions:

$$\delta_{\text{push}} : Q \times \Sigma \to 2^Q \qquad \delta_{\text{flush}} : Q \times Q \to 2^Q$$

A nondeterministic precedence automaton can be represented by a graph with Q as the set of vertices and $\Sigma \cup Q$ as the set of edge labellings: there is an edge from state q to state p labelled by $a \in \Sigma$ if and only if $p \in \delta_{push}(q, a)$ and there is an edge from state q to state p labelled by $r \in Q$ if and only if $p \in \delta_{flush}(q, r)$. To distinguish flush transitions from push transitions we denote the former ones by a double arrow.

To define the semantics of the automaton, we introduce some notations. We use letters $p, q, p_i, q_i, \ldots$ for states in Q and we set $\Sigma' = \{a' \mid a \in \Sigma\}$; symbols in Σ' are called *marked* symbols. Let $\Gamma = (\Sigma \cup \Sigma' \cup \{\#\}) \times Q$; we denote symbols in Γ as $[a\ q]$, $[a'\ q]$, or $[\#\ q]$, respectively. We set $symbol([a\ q]) = symbol([a'\ q]) = a$, $symbol([\#\ q]) = \#$, and $state([a\ q]) = state([a'\ q]) = state([\#\ q]) = q$. Given a string $\beta = B_1 B_2 \ldots B_n$ with $B_i \in \Gamma$, we set $state(\beta) = state(B_n)$.

We call a *configuration* any pair $C = \langle \beta\ ,\ w \rangle$, where $\beta = B_1 B_2 \ldots B_n \in \Gamma^*$, $symbol(B_1) = \#$, and $w = a_1 a_2 \ldots a_m \in \Sigma^* \#$. A configuration represents both the contents β of the stack and the part of input w still to process. We also set $top(C) = symbol(B_n)$ and $input(C) = a_1$.

A computation of the automaton is a finite sequence of moves $C \vdash C_1$; there are three kinds of moves, depending on the precedence relation between $top(C)$ and $input(C)$:

push move: if $top(C) \doteq input(C)$ then $\langle \beta\ ,\ aw \rangle \vdash \langle \beta[a\ q]\ ,\ w \rangle$, $\forall q \in \delta_{push}(state(\beta), a)$;

mark move: if $top(C) \lessdot input(C)$ then $\langle \beta\ ,\ aw \rangle \vdash \langle \beta[a'\ q]\ ,\ w \rangle$, $\forall q \in \delta_{push}(state(\beta), a)$;

flush move: if $top(C) \gtrdot input(C)$ then let $\beta = B_1 B_2 \ldots B_n$ with $B_j = [x_j\ q_j]$, $x_j \in \Sigma \cup \Sigma'$ and let i the greatest index such that B_i belongs to $\Sigma' \times Q$. Then $\langle \beta\ ,\ aw \rangle \vdash \langle B_1 B_2 \ldots B_{i-2}[x_{i-1}\ \ ],\ aw \rangle$, $\forall q \in \delta_{flush}(q_n, q_{i-1})$.

Push and mark moves both push the input symbol on the top of the stack, together with the new state computed by δ_{push}; such moves differ only in the marking of the symbol on top of the stack. The flush move is more complex: the symbols on the top of the stack are removed until the first marked symbol (*included*), and the state of the next symbol below them in the stack is updated by δ_{flush} according to the pair of states that delimit the portion of the stack to be removed; notice that in this move the input symbol is not relevant and it remains available for the following move.

Finally, we say that a configuration $[\#\ q_I]$ is *starting* if $q_I \in I$ and a configuration $[\#\ q_F]$ is *accepting* if $q_F \in F$. The language accepted by the automaton is defined as:

$$L(\mathcal{A}) = \left\{ x \mid \langle [\#\ q_I]\ ,\ x\# \rangle \overset{*}{\vdash} \langle [\#\ q_F]\ ,\ \# \rangle, q_I \in I, q_F \in F \right\}.$$

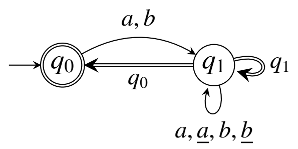
Example 2. The automaton depicted in Figure 1 accepts the Dyck language L_D of balanced strings of parentheses, with two parentheses pairs $a, \underline{a}$, and $b, \underline{b}$. The same figure also shows an accepting computation on input $aba\underline{a}b\underline{a}a\underline{a}$.

A Floyd automaton is called *deterministic* when $\delta_{\text{push}}(q, a)$ and $\delta_{\text{flush}}(q, p)$ have at most one element, for every $q, p \in Q$ and $a \in \Sigma$. Here we prove that deterministic Floyd automata are equivalent to nondeterministic ones, with a power-set construction similar to the one used for classical finite state automata.

Theorem 1. *Deterministic Floyd automata are equivalent to nondeterministic ones.*

Given a nondeterministic automaton $\mathcal{A} = \langle \Sigma, M, Q, I, F, \delta \rangle$, consider the deterministic automaton $\tilde{\mathcal{A}} = \langle \Sigma, M, \tilde{Q}, \tilde{I}, \tilde{F}, \tilde{\delta} \rangle$ where:

- $\tilde{Q} = 2^Q$ is the set of subsets of Q,
- $\tilde{I} = I \in \tilde{Q}$ is the set of initial states of $\mathcal{A}$,



	a	$\underline{a}$	b	$\underline{b}$	$\#$
a	$\lessdot$	$\doteq$	$\lessdot$		
$\underline{a}$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$	$\gtrdot$
b	$\lessdot$		$\lessdot$	$\doteq$	
$\underline{b}$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$	$\gtrdot$
$\#$	$\lessdot$		$\lessdot$		$\doteq$

	$\langle [\# \, q_0]$	$, \, aba\underline{a}\underline{b}\underline{a}a\underline{a}\#\rangle$
mark	$\langle [\# \, q_0][a' \, q_1]$	$, \, ba\underline{a}\underline{b}\underline{a}a\underline{a}\#\rangle$
mark	$\langle [\# \, q_0][a' \, q_1][b' \, q_1]$	$, \, a\underline{a}\underline{b}\underline{a}a\underline{a}\#\rangle$
mark	$\langle [\# \, q_0][a' \, q_1][b' \, q_1][a' \, q_1]$	$, \, \underline{a}\underline{b}\underline{a}a\underline{a}\#\rangle$
push	$\langle [\# \, q_0][a' \, q_1][b' \, q_1][a' \, q_1][\underline{a} \, q_1]$	$, \, \underline{b}\underline{a}a\underline{a}\#\rangle$
flush	$\langle [\# \, q_0][a' \, q_1][b' \, q_1]$	$, \, \underline{b}\underline{a}a\underline{a}\#\rangle$
push	$\langle [\# \, q_0][a' \, q_1][b' \, q_1][\underline{b} \, q_1]$	$, \, \underline{a}a\underline{a}\#\rangle$
flush	$\langle [\# \, q_0][a' \, q_1]$	$, \, \underline{a}a\underline{a}\#\rangle$
push	$\langle [\# \, q_0][a' \, q_1][\underline{a} \, q_1]$	$, \, a\underline{a}\#\rangle$
mark	$\langle [\# \, q_0][a' \, q_1][\underline{a} \, q_1][a' \, q_1]$	$, \, \underline{a}\#\rangle$
push	$\langle [\# \, q_0][a' \, q_1][\underline{a} \, q_1][a' \, q_1][\underline{a} \, q_1]$	$, \, \#\rangle$
flush	$\langle [\# \, q_0][a' \, q_1][\underline{a} \, q_1]$	$, \, \#\rangle$
flush	$\langle [\# \, q_0]$	$, \, \#\rangle$

Fig. 1. Automaton, precedence matrix, and example of computation for language L_D.

- $\tilde{F} = \{J \subseteq Q \mid J \cap F \neq \emptyset\} \subseteq \tilde{Q}$, i.e. $\tilde{F}$ is the set of subsets of Q containing at least one final state of $\mathcal{A}$,
- $\tilde{\delta} : \tilde{Q} \times (\Sigma \cup \tilde{Q}) \to \tilde{Q}$ is the transition function defined as follows. The push transition $\tilde{\delta}_{\text{push}} : \tilde{Q} \times \Sigma \to \tilde{Q}$ is defined by

  $$\tilde{\delta}_{\text{push}}(J, a) = \bigcup_{p \in J} \delta(p, a);$$

  whereas the flush transition $\tilde{\delta}_{\text{flush}} : \tilde{Q} \times \tilde{Q} \to \tilde{Q}$ is defined only on pairs $(J, K) \in \tilde{Q} \times \tilde{Q}$ such that $\delta_{\text{flush}}(p, q_1) = \delta_{\text{flush}}(p, q_2)$ for every $p \in J$ and $q_1, q_2 \in K$; in this case we set

  $$\tilde{\delta}_{\text{flush}}(J, K) = \bigcup_{p \in J} \delta_{\text{flush}}(p, q)$$

  where q is any element of K.

The proof is presented in the Appendix for space reasons.

4 Floyd automata vs Floyd grammars

The main result of this paper is the perfect match between FGs and FAs.

4.1 From Floyd grammars to Floyd automata

Theorem 2. *Any L generated by a Floyd grammar can be recognized by a Floyd automaton*

We provide a constructive proof of the theorem: given a Floyd grammar G we build an equivalent nondeterministic Floyd automaton $\mathcal{A} = \langle \Sigma, M, Q, I, F, \delta \rangle$, whose precedence matrix M is the same as the one associated with G. A successful computation of $\mathcal{A}$ will correspond to a derivation tree in G: intuitively, a push transition tries to guess the parent of the symbol currently under the input head (i.e. it determines the l.h.s of a rule of G whose r.h.s contains the current symbol); a flush transition is performed whenever the r.h.s of a rule is completed, and determines the corresponding l.h.s., thus confirming some previous guesses.

In order to keep the construction as simple as possible, we avoid introducing any optimization. Also, without loss of generality, we assume that the grammar $G = \langle \Sigma, N, P, S \rangle$ satisfies the following properties: the axiom S does not occur in the r.h.s. of any rule, no empty rule exists except possibly $S \rightarrow \varepsilon$, the other rules having S as l.h.s are renaming, and no other renaming rules exist (in other words, we assume that the G is in Fischer normal form except it is not necessarily invertible).

First of all, we introduce some notation. Enumerate the productions as follows: for any nonterminal $A \in N$, let $P_1(A)$, $P_2(A)$, $\dots P_{n(A)}(A)$ be the productions having A as l.h.s. (i.e. $n(A)$ is the number of productions having A as l.h.s.). Then, consider the set of *extended nonterminals* $EN = \{A_i \mid A \in N, i = 1, 2, \dots n(A)\}$ and define $Q = EN \times (EN \cup \{\perp\})$, where $\perp$ is a new symbol whose meaning is *undefined*. To distinguish between nonterminals and extended nonterminals, we will use capital letters $A, B, C, \dots$ and $X, Y, Z, \dots$, respectively.

When considering derivation trees of G, we label internal nodes with extended nonterminals (where the subscript of the nonterminal corresponds to the rule applied in the node). Moreover, with a slight abuse of notation, we sometimes confuse nodes and their labels, using the above convention also for internal nodes and leaves.

To define the push transition function $\delta_{push} : Q \times \Sigma \rightarrow 2^Q$, consider any derivation tree τ of G with any leaf a and let X be a's parent in τ. Figure 2 represents the various configurations that τ may exhibit.

- Case 0: if there is no leaf that precedes a in the in-order visit of τ and has depth not greater than a's depth, then let Y be the topmost ancestor of X, i.e., $Y = S_i$ for some i; this also means that $\# \lessdot a$;
- Otherwise, let b be the rightmost such leaf, and let Y be y's parent. Notice that, G being an operator grammar, Y is the nearest common ancestor of a and b. Then there are two possibilities:
  - Case 1: $X = Y$, i.e. $b \doteq a$;
  - Case 2: $X \neq Y$, and in this case b has lower depth than a, so $b \lessdot a$.

In all cases, node Z may be missing, or there may be other leaves between b and a (namely, Z's descendants); let $\hat{Z} = \perp$ if Z is missing, $\hat{Z} = Z$ otherwise. Then, for each such triple (a, X, Y), define the (a, X, Y)-*push transition*:

$$\delta_{push}((Y, \hat{Z}), a) \ni \begin{cases} (X, X) \text{ if } a \text{ is the rightmost child of } X, \\ (X, \perp) \text{ otherwise.} \end{cases} \quad (2)$$

Hence, a push transition essentially determines the parent of the symbol under the input head (actually, a "candidate" parent, since the automaton is non-deterministic).

A similar construction holds for the flush transition function $\delta_{flush} : Q \times Q \rightarrow 2^Q$. For every derivation tree with internal node X, let f and ℓ be the first and last child, respectively, of node X. Notice that both f and ℓ may be either internal nodes or leaves. Then there are two possibilities, as depicted in Figure 3:

- Case 3: there is no leaf at the left of X, then let Y be the topmost ancestor of X, i.e., $Y = S_i$ for some i;
- Case 4: otherwise, let b be the rightmost leaf at the left of X and let Y be b's parent (again, notice that Y is the nearest common ancestor of X and b, G being an operator grammar).

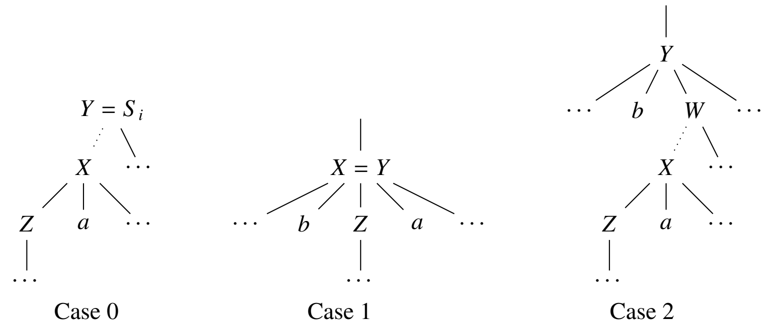


Fig. 2. Derivation tree configurations for the push transition function (nodes labelled as . . . could be missing).

Also, let $\ell_{/X}$ be ℓ if ℓ is an internal node, X otherwise; let $\tilde{f}$ be f if f is an internal node, $\perp$ otherwise. Then, for each such pair (X, Y) define the (X, Y)-*flush transition*:

$$\delta_{flush}((X, \ell_{/X}), (Y, \tilde{f})) \ni (Y, X). \tag{3}$$

Hence, the state computed by a flush transition contains two pieces of information: the first component determines the nearest ancestor of both X and b (or the axioms if b does not exist), while the second component determines the nonterminal corresponding to the r.h.s. just completed.

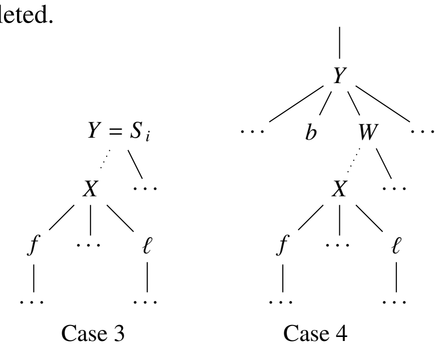


Fig. 3. Derivation tree configurations for the flush transition function (all nodes marked as . . . could be missing).

Finally, initial and final states are defined as follows.

$$I = \{(S_i, \perp) \mid 1 \le i \le n(S)\}, \qquad F = \{(S_i, A_j) \mid S \to A \in P, 1 \le i \le n(S), 1 \le j \le n(A)\}.$$

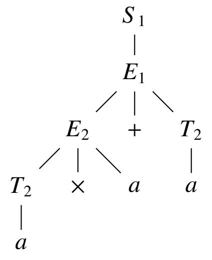
Notice that the above construction is effective. All triples (a, X, Y) involved by some push transition can be found starting from any rule $X \to \alpha$ with α containing a: if a is not the leftmost terminal of α, then take the triple (a, X, X), else apply backwards any rule with r.h.s starting with X and extend this process until all productions have been examined. Similarly for the flush transitions.

Example 3. Let G be the grammar introduced in Example 1. Following the above construction, number the rules of the grammar in the order they appear in the definition of

G (for instance, $P_2(E)$ is $E \to T \times a$). The transitions defined by the derivation tree of string $a \times a + a$, depicted in Figure 4 (left), are the following:

$$\begin{array}{ll} \delta_{push}((S_1, \bot), a) \ni (T_2, T_2) & \\ \delta_{push}((S_1, T_2), \times) \ni (E_2, \bot) & \delta_{flush}((T_2, T_2), (E_1, \bot)) \ni (E_1, T_2) \\ \delta_{push}((S_1, E_2), +) \ni (E_1, \bot) & \delta_{flush}((T_2, T_2), (S_1, T_2)) \ni (S_1, T_2) \\ \delta_{push}((E_2, \bot), a) \ni (E_2, E_2) & \delta_{flush}((E_2, E_2), (S_1, T_2)) \ni (S_1, E_2) \\ \delta_{push}((E_1, \bot), a) \ni (T_2, T_2) & \delta_{flush}((E_1, T_2), (S_1, E_2)) \ni (S_1, E_1) \end{array} \qquad (4)$$

The first one is the (a, T_2, S_1)-push transition obtained by starting from the left-most leaf (Case 0). Case 0 occurs also for the second and third push transitions, obtained considering the leaves labeled by $\times$ and $+$, respectively. The other push transitions represent instances of Cases 1 and 2, in this order. As far as flush transitions are concerned, Case 4 occurs only in the first stated transition, with $X = T_2$, $b = +$ and $Y = E_1$, whereas all other productions represent instances of Case 3. Hence, on input $a \times a + a$, the automaton $\mathcal{A}$ obtained from G may execute the computation represented in Figure 4 (right).



$$\begin{array}{lll} & \langle [\#\,(S_1, \bot)] & ,\ a \times a + a\,\#\rangle \\ \text{mark} & \langle [\#\,(S_1, \bot)][a'\,(T_2, T_2)] & ,\ \times a + a\,\#\rangle \\ \text{flush} & \langle [\#\,(S_1, T_2)] & ,\ \times a + a\,\#\rangle \\ \text{mark} & \langle [\#\,(S_1, T_2)][\times'\,(E_2, \bot)] & ,\ a + a\,\#\rangle \\ \text{push} & \langle [\#\,(S_1, T_2)][\times'\,(E_2, \bot)][a\,(E_2, E_2)] & ,\ + a\,\#\rangle \\ \text{flush} & \langle [\#\,(S_1, E_2)] & ,\ + a\,\#\rangle \\ \text{mark} & \langle [\#\,(S_1, E_2)][+'\,(E_1, \bot)] & ,\ a\,\#\rangle \\ \text{mark} & \langle [\#\,(S_1, E_2)][+'\,(E_1, \bot)][a'\,(T_2, T_2)] & ,\ \#\rangle \\ \text{flush} & \langle [\#\,(S_1, E_2)][+'\,(E_1, T_2)] & ,\ \#\rangle \\ \text{flush} & \langle [\#\,(S_1, E_1)] & ,\ \#\rangle \end{array}$$

Fig. 4. Derivation tree (left) and computation (right) for the string $a \times a + a$.

The equivalence between G and the automaton described above is based on the following lemma, whose proof can be found in the Appendix. As usual we set $\Gamma = (\Sigma \cup \Sigma') \times Q = (\Sigma \cup \Sigma') \times (EN \times (EN \cup \{\bot\}))$ and we denote an element in Γ as $[a\ (X, Y)]$. To avoid an excessively cumbersome notation, when describing the transitions between configurations, we omit the extreme parts (i.e. the lower part of the stack and a suffix of the input string) which are not affected by the computation.

We define the *depth of a computation* $C_1 \overset{*}{\vdash} C_2$ as the maximum number of marked symbols in one of the traversed configurations, minus the number of marked symbol on the stack in configuration C_1; we define the *depth of a derivation* $W \overset{*}{\Rightarrow} \alpha$ as the depth of the corresponding derivation tree. When useful, we make the depth h of a computation or a derivation explicit as in $C_1 \overset{[h]}{\vdash} C_2$ and $X \overset{[h]}{\Rightarrow} \alpha$.

Lemma 1. *Let Y, W be extended nonterminals of G, $v \in \Sigma^*$, $a \lessdot v \gtrdot b$, and $\bar{a} \in \{a, a'\}$. Then for all $h \geq 1$:*

$$\langle [\bar{a}\ (Y, \bot)]\ ,\ vb\rangle \overset{[h]}{\vdash} \langle [\bar{a}\ (Y, W)]\ ,\ b\rangle \quad \textit{iff} \quad \exists \alpha, \beta \textit{ such that } Y \to \alpha a W \beta,\ W \overset{[h]}{\Rightarrow} v \textit{ in } G.$$

From the lemma the theorem easily follows by using a special case $S \to A$ (with implicit # as a and b).

4.2 From Floyd automata to Floyd grammars

Given a Floyd automaton $\mathcal{A} = \langle \Sigma, M, Q, I, F, \delta \rangle$, we show how to build an equivalent Floyd grammar G having operator precedence matrix M. In order to keep the construction as easy as possible, w.l.o.g we assume that M is $\doteq$-acyclic. Remind that, as discussed in Section 2, this hypothesis could be replaced by weaker ones.

We need some notation and definitions. First of all, we shall represent a push transition with a simple arrow $\longrightarrow$, a flush transition with a double arrow $\Longrightarrow$, and a path defined by a sequence of transitions with a wavy arrow $\rightsquigarrow$.

We define *chains* in $\mathcal{A}$ recursively. A *simple chain* is a word $a_0 a_1 a_2 \dots a_n a_{n+1}$, written as $\langle {}^{a_0} a_1 a_2 \dots a_n {}^{a_{n+1}} \rangle$, such that: $a_0, a_{n+1} \in \Sigma \cup \{\#\}$, $a_i \in \Sigma$ for every $i = 1, 2, \dots n$, $M_{a_0, a_{n+1}} \neq \emptyset$, and $a_0 \lessdot a_1 \doteq a_2 \dots a_{n-1} \doteq a_n \gtrdot a_{n+1}$. A *composed chain* in $\mathcal{A}$ is a word $a_0 x_0 a_1 x_1 a_2 \dots a_n x_n a_{n+1}$, where $\langle {}^{a_0} a_1 a_2 \dots a_n {}^{a_{n+1}} \rangle$ is a simple chain, and $x_i \in \Sigma^*$ is the empty word or is such that $\langle {}^{a_i} x_i {}^{a_{i+1}} \rangle$ is a chain (simple or composed), for every $i = 0, 1, \dots, n-1$. Such a composed chain will be written as $\langle {}^{a_0} x_0 a_1 x_1 a_2 \dots a_n x_n {}^{a_{n+1}} \rangle$.

We call a *support* for the simple chain $\langle {}^{a_0} a_1 a_2 \dots a_n {}^{a_{n+1}} \rangle$ any path in $\mathcal{A}$ of the form

$$q_0 \xrightarrow{a_1} q_1 \longrightarrow \dots \longrightarrow q_{n-1} \xrightarrow{a_n} q_n \overset{q_0}{\Longrightarrow} q_{n+1} \tag{5}$$

Notice that the label of the last (and only) flush is exactly q_0, i.e. the first state of the path; this flush is executed because of relation $a_n \gtrdot a_{n+1}$. We call a *support for the composed chain* $\langle {}^{a_0} x_0 a_1 x_1 a_2 \dots a_n x_n {}^{a_{n+1}} \rangle$ any path in $\mathcal{A}$ of the form

$$q_0 \overset{x_0}{\rightsquigarrow} q'_0 \xrightarrow{a_1} q_1 \overset{x_1}{\rightsquigarrow} q'_1 \xrightarrow{a_2} \dots \xrightarrow{a_n} q_n \overset{x_n}{\rightsquigarrow} q'_n \overset{q'_0}{\Longrightarrow} q_{n+1} \tag{6}$$

where, for every $i = 0, 1, \dots, n$:

- if $x_i \neq \epsilon$, then $q_i \overset{x_i}{\rightsquigarrow} q'_i$ is a support for the chain $\langle {}^{a_i} x_i {}^{a_{i+1}} \rangle$, i.e., it can be decomposed as $q_i \overset{x_i}{\rightsquigarrow} q''_i \overset{q_i}{\Longrightarrow} q'_i$.
- if $x_i = \epsilon$, then $q'_i = q_i$.

Notice that the label of the last flush is exactly q'_0.

We are now able to define a Floyd grammar $G = \langle \Sigma, N, S, P \rangle$. Nonterminals are the 4-tuples $(a, q, p, b) \in \Sigma \times Q \times Q \times \Sigma$, written as $\langle {}^a p, q^b \rangle$, plus the axiom S. Rules are built as follows:

- for every support of type (5) of a simple chain, add the rule
$$\langle {}^{a_0} q_0, q_{n+1} {}^{a_{n+1}} \rangle \longrightarrow a_1 a_2 \dots a_n \ ;$$
if also $a_0 = a_{n+1} = \#$, q_0 is initial, and q_{n+1} is final, add the rule $S \to \langle {}^{\#} q_0, q_{n+1} {}^{\#} \rangle$;
- for every support of type (6) of a composed chain, add the rule
$$\langle {}^{a_0} q_0, q_{n+1} {}^{a_{n+1}} \rangle \longrightarrow N_0 a_1 N_1 a_2 \dots a_n N_n \ ;$$
where, for every $i = 0, 1, \dots, n$, $N_i = \langle {}^{a_i} q_i, q'_i {}^{a_{i+1}} \rangle$ if $x_i \neq \epsilon$ and $N_i = \epsilon$ otherwise.

Notice that the above construction is effective thanks to the hypothesis of $\doteq$-acyclicity of the OPM. This implies that the length of the r.h.s. is bounded (see Section 2); on the other hand, the cardinality of the nonterminal alphabet is finite. Hence there is only a finite number of possible productions for G and only a limited number of chains to be considered.

5 ω-languages

Having an operational model that defines Floyd Languages, it is now straightforward to introduce extensions to ω-languages.

For instance, the classical Büchi condition of acceptance can be easily adapted to FAs. Consider an infinite word $x \in \Sigma^\omega$, and an *infinite computation* of the automaton $\mathcal{A}_M = \langle \Sigma, M, Q, I, F, \delta \rangle$ on x, i.e. an ω-sequence of configurations $\mathcal{S} = \langle \beta_0\ ,\ x_0 \rangle \langle \beta_1\ ,\ x_1 \rangle \ldots$, such that $\langle \beta_0\ ,\ x_0 \rangle = \langle [\#\ q_I]\ ,\ x \rangle$, $q_I \in I$ and $\langle \beta_i\ ,\ x_i \rangle \vdash \langle \beta_{i+1}\ ,\ x_{i+1} \rangle$. We say that $x \in L(\mathcal{A})$ if and only if there exists $q_F \in F$ such that configurations with stack $[\#\ q_F]$ occur infinitely often in $\mathcal{S}$.

Quite naturally, ω-VPLs are a proper subset of this class of languages, as it is shown by the following example.

Example 4. We define here the stack management of a simple programming language that is able to handle nested exceptions. For simplicity, there are only two procedures, called a and b. Calls and returns are denoted by $call_a$, $call_b$, ret_a, ret_b, respectively. During execution, it is possible to install an exception handler hnd. The last signal that we use is rst, that is issued when an exception occur, or after a correct execution to uninstall the handler. With a rst the stack is "flushed", restoring the state right before the last hnd. The automaton is presented in Figure 5 (notice that it is an extension of the automaton in Figure 1). It is easy to modify this example to model the case of *unnested* exceptions, to fit with other application contexts.

	$call_a$	ret_a	$call_b$	ret_b	hnd	rst
$call_a$	$\lessdot$	$\doteq$	$\lessdot$		$\lessdot$	$\gtrdot$
ret_a	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$		$\gtrdot$
$call_b$	$\lessdot$		$\lessdot$	$\doteq$	$\lessdot$	$\gtrdot$
ret_b	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$		$\gtrdot$
hnd	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$		$\doteq$
rst	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$		
#	$\lessdot$		$\lessdot$		$\lessdot$	

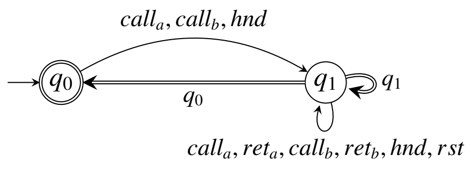


Fig. 5. Precedence matrix and automaton for an ω-language. There is no column indexed by # since words are infinite.

6 Conclusions and further research

Recently, we advocated that operator precedence grammars and languages, here renamed after their inventor Robert Floyd, deserve renewed attention in the realm of formal languages. The main reasons to support our claim are:

- The fact that this family of languages properly includes visibly pushdown languages [2], a new family that has been proposed with the main motivation of extending powerful model checking techniques beyond the limits of finite state machines.
- The fact that it enjoys all closure properties with respect to the main algebraic operations that are exhibited by regular languages and VPLs.
- The fact that, unlike other deterministic languages -either strictly more powerful than them, or incomparable with them- such as LR, LL, and simple precedence ones, FLs can be parsed without applying a strictly left-to-right order; this feature becomes particularly relevant in these days since it allows to exploit much better the gains in efficiency offered by massive parallelism.

In this paper we filled a rather surprising “hole” in the theory of these languages, namely the lack of an appropriate family of automata that perfectly matches the generative power of their grammars. We defined FAs with such a goal in mind and we proved their equivalence with FGs. Both facts turned out to be non-trivial jobs and showed further interesting peculiarities of this pioneering family of deterministic languages. A first “byproduct” of the new automata family is the extension of FLs to ω-languages, i.e., languages consisting of infinite strings, a more and more important aspect of formal language theory needed to deal with never ending computations. In this case too FL ω-languages proved to augment the descriptive capabilities of the original VPLs.

As a first step towards applicability of the results presented in this paper, and also to validate our approach with several practical examples, we implemented a simple prototypical tool, called *Flup*. Flup contains an interpreter for non-deterministic Floyd Automata, and a Floyd Grammar to Automata translator, that directly applies the construction presented in Section 4.1. All the examples presented in the paper were tried on, or generated by the tool.[5]

We are confident that suitable future research will further strengthen the importance of, and motivation for, re-inserting FLs in the main stream of formal language literature. In particular it would be interesting to complete the parallel analysis and comparison with VPLs by investigating a characterization in terms of suitable logic formulas [2]; by this way motivation for, and application of, strong model checking techniques would be further enhanced.

References

1. R. Alur and P. Madhusudan. Visibly pushdown languages. In *STOC: ACM Symposium on Theory of Computing (STOC)*, 2004.
2. R. Alur and P. Madhusudan. Adding nesting structure to words. *Journ. ACM*, 56(3), 2009.
3. J. Berstel and L. Boasson. Balanced grammars and their languages. In W. Brauer et al., editor, *Formal and Natural Computing*, volume 2300 of *LNCS*, pages 3–25. Springer, 2002.
4. D. Caucal. Boolean algebras of unambiguous context-free languages. In R. Hariharan, M. Mukund, and V. Vinay, editors, *FSTTCS 2008*, Dagstuhl, Germany, 2008.
5. S. Crespi Reghizzi and D. Mandrioli. Algebraic properties of structured context-free languages: old approaches and novel developments. In *WORDS 2009 - 7th Int. Conf. on Words, preprints*. available as http://arXiv.org/abs/0907.2130, 2009.
6. S. Crespi Reghizzi and D. Mandrioli. Operator precedence and the visibly pushdown property. In A. Horia Dediu, H. Fernau, and C. Martín-Vide, editors, *LATA*, volume 6031 of *LNCS*, pages 214–226. Springer, 2010.
7. S. Crespi Reghizzi, D. Mandrioli, and D. F. Martin. Algebraic properties of operator precedence languages. *Information and Control*, 37(2):115–133, May 1978.
8. M. J. Fischer. Some properties of precedence languages. In *STOC ’69: Proc. first annual ACM Symp. on Theory of Computing*, pages 181–190, New York, NY, USA, 1969. ACM.
9. R. W. Floyd. Syntactic analysis and operator precedence. *Journ. ACM*, 10(3):316–333, 1963.
10. D. Grune and C. J. Jacobs. *Parsing techniques: a practical guide*. Springer, New York, 2008.
11. M. A. Harrison. *Introduction to Formal Language Theory*. Addison Wesley, Reading, MA, 1978.
12. D. Nowotka and J. Srba. Height-deterministic pushdown automata. In L. Kucera and A. Kucera, editors, *MFCS 2007, Ceský Krumlov, Czech Republic, August 26-31, 2007, Proceedings*, volume 4708 of *LNCS*, pages 125–134. Springer, 2007.
13. A. K. Salomaa. *Formal Languages*. Academic Press, New York, NY, 1973.

[5] The prototype is freely available at *http://home.dei.polimi.it/pradella*.

I Appendix

I.1 Proof of Theorem 1

Theorem 1 is a direct consequence of the following two lemmata:

Lemma 2. *For every path $q_0 \xrightarrow{r_0} q_1 \xrightarrow{r_1} q_2 \xrightarrow{r_2} \dots \xrightarrow{r_n} q_n$ in $\mathcal{A}$, where $q_i \in Q$ and $r_i \in \Sigma \cup Q$, there is a path $J_0 \xrightarrow{s_0} J_1 \xrightarrow{s_1} J_2 \xrightarrow{s_2} \dots \xrightarrow{s_n} J_n$ in $\tilde{\mathcal{A}}$, where $J_i \ni q_i$, and $s_i = r_i$ if $r_i \in \Sigma$ or $s_i = \{r_i\}$ if $r_i \in Q$.*

Proof. It is enough to set $J_0 = \{q_0\}$ and $J_i = \delta(q_{i-1}, s_i)$ for every $i \geq 1$. Notice that, in this definition, $\delta = \delta_{\text{push}}$ if $s_i \in \Sigma$, while $\delta = \delta_{\text{flush}}$ if $s_i \in Q$.

Lemma 3. *For every path $J_0 \xrightarrow{s_0} J_1 \xrightarrow{s_1} J_2 \xrightarrow{s_2} \dots \xrightarrow{s_n} J_n$ in $\tilde{\mathcal{A}}$, with $s_i \in \Sigma \cup \tilde{Q}$, and for every $q_n \in J_n$, there is a path $q_0 \xrightarrow{r_0} q_1 \xrightarrow{r_1} q_2 \xrightarrow{r_2} \dots \xrightarrow{r_n} q_n$ in $\mathcal{A}$, where $q_i \in J_i$, and $r_i = s_i$ if $s_i \in \Sigma$ or $r_i \in s_i$ if $s_i \in \tilde{Q}$.*

Proof. We reason by induction on the length n of the path. If $n = 1$, let $J_0 \xrightarrow{s_0} J_1$ and $q_1 \in J_1$. First consider the case $s_0 \in \Sigma$. Then $J_1 = \tilde{\delta}_{\text{push}}(J_0, s_0) = \cup_{p \in J_0} \delta_{\text{push}}(p, s_0)$. Since $q_1 \in J_1$, there exists $q_0 \in J_0$ such that $q_1 \in \delta_{\text{push}}(q_0, s_0)$ and hence $q_0 \xrightarrow{s_0} q_1$ in $\mathcal{A}$. In the case $x \in \tilde{Q}$, we have $J_1 = \tilde{\delta}_{\text{flush}}(J_0, s_0) = \bigcup_{p \in J_0} \delta_{\text{flush}}(p, r_0)$ for any $r_0 \in s_0$; then, since $q_1 \in J_1$, there exists $q_0 \in J_0$ such that $q_1 \in \delta_{\text{flush}}(q_0, r_0)$ and hence $q_0 \xrightarrow{r_0} q_1$ in $\mathcal{A}$.

If $n \geq 1$, consider $J_0 \xrightarrow{s_0} J_1 \xrightarrow{s_1} J_2 \xrightarrow{s_2} \dots \xrightarrow{s_n} J_n$ in $\tilde{\mathcal{A}}$, $q_n \in J_n$, and assume by induction that in $\mathcal{A}$ there is a path $q_1 \xrightarrow{r_1} q_2 \xrightarrow{r_2} \dots \xrightarrow{r_n} q_n$ with $q_i \in J_i$, and $r_i = s_i$ if $s_i \in \Sigma$ or $r_i \in s_i$ if $s_i \in \tilde{Q}$. In the case $s_0 \in \Sigma$ we have $q_1 \in J_1 = \tilde{\delta}_{\text{push}}(J_0, s_0) = \bigcup_{p \in J_0} \delta_{\text{push}}(p, s_0)$, hence there exists $q_0 \in J_0$ such that $q_1 \in \delta_{\text{push}}(q_0, s_0)$; thus $q_0 \xrightarrow{s_0} q_1$ in $\mathcal{A}$. In the case $s_0 \in \tilde{Q}$ we have $q_1 \in J_1 = \tilde{\delta}_{\text{flush}}(J_0, s_0) = \bigcup_{p \in J_0} \delta_{\text{flush}}(p, r_0)$ for any $r_0 \in s_0$; then there exists $q_0 \in J_0$ such that $q_1 \in \delta_{\text{flush}}(q_0, r_0)$ and hence $q_0 \xrightarrow{r_0} q_1$ in $\mathcal{A}$.

At this point the theorem follows immediately by considering paths beginning in a initial state and ending in a final state, and observing that the stacks of the two automata evolve in parallel.

I.2 Proof of Lemma 1

The lemma is equivalent to the following two properties.

(i) For every Y, X, $a \lessdot c \doteq x \gtrdot d$, $\mathcal{A}$ admits the computation

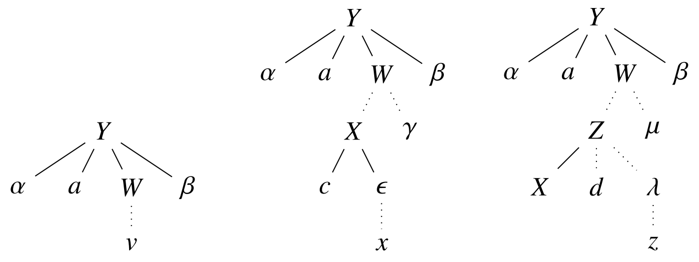
$$\langle [\bar{a}\ (Y, \perp)]\ ,\ cxd \rangle \overset{[k]}{\vdash} \langle [\bar{a}\ (Y, X)]\ ,\ d \rangle$$

if and only if there exist $W, \alpha, \beta, \gamma, \epsilon$ such that $Y \to \alpha a W \beta$, $W \overset{*}{\Rightarrow} X\gamma$, $X \to c\epsilon$, $\epsilon \overset{[k]}{\Rightarrow} x$.

(ii) For every Y, X, Z, $a \lessdot d \underline{\lessdot} z \gtrdot e$, $\mathcal{A}$ admits the computation

$$\langle [\bar{a}\ (Y, X)]\ ,\ dze\rangle \overset{[k]}{\vdash} \langle [\bar{a}\ (Y, Z)]\ ,\ d\rangle$$

if and only if there exist $W, \alpha, \beta, \mu, \lambda$ such that $Y \to \alpha a W \beta$, $W \overset{*}{\Rightarrow} Z\mu$, $Z \to Xd\lambda$, $\lambda \overset{[k]}{\Rightarrow} z$.



Statement of Lemma Property (i) Property (ii)

Notice that in (i) W and X may coincide (i.e., γ may be empty), and in (ii) W and Z may coincide (i.e., μ may be empty). For $h = 1$, the lemma is given by property (i) with $W = X$ and $k = 0$ (for $cx = v$, $d = b$); for $h > 1$ we have $v = cxd_1z_1d_2 \ldots d_nz_n$ for some $c \underline{\lessdot} x \gtrdot d_1$, $d_i \underline{\lessdot} z_i \gtrdot d_{i+1}$ (with x, z_i possibly empty). Then, applying first property (i) and then, repeatedly, property (ii), one gets the lemma.

We prove property (i) reasoning by induction on k. First let $k = 0$; in this case $\epsilon = x$, i.e. $X \to cx$. Hence, if $x = c_1 \ldots c_n$, during the computation defined in (i), $\mathcal{A}$ has to execute the following series of moves: a marked (c, X_0, Y)-push transition (case 2 without Z), then a sequence of (c_i, X_0, X_0)-push transitions (case 1 without Z), and finally a (X_0, Y)-flush transition, for a suitable X_0:

$$\langle \eta[\bar{a}\ (Y, \bot)][c'\ (X_0, \bot)][c_1\ (X_0, \bot)] \ldots [c_n\ (X_0, X_0)]\ ,\ d\rangle \vdash \langle \eta[\bar{a}\ (Y, X_0)]\ ,\ d\rangle.$$

To end in the right configuration, we necessarily have $X_0 = X$. Moreover, by the definition of transitions in $\mathcal{A}$, X must satisfy exactly the relations defined in (i). Viceversa, if the grammar admits the derivation defined in (i), then obviously the automaton $\mathcal{A}$ admits the previous moves.

One can prove similarly property (ii) for $k = 0$: in this case, both the marked (d, Z, Y)-push transition and the (Z, Y)-flush transition involve the extended nonterminal X (i.e., the second component of the state on the top of the stack).

Now, assuming that properties (i) and (ii) hold for depths lower than k, we prove them for k. First consider (i) and let $x = u_0c_1u_1c_2 \ldots c_mu_m$ with $c \lessdot u_0$, $u_{i-1} \gtrdot c_i \lessdot u_i$ (with any u_i possibly empty), and $c_i \doteq c_{i+1}$. By the definition of the transition function, $\mathcal{A}$ admits the computation in (i) if and only if there exist $W, \alpha, \beta, \gamma, \epsilon$ as in (i) and moreover there exist $U_0, \cdots U_m$ such that $\epsilon = U_0c_1U_1 \ldots c_mU_m$ and $U_i \overset{[k_i]}{\Rightarrow} u_i$ with $k_i < k$ (U_i is missing iff u_i is empty). Hence one can apply the inductive hypothesis and get the result.

One can prove similarly property (ii) for k greater than 0: again, in this case, both the marked (d, Z, Y)-push transition and the (Z, Y)-flush transition involve the extended nonterminal X. $\square$