The Parsing of Extended Context-Free Grammars

Anne Brüggemann-Klein[1] and Derick Wood[2]

[1] Institut für Informatik, Technische Universität München, Arcisstr. 21, 80290 München, Germany. brueggem@informatik.tu-muenchen.de
[2] Department of Computer Science, Hong Kong University of Science & Technology, Clear Water Bay, Kowloon, Hong Kong SAR. dwood@cs.ust.hk

Abstract. Extended context-free grammars are context-free grammars in which the right-hand sides of productions are allowed to be any regular language rather than being restricted to be any finite language. We develop a novel approach to top-down predictive parser construction for extended context-free grammars that is based on rewriting of partial syntax trees.

This work is motivated by our development of ECFG, a Java toolkit for the manipulation of extended context-free grammars, and by our continuing investigation of XML.

1 Introduction

We have been investigating SGML [9, 7] and XML [1], and their properties and use for a number of years [21, 2–5, 10]. One major obstacle in the development of XML applications has been that current XML parsers are black boxes that limit and hinder user customization of parser output. For this reason we have designed and are currently implementing a more transparent approach to XML-parser construction.

Our strategy is not surprising—we use the traditional programming-language approach, as exemplified by `lex`, `flex`, `bison` and `yacc`, to generate parsers automatically from grammars. That this has not been done for SGML DTDs or for XML DTDs is because of the lack of transparency of their definitions and because it is unclear whether programming-language techniques can be used at all.

We present, however, a novel approach to generate **eLL(1) parsers** for XML documents from XML DTDs, where eLL(1) parsers are an extension of LL(1) parsers (for context-free grammars) for extended context-free grammars. The approach is based on what we call partial syntax trees. The parser generator is intended to be one tool among many in our tool kit for extended context-free grammars that we call ECFG. In previous work [3], we characterized the rhs-languages of a DTD as regular languages that are exactly the one-deterministic (or unambiguous) regular languages. We make use of this property to generate parsers from eLL(1) extended context-free grammars and from XML DTDs.

LaLonde [11] appears to have been the first person to seriously consider the construction of parsers for extended context-free grammars. The construction of LL(1)-like parsers for extended context-free grammars has been discussed by Heckmann [8], by Lewi and his coworkers [12–15], and by Sippu and Soisalon-Soininen [16, 17] and by Wilhelm and Maurer [20]. Warmer and his coworkers [18, 19] and Clark [6] developed parsers for SGML.

We first define the basic notation and terminology that we need, in Section 2, before introducing partial syntax trees, in Section 3. We then, in Section 4, describe an algorithm eNLL to compute a leftmost construction. To complete this approach we demonstrate how to generate a table-driven version of eNLL, in Section 5, before characterizing LL(1) grammars in terms of eLL(1) extended grammars, in Section 6.

2 Notation and terminology

An **extended context-free grammar** G is specified by a tuple of the form (N, Σ, P, S), where N is a nonterminal alphabet, Σ is a terminal alphabet, P is a set of **production schemas** of the form $A \longrightarrow \mathrm{L}_A$, such that A is a nonterminal and L_A is a regular language over the alphabet $\Sigma \cup N$, and S, the sentence symbol, is a nonterminal. Given a production schema $A \longrightarrow \mathrm{L}_A$ such that α is a string in L_A, we say that $A \longrightarrow \alpha$ is a **production** of G. We call L_A the **rhs-language** of A. We may assume that the language L_A is represented by a nondeterministic finite-state automaton $\mathrm{M}_A = (Q_A, \Sigma \cup N, \delta_A, s_A, F_A)$, where Q_A is a finite set of states, δ_A is the transition relation, s_A is the start state and F_A is a set of final states.

For a grammar $G = (N, \Sigma, P, S)$, a nonterminal symbol A in N gives rise to two languages, namely:

1. The rhs-language L_A over $N \cup \Sigma$, that is assigned to A via P.
2. The language of all Σ-strings that can be derived from A using productions in G; as usual, we call this language the **language** of A and denote it by $\mathrm{L}(A)$.

More generally, if α is a string over $N \cup \Sigma$, then the **language** $\mathrm{L}(\alpha)$ of α is the set of all terminal strings that can be derived from α using productions in G. In particular, $\mathrm{L}(S)$ is the **language $L(G)$ of grammar G.**

3 Partial syntax trees

Let $G = (N, \Sigma, P, S)$ be an extended context-free grammar. We assume that the rhs-language of a nonterminal A is specified by a finite-state automaton M_A.

We will base our approach to parsing on the use of **partial syntax trees** whose nodes are labeled with symbols in $\Sigma \cup N$. Internal nodes are always labeled with nonterminal symbols from N; external nodes are labeled with symbols from either alphabet. A **nonterminal node** of a partial syntax tree is labeled with a nonterminal symbol from N, whereas a **terminal node** is labeled with a symbol from Σ. Nonterminal nodes that have an annotation are **active nodes** and the remaining nonterminal and terminal nodes are **inactive nodes.**

Note that we do not allow an external node to be labeled with the empty string λ. We avoid the explicit use of the empty string in a partial syntax tree; instead we represent it implicitly as we explain later.

A **partial syntax tree** for G is a partial tree with nonterminal alphabet N whose active nodes are annotated with states in their labels' automata. The root of a partial syntax tree must be labeled with G's sentence symbol S. Furthermore, the labels and annotations in the tree must conform to the grammar in the following way:

- For each inactive node u, the labels of the children of u must spell out a string in the rhs-language of u's label.
- For each active node u, its state is determined by the input string formed by the sequence of labels of u's children.

In particular, each external active node u must be annotated with the start state of u's automaton and the language of an external inactive node's label must contain the empty string.

We are particularly interested in **leftmost partial syntax trees** that are defined as partial syntax trees in which the active nodes form a prefix of the rightmost branch of the tree.

The frontier of a leftmost partial syntax tree is a sequence, from left to right, of (terminal and nonterminal) inactive nodes, followed by at most one active node. The sequence of inactive nodes in the frontier yields a terminal string over Σ, which we call the **partial yield** of the leftmost partial syntax tree.

XXXcomment on inactive external nodes that are nonterminal - empty strings.

A partial syntax tree is a **syntax tree** if it has no active nodes. The **yield** of a syntax tree is the string formed from the labels of the *terminal nodes* in its frontier when read from left to right.

Observation 1 *The yield of every syntax tree of an extended context-free grammar G is a string in $\mathrm{L}(G)$ and, conversely, every string in $\mathrm{L}(G)$ is the yield of a syntax tree of G.*

A partial syntax tree of a grammar G can be constructed incrementally by beginning with an initial one-node partial syntax tree and then, step by step, adding nodes, changing the states that are associated with nodes and making active nodes inactive. We can view this process as a sequence of transformations on partial syntax trees. A single transformation manipulates, nondeterministically, an *active node* v of a partial syntax tree in one of the following three ways, where we assume that v is labeled with a nonterminal A and is annotated with a state p of M_A:

1. If p is a final state of M_A, then v is made inactive by removing the state p from v.
2. If M_A has a transition from p to p' on a terminal a, then a new node v' is added as the new rightmost sibling of v's children. The new node is labeled with a and v is annotated with state p'.
3. If M_A has a transition on a nonterminal B from p to p', then a new active node v' is added as the new rightmost sibling of v's children. The new node is labeled with B and is annotated with the start state of M_B. Futhermore, v is annotated with state p'.

A **construction** of a partial syntax tree is a sequence of transformations of one of these three types that begins with the initial partial syntax tree whose one node is active, is labeled with the sentence symbol S and is annotated with the start state of M_S; see Figure **??**.

A **leftmost construction** is a construction that begins with the initial partial syntax tree and consists of only leftmost partial syntax trees. (Note that the initial partial syntax tree and all syntax trees are leftmost.)

Observation 2 *A leftmost construction manipulates the partial syntax trees at their deepest active nodes at each step.*

Proof. We argue by contradiction. Assume that v is an active node in a leftmost partial syntax tree that has an active proper descendant. Then, v must also have an active child, since the active nodes of a leftmost partial syntax tree form a prefix of the tree's rightmost branch. We demonstrate that each of the three types of transformation steps, when applied to v, result in a partial syntax tree that is no longer leftmost.

First, if node v is made inactive, then the resulting partial syntax tree is no longer leftmost since the *inactive node* v has an active child.

Second, if a new node labeled with a terminal symbol is added as the new rightmost child of v, then v has, in the new partial syntax tree, an active child that has a right sibling. Hence, the resulting partial syntax tree is no longer leftmost.

Finally, if a new active node is added as a rightmost child of v, then among the children of v there are at least two active siblings. Hence, the resulting partial syntax tree is not leftmost.

Observation 3 *For each leftmost partial syntax tree, there is a leftmost construction.*

Observation 4 *For every string in* $\mathrm{L}(G)$*, there is a leftmost construction.*

Proof. This observation follows from Observations 1 and 3 since syntax trees are trivially leftmost.

Observation 5 *The frontier of a leftmost partial syntax tree is a sequence of (terminal or nonterminal) inactive nodes followed by at most one active node.*

Proof. An active node that is not rightmost in the frontier must have either a right sibling or an active ancestor that has a right sibling.

The labels of the terminal nodes in the frontier of a leftmost partial syntax tree form the **terminal prefix string** of the tree.

Observation 6 *Any transformation step in a leftmost construction adds at most one terminal symbol to the partial yield.*

Proof. The transformation steps of types 1 and 3 leave the partial yield unchanged; a transformation step of type 2 appends one terminal symbol.

A leftmost partial syntax tree is **compatible** with a given terminal string if its partial yield is a prefix of the terminal string.

From here on in, we assume that a grammar $G = (N, \Sigma, P, S)$ is **normalized** in the following sense:

- Σ has an endmarker symbol $\dashv$ such that:
  - L_S is a subset of $((N \cup \Sigma) \setminus \{\dashv\})^* \dashv$
  - M_S has exactly one final state and no out transitions from this state
  - Only the finite-state automaton M_S of S has a transition on $\dashv$.
  - No M_A has a transition on S

  These conditions imply that $\mathrm{L}(G)$ is a subset of $(\Sigma \setminus \{\dashv\})^* \dashv$.
- For each A in N, the language $\mathrm{L}(A)$ is neither the empty language nor the empty-string language.
- For each A in N, there is a partial syntax tree of G that has a node labeled with A
- In each M_A, each state is reachable from the start state and, from each state, some final state is reachable; that is, M_A is reduced.

To summarize, a grammar is normalized if it has an endmarker and every nonterminal and every state is reachable and useful.

4 Predictive parsing

We present a nondeterministic algorithm eNLL to compute a leftmost construction in which each leftmost partial syntax tree is compatible with a given input string $a_1 \cdots a_n$ of terminals. When given a leftmost partial syntax tree that is compatible with the input string, the algorithm expands the tree's deepest active node using one of the three transformation types given in Section **??**.

When choosing a transformation type, the algorithm eNLL takes into account the input symbol immediately to the right of the partial syntax tree's partial yield. We call this symbol the **current input symbol.** If the algorithm detects that there is no current input symbol, then the current input symbol is defined to be the empty string λ. Furthermore, we consider only input strings that end with the endmarker symbol $\dashv$.

We need to introduce a number of concepts to describe the behaviour of eNLL. First, let $\mathrm{L}(A, p, X, p')$ be the language of all strings over Σ that are the yields of syntax trees constructed from the one-node initial tree with root label A and root state p, and whose first transformation is to add a child labeled X to the root and to move the root label's automaton into state p'. Formally, $\mathrm{L}(A, p, X, p')$ is the union of all languages $\mathrm{L}(X\alpha)$, where $\alpha \in (N \cup \Sigma)^*$, such that M_A has a transition from p to p' on X and α in $(N \cup \Sigma)^*$ moves the state of M_A from p' into a final state.

Then, *follow*(A) is the set of all terminals that can occur immediately after A in a string that G can derive from S. More formally, *follow*(A) consists of all a in Σ such that G derives $\alpha A a \beta$ from S, for some α and β in $(N \cup \sigma)^*$. In addition, we add λ to *follow*(S).

Next, we consider every leftmost partial syntax tree whose deepest active node v is labeled A and is annotated with state p, and any construction whose first transformation step adds a new rightmost child of v that is labeled X and moves M_A into state p'. The first terminals that are derived by such constructions form the set *first*(A, p, X, p'). To be precise, *first*(A, p, X, p') consists of the first symbols of the strings in $\mathrm{L}(A, p, X, p')$ *follow*(A).

The set *first*(A, p, X) is the union of all sets *first*(A, p, X, p').

Finally, *transit*(A, p) consists of all grammar symbols X in $N \cup \Sigma$ such that M_A has a transition on X from p.

Observation 7 *follow*$(S) = \{\lambda\}$*; follow*(A) *is never empty.*

Proof. Because grammar G is normalized, no M_A has a transition on S. Therefore, if G derives some string α from S, then either $\alpha = S$ or S does not occur in α. Hence, *follow*$(S) \cap \Sigma = \emptyset$; that is, *follow*$(S) = \{\lambda\}$.

If A is different from S and G derives some string $\alpha\, A\, \beta$ from S, then $\alpha\, A\, \beta \neq S$. Now, the last symbol of β is $\dashv$ since G is normalized. Hence, β derives a nonempty terminal string, whose first symbol is in *follow*(A).

Observation 8 *first*(A, p, a, p') *is the singleton set* $\{a\}$ *if* M_A *has a transition on* a *from* p *to* p'*; otherwise, it is the empty set.*

Observation 9 *If* M_A *has a transition on* B *from some state* p*, then first*(A, p, B) *is nonempty.*

Proof. First, assume that M_A have a transition on B from p to p'. Then, $\mathrm{L}(A, p, B, p')$ is nonempty.

Now, if $A = S$, then $\mathrm{L}(A, p, B, p')$ is a nonempty subset of $\Sigma^*\{\dashv\}$ and *follow*$(A) = \{\lambda\}$. Hence, *first*(A, p, B) is nonempty.

Finally, if $A \neq S$, then *follow*(A) is a nonempty subset of Σ. Hence, *first*(A, p, B) is nonempty.

We now return to our original goal; namely, to compute a leftmost construction in which each partial syntax tree is compatible with a given input string of terminals. The algorithm begins with the initial partial syntax tree for G. The partial yield of this initial partial syntax tree is the empty string; hence, it is compatible with the input string.

Now, assume that eNLL is currently at the deepest active node v that is labeled with A and is annotated with state p. Furthermore, assume that the current input symbol is a (either a is in Σ or $a = \lambda$). Then, eNLL chooses one of the following steps to continue its computation:

Reduce step: If p is final in M_A and $a \in \mathit{follow}(A)$, then eNLL turns node v inactive. eNLL will in its next computation step manipulate the parent of v (if any); a will still be the current input symbol. We call this a **reduce step**.

Shift step: If M_A has a transition on a from p to some p', then eNLL adds a node labeled a to the right of v's children, annotates v with state p' and advances the input string by one position. eNLL will in its next computation step again manipulate the node v, but the input symbol will be advanced by one position. We call this a **shift step**.

Expand step: If $a \in \mathit{first}(A, p, B, p')$, then eNLL annotates node v with state p' and adds a new, active, node v' as a rightmost child of v. The new node gets the label B and is annotated with the initial state of M_B. eNLL will in its next computation step manipulate the new node v'; a will still be the current input symbol. We call this an **expand step**.

Figures 1, 2 and 3 illustrate the three eNLL-transformation types.

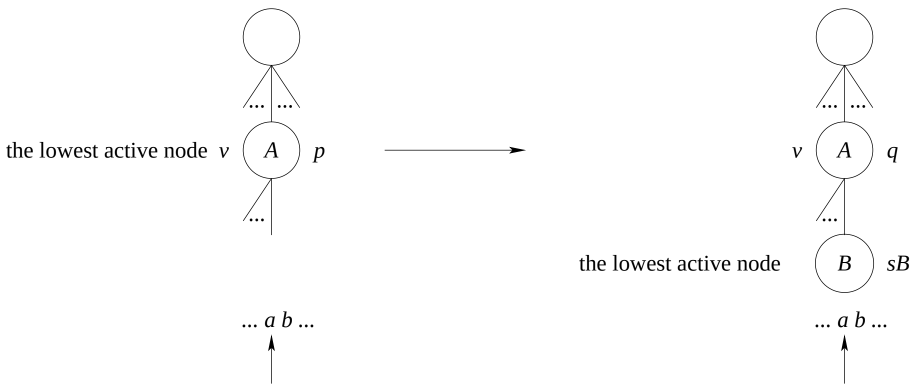

Fig. 1 The expand transition in algorithm eNLL.

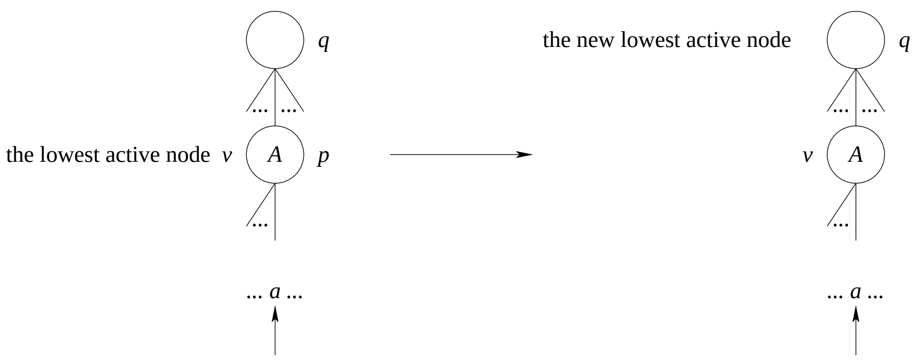

Fig. 2 The reduce transition in algorithm eNLL.

An **eNLL computation** terminates if no further transformation steps are possible. A **legal eNLL computation** is an eNLL computation that terminates with a partial syntax tree that

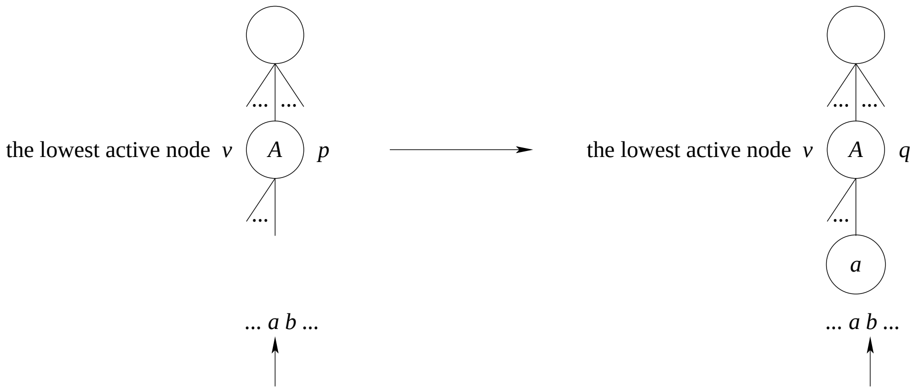

Fig. 3 The shift transition in algorithm eNLL.

is a syntax tree. The eNLL algorithm accepts an input string if it has a legal computation and the current input symbol is λ. ???

Observation 10 *A terminal string belongs to* $\mathrm{L}(G)$ *if and only if eNLL accepts it.*

Proof. The eNLL algorithm computes, nondeterministically, all leftmost constructions of partial syntax trees for the given input string and it terminates any computations that cannot possibly succeed because of the next input symbol.

Since we are primarily interested in deterministic parsers, we consider all possible conflicts that may cause eNLL to be properly nondeterministic. There are exactly six types of conflicts:

1. A **reduce-shift conflict;** that is, for some final state p of some M_A, the sets *follow*(A) and *transit*(A,p) have a nonempty intersection.
2. A **reduce-expand conflict;** that is, for some final state p of some M_A and some B in *transit*(A,p), the sets *follow*(A) and *first*(A,p,B) have a nonempty intersection.
3. A **shift-shift conflict;** that is, M_A has a transition on some terminal a from p to two different states p' and p''.
4. A **shift-expand conflict;** that is, *transit*(A,p) and *first*(A,p,B) have a nonempty intersection.
5. A **simple expand-expand conflict;** that is, M_A has a transition on some nonterminal B from p to two different states p' and p''.
6. An **expand-expand conflict;** that is, the sets *first*(A,p,B) and *first*(A,p,B') have a nonempty intersection for two different nonterminals B and B' in *transit*(A,p),

Note that for a simple expand-expand conflict, the sets *first*(A,p,B,p') and *first*(A,p,B,p'') have a nonempty intersection, since they both include the first symbols of strings in $\mathrm{L}(B)$. The set of first symbols of $\mathrm{L}(B)$ is nonempty, since grammar G is normalized; hence, $\mathrm{L}(B)$ is neither the empty set nor the empty-string set.

We say that grammar G is **eLL(1)** if and only if the eNLL algorithm for G is deterministic; that is, if none of the six types of conflict occur.

If G is eLL(1), then because neither shift-shift conflicts nor simple expand-expand conflicts occur, for every A in N, M_A is deterministic.

In practice, the rhs-language L_A of a nonterminal A is often represented by a regular expression. We have argued elsewhere [3] that *the* finite-state automaton that corresponds most naturally to an expression is its so-called Glushkov automaton. Hence, for the purposes of defining the eLL(1) property for extended context-free grammars whose rhs-languages are represented by regular expressions, we convert the expressions into Glushkov automata and then apply our definition of eLL(1). We then conclude that, for an eLL(1) extended context-free grammar, the regular expressions that represent rhs-languages must be one-unambiguous [3].

Theorem 1. *An extended context-free grammar G is eLL(1) if and only if it satisfies the following two conditions:*

1. *If p is a final state of M_A, then $first(A, p, X)$ and $follow(A)$ are disjoint for each grammar symbol X in $N \cup \Sigma$.*
2. *If $first(A, p, X, p')$ and $first(A, p, X', p'')$ have a nonempty intersection, then $X = X'$ and $p' = p''$.*

Proof. We apply Observation 8. The first condition of the theorem covers reduce-shift and reduce-expand conflicts, whereas the second condition covers the other four types of conflicts.

5 A table-driven version of the parser eNLL

Let $G = (N, \Sigma, P, S)$ be an extended context-free grammar and assume that the rhs-language of a nonterminal A is specified by a finite-state automaton M_A. Lastly, let Q to be the disjoint union of the state sets of M_A, A in N.

We define a **parsing table** T_G of an extended context-free grammar G to be a finite table that is indexed by the set N of nonterminal symbols, the set Q of states, and by the set $\Sigma \cup \{\lambda\}$. The table entries are (possibly empty) sets of parse actions. A **parse action** is a *reduce action*, a *shift action* or an *expand action*. A shift action has a state in Q as its parameter and an expand action has three parameters: a state in Q, a symbol in N and a second state in Q.

A **table-driven parser** operates from left to right on an input string $a_1 \cdots a_n$ of terminal symbols. It builds a leftmost partial syntax tree whose terminal prefix string is a prefix of the input string.

It begins with a one-node tree that is labeled with the sentence symbol S and is annotated with the start state in Q_S. The current input symbol a in $\Sigma \cup \{\lambda\}$ is a_i, if $n \geq 1$, and is λ, if $n = 0$.

The actions of the parser follow the actions in the corresponding cells of the parsing table. If the deepest active node v of the partial syntax tree constructed by the parser is labeled with

A in N and annotated with p in Q and if a in $\Sigma \cup \{\lambda\}$ is the current input symbol, then the parser nondeterministically performs one of the actions in the table cell that has index (A, p, a), if there is any.

To perform a reduce action, the parser makes v inactive. To perform a shift(p') action, it adds a new node labeled with a as the rightmost child of v, annotates v with the new state p' and advances the input by one position. To perform an expand(p', B, p'') action, it annotates node v with state p', adds a new node labeled with B as the rightmost child of v and annotates it with p''.

After performing an action specified by the parsing table, the parser continues to process the newly created partial syntax tree until no further parse actions can be performed. If at that time the partial syntax tree is a syntax tree and if the parser has completely read the input string (it has obtained λ as its current input symbol), then and only then does the parser accept the input string.

A table-driven parser does not need to maintain the whole partial syntax tree, it only needs to maintain a stack of active nodes; that is, a stack of their labels and annotations. Hence, the set of strings over Σ that are accepted by a table-driven parser can be recognized by a nondeterministic pushdown automaton; therefore, it is a context-free language over Σ.

Conversely, we can to build a parsing table T_G for an extended context-free grammar G. We add the following parse actions to each table cell (A, p, a), where A is in N, p is in Q_A, a is in $\Sigma \cup \{\lambda\}$:

Reduce: If p is a final state in M_A and $a \in \mathit{follow}(A)$.

Shift(p')**:** If M_A has a transition on a from p to p'.

Expand(p', B, s_B)**:** If $a \in \mathit{first}(A, p, B, p')$ and s_B is the start state of M_B.

Returning to the definition of eNLL(1), we observe that the parser that is driven by T_G, when started with the sentence symbol S and the start state of M_S, performs exactly the same computation steps as eNLL(1).

In particular, the table-driven parser either constructs a syntax tree for a given input string or it fails to do so.

Observation 11 *An extended context-free grammar G is eLL(1) if and only if G's parsing table T_G has no multiple entries.*

6 The characterization of LL(1) grammars in terms of eLL(1) grammars

Each context-free grammar can be trivially converted into an extended context-free grammar. Our next theorem proves that for context-free grammars the well-known LL(1) condition and our new eLL(1) condition are equivalent.

Before we state and prove this theorem, we need additional notation. If G derives a string α from A, then let $\mathit{first}(A, \alpha)$ consist of all terminals that start a string in $\mathrm{L}(\alpha)\,\mathit{follow}(A)$. Otherwise, G does not derive α from A, in which case, $\mathit{first}(A, \alpha)$ is defined to be the empty set.

Observation 12 *A context-free grammar G is LL(1) if and only if, for all disjoint pairs of productions $A \longrightarrow \alpha$ and $A \longrightarrow \beta$, the sets $\mathit{first}(A, \alpha)$ and $\mathit{first}(A, \beta)$ are disjoint.*

How do we convert a context-free grammar G into an extended context-free grammar G_e? If G has the $n \geq 1$ productions $A \longrightarrow X_i\alpha_i$, $1 \leq i \leq n$, such that X_i is in $N \cup \Sigma \cup \{\lambda\}$, α_i is in $(N \cup \Sigma)^*$ and $\alpha_i = \lambda$ if $X_i = \lambda$, then M_A identifies the start states of the "string automata" for $X_i\alpha_i$. As is demonstrated in Figure **??**, an input string $X_i\alpha_i$ visits the states p_0, p_1, some intermediate states and ends up in a final state p_{j_i}. Obviously, if one X_i is the empty string, then p_0 is final.

We consider only context-free grammars whose extended version is normalized.

Observation 13

- $\mathit{first}(A, X_i\alpha_i) = \mathit{first}(A, p_0, X_i, p_i)$ *if X_i is not the empty string.*
- $\mathit{first}(A, X_i\alpha_i) = \mathit{follow}(A)$ *if X_i is the empty string.*

Theorem 2. *A context-free grammar G is LL(1) if and only if G_e is eLL(1).*

Proof. Let G be a context-free grammar and let G_e be its extended version.

Then, for any state p of some M_A that is not the start state, there is at most one pair (X, p') such that $\mathit{first}(A, p, X, p')$ is nonempty. Hence, G satisfies the first condition of Theorem 1 if and only if $\mathit{first}(A, p_0, X_i, p_i)$ and $\mathit{first}(A, p_0, X_j, p_j)$ are disjoint for $i \neq j$; that is, if $\mathit{first}(X_i\alpha_i)$ and $\mathit{first}(X_j\alpha_j)$ are disjoint for $i \neq j$.

Next, if p is a final state in M_A, then $\mathit{first}(A, p, X)$ can only be nonempty if p is the start state of M_A. If, on the other hand, the start state of M_A is final, then there is exactly one i such that X_i is the empty string. Hence, G satisfies the second condition of Theorem 1 for p if and only if $\mathit{first}(A, p_0, X_j)$ and $\mathit{follow}(A)$ are disjoint for each $j \neq i$; that is, if $\mathit{first}(A, X_j\alpha_j)$ and $\mathit{first}(A, X_i\alpha_i)$ are disjoint for each $j \neq i$.

By Theorem 1 and Observation 12 we may conclude that G_e is eLL(1) if and only G is LL(1).

Acknowledgements

The work of both authors was supported partially by a joint DAAD-HK grant. In addition, the work of the second author was supported under a grant from the Research Grants Council of Hong Kong.

References

1. T. Bray, J. P. Paoh, and C. M.Sperberg-Mc Queen. Extensible markup language (XML) 1.0. http://www.w3.org/TR/1998/REC-xml-199 80210/, February 1998.
2. A. Brüggemann-Klein and D. Wood. The validation of SGML content models. *Mathematical and Computer Modelling*, 25:73–84, 1997.
3. A. Brüggemann-Klein and D. Wood. One-unambiguous regular languages. *Information and Computation*, 140:229–253, 1998.
4. A. Brüggemann-Klein and D. Wood. Caterpillars: A context specification technique. *Markup Languages: Theory & Practice*, 2(1):81–106, 2000.
5. A. Brüggemann-Klein and D. Wood. The regularity of two-way nondeterministic tree automata languages. *International Journal of Foundations of Computer Science*, :, 2001.
6. J. Clark, 1992. Source code for SGMLS. Available by anonymous ftp from ftp.uu.net and sgml1.ex.ac.uk.
7. C. F. Goldfarb. *The SGML Handbook*. Clarendon Press, Oxford, 1990.
8. Reinhold Heckmann. An efficient ELL(1)-parser generator. *Acta Informatica*, 23:127–148, 1986.
9. ISO 8879: Information processing—Text and office systems—Standard Generalized Markup Language (SGML), October 1986. International Organization for Standardization.
10. P. Kilpeläinen and D. Wood. Sgml and xml document grammars and exceptions. *Information and Computation*, 169:230–251, 2001.
11. W.R. LaLonde. Regular right part grammars and their parsers. *Communications of the ACM*, 20:731–741, 1977.
12. J. Lewi, K. De Vlaminck, J. Huens, and M. Huybrechts. The ELL(1) parser generator and error recovery mechanism. *Acta Informatica*, 10:209–228, 1978.
13. J. Lewi, K. de Vlaminck, J. Huens, and M. Huybrechts. *A Programming Methodology in Compiler Construction: Part 1: Concepts*. North-Holland, Amsterdam, The Netherlands, 1979.
14. J. Lewi, K. De Vlaminck, J. Huens, and E. Steegmans. *A Programming Methodology in Compiler Construction, Part 2: Implementation*. North-Holland, Amsterdam, The Netherlands, 1982.
15. J. Lewi, K. de Vlaminck, E. Steegmans, and I. Van Horebeek. *Software Develepment by LL(1) Syntax Description*. John Wiley & Sons, Inc., New York, NY, 1992.
16. S. Sippu and E. Soisalon-Soininen. *Parsing Theory: I Languages and Parsing*. EATCS Monographs on Theoretical Computer Science, Volume 15. Springer-Verlag, New York, NY, 1988.
17. S. Sippu and E. Soisalon-Soininen. *Parsing Theory: II LR(k) and LL(k) Parsing*. EATCS Monographs on Theoretical Computer Science, Volume 20. Springer-Verlag, New York, NY, 1990.
18. J. Warmer and S. Townsend. The implementation of the Amsterdam SGML pa rser. *Electronic Publishing, Origination, Dissemination, and Design*, 2:65–90, 1989.
19. J. Warmer and H. van Vliet. Processing SGML documents. *Electronic Publishing, Origination, Dissemination, and Design*, 4(1):3–26, March 1991.
20. Reinhard Wilhelm and Dieter Maurer. *Compiler Design*. Addison-Wesley, Reading, MA, 1995.
21. D. Wood. Standard generalized markup language: Mathematical and philosophical issues. In Jan van Leeuwen, editor, *Computer Science Today*, volume 1000 of *Lecture Notes in Computer Science*, pages 344–365. Springer-Verlag, New York, NY, 1995.