Chapter 10

IDL-Expressions: A Compact Representation for Finite Languages in Generation Systems

MARK-JAN NEDERHOF AND GIORGIO SATTA

ABSTRACT. We propose a representation for finite languages, called the class of *IDL-expressions*, which combines concepts that were only considered in isolation in previous representations. The suggested application is in generation systems where a sentence is obtained by first generating a large set of candidate sentences, represented in a compact way, and then filtering such a set through a parser. We study several formal properties of IDL-expressions, present a novel parsing algorithm for this representation and prove a non-trivial upper bound on its time complexity.

10.1 Introduction

During the last decade, an approach to natural language generation has been investigated that consists of two phases. In the first phase one generates a large set of candidate sentences by a relatively simple process. The set of candidate sentences may be produced on the basis of an input sentence in another language, in the case of machine translation, or more generally on the basis of some logical/semantic representation denoting the meaning that the output sentence should convey. This first phase involves no or only few intricacies of the target language, and the set of candidate sentences may contain many that are ungrammatical or that can otherwise be seen as less desirable than others.

In the second phase one or more preferred sentences are then selected from the collection of candidates, by using some form of syntactic processing that more heavily relies on properties of the target language than the first phase. This syntactic processing may involve language models as simple as bigrams or it may consist of parsing on the basis of more sophisticated models such as context-free grammars or even unification-based grammars. Since the set of candidates is typically very large, the feasibility of this second phase relies on the compactness of the representation of this set, and the ability of syntactic processing to benefit from this compactness.

One way to obtain a compact representation of a large set of sentences is to specify a bag (or multiset) of words that the output sentence should consist of,

Proceedings of Formal Grammar 2002
G. Jäger, P. Monachesi, G. Penn & S. Wintner (editors).

together with constraints on word order (Brew 1992; Whitelock 1992). If no constraints at all are present, then there are exponentially many candidate sentences that are consistent with a given bag, which shows that a bag can be a very compact representation. However, in this framework one cannot express alternatives for words or phrases.

A different approach is to produce an acyclic finite automaton, which allows a choice of phrases by alternative paths through the automaton (Knight and Hatzivassiloglou 1995; Langkilde and Knight 1998; Bangalore and Rambow 2000). However, by using this approach we cannot express that a bag of words or a bag of phrases can occur in many different orders, other than by expanding all these possibilities. Such expansion is less costly if we represent the structure of the acyclic finite automaton by means of a non-recursive context-free grammar, as proposed by Langkilde (2000), but that does not address the root of the problem, which is the lack of a primitive for indicating a permutation of words or phrases.

The main contribution of this article is twofold:

- we propose a new representation, called the class of **IDL-expressions**, that combines the two ideas of (i) specifying bags along with constraints on word order; and (ii) representing alternatives by different paths through a finite automaton;
- we develop a tabular algorithm for parsing our representation using a context-free grammar, and prove a non-trivial upper bound on its time complexity.

The paper is organized as follows. Section 10.2 introduces IDL-expressions and Section 10.3 reports our parsing algorithm; Section 10.4 investigates the time complexity of the algorithm and Section 10.5 concludes with some discussion. Most of the technical notation used in the paper will be introduced where appropriate. We also use the following conventions. For string x and languages L and L', we let $x \cdot L = \{xy \mid y \in L\}$ and $L \cdot L' = \{xy \mid x \in L, y \in L'\}$. We remind the reader that a function f from an alphabet V_T to V_T^* can be extended to a homomorphism over V_T^* by letting $f(\varepsilon) = \varepsilon$ and $f(ax) = f(a)f(x)$ for $a \in V_T$ and $x \in V_T^*$. We also let $f(L) = \{f(x) \mid x \in L\}$.

10.2 IDL-expressions

In this section we define the class of IDL-expressions. The main idea is to represent choices of words or phrases and their relative ordering by means of the standard concatenation operator $\cdot$, along with the following three additional operators. Operator $\|$, called **interleave**, interleaves strings resulting from its argument expressions; operator $\vee$, called **disjunction**, allows a choice between strings resulting from its argument expressions; operator $\times$, called **lock**, states that no additional material can be interleaved with a string resulting from its single argument expression. As an introductory example, a well-formed IDL-expression is

$$\|(\vee(\text{necessarily, must}), \text{we} \times(\text{play} \cdot \text{piano})).$$

The following sentences, among others, are satisfied by the above expression: "necessarily we play piano", "must we play piano", "we must play piano" and "we play piano necessarily". However, the following sentences do not satisfy the above expression: "we play necessarily piano" and "necessarily must we play piano".

Definition 1 *Let V_T be some finite alphabet, not containing symbol $\mathcal{E}$. An* **IDL-expression** *over V_T is a string π satisfying one of the following conditions:*

(i) $\pi = a$, *with* $a \in V_T \cup \{\mathcal{E}\}$;

(ii) $\pi = \times(\pi')$, *with* π' *an IDL-expression;*

(iii) $\pi = \vee(\pi_1, \pi_2, \ldots, \pi_n)$ *or* $\pi = \|(\pi_1, \pi_2, \ldots, \pi_n)$, *with* $n \geq 2$ *and* π_i *an IDL-expression for each i,* $1 \leq i \leq n$;

(iv) $\pi = \pi_1 \cdot \pi_2$, *with* π_1 *and* π_2 *both IDL-expressions.*

We now provide a precise semantics for IDL-expressions. The only technical difficulty in doing so arises with the proper treatment of the lock operator. Let x be a string over V_T. The basic idea below is to use a new symbol $\diamond$, not already in V_T, to mark every internal position of x where additional strings can be inserted. Consequently, the lock operator corresponds to the removal of every occurrence of $\diamond$ from a string.

Strings in $(V_T \cup \{\diamond\})^*$ will be used to represent sequences of strings over V_T; $\diamond$ is used to separate the strings in the sequence. Furthermore, we introduce a homomorphism lock over $(V_T \cup \{\diamond\})^*$ by letting $\mathsf{lock}(a) = a$ for $a \in V_T$ and $\mathsf{lock}(\diamond) = \varepsilon$. The operation lock can be seen as a concatenation of the strings in its input sequence.

We can now define a basic operation comb that plays an important role in the sequel. This operation composes a pair of sequences of strings, represented as explained above, into a set of new sequences. This is done by "interleaving" the two input sequences in every possible way. Operation comb makes use of an auxiliary operation comb', that attempts to construct a new sequence starting with the first string in its first argument. For $x, y \in (V_T \cup \{\diamond\})^*$, we have:

$$\begin{aligned}
\mathsf{comb}(x,y) &= \mathsf{comb}'(x,y) \cup \mathsf{comb}'(y,x) \\
\mathsf{comb}'(x,y) &= \begin{cases} \{x \diamond y\}, & \text{if } x \in V_T^*; \\ \{x' \diamond\} \cdot \mathsf{comb}(x'', y), & \text{if there are } x' \in V_T^* \text{ and } x'' \\ & \text{such that } x = x' \diamond x''. \end{cases}
\end{aligned}$$

As an example, we have $\mathsf{comb}(a \diamond bb \diamond c, d \diamond e) = \{a \diamond bb \diamond c \diamond d \diamond e,\ a \diamond bb \diamond d \diamond c \diamond e,\ a \diamond bb \diamond d \diamond e \diamond c,\ a \diamond d \diamond bb \diamond c \diamond e,\ a \diamond d \diamond bb \diamond e \diamond c,\ a \diamond d \diamond e \diamond bb \diamond c,\ d \diamond a \diamond bb \diamond c \diamond e,\ d \diamond a \diamond bb \diamond e \diamond c,\ d \diamond a \diamond e \diamond bb \diamond c,\ d \diamond e \diamond a \diamond bb \diamond c\}$.

For languages $L_1, L_2, \ldots, L_d$, $d \geq 2$, we define $\mathsf{comb}(L_1, L_2) = \cup_{u \in L_1, v \in L_2} \mathsf{comb}(u, v)$, $\mathsf{comb}_{i=1}^{d} L_i = \mathsf{comb}(L_1, L_2)$ for $d = 2$, and $\mathsf{comb}_{i=1}^{d} L_i = \mathsf{comb}(L_1, \mathsf{comb}_{i=2}^{d} L_i)$ for $d > 2$.

Definition 2 *Let V_T be some finite alphabet and let σ be a function mapping IDL-expressions over V_T to subsets of $(V_T \cup \{\diamond\})^*$, specified by the following conditions:*

(i) $\sigma(a) = \{a\}$ *for* $a \in V_T$, *and* $\sigma(\mathcal{E}) = \{\varepsilon\}$;

(ii) $\sigma(\times(\pi)) = \mathsf{lock}(\sigma(\pi))$;

(iii) $\sigma(\vee(\pi_1, \pi_2, \ldots, \pi_n)) = \cup_{i=1}^{n} \sigma(\pi_i)$;

(iv) $\sigma(\|(\pi_1, \pi_2, \ldots, \pi_n)) = \mathsf{comb}_{i=1}^{n} \sigma(\pi_i)$;

(v) $\sigma(\pi \cdot \pi') = \sigma(\pi) \diamond \sigma(\pi')$.

The set of strings that satisfy an IDL-expression π, written $L(\pi)$, is given by $L(\pi) =$ $\mathsf{lock}(\sigma(\pi))$.

10.3 CFG parsing for IDL-expressions

In this section we develop an algorithm to parse IDL-expressions according to a given context-free grammar. The basic idea is to associate each IDL-expression π with a finite automaton $\mu(\pi)$ that generates the language $L(\pi)$. The next step is then the formulation of a context-free parsing algorithm that takes the states of $\mu(\pi)$ as input positions in its parse items. Crucially, during the parsing process we only construct and cache the portion of the automaton that is visited by the context-free parser.

In what follows we deal with finite automata that are nondeterministic and have ε-transitions (see for instance Hopcroft and Ullman 1979) and always use a single final state. A finite automaton is denoted as $M = (Q, V_T, \Delta, q_0, q_f)$, with Q and V_T the finite set of states and the alphabet, respectively, q_0 and q_f the initial and final states, respectively, and Δ the set of transitions. Each transition is denoted as a triple (q, X, q'), $q, q' \in Q$ and $X \in V_T \cup \{\varepsilon\}$.

In the next definition, the automaton associated with IDL-expression π is constructed from the automata associated with π's immediate subexpressions. This is done by viewing states of the automaton as strings, and by applying standard string concatenation to combine old states into new states. Crucially, the definition "marks" some of the states of the automata it specifies. For each automaton M we denote by D_M the set of all its marked states. Informally, when in a marked state of M, we cannot interleave M's computation with any other computation.

Definition 3 *Let V_T be some finite alphabet. Each IDL-expression π over V_T is associated with a finite automaton $\mu(\pi) = (Q, V_T, \Delta, q_0, q_f)$ specified as follows.*

(i) *If* $\pi = a$, $a \in V_T \cup \{\mathcal{E}\}$, *we have* $Q = \{q_0, q_f\}$ *and* $\Delta = \{(q_0, X, q_f)\}$, *where* q_0 *and* q_f *are two distinct states, and* $X = a$ *if* $a \in V_T$ *and* $X = \varepsilon$ *if* $a = \mathcal{E}$. *We let* $D_{\mu(\pi)} = \emptyset$.

(ii) *If* $\pi = \times(\pi')$ *with* $\mu(\pi') = (Q', V_T, \Delta', q'_0, q'_f)$, *we have* $Q = Q' \cup \{q_0, q_f\}$ *and* $\Delta = \Delta' \cup \{(q_0, \varepsilon, q'_0), (q'_f, \varepsilon, q_f)\}$, *where* q_0 *and* q_f *are two distinct states not in* Q'. *We let* $D_{\mu(\pi)} = Q'$.

(iii) *If* $\pi = \vee(\pi_1, \pi_2, \ldots, \pi_n)$ *with* $\mu(\pi_i) = (Q_i, V_T, \Delta_i, q_{0i}, q_{fi})$, *we have* $Q = (\cup_{i=1}^n Q_i) \cup \{q_0, q_f\}$ *and* $\Delta = (\cup_{i=1}^n \Delta_i) \cup \{(q_0, \varepsilon, q_{0i}) \mid 1 \leq i \leq n\} \cup \{(q_{fi}, \varepsilon, q_f) \mid 1 \leq i \leq n\}$. *We assume all* Q_i, $1 \leq i \leq n$, *are pairwise disjoint, and* q_0 *and* q_f *are two distinct states not in* $\cup_{i=1}^n Q_i$. *We also let* $D_{\mu(\pi)} = \cup_{i=1}^n D_{\mu(\pi_i)}$.

(iv) *If* $\pi = \|(\pi_1, \pi_2, \ldots, \pi_n)$ *with* $\mu(\pi_i) = (Q_i, V_T, \Delta_i, q_{0i}, q_{fi})$, *we have*

$$Q = Q_1 Q_2 \cdots Q_n \cup \{q_0, q_f\}.$$

We again assume all Q_i, $1 \leq i \leq n$, *are pairwise disjoint, and* q_0 *and* q_f *are two distinct states not in* $\cup_{i=1}^n Q_i$. *Set* Δ *contains all and only the following transitions:*

(a) $(q_1 q_2 \cdots q_n, X, q_1 q_2 \cdots q_{i-1} q q_{i+1} \cdots q_n)$, *for each* $q_1 q_2 \cdots q_n \in Q_1 Q_2 \cdots Q_n$ *and for each* $(q_i, X, q) \in \Delta_i$, $1 \leq i \leq n$, *such that* $q_j \notin D_{\mu(\pi_j)}$ *for every* j *with* $1 \leq j \leq n$ *and* $j \neq i$;

(b) $(q_0, \varepsilon, q_{01} q_{02} \cdots q_{0n})$ *and* $(q_{f1} q_{f2} \cdots q_{fn}, \varepsilon, q_f)$.

For all $q_1 q_2 \cdots q_n \in Q$ *we let* $q_1 q_2 \cdots q_n \in D_{\mu(\pi)}$ *iff* $q_i \in D_{\mu(\pi_i)}$ *for some* i.

(v) *If* $\pi = \pi_1 \cdot \pi_2$ *with* $\mu(\pi_i) = (Q_i, V_T, \Delta_i, q_{0i}, q_{fi})$, $i = 1, 2$, *we have* $Q = Q_1 \cup Q_2 \cup \{q_0, q_f\}$ *and* $\Delta = \Delta_1 \cup \Delta_2 \cup \{(q_0, \varepsilon, q_{01}), (q_{f1}, \varepsilon, q_{02}), (q_{f2}, \varepsilon, q_f)\}$. *We assume* Q_1 *and* Q_2 *are disjoint and* q_0 *and* q_f *are two distinct states not in* $Q_1 \cup Q_2$. *We also let* $D_{\mu(\pi)} = D_{\mu(\pi_1)} \cup D_{\mu(\pi_2)}$.

Items (i), (iii) and (v) above closely resemble the corresponding steps in standard constructions of finite automata from regular expressions. Item (ii) marks every state of $\mu(\pi')$ to prevent interleaving with other computations. Item (iv) deserves some discussion here. This item defines an automaton $\mu(\pi)$ that simulates, in an interleaved way, the computations of the automata $\mu(\pi_i)$. As already mentioned, this is done by using states that are string concatenations of the states from the $\mu(\pi_i)$, $1 \leq i \leq n$. It is not difficult to see that for each state of the form $q_1 q_2 \cdots q_n$ that can be reached in $\mu(\pi)$ no more than one state among the q_i, $1 \leq i \leq n$, can be marked, and q_f can be reached from every such state $q_1 q_2 \cdots q_n$. The condition in (a) ensures that, in case a marked state exists among the q_i, we always select that state when making the next transition, so that interleaving with other computations is avoided. On the other hand, if no state among the q_i is marked then any of the q_i can be selected for the next transition, so that interleaving takes place.

$$\frac{}{[S \to \bullet\, \alpha, q_0, q_0]} \left\{\ S \to \alpha \right. \tag{10.1}$$

$$\frac{[A \to \alpha \bullet B\beta, q_1, q_2]}{[B \to \bullet\, \gamma, q_2, q_2]} \left\{\ B \to \gamma \right. \tag{10.2}$$

$$\frac{[A \to \alpha \bullet a\beta, q_1, q_2]}{[q_2, q_2]} \tag{10.3}$$

$$\frac{\begin{array}{l}[A \to \alpha \bullet a\beta, q_1, q_2] \\ [a, q_2, q_3]\end{array}}{[A \to \alpha a \bullet \beta, q_1, q_3]} \tag{10.4}$$

$$\frac{\begin{array}{l}[A \to \alpha \bullet B\beta, q_1, q_2] \\ [B \to \gamma\, \bullet, q_2, q_3]\end{array}}{[A \to \alpha B \bullet \beta, q_1, q_3]} \tag{10.5}$$

$$\frac{[q_1, q_2]}{[q_1, q_3]} \left\{\ (q_2, \varepsilon, q_3) \in \Delta \right. \tag{10.6}$$

$$\frac{[q_1, q_2]}{[a, q_1, q_3]} \left\{\ (q_2, a, q_3) \in \Delta,\ a \in V_T \right. \tag{10.7}$$

$$\frac{[S \to \alpha\, \bullet, q_0, q_1]}{[S \to \alpha\, \bullet, q_0, q_2]} \left\{\ (q_1, \varepsilon, q_2) \in \Delta \right. \tag{10.8}$$

Figure 10.1: Earley's algorithm extended to IDL-expressions. Goal items are $[S \to \alpha\, \bullet, q_0, q_f]$, where q_0 and q_f are the initial and final states of $\mu(\pi)$.

Application of Definition 3 results in states that are strings of the form $q_1 q_2 \cdots q_k$, $k \geq 1$, where each q_i is an atomic symbol that originated as a state of an automaton associated with some subexpression of π and where k depends on the structure of π. An intuitive interpretation of states $q_1 q_2 \cdots q_k$ is as follows. In the attempt to generate a string in $L(\pi)$, we evaluate "in parallel" some of the subexpressions of π. Each q_i in $q_1 q_2 \cdots q_k$ refers to a position in these subexpressions, and k is the number of evaluations that we are carrying out in parallel. This will be discussed again in the next section

We can now turn to the definition of our parsing algorithm. A context-free grammar is denoted as $G = (V_N, V_T, P, S)$, where V_N and V_T are finite sets of nonterminals and terminals, respectively, S is the start nonterminal and P is the set of productions. Without loss of generality, we assume S is not recursive. We reformulate Earley's algorithm (Earley 1970) to handle IDL-expressions π. The formulation makes use of relations from an automaton $\mu(\pi)$, but this does not mean that $\mu(\pi)$ needs to be fully constructed before Earley's algorithm is invoked. As already mentioned, we can construct transitions of the automaton on-the-fly when we first visit them, and cache these transitions for possible later use.

The algorithm uses items of the form $[A \rightarrow \alpha \bullet \beta, q_1, q_2]$, where $A \rightarrow \alpha\beta$ is a production of the working grammar and where q_1 and q_2 are states of $\mu(\pi)$. These items have the same meaning as the dotted productions and associated pairs of input positions in the original Earley algorithm, but they now refer to paths in $\mu(\pi)$ starting at state q_1 and ending at state q_2. We also use items of the forms $[q_1, q_2]$ and $[a, q_1, q_2]$, $a \in V_T$, q_1 and q_2 states of $\mu(\pi)$, to by-pass sequences of ε-transitions ending with a transition labelled by a. Figure 10.1 presents the algorithm in the form of a deduction system (see for instance Shieber et al. 1995), where each inference rule specifies a different type of elementary step that the algorithm applies in finding derivations.

Steps (10.1), (10.2) and (10.5) closely resemble the initialization part, the predictor step and the completer step, respectively, of the original Earley algorithm. Steps (10.3), (10.6) and (10.7) can be thought of as computing, through dynamic programming, the standard ε-closure for finite automata (see for instance Hopcroft and Ullman 1979). This only happens at those portions of the automaton where the scanner step needs to be applied. The scanner step itself is then implemented through step (10.4). Finally, step (10.8) attempts to reach the goal item through ε-transitions.

10.4 Complexity and implementation

In this section we investigate the computational complexity of our parsing algorithm and discuss two possible implementations. We start with some notation.

For an IDL-expression π, let $\#_p(\pi)$ be the number of operator occurrences (I, D, L and concatenation) appearing within π. Similarly, let $\#_s(\pi)$ be the number of occurrences of symbols from $V_T \cup \{\mathcal{E}\}$ in π. We define $|\pi| = \#_p(\pi) + \#_s(\pi)$. Note that $|\pi|$ is a reasonable measure of the size of an IDL-expression given as input to our parsing algorithm.

Let Q be the set of states of the automaton $\mu(\pi)$. We have already remarked in Section 10.3 that a state $q \in Q$ can be viewed as a string over some alphabet of atomic symbols: these symbols are states q_0 or q_f introduced by automata associated with subexpressions of π. In what follows we call $A(Q)$ the set of such atomic symbols, and view Q as a finite subset of $A(Q)^*$. It is easy to prove from Definition 3 that $|A(Q)| = 2\,|\pi|$, by induction on $|\pi|$. Let $\mathsf{width}(\pi) = \max_{q \in Q} |q|$. Again, it is not difficult to prove from Definition 3 that $\mathsf{width}(\pi) \leq |\pi|$. As already remarked, $\mathsf{width}(\pi)$ represents the maximum number of subexpressions of π that need to be evaluated "in parallel" when generating a string in $L(\pi)$.

We can now give an upper bound for quantity $|Q|$.

Lemma 1 *Let π be an IDL-expression, Q the set of states of $\mu(\pi)$ and $k = \mathsf{width}(\pi)$. We then have $|Q| \leq \left(\frac{2|\pi|}{k}\right)^k$.*

Proof. Below we use the following inequality. For any real values $x_i > 0$, $1 \leq i \leq h$, $h \geq 2$, we have

$$\Pi_{i=1}^{h} x_i \quad \leq \quad \left(\frac{\Sigma_{i=1}^{h} x_i}{h}\right)^h. \tag{10.9}$$

We prove this in the equivalent form $\Pi_{i=1}^{h} (c + y_i) \leq c^h$, where $c > 0$, $y_i > -c$ and $\Sigma_{i=1}^{h} y_i = 0$. If all the y_i are zero then the inequality is vacuously true; otherwise, we may assume without loss of generality that $y_1 y_2 < 0$. Since $(c+y_1)(c+y_2) < c(c+y_1+y_2)$, the above can be proved if we can prove that $c(c+y_1+y_2)\Pi_{i=3}^{h} (c+y_i) \leq c^h$, which has the same form as our starting inequality but with fewer non-zero y_i's. We can therefore iterate until all y_i's become zero valued.

Recall now that each state $q \in Q$ is a string over $A(Q)$ without duplicate occurrences of its symbols. Below we will prove the following claim.

Claim. We can partition $A(Q)$ into subsets $S[\pi, j]$, $1 \leq j \leq k$, such that for every $q \in Q$ and for every $q_1, q_2 \in A(Q)$ occurring in q, q_1 and q_2 do not belong to the same subset $S[\pi, j]$.

We can then write

$$\begin{array}{rcll} |Q| & \leq & \Pi_{j=1}^{k} |S[\pi, j]| & \text{(by our Claim)} \\ & \leq & \left(\frac{\Sigma_{j=1}^{k} |S[\pi, j]|}{k}\right)^k & \text{(by (10.9))} \\ & = & \left(\frac{2|\pi|}{k}\right)^k & \text{(since } |A(Q)| = 2\,|\pi|\text{).} \end{array}$$

We now turn to the proof of the Claim above. We first define a set $A_0(Q) \subseteq A(Q)$ consisting of only those states that were introduced by symbols or operators that are not within the scope of an occurrence of $\times$. We also define $\mathsf{0\text{-}width}(\pi) = \max_{q \in Q \cap A_0(Q)^*} |q|$.

We prove a slightly stronger claim than the one above: We can partition $A(Q)$ into subsets $S[\pi, j]$, $1 \leq j \leq k = \mathsf{width}(\pi)$, such that for every $q \in Q$ and for every $q_1, q_2 \in A(Q)$ occurring in q, q_1 and q_2 do not belong to the same subset $S[\pi, j]$, and such that all symbols from $A_0(Q)$ are included in some $S[\pi, j]$, $1 \leq j \leq \mathsf{0\text{-}width}(\pi)$; in other words, the subsets $S[\pi, j]$, $\mathsf{0\text{-}width}(\pi) < j \leq \mathsf{width}(\pi)$, contain only symbols from $A(Q) \backslash A_0(Q)$.

We use induction on $\#_p(\pi)$, the number of operator occurrences in π.

Base: $\#_p(\pi) = 0$. Then $\pi = a$, with $a \in V_T \cup \{\mathcal{E}\}$. We have $Q = A(Q) = A_0(Q) = \{q_0, q_f\}$ and $\mathsf{width}(\pi) = \mathsf{0\text{-}width}(\pi) = 1$. We set $S[\pi, 1] = A(Q)$.

Induction: $\#_p(\pi) > 0$.

Case 1: $\pi = \vee(\pi_1, \pi_2, \ldots, \pi_n)$. We have $Q = (\cup_{i=1}^n Q_i) \cup \{q_0, q_f\}$, where each Q_i is the state set of $\mu(\pi_i)$. We also have $\mathsf{width}(\pi) = \max_{i=1}^n \mathsf{width}(\pi_i)$ and $\mathsf{0\text{-}width}(\pi) = \max_{i=1}^n \mathsf{0\text{-}width}(\pi_i)$. If we define $S[\pi_i, j] = \emptyset$ for $j > \mathsf{width}(\pi_i)$, then we can set $S[\pi, 1] = (\cup_{i=1}^n S[\pi_i, 1]) \cup \{q_0, q_f\}$, and $S[\pi, j] = \cup_{i=1}^n S[\pi_i, j]$ for $2 \leq j \leq \mathsf{width}(\pi)$.
Case 2: $\pi = \times(\pi')$ or $\pi = \pi_1 \cdot \pi_2$. The proof is almost identical to that of Case 1, with $n = 1$ or $n = 2$, respectively.
Case 3: $\pi = \|(\pi_1, \pi_2, \ldots, \pi_n)$. We have $Q = (Q_1 Q_2 \ldots Q_n) \cup \{q_0, q_f\}$, Q_i the state set of $\mu(\pi_i)$. We can show that

$$\begin{aligned} \mathsf{0\text{-}width}(\pi) &= \Sigma_{j=1}^n \mathsf{0\text{-}width}(\pi_j) \\ \mathsf{width}(\pi) &= \Sigma_{j=1}^n \mathsf{0\text{-}width}(\pi_j) + \max_{j=1}^n (\mathsf{width}(\pi_j) - \mathsf{0\text{-}width}(\pi_j)). \end{aligned}$$

We now set $S[\pi, 1] = S[\pi_1, 1] \cup \{q_0, q_f\}$, $S[\pi, j] = S[\pi_1, j]$ for $2 \leq j \leq \mathsf{0\text{-}width}(\pi_1)$, $S[\pi, j + \mathsf{0\text{-}width}(\pi_1)] = S[\pi_2, j]$ for $1 \leq j \leq \mathsf{0\text{-}width}(\pi_2)$, etc. If we again define $S[\pi_i, j] = \emptyset$ for $j > \mathsf{width}(\pi_i)$, then we can set $S[\pi, j + \Sigma_{i=1}^n \mathsf{0\text{-}width}(\pi_i)] = \cup_{i=1}^n S[\pi_i, j + \mathsf{0\text{-}width}(\pi_j)]$ for $1 \leq j \leq \max_{j=1}^n (\mathsf{width}(\pi_j) - \mathsf{0\text{-}width}(\pi_j))$. ∎

We now turn to the discussion of the worst case running time for the algorithm in Figure 10.1. To simplify the presentation, let us ignore for the moment any term that solely depends on the input grammar G. To store and retrieve items $[A \rightarrow \alpha \bullet \beta, q_1, q_2]$, $[q_1, q_2]$ and $[a, q_1, q_2]$, for any pair of states q_1 and q_2 in Q, we use a data structure T and access T using q_1 and q_2 as indices. In what follows we make the assumption that each access operation on T can be carried out in an amount of time $O(d(k))$, where $k = \mathsf{width}(\pi)$ and d is some function that depends on the implementation of the algorithm, to be discussed later.

The worst case time complexity of the algorithm is dominated by the steps in Figure 10.1 that involve the maximum number of states from Q, namely steps (10.4), (10.5), (10.6) and (10.7), with three states each. The number of different possible calls to these steps is then proportional to $|Q|^3$. Considering our assumption on T, the total amount of time that is charged to the execution of all these steps is then $O(d(k)\,|Q|^3)$. As in the case of the standard Earley algorithm, when the working grammar G is taken into account we must include a factor of $|G|^2$, which can be further reduced to $|G|$ using techniques discussed by Graham et al. (1980).

We also need to consider the amount of time required by the on-the-fly construction of the underlying automaton $\mu(\pi)$. This might take place at steps (10.6), (10.7) and (10.8), where the transition function needs to be read. Whenever we need to newly visit a transition emanating from some state q_1, we compute the transition and the reached state q_2, and cache these values for possible later use. (Note however that q_2 may already exist, in which case we only cache the new transition.) All this can be easily carried out in time $O(\max\{|q_1|, |q_2|\})$. Then in the worst case the total time required by the on-the-fly construction of $\mu(\pi)$ is $O(k\,|\Delta|)$, Δ the transition relation of the automaton.

From the above observations we can conclude that, in the worst case, the algorithm in Figure 10.1 takes an amount of time $O(|G|\,d(k)\,|Q|^3 + k\,|\Delta|)$. From the definition of $\mu(\pi)$ we can easily see that the number of arcs between any two states of the automaton is always bounded by one, and therefore $|\Delta| \leq |Q|^2$. As already discussed, we also have $1 \leq k \leq |\pi|$. We can thus write $k \leq 2^k \leq (\frac{2|\pi|}{k})^k$. Putting together these two upper bounds and using Lemma 1the upper bound on the running time of our parsing algorithm becomes $O(|G|\, d(k) (\frac{2|\pi|}{k})^{3k})$.

We now more closely consider the choice of the data structure T and the issue of its implementation. We discuss two possible solutions. Our first solution can be used when $|Q|$ is small enough so that we can store $|Q|^2$ pointers in the computer's RAM. In this case we can implement T as a square array of pointers. Each state in Q is then uniquely encoded by some positive integer, and these integers are used to access the array. This solution in practice comes down to the standard implementation of the Earley algorithm through a parse table, as in Graham et al. (1980). We then have $d(k) = O(1)$ and our algorithm has time complexity $O(|G|\, (\frac{2|\pi|}{k})^{3k})$.

As a second solution, when $|Q|$ is quite large, we can implement T as a trie data structure (Gusfield 1997). In this case each state in Q is treated as a string over $A(Q)$, and we look up string $q_1\#q_2$ in T in order to retrieve all items involving states q_1, q_2 that have been induced so far (# is a symbol not in $A(Q)$). We then obtain $d(k) = O(k)$ and our algorithm has time complexity $O(|G|\, k (\frac{2|\pi|}{k})^{3k})$.

The first solution above is faster than the second one by a factor of k, but in practice k can be quite small, say four or five. However, the first solution has the obvious disadvantage of storage requirement, since not all state pairs might correspond to some grammar constituent, and the array T can be very sparse in practice.

Observe that when we ignore the factor of k in the above upper bound, we get the same time complexity as in the implementation of the Earley algorithm for strings presented by Graham et al. (1980), but now function $(\frac{2|\pi|}{k})^k$ takes the place of the input sentence length. In the worst case we can have $k = |\pi|$, which maximizes the previous function to $2^{|\pi|}$. This corresponds to cases where no precedence constraints at all are given among the input words, and we would observe an exponential behaviour. This is of no surprise, since in this case the problem becomes the one of bag parsing, which is known to be NP-complete for the class of CFGs (Brew 1992). The running time becomes much closer to polynomial behaviour as soon as only few subexpressions at a time from π can be processed "in parallel", with k far smaller than π.

Some additional remarks are in order here concerning the on-the-fly construction of the finite automata $\mu(\pi)$. For space reasons, this paper only includes a recursive definition of automata $\mu(\pi)$ (Definition 3). However, our definition does not directly show how to incrementally construct $\mu(\pi)$, starting from its initial state and building only those parts of the automaton that correspond to prefixes of strings generated by the context-free grammar. In a longer version of this paper we provide a method for effectively computing transitions $(q_1, X, q_2) \in \Delta$, Δ the transition relation of $\mu(\pi)$, starting from q_1 and X and in time $O(\mathsf{width}(\pi))$, as required by the previous analysis. This method involves the new notion of IDL-graph.

An **IDL-graph** is a planar graph encoding the IDL-expression π, whose vertices are the elements of $A(Q)$. Each IDL-graph has a starting vertex and an ending vertex, and has the important property that each state of the automaton $\mu(\pi)$ corresponds to a “cut” through the graph. Roughly speaking, such a cut is obtained by drawing a straight line through the planar graph that separates the starting vertex from the ending vertex. For a graph corresponding to a disjunction, we nondeterministically choose a subgraph corresponding to an argument of the disjunction and recursively obtain the cut from the part of the line through that subgraph. For a graph corresponding to an interleave, the cuts obtained recursively for the subgraphs are concatenated. Locking is represented through ranking of vertices.

On the basis of an IDL-graph, it is easy to specify the desired method for the incremental computation of the transition relation Δ. More specifically, given q_1 and X we can compute all transitions $(q_1, X, q_2) \in \Delta$ by investigating the outgoing edges of the IDL-graph at the vertices in the cut corresponding to state q_1. The details are not pursued here for reasons of space.

10.5 Final remarks

IDL-expressions are similar in spirit to formalisms developed in the programming language literature for the representation of the semantics of concurrent programs. More specifically, partially ordered multisets, or pomsets, have been proposed by Gischer (1988) to represent choice and parallelism among processes. However, the basic idea of a lock operator is absent from pomsets. As pointed out by a reviewer, there is also a superficial similarity between IDL-expressions and so called immediate dominance and linear precedence context-free grammars, or ID/LP CFGs (Gazdar et al. 1985). ID/LP CFGs relax the usual linear order of symbols in the right-hand sides of productions, to capture structural syntactic variations. From a different perspective, IDL-expressions use partial order relations to express uncertainty on word/phrase order within a string realization, due to the lack of knowledge in the application at hand.

As a final remark we point out that, apart from the on-the-fly construction of the automata $\mu(\pi)$, the algorithm in Figure 10.1 should be considered as an application of the well-known construction from Bar-Hillel et al. (1964) for the intersection of a CFG and a finite automaton.

Bibliography

Bangalore, S. and O. Rambow (2000). Exploiting a probabilistic hierarchical model for generation. In *The 18th International Conference on Computational Linguistics*, volume 1, pp. 42–48. Saarbrücken, Germany.

Bar-Hillel, Y., M. Perles, and E. Shamir (1964). On formal properties of simple phrase structure grammars. In Y. Bar-Hillel, ed., *Language and Informa-*

tion: Selected Essays on their Theory and Application, chapter 9, pp. 116–150. Addison-Wesley.

Brew, C. (1992). Letting the cat out of the bag: generation for Shake-and-Bake MT. In *Proc. of the fifteenth International Conference on Computational Linguistics*, volume 2, pp. 610–616. Nantes.

Earley, J. (1970). An efficient context-free parsing algorithm. *Communications of the ACM*, **13**(2):94–102.

Gazdar, G., E. Klein, G. Pullum, and I. Sag (1985). *Generalized Phrase Structure Grammar*. Harvard University Press, Cambridge, MA.

Gischer, J. (1988). The equational theory of pomsets. *Theoretical Computer Science*, **61**:199–224.

Graham, S., M. Harrison, and W. Ruzzo (1980). An improved context-free recognizer. *ACM Transactions on Programming Languages and Systems*, **2**(3):415–462.

Gusfield, D. (1997). *Algorithms on Strings, Trees and Sequences*. Cambridge University Press, Cambridge, UK.

Hopcroft, J. and J. Ullman (1979). *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley.

Knight, K. and V. Hatzivassiloglou (1995). Two-level, many-paths generation. In *33rd Annual Meeting of the Association for Computational Linguistics, Proceedings of the Conference*, pp. 252–260. Cambridge, Massachusetts, USA.

Langkilde, I. (2000). Forest-based statistical sentence generation. In *6th Applied Natural Language Processing Conference and 1st Meeting of the North American Chapter of the Association for Computational Linguistics*, Section 2, pp. 170–177. Seattle, Washington, USA.

Langkilde, I. and K. Knight (1998). Generation that exploits corpus-based statistical knowledge. In *36th Annual Meeting of the Association for Computational Linguistics and 17th International Conference on Computational Linguistics*, volume 1, pp. 704–710. Montreal, Quebec, Canada.

Shieber, S., Y. Schabes, and F. Pereira (1995). Principles and implementation of deductive parsing. *Journal of Logic Programming*, **24**:3–36.

Whitelock, P. (1992). Shake-and-Bake translation. In *Proc. of the fifteenth International Conference on Computational Linguistics*, volume 2, pp. 784–790. Nantes.