A Parsing Algorithm for Unification Grammar

Andrew Haas

Department of Computer Science
State University of New York at Albany
Albany, New York 12222

We describe a table-driven parser for unification grammar that combines bottom-up construction of phrases with top-down filtering. This algorithm works on a class of grammars called depth-bounded grammars, and it is guaranteed to halt for any input string. Unlike many unification parsers, our algorithm works directly on a unification grammar—it does not require that we divide the grammar into a context-free "backbone" and a set of feature agreement constraints. We give a detailed proof of correctness. For the case of a pure bottom-up parser, our proof does not rely on the details of unification —it works for any pattern-matching technique that satisfies certain simple conditions.

1 Introduction

Unrestricted unification grammars have the formal power of a Turing machine. Thus there is no algorithm that finds all parses of a given sentence in any unification grammar and always halts. Some unification grammar systems just live with this problem. Any general parsing method for definite clause grammar will enter an infinite loop in some cases, and it is the task of the grammar writer to avoid this. Generalized phrase structure grammar avoids the problem because it has only the formal power of context-free grammar (Gazdar et al. 1985), but according to Shieber (1985a) this is not adequate for describing human language.

Lexical functional grammar employs a better solution. A lexical functional grammar must include a finitely ambiguous context-free grammar, which we will call the context-free backbone (Barton 1987). A parser for lexical functional grammar first builds the finite set of context-free parses of the input and then eliminates those that don't meet the other requirements of the grammar. This method guarantees that the parser will halt.

This solution may be adequate for lexical functional grammars, but for other unification grammars finding a finitely ambiguous context-free backbone is a problem. In a definite clause grammar, an obvious way to build a context-free backbone is to keep only the topmost function letters in each rule. Thus the rule

$s \rightarrow np(P,N)\ vp(P,N)$

becomes

$s \rightarrow np\ vp$

(In this example we use the notation of Pereira and Warren 1980, except that we do not put square brackets around terminals, because this conflicts with standard notation for context-free grammars.) Suppose we use a simple X-bar theory. Let major-category (Type, Bar-level) denote a phrase in a major category. A noun phrase may consist of a single noun, for instance, *John*. This suggests a rule like this:

major-category (n,2) → major-category (n,1)

In the context-free backbone this becomes

major-category → major-category

so the context-free backbone is infinitely ambiguous. One could devise more elaborate examples, but this one suffices to make the point: not every natural unification grammar has an obvious context-free backbone. Therefore it is useful to have a parser that does not require us to find a context-free backbone, but works directly on a unification grammar (Shieber 1985b).

We propose to guarantee that the parsing problem is solvable by restricting ourselves to depth-bounded grammars. A unification grammar is depth-bounded if for every $L > 0$ there is a $D > 0$ such that every parse tree for a sentential form of L symbols has depth less than D. In other words, the depth of a tree is bounded by the length of the string it derives. A context-free grammar is depth-bounded if and only if every string of symbols is finitely ambiguous. We will generalize the notion of finite ambiguity to unification grammars and show that for unification grammars, depth-boundedness is a stronger property than finite ambiguity.

0362-613X/89/010219–232$03.00

Depth-bounded unification grammars have more formal power than context-free grammars. As an example we give a depth-bounded grammar for the language xx, which is not context-free. Suppose the terminal symbols are a through z. We introduce function letters a' through z' to represent the terminals. The rules of the grammar are as follows, with e denoting the empty string.

s → x(L)x(L)
x(cons(A,L)) → pre-terminal(A) x(L)
x(nil) → e
pre-terminal(a') → a
...
pre-terminal(z') → z

The reasoning behind the grammar should be clear—x(cons(a',cons(b',nil))) derives ab, and the first rule guarantees that every sentence has the form xx. The grammar is depth-bounded because the depth of a tree is a linear function of the length of the string it derives. A similar grammar can derive the crossed serial dependencies of Swiss German, which according to Shieber (1985a) no context-free grammar can derive. It is clear where the extra formal power comes from: a context-free grammar has a finite set of nonterminals, but a unification grammar can build arbitrarily large nonterminal symbols.

It remains to show that there is a parsing algorithm for depth-bounded unification grammars. We have developed such an algorithm, based on the context-free parser of Graham et al. 1980, which is a table-driven parser. If we generalize the table-building algorithm to a unification grammar in an obvious way, we get an algorithm that is guaranteed to halt for all depth-bounded grammars (not for all unification grammars). Given that the tables can be built, it is easy to show that the parser halts on every input. This is not a special property of our parser—a straightforward bottom-up parser will also halt on all depth-bounded grammars, because it builds partial parse trees in order of their depth. Our contribution is to show that a simple algorithm will verify depth-boundedness when in fact it holds. If the grammar is not depth-bounded, the table-building algorithm will enter an infinite loop, and it is up to the grammar writer to fix this. In practice we have not found this troublesome, but it is still an unpleasant property of our method. Section 7 will describe a possible solution for this problem.

Sections 2 and 3 of this paper define the basic concepts of our formalism. Section 4 proves the soundness and completeness of our simplest parser, which is purely bottom-up and excludes rules with empty right-hand sides. Section 5 admits rules with empty right sides, and section 6 adds top-down filtering. Section 7 discusses the implementation and possible extensions.

2 Basic Concepts

The following definitions are from Gallier 1986. Let S be a finite, nonempty set of sorts. An **S-ranked alphabet** is a pair (Σ,r) consisting of a set Σ together with a function $r : \Sigma \rightarrow S^* \times S$ assigning a **rank** (u,s) to each symbol f in Σ. The string u in S^* is the **arity** of f and s is the **type** of f.

The S-ranked alphabets used in this paper have the following property. For every sort $s \in S$ there is a countably infinite set V_s of symbols of sort s called variables. The rank of each variable in V_s is (e,s), where e is the empty string. Variables are written as strings beginning with capitals—for instance X, Y, Z. Symbols that are not variables are called function letters, and function letters whose arity is e are called constants. There can be only a finite number of function letters in any sort.

The set of terms is defined recursively as follows. For every symbol f of rank $(u_1 \ldots u_n, s)$ and any terms $t_1 \ldots t_n$, with each t_i of sort u_i, $f(t_1, \ldots t_n)$ is a term of sort s. Since every sort in S includes variables, whose arity is e, it is clear that there are terms of every sort.

A term is called a ground term if it contains no variables. We make one further requirement on our ranked alphabets: that every sort contains a ground term. This can be guaranteed by just requiring at least one constant of every sort. It is not clear, however, that this solution is linguistically acceptable—we do not wish to include constants without linguistic significance just to make sure that every sort includes a ground term. Therefore, we give a simple algorithm for checking that every sort in S includes a ground term.

Let T_1 be the set of sorts in S that include a constant. Let T_{i+1} be the union of T_i and the set of all s in S such that for some function letter f of sort s, the arity of f is $u_1 \ldots u_n$ and the sorts $u_1, \ldots, u_n$ are in T_i. Every sort in T_1 includes a ground term, and if every sort in T_i includes a ground term then every sort in T_{i+1} includes a ground term. Then for all n, every sort in T_n includes a ground term. The algorithm will compute T_n for successive values of n until it finds an N such that $T_N = T_{N+1}$ (this N must exist, because S is finite). If $T_N = S$, then every sort in S includes a ground term, otherwise not.

As an illustration, let S = {phrase, person, number}. Let the function letters of Σ be {np, vp, s, 1st, 2nd, 3rd, singular, plural}. Let ranks be assigned to the function letters as follows, omitting the variables.

r(np) = ([person, number], phrase)
r(vp) = ([person, number], phrase)
r(s) = (e, phrase)
r(1st) = (e, number)
r(2nd) = (e, number)
r(3rd) = (e, number)
r [(singular) = (e, person)
r(plural) = (e,person)

We have used the notation $[a,b,c]$ for the string of a, b, and c. Typical terms of this ranked alphabet are np(1st,

singular) and vp(2nd, plural). The reader can verify, using the above algorithm, that every sort includes a ground term. In this case, T_1 = {person, number}, T_2 = {person, number, phrase}, and $T_3 = T_2$.

To summarize: we define ranked alphabets in a standard way, adding the requirements that every sort includes a countable infinity of variables, a finite number of function letters, and at least one ground term. We then define the set of terms in a standard way. All unification in this paper is unification of terms, as in Robinson 1965—not graphs or other structures, as in much recent work (Shieber 1985b).

A **unification grammar** is a five-tuple $G = (S, (\Sigma,r)\ T, P, Z)$ where S is a set of sorts, (Σ,r) an S-ranked alphabet, T a finite set of terminal symbols, and Z a function letter of arity e in (Σ,r). Z is called the start symbol of the grammar (the standard notation is S not Z, but by bad luck that conflicts with standard notation for the set of sorts). P is a finite set of **rules**; each rule has the form $(A \rightarrow \alpha)$, where A is a term of the ranked alphabet and α is a sequence of terms of the ranked alphabet and symbols from T.

We define **substitution** and **substitution instances** of terms in the standard way (Gallier 1986). We also define instances of rules: if s is a substitution and $(A \rightarrow B_1 \ldots B_n)$ is a rule, then $(s(A) \rightarrow s(B_1)\ldots s(B_n))$ is an instance of the rule $(A \rightarrow B_1 \ldots B_n)$. A **ground instance** of a term or rule is an instance that contains no variables.

Here is an example, using the set of sorts S from the previous example. Let the variables of sort *person* be P_1, P_2,... and the variables of sort *number* be N_1, N_2... etc. Then the rule $(start \rightarrow np(P_1, N_1)\ vp(P_1, N_1)$ has six ground instances, since there are three possible substitutions for the variable P_1 and two possible substitutions for N_1.

We come now to the key definition of this paper. Let $G = (S, (\Sigma,r)\ T, P, Z)$ be a unification grammar. The ground grammar for G is the four-tuple (N, T, P', Z), where N is the set of all ground terms of (Σ,r), T is the set of terminals of G, P' is the set of all ground instances of rules in P, and Z is the start symbol of G. If N and P' are finite, the ground grammar is a context-free grammar. If N or P' is infinite, the ground grammar is not a context-free grammar, and it may generate a language that is not context-free. Nonetheless we can define derivation trees just as in a cfg. Following Hopcroft and Ullman (1969), we allow derivation trees with nonterminals at their leaves. Thus a derivation tree may represent a partial derivation. We differ from Hopcroft and Ullman by allowing nonterminals other than the start symbol to label the root of the tree. A derivation tree is an A-tree if the non-terminal A labels its root. The yield of a derivation tree is the string formed by reading the symbols at its leaves from left to right. As in a cfg, $A \overset{*}{\Rightarrow} \alpha$ iff there is an A-tree with yield α. The language generated by a ground grammar is the set of terminal strings derived from the start symbol. The language generated by a unification grammar is the language generated by its ground grammar.

The central idea of this approach is to regard a unification grammar as an abbreviation for its ground grammar. Ground grammars are not always cfgs, but they share many properties of cfgs. Therefore if we regard unification grammars as abbreviations for ground grammars, our understanding of cfgs will help us to understand unification grammars. This is of course inspired by Robinson's work on resolution, in which he showed how to "lift" a proof procedure for propositional logic up to a proof procedure for general first-order logic (Robinson 1965).

The case of a finite ground grammar is important, since it is adequate for describing many syntactic phenomena. A simple condition will guarantee that the ground grammar is finite. Suppose s_1 and s_2 are sorts, and there is a function letter of sort s_1 that has an argument of sort s_2. Then we say that $s_1 > s_2$. Let $>^*$ be the transitive closure of this relation. If $>^*$ is irreflexive, and D is the number of sorts, every term of the ground grammar has depth $\leq D$. To see this, think of a ground term as a labeled tree. A path from the root to a leaf generates a sequence of sorts: the sorts of the variables and functions letters encountered on that path. It is a strictly decreasing sequence according to $>^*$. Therefore, no sort occurs twice; therefore, the length of the sequence is at most D. Since there are only a finite number of function letters in the ranked alphabet, each taking a fixed number of arguments, the number of possible ground terms of depth D is finite. Then the ground grammar is finite.

A ground grammar G' is depth-bounded if for every integer n there exists an integer d such that every derivation tree in G' with a yield of length n has a depth less than d. In other words, a depth-bounded grammar cannot build an unbounded amount of tree structure from a bounded number of symbols. Remember that these symbols may be either terminals or nonterminals, because we allow nonterminals at the leaves of a derivation tree. A unification grammar G is depth-bounded if its ground grammar is depth-bounded.

We say that a unification grammar is finitely ambiguous if its ground grammar is finitely ambiguous. We can now prove the result claimed above: that a unification grammar can be finitely ambiguous but not depth-bounded. In fact, the following grammar is completely unambiguous but still not depth-bounded. It has just one terminal symbol, b, and its start symbol is *start*.

$start \rightarrow p(0)$
$p(N) \rightarrow p(\text{succ}(N))$
$p(N) \rightarrow q(N)$
$q\ (\text{succ}(N)) \rightarrow b\ q(N)$
$q(0) \rightarrow e$

The function letter "succ" represents the successor function on the integers, and the terms 0, succ(0), succ(succ(0))... represent the integers 0, 1, 2... etc. For convenience, we identify these terms with the integers

they represent. A string of N occurrences of b has just one parse tree. In this tree the start symbol derives $p(0)$, which derives $p(N)$ by N applications of the second rule. $p(N)$ derives $q(N)$, which derives N occurrences of b by N applications of the fourth rule and one application of the last rule. The reader can verify that this derivation is the only possible one, so the grammar is unambiguous. Yet the start symbol derives $p(N)$ by a tree of depth N, for every N. Thus trees whose frontier has only one symbol can still be arbitrarily deep. Then the grammar cannot be depth-bounded.

We have defined the semantics of our grammar formalism without mentioning unification. This is deliberate; for us unification is a computational tool, not a part of the formalism. It might be better to call the formalism "substitution grammar," but the other name is already established.

Notation: The letters A, B, and C denote symbols of a ground grammar, including terminals and nonterminals. Lowercase Greek letters denote strings of symbols. $\alpha\ [i\ k]$ is the substring of α from space i to space k, where the space before the first symbol is space zero. e is always the empty string. We write $x \cup y$ or $\cup(x,y)$ for the union of sets x and y, and also $(\cup\ i<j<k\ f(j))$ for the union of the sets $f(j)$ for all j such that $i < j < k$.

If α is the yield of a tree t, then to every occurrence of a symbol A in α there corresponds a leaf of t labeled with A. To every node in t there corresponds an occurrence of a substring in α—the substring dominated by that node. Here is a lemma about trees and their yields that will be useful when we consider top-down filtering.

Lemma 2.1. Suppose t is a tree with yield $\alpha\beta\alpha'$ and n is the node of t corresponding to the occurrence of β after α in $\alpha\beta\alpha'$. Let A be the label of n. If t' is the tree formed by deleting all descendants of n from t, the yield of t' is $\alpha A\alpha'$.

Proof: This is intuitively clear, but the careful reader may prove it by induction on the depth of t.

3 Operations on Sets of Rules and Terms

The parser must find the set of ground terms that derive the input string and check whether the start symbol is one of them. We have taken the rules of a unification grammar as an abbreviation for the set of all their ground instances. In the same way, the parser will use sets of terms and rules containing variables as a representation for sets of ground terms and ground rules. In this section we show how various functions needed for parsing can be computed using this representation.

A grammatical expression, or *g-expression*, is either a term of L, the special symbol *nil*, or a pair of g-expressions. The letters u, v, w, x, y, and z denote g-expressions, and X, Y, and Z denote sets of g-expressions. We use the usual LISP functions and predicates to describe g-expressions. $[x\ y]$ is another notation for cons (x,y). For any substitution s, s (cons (x,y)) = cons (s(x),s(y)) and s(*Nil*) = *Nil*. A selector is a function from g-expressions to g-expressions formed by composition from the functions *car*, *cdr*, and *identity*. Thus a selector picks out a subexpression from a g-expression. A constructor is a function that maps two g-expressions to a g-expression, formed by composition from the functions *cons*, *car*, *cdr*, *nil*, $(\lambda\ x\ y.\ x)$, and $(\lambda\ x\ y.\ y)$. A constructor builds a new g-expression from parts of two given g-expressions. A g-predicate is a function from g-expressions to Booleans formed by composition from the basic functions *car*, *cdr*, $(\lambda\ x.\ x)$, *consP*, and *null*.

Let ground(X) be the set of ground instances of g-expressions in X. If f is a selector function, let $f(X)$ be the set of all $f(x)$ such that $x \in X$. If p is a g-predicate, let separate (p,X) be the set of all $x \in X$ such that $p(x)$. The following lemmas are easily established from the definition of s(x) for a g-expression x.

Lemma 2.2. If f is a selector function, f(ground(X)) = ground ($f(X)$).

Lemma 2.3. If p is a g-predicate, separate (p,ground (X)) = ground(separate (p,X)).

Lemma 2.4. Ground ($X \cup Y$) = ground (X) $\cup$ ground (Y).

Lemma 2.5. If x is a ground term, $x \in$ ground(X) iff x is an instance of some $y \in X$.

Lemma 2.6. Ground (X) is empty iff X is empty.

Proof. A nonempty set of terms must have a nonempty set of ground instances, because every variable belongs to a sort and every sort includes at least one ground term.

These lemmas tell us that if we use sets X and Y of terms to represent the sets ground(X) and ground(Y) of ground terms, we can easily construct representations for f(ground(X)), separate(p,ground (X)), and ground (X) $\cup$ ground (Y). Also we can decide whether a given ground term is contained in ground(X) and whether ground(X) is empty. All these operations will be needed in the parser.

The parser requires one more type of operation, defined as follows.

Definition. Let f_1 and f_2 be selectors and g a constructor, and suppose $g(x,y)$ is well defined whenever $f_1(x)$ and $f_2(y)$ are well defined. The **symbolic product** defined by f_1, f_2, and g is the function

$(\lambda\ X\ Y.\ \{\ g(x,y)\ |\ x \in X \wedge y \in Y \wedge f_1(x) = f_2(y)\ \})$

where X and Y range over sets of ground g-expressions. Note that $f_1(x) = f_2(y)$ is considered false if either side of the equation is undefined.

The symbolic product matches every x in X against every y in Y. If $f_1(x)$ equals $f_2(y)$, it builds a new structure from x and y using the function g. As an example, suppose X and Y are sets of pairs of ground terms, and we need to find all pairs $[A\ C]$ such that for some B, $[A\ B]$ is in X and $[B\ C]$ is in Y. We can do this by finding the symbolic product with $f_1 = cdr$, $f_2 = car$, and $g = (\lambda\ x\ y.\ cons(car(x), cdr(y)))$. To see that this is correct, notice that if $[A\ B]$ is in X and $[B\ C]$ is in Y, then

$f_1([A\ B]) = f_2([B\ C])$, so the pair $g([A\ B],[B\ C]) = [A\ C]$ must be in the answer set.

A second example: we can find the intersection of two sets of terms by using a symbolic product with $f_1 = (\lambda\ x\ .\ x)$, $f_2 = (\lambda\ x\ .\ x)$, and $g = (\lambda\ x\ y.\ x)$.

If X is a set of g-expressions and n an integer, rename(X,n) is an alphabetic variant of X. For all X, Y, m, and n, if $m \neq n$ then rename(X,n) and rename(Y,m) have no variables in common. The following theorem tells us that if we use sets of terms X and Y to represent the sets ground(X) and ground(Y) of ground terms, we can use unification to compute any symbolic product of ground(X) and ground(Y). We assume the basic facts about unification as in Robinson (1965).

Theorem 2.1. If h is the symbolic product defined by f_1, f_2 and g, and X and Y are sets of g-expressions, then

h (ground(X),ground(Y)) =
ground({s(g(u,v)) | $u \in$ rename(X,1) $\wedge$ v $\in$ rename(Y,2)
$\wedge$ s is the m.g.u. of f_1(u) and f_2(v)})

Proof. The first step is to show that if Z and W share no variables

(1) {g(z,w) | $z \in$ ground(Z) $\wedge$ w $\in$ ground(W) $\wedge$ f_1(z) = f_2 (w)} = ground({s(g(u,v)) | u $\in Z \wedge$ v $\in$ W $\wedge$ s is the m.g.u. of f_1(u) and f_2(v) })

Consider any element of the right side of equation (1). It must be a ground instance of $s(g(u,v))$, where $u \in Z$, $v \in W$, and s is the m.g.u. of $f_1(u)$ and $f_2(v)$. Any ground instance of $s(g(u,v))$ can be written as $s'(s(g(u,v)))$, where s' is chosen so that $s'(s(u))$ and $s'(s(v))$ are ground terms. Then $s'(s(g(u,v))) = g(s'(s(u)),s'(s(v)))$ and $f_1(s'(s(u))) = s'(s(f_1(u))) = s'(s(f_2\ (v))) = f_2(s'(s(v)))$. Therefore $s'(s(g(u,v)))$ belongs to the set on the left side of equation (1).

Next consider any element of the left side of (1). It must have the form $g(z,w)$, where $z \in$ ground(Z), $w \in$ ground(W), and $f_1\ (z) = f_2\ (w)$. Then for some $u \in Z$ and $v \in W$, z is a ground instance of u and w is a ground instance of v. Since u and v share no variables, there is a substitution s' such that $s'(u) = z$ and $s'(v) = w$. Then $s'(f_1\ (u)) = f_1\ (s'(u)) = f_2\ (s'(v)) = s'(f_2\ (v))$, so there exists a most general unifier s for $f_1\ (u)$ and $f_2\ (v)$, and s' is the composition of s and some substitution s''. Then $g(z,w) = g(s(s(u))\ s(s(v))) = s(s(g(u,v)))$. $g(z,w)$ is a ground term because z and w are ground terms, so $g(z,w)$ is a ground instance of $s(g(u,v))$ and therefore belongs to the set on the right side of equation (1).

We have proved that if Z and W share no variables,

(2) h(ground(Z),ground(W)) = ground({s(g(u,v)) | u $\in Z \wedge$ v $\in$ W $\wedge$ s is the m.g.u. of f_1(u) and f_2(v)})

For any X and Y, rename(X,1) and rename(Y,2) share no variables. Then we can let Z = rename(X,1) and W = rename(Y,2) in formula (2). Since h(ground(X), ground(Y)) = h(ground(rename(X,1)), ground(rename (Y,2))), the theorem follows by transitivity of equality. This completes the proof.□

As an example, suppose $X = \{[a(F)\ b(F)]\}$ and $Y = \{[b(G)\ c(G)]\}$. Suppose the variables F and G belong to a sort s that includes just two ground terms, m and n. We wish to compute the symbolic product of ground(X) and ground(Y), using $f_1 = cdr$, $f_2 = car$, and $g = (\lambda\ x\ y.\ cons(car(x),\ cdr(y)))$ (as in our previous example). ground(X) equals $\{[a(m)\ b(m)],[a(n)\ b(n)]\}$ and ground(Y) equals $\{[b(m)\ c(m)],[b(n)\ c(n)]\}$, so the symbolic product is $\{[a(m)\ c(m)],[a(n)\ c(n)]\}$. We will verify that the unification method gets the same result. Since X and Y share no variables, we can skip the renaming step. Let $x = [a(F)\ b(F)]$ and $y = [b(G)\ c(G)]$. Then $f_1\ (x) = b(F)$, $f_2\ (y) = b(G)$, and the most general unifier is the substitution s that replaces F with G. Then $g(x,y) = [a(F)\ c(G)]$ and $s(g(x,y)) = [A(G)\ C(G)]$. The set of ground instances of this g-expression is $\{[A(m)\ C(m)], [A(n)C(n)]\}$, as desired.

Definition. Let f be a function from sets of g-expressions to sets of g-expressions, and suppose that when $X \subseteq X'$ and $Y \subseteq Y'$, $f(X,Y) \subseteq f(X',Y')$. Then f is **monotonic**.

All symbolic products are monotonic functions, as the reader can easily show from the definition of symbolic products. Indeed, every function in the parser that returns a set of g-expressions is monotonic.

4 The Parser without Empty Symbols

Our first parser does not allow rules with empty right sides, since these create complications that obscure the main ideas. Therefore, throughout this section let G be a ground grammar in which no rule has an empty side. When we say that α derives β we mean that α derives β in G. Thus $\alpha \overset{*}{\Rightarrow} e$ iff $\alpha = e$.

A dotted rule in G is a rule of G with the right side divided into two parts by a dot. The symbols to the left of the dot are said to be before the dot, those to the right are after the dot. DR is the set of all dotted rules in G. A dotted rule $(A \rightarrow \alpha.\beta)$ derives a string if α derives that string. To compute symbolic products on sets of rules or dotted rules, we must represent them as g-expressions. We represent the rule $(A \rightarrow B\ C)$ as the list $(A\ B\ C)$, and the dotted rule $(A \rightarrow B.C)$ as the pair $[(A\ B\ C)\ (C)\ ]$.

We write $A \Rightarrow B$ if A derives B by a tree with more than one node. The parser relies on a chain table—a table of all pairs $[A\ B]$ such that $A \Rightarrow$ B. Let C_d be the set of all $[A\ B]$ such that $A \Rightarrow B$ by a derivation tree of depth d. Clearly C_1 is the set of all $[A\ B]$ such that $(A \rightarrow B)$ is a rule of G. If S_1 and S_2 are sets of pairs of terms, define

link(S_1,S_2) = {[A C] | ($\exists$ B. [A B] $\in S_1 \wedge$ [B C] $\in S_2$)}

The function *link* is equal to the symbolic product defined by $f_1 = cdr$, $f_2 = car$, and $g = (\lambda\ x\ y\ .\ cons(car(x),\ cdr(y)))$. Therefore we can compute link $(S_1,\ S_2)$ by applying Theorem 2.1. Clearly $C_{d\ +\ 1}$ = link(C_d,C_1). Since the grammar is depth-bounded, there exists a number D such that every derivation tree whose yield contains exactly one symbol has depth less than D. Then C_D is empty. The algorithm for building the chain table is this: compute C_n for increasing values of

n until C_n is empty. Then the union of all the C_n's is the chain table.

We give an example from a finite ground grammar. Suppose the rules are

$(a \rightarrow b)$
$(b \rightarrow c)$
$(c \rightarrow d)$
$(d \rightarrow k\ f)$
$(k \rightarrow g)$
$(f \rightarrow h)$

The terminal symbols are g and h . Then $C_1 = \{[a\ b], [b\ c], [c\ d]\}$, $C_2 = \{[a\ c],[b\ d]\}$, and $C_3 = \{[a\ d]\}$. C_4 is empty.

Definitions. ChainTable is the set of all $[A\ B]$ such that $A \stackrel{+}{\Rightarrow} B$. If S is a set of dotted pairs of symbols and S' a set of symbols, ChainUp(S,S') is the set of symbols A such that $[A\ B] \in S$ for some $B \in S'$. "ChainUp" is clearly a symbolic product. If S is a set of symbols, close(S) is the union of S and ChainUp(ChainTable,S). By the definition of ChainTable, close(S) is the set of symbols that derive a symbol of S.

In the example grammar, ChainTable is the union of C_1, C_2, and C_3—that is, the set $\{[\ a\ b],[b\ c],[c\ d],[a\ c], [b\ d],[a\ d]\}$. ChainUp($\{\ a\}$) = {}, but ChainUp($\{\ d\}$) = $\{\ a,b,c\}$. close($\{\ a\}$) = $\{\ a\}$, while close($\{\ d\}$) = $\{\ a,b,c,d\}$.

Let α be an input string of length $L > 0$. For each $\alpha[i\ k]$ the parser will construct the set of dotted rules that derive $\alpha[i\ k]$. The start symbol appears on the left side of one of these rules iff $\alpha[i\ k]$ is a sentence of G. By lemma 2.5 this can be tested, so we have a recognizer for the language generated by G. With a small modification the algorithm can find the set of derivation trees of α. We omit details and speak of the algorithm as a *parser* when strictly speaking it is a recognizer only.

The dotted rules that derive $\alpha[i\ k]$ can be partitioned into two sets: rules with many symbols before the dot and rules with exactly one. For each $\alpha[i\ k]$, the algorithm will carry out three steps. First it collects all dotted rules that derive $\alpha[i\ k]$ and have many symbols before the dot. From this set it constructs the set of all symbols that derive $\alpha[i\ k]$, and from these symbols it constructs the set of all dotted rules that derive $\alpha[i\ k]$ with one symbol before the dot. The union of the two sets of dotted rules is the set of all dotted rules that derive $\alpha[i\ k]$. Note that a dotted rule derives $\alpha[i\ k]$ with more than one symbol before the dot iff it can be written in the form $(A \rightarrow \beta B.\beta')$ where $\beta \stackrel{*}{\Rightarrow} \alpha[i\ j]$, $B \stackrel{*}{\Rightarrow} \alpha[j\ k]$, and $i < j < k$ (this follows because a nonempty string β can never derive the empty string in G).

If $(A \rightarrow B\ .\ C)$ derives $\alpha[i\ j]$ and B derives $\alpha[j\ k]$, then $(A \rightarrow B\ C\ .)$ derives $\alpha[i\ k]$. This observation motivates the following.

Definition. If S is a set of dotted rules and S' a set of symbols, AdvanceDot(S,S') is the set of rules $(A \rightarrow \alpha B.\beta)$ such that $(A \rightarrow \alpha.B\beta) \in S$ and $B \in S'$. Clearly *AdvanceDot* is a symbolic product.

For example, AdvanceDot($\{(\ d \rightarrow k\ .\ f)\},\{\ a,f\}$) equals $\{(\ d \rightarrow k\ f\ .)\}$.

Suppose that for each proper substring of $\alpha[i\ k]$ we have already found the dotted rules and symbols that derive that substring. The following lemma tells us that we can then find the set of dotted rules that derive $\alpha[i\ k]$ with many symbols before the dot.

Lemma 3.1. For $i < j < k$, let $S(i,j)$ be the set of dotted rules that derive $\alpha[i\ j]$, and $S'(j,k)$ the set of symbols that derive $\alpha[j\ k]$. The set of dotted rules that derive $\alpha[i\ k]$ with many symbols before the dot is

$$\bigcup_{i<j<k} \text{AdvanceDot}(S(i,j),S'(j,k))$$

Proof. We have

$$\bigcup_{i<j<k} \text{AdvanceDot}(\{(B \rightarrow \beta.\beta_1) \in DR \mid \beta \stackrel{*}{\Rightarrow} \alpha[i\ j]\}, \{A \mid A \stackrel{*}{\Rightarrow} \alpha[j\ k]\})$$

=

$$\bigcup_{i<j<k} \{(B \rightarrow \beta A.\beta_2) \in DR \mid \beta \stackrel{*}{\Rightarrow} \alpha[i\ j] \wedge A \stackrel{*}{\Rightarrow} \alpha[j\ k]\}$$

by definition of AdvanceDot

=

$\{(B \rightarrow \beta A.\beta_2) \in DR \mid (\exists\ j.\ i < j < k \wedge \beta \stackrel{*}{\Rightarrow} \alpha[i\ j] \wedge A \stackrel{*}{\Rightarrow} \alpha[j\ k])\}$□

As noted above, this is the set of dotted rules that derive $\alpha[i\ k]$ with more than one symbol before the dot.

Definition. If S is a set of dotted rules, finished(S) = $\{\ A \mid (A \rightarrow \beta.) \in S\ \}$.

When the dot reaches the end of the right side of a rule, the parser has finished building the symbol on the left side—hence the name *finished*. For example, finished($\{(\ d \rightarrow k\ f\ .),(a \rightarrow\ .\ b)\}$) is the set $\{\ d\}$.

The next lemma tells us that if we have the set of dotted rules that derive $\alpha[i\ k]$ with many symbols before the dot, we can construct the set of symbols that derive $\alpha[i\ k]$.

Lemma 3.2. Suppose length(α) > 1 and S is the set of dotted rules that derive α with more than one symbol before the dot. The set of symbols that derive α is close(finished(S)).

Proof. Suppose first that $A \in$ close(finished(S)). Then for some B, $A \stackrel{+}{\Rightarrow} B$, $(B \rightarrow \beta.)$ is a dotted rule, and $\beta \stackrel{*}{\Rightarrow} \alpha$. Then $A \stackrel{+}{\Rightarrow} \alpha$. Suppose next that A derives α. We show by induction that if t is a derivation tree in G and $A \stackrel{+}{\Rightarrow} \alpha$ by t, then $A \in$ close(finished (S)). t contains more than one node because length(α) > 1, so there is a rule $(A \rightarrow A_1 \ldots A_n)$ that admits the root of t. If $n > 1$, $(A \rightarrow A_1 \ldots A_n.) \in S$ and A is in close(finished(S)). If $n = 1$ then $A_1 \stackrel{+}{\Rightarrow} \alpha$ and by induction hypothesis $A_1 \in$ close(finished(S)). Since $A \stackrel{+}{\Rightarrow} A_1$, $A \in$ close(finished(S)).

In our example grammar, the set of dotted rules deriving $\alpha[0\ 2] = gh$ with more than one symbol before the dot is $\{(d \rightarrow k f.)\}$. finished($\{(\ d \rightarrow k f.)\}$) is $\{\ d\}$, and close($\{\ d\}$) = $\{\ a,b,c,d\}$. It is easy to check that these are all the symbols that derive gh.□

Definitions. RuleTable is the set of dotted rules $(A \rightarrow .\alpha)$ such that $(A \rightarrow \alpha)$ is a rule of G. If S is a set of symbols, NewRules(S) is AdvanceDot(RuleTable, S).

In our example grammar, NewRules ($\{k\}$) = $\{(\ d \rightarrow k\ .\ f)\}$.

Lemma 3.3. If S is the set of symbols that derive α,

the set of dotted rules that derive α with one symbol before the dot is NewRules(S).

Proof. Expanding the definitions gives Advance Dot({($A \rightarrow .\beta \mid (A \rightarrow \beta) \in P$}, { $C \mid C \stackrel{*}{\Rightarrow} \alpha$}) = {($A \rightarrow C.\beta'$) ($A \rightarrow C\beta'$) $\in P \wedge C \stackrel{*}{\Rightarrow} \alpha$}. This is the set of dotted rules that derive α with one symbol before the dot.

Let terminals(i,k) be the set of terminals that derive $\alpha[i\ k]$; that is, if $i + 1 = k$ then terminals(i,k) = {$\alpha[i\ k]$}, and otherwise terminals(i,k) = $\emptyset$. Let α be a string of length $L > 0$. For $0 \leq i < k \leq L$, define

dr(i,k) =
if i + 1 = k
then NewRules(close({ $\alpha[i\ i + 1]$}))
else (let $\text{rules}_1 = \bigcup_{i<j<k}$ AdvanceDot(dr(i,j), [finished(dr(j,k)) ∪ terminals (j,k)])
(let rules_2 = NewRules(close(finished(rules_1)))
$\text{rules}_1 \cup \text{rules}_2$))

Theorem 3.1. For $0 \leq i < k \leq L$, $dr(i,k)$ is the set of dotted rules that derive $\alpha[i\ k]$.

Proof. By induction on the length of $\alpha[i\ k]$. If the length is 1, then $i + 1 = k$. The algorithm returns NewRules(close({$\alpha[i\ i + 1]$})). close({ $\alpha[i\ i + 1]$}) is the set of symbols that derive $\alpha[\ i\ i + 1]$ (by the definition of ChainTable), and NewRules(close({$\alpha[i\ i + 1]$})) is the set of dotted rules that derive $\alpha[i\ i + 1]$ with one symbol before the dot (by lemma 3.3). No rule can derive $\alpha[i\ i + 1]$ with many symbols before the dot, because $\alpha[i\ i + 1]$ has only one symbol. Then NewRules(close ({$\alpha[i\ k]$})) is the set of all dotted rules that derive $\alpha[i\ k]$.

Suppose $\alpha[i\ k]$ has a length greater than 1. If $i < j < k$, $dr(i,j)$ contains the dotted rules that derive $\alpha[i\ j]$ and $dr(j,k)$ contains the dotted rules that derive $\alpha[j\ k]$, by induction hypothesis. Then finished($dr(j,k)$) is the set of nonterminals that derive $\alpha[j\ k]$, and terminals(j,k) is the set of terminals that derive $\alpha[j\ k]$, so the union of these two sets is the set of all symbols that derive $\alpha[j\ k]$. By lemma 3.1, rules_1 is the set of dotted rules that derive $\alpha[i\ k]$ with many symbols before the dot. By lemma 3.2, close(finished(rules_1)) is the set of symbols that derive $\alpha[i\ k]$, so by lemma 3.3 rules_2 is the set of dotted rules that derive $\alpha[i\ k]$ with one symbol before the dot. The union of rules_1 and rules_2 is the set of dotted rules that derive $\alpha[i\ k]$, and this completes the proof.□

Suppose we are parsing the string gh with our example grammar. We have

$dr(0,1) = \{(k \rightarrow g\ .), (d \rightarrow k\ .\ f)\}$
$dr(1,2) = \{(f \rightarrow h\ .)\}$
$dr(0,2) = \{(d \rightarrow kf\ .), (c \rightarrow d\ .\ ), (b \rightarrow c\ .\ ), (a \rightarrow b\ .\ )\}$

5 The Parser with Empty Symbols

Throughout this section, G is an arbitrary depth-bounded unification grammar, which may contain rules whose right side is empty. If there are empty rules in the grammar, the parser will require a table of symbols that derive the empty string, which we also call the table of empty symbols. Let E_d be the set of symbols that derive the empty string by a derivation of depth d, and let E'_d be the set of symbols that derive the empty string by a derivation of depth d or less. Since the grammar is depth-bounded, it suffices to construct E_d for successive values of d until a $D > 0$ is found such that E_D is the empty set.

E_1 is the set of symbols that immediately derive the empty string; that is, the set of all A such that $(A \rightarrow e)$ is a rule. $A \in E_{d+1}$ iff there is a rule $(A \rightarrow B_1 \ldots B_n)$ such that for each i, $B_i \stackrel{*}{\Rightarrow} e$ at depth d_i, and d is the maximum of the d_i's. In other words: $A \in E_{d+1}$ iff there is a rule $(A \rightarrow \alpha B \beta)$ such that $B \in E_d$ and every symbol of α and β is in E'_d.

Let DR be a set of dotted rules and S a set of symbols. Define

AdvanceDot*(DR,S) =
if DR = $\emptyset$ then $\emptyset$
else (DR ∪ AdvanceDot*(AdvanceDot(DR,S),S))

If DR is the set of ground instances of a finite set of rules with variables, there is a finite bound on the length of the right sides of rules in DR (because the right side of a ground instance of a rule r has the same length as the right side of r). If L is the length of the right side of the longest rule in DR, then AdvanceDot*(DR,S) is well defined because the recursion stops at depth L or before. Clearly AdvanceDot*(DR,S) is the set of rules $(A \rightarrow \alpha\beta.\gamma)$ such that $(A \rightarrow \alpha.\ \beta\gamma) \in DR$ and every symbol of β is in S.

Let
S_1 = AdvanceDot*(RuleTable, E'_d)
S_2 = AdvanceDot(S_1, E'_d)
S_3 = AdvanceDot*(S_2, E'_d)
S_4 = finished(S_3)

S_1 is the set of dotted rules $(A \rightarrow \alpha.\beta_0)$ such that every symbol of α is in E'_d. S_2 is then the set of dotted rules $(A \rightarrow \alpha B.\beta_1)$ such that $B \in E_d$ and every symbol of α is in E'_d. Therefore S_3 is the set of dotted rules $(A \rightarrow \alpha B \beta.\beta_2)$ such that $B \in E_d$ and every symbol of α and β is in E'_d. Finally $S4$ is the set of symbols A such that for some rule $(A \rightarrow \alpha B \beta)$, $B \in E_d$ and every symbol of α and β is in E'_d. Then S_4 is E_{d+1}. In this way we can construct E_d for increasing values of d until the table of empty symbols is complete.

Here is an example grammar with symbols that derive the empty string:

$(a \rightarrow e)$
$(b \rightarrow e)$
$(c \rightarrow ab)$
$(k \rightarrow cfcgc)$
$(f \rightarrow r)$
$(g \rightarrow s)$

The terminal symbols are r and s. In this grammar, $E_1 = \{a,b\}$, $E_2 = \{c\}$, and $E_3 = \emptyset$.

Definitions Let EmptyTable be the set of symbols that derive the empty string. If S is a set of dotted rules, let SkipEmpty(S) be AdvanceDot*(S, EmptyTable).

Note that SkipEmpty(S) is the set of dotted rules $(A \rightarrow \alpha\beta_1.\beta_2)$ such that $(A \rightarrow \alpha.\beta_1\beta_2) \in S$ and $\beta_1 \stackrel{*}{\Rightarrow} e$.

SkipEmpty(S) contains every dotted rule that can be formed from a rule in S by moving the dot past zero or more symbols that derive the empty string. In the example grammar EmptyTable = $\{a,b,c\}$, so SkipEmpty($\{(k \rightarrow . cfcgc)\}$) = $\{(k \rightarrow . cfcgc), (k \rightarrow c . fcgc)\}$. If the dotted rules in S all derive α, then the dotted rules in SkipEmpty(S) also derive α.

Let C_d be the set of pairs $[A\ B]$ such that $A \stackrel{*}{\Rightarrow} B$ by a derivation tree in which the unique leaf labelled B is at depth d (note: this does not imply that the tree is of depth d). C_1 is the set of pairs $[A\ B]$ such that $(A \rightarrow \alpha B\beta)$ is a rule and every symbol of α and β derives the empty string. C_1 is easily computed using SkipEmpty. Also $C_{d+1} = \text{link}(C_d, C_1)$, so we can construct the chain table as before.

In the example grammar there are no A and B such that $A \stackrel{*}{\Rightarrow} B$, but if we added the rule $(k \rightarrow cfc)$, we would have $k \stackrel{*}{\Rightarrow} f$. Note that k derives f by a tree of depth 3, but the path from the root of this tree to the leaf labeled f is of length one. Therefore the pair $[k\ f]$ is in C_1.

The parser of Section 4 relied on the distinction between dotted rules with one and many symbols before the dot. If empty symbols are present, we need a slightly more complex distinction. We say that the string α derives β using one symbol if there is a derivation of β from α in which exactly one symbol of α derives a non-empty string. We say that α derives β using many symbols if there is a derivation of β from α in which more than one symbol of α derives a nonempty string. If a string α derives a string β, then α derives β using one symbol, or α derives β using many symbols, or both. In the example grammar, cfc derives r using one symbol, and $cfcg$ derives rs using many symbols.

We say that a dotted rule derives β using one (or many) symbols if the string before the dot derives β using one (or many) symbols. Note that a dotted rule derives $\alpha[i\ k]$ using many symbols iff it can be written as $(A \rightarrow \beta B\beta'.\beta_1)$ where $\beta \stackrel{*}{\Rightarrow} \alpha[i\ j]$, $B \stackrel{*}{\Rightarrow} \alpha[j\ k]$, $\beta' \stackrel{*}{\Rightarrow} e$, and $i < j < k$. This is true because whenever a dotted rule derives a string using many symbols, there must be a last symbol before the dot that derives a nonempty string. Let B be that symbol; it is followed by a β' that derives the empty string, and preceded by a β that must contain at least one more symbol deriving a non-empty string.

We prove lemmas analogous to 3.1, 3.2, and 3.3.

Lemma 4.1. For $i < j < k$ let $S(i,j)$ be the set of dotted rules that derive $\alpha[i\ j]$ and $S'(j,k)$ the set of symbols that derive $\alpha[j\ k\ ]$. The set of dotted rules that derive $\alpha[i\ k]$ using many symbols is

$$\text{SkipEmpty}(\bigcup_{i<j<k} \text{AdvanceDot}(S(i,j), S'(j,k)))$$

Proof. Expanding definitions and using the argument of lemma 3.3 we have

$$\text{SkipEmpty}(\bigcup_{i<j<k} \text{AdvanceDot}(\{(B \rightarrow \beta.\beta_1) \in DR \mid \beta \stackrel{*}{\Rightarrow} \alpha[i\ j]\}, \{A \mid A \stackrel{*}{\Rightarrow} \alpha[j\ k]\})) =$$

$$\text{SkipEmpty}(\{(B \rightarrow \beta A.\beta_2) \in DR \mid (\exists\ j.\ i < j < k \wedge \beta \stackrel{*}{\Rightarrow} \alpha[i\ j] \wedge A \stackrel{*}{\Rightarrow} \alpha[j\ k])\})$$

This in turn is equal to

$$\{(B \rightarrow \beta A\beta'.\beta_3) \in DR \mid (\exists\ j.\ i < j < k \wedge \beta \stackrel{*}{\Rightarrow} \alpha[i\ j] \wedge A \stackrel{*}{\Rightarrow} \alpha[j\ k]) \wedge \beta' \stackrel{*}{\Rightarrow} e\}$$

This is the set of rules that derive $\alpha[i\ k]$ using many symbols, as noted above.

If we have $\alpha = rs$, then the set of dotted rules that derive $\alpha[0\ 1]$ is

$$\{(f \rightarrow r\ .), (k \rightarrow cf\ .\ cgc), (k \rightarrow cfc\ .\ gc)\}$$

The set of symbols that derive $\alpha[1\ 2]$ is $\{g,s\}$. The set of dotted rules that derive $\alpha[0\ 2]$ using many symbols is

$$\{(k \rightarrow cfcg\ .\ c), (k \rightarrow cfcgc\ .)\}$$

Lemma 4.1 tells us that to compute this set we must apply SkipEmpty to the output of AdvanceDot. If we failed to apply SkipEmpty we would omit the dotted rule $(k \rightarrow cfcgc\ .)$ from our answer.

Lemma 4.2. Suppose length(α) > 1 and S is the set of dotted rules that derive α using many symbols. The set of symbols that derive α is close(finished(S)).

Proof. By induction as in Lemma 3.2.

Definitions. Let RuleTable′ be SkipEmpty($\{(A \rightarrow .\alpha)\ (A \rightarrow \alpha) \in P\}$) = $\{(A \rightarrow \alpha.\alpha') \in DR \mid \alpha \stackrel{*}{\Rightarrow} e\}$. If S is a set of symbols let NewRules′(S) be SkipEmpty(AdvanceDot(RuleTable′,S)).

RuleTable′ is like the RuleTable defined in Section 4, except that we apply SkipEmpty. In the example grammar, this means adding the following dotted rules:

$(c \rightarrow a\ .\ b)$
$(c \rightarrow ab\ .)$
$(k \rightarrow c\ .\ fcgc)$

NewRules′($\{f\}$) is equal to $\{(k \rightarrow cf\ .\ cgc), (k \rightarrow cfc\ .\ gc)\}$.

The following lemma tells us that NewRules′ will perform the task that NewRules performed in Section 4.

Lemma 4.3. If S is the set of symbols that derive α, the set of dotted rules that derive α using one symbol is NewRules′(S).

Proof. Expanding definitions gives

$$\text{SkipEmpty}(\text{AdvanceDot}(\{(A \rightarrow \beta.\beta_1) \in DR \mid \beta \stackrel{*}{\Rightarrow} e\}, \{C \mid C \stackrel{*}{\Rightarrow} \alpha\}))$$

$$=$$

$$\text{SkipEmpty}(\{(A \rightarrow \beta C.\beta_2) \in DR \mid \beta \stackrel{*}{\Rightarrow} e \wedge C \stackrel{*}{\Rightarrow} \alpha\})$$

$$=$$

$$\{(A \rightarrow \beta C\beta'.\beta_3) \in DR \mid \beta \stackrel{*}{\Rightarrow} e \wedge C \stackrel{*}{\Rightarrow} \alpha \wedge \beta' \stackrel{*}{\Rightarrow} e\}$$

This is the set of dotted rules that derive α using one symbol, by definition.

Let α be a string of length L. For $0 \le i < k \le L$, define

dr(i,k) =
if i + 1 = k
then NewRules′(close($\{\alpha[i\ k]\}$))
else (let rules$_1$ =

SkipEmpty($\bigcup_{i<j<k}$ AdvanceDot(dr(i,j), [finished(dr(j,k)) ∪ terminals (j,k)]))
(let rules$_2$ = NewRules'(close (finished(rules$_1$)))
rules$_1$ ∪ rules$_2$))

Theorem 4.1. *dr(i,k)* is the set of dotted rules that derive $\alpha[i\ k]$.

Proof. By induction on the length of $\alpha[i\ k]$ as in the proof of theorem 3.1, but with lemmas 4.1, 4.2, and 4.3 replacing 3.1, 3.2, and 3.3, respectively.□

If α = rs we find that
dr(0,1) = {(*f* → *r* .),(*k* → *cf* . *cgc*),(*k* → *cfc* . *gc*)}
dr(1,2) = {(*g* → *s*.)}
dr(0,2) = {(*k* → *cfcg* . *c*),(*k* → *cfcgc* .)}

6 The Parser with Top-Down Filtering

We have described two parsers that set *dr(i,k)* to the set of dotted rules that derive $\alpha[i\ k]$. We now consider a parser that uses top-down filtering to eliminate some useless rules from *dr(i, k)*. Let us say that *A* follows β if the start symbol derives a string beginning with βA. A dotted rule $(A \rightarrow \chi)$ follows β if *A* follows β. The new algorithm will set *dr(i,k)* to the set of dotted rules that derive $\alpha[i\ k]$ and follow $\alpha[0\ i]$.

If *A* derives a string beginning with *B*, we say that *A* can begin with *B*. The new algorithm requires a prediction table, which contains all pairs [*A B*] such that *A* can begin with *B*. Let P_1 be the set of pairs [*A B*] such that $(A \rightarrow \beta B \beta')$ is a rule and $\beta \stackrel{*}{\Rightarrow} e$. Let P_{n+1} be $P_n \cup$ Link(P_n, P_1).

Lemma 5.1. The set of pairs [*A B*] such that *A* can begin with *B* is the union of P_n for all $n \geq 1$.

Proof. By induction on the tree by which *A* derives a string beginning with *B*. Details are left to the reader.□

Our problem is to construct a finite representation for the prediction table. To see why this is difficult, consider a grammar containing the rule
(f(a,s(X)) → f(a,X) g)
Computing the P_ns gives us the following pairs of terms:
[f(a,s(X)) f(a,X)]
[f(a,s(s(Y))) f(a,Y)]
[f(a,s(s(s(Z)))) f(a,Z)]
. . .
Thus if we try to build the prediction table in the obvious way, we get an infinite set of pairs of terms.

The key to this problem is to recognize that it is not necessary or even useful to predict every possible feature of the next input. It makes sense to predict the presence of traces, but predicting the subcategorization frame of a verb will cost more than it saves. To avoid predicting certain features, we use a weak prediction table; that is, a set of pairs of symbols that properly contains the set of all [*A B*] such that $A \stackrel{*}{\Rightarrow} B$. This weak prediction table is guaranteed to eliminate no more than what the ideal prediction table eliminates. It may leave some dotted rules in *dr(i,k)* that the ideal prediction table would remove, but it may also cost less to use.

Sato and Tamaki (1984) proposed to analyze the behavior of Prolog programs, including parsers, by using something much like a weak prediction table. To guarantee that the table was finite, they restricted the depth of terms occurring in the table. Shieber (1985b) offered a more selective approach—his program predicts only those features chosen by the user as most useful for prediction. Pereira and Shieber (1987) discuss both approaches. We will present a variation of Shieber's ideas that depends on using a sorted language.

To build a weak prediction table we begin with a set Q_1 of terms such that $P_1 \subseteq$ ground(Q_1). Define
LP(Q,Q') = {(s [x z]) | (∃ y,y'. [x y] ∈ Q ∧ [y' z] ∈ Q' ∧ s = m.g.u. of y and y')}
By Theorem 2.1, ground(*LP*(Q,Q')) = Link(ground(Q), ground(Q')). Let Q_{i+1} equal $Q_i \cup LP(Q_i,Q_1)$. Then by lemma 2.3 and induction,
$\bigcup_{i \geq 1} Pi \subseteq$ ground($\bigcup_{i \geq 1} Q_i$)
That is, the union of the Q_is represents a weak prediction table. Thus we have shown that if a weak prediction table is adequate, we are free to choose any Q_1 such that $P_1 \subseteq$ ground(Q_1).

Suppose that Q_D subsumes $LP(Q_D,Q_1)$. Then ground($LP(Q_D,Q_1)$) $\subseteq$ ground(Q_D) and ground(Q_{D+1}) = ground(Q_D) ∪ ground($LP(Q_D, Q_1)$) = ground(Q_D). Since ground(Q_{i+1}) is a function of ground(Q_i) for all i, it follows that ground(Q_i) = ground(Q_D) for all $i \geq D$, so ground(Q_D) = ($\cup\ i \geq 1$ ground(Q_i)). That is, Q_D is a finite representation of a weak prediction table. Our problem is to choose Q_1 so that Q_D subsumes Q_{D+1} for some D.

Let s_1 and s_2 be sorts. In section 2 we defined $s_1 > s_2$ if there is a function letter of sort s_1 that has an argument of sort s_2. Let $>^*$ be the transitive closure of $>$; a sort t is cyclic if $t >^* t$, and a term is cyclic if its sort is cyclic. P_1 is equal to
{[A B] | (A → β.Bβ') ∈ RuleTable'}
so we can build a representation for P_1. Let us form Q_1 from this representation by replacing all cyclic terms with new variables. More exactly, we apply the following recursive transformation to each term t in the representation of P_1:
transform(f(t$_1$...t$_n$)) =
if the sort of f is cyclic
then new-variable()
else f(transform (t$_1$)...transform(t_n))
where *new-variable* is a function that returns a new variable each time it is called.

Then $P_1 \subseteq$ ground(Q_1), and Q_1 contains no function letters of cyclic sorts. For example, if the function letter s belongs to a cyclic sort, we will turn
[*f(a,s(s(X))) f(a,X)*]
into
[*f(a,Z) f(a,Y)*]
If Q_1 = {[*f*(*a*,*Z*),*f*(*a*,*Y*)]}, then Q_2 = {[*f*(*a*,*V*),*f*(*a*,*W*)], so Q_1 subsumes Q_2, and Q_1 is already a finite representation of a weak prediction table. The following lemma

shows that in general, the Q_1 defined above allows us to build a finite representation of a weak prediction table.

Lemma 5.2. Let Q_1 be a set of pairs of terms that contains no function letters of cyclic sorts, and let Q_i be as defined above for all $i > 1$. Then for some D, Q_D subsumes $LP(Q_D,Q_1)$.

Proof. Note first that since unification never introduces a function letter that did not occur in the input, Q_i contains no function letters of cyclic sort for any $i \geq 1$.

Let C be the number of noncyclic sorts in the language. Then the maximum depth of a term that contains no function letters of cyclic sorts is $C + 1$. Consider a term as a labeled tree, and consider any path from the root of such a tree to one of its leaves. The path can contain at most one variable or function letter of each noncyclic sort, plus one variable of a cyclic sort. Then its length is at most $C + 1$.

Consider the set S of all pairs of terms in L that contain no function letters of cyclic sorts. Let us partition this set into equivalence classes, counting two terms equivalent if they are alphabetic variants. We claim that the number of equivalence classes is finite. Since there is a finite bound on the depths of terms in S, and a finite bound on the number of arguments of a function letter in S, there is a finite bound V on the number of variables in any term of S. Let $v_1 \ldots v_K$ be a list of variables containing V variables from each sort. Then there is a finite number of pairs in S that use only variables from $v_1 \ldots v_K$; let S' be the set of all such pairs. Now each pair p in S is an alphabetic variant of a pair in S', for we can replace the variables of p one-for-one with variables from $v_1 \ldots v_K$. Therefore the number of equivalence classes is no more than the number of elements in S'. We call this number E. We claim that Q_D subsumes $LP(Q_D,Q_1)$ for some $D \leq E$.

To see this, suppose that Q_i does not subsume $LP(Q_i,Q_1)$ for all $i < E$. If Q_i does not subsume $LP(Q_i,Q_1)$, then Q_{i+1} intersects more equivalence classes than Q_i does. Since Q_1 intersects at least one equivalence class, Q_E intersects all the equivalence classes. Therefore Q_E subsumes $LP(Q_E,Q_1)$, which was to be proved.□

This lemma tells us that we can build a weak prediction table for any grammar by throwing away all subterms of cyclic sort. In the worst case, such a table might be too weak to be useful, but our experience suggests that for natural grammars a prediction table of this kind is very effective in reducing the size of the $dr(i,k)$ s. In the following discussion we will assume that we have a complete prediction table; at the end of this section we will once again consider weak prediction tables.

Definitions. If S is a set of symbols, let first(S) = $S \cup \{ B \mid (\exists A \in S.\ [A\ B] \in \text{PredTable} \}$. If PredTable is indeed a complete prediction table, first(S) is the set of symbols B such that some symbol in S can begin with B. If R is a set of dotted rules let next(R) = $\{B \mid (\exists A,\beta,\beta'.\ (A \rightarrow \beta.B\beta') \in R \}$.

Consider the following example grammar:

start → a
$a \rightarrow rg$
$c \rightarrow rh$
$g \rightarrow s$
$h \rightarrow s$

The terminal symbols are r and s. In this grammar first({start}) = {start, a,r}, and next({($a \rightarrow r\ .\ g$)}) = {g}.〉

The following lemma shows that we can find the set of symbols that follow $\alpha[0\ j]$ if we have a prediction table and the sets of dotted rules that derive $\alpha[i\ j]$ for all $i < j$.

Lemma 5.3. Let j satisfy $0 \leq j \leq \text{length}(\alpha)$. Suppose that for $0 \leq i < j$, $S(i)$ is the set of dotted rules that follow $\alpha[0\ i]$ and derive $\alpha[i\ j]$ (if $j = 0$ this is vacuous). Let *start* be the start symbol of the grammar. Then the set of symbols that follow $\alpha[0\ j]$ is

first(if j = 0
{start}
($\bigcup_{0 \leq i \leq < j}$ next(S(i))))

Proof. We show first that every member of the given set follows $\alpha[0\ j]$. If $j = 0$, certainly every member of first({start}) follows $\alpha[0\ 0] = e$. If $j > 0$, suppose that C follows $\alpha[0\ i]$, $(C \rightarrow \beta B\beta')$ is a rule, and $\beta \stackrel{*}{\Rightarrow} \alpha[i\ j]$; then clearly B follows $\alpha[0\ j]$.

Next we show that if A follows $\alpha[0\ j]$, A is in the given set. We prove by induction on d that if start $\stackrel{*}{\Rightarrow} \alpha[0\ j]A\alpha'$ by a tree t, and the leaf corresponding to the occurrence of A after $\alpha[0\ j]$ is at depth d in t, then A belongs to the given set. If $d = 0$, then A = start, and $j = 0$. We must prove that start ∈ first({start}), which is obvious.

If $d > 0$ there are two cases. Suppose first that the leaf n corresponding to the occurrence of A after $\alpha[0\ j]$ has younger brothers dominating a nonempty string (younger brothers of n are children of the same father occurring to the left of n). Then the father of n is admitted by a rule of the form $(C \rightarrow \beta A\beta')$. C is the label of the father of n, and β consists of the labels of the younger brothers of n in order. Then $\beta \stackrel{*}{\Rightarrow} \alpha[i\ j]$, where $0 \leq i < j$. Removing the descendants of n's father from t gives a tree t' whose yield is $\alpha[0\ i]C\alpha'$. Therefore C follows $\alpha[0\ i]$. We have shown that $(C \rightarrow \beta A\beta')$ is a rule, C follows $\alpha[0\ i]$, and $\beta \stackrel{*}{\Rightarrow} \alpha[i\ j]$. Then $(C \rightarrow \beta.A\beta') \in S(i)$, $A \in \text{next}(S(i))$, and $A \in (\cup\ 0 \leq i < j\ \text{next}(S(i)))$.

Finally suppose that the younger brothers of n dominate the empty string in t. Then if C is the label of n's father, C can begin with A. Removing the descendants of n's father from t gives a tree t' whose yield begins with $\alpha[0\ j]C$. Then C belongs to the given set by induction hypothesis. If $C \in \text{first}(X)$ and C can begin with A, then $A \in \text{first}(X)$. Therefore A belongs to the given set. This completes the proof.

As an example, let $\alpha = rs$. Then the set of dotted rules that derive $\alpha[0\ 1]$ and follow $\alpha[0\ 0]$ is $\{(a \rightarrow r\ .\ g)\}$. The dotted rule $(c \rightarrow r\ .\ h)$ derives $\alpha[0\ 1]$, but it does not follow $\alpha[0\ 0]$ because c is not an element of first({start}).

We are finally ready to present the analogs of lemmas 3.1, 3.2, and 3.3 for the parser with prediction. Where the earlier lemmas mentioned the set of symbols (or dotted rules) that derive $\alpha[i\ j]$, these lemmas mention the set of symbols (or dotted rules) that follow $\alpha[0\ i]$ and derive $\alpha[i\ j]$.

Lemma 5.4. Let α be a nonempty string. Suppose that for $i < j < k$, $S(i,j)$ is the set of dotted rules that follow $\alpha[0\ i]$ and derive $\alpha[i\ j]$, while $S'(j,k)$ is the set of symbols that follow $\alpha[0\ j]$ and derive $\alpha[j\ k]$. The set of dotted rules that follow $\alpha[0\ i]$ and derive $\alpha[i\ k]$ using many symbols is

SkipEmpty($\bigcup_{i<j<k}$ AdvanceDot(S(i,j),S'(j,k)))

Proof. Expanding definitions and using the same argument as in lemma 3.1, we have

SkipEmpty($\bigcup_{i<j<k}$ AdvanceDot({(B → $\beta.\beta_1$) ∈ DR | B follows α[0 i] ∧ $\beta \overset{*}{\Rightarrow} \alpha$[i j]}
{A | A follows α[0 j] ∧ A $\overset{*}{\Rightarrow} \alpha$[j k]}) =
SkipEmpty({(B → βA.β_2) ∈ DR | B follows α[0 i]
∧ (∃ j. i<j <k ∧ $\beta \overset{*}{\Rightarrow} \alpha$[i j] ∧ A follows α[0 j] ∧ A $\overset{*}{\Rightarrow}$ α[j k])})

If B follows $\alpha[0\ i]$, $(B \rightarrow \beta A \beta_2)$ is a rule, and $\beta \overset{*}{\Rightarrow} \alpha[i\ j]$, then A follows $\alpha[0\ j]$. Therefore the statement that A follows $\alpha[0\ j]$ is redundant and can be deleted, giving

SkipEmpty({(B → βA.β_2) ∈ DR | B follows α[0 i]
∧ (∃ j. i<j<k ∧ $\beta \overset{*}{\Rightarrow} \alpha$[i j] ∧ A $\overset{*}{\Rightarrow} \alpha$[j k])})

This in turn is equal to

{(B → βA$\beta'.\beta_3$) ∈ DR | B follows α[0 i]
∧ (∃ j. i<j<k ∧ $\beta \overset{*}{\Rightarrow} \alpha$[i j] ∧ A $\overset{*}{\Rightarrow} \alpha$[j k]) ∧ $\beta' \overset{*}{\Rightarrow}$ e}

This is the set of dotted rules that follow $\alpha[0\ i]$ and derive $\alpha[i\ k]$ using many symbols.□

Lemma 5.5. Suppose length($\alpha[i\ j]$) > 1, S is the set of symbols that follow $\alpha[0\ i]$, and S' is the set of dotted rules that follow $\alpha[0\ i]$ and derive $\alpha[i\ j]$ using many symbols. Then $S \cap$ close(finished(S')) is the set of symbols that follow $\alpha[0\ i]$ and derive $\alpha[i\ j]$.

Proof. S' is a subset of the set of dotted rules that derive $\alpha[i\ j]$, so by lemma 4.2 and monotonicity, close(finished(S')) is a subset of the set of symbols that derive $\alpha[i\ j]$. Therefore every symbol in $S \cap$ close(finished(S'))) derives $\alpha[i\ j]$ and follows $\alpha[0\ i]$. This proves inclusion in one direction.

For the other direction, suppose A follows $\alpha[0\ i]$ and derives $\alpha[i\ j]$. Then by lemma 4.2 there is a dotted rule $(B \rightarrow \beta.)$ such that $\beta \overset{*}{\Rightarrow} \alpha[i\ j]$ using many symbols and $A \overset{*}{\Rightarrow} B$. Then B follows $\alpha[0\ i]$, so B is in finished(S'), which means that A is in $S \cap$ close(finished(S')).□

Definition. If S is a set of symbols and R a set of dotted rules, filter(S,R) is the set of rules in R whose left sides are in S. In other words, filter(S,R) = {($A \rightarrow \beta.\beta'$) ∈ R | $A \in S$ }.

Lemma 5.6. Suppose S is the set of symbols that follow $\alpha[0\ i]$, and S' is the set of symbols that follow $\alpha[0\ i]$ and derive $\alpha[i\ j]$. Then the set of rules that follow $\alpha[0\ i]$ and derive $\alpha[i\ j]$ using one symbol is filter(S,NewRules'(S')).

Proof: S' is a subset of the set of symbols that derive $\alpha[i\ j]$. By lemma 4.3 and monotonicity, we know that every dotted rule in NewRules'(S') derives $\alpha[i\ j]$ using one symbol. Therefore every dotted rule in filter(S,NewRules(S')) follows $\alpha[0\ i]$ and derives $\alpha[i\ j]$ using one symbol. This proves inclusion in one direction.

For the other direction, consider any dotted rule that follows $\alpha[0\ i]$ and derives $\alpha[i\ j]$ using one symbol; it can be written in the form $(A \rightarrow \beta B \beta'.\beta_1)$, where β and β' derive e, B derives $\alpha[i\ j]$, and A follows $\alpha[0\ i]$. Since $\beta \overset{*}{\Rightarrow} e$, B follows $\alpha[0\ i]$. Therefore $B \in S'$ and $(A \rightarrow \beta B \beta'.\beta_1)$ is in NewRules'(S'). Since A follows $\alpha[0\ i]$, $(A \rightarrow \beta B \beta'.\beta_1)$ is in filter(S,NewRules'(S')).

Let α be a string of length L. For $0 \le i < k \le L$, define

```
pred(j) =
first(if j = 0
    then {Start}
    else (∪ 0≤i<j next(dr(i,j))))
dr(i,k) =
  if i + 1 = k
  then filter(pred(i),NewRules'([pred(i) ∩ close
    ({α[i k]})]))
  else (let rules1 =
            SkipEmpty(∪ AdvanceDot(dr(i,j),
                        i<j<k        [finished(dr(j,k))
                                   ∪ terminals(j,k)])
      (let rules2 = filter(pred(i),
                        NewRules'([pred(i) ∩
                          close(finished(rules1))]))
        rules1 ∪ rules2))
```

Note that the new version of $dr(i,k)$ is exactly like the previous version except that we filter the output of close by intersecting it with pred(i), and we filter the output of NewRules' by applying the function filter.

Theorem 5.6 For $0 \le k \le L$, $pred(k)$ is the set of symbols that follow $\alpha[0\ i]$, and if $0 \le i < k$, $dr(i,k)$ is the set of dotted rules that follow $\alpha[0\ i]$ and derive $\alpha[i\ k]$.

Proof. This proof is similar to the proof of theorem 3.4, but it is more involved because we must show that $pred(k)$ has the desired values. Once more we argue by induction, but this time it is a double induction: an outer induction on k, and an inner induction on the length of strings that end at k.

We show by induction on k that $pred(k)$ has the desired value and for $0 \le i < k$, $dr(i,k)$ has the desired value. If $k = 0$, lemma 5.3 tells us that $pred(0)$ is the set of symbols that follow $\alpha[0\ 0]$, and the second part of the induction hypothesis is vacuously true.

If $k > 0$, we first show by induction on the length of $\alpha[i\ k]$ that $dr(i,k)$ has the desired value for $0 \le i < k$. This part of the proof is much like the proof of 3.4. If $\alpha[i\ k]$ has length 1, then $pred(i)$ is the set of symbols that follow $\alpha[0\ i]$ by the hypothesis of the induction on k. Then $pred(i) \cap$ close({$\alpha[i\ k]$}) is the set of symbols that follow $\alpha[0\ i]$ and derive $\alpha[i\ k]$, so lemma 5.6 tells us that

filter($pred$ (i),NewRules'($pred(i) \cap$ close({$\alpha[i\ k]$}))))

is the set of dotted rules that follow $\alpha[0\ i]$ and derive $\alpha[i\ k]$.

If length($\alpha[i\ k]$) > 1, consider any j such that $i < j < k$. $dr(i,j)$ and $dr(j,k)$ have the desired values by induction

hypothesis. Then lemma 5.4 tells us that $rules_1$ is the set of dotted rules that follow $\alpha[0\ i]$ and derive $\alpha[i\ k]$ using many symbols. $pred(i)$ is the set of symbols that follow $\alpha[0\ i]$, so $pred(i) \cap$ close(finished($rules_1$)) is the set of symbols that follow $\alpha[0\ i]$ and derive $\alpha[i\ k]$, by lemma 5.5. Therefore $rules_2$ is the set of dotted rules that follow $\alpha[0\ i]$ and derive $\alpha[i\ k]$ using one symbol, by lemma 5.6. The union of $rules_1$ and $rules_2$ is the set of dotted rules that follow $\alpha[0\ i]$ and derive $\alpha[i\ k]$, and this completes the inner induction.

To complete the outer induction, we use lemma 5.3 to show that $pred(k)$ is the set of symbols that follow $\alpha[0\ k]$. This completes the proof.□

Corollary: $Start \in$ finished($dr(0,L)$) iff α is a sentence of the language generated by G.

Suppose we are parsing the string rs using the example grammar. Then we have

$pred(0) = \{start,a,r\}$
$dr(0,1) = \{(a \rightarrow r\ .\ g)\}$
$pred(1) = \{g,s\}$
$dr(1,2) = \{(g \rightarrow s\ .)\}$
$dr(0,2) = \{(a \rightarrow rg\ .),(start \rightarrow a\ .)\}$

We have proved the correctness of the parser when it uses an ideal prediction table. We must still consider what happens when the parser uses a weak prediction table.

Theorem 5.7. If PredTable is a superset of the set of all $[A\ B]$ such that A can begin with B, then $start \in$ finished($dr(0,L)$) iff α is a sentence of the language generated by G.

Proof. Note that the parser with filtering always builds a smaller $dr(i,k)$ than the parser without filtering. Since all the operations of the parser are monotonic, this is an easy induction. So if the parser with filtering puts the start symbol in $dr(0,L)$, the parser without filtering will do this also, implying that α is a sentence. Note also that the parser with filtering produces a larger $dr(i,k)$ given a larger PredTable (again, this follows easily because all operations in the parser are monotonic). So if α is a sentence, the parser with the ideal prediction table includes Start in $dr(0,L)$, and so does the parser with the weak prediction table.□

7 Discussion and Implementation Notes

7.1 Related Work and Possible Extensions

The chief contribution of the present paper is to define a class of grammars on which bottom-up parsers always halt, and to give a semi-decision procedure for this class. This in turn makes it possible to prove a completeness theorem, which is impossible if one considers arbitrary unification grammars. One can obtain similar results for the class of grammars whose context-free backbone is finitely ambiguous—what Pereira and Warren (1983) called the offline-parsable grammars. However, as Shieber (1985b) observed, this class of grammars excludes many linguistically interesting grammars that do not use atomic category symbols.

The present parser (as opposed to the table-building algorithm) is much like those in the literature. Like nearly all parsers using term unification, it is a special case of Earley deduction (Pereira and Warren 1985). The tables are simply collections of theorems proved in advance and added to the program component of Earley deduction. Earley deduction is a framework for parsing rather than a parser. Among implemented parsers, BUP (Matsumota et al. 1983) is particularly close to the present work. It is a bottom-up left-corner parser using term unification. It is written in Prolog and uses backtracking, but by recording its results as clauses in the Prolog database it avoids most backtracking, so that it is close to a chart parser. It also includes top-down filtering, although it uses only category symbols in filtering. The paper includes suggestions for handling rules with empty right sides as well. The main difference from the present work is that the authors do not describe the class of grammars on which their algorithm halts, and as a result they cannot prove completeness.

The grammar formalism presented here is much simpler than many formalisms called "unification grammars." There are no meta-rules, no default values of features, no general agreement principles (Gazdar et al. 1986). We have found this formalism adequate to describe a substantial part of English syntax—at least, substantial by present-day standards. Our grammar currently contains about 300 syntactic rules, not counting simple rules that introduce single terminals. It includes a thorough treatment of verb subcategorization and less thorough treatments of noun and adjective subcategorization. It covers major construction types: raising, control, passive, subject-aux inversion, imperatives, *wh*-movement (both questions and relative clauses), determiners, and comparatives. It assigns parses to 85% of a corpus of 791 sentences. See Ayuso et al. 1988 for a description of the grammar.

It is clear that some generalizations are being missed. For example, to handle passive we enumerate by hand the rules that other formalisms would derive by meta-rule. We are certainly missing a generalization here, but we have found this crude approach quite practical—our coverage is wide and our grammar is not hard to maintain. Nevertheless, we would like to add meta-rules and probably some general feature-passing principles. We hope to treat them as abbreviation mechanisms—we would define the semantics of a general feature-passing principal by showing how a grammar using that principal can be translated into a grammar written in our original formalism. We also hope to add feature disjunction to our grammar (see Kasper 1987; Kasper and Rounds 1986).

Though our formalism is limited, it has one property that is theoretically interesting: a sharp separation between the details of unification and the parsing mechanism. We proved in Section 3 that unification allows us to compute certain functions and predicates on sets of grammatical expressions—symbolic products, unions,

and so forth. In Section 4 and 5 we assumed that these functions were available as primitives and used them to build bottom-up parsers. Nothing in Sections 4 and 5 depends on the details of unification. If we replace standard unification with another mechanism, we have only to re-prove the results of Section 3 and the correctness theorems of Sections 4 and 5 follow at once. To see that this is not a trivial result, notice that we failed to maintain this separation in Section 6. To show that one can build a complete prediction table, we had to consider the details of unification: we mentioned terms like "alphabetic variant" and "subsumption." We have presented a theory of bottom-up parsing that is general in the sense that it does not rely on a particular pattern-matching mechanism—it applies to any mechanism for which the results of Section 3 hold. We claim that these results should hold for any reasonable pattern-matching mechanism; the reader must judge this claim by his or her own intuition.

One drawback of this work is that depth-boundedness is undecidable. To prove this, show that any Turing machine can be represented as a unification grammar, and then show that an algorithm that decides depth-boundedness can also solve the halting problem. This result raises the question: is there a subset of the depth-bounded grammars that is strong enough to describe natural language, and for which membership is decidable?

Recall the context-free backbone of a grammar, described in the Introduction. One can form a context-free backbone for a unification grammar by keeping only the topmost function letters in each rule. There is an algorithm to decide whether this backbone is depth-bounded, and if the backbone is depth-bounded, so is the original grammar (because the backbone admits every derivation tree that the original grammar admits). Unfortunately this class of grammars is too restricted—it excludes rules like (major-category(n,2) → major-category(n,1)), which may well be needed in grammars for natural language.

Erasing everything but the top function letter of each term is drastic. Instead, let us form a "backbone" by applying the transformation of Section 6, which eliminates cyclic function letters. We can call the resulting grammar the acyclic backbone of the original grammar. We showed in Section 6 that if we eliminate cyclic function letters, then the relation of alphabetic variance will partition the set of all terms into a finite number of equivalence classes. We used this fact to prove that the algorithm for building a weak prediction table always halts. By similar methods we can construct an algorithm that decides depth-boundedness for grammars without cyclic function letters. Then the grammars whose acyclic backbones are depth-bounded form a decidable subset of the depth-bounded grammars. One can prove that this class of grammars generates the same languages as the off-line parsable grammars. Unlike the off-line parsable grammars, they do not require atomic category symbols. A forthcoming paper will discuss these matters in detail.

7.2 THE IMPLEMENTATION

Our implementation is a Common Lisp program on a Symbolics Lisp Machine. The algorithm as stated is recursive, but the implementation is a chart parser. It builds a matrix called "rules" and sets rules[i k] equal to $dr(i,k)$, considering pairs [i k] in the same order used for the induction argument in the proof. It also builds a matrix "symbols" and sets symbols[i k] to the set of symbols that derive α[i k], and a matrix *pred* with pred[i] equal to the set of symbols that follow α[0 i]. Currently the standard parser does not incorporate prediction. We have found that prediction reduces the size of the chart dramatically, but the cost of prediction is so great that a purely bottom-up parser runs faster.

Table 1. Chart Sizes and Total Time for Parsing with Prediction

Sentence	No Prediction	Categories	Traces	Traces and Verb Form
1	524	517	248	150
2	878	867	686	667
3	799	713	500	387
4	936	921	558	467
5	283	279	145	90
6	997	969	524	368
7	531	525	323	247
8	982	950	640	507
9	1519	1503	1007	711
10	930	920	495	400
11	2034	2014	1128	771
total time (in seconds)	917	2201	1538	1085

Table 1 presents the results of predicting different features on a sample of 11 sentences. It describes parsing without prediction, with prediction of categories only, with traces and categories, and finally with categories, traces, and verb form information. In each case it lists the total number of entries in the matrices "rules" and "symbols" for every sentence, and the total time to parse the 11 sentences. The reader should compare this table with the one in Shieber 1985. Shieber tried predicting subcategorization information along with categories. In our grammar there is a separate VP rule for each subcategorization frame, and this rule gives the categories of all arguments of the verb. Shieber eliminated these multiple VP rules by making the list of arguments a feature of the verb. Therefore by predicting categories alone, we get the same information that Shieber got by predicting subcategorization information. The table shows that for our grammar, prediction reduces the chart size drastically, but it is so costly that a straight bottom-up parser runs faster than any version of prediction.

The parsing tables for the present grammar are quite tractable. The largest table is the table of chain rules, which has 2,270 entries and takes under ten minutes to build. A prediction table that predicts categories, traces, and verb forms has 1,510 entries and takes six minutes to build.

In the special case of a context-free grammar, our parsing program is essentially the same as the parser of Graham et al. (1980), in particular algorithm 2.2 of that paper. The only significant differences are that their chart includes entries for empty substrings, which we omit, and that we record symbols while they record only dotted rules. When running on a context-free grammar, the parser takes time proportional to the cube of the length n of the input string—because the number of symbolic products is proportional to n^3, and the time for a symbolic product is independent of the input string. This result also holds for a grammar without cyclic function letters. If there are cyclic function letters, the size of the nonterminals built by the parser depends on the length of the input, so the time for unifications and symbolic products is no longer independent of the input, and the parsing time is not bounded by n^3.

To save storage we use a simplified version of structure-sharing (Boyer and Moore 1972). Following the suggestion of Pereira and Warren (1983), we use structure-sharing only for dotted rules with symbols remaining after the dot. When the dot reaches the end of the right side of a rule, we translate the left side of the rule back to standard representation. This method guarantees that in each resolution only one resolvent is in structure-sharing representation. Instead of general resolution we are doing what the theorem-proving literature calls input resolution. This allows us to represent a substitution as a simple association list, using the function *assoc* to retrieve the substitutions that have been made for variables.

Pereira (1985) describes a more sophisticated version of structure-sharing. This method has two advantages over our version. First, the time to retrieve a substitution is O(*log n*), where *n* is the length of the derivation, compared to O(*n*) for Boyer-Moore. Second, only symbols that derive the empty string need to be translated from structure-sharing form to the standard representation, and this saves storage. The first advantage may not be important, for two reasons. By using a single *assoc* to retrieve a substitution, we reduce the constant factor in O(*n*). Also by eliminating the structure sharing each time the dot reaches the end of a rule, we keep our derivations short—*n* is no more than the length of the right side of the longest rule. The second advantage of Pereira's method is more important, since our current parser uses a lot of storage.

The other optimizations are fairly obvious. As usual we skip the occur check in our unifications (as long as there are no cyclic sorts, this is guaranteed to be safe). In each symbolic product, one set is indexed by the topmost function letter of the term to be matched, which saves a good number of failed unifications. These simple techniques gave us adequate performance for some time, but as the grammar grew the parser slowed down, and we decided to rewrite the program in *C*. This version, running on a Sun 4, is much more efficient. It parses a corpus of 790 sentences, with an average length of nine words, in half an hour.

Acknowledgements

I wish to thank an anonymous referee, whose careful reading and detailed comments greatly improved this paper. This work was supported by the Defense Advanced Research Projects Agency under contract numbers N00014–87-C-0085 and N00014–85-C-0079.

References

Ayuso, Damaris; Chow, Yen-lu; Haas, Andrew; Ingria, Robert; Roucos, Salim; Scha, Remko; and Stallard, David 1988 Integration of Speech and Natural Language Interim Report. Report No. 6813, BBN Laboratories Inc., Cambridge, MA.

Barton, G. Edward; Berwick, Robert C.; and Ristad, Eric S. 1987 *Computational Complexity and Natural Language*. MIT Press, Cambridge, MA.

Boyer, Robert and Moore, Jay S. 1972 The Sharing of Structure in Theorem-Proving Programs. In: Meltzer, Bernard and Michie, Donald (eds.), *Machine Intelligence 7*. John Wiley and Sons, New York, NY: 101–116.

Gallier, Jean 1986 *Logic for Computer Science*. Harper and Row, New York, NY.

Gazdar, Gerald; Klein, Ewan; Pullum, Geoffrey; and Sag, Ivan 1985 *Generalized Phrase Structure Grammar*. Harvard University Press, Cambridge, MA.

Graham, Susan L.; Harrison, Michael A.; and Ruzzo, Walter L. 1980 An Improved Context-free Recognizer. *ACM Transactions on Programming Languages and Systems* 2(3): 415–462.

Hopcroft, John E. and Ullman, Jeffrey D. 1969 *Formal Languages and Their Relation to Automata*. Addison-Wesley Publishing Company, Reading, MA.

Kasper, Robert 1987 Feature Structures: A Logical Theory with Application to Language Analysis. Ph.D. Thesis, University of Michigan, Ann Arbor, MI.

Kasper, Robert and Rounds, William 1986 A Logical Semantics for Feature Structures. In: *Proceedings of the 24th Annual Meeting of Association for Computational Linguistics*. Columbia University, New York, NY: 257–266.

Matsumoto, Yuji; Tanaka, Hozumi; Hirakawa, Hideki; Miyoshi, Hideo; and Yasukawa, Hideki 1983 BUP: A Bottom-up Parser Embedded in Prolog. *New Generation Computing*, 1(2): 145–158.

Pereira, Fernando 1985 A Structure-Sharing Representation for Unification-Based Grammar Formalisms. In: *Proceedings of the 23rd Annual Meeting of the Association for Computational Linguistics*. University of Chicago, Chicago, IL: 137–144.

Pereira, Fernando and Sheiber, Stuart 1987 *Prolog and Natural-Language Analysis*. Center for the Study of Language and Information, Stanford, CA. Distributed by Chicago University Press.

Pereira, Fernando and Warren, David H. D. 1980 Definite Clause Grammars for Natural Language Analysis—A Survey of the Formalism and a Comparison with Augmented Transition Networks. *Artificial Intelligence* 13(3): 231-278.

Robinson, John A. 1965 A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the ACM* 12(1): 23–41.

Sato, Taisuke and Tamaki, Hisao 1984 Enumeration of Success Patterns in Logic Programs. *Theoretical Computer Science* 34: 227–240.

Shieber, Stuart 1985 Evidence against the Context-Freeness of Natural Language. *Linguistics and Philosophy* 8(3): 333–343.

Shieber, Stuart 1985 Using Restriction to Extend Parsing Algorithms for Complex-Feature-Based Formalisms. In: *Proceedings of the 23rd Annual Meeting of the Association for Computational Linguistics*. University of Chicago, Chicago, IL: 145–152.