A Graphical Model for Context-Free Grammar Parsing

Keshav Pingali[1] and Gianfranco Bilardi[2]

[1] The University of Texas, Austin,
Texas 78712, USA
pingali@cs.utexas.edu
[2] Università di Padova
35131 Padova, Italy
bilardi@dei.unipd.it

Abstract. In the compiler literature, parsing algorithms for context-free grammars are presented using string rewriting systems or abstract machines such as pushdown automata. Unfortunately, the resulting descriptions can be baroque, and even a basic understanding of some parsing algorithms, such as Earley's algorithm for general context-free grammars, can be elusive. In this paper, we present a graphical representation of context-free grammars called the Grammar Flow Graph (GFG) that permits parsing problems to be phrased as path problems in graphs; intuitively, the GFG plays the same role for context-free grammars that nondeterministic finite-state automata play for regular grammars. We show that the GFG permits an elementary treatment of Earley's algorithm that is much easier to understand than previous descriptions of this algorithm. In addition, look-ahead computation can be expressed as a simple inter-procedural dataflow analysis problem, providing an unexpected link between front-end and back-end technologies in compilers. These results suggest that the GFG can be a new foundation for the study of context-free grammars.

भज गोविन्दं भज गोविन्दं, गोविन्दं भज मूढ़मते।
संप्राप्ते सन्निहिते काले, न हि न हि रक्षति डुकृञ् करणे ॥१॥

Adore the Lord, adore the Lord, adore the Lord, you fool.
When death comes at its appointed time, the rules of grammar will not save you.

From ``Bhaja Govindam''
Adi Sankara (788 AD-820 AD)

1 Introduction

The development of elegant and practical parsing algorithms for context-free grammars is one of the major accomplishments of 20^{th} century Computer Science. Two abstractions are used to present these algorithms: string rewriting systems and pushdown automata, but the resulting descriptions are unsatisfactory for several reasons.

– Even an elementary understanding of some grammar classes requires mastering a formidable number of complex concepts. For example, LR(k) parsing requires an understanding of rightmost derivations, right sentential forms, viable prefixes, handles, complete valid items, and conflicts, among other notions.

B. Franke (Ed.): CC 2015, LNCS 9031, pp. 3–27, 2015.
DOI: 10.1007/978-3-662-46663-6_1

- Parsing algorithms for different grammar classes are presented using different abstractions; for example, LL grammars are presented using recursive-descent, while LR grammars are presented using shift-reduce parsers. This obscures connections between different grammar classes and parsing techniques.
- Although regular grammars are a proper subset of context-free grammars, parsing algorithms for regular grammars, which are presented using finite-state automata, appear to be entirely unrelated to parsing algorithms for context-free grammars.

In this paper, we present a novel approach to context-free grammar parsing that is based on a graphical representation of context-free grammars called the *Grammar Flow Graph*(GFG). *Intuitively, the GFG plays the same role for context-free grammars that the nondeterministic finite-state automaton (NFA) does for regular grammars*: parsing problems can be formulated as path problems in the GFG, and parsing algorithms become algorithms for solving these path problems. The GFG simplifies and unifies the presentation of parsing algorithms for different grammar classes; in addition, finite-state automata can be seen as an optimization of the GFG for the special case of regular grammars, providing a pleasing connection between regular and context-free grammars.

Section 2 introduces the GFG, and shows how the GFG for a given context-free grammar can be constructed in a straight-foward way. Membership of a string in the language generated by the grammar can be proved by finding what we call a *complete balanced GFG path* that generates this string. Since every regular grammar is also a context-free grammar, a regular grammar has both a GFG and an NFA representation. In Section 2.4, we establish a connection between these representations: we show that applying the continuation-passing style (CPS) optimization [1,2] to the GFG of a right-linear regular grammar produces an NFA that is similar to the NFA produced by the standard algorithm for converting a right-linear regular grammar to an NFA.

Earley's algorithm[3] for parsing general context-free grammars is one of the more complicated parsing algorithms in the literature [4]. The GFG reveals that this algorithm is a straightforward extension of the well-known "ϵ-closure" algorithm for simulating all the moves of an NFA (Section 3). The resulting description is much simpler than previous descriptions of this algorithm, which are based on dynamic programming, abstract interpretation, and Galois connections [3,5,6].

Look-ahead is usually presented in the context of particular parsing strategies such as SLL(1) parsing. In Section 4, we show that the GFG permits look-ahead computation to be formulated independently of the parsing strategy as a simple inter-procedural dataflow analysis problem, unifying algorithmic techniques for compiler front-ends and back-ends. The GFG also enables a simple description of parsers for LL and LR grammars and their sub-classes such as SLL, SLR and LALR grammars, although we do not discuss this in this paper.

Section 5 describes related work. Structurally, the GFG resembles the recursive transition network (RTN) [7], which is used in natural language processing and parsers like ANTLR [8], but there are crucial differences. In particular, the GFG is a single graph in which certain paths are of interest, not a collection of recursive state machines with an operational model like chart parsing for their interpretation. Although motivated by similar concerns, complete balanced paths are different from CFL-paths [9].

Proofs of the main theorems are given in the appendix.

2 Grammar Flow Graph (GFG) and Complete Balanced Paths

A context-free grammar Γ is a 4-tuple $<N, T, P, S>$ where N is a finite set of non-terminals, T is a finite set of terminals, $P \subseteq N \times (N \cup T)^*$ is the set of productions, and $S \in N$ is the start symbol. To simplify the development, we make the following standard assumptions about Γ throughout this paper.

- A1: S does not appear on the righthand side of any production.
- A2: Every non-terminal is used in a derivation of some string of terminals from S (no useless non-terminals [4]).

Any grammar Γ' can be transformed in time $O(|\Gamma'|)$ into an equivalent grammar Γ satisfying the above assumptions [10]. The running example in this paper is this grammar: $E \rightarrow int | (E + E) | E + E$. An equivalent grammar is shown in Figure 1, where the production $S \rightarrow E$ has been added to comply with A1.

2.1 Grammar Flow Graph (GFG)

Figure 1 shows the GFG for the expression grammar. Some edges are labeled explicitly with terminal symbols, and the others are implicitly labeled with ϵ. The GFG can be understood by analogy with inter-procedural control-flow graphs: each production is represented by a "procedure" whose control-flow graph represents the righthand side of that production, and a non-terminal A is represented by a pair of nodes $\bullet A$ and $A\bullet$, called the $start$ and end nodes for A, that gather together the control-flow graphs for the productions of that non-terminal. An occurrence of a non-terminal in the righthand side of a production is treated as an invocation of that non-terminal.

The control-flow graph for a production $A \rightarrow u_1 u_2 .. u_r$ has $r + 1$ nodes. As in finite-state automata, node labels in a GFG do not play a role in parsing and can be chosen arbitrarily, but it is convenient to label these nodes $A \rightarrow \bullet u_1 u_2 .. u_r$ through $A \rightarrow u_1 u_2 .. u_r \bullet$; intuitively, the $\bullet$ indicates how far parsing has progressed through a production (these labels are related to *items* [4]). The first and last nodes in this sequence are called the *entry* and *exit* nodes for that production. If u_i is a terminal, there is a *scan edge* with that label from the *scan node* $A \rightarrow u_1 .. u_{i-1} \bullet u_i .. u_r$ to node $A \rightarrow u_1 .. u_i \bullet u_{i+1} .. u_r$, just as in finite-state automata. If u_i is a non-terminal, it is considered to be an "invocation" of that non-terminal, so there are *call* and *return* edges that connect nodes $A \rightarrow u_1 .. u_{i-1} \bullet u_i .. u_r$ to the $start$ node of non-terminal u_i and its end node to $A \rightarrow u_1 .. u_i \bullet u_{i+1} .. u_r$.

Formally, the GFG for a grammar Γ is denoted by $GFG(\Gamma)$ and it is defined as shown in Definition 1. It is easy to construct the GFG for a grammar Γ in $O(|\Gamma|)$ time and space using Definition 1.

Definition 1. *If $\Gamma = <N, T, P, S>$ is a context-free grammar, $G = GFG(\Gamma)$ is the smallest directed graph $(V(\Gamma), E(\Gamma))$ that satisfies the following properties.*

- ◇ *For each non-terminal $A \in N$, $V(\Gamma)$ contains nodes labeled $\bullet A$ and $A\bullet$, called the* start *and* end *nodes respectively for A.*
- ◇ *For each production $A \rightarrow \epsilon$, $V(\Gamma)$ contains a node labeled $A \rightarrow \bullet$, and $E(\Gamma)$ contains edges $(\bullet A, A \rightarrow \bullet)$, and $(A \rightarrow \bullet, A\bullet)$.*

- ◇ *For each production* $A \rightarrow u_1 u_2 ... u_r$
  - $V(\Gamma)$ *contains* $(r+1)$ *nodes labeled* $A \rightarrow \bullet u_1 ... u_r$, $A \rightarrow u_1 \bullet ... u_r$, ..., $A \rightarrow u_1 ... u_r \bullet$,
  - $E(\Gamma)$ *contains* entry *edge* $(\bullet A, A \rightarrow \bullet u_1 ... u_r)$, *and* exit *edge* $(A \rightarrow u_1 ... u_r \bullet, A \bullet)$,
  - *for each* $u_i \in T$, $E(\Gamma)$ *contains a* scan *edge* $(A \rightarrow u_1 ... u_{i-1} \bullet u_i .. u_r, A \rightarrow u_1 ... u_i \bullet u_{i+1} .. u_r)$ *labeled* u_i,
  - *for each* $u_i \in N$, $E(\Gamma)$ *contains a* call *edge* $(A \rightarrow u_1 ... u_{i-1} \bullet u_i ... u_r, \bullet u_i)$ *and a* return *edge* $(u_i \bullet, A \rightarrow u_1 ... u_i \bullet u_{i+1} ... u_r)$.
    Node $A \rightarrow u_1 ... u_{i-1} \bullet u_i ... u_r$ *is a* call *node, and* matches *the* return *node* $A \rightarrow u_1 ... u_i \bullet u_{i+1} ... u_r$.
- ◇ *Edges other than scan edges are labeled with* ϵ.

When the grammar is obvious from the context, a GFG will be denoted by $G{=}(V, E)$. Note that $start$ and end nodes are the only nodes that can have a fan-out greater than one. This fact will be important when we interpret the GFG as a nondeterministic automaton in Section 2.3.

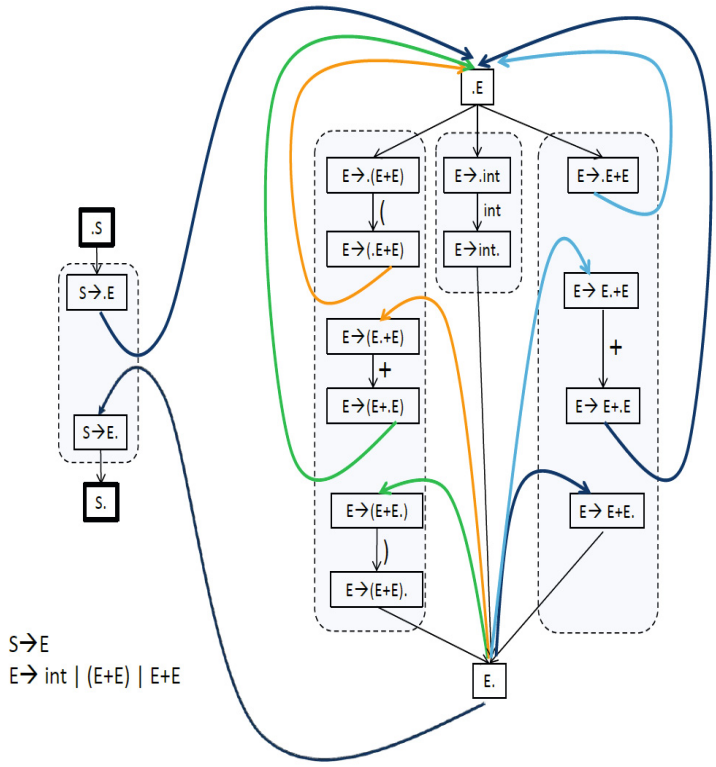


Fig. 1. Grammar Flow Graph example

Table 1. Classification of GFG nodes: a node can belong to several categories. ($A, B \in N$, $t \in T$, and $\alpha, \gamma \in (T + N)^*$)

Node type	Description
$start$	Node labeled $\bullet A$
end	Node labeled $A\bullet$
$call$	Node labeled $A \rightarrow \alpha \bullet B\gamma$
$return$	Node labeled $A \rightarrow \alpha B \bullet \gamma$
$entry$	Node labeled $A \rightarrow \bullet\alpha$
$exit$	Node labeled $A \rightarrow \alpha\bullet$
$scan$	Node labeled $A \rightarrow \alpha \bullet t\gamma$

2.2 Balanced Paths

The following definition is standard.

Definition 2. *A path in a GFG $G{=}(V, E)$ is a non-empty sequence of nodes $v_0, \ldots, v_l$, such that $(v_0, v_1), (v_1, v_2), ..., (v_{l-1}, v_l)$ are all edges in E.*

In a given GFG, the notation $v_1 \rightsquigarrow v_2$ denotes the edge from v_1 to v_2, and the notation $v_1 \rightsquigarrow^* v_n$ denotes a path from v_1 to v_n; the symbol "$\rightarrow$" is reserved for productions and derivations. If $Q_1{:}v_1 \rightsquigarrow^* v_m$ and $Q_2{:}v_m \rightsquigarrow^* v_r$ are paths in a GFG, the notation $Q_1{+}Q_2$ denotes the concatenation of paths Q_1 and Q_2. In this spirit, we denote string concatenation by $+$ as well. It is convenient to define the following terms to talk about certain paths of interest in the GFG.

Definition 3. *A* complete path *in a GFG is a path whose first node is $\bullet S$ and whose last node is $S\bullet$.*

A path is said to generate *the word w resulting from the concatenation of the labels on its sequence of edges. By convention, $w = \epsilon$ for a path with a single node.*

The GFG can be viewed as a nondeterministic *finite-state* automaton (NFA) whose start state is $\bullet S$, whose accepting state is $S\bullet$, and which makes nondeterministic choices at $start$ and end nodes that have a fan-out more than one. Each complete GFG path generates a word in the regular language recognized by this NFA. In Figure 1, the path Q: $\bullet S \rightsquigarrow S{\rightarrow}\bullet E \rightsquigarrow \bullet E \rightsquigarrow E{\rightarrow}\bullet(E + E) \rightsquigarrow E{\rightarrow}(\bullet E + E) \rightsquigarrow \bullet E \rightsquigarrow E{\rightarrow}\bullet int \rightsquigarrow E{\rightarrow}int\bullet \rightsquigarrow E\bullet \rightsquigarrow S{\rightarrow}E\bullet \rightsquigarrow S\bullet$ generates the string "(int". However, this string is not generated by the context-free grammar from which this GFG was constructed.

To interpret the GFG as the representation of a context-free grammar, it is necessary to restrict the paths that can be followed by the automaton. Going back to the intuition that the GFG is similar to an inter-procedural call graph, we see that Q is not an *interprocedurally valid path* [11]: at $E\bullet$, it is necessary to take the return edge to node $E{\rightarrow}(E\bullet + E)$ since the call of E that is being completed was made at node $E{\rightarrow}(\bullet E + E)$. In general, the automaton can make a free choice at $start$ nodes just like an NFA, but at end nodes, the return edge to be taken is determined by the call that is being completed.

The paths the automaton is allowed to follow are called *complete balanced paths* in this paper. Intuitively, if we consider matching $call$ and $return$ nodes to be opening

and closing parentheses respectively of a unique color, the parentheses on a complete balanced path must be properly nested [12]. In the formal definition below, if K is a sequence of nodes, we let v, K, w represent the sequence of nodes obtained by prepending node v and appending node w to K.

Definition 4. *Given a GFG for a grammar Γ=<N, T, P, S>, the set of* balanced sequences of $call$ and $return$ nodes *is the smallest set of sequences of $call$ and $return$ nodes that is closed under the following conditions.*

- *The empty sequence is balanced.*
- *The sequence $(A \to \alpha \bullet B\gamma), K, (A \to \alpha B \bullet \gamma)$ is balanced if K is a balanced sequence, and production $(A \to \alpha B\gamma) \in P$.*
- *The concatenation of two balanced sequences $v_1 ... v_f$ and $y_1 ... y_s$ is balanced if $v_f \neq y_1$. If $v_f = y_1$, the sequence $v_1 ... v_f y_2 ... y_s$ is balanced.*

This definition is essentially the same as the standard definition of balanced sequences of parentheses; the only difference is the case of $v_f = y_1$ in the last clause, which arises because a node of the form $A \to \alpha X \bullet Y \beta$ is both a $return$ node and a $call$ node.

Definition 5. *A GFG path $v_0 \rightsquigarrow^* v_l$ is said to be a* balanced path *if its subsequence of $call$ and $return$ nodes is balanced.*

Theorem 1. *If Γ=<N, T, P, S> is a context-free grammar and $w \in T^*$, w is in the language generated by Γ iff it is generated by a complete balanced path in $GFG(\Gamma)$.*

Proof. This is a special case of Theorem 4 in the Appendix.

Therefore, the parsing problem for a context-free grammar Γ can be framed in GFG terms as follows: given a string w, determine if there are complete balanced paths in $GFG(\Gamma)$ that generate w (recognition), and if so, produce a representation of these paths (parsing). If the grammar is unambiguous, each string in the language is generated by exactly one such path.

The parsing techniques considered in this paper read the input string w from left to right one symbol at a time, and determine reachability along certain paths starting at $\bullet S$. These paths are always *prefixes* of complete balanced paths, and if a prefix u of w has been read up to that point, all these paths generate u. For the technical development, similar paths are needed even though they begin at nodes other than $\bullet S$. Intuitively, these *call-return* paths (*CR-paths* for short) are just segments of complete balanced paths; they may contain *unmatched* $call$ and $return$ nodes, but they do not have *mismatched* $call$ and $return$ nodes, so they can always be extended to complete balanced paths.

Definition 6. *Given a GFG, a* CR-sequence *is a sequence of $call$ and $return$ nodes that does not contain a subsequence v_c, K, v_r where $v_c \in call$, K is balanced, $v_r \in return$, and v_c and v_r are not matched.*

Definition 7. *A GFG path is said to be a* CR-path *if its subsequence of $call$ and $return$ nodes is a CR-sequence.*

Unless otherwise specified, the origin of a CR-path will be assumed to be $\bullet S$, the case that arises most frequently.

2.3 Nondeterministic GFG Automaton (NGA)

Figure 2 specifies a push down automaton (PDA), called the nondeterministic GFG automaton (NGA), that traverses complete balanced paths in a GFG under the control of the input string. To match $call$'s with $return$'s, it uses a stack of "return addresses" as is done in implementations of procedural languages. The configuration of the automaton is a three-tuple consisting of the GFG node where the automaton currently is (this is called the PC), a stack of $return$ nodes, and the partially read input string. The symbol $\longmapsto$ denotes a state transition.

The NGA begins at $\bullet S$ with the empty stack. At a $call$ node, it pushes the matching $return$ node on the stack. At a $start$ node, it chooses the production *nondeterministically*. At an end node, it pops a $return$ node from the stack and continues the traversal from there. If the input string is in the language generated by the grammar, the automaton will reach $S\bullet$ with the empty stack (the end rule cannot fire at $S\bullet$ because the stack is empty). We will call this a *nondeterministic GFG automaton* or *NGA* for short. It is a special kind of pushdown automaton (PDA). It is not difficult to prove that the NGA accepts exactly those strings that can be generated by some complete balanced path in $GFG(\Gamma)$ whence, by Theorem 1, the NGA accepts the language of Γ. (Technically, acceptance is by final state [13], but it is easily shown that the final state $S\bullet$ can only be reached with an empty stack.)

The nondeterminism in the NGA is called *globally angelic* nondeterminism [14] because the nondeterministic transitions at $start$ nodes have to ensure that the NGA ultimately reaches $S\bullet$ if the string is in the language generated by the grammar. The recognition algorithms described in this paper are concerned with deterministic implementations of the globally angelic nondeterminism in the NGA.

NGA configuration $(PC \times C \times K)$, where:

Program counter $PC \in V(\Gamma)$ (a state of the finite control)
Partially-read input strings $C \in T^* \times T^*$
$(C = (u, v)$, where prefix u of input string $w = uv$ has been read)
Stack of return nodes $K \in V_R(\Gamma)^*$, where $V_R(\Gamma)$ is the set of return nodes

Initial Configuration: $<\bullet S, [\,], \bullet w>$
Accepting configuration: $<S\bullet, [\,], w\bullet>$

Transition function:

$$\text{CALL } <A\to\alpha\bullet B\gamma, C, K> \quad \longmapsto \quad <\bullet B, C, (A\to\alpha B\bullet\gamma, K)>$$

$$\text{START } <\bullet B, C, K> \quad \longmapsto \quad <B\to\bullet\beta, C, K> \text{ (nondeterministic choice)}$$

$$\text{EXIT } <B\to\beta\bullet, C, K> \quad \longmapsto \quad <B\bullet, C, K>$$

$$\text{END } <B\bullet, C, (A\to\alpha B\bullet\gamma, K)> \quad \longmapsto \quad <A\to\alpha B\bullet\gamma, C, K>$$

$$\text{SCAN } <A\to\alpha\bullet t\gamma, u\bullet tv, K> \quad \longmapsto \quad <A\to\alpha t\bullet\gamma, ut\bullet v, K>$$

Fig. 2. Nondeterministic GFG Automaton (NGA)

2.4 Relationship between NFA and GFG for Regular Grammars

Every regular grammar is also a context-free grammar, so a regular grammar has two graphical representations, an NFA and a GFG. A natural question is whether there is a connection between these graphs. We show that applying the continuation-passing style (CPS) optimization [1,2] to the NGA of a context-free grammar that is a right-linear regular grammar[1] produces an NFA for that grammar.

For any context-free grammar, consider a production $A \rightarrow \alpha B$ in which the last symbol on the righthand side is a non-terminal. The canonical NGA in Figure 2 will push the return node $A \rightarrow \alpha B \bullet$ before invoking B, but after returning to this exit node, the NGA just transitions to $A \bullet$ and pops the return node for the invocation of A. Had a return address not been pushed when the call to B was made, the NGA would still recognize the input string correctly because when the invocation of B completes, the NGA would pop the return stack and transition directly to the return node for the invocation of A. This optimization is similar to the continuation-passing style (CPS) transformation, which is used in programming language implementations to convert tail-recursion to iteration.

To implement the CPS optimization in the context of the GFG, it is useful to introduce a new type of node called a *no-op* node, which represents a node at which the NGA does nothing other than to transition to the successor of that node. If a production for a non-terminal other than S ends with a non-terminal, the corresponding $call$ is replaced with a no-op node; since the NGA will never come back to the corresponding $return$ node, this node can be replaced with a no-op node as well. For a right-linear regular grammar, there are no $call$ or $return$ nodes in the optimized GFG. The resulting GFG is just an NFA, and it is a variation of the NFA that is produced by using the standard algorithms for producing an NFA from a right-linear regular grammar [13].

3 Parsing of General Context-Free Grammars

General context-free grammars can be parsed using an algorithm due to Earley [3]. Described using derivations, the algorithm is not very intuitive and seems unrelated to other parsing algorithms. For example, the monograph on parsing by Sippu and Soisalon-Soininen [10] omits it, Grune and Jacobs' book describes it as "top-down restricted breadth-first bottom-up parsing" [5], and the "Dragon book" [4] mentions it only in the bibliography as "a complex, general-purpose algorithm due to Earley that tabulates LR-items for each substring of the input." Cousot and Cousot use Galois connections between lattices to show that Earley's algorithm is an abstract interpretation of a refinement of the derivation semantics of context-free grammars [6].

In contrast to these complicated narratives, a remarkably simple interpretation of Earley's algorithm emerges when it is viewed in terms of the GFG: *Earley's algorithm is the context-free grammar analog of the well-known simulation algorithm for nondeterministic finite-state automata (NFA)* [4]. While the latter tracks reachability along prefixes of complete paths, the former tracks reachability along prefixes of complete *balanced* paths.

[1] A right-linear regular grammar is a regular grammar in which the righthand side of a production consists of a string of zero or more terminals followed by at most one non-terminal.

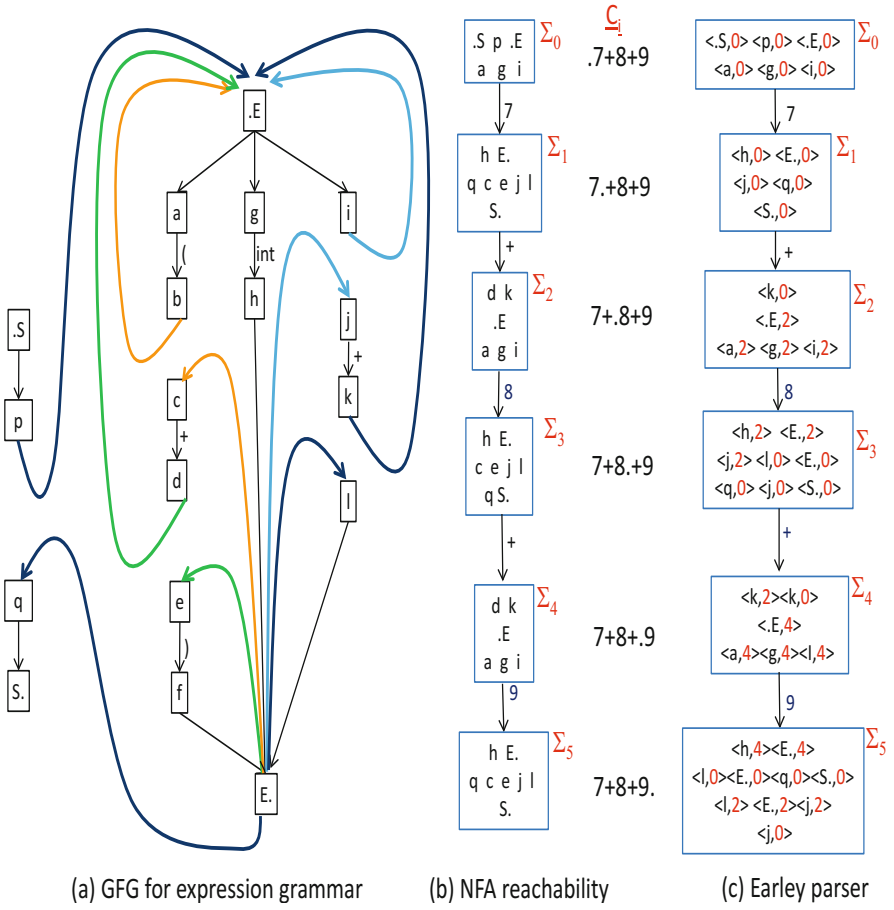


Fig. 3. Earley parser: example

3.1 NFA Simulation Algorithm

As a step towards Earley's algorithm, consider interpreting the GFG as an NFA (so non-deterministic choices are permitted at both *start* and *end* nodes). The NFA simulation on a given an input word $w[1..n]$ can be viewed as the construction of a sequence of node sets $\Sigma_0, ..., \Sigma_n$. Here, Σ_0 is the ϵ-closure of $\{\bullet S\}$. For $i = 1, \ldots, n$, set Σ_i is the ϵ-closure of the set of nodes reachable from nodes in Σ_{i-1} by scan edges labeled $w[i]$. The string w is in the language recognized by the NFA if and only if $S\bullet \in \Sigma_n$.

Figure 3(a) shows the GFG of Figure 1, but with simple node labels. Figure 3(b) illustrates the behavior of the NFA simulation algorithm for the input string "7+8+9". Each Σ_i is associated with a terminal string pair C_i=$u.v$, which indicates that prefix u of the input string $w = uv$ has been read up to that point.

(a) NFA Simulation of GFG

Sets of GFG nodes Σ: $\mathscr{P}(V(\Gamma))$
Partially-read input strings $C : T^* \times T^*$
Recognizer configurations $(\Sigma \times C)^+$
Acceptance: $S\bullet \in \Sigma_{|w|}$

$$\text{INIT}\ \frac{}{(\bullet S\ \in \Sigma_0)\ \wedge\ (C^0 = \bullet w)} \qquad \text{CALL}\ \frac{A{\to}\alpha\bullet B\gamma \in \Sigma_j}{\bullet B \in \Sigma_j} \qquad \text{START}\ \frac{\bullet B \in \Sigma_j}{B{\to}\bullet\beta \in \Sigma_j}$$

$$\text{EXIT}\ \frac{B{\to}\beta\bullet \in \Sigma_j}{B\bullet \in \Sigma_j} \qquad \text{END}\ \frac{B\bullet \in \Sigma_j}{A{\to}\alpha B\bullet\gamma \in \Sigma_j}$$

$$\text{SCAN}\ \frac{A{\to}\alpha\bullet t\gamma \in \Sigma_j \qquad C^j = u\bullet tv}{(A{\to}\alpha t\bullet\gamma \in \Sigma_{j+1}) \wedge (C^{j+1} = ut\bullet v)}$$

(b) Earley recognizer

Non-negative integers: $\mathcal{N}$
Sets of tagged GFG nodes Σ: $\mathscr{P}(V(\Gamma) \times \mathcal{N})$
Partially-read input strings $C : T^* \times T^*$
Recognizer configurations $(\Sigma \times C)^+$
Acceptance: $<S\bullet, 0> \in \Sigma_{|w|}$

$$\text{INIT}\ \frac{}{(<\bullet S, 0> \in \Sigma_0) \wedge (C^0 = \bullet w)} \qquad \text{CALL}\ \frac{<A{\to}\alpha\bullet B\gamma, i> \in \Sigma_j}{<\bullet B, j> \in \Sigma_j}$$

$$\text{START}\ \frac{<\bullet B, j> \in \Sigma_j}{<B{\to}\bullet\beta, j> \in \Sigma_j} \qquad \text{EXIT}\ \frac{<B{\to}\beta\bullet, k> \in \Sigma_j}{<B\bullet, k> \in \Sigma_j}$$

$$\text{END}\ \frac{<B\bullet, k> \in \Sigma_j \qquad <A{\to}\alpha\bullet B\gamma, i> \in \Sigma_k}{<A{\to}\alpha B\bullet\gamma, i> \in \Sigma_j}$$

$$\text{SCAN}\ \frac{<A{\to}\alpha\bullet t\gamma, i> \in \Sigma_j \qquad C^j = u\bullet tv}{(<A{\to}\alpha t\bullet\gamma, i> \in \Sigma_{j+1}) \wedge (C^{j+1} = ut\bullet v)}$$

(c) Earley parser

Non-negative integers: $\mathcal{N}$
Program counter PC: $V(\Gamma) \times \mathcal{N}$
Stack of call nodes K: $V_R(\Gamma)^*$
Parser configurations: $(PC \times \mathcal{N} \times K)$
Acceptance: final configuration is $<<\bullet S, 0>, 0, [\,]>$

$$\text{INIT}\ <<S\bullet, 0>, |w|, [\,]>$$

$$\text{CALL}^{-1}\ <<\bullet B, j>, j, (<A{\to}\alpha\bullet B\gamma, i>, K)> \quad\longmapsto\quad <<A{\to}\alpha\bullet B\gamma, i>, j, K>$$

$$\text{START}^{-1}\ <<B{\to}\bullet\beta, j>, j, K> \quad\longmapsto\quad <<\bullet B, j>, j, K>$$

$$\text{EXIT}^{-1}\ <<B\bullet, k>, j, K> \quad\longmapsto\quad <<B{\to}\beta\bullet, k>, j, K>$$

$$\text{if}\ (<B{\to}\beta\bullet, k>{\in}\Sigma_j)(\text{non}{-}\text{determinism})$$

$$\text{END}^{-1}\ <<A{\to}\alpha B\bullet\gamma, i>, j, K> \quad\longmapsto\quad <<B\bullet, k>, j, (<A{\to}\alpha\bullet B\gamma, i>, K)>$$

$$\text{if}\ (<B\bullet, k>{\in}\Sigma_j\ \text{and}\ <A{\to}\alpha\bullet B\gamma, i>{\in}\Sigma_k)(\text{non}{-}\text{determinism})$$

$$\text{SCAN}^{-1}\ <<A{\to}\alpha t\bullet\gamma, i>, (j+1), K> \quad\longmapsto\quad <<A{\to}\alpha\bullet t\gamma, i>, j, K>$$

Fig. 4. NFA, Earley recognizer, and Earley parser: input word is w

The behavior of this *NFA ϵ-closure algorithm* on a GFG is described concisely by the rules shown in Figure 4(a). Each rule is an inference rule or constraint; in some rules, the premises have multiple consequents. It is straightforward to use these rules to compute the smallest Σ-sets that satisfy all the constraints. The INIT rule enters $\bullet S$ into Σ_0. Each of the other rules is associated with traversing a GFG edge from the node in its assumption to the node in its consequence. Thus, the CALL, START, END, and EXIT rules compute the ϵ-closure of a Σ-set; notice that the END rule is applied to all outgoing edges from END nodes.

3.2 Earley's Algorithm

Like the NFA ϵ-closure algorithm, Earley's algorithm builds Σ sets, but it computes reachability only along CR-paths starting at $\bullet S$. Therefore, the main difference between the two algorithms is at end nodes: a CR-path that reaches an end node should be extended only to the $return$ node corresponding to the last unmatched $call$ node on that path.

One way to find this $call$ node is to *tag* each $start$ node with a unique ID (tag) when it is entered into a Σ-set, and propagate this tag through the nodes of productions for this non-terminal all the way to the end node. At the end node, this unique ID can be used to identify the Σ-set containing corresponding $start$ node. The last unmatched $call$ node on the path must be contained in that set as well, and from that node, the $return$ node to which the path should be extended can easily be determined.

To implement the tag, it is simple to use the number of the Σ-set to which the $start$ node is added, as shown in Figure 4(b). When the CALL rule enters a $start$ node into a Σ set, the tag assigned to this node is the number of that Σ set. The END rule is the only rule that actually uses tags; all other rules propagate tags. If $<B\bullet, k> \in \Sigma_j$, then the matching $start$ and $call$ nodes are in Σ_k, so Σ_k is examined to determine which of the immediate predecessors of node $\bullet B$ occur in this set. These must be $call$ nodes of the form $A \rightarrow \alpha \bullet B\gamma$, so the matching $return$ nodes $A \rightarrow \alpha B \bullet \gamma$ are added to Σ_j with the tags of the corresponding $call$ nodes. For a given grammar, this can easily be done in time constant with respect to the length of the input string. A string is in the language recognized by the GFG iff Σ_n contains $<S\bullet, 0>$. Figure 3(c) shows the Σ sets computed by the Earley algorithm for the input string "7+8+9".

We discuss a small detail in using the rules of Figures 4(a,b) to construct Σ-sets for a given GFG and input word. The existence of a unique smallest sequence of Σ-sets can be proved in many ways, such as by observing that the rules have the diamond property and are strongly normalizing [15]. A canonical order of rule application for the NFA rules is the following. We give a unique number to each GFG edge, and associate the index $\langle j, m \rangle$ with a rule instance that corresponds to traversing edge m and adding the destination node to Σ_j; the scheduler always pick the rule instance with the smallest index. This order completes the Σ sets in sequence, but many other orders are possible. The same order can be used for the rules in Figure 4(b) except that for the END rule, we use the number on the edge ($B\bullet$, $A \rightarrow \alpha B \bullet \gamma$).

Correctness of the rules of Figure 4(b) follows from Theorem 2.

Theorem 2. *For a grammar Γ=<N, T, P, S> and an input word w,< $S\bullet, 0$ >$\in \Sigma_{|w|}$ iff w is a word generated by grammar Γ.*

Proof. See Section A.2.

The proof of Theorem 2 shows the following result, which is useful as a characterization of the contents of Σ sets. Let $w[i..j]$ denote the substring of input w from position i to position j inclusive if $i \leq j$, and let it denote ϵ if $i > j$. It is shown that $<A \rightarrow \alpha \bullet \beta, i> \in \Sigma_j$ iff there is a CR-path $P : \bullet S \rightsquigarrow^* \bullet A \rightsquigarrow^* (A \rightarrow \alpha \bullet \beta)$ such that

1. $\bullet S \rightsquigarrow^* \bullet A$ generates $w[1..i]$, and
2. $\bullet A \rightsquigarrow^* (A \rightarrow \alpha \bullet \beta)$ is balanced and generates $w[(i+1)..j]$.

Like the NFA algorithm, Earley's algorithm determines reachability along certain paths but does not represent paths explicitly. Both algorithms permit such implicitly maintained paths to share "sub-paths": in Figure 3(c), $E\bullet$ in Σ_1 is reached by two CR-paths, Q_1: $(\bullet S \rightsquigarrow p \rightsquigarrow \bullet E \rightsquigarrow g \rightsquigarrow h \rightsquigarrow E\bullet)$, and Q_2: $(\bullet S \rightsquigarrow p \rightsquigarrow \bullet E \rightsquigarrow i \rightsquigarrow \bullet E \rightsquigarrow g \rightsquigarrow h \rightsquigarrow E\bullet)$, and they share the sub-path $(\bullet E \rightsquigarrow g \rightsquigarrow h \rightsquigarrow E\bullet)$. This path sharing permits Earley's algorithm to run in $O(|w|^3)$ time for any grammar (improved to $O(|w|^3/log|w|)$ by Graham *et al* [16]), and $O(|w|^2)$ time for any unambiguous grammar, as we show in Theorem 3.

Theorem 3. *For a given GFG $G = (V, E)$ and input word w, Earley's algorithm requires $O(|w|^2)$ space and $O(|w|^3)$ time. If the grammar is unambiguous, the time complexity is reduced to $O(|w|^2)$.*

Proof. See Section A.2

Earley Parser. The rules in Figure 4(b) define a recognizer. To get a parser, we need a way to enumerate a representation of the parse tree, such as a complete, balanced GFG path, from the Σ sets; if the grammar is ambiguous, there may be multiple complete, balanced paths that generate the input word.

Figure 4(c) shows a state transition system that constructs such a path in reverse; if there are multiple paths that generate the string, one of these paths is reconstructed non-deterministically. The parser starts with the entry $<S\bullet, 0>$ in the last Σ set, and reconstructs in reverse the inference chain that produced it from the entry $<\bullet S, 0>$ in Σ_0; intuitively, it traverses the GFG in reverse from $S\bullet$ to $\bullet S$, using the Σ set entries to guide the traversal. Like the NGA, it maintains a stack, but it pushes the matching $call$ node when it traverses a $return$ node, and pops the stack at a $start$ node to determine how to continue the traversal.

The state of the parser is a three-tuple: a Σ set entry, the number of that Σ set, and the stack. The parser begins at $<S\bullet, 0>$ in Σ_n and an empty stack. It terminates when it reaches $<\bullet S, 0>$ in Σ_0. The sequence of GFG nodes in the reverse path can be output during the execution of the transitions. It is easy to output other representations of parse trees if needed; for example, the parse tree can be produced in reverse post-order by outputting the terminal symbol or production name whenever a scan edge or exit node respectively is traversed in reverse by the parser.

To eliminate the need to look up Σ sets for the $\texttt{EXIT}^{-1}$ and $\texttt{END}^{-1}$ rules, the recognizer can save information relevant for the parser in a data structure associated with each Σ set. This data structure is a relation between the consequent and the premise(s)

of each rule application; given a consequent, it returns the premise(s) that produced that consequent during recognition. If the grammar is ambiguous, there may be multiple premise(s) that produced a given consequent, and the data structure returns one of them non-deterministically. By enumerating these non-deterministic choices, it is possible to enumerate different parse trees for the given input string. Note that if the grammar is cyclic (that is, $A \stackrel{+}{\rightarrow} A$ for some non-terminal A), there may be an infinite number of parse trees for some strings.

3.3 Discussion

In Earley's paper, the $call$ and $start$ rules were combined into a single rule called *prediction*, and the $exit$ and end rules were combined into a single rule called *completion* [3]. Aycock and Horspool pre-compute some of the contents of Σ-sets to improve the running time in practice [17].

Erasing tags from the rules in Figure 4(b) for the Earley recognizer produces the rules for the NFA ϵ-closure algorithm in Figure4(a). The only nontrivial erasure is for the end rule: k, the tag of the tuple $<B\bullet, k>$, becomes undefined when tags are deleted, so the antecedent $<A \rightarrow \alpha \bullet B\gamma, i> \in \Sigma_k$ for this rule is erased. Erasure of tags demonstrates lucidly the close and previously unknown connection between the NFA ϵ-closure algorithm and Earley's algorithm.

4 Preprocessing the GFG: Look-Ahead

Preprocessing the GFG is useful when many strings have to be parsed since the investment in preprocessing time and space is amortized over the parsing of multiple strings. *Look-ahead computation* is a form of preprocessing that permits pruning of the set of paths that need to be explored for a given input string.

Given a CR-path Q: $\bullet S \rightsquigarrow^* v$ which generates a string of terminals u, consider the set of all strings of k terminals that can be encountered along any CR extension of Q. When parsing a string $u\ell z$ with $\ell \in T^k$, extensions of path Q can be safely ignored if ℓ does not belong to this set. We call this set the *context-dependent look-ahead set* at v for path Q, which we will write as $CDL_k(Q)$ (in the literature on program optimization, Q is called the *calling context* for its last node v). LL(k) and LR(k) parsers use context-dependent look-ahead sets.

We note that for pruning paths, it is safe to use any superset of $CDL_k(Q)$: larger supersets may be easier to compute off-line, possibly at the price of less pruning on-line. In this spirit, a widely used superset is $FOLLOW_k(v)$, associated with GFG node v, which we call the *context-independent look-ahead set*. It is the union of the sets $CDL_k(Q)$, over all CR-paths Q: $\bullet S \rightsquigarrow^* v$. Context-independent look-ahead is used by SLL(k) and SLR(k) parsers. It has also been used to enhance Earley's algorithm. Look-ahead sets intermediate between $CDL_k(Q)$ and $FOLLOW_k(v)$ have also been exploited, for example in LALR(k) and LALL(k) parsers [10].

The presentation of look-ahead computations algorithms is simplified if, at every stage of parsing, there is always a string ℓ of k symbols that has not yet been read. This can be accomplished by (i) padding the input string w with k \$ symbols to form $w\$^k$,

where $\$ \notin (T + N)$ and (ii) replacing $\Gamma = <N, T, P, S>$, with the *augmented grammar* $\Gamma' = <N' = N \cup \{S'\}, T' = T \cup \{\$\}, P' = P \cup \{S' \rightarrow S\$^k\}, S'>$.

Figure 5(a) shows an example using a stylized GFG, with node labels omitted for brevity. The set $FOLLOW_2(v)$ is shown in braces next to node v. If the word to be parsed is ybc, the parser can see that $yb \notin FOLLOW_2(v)$ for $v = S \rightarrow \bullet yLab$, so it can avoid exploration downstream of that node.

The influence of context is illustrated for node $v = L \rightarrow \bullet a$, in Figure 5(a). Since the end node $L\bullet$ is reached before two terminal symbols are encountered, it is necessary to look beyond node $L\bullet$, but the path relevant to look-ahead depends on the path that was taken to node $\bullet L$. If the path taken was Q: $\bullet S' \rightsquigarrow^* (S \rightarrow y \bullet Lab) \rightsquigarrow (\bullet L) \rightsquigarrow L \rightarrow \bullet a$, then relevant path for look-ahead is $L\bullet \rightsquigarrow (S \rightarrow yL \bullet ab) \rightsquigarrow^* S'\bullet$, so that $CDL_2(Q) = \{aa\}$. If the path taken was R: $\bullet S' \rightsquigarrow^* (S \rightarrow y \bullet Lbc) \rightsquigarrow (\bullet L) \rightsquigarrow L \rightarrow \bullet a$, then the relevant path for look-ahead is $(L\bullet) \rightsquigarrow (S \rightarrow yL \bullet bc) \rightsquigarrow^* S'\bullet$, and $CDL_2(R) = \{ab\}$.

We define these concepts formally next.

Definition 8. Context-dependent look-ahead: *If v is a node in the GFG of an augmented grammar $\Gamma' = <N', T', P', S'>$, the context-dependent look-ahead $CDL_k(Q)$ for a CR-path Q: $\bullet S' \rightsquigarrow^* v$ is the set of all k-prefixes of strings generated by paths $Q_s : v \rightsquigarrow^* S'\bullet$ where $Q + Q_s$ is a complete CR-path.*

Definition 9. Context-independent look-ahead: *If v is a node in the GFG for an augmented grammar $\Gamma' = <N', T', P', S'>$, $FOLLOW_k(v)$ is the set of all k-prefixes of strings generated by CR-paths $v \rightsquigarrow^* S'\bullet$.*

As customary, we let $FOLLOW_k(A)$ and $FOLLOW(A)$ respectively denote $FOLLOW_k(A\bullet)$ and $FOLLOW_1(A\bullet)$.

The rest of this section is devoted to the computation of look-ahead sets. It is convenient to introduce the function $s_1 +_k s_2$ of strings s_1 and s_2, which returns their concatenation truncated to k symbols. In Definition 10, this operation is lifted to sets of strings.

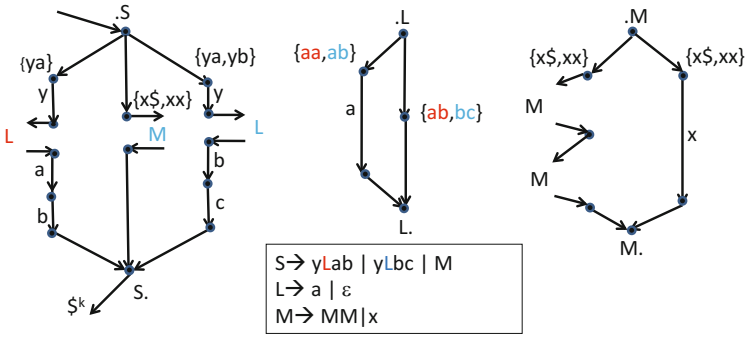
Definition 10. *Let T^* denote the set of strings of symbols from alphabet T.*

- *For $E \subseteq T^*$, $(E)_k$ is set of k-prefixes of strings in E.*
- *For $E, F \in T^*$, $E +_k F = (E + F)_k$.*

If $E = \{\epsilon, t, tu, abc\}$ and $F = \{\epsilon, x, xy, xya\}$, $(E)_2 = \{\epsilon, t, tu, ab\}$ and $(F)_2 = \{\epsilon, x, xy\}$. $E +_2 F = (E+F)_2 = \{\epsilon, x, xy, t, tx, tu, ab\}$. Lemma 1(a) says that concatenation followed by truncation is equivalent to "pre-truncation" followed by concatenation and truncation; this permits look-ahead computation algorithms to work with strings of length at most k throughout the computation rather than with strings of unbounded length truncated to k only at the end.

Lemma 1. *Function $+_k$ has the following properties.*

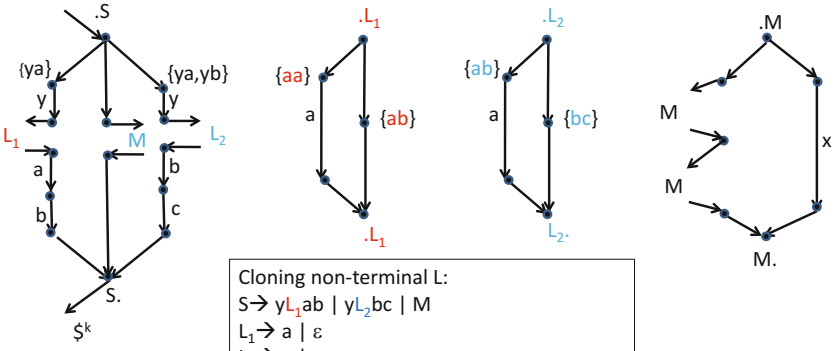
(a) $E +_k F = (E)_k +_k (F)_k$.
(b) $+_k$ *is associative and distributes over set union.*



$FIRST_2(S) = (y+_2 FIRST_2(L)+_2 \{ab\}) \cup (y+_2 FIRST_2(L)+_2 \{bc\}) \cup FIRST_2(M)$
$FIRST_2(L) = \{a\} \cup \{\varepsilon\}$
$FIRST_2(M) = \{x\} \cup (FIRST_2(M)+_2 FIRST_2(M))$
Solution:
$FIRST_2(S) = \{ya,yb,x,xx\}$ $FIRST_2(M) = \{x,xx\}$ $FIRST_2(L) = \{a,\varepsilon\}$

$FOLLOW_2(S) = \{\$\$\}$
$FOLLOW_2(L) = \{ab\} \cup \{bc\}$
$FOLLOW_2(M) = FOLLOW_2(S) \cup (FIRST_2(M) +_2 FOLLOW_2(M)) \cup FOLLOW_2(M))$
Solution:
$FOLLOW_2(S) = \{\$\$\}$ $FOLLOW_2(L) = \{ab,bc\}$ $FOLLOW_2(M) = \{\$\$,xx,x\$\}$

(a) $FIRST_2$ and $FOLLOW_2$ computation



(b) Partial and full 2-look-ahead cloning

Fig. 5. Look-head computation example

4.1 Context-Independent Look-Ahead

$FOLLOW_k(v)$ can be computed by exploring CR-paths from v to $S'\bullet$. However, for the "bulk" problem of computing these sets for *many* GFG nodes, such as all $entry$ nodes in a GFG, coordination of path explorations at different nodes can yield greater efficiency.

Although we do not use this approach directly, the GFG permits $FOLLOW_k$ computation to be viewed as an *inter-procedural backward dataflow analysis problem* [11]. Dataflow facts are possible $FOLLOW_k$ sets, which are the subsets of T^k, and which form a finite lattice under subset ordering (the empty set is the least element). For an edge e with label t, the dataflow transfer function $F_e(X)$ is $\{t\} +_k X$ (for ϵ edges, this reduces to the identity function as expected). For a path Q with edges labeled $t_1, ...t_n$, the composite transfer function is $(\{t_1\} +_k (\{t_2\} +_k ...(\{t_n\} +_k X))$, which can be written as $(\{t_1\} +_k \{t_2\} +_k ...\{t_n\}) +_k X$ because $+_k$ is associative. If we denote the k-prefix of the terminal string generated by Q by $FIRST_k(Q)$, the composite transfer function for a path Q is $FIRST_k(Q) +_k X$. The confluence operator is set union. To ensure that dataflow information is propagated only along (reverse) CR-paths, it is necessary to find *inter-procedural summary functions* that permit look-ahead sets to be propagated directly from a $return$ node to its matching $call$ node. These summary functions are hard to compute for most dataflow problems but this is easy for $FOLLOW_k$ computation because the lattice L is finite, the transfer functions distribute over set union, and the $+_k$ operation is associative. For a non-terminal A, the summary function is $F_A(X) = FIRST_k(A) +_k X$, where $FIRST_k(A)$ is the set of k-prefixes of terminal strings generated by balanced paths from $\bullet A$ to $A\bullet$. The $FIRST_k$ relation can be computed efficiently as described in Section 4.1. This permits the use of the *functional approach* to inter-procedural dataflow analysis [11] to solve the $FOLLOW_k$ computation problem (the development below does not rely on any results from this framework).

***$FIRST_k$* Computation.** For Γ=<N, T, P, S>, $FIRST_k(A)$ for $A \in N$ is defined canonically as the set of k-prefixes of terminal strings derived from A [10]. This is equivalent to the following, as we show in Theorem 4.

Definition 11. *Given a grammar Γ=<N, T, P, S>, a positive integer k and $A \in N$, $FIRST_k(A)$ is the set of k-prefixes of terminal strings generated by balanced paths from $\bullet A$ to $A\bullet$.*

Following convention, we write $FIRST(A)$ to mean $FIRST_1(A)$.

Definition 12. *$FIRST_k$ is extended to a string $u_1u_2...u_n \in (N \cup T)^*$ as follows.*

$$FIRST_k(\epsilon) = \{\epsilon\}$$
$$FIRST_k(t \in T) = \{t\}$$
$$FIRST_k(u_1u_2...u_n) = FIRST_k(u_1) +_k ... +_k FIRST_k(u_n)$$

$FIRST_k$ sets for non-terminals can be computed as the least solution of a system of equations derived from the grammar.

Algorithm 1. *For Γ=<N, T, P, S> and positive integer k, let $\mathcal{M}$ be the finite lattice whose elements are sets of terminal strings of length at most k, ordered by containment with the empty set being the least element. The $FIRST_k$ sets for the non-terminals are given by the least solution in $\mathcal{M}$ of this equational system:*

$$\forall A \in N \;\; FIRST_k(A) = \bigcup_{A \to \alpha} FIRST_k(\alpha)$$

Figure 5(a) shows an example.

$FOLLOW_k$ Computation

Algorithm 2. *Given an augmented grammar Γ'=<N', T', P', S'> and positive integer k, let $\mathcal{L}$ be the lattice whose elements are sets of terminal strings of length k, ordered by containment with the empty set being the least element. The $FOLLOW_k$ sets for non-terminals other than S' are given by the least solution of this equational system:*

$$FOLLOW_k(S) = \{\$^k\}$$
$$\forall B \in N - \{S, S'\}. FOLLOW_k(B) = \bigcup_{A \to \alpha B \gamma} FIRST_k(\gamma) +_k FOLLOW_k(A)$$

Given $FOLLOW_k$ sets for non-terminals, $FOLLOW_k$ sets at all GFG nodes are computed by interpolation:
$FOLLOW_k(A \to \alpha \bullet \beta) = FIRST_k(\beta) +_k FOLLOW_k(A)$.

Figure 5(a) shows an example. M occurs in three places on the righthand sides of the grammar productions, so the righthand side of the equation for $FOLLOW_k(M)$ is the union of three sets: the first from $S \to M\bullet$, the second from $M \to M\bullet M$, and the third from $M \to MM\bullet$.

Using Context-Independent Look-Ahead in the Earley Parser. Some implementations of Earley's parser use a context-independent look-ahead of one symbol at $start$ nodes and end nodes (this is called *prediction look-ahead* and *completion look-ahead* respectively) [3]. The practical benefit of using look-ahead in the Earley parser has been debated in the literature. The implementation of Graham *et al* does not use look-ahead [16]; other studies argue that some benefits accrue from using prediction look-ahead [5]. Prediction look-ahead is implemented by modifying the START rule in Figure 4(b):the production $B \to \beta$ is explored only if β might produce the empty string or a string that starts with the first look-ahead symbol. For this, the following formula is added to the antecedents of the START rule: $(\epsilon \in FIRST(\beta)) \vee (C^j = u.tv \wedge t \in FIRST(\beta))$.

Completion look-ahead requires adding the following check to the antecedents of the END rule in Figure 4(b):
$(C^j = u.tv) \wedge (t \in FIRST(\gamma) \vee (\epsilon \in FIRST(\gamma) \wedge t \in FOLLOW(A)))$.

4.2 Context-Dependent Look-Ahead

LL(k) and LR(k) parsers use context-dependent k-look-ahead. As one would expect, exploiting context enables a parser to rule out more paths than if it uses context-independent look-ahead. One way to implement context-dependent look-ahead for a grammar Γ is to reduce it to the problem of computing context-independent look-ahead for a related grammar Γ^c through an operation similar to *procedure cloning*.

In general, *cloning a non-terminal* A in a grammar Γ creates a new grammar in which (i) non-terminal A is replaced by some number of new non-terminals A_1,A_2,...A_c ($c \geq 2$) with the same productions as A, and (ii) all occurrences of A in the righthand sides of productions are replaced by some A_j ($1 \leq j \leq c$). Figure 5(b) shows the result of cloning non-terminal L in the grammar of Figure 5(a) into two new non-terminals L_1, L_2. Cloning obviously does not change the language recognized by the grammar.

The intuitive idea behind the use of cloning to implement context-dependent look-ahead is to create a cloned grammar that has a copy of each production in Γ for each context in which that production may be invoked, so as to "de-alias" look-ahead sets. In general, it is infeasible to clone a non-terminal for every one of its calling contexts, which can be infinitely many. Fortunately, contexts with the same look-ahead set can be represented by the same clone. Therefore, the number of necessary clones is bounded by the number of possible k-look-ahead sets for a node, which is $2^{|T|^k}$. Since this number grows rapidly with k, cloning is practical only for small values of k, but the principle is clear.

Algorithm 3. *Given an augmented grammar* $\Gamma'=(N', T', P', S')$*, and a positive integer* k*,* $T_k(\Gamma')$ *is following grammar:*

- *Nonterminals:* $\{S'\} \cup \{[A, R] | A \in (N'-S'), R \subseteq T'^k\}$
- *Terminals: T'*
- *Start symbol:* S'
- *Productions:*
  - $S' \rightarrow \alpha$ *where* $S' \rightarrow \alpha \in \Gamma'$
  - *all productions* $[A, R] \rightarrow Y_1 Y_2 ... Y_m$ *where for some* $A \rightarrow X_1 X_2 X_3 ... X_m \in P'$
    $Y_i = X_i$ *if* X_i *is a terminal, and*
    $Y_i = [X_i, FIRST_k(X_{i+1} ... X_m) +_k R]$ *otherwise.*

Therefore, to convert the context-dependent look-ahead problem to the context-independent problem, cloning is performed as follows. For a given k, each non-terminal A in the original grammar is replaced by a set of non-terminals $[A, R]$ for every $R \subseteq T^k$ (intuitively, R will end up being the context-independent look-ahead at $[A, R]\bullet$ in the cloned grammar). The look-ahead R is then interpolated into each production of A to determine the new productions as shown in Algorithm 3.Figure 5(b) shows the result of full 2-look-ahead cloning of the grammar in Figure 5(a) after useless non-terminals have been removed.

5 Related Work

The connection between context-free grammars and procedure call/return in programming languages was made in the early 1960's when the first recursive-descent parsers

were developed. The approach taken in this paper is to formulate parsing problems as path problems in the GFG, and the procedure call/return mechanism is used only to build intuition.

In 1970, Woods defined a generalization of finite-state automata called *recursive transition networks* (RTNs) [7]. Perlin defines an RTN as "..a forest of disconnected transition networks, each identified by a nonterminal label. All other labels are terminal labels. When, in traversing a transition network, a nonterminal label is encountered, control recursively passes to the beginning of the correspondingly labeled transition network. Should this labeled network be successfully traversed, on exit, control returns back to the labeled calling node" [18]. The RTN was the first graphical representation of context-free grammars, and all subsequent graphical representations including the GFG are variations on this theme. Notation similar to GFG $start$ and end nodes was first introduced by Graham *et al* in their study of the Earley parser [16].

The RTN with this extension is used in the ANTLR system for LL(*) grammars [8].

The key difference between RTNs and GFGs is in the *interpretation* of the graphical representation. An interpretation based on a single locus of control that flows between productions is adequate for SLL(k)/LL(k)/LL(*) languages but inadequate for handling more general grammars for which multiple paths through the GFG must be followed, so some notion of multiple threads of control needs to be added to the basic interpretation of the RTN. For example, Perlin models LR grammars using a chart parsing strategy in which portions of the transition network are copied dynamically [18]. In contrast, the GFG is a *single* graph, and all parsing problems are formulated as path problems in this graph; there is no operational notion of a locus of control that is transferred between productions. In particular, the similarity between Earley's algorithm and the NFA simulation algorithm emerges only if parsing problems are framed as path problems in a single graph. We note that the importance of the distinction between the two viewpoints was highlighted by Sharir and Pnueli in their seminal work on inter-procedural dataflow analysis [11].

The logic programming community has explored the notion of "parsing as deduction" [19,20,21] in which the rules of the Earley recognizer in Figure 4(b) are considered to be inference rules derived from a grammar, and recognition is viewed as the construction of a proof that a given string is in the language generated by that grammar. The GFG shows that this proof construction can be interpreted as constructing complete balanced paths in a graphical representation of the grammar.

An important connection between inter-procedural dataflow analysis and reachability computation was made by Yannakakis [9], who introduced the notion of *CFL-paths*. Given a graph with labeled edges and a context-free grammar, CFL-paths are paths that generate strings recognized by the given context-free grammar. Therefore, the context-free grammar is external to the graph, whereas the GFG is a direct representation of a context-free grammar with labeled nodes ($start$ and end nodes must be known) and labeled edges. If node labels are erased from a GFG and CFL-paths for the given grammar are computed, this set of paths will include all the complete balanced paths but in general, it will also include non-CR-paths that happen to generate strings in the language recognized by the context-free grammar.

6 Conclusions

In other work, we have shown that the GFG permits an elementary presentation of LL, SLL, LR, SLR, and LALR grammars in terms of GFG paths. These results and the results in this paper suggest that the GFG can be a new foundation for the study of context-free grammars.

Acknowledgments. We would like to thank Laura Kallmeyer for pointing us to the literature on parsing in the logic programming community, and Giorgio Satta and Lillian Lee for useful discussions about parsing.

References

1. Reynolds, J.C.: On the relation between direct and continuation semantics. In: Loeckx, I.J. (ed.) ICALP 1974. LNCS, vol. 14, pp. 141–156. Springer, Heidelberg (1974)
2. Sussman, G., Steele, G.: Scheme: An interpreter for extended lambda calculus. Technical Report AI Memo 349, AI Lab, M.I.T. (December 1975)
3. Earley, J.: An efficient context-free parsing algorithm. Commun. ACM 13(2), 94–102 (1970)
4. Aho, A., Lam, M., Sethi, R., Ullman, J.: Compilers: principles, techniques, and tools. Addison Wesley (2007)
5. Grune, D., Jacobs, C.: Parsing Techniques: A practical guide. Springer (2010)
6. Cousot, P., Cousot, R.: Parsing as abstract interpretation of grammar semantics. Theoret. Comput. Sci. 290, 531–544 (2003)
7. Woods, W.A.: Transition network grammars for natural language analysis. Commun. ACM 13(10) (October 1970)
8. Parr, T., Fisher, K.: LL(*): The foundation of the ANTLR parser generator. In: PLDI (2011)
9. Yannakakis, M.: Graph-theoretic methods in database theory. In: Principles of Database Systems (1990)
10. Sippu, S., Soisalon-Soininen, E.: Parsing theory. Springer (1988)
11. Sharir, M., Pnueli, A.: Two approaches to interprocedural dataflow analysis. In: Program Flow Analysis: Theory and Applications, pp. 189–234. Prentice-Hall (1981)
12. Cormen, T., Leiserson, C., Rivest, R., Stein, C. (eds.): Introduction to Algorithms. MIT Press (2001)
13. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation, 3rd edn. Addison-Wesley Longman Publishing Co., Inc., Boston (2006)
14. Clinger, W.D., Halpern, C.: Alternative semantics for mccarthy's amb. In: Brookes, S.D., Roscoe, A.W., Winskel, G. (eds.) Seminar on Concurrency. LNCS, vol. 197, pp. 467–478. Springer, Heidelberg (1985)
15. Terese: Term Rewriting Systems. Combridge University Press (2003)
16. Graham, S.L., Ruzzo, W.L., Harrison, M.: An improved context-free recognizer. ACM TOPLAS 2(3), 415–462 (1980)
17. Aycock, J., Horspool, N.: Practical Earley parsing. The Computer Journal 45(6), 620–630 (2002)
18. Perlin, M.: LR recursive transition networks for Earley and Tomita parsing. In: ACL 1991 (1991)
19. Pereira, F.C.N., Warren, D.: Parsing as deduction. In: 21st Annual Meeting of the Association for Computational Linguistics, pp. 137–144. MIT, Cambridge (1983)
20. Shieber, S.M., Schabes, Y., Pereira, F.C.N.: Principles and implementation of deductive parsing. Journal of Logic Programming 24(1 & 2), 3–36 (1995)
21. Sikkel, K.: Parsing Schemata. Texts in Theoretical Computer Science. Springer, Heidelberg (1997)

A Appendix

A.1 Derivations, Parse Trees and GFG Paths

The following result connects complete balanced paths to parse trees.

Theorem 4. *Let Γ=<N, T, P, S> be a context-free grammar and $G = GFG(\Gamma)$ the corresponding grammar flow graph. Let $A \in N$. There exists a balanced path from $\bullet A$ to $A\bullet$ with n_{cr} call-return pairs that generates a string $w \in T^*$ if and only if there exists a parse tree for w with $n_{int} = n_{cr} + 1$ internal nodes.*

Proof. We proceed by induction on n_{cr}. The base case, $n_{cr} = 0$, arises for a production $A \rightarrow u_1 u_2 \dots u_r$ where each u_j is a terminal. The GFG balanced path contains the sequence of nodes

$$\bullet A, A \rightarrow \bullet u_1 u_2 \dots u_r, \dots A \rightarrow u_1 u_2 \dots u_r \bullet, A\bullet$$

The corresponding parse tree has a root with label A and r children respectively labeled $u_1, u_2, \dots, u_r$ (from left to right), with $n_{int} = 1$ internal node. The string generated by the path and derived from the tree is $w = u_1 u_2 \dots u_r$.

Assume now inductively the stated property for paths with fewer than n_{cr} call-return pairs and trees with fewer than n_{int} internal nodes. Let Q be a path from $\bullet A$ to $A\bullet$ with n_{cr} call-return pairs. Let $A \rightarrow u_1 u_2 \dots u_r$ be the "top production" used by Q, *i.e.*, the second node on the path is $A \rightarrow \bullet u_1 u_2 \dots u_r$. If $u_j \in N$, then Q will contain a segment of the form

$$A \rightarrow u_1 \dots u_{j-1} \bullet u_j \dots u_r, Q_j, A \rightarrow u_1 \dots u_j \bullet u_{j+1} \dots u_r$$

where Q_j is a balanced path from $\bullet u_j$ to $u_j \bullet$, generating some word w_j. Let $\mathcal{T}_j$ be a parse tree for w_j with root labeled u_j, whose existence follows by the inductive hypothesis. If instead $u_j \in T$, then Q will contain the scan edge

$$(A \rightarrow u_1 \dots u_{j-1} \bullet u_j \dots u_r, A \rightarrow u_1 \dots u_j \bullet u_{j+1} \dots u_r)$$

generating the word $w_j = u_j$. Let $\mathcal{T}_j$ be a tree with a single node labeled $w_j = u_j$. The word generated by Q is $w = w_1 w_2 \dots w_r$. Clearly, the tree $\mathcal{T}$ with a root labeled A and r subtrees equal (from left to right) to $\mathcal{T}_1, \mathcal{T}_2, \dots, \mathcal{T}_r$ derives string w. Finally, it is simple to show that $\mathcal{T}$ has $n_{int} = n_{cr} + 1$ internal nodes.

The construction of a balanced path generating w from a tree deriving w follows the same structure.

A.2 Correctness and Complexity of Earley's Algorithm

The following result is an "inductive version" of Theorem 2, which asserts the correctness of the rules for the Earley parser.

Theorem 5. *Consider the execution of Earley's algorithm on input string $w = a_1 a_2 \dots a_n$. Let z be a GFG node and i and j be integers such that $0 \leq i \leq j \leq n$.*

The following two properties are equivalent.

(A) The algorithm creates an entry $<z, i>$ in Σ_j.

(B) There is a CR-path $Q = (\bullet S)Q'z$ (represented as a sequence of GFG nodes beginning at $\bullet S$ and ending at z) that generates $a_1 a_2 \ldots a_j$ and whose prefix preceding the last unmatched call edge generates $a_1 a_2 \ldots a_i$.

Proof. Intuitively, the key fact is that each rule of Earley's algorithm (aside from initialization) uses an entry $<y, i'> \in \Sigma_{j'}$ and a GFG edge (y, z) to create an entry $<z, i> \in \Sigma_j$, where the dependence of i and j upon i' and j' depends on the type of edge (y, z). For a return edge, a suitable entry $<z', k> \in \Sigma_{i'}$ is also consulted. In essence, if a CR-path can be extended by an edge, then (and only then) the appropriate rule creates the entry for the extended path. The formal proof is an inductive formulation of this intuition and carries out a case analysis with respect to the type of edge that extends the path.

Part I. $B \Rightarrow A$ (*from CR-path to Earley entry*). The argument proceeds by induction on the length (number of edges) ℓ of path Q.
- Base cases ($\ell = 0, 1$).
The only path with no edges is $Q = (\bullet S)$, for which $i = j = 0$. The INIT rule produces the corresponding entry $<\bullet S, 0> \in \Sigma_0$. The paths with just one edge are also easily dealt with, as they are of the form $Q = (\bullet S)(S \rightarrow \bullet\sigma)$, that is, they contain one ENTRY edge.
- Inductive step (from $\ell - 1 \geq 1$ to ℓ).
Consider a CR-path $Q = (\bullet S)Ryz$ of length ℓ. It is straightforward to check that $Q' = (\bullet S)Ry$ is also a CR-path, of length $\ell - 1$. Hence, by the inductive hypothesis, an entry $<y, i'>$ is created by the algorithm in some $\Sigma_{j'}$, with Q' generating $a_1 a_2 \ldots a_{j'}$ and with the prefix of Q' preceding its last unmatched call edge generating $a_1 a_2 \ldots a_{i'}$.

Inspection of the rules for the Earley parser in Figure 4 reveals that, given $<y, i'> \in \Sigma_{j'}$ and given the presence edge (y, z) in the CR-path Q, an entry $<z, i> \in \Sigma_j$ is always created by the algorithm. It remains to show that i and j have, with respect to path Q, the relationship stated in property (B).

- Frame number j. We observe that the string of terminals generated by Q is the same as the string generated by Q', except when (y, z) is a scan edge, in which case Q does generate $a_1 a_2 \ldots a_{j'+1}$. Correspondingly, the algorithm sets $j = j'$, except when (y, z) is a scan edge, in which case it sets $j = j' + 1$.

- Tag i. We distinguish three cases, based on the type of edge.

- When (y, z) is an *entry*, *scan*, or *exit* edge, Q has the same last unmatched call edge as Q'. Correspondingly, $i = i'$.
- When (y, z) is a *call* edge, then (y, z) is the last unmatched call edge on Q. The algorithm correctly sets $i = j' = j$.
- Finally, let (y, z) be a *return* edge, with $y = B\bullet$ and $z = A \rightarrow \alpha B\bullet\gamma$. Since Q is a CR-path, (y, z) must match the last unmatched call edge in Q', say, (z', y'), with $z' = A \rightarrow \alpha\bullet B\gamma$, and $y' = \bullet B$. We can then write $Q = (\bullet S)Q_1 z' y' Q_2 yz$ where Q_2 is balanced, whence Q and $(\bullet S)Q_1 z'$ have the same last unmatched call edge,

say (u, v). Let i' be such that the prefix of Q ending at z' generates $a_1 a_2 \dots a_{i'}$ and let $k \leq i'$ be such that the prefix of Q ending at u generates $a_1 a_2 \dots a_k$. By the inductive hypothesis, corresponding to path $(\bullet S)Q_1 z'$, the algorithm will have created entry $<z' = A \to \alpha \bullet B\gamma, k> \in \Sigma_{i'}$. From entries $<y = B\bullet, i'> \in \Sigma_{j'}$ and $<z' = A \to \alpha \bullet B\gamma, k> \in \Sigma_{i'}$ as well as from return edge (y, z), the END rule of the algorithm, as written in Figure 4, creates $<z = A \to \alpha B \bullet \gamma, i = i'> \in \Sigma_j$.

Part II. $A \Rightarrow B$ (*from Earley entry to CR-path*). The argument proceeds by induction on the number q of rule applications executed by the algorithm when entry $<z, i>$ is first added to Σ_j. (Further "discoveries" that $<z, i> \in \Sigma_j$ are possible, but the entry is added only once.)
- Base case ($q = 1$). The only rule applicable at first is INIT, creating the entry $<\bullet S, 0> \in \Sigma_0$, whose corresponding path is clearly $Q = (\bullet S)$.
- Inductive step (from $q - 1 \geq 1$ to q). Let the q-th rule application of the algorithm be based on GFG edge (y, z) and on entry $<y, i'> \in \Sigma_{j'}$. Also let $<z, i> \in \Sigma_j$ be the entry created by the algorithm as a result of said rule application. By the inductive hypothesis, there is a CR-path $(\bullet S)Q'y$ generating $a_1 a_2 \dots a_{j'}$ and with the prefix of Q' preceding its last unmatched call edge generating $a_1 a_2 \dots a_{i'}$. To show that to entry $<z, i> \in \Sigma_j$ there corresponds a CR-path Q as in (B), we consider two cases, based on the type of edge (y, z).

- When (y, z) is an *entry*, *scan*, *exit* or *call* edge, we consider the path $Q = (\bullet S)Q'yz$. Arguments symmetric to those employed in Part I of the proof show that path the Q does satisfy property (B), with exactly the values i and j of the entry $<z, i> \in \Sigma_j$ produced by the algorithm.
- When (y, z) is a *return* edge, the identification of path Q requires more care. Let $y = B\bullet$ and $z = A \to \alpha B \bullet \gamma$. The END rule of Earley's algorithm creates entry $<z, i> \in \Sigma_j$ based on two previously created entries to each of which, by the inductive hypothesis, there corresponds a path, as discussed next.
  To entry $<y = B\bullet, k> \in \Sigma_j$, there correspond a CR-path of the form $Q' = (\bullet S)Q'_1 x' y' Q_2 y$, with last unmatched call edge (x', y'), where $y' = \bullet B$ and Q_2 is balanced.
  To entry $<z' = A \to \alpha \bullet B\gamma, i> \in \Sigma_k$ there correspond a CR-path of the form $Q'' = (\bullet S)Q_1 z'$, where $z' = A \to \alpha \bullet B\gamma$.
  From the above two paths, as well as from return edge (y, z), we can form a third CR-path $Q = (\bullet S)Q_1 z' y' Q_2 yz$. We observe that is is legitimate to concatenate $(\bullet S)Q_1 z'$ with $y'Q_2 y$ via the call edge (z', y') since $y'Q_2 y$ is balanced. It is also legitimate to append return edge (y, z) to $(\bullet S)Q_1 z' y' Q_2 y$ (thus obtaining Q), since such edge does match (z', y'), the last unmatched call edge of said path.
  It is finally straightforward to check that the frame number j and the tag i are appropriate for Q.

Proof of Theorem 3. For a given GFG $G = (V, E)$ and input word w, Earley's algorithm requires $O(|w|^2)$ space and $O(|w|^3)$ time. If the grammar is unambiguous, the time complexity is reduced to $O(|w|^2)$.

Proof. – *Space complexity*: There are $|w|+1$ Σ-sets, and each Σ-set can have at most $|V||w|$ elements since there are $|w|+1$ possible tags. Therefore, the space complexity of the algorithm is $O(|w|^2)$.

– *Time complexity*: For the time complexity, we need to estimate the number of distinct rule instances that can be invoked and the time to execute each one (intuitively, the number of times each rule can "fire" and the cost of each firing).
For the time to execute each rule instance, we note that the only non-trivial rule is the end rule: when $<B\bullet, k>$ is added to Σ_j, we must look up Σ_k to find entries of the form $<A\rightarrow\alpha\bullet B\gamma, i>$. To permit this search to be done in constant time per entry, we maintain a data structure with each Σ set, indexed by a non-terminal, which returns the list of such entries for that non-terminal. Therefore, all rule instances can be executed in constant time per instance.
We now compute an upper bound on the number of distinct rule instances for each rule schema. The $init$ rule schema has only one instance. The $start$ rule schema has a two parameters: the particular $start$ node in the GFG at which this rule schema is being applied and the tag j, and it can be applied for each outgoing edge of that $start$ node, so the number of instances of this rule is $O(|V|*|V|*|w|)$; for a given GFG, this is $O(|w|)$.
Similarly, the end rule schema has four parameters: the particular end node in the GFG, and the values of i, j, k; the relevant $return$ node is determined by these parameters. Therefore, an upper bound on the number of instances of this schema is $O(|V||w|^3)$, which is $O(|w|^3)$ for a given GFG.
A similar argument shows that the complexity of $call$, $exit$ and $scan$ rule schema instances is $O(|w|^2)$.
Therefore the complexity of the overall algorithm is $O(|w|^3)$.

– *Unambiguous grammar*: As shown above, the cubic complexity of Earley's algorithm arises from the end rule. Consider the consequent of the end rule. The proof of Theorem 2 shows that $<A\rightarrow\alpha B\bullet\gamma, i> \in \Sigma_j$ iff $w[i..(j-1)]$ can be derived from αB. If the grammar is unambiguous, there can be only one such derivation; considering the antecedents of the end rule, this means that for a given return node $A\rightarrow\alpha B\bullet\gamma$ and given values of i and j, there can be exactly one k for which the antecedents of the end rule are true. Therefore, for an unambiguous grammar, the end rule schema can be instantiated at most $O(|w|^2)$ times for a given grammar. Since all other rules are bounded above similarly, we conclude that Earley's algorithm runs in time $O(|w|^2)$ for an unambiguous grammar.

A.3 Look-Ahead Computation

Proof of correctness of Algorithm 1:

Proof. The system of equations can be solved using Jacobi iteration, with $FIRST_k(A) = \{\}$ as the initial approximation for $A \in N$. If the sequence of approximate solutions for the system is $\mathcal{X}_0; \mathcal{X}_1; ...$, the set $\mathcal{X}_i[A]$ $(i \geq 1)$ contains k-prefixes of terminal strings generated by balanced paths from $\bullet A$ to $A\bullet$ in which the number of call-return pairs is

at most $(i-1)$. Termination follows from monotonicity of set union and $+_k$, and finiteness of $\mathcal{M}$.

Proof of correctness of Algorithm 2:

Proof. The system of equations can be solved using Jacobi iteration. If the sequence of approximate solutions is $\mathcal{X}_0; \mathcal{X}_1; ...$, then $\mathcal{X}_i[B]$ $(i \geq 1)$ contains the k-prefixes of terminal strings generated by CR-paths from $B\bullet$ to $S'\bullet$ in which there are i or fewer unmatched $return$ nodes.