DOI 10.2298/CSIS101109004J

A Generic Parser for Strings and Trees

Riad S Jabri

Computer Science Department
King Abdullah II School for Information Technology
University of Jordan, Amman 11942 –Jordan
jabri@ju.edu.jo

Abstract. In this paper, we propose a two fold generic parser. First, it simulates the behavior of multiple parsing automata. Second, it parses strings drawn from either a context free grammar, a regular tree grammar, or from both. The proposed parser is based on an approach that defines an extended version of an automaton, called position-parsing automaton (PPA) using concepts from LR and regular tree automata, combined with a newly introduced concept, called state instantiation and transition cloning. It is constructed as a direct mapping from a grammar, represented in an expanded list format. However, PPA is a non-deterministic automaton with a generic bottom–up parsing behavior. Hence, it is efficiently transformed into a reduced one (RBA). The proposed parser is then constructed to simulate the run of the RBA automaton on input strings derived from a respective grammar. Without loss of generality, the proposed parser is used within the framework of pattern matching and code generation. Comparisons with similar and well-known approaches, such as LR and RI, have shown that our parsing algorithm is conceptually simpler and requires less space and states.

Keywords: bottom-up automata, parsing, regular tree grammars.

1. Introduction

Without loss of generality, in this paper we propose a generic parser within the framework of code generators, in which machine instructions are specified by patterns that are drawn either from context-free grammars (LALR), regular tree grammars or from rewrite systems [1, 4, 5,12,15]. Pattern matching is then performed using respective parsing automata to generate an optimal set of machine instructions [6, 15]. The LALR parsing automata are efficient but too restrictive to handle patterns drawn from ambiguous context-free grammars. In contrast, the tree parsers are not restrictive but their optimization is a complex task. Approaches to handle ambiguous context-free grammars have been suggested in [14]. However, they are not intended to handle regular tree grammars. On the other hand, approaches to combine LR parsing and bottom-up tree parsing do exist, such as the one suggested in [9]. It is based on transforming a tree automaton recognizing a regular tree language to a pushdown automaton recognizing the same language in postfix notation. In addition to the transformation overhead, this approach assumes that the transformed grammar is in normal form, deterministic and without hidden-left and right recursion [9]. Hence, there is a need for a parsing approach that admits ambiguous grammar and maintains the efficiency of LALR parsers and the expressive power of tree parsers. To satisfy such a need, we propose an approach for a hybrid parsing of both general context-free and regular tree grammars. According to the proposed approach, and as a framework for parsing, a generic grammar (GG) is defined as one that can be either instantiated by a context-free grammar or by a regular tree grammar. GG is then represented in an expanded list format (PLF). As such, PLF constitutes a prefix representation of derivation trees for the grammar GG, where the ranked terminals are considered as nonterminals and the recursive terminal symbols are terminalized. Hence, the repeated expansion in the derivation trees is incorporated as recursion-invocation and recursion-termination. The recursion-invocation constitutes an ε-transition from the recursive occurrence to its respective head, while recursion-termination implies an ε-transition from the recursive-head to its respective occurrence. Hence, the recursive occurrence initiates a derivation/reduction path as the one initiated by its respective head and considered as an instance of the original one**.** To distinguish between instances initiated by different recursive occurrences, an instance-identifier (ID) is assigned to each initiated path. Therefore, PLF with incorporated recursion implies initiation and termination of derivations/ reductions paths, as well as respective instances of these paths. A nondeterministic parsing automaton, called position parsing automaton (PPA), is then constructed in terms of a set of state instances and respective parsing actions as a direct mapping from the PLF of the production respective to the start symbol S of GG. PPA is defined based on a newly introduced concept of state and transition instances. According to such concept, Each PPA state (q, (inst)) is defined with an implicit index (inst) $=\varepsilon$. State instances are then created and terminated by appending and deleting instance-identifiers (ID) atop of the index (inst). The PPA transitions are augmented by semantic-actions that are performed at run time to create and terminate instances of PPA states and transitions in accordance to the incorporated recursion-invocation and recursion-termination. In addition to the newly introduced concept of state and transition instantiation, the states and transitions of the PPA automaton are defined based on combined concepts from LR (0) automata, regular tree automata. PPA has been adopted in [8] to construct a pattern matcher, where its parsing behavior has been adapted and synchronized with pattern matching and code selection. In this research, we emphasize the generality and soundness of PPA parsing behavior. Hence, we extend its behavior to cover subtle grammar cases. This includes direct and indirect (hidden) left / right recursion and ambiguous grammars with shift/reduce and reduce/reduce conflicts. In addition, we optimize its mapping from PLF by following a concurrent and a gradual construction approach for both PLF and PPA. As a simulator for tree parsing automata, finite automata and shift–reduce automata, the generic behavior of

PPA has been enriched by conceptual explanation, theory and graphical representation. However, PPA is a nondeterministic automaton. To obtain a more deterministic parsing behavior, PPA is efficiently transformed into a reduced one, called RBA, according to a proposed subset construction approach. The RBA automaton is represented by a parsing table that specifies the parsing actions respective to a given state and each input symbol. A parser is then constructed to simulate the run the RBA automaton on strings derived from a generic grammar. Furthermore, the proposed parser is based on formalizing the parsing process and its solution in a way that guarantees its soundness, generalization and its efficient implementation. This was demonstrated by the proposed theory and by the respective implementation algorithms.

Our parsing approach is motivated by reduction incorporated parsing introduced in [2, 3] and further developed in [10, 14]. According to these approaches, a finite automaton for terminalized grammar (RIA) is constructed to handle the regular parts of the language. Such automaton is then extended by recursive call automaton (RCA) to handle recursion. However, the suggested automaton in [2,3] does not handle hidden left recursion. In addition, the construction approach is based on by hand generated terminalization. Once terminalization has been detected, RCA is constructed. In [14] RIA has been extended to handle hidden-left recursion, while in [10] an automatic computation for terminalization has been suggested. In contrast, our approach handles hidden and direct left/right recursion. Terminalization is handled as recursion-invocation and recursion-termination during the concurrent construction of PLF and PPA. In addition, and rather than, the static creation of RCA, an instance is dynamically created during parsing. This achieved by the newly introduced concept of state instantiations and embedded semantic actions. Furthermore, our approach is embedded by appropriate concepts to parse regular trees. Further comparisons with these approaches and similar ones are given in Section 6.

The remainder of this paper is organized as follows. Section 2 presents preliminaries. Section 3 presents the proposed parsing approach, followed by the definitions of the newly introduced concepts. This includes the proposed nondeterministic parsing automaton (PPA) and the theory on which it is based. Section 4 presents the PPA construction algorithm. Section 5 presents the subset construction and subsequently the proposed parser, followed by a discussion and a conclusion that are given in section 6 and section 7 respectively.

2. Preliminaries

For our further discussions, we assume the following definitions based on the ones given in [1, 8, 11, 12, 13, 15].

Definition 1. The 4-tuple G = (Σ, N, P, S) defines a context-free grammar, where:

- Σ is an alphabet of terminal symbols.
- N is a finite set of nonterminal symbols, where $S \in N$ is the start symbol.
- P is a finite set of productions p having the form p: $A \rightarrow V$, where $A \in N$ and $V \in (\Sigma \cup N)^*$.

Definition 2. A ranked alphabet Σ is a finite set of symbols (operators, constructors) such that each member of Σ has a nonnegative integer, called a rank (arity). The rank is defined by the function *arity*: $\Sigma \rightarrow$ **N,** where

- **N** denotes the set of natural numbers.
- The terminals having a rank ≥ 1 are called operators and the ones having a rank =0 are called constants.
- The members of Σ are grouped into the subsets $\Sigma_0, \ldots, \Sigma_n$ such that $\Sigma_n = \{a \in \Sigma \mid \text{arity}(a) = i, i \in (0,1,\ldots,n)\}$.

Definition 3. A tree language over Σ, denoted by T (Σ), consists of all possible terms that are inductively defined as:

- If $a \in \Sigma_0$ then $a \in T(\Sigma)$
- If $t_1, \ldots, t_n \in T(\Sigma)$ and $a \in \Sigma_n$ then $a(t_1, \ldots, t_n) \in T(\Sigma)$.
- These terms represent trees with internal nodes and leaves labeled by operators and constants respectively. The number of children of a node labeled by an operator a is *arity* (a).

Definition 4. (Regular tree grammar). The 4-tuple G = (Σ, N, P, S) defines a regular tree grammar, where:

- Σ is a ranked alphabet
- N is a finite set of nonterminals with an assumed rank =0, where $S \in N$ is the start symbol.
- P is a finite set of productions p having the form p: $A \rightarrow V$, where $A \in N$ and $V \in T(\Sigma \cup N)$ is a prefix encoding of a tree over $(\Sigma \cup N)$.

We denote A and V by LHS(p) and RHS(p), to represent the left hand side and the right hand side of the production p respectively. Given p: $A \rightarrow V$ and $t1 \in T(\Sigma \cup N)$, we say t1 derives $t2 \in T(\Sigma \cup N)$ in one-step, denoted by $t1 \Rightarrow t2$, if LHS (p) $\in t1$ and RHS (p)$\in t2$. Applying zero or more such derivation steps on t1 is denoted by $t1 \Rightarrow^* t2$, where each step derives a tree $t \in T(\Sigma \cup N)$.

Definition 5. (Regular tree language). The language generated by the grammar G is defined by the set L (G) = $\{t \mid t \in T(\Sigma) \text{ and } S \Rightarrow^* t\}$.

Definition 6. (Regular tree parsing). Parsing a tree $t \in L(G)$ is the process of constructing a possible S– derivation tree for t, which is inductively defined as a tree with a root labeled by the start grammar symbol S and with children labeled by the RHS(S), where:

- The children labeled by terminals $\in \Sigma_0$ are leaves. The children labeled by nonterminals are interior nodes and constitute roots for V-derivation trees, each one of which is inductively defined as S-derivation tree.

- The children labeled by ranked terminals $\in \Sigma_n$ are interior nodes and constitute roots for sub trees, each one of which is inductively defined as S-derivation tree. However, the children of such sub trees are labeled by the grammar symbols composing the arity of their respective roots.

Definition 7. Generic Grammar (GG) A generic grammar is a 4-tuple GG= (Σ, N, P, S) that is either instantiated by a context-free grammar, as given by Definition 1, or by a regular tree grammar, as given by Definition 4. However, the ranked terminals are nonterminalized as given by Definition 8. Subsequently, in our further discussion, we use the following generic terms.

- Derivation tree (DT)**:** Denotes either an S-derivation tree or a classical one for the context free grammars.
- String: Denotes either a prefix encoding of the input trees or sentences generated by a context free grammar.

Definition 8. Given a $(t_1,\ldots,tn) \in T(\Sigma)$, the ranked terminal ($a \in \Sigma_n$) is nonterminalized as $a \rightarrow t1\ldots tn$, where each ranked terminal in (t1,…, tn) is inductively nonterminalized.

Example 1. Given GG = (Σ, N, P, S), it can be instantiated with a regular tree grammar as follows:

- $\Sigma = \{\Sigma_0, \Sigma_1, \Sigma_2\}$, where $\Sigma_0= \{c\}$, $\Sigma_1= \{m\}$ and $\Sigma_2 = \{+\}$
- N = {R}, R = S and P= { R→ +(m(c),R) | + (m(R),R) | c }
- Such grammar generates strings (trees) of the forms: c, + (m(c), c),…, +(m(c),+ (m(c),c)).
- GG can be instantiated with a context-free grammar as follows:
- $\Sigma= \Sigma_0 = \{a, b, c\}$; N = {S, A , B} and S is the start symbol
- P = {S → A b , A → a B , B → c | c B}
- Such grammar generates strings (sentences) of the forms: acb, accb, accccb,...

3. The proposed parsing approach

Given a generic grammar GG, we propose a parsing approach based on simulating the run of a reduced bottom--up automaton (RBA) on strings generated by GG. RBA is obtained as result of a subset construction applied on a proposed nondeterministic automaton (PPA). Such automaton is an extended version of a one introduced in [8]. It is constructed in terms of a set of states and parsing actions as a direct mapping from the production respective to the start symbol S of GG, which is represented in the expanded list format (PLF). According to PLF, a production p: $A \rightarrow \alpha$ in GG is defined as PLF (A) = A (α), where each nonterminal $\neq$ A in α is inductively replaced by the PLF respective to its corresponding production(s). The recursive occurrence of the symbol A in RHS (p) is not replaced by a corresponding PLF. Only, it is designated as recursive instance of its respective head. In addition, each grammar symbol in PLF is assigned a doted integer as an index to reflect its occurrence order (position), as well as its nesting depth. As such, PLF constitutes a prefix representation of derivation trees for the grammar GG, where the repeated expansion in the derivation trees is represented as an incorporated recursion. Further, the positions assigned to the grammar symbols of PLF represent their paths in a respective derivation tree. Hence, the types of the grammar symbols and their occurrence order in PLF imply derivation/reduction relationships. The PPA states and parsing actions are then defined based on such relationships, using concepts from tree parsing and shift-reduce automata that are combined with newly introduced ones to handle the incorporation of recursion in PLF. Such concepts include state-instantiation, transition-cloning and transitions having embedded semantic-actions. In this section, we present the PLF form and PPA automaton. In the following sections, we present the subset construction of PPA into the reduced RBA automaton and the proposed parser. In addition and where appropriate, we present theory to demonstrate the generality and the soundness of the proposed approach. However, we immediately present an example to illustrate the proposed approach and to facilitate our further discussion.

Example 2. Consider the grammar of Example 1. A PLF form respective to the start grammar symbol S is defined as PLF (S) = S_0 (A_1 ($a_{1.1}$, $B_{1.2}$ ($c_{1.2.1}$)), b_2). It is a prefix representation of the derivation tree for GG, as shown in Fig. 1(a). The grammar symbols are attached an index to represent their respective positions, for examples: The index (0) attached to S, defines S as a head and the indexes assigned to A and b indicate that they constitute the first and the second subordinates of S respectively.

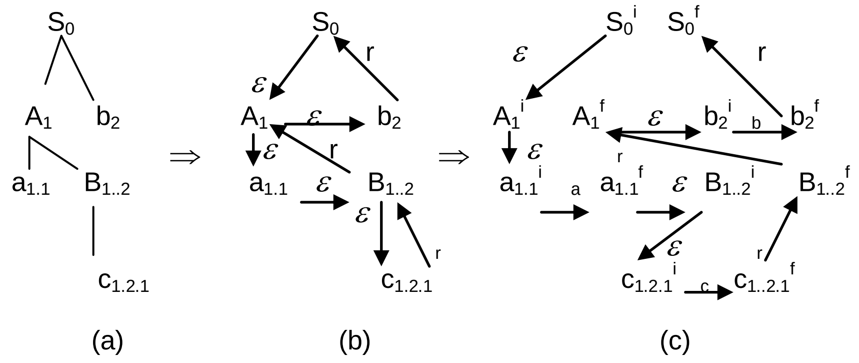

Fig. 1. (a) A derivation tree for the grammar GG of Example1. (b) Derivation/reduction relationships implied by PLF respective to GG. (c) PPA automaton respective to GG.

The derivations/reductions implied by PLF (S) are shown in Fig.1 (b) in terms of the nested paths $S_0 \xrightarrow{\varepsilon} A_1$ **(** $\xrightarrow{\varepsilon}$($a_{1.1} \xrightarrow{\varepsilon} B_{1..2}$ ($\xrightarrow{\varepsilon} c_{1.2.1}$) $\xrightarrow{r} B_{1..2}$)$\xrightarrow{r} A_1$ **)** $\xrightarrow{\varepsilon} b_2 \xrightarrow{r} S_0$, where the outermost path indicates the derivation $S_0 \Rightarrow A_1\ b_2$ and the reduction to S_0. The innermost paths indicate the derivations $A_1 \Rightarrow a_{1.1}\ B_{1.2}$ and $B_{1..2} \Rightarrow c_{1.2.1}$ as well as the reductions to $B_{1..2}$ and A_1. Further, the derivations are performed according to the rightmost derivation first, and the reductions according to the leftmost phrase first. Hence, a bottom-up parsing automaton (PPA), as shown in Fig. 1(c), can be constructed by a direct mapping from PLF(S) as the tuple (T, q_{in}, q_{fin}, Q, SPA, RPA), where:

- T = ($\Sigma \cup \varepsilon$) is the input alphabet, where Σ represent strings generated by the GG grammar.
- Q is the set of PPA states, where q_{in} and q_{fin} are the initial and the final ones.
- SPA and RPA are the sets of shift and reduce parsing actions respectively.

The PPA construction proceeds as follows:

- Each grammar symbol $V \in (\Sigma \cup N)$ is represented by a pair of abstract ones, defined as (V^i, V^f), to indicate a prediction and an acceptance of V in an assumed parsing. Consequently, respective PPA states are defined by the pair ($q^i = V^i$, $q^f = V^f$), where: q^i is an initial state instantiated by V^i and acts as a predictor (scanner) for V. The state q^f is a final one, instantiated by V^f and acts as its respective acceptor. Hence, the PPA states are constructed by the set Q = { ($q_p{}^i = Vp^i$, $q_p{}^f = Vp^f$) | $V_p \in$ PLF (p(S))}. For example, Fig. 2(c) shows a transition graph constructed in terms of a set of nodes instantiated by respective grammar symbols from PLF(p(S)). These nodes constitute the set Q. For simplicity, the symbols (q^i , q^f) and (V^i, V^f) are interchangeably used to denote respective PPA states.
- The derivations/reductions implied by PLF (S) are mapped into respective PPA parsing actions as follows:
  - The ε-transitions are mapped into the set {(σ (q_1, ε) = q_j)}$\in$SPA, for example the ε-transitions $S_0{}^i \xrightarrow{\varepsilon} A_1{}^i$; $b_2{}^f \xrightarrow{r} S_0{}^f$ and $a_{1.1}{}^f \xrightarrow{\varepsilon} B_{1..2}{}^i$ are mapped into σ ($S_0{}^i$, ε) = $A_1{}^i$; σ ($b_2{}^f$ ε) = $S_0{}^f$ and σ ($a_{1.1}{}^f$, ε) = $B_{1..2}{}^i$. Such parsing actions represent the following: ε-transitions from the states instantiated by head grammar symbols to the states of their immediate subordinates; ε-transitions from the states of last subordinates to the states of their respective heads; and ε-transitions from states of grammar symbols to the states of their respective successors (siblings).
  - The derivations of terminal symbols are mapped into the set {(σ (q_1, V) = q_j)} $\in$SPA of respective move-transitions defined between their corresponding pairs of initial and final

states. For example, the derivation of the symbols a and c are mapped into the move-transitions σ (q_1, a) = $a_{1.1}^f$ and σ ($c_{1.2.1}^i$, a) = $c_{1..2.1}^f$.

- The reduce-transitions $\xrightarrow{r}$ indicate reductions to LHSs of the productions for respective nonterminal symbols. Hence, these transitions are mapped by the set {δ (q, V) = reduce (p)| V $\in$ N, q= V^f and V is LHS (p)} of reduce parsing actions for final states of their respective nonterminals. For example, the reduce-transition $b_2^f \xrightarrow{r} S_0^f$ is mapped as the reduce parsing action δ (S_0^f , S) = reduce(S → A b).

Once PPA has been constructed, a subset construction (ε-closure) is then applied on the resulting automaton to obtain a reduced one (RBA), as shown in Fig.2, where:

- The RBA states are constructed as the set Q = { q_0= (S_0^i, A_1^i , $a_{1.1}^i$), q_1 = ($C_{1.21}^i$, $B_{1.2}^i$, $a_{1.1}^f$), q_2 = (b_2^i, A_1^f , $B_{1.2}^f$, $c_{1.2.1}^f$), q_3 = (S_0^f , b_2^f)} of ε-closures
- The RBA parsing actions are constructed as shift parsing actions defined by the set SPA= {σ (q_0, a) = q_1, σ (q_1, c) = q_2, σ (q_2, b) = q_3} and reduce parsing actions defined by the set RPA = {δ (q_2, B) = reduce (B → c), δ (q_2, A) = reduce (A →aB), δ (q_3, S) = reduce (S →Ab)}.
- Finally, the run of RBA on the input acb proceeds according to the transitions $q_0 \xrightarrow{a} q_1 \xrightarrow{c} q_2 \xrightarrow{b} q_3$, where the reductions B → c, A →aB and S →Ab are performed at the states q_2, q_2 and q_3 respectively.

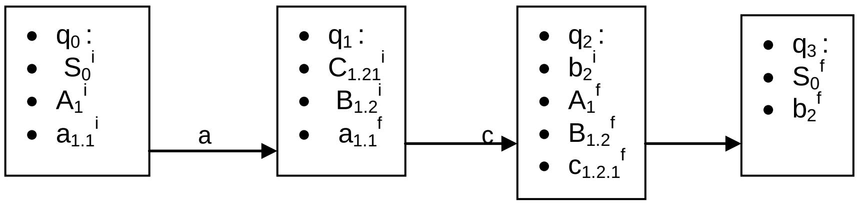

Fig. 2. RBA automaton for the grammar of Example 1

Example 2 demonstrates the proposed parsing approach using a simple context-free grammar. However, such approach handles regular tree grammars and subtle cases such as embedded recursion and grammar productions with different alternatives, as presented in the following sections.

3.1. The production list format (PLF)

Let the 4-tuple (Σ, N, P, S) be a GG grammar, where: (p: A →α) $\in$ P and α= (V1…Vj…Vn) $\in$ (Σ $\cup$ N)*. We inductively define the production list format respective to p as PLF (A) = A (PLF (V1)),…, PLF(Vj),…, PLF(Vn)), where:

- Each nonterminal Vj is replaced by the PLF (Vj) defined for its respective production.

- Each ranked terminal Vj is rewritten by its respective PLF (Vj), defined as PLF (Vj) = (PLF (Vj1),…,PLF (Vji),…,PLF (Vjn)), where: Vj (Vj1, Vji,...,Vjn) $\in$ T($\Sigma \cup$ N).
- PLF(Vj) = Vj, if Vj $\in \Sigma_0$.
- Each grammar symbol in PLF (A) is assigned an index reflecting the position at which it occurs within PLF (A). The index is computed using a function (IND) that is inductively defined as:

  - IND (Vj) = { ε}, if Vj $\in \Sigma_0$.

  - IND (PLF(Vj)) = {0, 1. IND (PLF (V1j)),…,n. IND (PLF (Vnj))}, if Vj $\in$ N and $\exists$ a production (Vj $\rightarrow$ V1j… Vnj) $\in$ P

  - IND (PLF(Vj)) = { ε, 1. IND (PLF (V1j)),…,n. IND (PLF (Vnj))}, if Vj $\in \Sigma$n and Vj(V1j… Vnj) $\in$ T($\Sigma \cup$ N).

To cover the alternative productions and the productions having embedded recursion, the definition for PLF is extended as follows:

- Let (p: A $\rightarrow \alpha$) be a production having recursion, where α= V1…Ai…Vn. The PLF respective to p is defined as PLF (A) = A_{r*0} (PLF (V1),..., A_{r*i} ,…,PLF (Vn)), where the grammar symbol A is expanded by its definition (production) while its respective recursive-occurrence Ai is rewritten, without further expansion (terminalized). In addition, the head A_{r*0} and the recursive-occurrence A_{r*i} are designated by appending the mark (r*) as a prefix to their respective index.
- Let (p: A $\rightarrow$\{ αn\}) be a production with alternatives. We represent p as A$\rightarrow$ A(1) | A(2) |…|A(j)|…| A(n), where A(j) $\rightarrow$αj is the alternative (j). The PLF form respective to p is then defined as PLF (A) = A0 ((PLF (A (1)) |….| PLF (A (n))), where the grammar symbols of different alternatives are designated by the number of the alternative to which they belong. However, they are indexed according to their positions in the designated alternative.

PLF in its extended form constitutes a prefix representation of alternative derivation trees with incorporated recursion. Hence, it implies alternative sequences of derivations and reductions, where the embedded recursion is incorporated as recursion-invocation and recursion-termination. The recursion-invocation constitutes an ε-transition from the recursive occurrence to its respective head, while recursion-termination implies an ε-transition from the recursive-head to its respective occurrence. Hence, the recursive occurrence initiates a derivation/reduction path as the one initiated by its respective head and considered as an instance of the original one**.** To distinguish between instances initiated by different recursive occurrences, an instance-identifier (ID) is assigned to each initiated path.

Example 3. Let GG be a regular tree grammar as given in Example1. A PLF form respective to the start symbol R is defined as

PLF (R) = Rr^*_0 (PLF (R (1)) │ PLF (R (2)) │ PLF (R(3)),where:

PLF (R(1) = Rr^*_0 (1) ($+_1$(1) ($m_{1.1}$(1) ($c_{1.1.1}$(1))**,** $Rr^*_{1.2}$(1)) ;
PLF (R(2) = Rr^*_0 (2) ($+_1$(2) ($m_{1.1}$(2)($Rr^*_{1.1.1}$(2))**,** $Rr^*_{1.2}$(2)) and

PLF (R(3) = Rr*$_0$ (3)(c$_1$(3)).

PLF(R) is a prefix representation of three alternative derivation trees with incorporated recursion, as shown in Fig. 3, where **Rr*$_{1.2}$(1)**, **Rr*$_{1.1.1}$(2)** and **Rr*$_{1..2}$(2)** designate such recursion. Consequently, PLF(R) implies three alternative sequences of derivations and reductions as shown in Fig. 4.

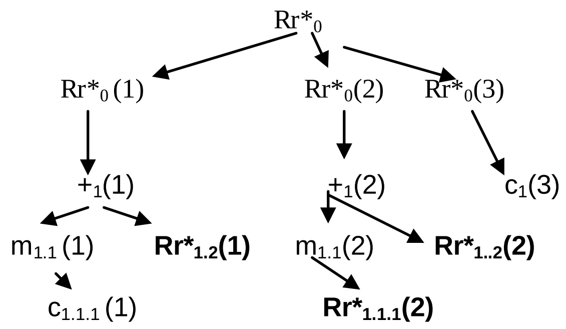

Fig. 3. Derivations trees respective to the grammar of Example 3

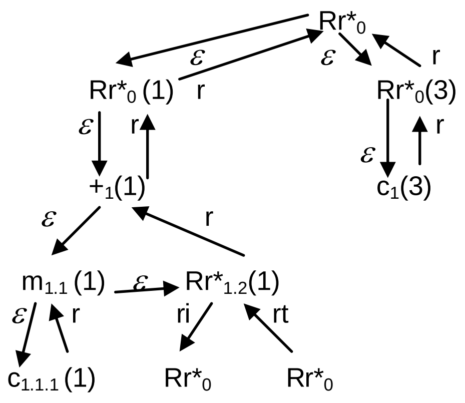

Fig. 4. The derivations/ reductions implied by PLF respective to the grammar (3).

The recursion-invocation is denoted by ri and constitutes the ε-transition from Rr*$_{1.2}$(1) $\xrightarrow{ri}$ Rr*$_0$, while the recursion-termination is denoted by rt and constitutes the ε-transition Rr*$_0$ $\xrightarrow{rt}$Rr*$_{1.2}$(1). Hence, the recursive occurrence (Rr*$_{1.2}$(1)) initiates a derivation/reduction path as the one initiated by its respective head (Rr*$_0$). An instance-identifier (ID) is assigned to the initiated path from Rr*$_{1.2}$(1) as ID= Rr*$_{1.2}$(1), while the original one has ID=ε. The instantiated path ends upon a reduction to the recursive-head, where then recursion-termination is performed. Hence, PLF with incorporated recursion implies initiation and termination of derivations/ reductions paths, as well as respective instances of these paths.

3.2. The position parsing automata (PPA)

PPA is a nondeterministic automaton constructed in terms of a set of states and parsing actions respective to a GG grammar, represented in its expanded list format PLF(S). For example, Fig. 5 shows PPA respective to the grammar of Example 3. As such, PPA is defined as given below based on the following concepts and assumptions for its respective states and parsing actions:

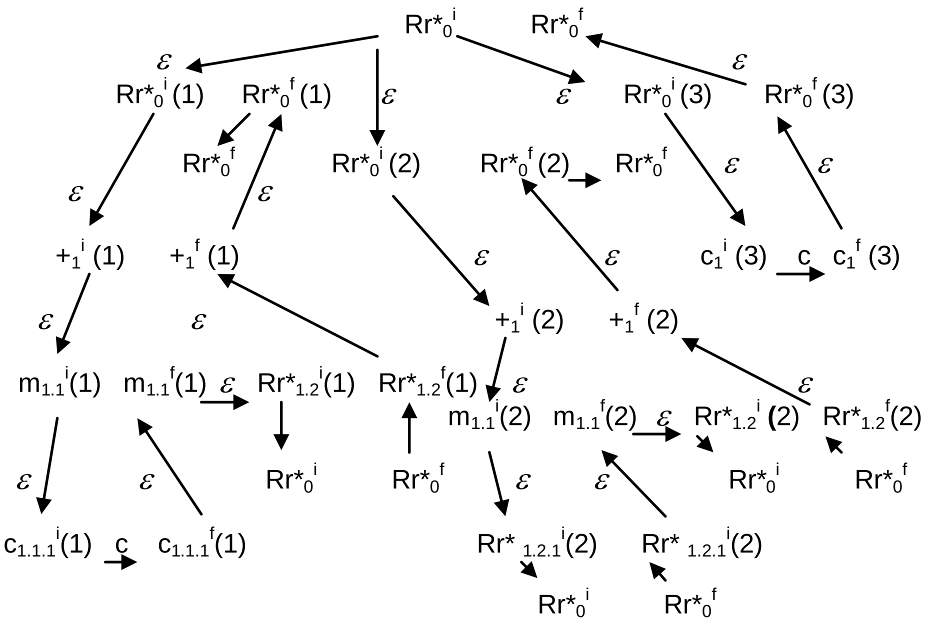

Fig. 5. PPA respective to the grammar of Example 3.

- GG is either instantiated by a context-free grammar or as a regular tree grammar. The set of productions P is generic and is either instantiated with grammar productions or ranked trees (terms). Hence, PPA includes parsing actions respective to the derivations and reductions of ranked trees. The derivations include shifting (reading) the subordinates of their respective ranked terminal. The reductions indicate the completion of such read. Therefore, reading the subordinates of a given ranked terminal are considered as a respective reduction, denoted by coherent-read, for example the string m(c) represents the ranked terminal m. Its respective derivations / reductions as shown in Fig.4 are: $m_{1.1}$ (1) $\xrightarrow{r}$ $c_{1.1.1}$ (1)) $\xrightarrow{r}$ $m_{1.1}$ (1).They include an ε -transition and a reduce-transition which are mapped into the PPA states $m_{1.1}^i(1)$, $m_{1.1}^f(1)$, $c_{1.1.1}^i(1)$ and $c_{1.1.1}^f(1)$ as shown in Fig. 5 with the following parsing actions: σ ($m_{1.1}^i(1)$, ε) = $c_{1.1.1}^i(1)$ $\in$SPA,σ ($c_{1.1.1}^i(1)$, c) = $c_{1.1.1}^f(1)$ $\in$SPA, σ ($c_{1.1.1}^f(1)$, ε) =

$m_{1.1}^f(1) \in SPA$ and $\delta(m_{1.1}^f(1), m)$ = reduce $(m \rightarrow m(c))$, where the subordinate c of m is read, and a reduction to the nonterminalized m is performed. Such reduction is considered as a coherent read of m(c).

- PLF (S) implies instances of alternative derivations/ reductions, distinguished by respective instance identifiers (ID), as explained in Example 3. Therefore, such PLF is mapped into instances of alternative of PPA states and parsing actions. Such mapping proceeds as given in Example 2, where PPA is formulated as the tuple (T, q_{in}, q_{fin}, Q, SPA, RPA). However, it is extended to include instances of alternative states and parsing actions as follows:
  - The set Q = { ($q_p^i = Vp^i$, $q_p^f = Vp^f$) | $V_p \in$ PLF (S)} of PPA states, as given in Example 1, is extended by the concept of a state-instance to have the form Q = {(q_p^i (alt)(inst) = Vp^i(alt)(inst)), (q_p^f(alt)(inst) = Vp^f(alt)(inst))}, where the index (alt) represents the alternative to which each pair of states (q_p^i, q_p^f) belongs and (inst) is an implicit index to represent different instances of the pair. Initially, the PPA states are created with the implicit index (inst) = ε . At run-time, state-instances are created upon initiation of an instance of a derivation/reduction path. Such creation is indicated by appending the instance-identifier (ID) of the initiated derivation/reduction path atop of the implicit index (inst) of its respective states. For example, let the derivations/ reductions implied by PLF (S) be as given in Fig.4. A possible path initiated by the transition $Rr^*_{1.2}(1) \xrightarrow{ri} Rr^*_0$ is: $Rr^*_0 \xrightarrow{ri} R_{r^*0}(3) \xrightarrow{\varepsilon} c_1$ (3) Originally, such path has respective PPA states $R_{r^*0}^i(3)(\varepsilon)$ and c_1^i $(3)(\varepsilon)$ as shown in Fig.5. However, upon the transition $Rr^*_{1.2}(1) \xrightarrow{ri} Rr^*_0$, respective state instances are created as $R_{r^*0}^i(3)((\varepsilon)(Rr^*_{1.2}(1))$ and c_1^i $(3)((\varepsilon)Rr^*_{1.2}(1))$. Since recursion-invocation and recursion-termination establish instances of derivation/reduction paths with nested life-time, the (inst) used as an index for PPA states is organized as a stack structure, onto which an ID is pushed upon recursion-invocation and thereafter popped upon recursion-termination.
  - To handle the initiation and termination of instances of PPA states, the specifications of the parsing actions SPA and RPA are extended by semantic actions which are performed at run time as integral parts of the performed transitions and reductions. Thus, SPA and RBA are specified as SPA = {σ(q_1(alt)(inst), V) = q_2(alt)(inst)) :: { semantic-action} and RPA = { δ (q_1(alt)(inst), V) = reduce(p(V)) :: { semantic-action} respectively. The specified semantic-actions constitute program segments defined in accordance with recursion-invocation and recursion-termination as follows:
    - The recursion-invocation is initiated by a transition of the form: σ(q_i(alt)(inst), V) = (q_{rj}(alt)(inst)), where q_i is a state which immediately precedes the one respective to a recursive-occurrence(q_{rj}(alt)(inst)). The ε–transition σ(q_{rj}(alt)(inst), ε) = (q_{r0j}(alt)(inst)) :: { recursion-initiation } is then performed to the

respective recursive-head (q_{r0}(alt)(inst)), where recursion-initiation is a semantic-action defined as: Top(inst(q_{r0})) = ID (q_{rj}) to push the instance-identifier atop of the implied index (inst) of q_{r0} . Further on, instances of PPA states are subsequently created according to the transitions respective to q_{r0}. Such creation is achieved by augmenting each transition σ (q_i(alt)(inst), V) = q_j(alt)(inst)) $\in$ SPA by an implicit semantic-action defined as Top(inst(q_j)) = Top(inst(q_i)) to propagate the instance-identifier from q_i to q_j. Hence any transition σ (q_{r0}(alt)(inst), V) = q_j(alt)(inst)) $\in$ SPA will propagate ID (q_{rj}) and a respective instance of a derivation/reduction path will be established. For example, and considering Fig. 4, the PPA transition which initiates an instance of the path $Rr^*_{1.2}(1) \xrightarrow{ri} Rr^*_0$ is: σ ($Rr^*_{1.2}(1)(\varepsilon),\varepsilon$) = Rr^*_0 (ε) :: { recursion-initiation} $\in$ SPA, where the ε–transition to $R_{r^*0}{}^{i}$ and { recursion-initiation} are performed. As a result, the instance-identifier $Rr^*_{1.2}(1)$ of the initiated path is pushed atop of (ins) respective to Rr^*_0 (ε). Subsequent transitions in accordance with current input are then performed. These transitions create instances of PPA states respective to the initiated derivations/reductions. Assuming an input (c), the subsequent PPA transitions, are: $R_{r^*0}{}^{i}(\varepsilon)(Rr^*_{1.2}(1)) \xrightarrow{\varepsilon} (R_{r^*0}{}^{i}(3)(\varepsilon)(Rr^*_{1.2}(1)) \xrightarrow{\varepsilon} (c_{1.1.1}{}^{i}(3)(\varepsilon)(Rr^*_{1.2}(1)) \xrightarrow{c}$ $c_{1.1.1}{}^{f}(3)(\varepsilon)(Rr^*_{1.2}(1))$,as shown in Fig.5.

– The recursion-termination is established by a transition from a state which immediately precedes the one respective to final state of a recursive-head, where then an ε–transition is performed to the final state respective recursive-occurrence. Hence, the reduce parsing action respective to the final state q of a recursive-head is extended by semantic-action, denoted by recursion-termination to execute a program segment ({IF (Top (ID (q)) $\neq \varepsilon$) { t = Pop(ID(q)); perform ε–transition to t }) that terminates the initiated path and returns the control (ε–transition) to the final state respective to a recursive-occurrence. For example, and assuming an input (c) the above-illustrated path is terminated by the following sequence of PPA parsing actions: $\sigma(c_{1.1.1}{}^{f}(1)\ (\varepsilon)(Rr^*_{1.2}(1)),\varepsilon)$ = $R_{r^*0}{}^{f}(3)(\varepsilon)(Rr^*_{1.2}(1))$, $\sigma(R_{r^*0}{}^{f}(3)(\varepsilon)\ (Rr^*_{1.2}(1),\ \varepsilon)$ = $R_{r^*0}{}^{f}(\varepsilon)(Rr^*_{1.2}(1))$. Once a transition to $R_{r^*0}{}^{f}$ (3) has taken place, the reduce parsing action δ($R_{r^*0}{}^{f}$ (3),R(3)) = reduce(R$\rightarrow$c)::{recursion-termination} and the augmented semantic-action are performed. As a result, $Rr^*_{1.2}(1)$ is popped from (inst) respective to $R_{r^*0}{}^{f}$ (3), and ε–transition to $Rr^*_{1.2}(1)$ is performed. In addition, (inst) respective to $Rr^*_{1.2}(1)$ is established as ε .

– To handle direct and indirect left recursion, the recursive-heads, the recursive-occurrences and their mutual ε–transitions are designated as cyclic. The cyclic ε–transitions and {recursion-initiation} are not performed. Instead, an instance of the recursive-head is created with respect to the recursive-occurrence as having all the transitions other than the cyclic ones. In addition to its transitions, a cyclic recursion-termination (CRT) is added to the parsing actions of the final states respective to the recursive- heads. CRT constitutes a default ε – transitions to the final states respective to the recursive-occurrences, as Illustrated later by Example 9.

Based on the above assumptions, the definition of PPA is formalized as follows:

Definition 9. (Position Parsing Automaton). Let (Σ, N, P, S) be a GG grammar and PLF(S) be the PLF form respective to the start grammar symbol S. The 5-tuple PPA (p) = (T, Q, q_{in}, q_{fin}, SA, SPA, RPA, CR) constitutes an extended definition for the position parsing automaton (PPA), where:

1. T = ($\Sigma \cup \varepsilon$) and Σ represent strings generated by the GG grammar.
2. q_{in} and q_{fin} are the initial and final states.
3. Q $= \bigcup_{alt=1}^{n}$ (q_p^{i} (alt)(ε) = Vp^{i}(alt)(ε), q_p^{f} (alt)(ε) =Vp^{f}(alt)(ε)) is the set the PPA states respective to the individual grammar symbols Vp in n alternatives of PLF(p(S)). Having a stack-structured index (inst), each PPA state constitutes a run-time nested state-instance created by PPA parsing actions in accordance with a dynamically incorporated recursion. Initially the PPA states are created with (inst) = (ε).
4. SA = {recursion-initiation, instance-propagation, recursion-termination} is a set of semantic-actions that are responsible for initiation, creation and termination of instances of PPA states and transitions with respect to embedded recursion. These actions are embedded within the PPA parsing actions, and executed when such actions are applied during parsing.
5. SPA: σ (q_1(Alt)(inst), V) = q_2 (Alt)(inst),) :: {semantic-action} is a move parsing action that specifies the subsequent PPA state $q_2 \in$ Q for a given state $q_1 \in$ Q and a given grammar symbol V$\in$T. In addition, and whenever the transition is applied during parsing, the transition performs instance-propagation as implied semantic action and {semantic-action}$\in$SA as explicit one, if the transition is augmented with the later.
6. RPA: δ (q(Alt)(inst), V) = reduce(r) :: { semantic-action} is a reduce parsing action that performs a reduction rule (r), for every V $\in$ N , q $\in$Q such that q is the respective state to Vp^{f} and V is the LHS (r). RPA performs the indicated {semantic-action}, If the reduction is associated with such action.
7. CR: λ (q(Alt)(inst),V) = coherent read V (V1…Vn) is a parsing action, defined for every V $\in$ Σ_n and q $\in$Q such that q is the respective state to V^{f}. It represents the completion of the parsing process for the ranked terminal symbol V and its subordinate symbols (V1…Vn).

Based on the presented definition for the PPA, its construction is reduced to a direct mapping from the PLF (S) using the function: M (PLF(S)) → {PPA.Q, PPA. PPA.SPA, PPA.RPA, PPA.CR}, where M (PLF(S)) performs two major steps. In each step, it scans PLF(S) from left to right and considers the grammar symbols of each alternative of PLF(S) according to their occurrence order. However, the first step constructs the set of PPA states (PPA.Q) as union of the ones respective to the start symbol S_{r^*0} and to the grammar symbols of each alternative PLF (P(S(j). Hence, PPA.Q is constructed as the set { ($q_{in} = S_{r*0}{}^{I}$),($q_{fin} = S_{r*0}{}^{f}$), $\bigcup_{alt=1}^{n} \bigcup_{i}$ ((q_i^i(alt)(ε) = Vi^i(j)(ID)), (q_i^f (alt)(ε)=Vi^f(alt)(ε))). The second step establishes the PPA parsing actions for the start symbol S_{r*0} and for each grammar symbol Vi (j) in every alternative PLF (S(j)), according to their order and using Definition 9 as a mapping scheme.

3.3. Soundness and generality of the PPA construction

Let the 4-tuple (Σ, N, P, S) be a GG grammar, where $A \rightarrow \alpha \in$ P and ($\alpha \in$ T(Σ$\cup$N)or $\alpha \in$(Σ$\cup$N)). The constructed automaton PPA = (T, Q, q_{in}, q_{fin}, SPA, RPA, CR) using the mapping function M (PLF (A), as given in section 3.2, constitutes a two fold generic parsing automaton. First, its parsing behavior simulates tree parsing automata and shift–reduce automata, but with reduced stack activities. Second, it parses hybrid strings drawn from a given type of grammar, augmented by definitions from another type of a grammar, for examples:

- A context-free grammar augmented by constructs from regular tree grammar.
- A regular tree grammar extended by context- free grammar constructs.

The validity of the PPA properties and the soundness of its construction approach are demonstrated by the following lemmas.

Lemma 1. PPA constitutes a bottom up parsing automaton that simulates shift – reduce automata.

- **Proof.** Let GG be instantiated by a context-free G. Let a rightmost derivation be (S$\Rightarrow \beta A w \Rightarrow \beta\eta Y w \Rightarrow \beta\eta c w$). In shift-reduce parser, the reductions are performed according to the right most derivations, but in a reverse order. Thus, the reduction to Y is performed followed by one to A. In our approach, the above rightmost derivation is simulated by the following sequence of transitions: $S^i \xrightarrow{\varepsilon} \ldots \rightarrow$First($\beta$)^i $\rightarrow \ldots \rightarrow$ Last (β)^f $\rightarrow A^i \xrightarrow{\varepsilon}$First($\eta$)^I $\rightarrow \ldots \rightarrow$Last($\eta$)^f $\rightarrow Y^i \rightarrow c^i \rightarrow \ldots . \rightarrow c^f \rightarrow Y^f \rightarrow A^f \rightarrow$First(w)^i $\rightarrow \ldots \rightarrow$ Last(w)^f, where Y^f occurs before A^f ,that is, the reduction to Y is performed first. Thus, the reductions performed by our automata are according to the rightmost derivations, but in reverse order. Further more, a handle in a shift-reduce parser is defined as the right side of a production that is formed on the top of the parsing stack [1]. Once, such a handle is formed a reduction is performed. For this purpose, the parser performs the following stack activities: push subsequent terminal (input) symbols, pop parsing stack symbols formed as a handle and push its respective nonterminal. In contrast, the PPA shift activity is defined asε-transition and move transition on a terminal symbol. A handle is formed as a result of a sequence of transitions between the initial and the final states of a respective nonterminal. Subsequently, PPA reduction to a respective nonterminal is performed upon the ε-transition from the final state of the last handle's symbol to the final state of the respective nonterminal, for example, the sequence ($Y^i \xrightarrow{\varepsilon} c^i \xrightarrow{c} c^f \xrightarrow{\varepsilon} Y^f$) forms the handle (c) respective to (Y). Upon the ε-transition from c^f to Y^f_{pi}, the reduction to Y is performed. Hence, PPA simulates a shift-reduce automaton. Furthermore, it parses the regular parts of the language as a finite automaton. However, it behaves as a pushdown automaton by using state instantiation with an instance identifier organized as a stack to handle recursion. □

Lemma 2 PPA constitutes a bottom up parsing automaton that simulates the run of regular tree automata.

Proof. Let GG be instantiated by a regular tree grammar, where Σ_0= (b,x,y) Σ_2= c, (S,A)$\in$ N and P= { S $\rightarrow$c(Ab), A$\rightarrow$xy}. let S$\Rightarrow$cAb$\Rightarrow$cxyb be a GG derivation. Let a bottom-up regular tree automaton be defined by the 4-tuple RA = (Σ, Q, δ, Q^f), where: Σ is the input (ranked) alphabet, Q is a set of automaton states, δ = { Q x Σ_j x Q^j | j $\geq$0} is a set of transitions and Q^f is a set of final states. RA constructs S-derivation tree for cxyb as composed of sub trees constructed according to the following order: A(x,y), c(A,b) and S(c). Such construction is obtained a result of the following transitions: (q_x, x), (q_y, y), (q_A, A, q_x, q_y), (q_b, b), (q_c, c, q_A, q_b) and (q_S, S, q_c). On other hand, the run of PPA on cxyb simulates the same construction order, but according to the following sequence of states transitions: $S^i_0 \rightarrow c^i_1 \rightarrow A^i_{1.1} \rightarrow x^i_{1.1.1} \rightarrow x^f_{1.i.1} \rightarrow y^i_{1.1.2} \rightarrow y^f_{1.1.2} \rightarrow A^f_{1.1} \rightarrow b^i_{1.2} \rightarrow b^f_{1,2} \rightarrow c^f_1 \rightarrow S^f_0$, where: the reductions: A$\rightarrow$xy, c$\rightarrow$Ab (coherent read) and S $\rightarrow$c(Ab) are performed at $A^f_{1.1}$, c^f_1 and S^f_0 respectively. Thus, the reduction order represents a rightmost derivation in reverse order and it is equivalent to the bottom–up construction of the S-derivation tree. Furthermore, RA and PPA have the same interpretation of their transitions with respect to the input cxyb.For example,
the RA transitions (q_x, x) and (q_A, A, q_x, q_y) are considered as the mappings x$\rightarrow$ q_x and (A, q_x, q_y) $\rightarrow$ q_A. In contrast, the respective PPA transitions: $x^i_{1.1.1} \xrightarrow{x} x^f_{1.i.1}$ and $A^i_{1.1} \xrightarrow{x} \ldots \xrightarrow{x} A^f_{1.1}$, with implied reduction A$\rightarrow$xy, are considered as the mappings: x$\rightarrow$ $x^f_{1.i.1}$ and (A, $x^f_{1.i.1}$, $y^f_{1.1.2}$) $\rightarrow$ $A^f_{1.1}$ respectively. □

4. The PPA construction algorithm

Given a GG grammar, PPA (S) is then constructed as a direct mapping from its respective PLF(S), using the mapping function M (PLF(S)) as discussed in section 3. However, such function performs two passes (steps) over a fully expanded PLF(S). During the first pass, the PPA states are constructed, while during second one the PPA parsing actions are constructed. Since PLF(S) is prefix representations of derivation trees respective to GG, its full expansion is equivalent to the construction of such trees. In this section, we propose an alternative but more efficient approach and derive a respective construction algorithm. The proposed approach is a one-pass and based on applying M (PLF(S)) over PLF(S) while it is being expanded in incremental way. For this purpose, we consider a non-expanded form of PLF(S) and a respective but partially constructed PPA as follows:

- Let $S \rightarrow \alpha$, where α = (V1…Vj…Vn) $\in$ ($\Sigma \cup$ N). A non inductive form of PLF(S) is defined as :
- NPLF(S) = S_0 ($V1_{p1}$…Vj_{pj}…Vn_{pn}), where NPLF(S) implies a derivation sub
- tree, denoted PDT (S), with a root labeled by S_0 and children labeled by
- $V1_{p1}$, …,Vj_{pj},…, and Vn_{pn}.
- A partially constructed PPA (PPPA) respective to NPLF(S) is defined as the one that is obtained as a result of applying a modified version of M(NPLF(S)) over PDT(S). Such version is defined as a mapping function M (S_0, Children(PDT(S)) which assumes that the states respective to the root of PDT (S_0) has been created and performs the following:
- Create PPA states respective to the children of PDT($V1_{p1}$…Vj_{pj}…Vn_{pn})
- Establish the parsing actions that cover the transitions between the root and children; the transitions between the children; and the transitions of the states instantiated by recursive occurrences of the grammar symbols.

The construction of PPA is then proceeds according to Algorithm 1 as given below. Assuming an input consisting of the NPLF forms respective to a given GG grammar, Algorithm 1 constructs the PPA(S), using the following data structures and functions:

- The first step of the mapping function is implemented by two functions: CreateNode and CreatePDT to handle the construction of the individual PDTs. The function CreateNode constructs the individual nodes of PDT instantiated with respective grammar symbols. The nodes are indexed by the positions of grammar symbols as they occur within the NPLF forms. The function CreatePDT constructs the PDT respective to a given NPLF form. CreatePDT returns a record of two fields. The first field represents the root of the subtree PDT and the second one represents the children of the PDT as an array of nodes created by the function CreateNode.
- The second step of the mapping function is implemented by two functions: CreateGSPA and by ConstructPPA to handle the construction of the individual PPPA. The function CreateGSPA has a parameter of type Node and returns it's respective initial and final states. Also, CreateGSPA establishes the parsing actions respective to these states, including the ones with recursive occurrences. The function ConsructPPA constructs a PPPA respective to a transmitted PDT as a parameter. ConsructPPA calls the function CreateGSPA to create the states and the parsing actions for the individual nodes of the PDT's children. Then, it establishes the parsing actions that cover the transitions between the root and children; the

transitions between the children; and the transitions of the states, instantiated by recursive occurrences of the grammar symbols.

Algorithm 1

Input: A GG grammar , with start symbol S and its respective NPLF forms

Output : A bottom up automaton PPA = (Q, q_{in}, q_{fin}, SPA, PAR, CR)

Method: An incremental construction of PLF forms, coupled with their gradual transformation into the PPA(S), according to the program given in Fig. 8.

```
  Nod0 = Create-node(S, , 0, S); GSPA = CreateGSPA (Node0);
   PPA.$q_{in}$= GSPA.States[1].initial; PPA.$q_{fin}$ = GSPA.States[1].final;
  For (each alterntive of NPLF(S))
  {PDT = CreateNewPDT(Node0);
    ConstructPPPA( PDT.root, PDT.children)};}
 Set–of–Current-PDT = Set–of–Current-PDT $\cup$ PDT;
 While (Set–of –Current-PDT $\neq \varepsilon$ )
  {Current-PDT-Constructor = Select-next–PDT (Set–of–Current-PDT);
   Set–of–current-PDT = Set–of–Current-PDT \ Current-PDT-Constructor;
   New-PDT-Roots = (Current-PDT-Constructor).ChildrenLevel
   For i to MaxSize (New-PDT-Roots)
    {New-Root= New-PDT-Roots[i];
     For m = 1 to MaxAltenative (New-Root )
      { Alternative-Node = New-Root [m];
       if AlternativeNode is terminal { }
       Elseif { PDT= CreatePDT(AlternativeNode);
          Set–of–Current-PPT = Set–of–Current-PDT $\cup$ PDT ;
          ConstructPPPA( PDT.root, PDT.children); } } }
```

Fig. 6. A program for the construction of the PPA automaton

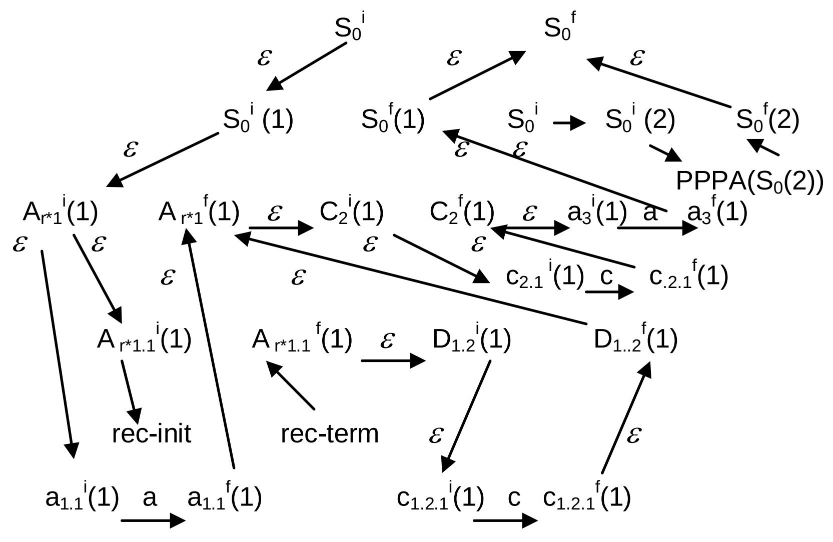

Fig. 7. A partial PPA automaton for the grammar of Example 4

*Example 4***.** Let G = (Σ, N, P, S) be a context free grammar, where: Σ = {a, b}; N= {A, B, C, D}; P= {p1: S→ACa; p2:S→BDb; p3:A→AD; p4:A→a; p5:B→Bc; p6: B→b; p7: C→c p8:D→c. This grammar has direct left recursion and reduce-reduce conflicts. The NPLF forms respective to G are as follows: NPLF(S_0) = S_0 (A_{r*1} (1), C_2(1), a_3(1) | B_{r*1} (1), D_2(1), b_3(1)); NPLF(A_{r*1} (1) = A_{r*0} (A_{r*1} (1) , D_2(1) | a_1(1)); NPLF(C_2 (1) = C_0 (c_1(1); NPLF(B_{r*1} (1)= B_{r*0} (B_{r*1} (1) , C_2(1) | b_1(1)); NPLF(D_2 (1) = D_0 (c_1(1)). Fig.7 shows the transition graph of PPA(S(1)) respective to alternative (1) of NPLF(S_0). Its incremental construction according to algorithm 1 proceeds as follows:

- At steps 1.1 and 1.2, the root PDT is constructed as composed of the single node $Node_0$ = S_0 with ε - transitions to the roots ($Node_1$(1) = S_0(1), $Node_0$(2) = S_0(2) of two alternative PDTs. The respective PPPA automaton is constructed in terms of the following:
- The PPA states: PPA.Q = {(PPA.q $_{in}$ = S_0^i), (PPA.q_{fin} = S_0^f), (q $_0^i$ (1) = S_0^i(1)), (q $_0^f$(1) = S_0^f (1)), (q $_0^i$ (2) = S_0^i (2)), (q $_0^f$(2) = S_0^f (2))}
- The PPA parsing action: PPA.SPA = {(δ (q $_{in}$, ε) = q $_0^i$ (1)),(δ (q $_{in}$, ε) = q $_0^i$ (2)), (δ (q $_{in}$, ε) = q $_0^i$ (3)), (δ (q $_0^f$(1), ε) =.q_{fin}), (δ (q $_0^f$(2), ε) =.q_{fin})}
- The PPA parsing action: δ (q_0^f(1),S)= reduce (r1(S)); δ (q $_0^f$ (2), S) = reduce (r2(R).
- At step 2, The PDT respective to the root $Node_0$ (1) is formed as: $Node_1$(1) = A_{r*1} (1), $Node_2$ (1) = C_2 (1) and $Node_3$ (1) = a (1) respectively. The respective PPPA automaton is constructed as consisting of the following:
- The PPA states: PPA.Q = PPA.Q ∪ { (q $_1^i$ (1) = A_{r*1}^i(1))) ,(q $_2^i$ (2) = C_2^i(1) (2)) ,(q $_3^i$ (1) = a_3^i (1)), (q $_1^f$(1) = A_{r*1}^f(1) (1)), (q_2^f(1) = C_2^f (1)), (q $_3^f$(1)= a_3^f (1)}.
- The parsing actions: PPA.PAS = PPA.PAS ∪ {(σ (q $_0^i$ (1), ε)= (q $_1^i$ (1)) , (σ (q $_1^f$ (1), ε)= (q $_2^i$ (1)), (σ (q $_2^f$ (1), ε)= (q $_3^i$ (1)), (σ (q $_3^i$ (1),a)= q $_3^f$(1)), (σ (q $_3^f$(1), ε)= q $_0^f$(1)).
- During the second iteration, the children of the ($Node_1$(1) and $Node_2$(1), are considered as roots for which subsequent PDTs and PPPA are constructed .This process is iterated until no further PDT can be constructed. As a result, the construction of PPA(S(1)) automaton respective to to alternative (1) of NPLF(S_0), is completed, as given in Fig.7.
- PPA(S(1)) contains the cyclic transition A_{r*1}^i(1) → A $_{r*1.1}^i$(1) → A_{r*1}^i(1). Hence, the transition A_{r*1}^i(1) $\xrightarrow{\varepsilon}$ A $_{r*1.1}^i$(1) and rec-init are freezed, and rec-term (A $_{r*1}^f$(1) $\xrightarrow{\varepsilon}$ A $_{r*1.1}^f$(1)) is considered as a cyclic one.

5. The Subset Construction Algorithm

In this section, we propose a subset construction (ε -closure) for PPA (G). It is an extension to the one for nondeterministic finite automata as given in [1].

Such extension is needed to cover a wider class of grammars including regular tree and context free grammars. The subset construction algorithm, denoted by Algorithm 2 is given below. Having the PPA states PPA.Q and their respective parsing actions (PPA.PAS, PPA.PAR, RBA.CR) as an input, Algorithm 2 constructs a reduced bottom-up automata RBA (G), represented by its respective states RBA.Q and a parsing table PAT. The table PAT is organized as matrix of the form: ParsingAction array [$q_o \ldots q_n$, V1…Vn] , where $q_o \ldots q_n \in$ RPA.Q and V1…Vn $\in \sum$ (the input alphabet of GG). The individual entries of ParsingAction specify the transitions, the reductions and the semantic actions to be made by RBA(G) during its run on an input alphabet, generated from the grammar G. Algorithm 2 computes the RBA(G) states and their respective parsing actions using an ε-closure function [1]. This function has a parameter of type state and returns set of states, constituting the ε-closure of the transmitted parameter. The function closes the initial states and the final states respective to the different grammar symbol types. However, it does not close the initial states instantiated by grammar symbols of type ranked terminals and the grammar symbols of type recursive instances. The steps of proposed algorithm handle their ε-closures, taking into consideration their peculiarities. It is worth mentioning that the ε-closure for the initial states is equivalent to the kernel item in LR parsing, while the one for final states is equivalent to the complete item.

Algorithm 2
Input: A nondeterministic PPA automaton represented by its respective states (PPA. q_{in}, PPA. q_{fin} , PPA.Q) and parsing actions (PPA.PAS, PPA.PAR and PPA.CR).
Output: A reduced bottom-up automata RBA(G) represented by its respective states (RPA.q_{in}, RPA.q_{fin}, RPA.Q) , parsing actions (RPA.PAS, RBA.PAR , RBA.CR) and by a parsing table PAT.
Method: Apply the subset construction on the states of the PPA(G), according to the following steps:
Step0:
- Initially, apply the ε-closure function on PPA. q_{in} to obtain its respective RPA.q_{in}
- Add PPA.qin to the set of RBA (G) states (RBA.Q), marked as unprocessed one.

Step1:
- Select an unprocessed state from the set (RBA.Q).
- Group the alternative states from which the selected state is composed into two classes. The first one includes the initial states instantiated by recursive instances. The second class includes the initial states respective to terminals and ranked terminals.

Step 2: Perform actions respective to each class as follows
2.1: Actions for the class of type recursive instance
- Create new RBA states for each state(s) in the group, instantiated by their respective recursive instances.

– Add the new states (s) to the set RBA.Q, marked as processed ones.
– Add to the table PAT parsing actions of type "move" from the selected state to the new one (s), augmented with semantic action (recursion-initiation).
– Set the parsing actions for the new state (s), as the ones for the state instantiated by the occurrence of its respective head. Compute ε-closure for the PPA state, instantiated by final symbol respective to the new state (s). Create new RBA state, instantiated by such closure. Add the new states to the set RBA.Q, marked as unprocessed.
– The actions respective to recursive instances designated as cyclic are the same as the above. However, their respective ε–transitions and {recursion-initiation} are designated as cyclic.

Table 1. The RBA parsing table for the grammar of Example 5

State	Input Symbols Parsing actions: Move(M), reduce(R) and semantic action (S)		
	+	m	c
q_0	M(q1, q2)		M(q3)
q1		M(q4)	
q2		M(q5), S(rec-initiation)	

q3	R(r3: R $\rightarrow$ c)
	S(rec-termination)
		R(r3: R $\rightarrow$ c)
		S(rec-termination)
			R(r3: R $\rightarrow$ c)
			S(rec-termination)

q4	M(q7 q8), c(m(c))
	S(rec-initiation)		

q5	M(q1, q2)		q3

q6	M(q10),
	S(Initial(q11))
	c(m(R))		

q8	M(q1, q2)		q3

q9	R(r), c(+(m(c,R))
	R(r1:
	R $\rightarrow$ +(m(c,R))
	S(rec-termination)
		R(r), c(+(m(c,R))
		R(r1:R $\rightarrow$ +(m(c,R))R(r),
		S(rec-termination)
			R(r), c(+(m(c,R))
			R(r1:
			R $\rightarrow$ +(m(c,R))R(r)
			S(rec-termination)

q10	M(q1, q2)		q3

q11	R(r), c(+(m(c),R))
	R(r2:
	R $\rightarrow$ +(m(R),R))
	S(rec-termination)
		R(r), c(+(m(c),R))
		R(r2: R $\rightarrow$ +(m(R),R))
		S(rec-termination)
			R(r), c(+(m(c),R))
			R(r2: R $\rightarrow$ +(m(R),R))
			S(rec-termination)

2.2: Actions for the class of type terminals and ranked terminals
– For each state in the class, select its respective parsing actions of type move (transition) as specified by the parsing table PPA.PAS.

- Compute the ε-closures for each destination state as defined by each selected transition.
- Create new RBA states, instantiated by the ε-closures.
- Add the new states to the set RBA.Q, marked as unprocessed ones.
- Add to the table PAT parsing actions "move" respective to the grammar symbol ,instantiating the state ; the selected state ;and the new ones.
- For each alternative of the new states of type "final" , add to the table PAT the respective parsing actions "Reduce or coherent read" as specified by the PPA parsing actions , including the augmented semantic-action, if any.

Example 5 Applying the subset construction on the PPA automaton of Example 3, will produce the RBA automaton, represented by Table 1 as its respective parsing table.

Example 6 Applying the subset construction on the PPA automaton of Example 4, will produce the RBA automaton, represented by its respective parsing table, given as Table 2.

5.1. The PA- Parser

In this section, we propose a parser, denoted PA-Parser, that simulates in pseudo-parallel the run of RBA (G) automaton on input strings, generated by the grammar G. In addition to the input string, the PA-Parser consults a parsing-table PAT respective to RBA(G), as constructed by the subset construction algorithm. The PA-Parser produces alternative parsing paths which represent a bottom-up construction of derivation trees respective to the input string. During paring, these paths are constructed in terms of performed state transitions and reductions as follows:

- The RBA initial state(q_{in}) is considered as the intial derivation. Hence, a respective derivation/reduction path is created as parsing-path$_{ind}$ = q_{in}, where ind =1 indicates the nesting depth of the path.
- For each alternative (J) of a subsequent transition or a recursion termination (q), a continuation for the current parsing-path$_{ind}$ is created as parsing-path$_{ind.j}$ = parsing-path$_{ind}$ $\cup$ q.

Based on the above- mentioned assumptions, PA-Parser is implemented by Algorithm3.

Algorithm 3 The PA- Parser
Input: An input string and the RBA(G) automaton respective to a grammar G and represented by its respective parsing-table PAT.
Output: Successful and erroneous parsing paths represented as a set of respective state transitions and reductions.

Table 2. The RBA parsing table for the grammar of Example 6

State	Input Symbols Parsing actions: Move(M), reduce(R) and semantic action (S)		
	a	c	b
q_0	M(q2); ε "-tran(q_{1r}^{i}) ; S(rec-initiation)"		M(q4); ε "-tran(q_{1r}^{i}) ; S(rec-initiation)"
q_{1r}^{i}	M(q1); ε "-tran(q_{1r}^{i}) ; S(rec-initiation)"		
q_{2r}^{i}		M(q5)	
q2		M(q7); R(r4: A$\rightarrow$a) S(rec-termination)"	
q_{3r}^{i}			M(q4); ε "-tran(q_{3r}^{i}) ; S(rec-initiation)"
q_{3r}^{f}		M(q6)	
q4	R(r6: B$\rightarrow$b) S(rec-ermination)"	M(q8); R(r6: B$\rightarrow$b) S(rec-termination)"	R(r6: B$\rightarrow$b) S(rec-termination)"
q5	R(r8: D$\rightarrow$c); R(r3:A$\rightarrow$AD); S(rec-termination)"	M(q7); R(r8: D$\rightarrow$c);R(r3: A$\rightarrow$AD);S(rec-termination)"	R(r8: D$\rightarrow$c); R(r3: A$\rightarrow$AD);S(rec-termination)"
q6	R(r5:B$\rightarrow$Bc); S(rec-termination)"	M(q8);R(r5:B$\rightarrow$Bc); S(rec-termination)"	R(r5:B$\rightarrow$Bc); S(rec-termination)"
q7	M(q9);R(r7: C$\rightarrow$c));	R(r7:C$\rightarrow$c));	R(r7: C$\rightarrow$c));
q8	R(r8: D$\rightarrow$c)	R(r8: D$\rightarrow$c)	M(q10);R(r8: D$\rightarrow$c)
q9	R(r1: S$\rightarrow$ACa);	R(r1: S$\rightarrow$ACa))	R(r1: S$\rightarrow$ACa))
q10	R(r1: S$\rightarrow$BDb);	R(r1: S$\rightarrow$BDb);	R(r1: S$\rightarrow$BDb);

Method:
Initially, the parsing algorithm considers the initial state of RBA (G) as the current parser state (ind, q_{in} (ε)) as well as the initial parsing path. Each state is considered as having its respective instance-identifier initialized to ε , In addition, and as a transition-state, it is associated with an attribute, denoted by ind, to indicate the parsing path to which it belongs. The algorithm then, iteratively, consults the parsing table PAT entry respective to the pair (current state, current input symbol) and performs the following:

– Determine the subsequent parser states , perform the implicit semantic action for the propagation of the state instance-identifiers and create respective parsing paths

– If the consulted parsing table entry specifies a parsing action reduction, the respective reduction is added to the current parsing path .
– If this entry specifies semantic action of type recursion-termination, the computed return state is considered as a continuation that is added to the set of the parser's next states and to respective parsing path.

Finally, upon reaching the end of the input string, the set { parsing-path$_{ind}$} is produced as an output, consisting of parsing paths in which RBA(G) has reached some of its final states and erroneous ones, otherwise.

Example 7. Considering RBA automaton as given in Example 5, the run PA-Parser on the input + (m(c), c) proceeds as shown in Table 3, where two parsing paths are formulated and produced as an output:
parsing-path$_{1,1.xxx}$ = {(q0, ε), (q1, ε), (q4, ε),(q7, ε), (q8, ε9), CR(m(c), (q3, ε9),(q9, ε),R(r3: R$\rightarrow$c), CR(m(R), R(r3: R$\rightarrow$c),CR(+(m(c),R)), R(r1: R$\rightarrow$+(m(c,R))} and
parsing-path$_{1,2.xxx}$ ={(q0, ε), (q2, ε),(q5, ε6), (q3, ε) (q6, ε), (q10, ε11), R(r3: R$\rightarrow$c), CR(m(R), (q3, ε11) (q11, ε), R(r3: R$\rightarrow$c), CR(+(m(R),R), R(r2: R$\rightarrow$+(m(R),R) }. These paths are equivalent to a bottom-up construction of their respective derivation trees (Fig. 4)

Table 3. The parse of the input +(m (c),c) by PA-Parser for grammar (5)

Current-parser-state	Parsing behavior		
	Current input	Parsing –action-Move(M) Next parser states	Parsing–actions: Reduce(R),Semantic-action (S), Coherent-read(CR)
(q0, ε)	+	{M(q1, ε), M (q2, ε)}	

(q1, ε)
(q2, ε)
	m	M(q4, ε)
		M(q5, ε6)
			
			S(rec-intiation (q6))

(q4, ε)
 	
(q5, ε6)
	c	M(q7, ε) M(q8, ε9)
		M(q3, ε) M(q6, ε) M(q10, ε11)
			CR(m(c), S(rec-intiation (q9))
			R(r3: R$\rightarrow$c), CR(m(R)
			S(rec-termination(q6))
			S(rec-intiation (q11))

(q8, ε9)
 	
(q10, ε11)
	c	M(q3, ε9) M(q9, ε)
		
		M(q3, ε11) M(q11, ε)
			R(r3: R$\rightarrow$c)
			S(rec-termination (q9)
			C(+(m(c,R))
			R(r1: R$\rightarrow$+(m(c,R))
			R(r3: R$\rightarrow$c)
			S(rec-termination (q11)
			C(+(m(R),R))
			R(r2: R$\rightarrow$+(m(R),R))

Example 8. Considering RBA automaton as given in Example 6, the run PA-Parser on the input acca proceeds as shown in Table 4, where three parsing paths are formulated and produced as an output:

parsing-path$_{1,1.xxx}$ = {(q0, ε), (q2, ε), R(r4:A$\rightarrow$a), (q7, ε), R(r7: C$\rightarrow$c) Error}

parsing-path$_{1,2.xxx}$ = {(q0, ε 1r), (q2, ε 1r), R(r4:A$\rightarrow$a),(q1r, ε 1r) ,(q5, ε 1r), R(r8:D$\rightarrow$c),R(r3:A$\rightarrow$AD),(q7, ε 1r), R(r7: C$\rightarrow$c), (q9, ε 1r), R(r1: S$\rightarrow$ACa)}

parsing-path$_{1,2,2.xxx}$={(q0, ε 1r),(q2, ε 1r),R(r4:A$\rightarrow$a),(q1r, ε 1r),(q5, ε 1r), R(r8:D$\rightarrow$c), R(r3: A$\rightarrow$AD), (q1r, ε 1r),(q5, ε 1r), R(r8:D$\rightarrow$c),R(r3:A$\rightarrow$AD), (q1r, ε 1r), error}

Among these parsing paths, parsing-path$_{1,2.xxx}$ constitutes a successful one.

Table 4. The parse of input acca by PA-Parser for the grammar of Example 8

state	Parsing behavior		
	input	Parsing–action-Move Next parser states	Parsing –action- Reduce Semantic-action
(q0, ε) (q0, ε 1r)	a	M(q2, ε) M (q2, ε 1r) M (q1r, ε 1r)	R(r4: A$\rightarrow$a) R(r4:A$\rightarrow$a);S(rec-termination(q1r))"
(q2, ε) (q1r, ε 1r)	c	M (q7, ε) M (q5, ε 1r)) M (q5, ε 1r)) M (q1r, ε 1r)	R(r7: C$\rightarrow$c), R(r8:D$\rightarrow$c), R(r3: A$\rightarrow$AD), R(r8:D$\rightarrow$c), R(r3: A$\rightarrow$AD),S(rec-termination (q1r))"
(q7, ε) (q5, ε 1r) (q1r, ε 1r)	c	Error M(q7, ε 1r) M (q5, ε 1r)) M (q1r, ε 1r)	 R(r7: C$\rightarrow$c), R(r8:D$\rightarrow$c), R(r3: A$\rightarrow$AD),S(rec-termination (q1r))"
(q7, ε 1r) (q1r, ε 1r)	a	M(q9, ε 1r)	R(r1: SAC$\rightarrow$a), R(r8:D$\rightarrow$c), R(r3: A$\rightarrow$AD),S(rec-termination (q1r))"

Example 9. Let G = (Σ, N, P, S) be a grammar with left and embedded recursion. Where (a,+) $\in$ Σ, (E,F)$\in$N and P= { E$\rightarrow$E+F, E$\rightarrow$F, F$\rightarrow$ a, F$\rightarrow$ (E)). Applying the subset construction Algorithm 2 on the PPA automaton respective to the grammar G, will produce the RBA automaton, represented by 12 states and their respective parsing actions. The PA- Parser

behavior on the input a+((a+a)) formulates the following parsing path as an output.
parsing-path$_{1,1.xxx}$ = {(q0, ε), (q1, ε1), (q6, ε1), R(r3: F$\rightarrow$a), R(r2: F$\rightarrow$E, (q2, ε) (q7,ε), (q8, ε), (q9, ε3.1) (q3, ε3.1 1.2) (q6, ε3.1 1.2) R(r3: F$\rightarrow$a), R(r2: F$\rightarrow$E) (q2, ε 3.1 1.2), (q7, ε3.1 1.2) (q10, ε3.1 1.2), R(r3: F$\rightarrow$a), R(r1: E$\rightarrow$E+F), (q5, ε3.1), (q12, ε3.1), R(r4: F$\rightarrow$(E)), (q1o, ε), (q13, ε), R(r4: F$\rightarrow$(E)), R(r1: E$\rightarrow$E+F)}.

6. Discussion

The experiments and the analysis of the derived algorithms for the proposed generic parser have shown the following:
1. The algorithms are characterized by the following calculated complexity:
- The PPA(G) construction algorithm (Algorithm 1) produces O(2G) states and O(2G+(G+1)+G)) transitions, where G = $\sum_{i=1}^{n} | pi |$ is the sum of the length(|*pi*|) of the individual productions. The construction time is O(L*(|N+Σ n|*MAX(|*pi*|, i=1,…n))+ (MAX(|*pi*|, i=1,…n)+1))* ALD$\leq C * \mathrm{G}$, where C is constant reflecting the levels (L) of the grammar's derivation tree and the number of alternative definitions (ALD).
- The PPA (G) subset construction algorithm (Algorithm 2) has a runtime O ((|$\Sigma + \Sigma$ n +Nr|* |ε- Transitions ($\Sigma + \Sigma$ n +Nr)| $\leq$ O (G^2)* s, where s is the number of the PAA reduced states.
- The parsing algorithm requires a time O (| shift-transitions| +|reduce-transitions|)*|input pattern|.
- The size of the parsing table is O (|Σ | * |s|).

To illustrate the above calculated complexity, we consider the grammar of Example 5, the construction of its respective PPA (Fig. 5) and RBA (Table 1) have the following characteristics:
- The size of the grammar G = 12; ALD =3; L=3 ; s = 12; | shift-transitions| = 19; |reduce-transitions|= 18.
- Number of PPA states and transitions (ε-transitions, move, reductions, coherent read) = 61.
- PPA construction (Algorithm 1) time is characterized by O(60) $\leq$ 6* 12, where the dominant operations are ConstructPPPA and CreatePDT.
- The subset construction (Algorithm 2) time is characterized by O(384) $\leq$ 1728, where the dominant operations are EmptyClosure and Add(PAT, parsing actions).
- The parsing table is a matrix of 36 elements. The parse of the input string +(m(c),c) is characterized by O(148), where the dominant operations are the access of the parsing table and the computation of the subsequent states.

1. The PA–Parser has a reduced nondeterministic behavior on an input drawn from ambiguous grammar. The parser generates multiple parsing paths. Since the parser is to be used in code selection, such paths are used to select pattern matches subject to minimization criteria. For example, the output of the parser on input +(m(c),c) drawn from the grammar (5) ,as shown in Table 3, represents two pattern matches; +(m(c), R)) and +(m(R),R)).The pattern with the minimum cost is then selected as the one that matches the input. Thus a pattern matcher can be adapted to the behavior of PA-Parser. However, an opposite approach has been suggested in [8], where PPA has been adopted and tightly coupled with the construction of a general pattern matcher.
2. The PA–Parser has a deterministic behavior on input drawn from non-ambiguous context free grammars. This is demonstrated by examples 2 and 9, where only one parsing path is constructed for the given input.

In addition to its generic behavior, a general comparison of PA-Parser with other bottom-up parsing algorithms such as LR, RI [1, 10] has proved that our algorithm is conceptually simpler and requires less states. The simplicity is achieved based on the fact that our approach is tabular and uses a variation of finite automata and its subset construction. Thus, it features their simplicity, as well as their performance with additional overhead due to the embedded semantic actions. However, a particular comparison with similar approaches is as follows:

– LR parsers require that the input grammar is a deterministic [1]. In contrast, the input grammar for PA-parser is generic which can be either instantiated by regular tree or by deterministic and nondeterministic context-free grammars Further more, parsing the same string by both parsers has shown that the PA-parser has less number of moves (shifts) by 20% than the ones for LR(0) as demonstrated by Example 9 , where the LR(0) [1] automaton for the same grammar consists of 12 states, while our parser consists of 15 states. In addition to the absence of parsing stack activities, no goto transitions on nonterminals are used by our parser. Hence, their pre-computation and run time overheads are eliminated.
– GLR parsers [14] cover nondeterministic context-free grammars by using a graph structured stack constructed at run time to represent in pseudo-parallel multiple parse contexts. In contrast, the proposed parser is based on a nondeterministic predictive automaton, the states and the parsing actions of which represents multiple parse contexts in terms of alternative derivation /reduction paths. At run time, these are regenerated in terms of alternative parsing paths (sequence of transitions and reductions) with respect to an input string. Further more, applying our parsing approach on a pathological example ($S \rightarrow$ SSS |SS | a) as given in [14], a considerable reductions in number of states and transitions (number of visited edges) are achieved. The number of the states is fixed, but they are instantiated. Hence, a trade off is made between a space and parsing time, due to states instantiation.
– Reduction incorporated parsers as introduced in [2] and further optimized in [3] are based on constructing a tier (RIA) that is extended to a pushdown automaton by RCA to handle recursion. In contrast our approach uses a nondeterministic automaton that is augmented by semantic actions to

dynamically create instances of RCA states during parsing. Compared to the tier constructed in [2], PPA has less number of states. Also, Its optimization to RBA produces an automaton with the same size as the optimized version of the pushdown automaton as given in [3]. This is demonstrated by Example 9 using the same grammar given in [3]. Our optimization approach is based on a subset construction (ε-closure). Hence, it is more efficient than the heuristic construction steps given in [3].

– Deterministic pushdown automata have been used to recognize regular tree languages as suggested in [9]. However, such use is based on creating context-free grammar that generates a regular language in postfix form. Such a grammar is in Reversed Griebach Normal Form [9]. In contrast, our approach is based on instantiating a generic grammar (GG) by a regular tree grammar that is then mapped into a recognizing automaton. GG is assumed to be a general context-free grammar and no need to transform the input string into a postfix notation.
– Shift-resolve parser [7] is based on a nondeterministic automation which is then determinized using an approach that generalizes similar construction for LR parsers. Using two stacks, it performs reductions with a pushback down to point where reductions should take place. In contrast, our approach generalizes similar construction for deterministic finite automata. The reductions are performed where they should take place using no parsing stack. The parse of the same string by the shift-resolve, as given in [7] and by our approach, as given in Example 8, shows a reduction in parsing-table size as well as in parsing steps.

7. Conclusion

In this paper, we have proposed and implemented a new parsing approach that is characterized by its soundness, generality and efficiency. The parsing approach is based on an extended version of a recently developed position parsing automaton (PPA). The states and the transitions of the PAA are defined based on concepts from the LR (0) items, the finite deterministic automata and a newly introduced concept of the so called state instantiations. The PAA constitutes a nondeterministic bottom–up automaton that is transformed into a reduced one (RBA) in efficient way. Such automaton simulates the parsing behavior of tree automata as well as the shift-reduce automata. Due to their simplified construction principle, the construction overhead for both PPA and RBA is maintained to a minimum. Considering grammars used by similar approaches, both have been shown as powerful parsing models for ambiguous context-free grammar as well as for regular tree grammars. Although, the considered grammars are not as sophisticated as real languages, they are representative ones. Compared to similar approaches, their respective parsing by the proposed one has produced less parser size and fewer shifts-reduce parsing steps. In fact, RBA is a finite automaton that is dynamically extended to incorporate recursion. Such extension is based on embedded semantic actions to create instances of the RBA states and transitions. Hence, it constitutes an additional overhead during parsing. However, this overhead is reduced due to the instantiation approach. According to such an approach, each RBA state is attached an index and subsequently several state instances can be created and terminated by appending and deleting different instance identifiers atop of the state's attached index. Thus, the space required by state instantiations is minimized and a trade off is made between space, RBA construction and parsing time. As a future work, further experiments well be performed toward achieving more deterministic behavior for the ambiguous grammars at a further reduction of the instantiation cost.

References

1. Aho A.V., Lam M., Sethi R. and Ullman J.D: Compilers Principles, Techniques &Tools, Second edition. Addison Wesley (2007)
2. Ayock J., Horspool R. N.: Faster Generalized LR Parsing. Compiler Construction, LNCS 1575, 32-46 (1999)
3. Ayock J., Horspool R. N., Janousek J., Melichar B.: Even Faster Generalized LR Parsing. Acta Infomatica 37(9), 633-651 (2001)
4. Borchardt B.: Code selection by tree series transducers. LNCS 3317, 57-67 (2005)
5. Cleophas L., Hemerik K. and Zwaan C.: Two related algorithms for root-to frontier tree pattern matching. International Journal of Foundation of Computer Science 17(6),1235-1272 (2006)
6. Ferdenand C., Seidi H., and Wilhelm R.: Tree automata for code selection. Acta Informatica 31(80), 741-760 (1994)
7. Galves J. F., Schmitz S., Farree J.:Shift-Resolve Parsing: Simple, Unbounded Lookahead, Linear Time. Lecture Notes in Computer Science 4094: 253-264 (2006)
8. Jabri S.: Pattern Matching Based on Regular Tree Grammars. International Journal of Electrical
9. Janousek J., Melichar B.: On Regular Tree Language and Deterministic Pushdown Automata. Acta Infomatica 46(7), 533-547 (2009)
10. Johnstone A. and Scott E.: Automatic recursion engineering of reduction incorporated parsers. Science of Computer Programming 68, 95-110 (2007)
11. Katoen J.-P., Nymeyer A.: Pattern-matching algorithm based on terms rewriting systems. Theoretical Computer Science 2378, 237-251 (2000)
12. Madhaven M. et al.: Techniques for Optimal Code Generation. ACM Transaction on Programming Languages and Systems 22(6),972-1000 (2000)
13. Nymeyer A., Katoen J.P.: Code generation based on formal BURS theory and heuristic search. Acta Infomatica 34(8), 597-635 (1997)
14. Scott E. and Johnstone A.: Generalized Bottom- up parsers with reduced stack activity. The Computer Journal 48 (5),565-587 (2005)
15. Shankar P., Ganttati A. , Yuvraj A.R. and Madhaven M.: A new algorithm for linear tree pattern matching. Theoretical Computer Science, 242, 125-142 (2000)

Riad Sadeddeen Jabri. Received the M.E. and Ph.D. in Computer Engineering from Higher Institute for Mechanical and Electrical Engineering/ Bulgaria in 1976 and 1981 respectively. He is currently a Professor and dean of Faculty of Science at Philadelphia University on leave from University of Jordan. His research interests are compilers, formal and programming languages, software systems and networks.

Received: November 9, 2010; Accepted: March 30, 2011.