International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:3, No:4, 2009

Pattern Matching Based on Regular Tree Grammars

Riad S. Jabri

Abstract—Pattern matching based on regular tree grammars have been widely used in many areas of computer science. In this paper, we propose a pattern matcher within the framework of code generation, based on a generic and a formalized approach. According to this approach, parsers for regular tree grammars are adapted to a general pattern matching solution, rather than adapting the pattern matching according to their parsing behavior. Hence, we first formalize the construction of the pattern matches respective to input trees drawn from a regular tree grammar in a form of the so-called match trees. Then, we adopt a recently developed generic parser and tightly couple its parsing behavior with such construction. In addition to its generality, the resulting pattern matcher is characterized by its soundness and efficient implementation. This is demonstrated by the proposed theory and by the derived algorithms for its implementation. A comparison with similar and well-known approaches, such as the ones based on tree automata and LR parsers, has shown that our pattern matcher can be applied to a broader class of grammars, and achieves better approximation of pattern matches in one pass. Furthermore, its use as a machine code selector is characterized by a minimized overhead, due to the balanced distribution of the cost computations into static ones, during parser generation time, and into dynamic ones, during parsing time.

Keywords— Bottom-up automata, Code selection, Pattern matching, Regular tree grammars, Match trees.

I. INTODUCTION

THIS paper introduces a pattern matcher within the framework of code generators. Where, machine instructions are represented by patterns drawn from LR grammars, Regular tree grammars or Rewrite systems [1], [2], [3]. Pattern matching is then performed using parsing automata to generate an optimal set of machine instructions, respective to the intermediate representation (IR) of the front-end compilers [4]. Although LALR parsing is a sound formalism, it is too restrictive to handle the ambiguity of the machine language specifications. On the other hand, the tree parsers are based on generating states for all possible trees. Hence, their minimization is a complex task that might harm the expressive power of the parsing automata. Further more, parsing and pattern matching should be tightly coupled; however, the selection of patterns is performed in a separate pass, in addition, it is associated with an extra overhead due to the computation of the minimization criterion that is based on the instructions costs. Such costs are computed either at parser generation time (static cost computation) or at parsing time (dynamic cost computation). The static cost computations improve the performance of the code selector. However, the dynamic cost computations simplify the tree parsers generators. Thus, there is a need for a pattern matching approach that is based on hybrid concepts from different parsing methods and on a balanced distribution of the time at which the cost computations are performed.

R. S. Jabri is with Philadelphia University, Amman, Jordan, on leave from University of Jordan, Amman,11942, Jordan; phone: (962-6)5355000; fax: (962-6)5355511; e-mail: jabri@ju.edu.jo.

To satisfy such need, we adopt a recently developed generic parsing automaton and follow an approach that adapts the parser to the pattern matcher behavior rather than adapting the pattern matching to the parser behavior. Hence, we first propose a pattern matching approach that is then tightly coupled with a reduced version of the adopted parsing automaton. The proposed pattern matching approach is based on a depth first traversal of an input tree, derived from a regular tree grammar, performing a respective pattern matching to each visited node of such tree subject to cost minimization criteria. This reduces the proposed pattern matching approach to adapting the parsing behavior of the reduced version of the generic automaton to perform pattern matches in a form of match trees, constructed in a bottom up order; and augmented with costs respective to pattern matches. The match tree with the minimum cost is then selected as the one respective to the input tree. We formalized the proposed approach and the parser adaptation methodology, based on which we derived a generic and efficient pattern matching algorithm that is distinguished from similar ones [2], [4] by the following:

- It is based on a different and more generalized parsing approach that covers a broader class of grammars such as context free grammars.
- It synchronizes parsing, pattern matching and code selection in one pass.

The remainder of this paper is organized as follows. Section 2 is a background; it presents a summary of related work and respective preliminaries. In section 3 we first present the proposed pattern matching approach. Then, we present the adopted parsing automaton; and its reduced version, augmented with cost minimization criteria and according to the adaptation methodology. Section 4 presents the theory on which the implementation of the proposed approach is based; followed by the proposed pattern matching algorithm. A discussion and conclusion are given in section 5 and section

6 respectively.

II. BACKGROUND

A. Related Work

Glanville and Graham [5] proposed a method, according to which IR is specified by LR grammar, and a parser generator is used as code generator. Machine instructions are generated as a side effect of parsing. This method was subject of further improvements by Ganapathi, Fisher and others [6]. It was found that LALR formalism is too restrictive to handle the ambiguity of the machine language specifications. Hence, the interest shifted toward bottom-up and top-down tree parsing, combined with dynamic programming either at tree parsing time or at tree parsing generation time, and based on two approaches. According to the first approach, IR and the machine instructions, augmented with costs, are specified by regular tree grammar rules, and a respective tree automaton is generated, as in the code selector generators like Burg [7]. The automaton traverses the input tree (IR), registering each match of the patterns (instruction). Upon traversal completion, the code, corresponding to the derivations with the minimal cost, is generated. The second approach is based on specifying the input trees and the instructions by terms rewriting systems, as in the code selector generators like BURS [8], [9]. A bottom-up traversal of the input tree is performed, labeling its nodes with the respective pattern matches. A top- down traversal is then performed to select the matches, and respectively the code with the least cost. Representative efforts for the first and the second approaches, as well as the hybrid ones are as follows:

- The approach proposed by Ferdinand et al [2]. Where, machine code augmented with cost is represented as a computation of tree automta. A subset construction is then applied to obtain deterministic automata. The transition functions are represented either by k-dimensional array or by decision trees. However, in both cases a table compression is needed in order to reduce the needed space.
- Borchardt [4] proposed tree series transducers. Where, the code selection is reduced to selecting the cheapest derivation of IR, during its parsing and its respective pattern matching.
- Cleophas et al [3] proposed a new version of the Hoffman and O'Donnell [10] tree pattern matching algorithm .Both algorithms are based on top down pattern matching using the string matching automata suggested by Aho and Corasick [11].
- Nymeyer and Katoen [6], [12] formalized BURS theory and proposed an algorithm that computes all pattern matches. A search algorithm, augmented with a cost heuristic, is then applied to perform pattern selection. The algorithm terminates if the rewriting system is finite.
- Boulytchev [13] applied the BURS technique to application specific processors and devised an algorithm that generates both the instruction set and the assembly code from the source program.
- Bravenboer [14] proposed programmable rewriting strategies for instruction selection. Where different strategies, such as inner- most or outer –most, are encoded in a language for program transformation, called Stratego [15]. An appropriate strategy can be selected, augmented with dynamic programming. A similar approach was suggested by Rety [16], where three different algorithms have been proposed according to different reducing strategies. The proposed algorithms can be adapted to different purposes.
- Ertl et al [17] proposed tree parsing automata based on computation of tree parsing states on demand rather than generating the automata at tree parser generation time . The objective is to deal with the finite number of trees that actually do occur, and not with all possible trees.
- Madhaven et al.[18] proposed a method that combines LR(0) parsing technique and the bottom- up tree parsing strategy to develop retagetable, locally optimal code generation with static cost computation. Where ambiguity is handled by deferring the parsing decisions beyond the points where the conflicts actually occurs. However, the method is based on transforming the regular grammar into a constrained context free grammar, where the productions are represented in normal form. Furthermore, the input is transformed into a postfix encoding. In addition, code generation is performed in two passes, the first pass is bottom-up, used to label the subject tree, and the second is top down, to generate the respective code.

Our parsing approach is motivated by the approaches followed by Ferdinand et al [2], Borchardt [4] and by Madhaven et al. [18]. However, it is based on a generic parsing/pattern matching approach that can handle a broader class of grammars, with a balanced distribution of the cost computations into static ones, during the parser generation time, and into dynamic ones, during the parsing time, as embedded semantic actions.

B. Preliminaries

For our further discussions, we assume the following definitions based on the ones given by Madhaven et al. [18] and by Nymeyer and Katoen [6], [12].

Definition 1: (Ranked alphabet). A ranked alphabet is a finite set Σ of terminals with ranking function, called arity. Σ_i denotes the set $\{a \in \Sigma \mid \text{arity}(a) = i, i \in N_0\}$. A tree language (terms) over Σ, denoted by T_Σ is the set T such that:

1) $\Sigma_0 \subseteq T$. If t1,..., tn are in T and a $\in \Sigma$n then a (t_1,..., t_n) $\subseteq$ T.
2) Positions where symbols occur in a (t1,..., tn) are denoted by sequence of doted integers. Such positions are defined by the set:Pos (a(t_1,..., t_n))= $\{\epsilon$,1.Pos(t_1),...,n.Pos(t_n))$\}$.

Definition 2: (Regular tree grammar).The 4-tuple (Σ, N, P, S) defines a regular tree grammar, denoted by G, where:

1) Σ is a ranked alphabet of terminals, N is a finite set of nonterminals and S is a starting symbol.
2) P is a finite set of productions {p}. Each production p has the form: A $\rightarrow$V, where A $\in$ N and V $\in T_{\Sigma \cup N}$.

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 $t_1 \in T_{\Sigma \cup N}$, we say t_1 derives $\in T_{\Sigma \cup N}$ in one step, denoted by $t_1 \Rightarrow t_2$, if LHS(p) $\in t_1$ and RHS(p) $\in t_2$. Applying zero or more such derivation steps on t_1 is denoted by $t_1 \stackrel{*}{\Longrightarrow} t_2$, where each step derives a tree $t \in T_{\Sigma \cup N}$.

We extend the definition of G to a one augmented by minimization criteria. In code selection, such criteria are the computations costs of the instructions modeled by the grammar G.

Definition 3: (Augmented regular tree grammar).The grammar G = (Σ, N, P, S) is an augmented regular tree grammar, denoted by (AG), if each production p $\in$ P has the form p: A $\rightarrow$V, c.Where A $\in$ N, V $\in T_{\Sigma \cup N}$ and c is nonnegative number reflects the computation cost of the instruction modeled by p.

Definition 4: (Regular tree language) The language generated by the grammar G is defined by the set L (G) = {t |t $\in T_\Sigma$ and S $\stackrel{*}{\Longrightarrow}$ t}.

Definition 5: (Regular tree Parsing). Let t L (G) is an input tree, a regular tree parser is the process of constructing a set of possible S–derivation trees for t, defined as follows:

S-Derivation tree for an input tree t $\in$ L (G) is a sub tree with a root that is labeled by S and with children that are labeled by RHS(S). Where:

1) The leaves are labeled by terminal symbols (a $\in \Sigma_0$).
2) The interior nodes are labeled by grammar symbols V $\in$ (N $\cup \Sigma_n$) and constitute roots for V-derivation trees that are recursively defined as the S-derivation tree. However, the children of the grammar symbols V $\in \Sigma_n$ are labeled by ones composing their respective arities.

Definition 6: Patterns, denoted by pn, are defined as the set PN = {(pn $\subseteq$ RHS (p)) | p $\in$ P}

Definition 7: (Pattern matches).Given a grammar G and an input tree t over T_Σ. We say a pattern (pn) matches at position (i) in t, if the following conditions are satisfied:

1) $\exists$ S $\stackrel{*}{\Longrightarrow}$ t (S-derivation tree for t),
2) $\exists$ V $\in$ N such that $\exists$ V-derivation tree $\subseteq$ S-derivation tree for t and
3) $\exists$ V $\Longrightarrow$ pn $\stackrel{*}{\Longrightarrow} t_1$ and t_1 is sub tree rooted at position (i) in t.

III. THE PATTERN MATCHING APPROACH

Let G = (Σ, N, P, S) is an augmented regular tree grammar, where: (p: A$\rightarrow \alpha$, c) $\in$ P and $\alpha \in T_{\Sigma \cup N}$. Let t = a ($t_1$,…, t_n) $\in$ L(G) is an input tree over T_Σ. Let all possible sets of augmented pattern matches respective to t are defined by the superset: APM = { $\bigcup_i$ (apm_j,c (apm_j)) | $apm_j = \bigcup_i$ (pn, i, c) and c (apm_j) = $\sum_i c_i$, for i = ϵ,1.Pos(t_1),…,n.Pos(t_n))}. Where:

1) apm_j is a set of augmented pattern matches respective to t.
2) pn $\in$ PN is a pattern matches at node i $\in$ position (t) according to Definition 7.
3) c is the cost of the pattern matches, defined as the cost of the production rule r (c =c(r)), if pn = RHS(r) and as c=0,if pn $\subset$ RHS (r).
4) c (apm_j) is the cost of the set apm_j, defined as the summation of the costs of production rules respective to the pattern matches from which apm_j is composed.

We define the pattern matcher as a process that first, constructs the super set APM; and then selects a set apm_j $\in$ APM such that its cost c (apm_j) is less than the cost of any other set apm $\in$ APM. To construct such pattern matcher, we proceed according to the following approach:

Each pattern (pn,i, c) $\in$ APM that matches at node i of the input tree is rewritten by the LHS(r) such that pn = RHS (r) and the conditions set by Definition 7 are satisfied. Such rewriting step is represented by the construction of the triple (r, i, c(r)). A set of pattern matches apm respective to the input tree t = a (t_1,…, t_n) is then constructed in a form of a set of such triples, called match tree (mt). Therefore, each augmented match tree is constructed as the pair:

(mt, c(mt)).Where: mt = (r, i, c(r)) and c(mt) =$\sum_i c_i$, for i = ϵ,1.Pos(t_1),…,n.Pos(t_n)).

The construction of pattern matcher is then reduced to the construction of all match trees respective to the input tree t, by executing the program given in Fig.1. Where, the construction of the individual match trees proceeds according to an order that is defined as a depth first traversal of the input tree. During such traversal, augmented triples(r, i, c(r)) are constructing for each visited node, provided its respective children have been traversed in left to right order. Concurrently, the cost of each constructed sub tree is then augmented by the cost(c), defined as an aggregation of the costs of its respective children. The match tree with minimum cost is then selected as the one respective to t. On the other hand, the construction program implicitly simulates the behavior of a bottom-up regular tree parser. Hence, the construction of the match trees can easily be coupled with the parsing behavior of such parser; and in away that satisfies the conditions set by Definition 7.

```
Construct-mt (node i,)
For each child m of node from left to right
{Construct-mt ( m )
mt1 = mt1 ∪ ( r, ,i, c) ; c(mt1) = c(mt1) +c;
.
mtn = mtn ∪ ( r, ,i, c)) ; c(mtn) = c(mtn) +c;
}
```

Fig. 1 The Pattern matching program

As result, the presented approach defines the pattern matcher as a bottom up regular tree parser, coupled with pattern matching. However, the parser should satisfy the following requirements:

- The parser should construct the S-derivation trees at reduced complexity and subsequently at a reduced cost of the computation of its respective states.
- The parser should permit adapting its parsing behavior to the proposed approach for the construction of the pattern matches
- The parser should permit adapting its states and their respective transitions to a framework for computing the costs of the pattern matches.

- The parser should permit performing the following in one pass:

  -Construction of match trees augmented with costs of the pattern matches.

  -Selection of a match tree that is augmented with the minimum cost.

Recently, we have developed a parser that meets the above requirements. In this paper, we adapt such parser to our pattern matching approach. We first present the adopted parsing approach and proceed by adapting its subset construction. We conclude by adapting its parsing configuration, within the framework of a proposed algorithm for pattern matching construction.

A. The Adopted Parsing Approach

The adopted parsing approach is based on simulating the run of a reduced bottom –up automata (RBA) on the input trees generated by a given grammar G. RBA is represented by its states and a parsing table that specifies the parsing actions respective to each input symbol. It is obtained as result of a subset construction applied on a nondeterministic automaton, called position parsing automaton (PPA). PPA is defined as a direct mapping from a list format of the production respective to the starting symbol S of the grammar G. We immediately give the definition of PPA, followed by the description of the parser behavior on the input trees from the grammar G.

B. A Position Parsing Automaton

Let S(V) be the list format ((PLF(S)) for the production, respective to the starting symbol S of a grammar G, where the non recursive occurrences of the nonterminals in PLF(S) are recursively replaced by their respective PLF formats. Thus, each grammar symbol in PLF(S) is expanded by its definition and is indexed by its respective position and nesting depth (pi). For each grammar symbol $V_{pi} \in$ PLF(S), we define two states: q^i_{pi} and $q^{f.}_{pi}$ Where: q^i_{pi} is an initial state, instantiated by the symbol V^i_{pi} and $q^{f.}_{pi}$ is a final state, instantiated by the symbol V^f_{pi}. The initial state q^i_{pi} acts as a predictor (scanner) for $V_{pi.}$ The final state $q^{f.}_{pi}$ acts as an acceptor for (V_{pi}), as well as a predictor for the grammar symbol that follows the symbol (V_{pi}) in PLF (p). Further, we augment each state with two arguments:

- An instance identifier (ID), with an objective to create multiple instances for the same state.
- A cost of a state, defined as $c(q^i_{pi})$ =0 and $c(q^f_{pi})$ = cost (V^f_{pi}).Where cost(V^f_{pi}) =0 if V $\in$ Σ and cost (V^f_{pi}) = cost(r), if V $\in$ N and r is the production rule respective to V.

Based on the presented state concept, the augmented parsing automata for the production PLF (S) is then defined by the 5-tuple PA(p) = (T, Q, $q_{in,}$ q_{fin}, SPA, RPA, CR) Where :

1) T = ($\Sigma \cup \epsilon$) and Σ is the input alphabet of G.
2) Q = { ($(q^i_{pi,ID}$, c) = V^i_{pi} ($q^f_{pi,ID}$, c)= V^f_{pi}) | V_{pi} $\in$ PLF(S)} is set the PA states, instantiated by the respective grammar symbols from PLF(G).
3) (q_{in} = S^i) and (q_{fin} = S^f) are the PPA initial and final states, instantiated by the symbols respective to the starting symbol of the grammar G.
4) SPA: σ($q_{1,ID}$, V) = ($q_{2,ID}$) $\cup$ {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 an input symbol V $\in$ T. It performs an augmented semantic action upon transitions, if the transition is associated with such action. In addition, an implicit semantic is defined to propagate the instance identifier of the state q_1 to the state q_2, by performing the assignment: ID (q_2) = ID (q_1).
5) RPA: δ(q, V) "reduce"(r) $\cup$ { semantic action} is a reduce parsing action that defines a reduction rule (r), for every V $\in$ N, q $\in$ Q such that q has been instantiated by V^f and V is the LHS (r). RPA performs the indicated {semantic action}, If the reduction is associated with such action.
6) CR: λ(q, V) = "coherent read" V(. . .) is a parsing action, defined for every V $\in \Sigma_n$ and q $\in$ Q such that q has been instantiated by V^f. It represents the completion of the parsing process for the ranked terminal V symbol and its subordinate symbols.

The PPA automaton is constructed as a direct mapping from the PLF (S), as demonstrated by the following example:

Example 1: Let G = (Σ, N, P, S), where: Σ_0=b, Σ_2= a, (V, B, G) $\in$ N, S = V and P is given by the following set of rules, augmented with their respective costs.

{r1 : V $\rightarrow$a(V, B), 0 ; r2: V $\rightarrow$a(G, V), 1; r3: V $\rightarrow$G, 1; r4:V $\rightarrow$b, 7 r5: G $\rightarrow$B, 1; r6: B $\rightarrow$b, 4}.

The PLF form respective to the grammar G is defined as:

PLF(V)= r*0({$a_{11} \cup a_{11} \cup G_1 \cup b_1$})({($V_{r*1.1}$,$G_{1.1} \cup B_{1.1}$}, {$B_{1.1.1} \cup b_{1.1.1}$)}({($b_{1.1.1.1}$)},({$B_{1.2} \cup V_{r*1.2}$})(($b_{1.2.1}$)}.

The augmented PPA (V) respective to the PLF(V) is given in Fig. 2. Its construction proceeds as follows:

The PPA states are constructed as:Q ={(($\bigcup_j$ (q^i_{pi}(j) = Vj^i_{pi}),(q^f_{pi}(j) = Vj^f_{pi}))| $Vj_{pi} \in$ PLF ((V)}. Where the grammar symbols from different alternatives (V_j) have the same attached index (pi),and are represented by the PPA states having the same index and the same respective alternatives.

The PPA parsing actions and state transitions for the individual alternative are constructed using the following rules.

- For each terminal symbol $V_{pi} \in$ {b_1, $b_{1.1.1}$, $b_{1.1.1.1}$, $b_{1.2.1}$} in PLF (V), the following move transitions are constructed:

  -(σ (q_1,V) = q_2) $\in$ SPA such that the pair (q_1, q_2) $\in$ Q has been instantiated by pair (V^i_{pi}, V^f_{pi}) respective to V.

  -(σ (q1, ϵ) = q2) $\in$ SPA such that the pair (q1, q2) $\in$ Q has been instantiated by (V^f_{pi}, V^i_{pi+1}), where V^i_{pi+1} is the subsequent symbol to V_{pi} in PLF(V).
- For each ranked terminal and each non terminal { a_1, G_1, $G_{1.1}$, $B_{1.1}$, $B_{1.1.1}$, $B_{1.2}$} in PLF (V) the following transitions and parsing actions are constructed:

  -(σ(q_1,ϵ) = q_2) $\in$ SPA such that the pair (q_1, q_2) $\in$ Q has been instantiated by (V^i_{pi}, $V^i_{pi.1}$), where $V_{pi} \in$ V and $V^i_{pi.1} \in$ (N $\cup\Sigma$) is its first subordinate.

-(σ(q_1,ϵ) = q_2) $\in$ SPA such that the pair (q_1, q_2) $\in$ Q has been instantiated by (V^i_{pi}, $V^i_{pi.n}$), where $V_{pi} \in$ V and $V^i_{pi.n} \in$ (N $\cup\Sigma$) is its last subordinate.
-(q, V) = "reduce"(r) RPA such that q has been instantiated by a nonterminal $V^f_{pi}$and V is the LHS (r).
-(q, V) = "coherent read" V(. . .) CR such that q has been instantiated by a ranked terminal .V^f_{pi}

- For each recursive occurrence $V_{r*pi} \in\{V_{1.1}, V_{1.2}\}$ in PLF (V) the following is performed

  -The states respective to V and V_{pi} are marked as recursive, by appending a prefix (r) to the their respective indexes: ($q^i_{r*0} = V^i_{r*0}$, $q^f_{r*0} = V^f_{r*0}$) and ($q^i_{r*pi} = q^i_{r*pi}$, $q^f_{r*pi} = V^f_{r*pi}$).

$$\begin{bmatrix} q^i_{r*0}(1) \\ V \\ q_{in} \end{bmatrix} \xrightarrow{\epsilon} \begin{bmatrix} q^i_1(1) \\ a \end{bmatrix} \xrightarrow{a} \begin{bmatrix} q^i_{r*1.1}(1) \\ V \end{bmatrix} \xrightarrow{\epsilon} q_{in} \cdots$$
$$q_{fin} \xrightarrow{\epsilon} \begin{bmatrix} q^f_{r*1.1}(1) \\ V \end{bmatrix} \xrightarrow{\epsilon} \begin{bmatrix} q^i_{1.2}(1) \\ B \end{bmatrix} \xrightarrow{\epsilon} \begin{bmatrix} q^i_{1.2.1}(1) \\ b \end{bmatrix} \xrightarrow{\epsilon} \cdot\cdot$$

$$\xrightarrow{\epsilon} \begin{bmatrix} q^f_{1.2.1}(1) \\ b \end{bmatrix} \xrightarrow{\epsilon} \begin{bmatrix} q^f_{1.2}(1) \\ B \end{bmatrix} \xrightarrow{\epsilon} \begin{bmatrix} q^f_1(1) \\ a \end{bmatrix} \xrightarrow{\epsilon} \cdot\cdot$$
$$\xrightarrow{\epsilon} \begin{bmatrix} q^f_{r*0}(1) \\ V \\ q_{fin} \end{bmatrix}$$

(a)

$$\begin{bmatrix} q^i_{r*0}(2) \\ V \\ q_{in} \end{bmatrix} \xrightarrow{\epsilon} \begin{bmatrix} q^i_1(2) \\ a \end{bmatrix} \xrightarrow{a} \begin{bmatrix} q^i_{1.1}(2) \\ G \end{bmatrix} \xrightarrow{\epsilon} \cdots$$

$$\xrightarrow{\epsilon} \begin{bmatrix} q^i_{1.1.1}(2) \\ B \end{bmatrix} \xrightarrow{\epsilon} \begin{bmatrix} q^i_{1.1.1.1}(2) \\ b \end{bmatrix} \xrightarrow{b} \begin{bmatrix} q^f_{1.1.1.1}(2) \\ b \end{bmatrix} \xrightarrow{\epsilon} \cdot\cdot$$

$$\xrightarrow{\epsilon} \begin{bmatrix} q^f_{1.1.1}(2) \\ B \end{bmatrix} \xrightarrow{\epsilon} \begin{bmatrix} q^f_{1.1}(2) \\ G \end{bmatrix} \xrightarrow{\epsilon} \begin{bmatrix} q^i_{r*1.2}(2) \\ V \end{bmatrix} \xrightarrow{\epsilon} q_{in}$$
$$q_{fin} \xrightarrow{\epsilon} \begin{bmatrix} q^f_{r*1.2}(2) \\ V \end{bmatrix} \xrightarrow{\epsilon} \begin{bmatrix} q^f_{r*0}(2) \\ V \\ q_{fin} \end{bmatrix}$$

(b)

$$\begin{bmatrix} q^i_{r*0}(3) \\ V \\ q_{in} \end{bmatrix} \xrightarrow{\epsilon} \begin{bmatrix} q^i_1(3) \\ b \end{bmatrix} \xrightarrow{b} \begin{bmatrix} q^f_1(3) \\ b \end{bmatrix} \xrightarrow{\epsilon} \cdot\cdot$$
$$\xrightarrow{\epsilon} \begin{bmatrix} q^f_{r*0}(3) \\ V \\ q_{fin} \end{bmatrix}$$

(c)

Fig. 2 The PPA automaton for three alternatives of grammar(1)

-A position parsing automaton PPA respective to V_{r*pi} (PPA(V_{r*pi})) is defined as an instance of the one respective to V (PPA (V)). PPA (V_{r*pi}) has the same states and transitions as the ones for PPA(V). However, the transitions of PPA (V_{r*pi}) start from the state $V^i_{r*pi.}$ and terminate at the state $V^f_{r*pi.}$ While the ones for PPA (V) start from V^i_{r*0} and end at V^f_{r*0}.

To facilitate proper initiation, creation, and termination of the transitions respective to such instantiation, the following move transitions and parsing actions are constructed.

- ($\sigma(q^f_{pi-1,ID})$, ϵ) = ($q_{r*pi,ID}$) $\cup$ semantic-action-initialization (V^i_{r*pi})
- $\sigma(q^i_{r*pi,ID}, \epsilon$) = ($q^i_{r*0,ID}$).
- $\sigma(q^f_{r*0,ID})$, V_{r*0}) = "reduce r"(V_{r*0})) $\cup$ (semantic-action- continuation (V^f_{r*0}).
- (($q^f_{r*pi,ID}$), ϵ) = ($q^i_{pi+1,ID}$)

The constructed transitions and parsing actions performs the following respectively:

- The ϵ- transition from the state q^f_{pi-1} instantiated by the final symbol of the grammar symbol immediately occurring before V_{r*pi} in the PLF (V), is augmented by a semantic-action- initialization (V^i_{r*pi}) to be performed during parsing. This action initializes the state identifier ID with the value V_{r*pi} .Since such ID will be propagated upon transitions from q^i_{r*pi}, these transitions are distinguished among others. To enable multiple recursion, ID is organized as a stack where the initialization action is defined to push the value V_{r*pi} on the top of ID.
- The ϵ-transition from q^i_{r*pi} to q^i_{r*0} defines the transitions paths for q^i_{r*pi} as the ones for q^i_{r*0}, but with a propagated instance identifier (ID= q_{r*pi}) instantiated by V_{r*pi}.
- The parsing action (($q^f_{r*0,ID}$), performs reduction of production rule respective to V_{r*0}. In addition, it is augmented with the semantic-action-continuation, defined as function to perform the following during parsing: Computes a move transition to the state at the top of the propagated instance identifier (ID) and then pops the top of ID. The computed transition (or equal to V^f_{r*pi}) is then added to a computed set of the next states of the parser. Thus, during parsing, the continuation semantic action acts as - transition from q^f_{r*0} to q^f_{r*pi} , if the ID is instantiated by V_{r*pi}.
- Finally, an ϵ-transition from $q^f_{r*0}$to q_{pi+1} is costructed.Where, q_{pi+1} is the state instantiated by the grammar symbol subsequent to symbol V^f_{pi*0} in the PLF (V) form.

C. An Augmented Reduced Automata (ARBA)

The reduced automaton RBA for the language generated by the grammar G, with a starting symbol S, is defined as the PPA automaton transformed from the PLF form respective to S and on which the subset construction is applied. In this section, we adapt the subset construction of the position parsing automata by imposing the requirements for its adoption within the framework of the proposed pattern approach. Mainly, we impose the cost computation, according to the following scheme:

- If the constructed RBA state q is not instantiated by recursive instances, then the cost of q is computed as: c (q) =$\sum_j$ c(q_j)+ c (q_k) , where:
  - 1≤j ≤ $|closure(q_j)|$ and closure (q_j) is the set of the states q_j that are closed in the state q as result of the subset construction.
  - q_k is a constructed RBA state with a move transition (q_k, a) = q_i, a ∈ Σ
- If the constructed RBA state q is instantiated by a recursive instance (V^i_{r*pi}), then the cost of q is computed as: c(q) = c (q_k), where q_k is a constructed RBA state with a move transition (q_k, ϵ) = q.
- If the constructed RBA state q is instantiated by a recursive instance (V^f_{r*pi}), then the cost of q is computed as: c(q) = c(q^i_{r*i}) +{c(q^f_{r*0})}. Where: c(q^i_{r*i}) is the cost of the RBA state , instantiated by the subordinate recursive instance (V^i_{r*i}) and c(q^f_{r*0}) is the cost of the RBA state, instantiated by head recursive instance V^f_{r*0}. The cost {c(q^f_{r*0})} is considered as dynamic one that is added, during parsing, to the computed cost c(q) upon a continuation transition to the state q from the state q^f_{r*0}. To facilitate such addition, the computed continuation transition (q^f_{r*0}, ϵ) = q) is augmented by semantic action to add its cost (c(q^f_{r*0})) to the computed cost of q. Such cost is then propagated upon the subsequent move transitions and added to the cost of the destination states.

Algorithm 1: Augmented Subset Construction.

Input: The nondeterministic APPA(S) automaton for a grammar G, represented by its respective states (PPA.q_{in}, PPA.q_{fin} and PPA.Q) and parsing actions (PPA.SPA and PPA.RPA).

Output: An augmented reduced bottom-up automaton ARBA(S), constructed in terms of its respective states (RBA.q_{in}, RBA.q_{fin} and RBA.Q) and parsing table PAT [RBA.Q T] that represents the parsing actions RBA.SPA and RBA.RPA respective to T. Where T is the input alphabet

Method:

Let NewState (states s) is a function that creates a new state q RBA.Q, instantiated by a set of states s PPA.Q; and augmented with two parameters to be used for its respective cost(c) and instance identifier(ID).

Let closure (state q) is a function that returns all the states reachable from the state q on ϵ- transitions, as given in [19]. However, the ϵ-closure function for the initial state q_i that is instantiated by recursive instances retunes the states themselves.

Let AddState (states RBA.Q , state q , processing flag p) is a function that adds a state q to the set of the RBA states either as processed or un processed.Let SetPAT(parsing action p) is function that sets the entry [q,t] of the parsing table PAT to a parsing action as specified by p ,with appropriate encoding respective to the different parsing actions SPA, RPA,(RPA ∪ semantic- action) and (SPA ∪ semantic-action).

Let MoveStates(q) is a structure, used to store the move transitions from the state q.

Algorithm 1 proceeds by executing the program given in Fig. 3, Where:

- The initial state of RBA.q_{in} is computed as the -closure of the it's respective PPA state PPA.q_{in}. It is added unprocessed to the set of the RBA states (RBA.O) for subsequent computation.

{Select next state q from RBA.Q; Mark q as processed
Step1:
for each (q^i_n =V^i) ∈q such that V ∈N andq_n is a marked recursive instance
{ If (q_n is a head){ (REOCC(q_n)) = q)}
Elseif
{AddState (RBA.Q ,(q^i_v = NewState (q_n)), processed);
RBA.q^f_v = NewState(Closure (PPA.Q ,q^f_n));
AddState (RBA.Q , q^f_v, unprocessed);
SetPAT(RBA.SPA ((q,ϵ)= RBA.q ∪ semantic-action- initialization(RBA.q^f_v));
y=RBA.SPA(HeadOccurence (RBA.q^i_v),x)
SetPAT(RBA.SPA((RBA.q,x),y);
MoveStates(q) = ∪{ (PPA.q^f_i, X))}
such that q^f_i,∈ RBA.q^f_v and X ∈ Σ;
c(RBA..q^f_v) = c(RBA.q^i_v)+(c(q^f_i),
∀ q^f_i ∈RBA..q^f_v;
}
Step.2:
For each (q ,V) ∈MoveStates(q);
{ Select next state q_v such that σ(q ,V)= q_v;
RBA.q = NewState(Closure (q_v));
AddState (RBA.Q , RBA.q, unprocessed)
MoveStates (RBA.q)= ∪(σ(PPA.q_j,X)|
∀ j ,(q_j ∈ q_v and X∈ Σ) ;
SetPAT(RBA.SPA (q , V))= RBA.q);
Select subsequent q^f_i,∈ RBA.q |
q^f_i,is with the lowest index j;
c (RBA.q) = c (RBA.q) + c(q^f_i,);
SetPAT(RBA.RPA((RBA.q,V)=
(PPA.RPA(RBA. q^f_i, V)), c (RBA.q));
Select subsequentq^f_i ∈RBA.q in order and perform
c (RBA.q) = c (RBA.q) + c(q^f_i,);
SetPAT (RBA.RPA ((RBA.q, V) =
(PPA.RPA(RBA. q^f_i, V)), c (RBA.q));
}

Fig. 3 The subset construction program

- The algorithm then, iteratively, selects the subsequent unmarked RBA state (q) and performs two major steps, until the processing of all the RBA states is completed. The first step handles the subset construction for the states instantiated by initial symbols (V^i) respective to the recursive instances. It creates new marked RBA.q^i state, instantiated by the initial symbols (V^i), and new unprocessed RBA.qf state, instantiated by their respective final symbols (V^f). Then, it computes the parsing actions, transitions and costs respective to the states

(qv^i) and (qv^f). The second step progressively consider the move transitions (MoveStates(q)) from the selected state q on terminal grammar symbols (V); and handles the computation of the RBA states subsequent to the selected one. This achieved by creating new unprocessed RBA states for each such transition, instantiated by the states computed by their respective ϵ-closure. Finally, it computes their respective transitions, parsing actions and costs.

*Example 2 :*Applying the augmented subset construction algorithm 1 on the PPA automaton for grammar (1), an augmented RBA automaton is obtained, consisting of ten states and a parsing table (PAT), as shown in Table I. The PAT table shows the cost of each state and the its respective parsing actions for the input alphabet (a,b) of the grammar 1 The parsing actions are encoded as follows:

-R(r) represents a reduction using the production rule r, augmented by its respective aggregated cost.
-M(q) represents a move transition to the stateq. M(q1)
-M(q2)represents a move to q1 and immediate transition to q2.
-C(. . .) represents a coherent read.
-S(initialization) and S(Continuation) represent theaugmented transition semantic actions.

TABLE I
THE PARSING TABLE FOR GRAMMAR (1)

.. Parsing actions		
State	Input symbols	
	a	b
q_0	M(q2)Mq4)	M(q1), M(q3)
	S(Initialization(q6))	
q_1	R(r4: V → b),c(r4)=7	R(r4: V →b),c(r4)=7
	S(Continuation)	S(Continuation)
q_2		M(q5)M(q7)
		S(Initial(q9))
q_3	R(r6: B→ b),c(r6)= 4	
	R(r5: G →B), c(r5)= 1	
	R(r3: V → G),c(r3)= 1	
	S(Continuation)	
q_4	M(q2)M(q4)	M(q1), M(q3)
	S(Initial(q6))	
q_5	R(r6: B b), c(r6)= 4	
	R(r5: G B), c(r5)= 1	
q_6	R(r)	R(r) , M(q8)
q_7	M(q2)M(q4)	M(q1), M(q3)
	S(Initial(q6)	
q_8	R(r6: B b), c(r6)= 4	R(r6: B b), c(r6)= 4
	C(a(V,B))	
	R(r1: V→ a(V,B)),c(r1)= 4	
	S(Continuation)	
q_9	R(r), c(r)= 5, C(a(G,V))	
	R(r2: V→ a(G,V)) , c(r2)= 6	
	S(Continuation)	

IV. THE PATTERN MATCHING CONSTRUCTION APPROACH

Our pattern matching approach defines the proposed pattern matcher as a parser that simulates the run of the augmented RBA automaton, as constructed in section 3, on input trees drawn from a regular tree grammar G. Such that the parser configuration is adapted to permit, it's tightly coupling with the construction of augmented pattern matches respective to the input trees in a form of match trees. We first formalize such adaptation by the following theorem and then proceed by giving the proposed pattern matching construction algorithm.

Theorem 1: Let G be a regular grammar and (t) is an input tree with preorder listing. Let RBA is the bottom-up parsing automaton for G as constructed by algorithm 1. Let PA is the parser that simulates the run of the RBA automaton on t. Then PA is tightly coupled with the construction of the set of augmented pattern matches, respective to the input tree t, if each position pi of the preorder listing of the input tree is distinguished by RBA states and a respective set of pattern matches. Such that if the RBA states are defined by set s_q ={q_j | $\delta(q_j, A_j)$ = "reduce A_j", RHS(A_j) =pn_j and r_j is the production rule respective to A_j }, then pattern matches are defined by the set of augmented triples
$\bigcup_l$((r_j, i, c)$_l$, for l=1 to max),where:
- ((r_j, i, c)$_l$ $\subseteq$ match tree mt $_l$and max is the max number of match trees respective to t
-c is the cost of pattern matches, defined as: c= c(q_j).

Proof: If t $\in$L(G) then S $\Longrightarrow$t then the subsequent runs of the RBA on t constitute a bottom-up construction of its respective S-derivation tree. Therefore, if the RBA run respective to the position pi is distinguished by the set s_q of the BA states for witch the reduce transitions { $\delta(q_j, A_j)$ ="reduce" r_j} are defined, then the run is also distinguished by the construction of A_j –derivation trees, such that { A_j } and A_j- derivation trees $\subseteq$ s-derivation tree. Hence, all the conditions given in definition 7 are satisfied ,and each augmented triple (r_j, i, c)$_l$ match tree mt $_l$ represents a subtree (pn_j) rooted at r_j and a parsed subtree of t rooted at position pi, where the pattern pn_j matches. Subsequently, the successful run of RBA on t is a bottom-up construction of S-derivation tree that is tightly- coupled with the construction of the respective match tree. On the other hand, the triple r_j, i, c)$_l$ is augmented by the cost of state q_j that is defined as an aggregation of the costs of its respective match sub trees(mt). This because the run of RBA respective to the position pi in t computes the cost of q_j as: c(q_j)=c(r_j)+ $\sum c(q_n)$, where qn is a state for which the reduce parsing action (q_n, V_n) = "reduce" V_n and $V_n \in$ RHS(r_j) constitutes the pattern respective to an augmented triple(r_n,. . . ,c(q_n)$_l$ $\in$ match tree mt$_l$. ■

Based on our pattern matching approach and its construction methodology; and applying the adaptation theorem, we present the following algorithm for the construction of the proposed pattern matcher. According to the presented algorithm, the run of a bottom up parser on an input tree is distinguished by sequences of states transitions; where each sequence represents a parsing path. We represent such transitions, and subsequently

the parsing paths, as "transition trees". Where, each parser state, during its run, is considered as a root of a transition tree; and the subsequent states, constituting its respective transitions, are considered as the children of such tree. Thus, iteratively, the children constitute the set of current states and are considered roots for transition trees that are constructed in accordance with their respective transitions. A correspondence is then established between the transition trees and the match trees; where for each constructed transition tree a respective match tree is constructed. Thus, as the run of the parser proceeds, such trees are constructed level by level and as a result, a set of mach trees respective to the set of parsing paths is constructed. In parallel, the match trees are augmented by the cost of their respective transitions trees. Such cost is reduced to the costs of the states, representing reductions, as given in their respective entries of the parsing table and propagated dynamic costs. The augmented cost is used as selection criterion by the program given in Figure 5 to generate the match tree mt with the minimum cost.

Algorithm 2: A pattern matcher
Input: An input tree t from a regular tree grammar G,represented in a respective list form.
A parsing table PAT, representing of the ARBA automaton for the grammar G.
Output: A match tree with a minimum cost.
Method:

Let IN[n] = {a1 a2, a3….} is an array of the input symbols respective to the preorder encoding of t . Let PRF [n] = { , 1, 1.1,….}are the respective positions of the input symbols in t.

Let POS (IN [i]) = PRF (IN [n])|i is a function that returns the position in t respective to an input symbol IN [i]

Let q_{in} , {q_{fin}}are the parser initial state and the set of the parser final states respectively.

Let TR PS[i] [m] is a matrix of type transition trees TR to represent the pattern matcher states for the input symbol IN [i]. Where:

- TR is defined as struct {root, children} to represent state transitions from the current state (Root) to the next states (children).
- m represents the alternative state transitions for IN [i].
- children is defined as struct{ set of states s , dynamiccost DC, instance identifier ID}.

Let MT PM [i][m] is a matrix of type mach tree to represent the pattern matches respective to IN [i]. Where:

- MT is defined as struct { rj ,(pn_j, POS (IN [i])}
- m represents the pattern matches respective to the alternative state transitions for IN [i]

Let Parsing-action is a set of type parsing actions, as defined and encoded by the parsing table PAT.

Let the pattern matcher is in its initial configuration, consisting from the initial state qin of the ARBA automaton.

As the individual input symbols are read, the pattern matcher computes the transition trees and their respective match trees, by executing the program given in Fig. 4.The one with a minimum cost is then selected.

The program implementing Algorithm 2 consists of three main parts .the first part is to compute the subsequent transition in a form of transition trees (PS [i] [j] .children, PS [i+1] [j] .children).In addition, it performs the semantic actions related to instance identifier propagation and initialization. The second part constructs a set of match trees by calling the function Construct-Match-trees, given in Fig.5, which synchronizes transition trees and reductions with pattern matches.In addition, it performs the semantic actions respective to continuations, dynamic cost computations and its propagation. Finally, it augments the match trees with their respective costs. The third part of the program determines the pattern matches with the minimum cost. It then regenerates their respective transition trees (parsing path) and match trees.

```
PS [1] [1] .Root = q_in; PS [1] [1] .children = q_in;
MaxAltern =1;
For i = 1 to MaxSize ( IN[])
{ j=0;
For m =1 to MaxAltern
{ q = PS [i] [m] .children
/* Construct transition trees */
Parsing - action = PAT [q, IN[i]];
For each action move-to M(s) in parsing action
{ Perform semantic action (ID);
{If (semantic action initializatin) is in Parsing - action
  {perform semantic- action – initialization (s)}
j= j+1; PS [i+1] [j].Root = m;
PS [i+1] [j] .children=s; MT[i+1] [j]=
Construct- Match- trees (s,i,j);
}
MaxAtern= j; }
/* select a match tree with a minimum cost*/
n = MaxSize ( IN[]);P= Position( MinCost(MT [n]);
SelectedMT=MT[n][p];SelectedTR=PS[n] p].children;
r =PS [n] [p].Root;
For i = n-1 to 1
{t = PS [i] [r].children; SelectedTR= t ∪SelectedTR
m = MT [i] [r]; SelectedMT= m ∪SelectedMT
r =PS [n] [r].Root;}
```

Fig. 4 The pattern matcher

```
Construct- Match-trees (state s, int i,j)
For each reduce action R(r) in PAT[s, IN[i]]
{ If (r is without continuation)
 { cost = c(r) + s.DC;
{If |r| >1{MT [i+1] [j]= MT [i+1][j]∪
    (POS (IN [i]), r, cost)}
Elseif {MT[i+1][j]=(Parent(POS(IN[i]), r, c(r)))}
Elseif (reduce action r is with continuation)
{t=perform semantic- action-continuation(s);
PS [i+1] [j] .children=s t;
cost = c(r) + s.DC+ c(t) ;
{If |r| >1 {MT [i+1] [j] = MT [i+1] [j]∪
(POS (IN [i]), r, cost)}
  Elseif {MT[i+1][j] = (Parent(POS (IN [i]), r,cost))};
```

Fig. 5 The match trees construction program

Example 3: Let G = (Σ, N, P, S) is the grammar 1. Let ARBA is its respective automaton as constructed in example

2. The application of the pattern matcher algorithm (Algorithm 2) on the input tree t = (a(a(b,b),b), derived from G, proceeds as follows:

- The input tree t is represented by its respective preorder encoding IN[ ]={a,a,b,b,b} and the corresponding positions PRF [ ] = { ϵ, 1, 1.1,1.2,2}) of the input symbols.
- As the input is read, the pattern matcher performs sequence of state transitions, coupled with the construction of the respective pattern matches. They are represented in a form of transition trees and match trees as given in Fig.6 and Fig.7 respectively.
- The match tree with the minimal cost and its respective transition tree (parsing path) are then selected. They are: mt = (1.1, r6, 4) ∪ (1.1, r5, 5) ∪ (1.1, r3, 6) ∪ (1.2, r6, 4) ∪(1, r2, 10)) ∪ (2, r6, 4) ∪ (∈, r1, 14)) and tr =q_{in} $\underrightarrow{a}$ { q_2, q_4} $\underrightarrow{a}$ { q_2, q_4} $\underrightarrow{b}$ { q_3, q_6} $\underrightarrow{b}$ { q_8,q_6} $\underrightarrow{b}${ q_8} respectively. The pattern matcher performs such selection because the alternative number 4 of the row_5 ∈ MT (Figure 8) has the least accumulated cost. Such alternative represents the root of a match tree, which is selected by the program, given in Figure 5, based on the fact that each row_i from the matrix TR (Figure 7) consists of alternative children of a transition subtree rooted at row_{i-1}; and the corresponding alternative of the row_i ∈ MT constitutes their respective match trees. For example, the alternative number 4 (3,{q_8, q_6}) of row_5 ∈ TR is a subtree rooted at the alternative number 3 of row_4. Its respective match tree ((1, r2, 10)) ∪ (2, r6, 4)) is the alternative number 4 of row_5 ∈ MT.

TR	Transition trees			
1	q $_{in}$			
2	1,{ q_2, q_4}			
3	1,{ q_5, q_7}	1,{q_3,q_9,q_6}	1,{q_3,q_6}	
4	1,{q_1,q_9,q_6}	1,{q_3,q_9,q_6	2,{q_8, 6}	3,{q_8, q_6}
5	1,{ q_8}	2,{ q_8}	3,{q_8}	4,{q_8}

Fig. 6 The transition trees (TR) for the input a (a (b,b),b)

MT	Matching trees			
3	(1.1,r6,4) (1.1,r5,5))	(1.1,r6,4)	(1.1,r6,4) (1.1,r5,5) 1.1,r3,6)	
4	(1.2,r4,12) (1,r2,13	(1.2,r5,5) (1.2,r6,11) (1,r2,12)	(1,r2,11))	(1.2,r6,4) (1,r2,10)
5	(2,r6,4) (,r1,17)	(2,r6,4) (ϵ,r1,16)	(2,r6,4) (ϵ,r1,15)	(2,r6,4) (ϵ ,r1,14)

Fig. 7 The matching trees (MT) for the input a(a(b,b),b).

V. DISCUSSION

The experiments and the analysis of the derived algorithms for the proposed pattern matcher have shown the following complexity:

The subset construction algorithm (algorithm 1) has a runtime:

O ((|Σ + Σn +N|× | ϵ-Transitions (Σ + Σ_n +N)| ≤O(G^2)×s .Where G = is the sum of the productions length(|pi|) and s is the number of the RBA reduced states

The pattern matching algorithm requires a time O(| shift-transitions| +|reduce-transitions|) ×|input pattern|. The size of the Match trees set is $\leq 2^{|P|\times MAX(|pi|)\times|inputpattern|}$

The size of the parsing table is O (|Σ | × |s|).

Compared with similar pattern matching algorithms, such as the ones proposed by Ferdinand et al [2] and by Madhaven et. al [18], the proposed algorithm is distinguished by the following:

- It can be applied to a broader class of grammars. This is because the adopted PPA automaton can handle context free grammars as well as regular tree grammars; and the pattern matching approach synchronizes the parsing behavior of the automaton in a generic way.
- It achieves better approximation of pattern matches in one pass, using less space and states. This is demonstrated by the fact that the state transitions of the pattern matcher explicitly reflect the positions of the input tree at which the pattern matches occur. For example, the sequence of state transitions for the input of example 2 at which pattern matches occur is{ q_3, q_6}{ q_8, q_6}{ q_8}.These states are instantiated by the PPA automaton closed states ($q^f_{1.1.1}$ $q^f_{1.1}$ $q^f_{1.2}$ q^f_1) that constitute the reductions respective to the pattern matches. On the other hand, pattern matches occur at the following positions of the input tree: 1.1, 1.2, 1, 2 and ϵ. These positions are directly correlated to the indexes, attached to the reduced states. Such correlation is a two fold. First, it implies accurate pattern matches with optimized number of state transitions. Second, it permits performing parsing, pattern matching and selection in one pass. In addition, applying the proposed algorithm on test samples given in [2], [18], such as the grammar of example 1, has shown not less than 20% reductions in states and tables size. Finally, the algorithms proposed in [2], [18] perform static cost computation while the proposed one performs static and dynamic cost computation in a balanced way.

VI. CONCLUSION

In this paper, we have proposed and implemented a pattern matching approach that is characterized by its soundness, generality and efficiency. The resulting pattern matcher operates like a bottom –up automaton that is tightly coupled with the construction and the selection of pattern matches subject to minimization criteria. Since the adopted parser and the proposed pattern matching approach are generic ones, the resulting pattern matcher handles a wider class of grammars other than the regular tree grammars. Thus, the pattern matcher can be used in areas other than code selection. Compared with similar pattern matching algorithms, we have achieved better approximation of pattern matches in one pass with considerable reductions of the automaton states and the size of the pattern matching /parsing tables. As a future work, further experiments well be performed on the code selection for real machines and on the application of the proposed algorithm and in areas other than code selection.

ACKNOWLEDGEMENT

This work has been carried out during sabbatical leave granted to the author (R. Jabri) from the University of Jordan during the academic year 2007/2008.

REFERENCES

[1] Shankar P., Ganttati A., Yuvraj A.R. and Madhaven M. (2000), A new algorithm for linear tree pattern matching, Theoretical Computer Science, 242, 125-142.
[2] Ferdenand C., Seidi H., and Wilhelm R. (1994), Tree automata for code selection. Acta nformatica , 31(80):741-760.
[3] Cleophas L., Hemerik K. and Zwaan C. (2006), Two related algorithms for root-to frontier tree pattern matching, International Journal of Foundation of Computer Science, 17(6),1235-1272.
[4] Borchardt B., Code selection by tree series transducers (2005), LNCS, 3317, 57-67.
[5] Glanville R. S.,. Graham S.L. (1978), A new method for compiler code generation, Proc.of the fifth Ann. ACM Symp. On Principles of Programming Languages, 231-240.
[6] Nymeyer A., Katoen J.P. (1997), Code generation based on formal BURS theory and heuristic search. Acta Infomatica , 34(8),597-635.
[7] Fraser C.W., Henry R.R. and Proebsting T.A. (1992), BURG-fast optimal instruction selection and tree parsing, ACM SIGPLAN Notices 27(4), 68-76.
[8] Llopart P. and Graham E. (1988), Optimal code generation for expression trees: An application of BURS theory. Proc. Of the Fifteen Ann. ACM symp. On Principles of Programming Languages , 294-308.
[9] Proebsting T.A. (1995), BURS automata generation, ACM Trans. On Prog. Lang. and Sys.3(17), 461-486.
[10] Hoffmanand C.M. and O'Donnell M.J. (1982), Pattern matching in trees, Journal of the ACM, 29(1),68-95
[11] Ahoand A.V. and Corasick M.J. (1975), Efficient string matching: an aid to bibliographic search, Communications of the ACM, 18, 333-340.
[12] Katoen J.-P., Nymeyer A. (2000), Pattern- matching algorithm based on terms rewriting systems, Theoretical Computer Science,238(1-20, 439-464.
[13] Boulytchev D. (2007) Burs-based instruction set selection, LNCS, 4378,431-437.
[14] Bravenboer M. and Visser E. (2002), Rewriting strategies for Instruction selection, LNCS, 2378, , 237-251.
[15] Visser E. (2001), Stratego: A language for program transformation based on rewriting strategies, LNCS, 2051 , 357-361.
[16] Rety P., Vuotto J. (2005), Tree automata for rewrite strategies, Journal of Symbolic Computation ,40 , 749-794.
[17] Ertl M.A., Casey K and Gregg D. (2006), Fast and flexible instruction selection with on –demand tree-parsing automata, In PLDI'06, Ottawa, Canada , 52-59.
[18] Madhaven M. et al. (2000), Techniques for Optimal Code Generation, ACM Transaction on Programming Languages and Systems , 22(6),972-1000.
[19] Aho A.V., Lam M., Sethi R. and Ullman J.D. (2007), Compilers Principles, Techniques, &Tools, Second edition, Addison Wesely.