2

Parse Completion: A Study of an Inductive Domain

Technical Report PCG-11

Steve Nowlan

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, PA 15213 U.S.A

DEPARTMENT
of
PSYCHOLOGY

Carnegie-Mellon University

2

Parse Completion: A Study of an Inductive Domain

Technical Report PCG-11

Steve Nowlan

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, PA 15213 U.S.A

July 1, 1987

Running head: Parse Completion

Acknowledgments

This research was supported by the Computer Science Division, Office of the Naval Research, and DARPA under Contract No. N00014-85-C-0678. Reproduction in whole or in part is permitted for any purpose of the United States Government. Approved for public release; distribution unlimited.

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION
Unclassified

1b. RESTRICTIVE MARKINGS

2a. SECURITY CLASSIFICATION AUTHORITY

2b. DECLASSIFICATION / DOWNGRADING SCHEDULE

3. DISTRIBUTION / AVAILABILITY OF REPORT
Approved for public release;
Distribution unlimited

4. PERFORMING ORGANIZATION REPORT NUMBER(S)
PCG-11

5. MONITORING ORGANIZATION REPORT NUMBER(S)
Same as Performing Organization

6a. NAME OF PERFORMING ORGANIZATION
Carnegie-Mellon University

6b. OFFICE SYMBOL *(If applicable)*

7a. NAME OF MONITORING ORGANIZATION
Information Sciences Division
Office of Naval Research (Code 1133)

6c. ADDRESS *(City, State, and ZIP Code)*
Departments of Psychology and Computer Science
Pittsburgh, PA 15213

7b. ADDRESS *(City, State, and ZIP Code)*
800 N. Quincy St.
Arlington, VA 22217-5000

8a. NAME OF FUNDING / SPONSORING ORGANIZATION
Same as Monitoring Organization

8b. OFFICE SYMBOL *(If applicable)*

9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
N00014-86-K-0678

8c. ADDRESS *(City, State, and ZIP Code)*

10. SOURCE OF FUNDING NUMBERS p400005ub201/7-4-86

PROGRAM ELEMENT NO	PROJECT NO	TASK NO	WORK UNIT ACCESSION NO
N/A	N/A	N/A	N/A

11. TITLE *(Include Security Classification)*
Parse Completion: A study of an Inductive Domain

12. PERSONAL AUTHOR(S)
Steve Nowlan

13a. TYPE OF REPORT
Technical

13b. TIME COVERED
FROM 86Sep15 TO 91Sep15

14. DATE OF REPORT *(Year, Month, Day)*
87 July 1

15. PAGE COUNT
52

16. SUPPLEMENTARY NOTATION

17. COSATI CODES

FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
Induction, machine learning, parse completion, grammatical inference, grammar induction

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

See reverse side.

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT
☐ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT ☐ DTIC USERS

21. ABSTRACT SECURITY CLASSIFICATION

22a. NAME OF RESPONSIBLE INDIVIDUAL
Alan Meyrowitz

22b. TELEPHONE *(Include Area Code)*
(202) 696-4302

22c. OFFICE SYMBOL
1143

DD FORM 1473, 84 MAR

83 APR edition may be used until exhausted
All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE
Unclassified

Abstract

Hierarchical knowledge structures are pervasive in Artificial Intelligence, yet very little is understood about how such structures may be effectively acquired. One way to represent the hierarchical component of knowledge structures is to use grammars. The grammar framework also provides a natural way to apply failure-driven learning to guide the induction of hierarchical knowledge structures. The conjunction of hierarchical knowledge structures and failure-driven learning defines a class of algorithms, which we call *Parse Completion* algorithms. This paper presents a theoretical exploration of this class that attempts to understand what makes this induction problem difficult, and to suggest where appropriate biases might lie to limit the search without overly restricting the richness of discoverable solutions. The explorations in this paper are not intended to produce a practical induction algorithm, although fruitful paths for such development are suggested. [illegible]

1 Introduction

Hierarchical knowledge structures are pervasive in artificial intelligence systems. Classic examples include semantic networks [21], scripts [24], and frames [15] and their descendants, which are employed in current knowledge representation technology. Planning and plan recognition systems [8, 7, 23] make extensive use of procedural hierarchical structures. Despite this pervasiveness, very little is understood about how such knowledge structures may be effectively acquired.

We focus on executable hierarchies: those that represent control strategies, plans or procedures [27, 10]. Grammars can provide a uniform representation for such hierarchical control structures. The hierarchy of control is implicit in the rules of the grammar, but becomes explicit in the derivation tree[1] for a particular string.[2] In this context, one can regard the rewrite rules of a grammar as a way of transforming some goal into a group of sub-goals. Consider for example the problem of multi-column subtraction. We can regard a subtraction problem as composed of several atomic procedures, such as one column subtraction (-), shift of attention left one column (l), shift of attention right (r), decrement (d) and add (a). The last two operations are needed to represent the decrement in one column and increment in the next required of the borrow operation. In this representation the subtraction problem $25-13$ could be represented by the string $-l-$ which would be interpreted as subtracting the first column, moving left and subtracting the second column. Figure 1 illustrates one possible grammar for multi-column problems that do not involve borrowing, and the incomplete derivation tree that is produced when the procedure represented by this grammar is applied to a problem that requires borrow operations.

The idea of creating sub-goals to hierarchically solve a complex problem goes back at least to GPS [18], and is the basis for several models of cognitive architecture [14, 1, 2, 26]. Different types of grammars correspond to different classes of control structure; the activation of a sub-goal may depend only on the presence of its parent goal, or it may also depend on the concurrent

[1] The term derivation tree is synonymous with parse tree, and in this context is equivalent to a trace of the subgoals in a procedure. The derivation tree can be a general graph for context-sensitive grammars.

[2] Throughout this paper we will use the term string to represent the end product of a derivation using a grammar. The string is not necessarily a string of characters, but may equally well be a sequence of operations for performing some task.

activation of other goals with different parents (*i.e.* context free vs. context sensitive grammars). An effective algorithm for inducing grammars may also be a powerful tool for learning hierarchical control structures from experience [27].

Mitchell [16] highlights the importance of biases in induction problems. Failure (or impasse) driven learning [25, 22, 27] is a particular bias that can make many induction problems more tractable. This bias favours using the existing knowledge structures as much as possible.to solve a problem. When an impasse is reached, it adds just enough additional control knowledge to bridge the gap and complete the solution. This bias may have psychological validity as well as practical utility [28].

The combination of failure-driven learning and grammar induction yields a class of algorithms which we have referred to as *Parse Completion* algorithms. In Parse Completion one attempts to build a derivation tree for some string, using existing rules of the grammar, until no existing rules can be applied. The process may be thought of as a combination of top-down and bottom-up parsing (Fig. 1). When the derivation tree is as complete as possible, new rules are added to the grammar to *fill in* any remaining gaps in the derivation tree.

```
   S                           Grammar
 /   \
1     S                       S → -
|   / | \                     S → 1 S
|  :  :  :                    S → - S
|        :
|        S
|       /  \
|       -  S
|       | / \
| : : : | 1 S
| | | | | | |
1 d r a - 1 -
                          : - Completion Site
```

Figure 1: Incomplete derivation tree prior to Parse Completion

Parse Completion is not a new idea. Specializations of it have been used in programs that learn plans [13], procedures [27, 10], programs from examples [3, 4], and models of cognitive skills [26]. Although the concept has been used by a variety of researchers, there has been no attempt to characterize the space of parse completion algorithms, or to systematically examine where biases [16, 27] may be most effectively introduced.

2 The Parse Completion Design Space

In this section the intent is to introduce the concept of parse completion at an intuitive level and to present some of the alternative design choices for induction algorithms based on the parse completion paradigm. Results are presented here in an incomplete fashion. Later sections expand on this outline and provide the missing details.

Parse completion is a particular approach to induction problems. An induction problem is the discovery of expressions in some representation language (generalizations) such that each is (1) consistent with the examples and (2) preferred by learning biases. The set of expressions is partially ordered by a *more-specific-than*[3] predicate [17]. The induction problem is to discover some expression that encompasses all positive examples and no negative ones, by searching the tangled hierarchy of expressions.

The goal of parse completion is to build a complete derivation tree for some string starting from an existing grammar. If the existing grammar is powerful enough to parse the string then a complete derivation tree may be built. The interesting case occurs when the existing grammar is inadequate to parse the string. If we attempt a top-down parse, we will produce a partial derivation tree that contains a number of *gaps* (Figure 1). Each gap becomes a *completion site*, a point at which additional rules must be added to the grammar to complete the derivation tree. The new grammar will be a generalization of the old grammar, since it will be able to parse at least one string that the old grammar could not parse.

Although the above may appear to be a tight description of an algorithm, there are in fact a wide variety of design choices to be made within this general framework. Each choice may produce an algorithm with dramatically different characteristics. We wish to explore these design choices in some systematic fashion.

At each completion site there usually exist many different ways in which the grammar may be generalized to allow the derivation to continue. Each of these new grammars represents an

[3] More-specific-than(x,y) is true iff the *denotation* of x (*i.e.*, all possible instances of x) is a subset of the denotation of y.

alternate node within the tangled generalization hierarchy of grammars. Our first decision point is whether to consider all or just one of these alternate generalizations. This is a least-commitment versus most-commitment distinction: a most-commitment algorithm will select just one alternative, and continue its search in a depth first fashion, back tracking if necessary. Most grammar induction algorithms fall in this category [9, 5, 19]. A least-commitment algorithm attempts to explore all of the generalization alternatives in parallel, without committing itself to one particular path. In this sense it is more like a breadth first search. The best known example of a least-commitment induction algorithm is the version space algorithm [17].

Least-commitment algorithms are memory intensive compared to their most committment counterparts, and are thus regarded with disfavour for most machine learning applications. However, if the induction domain itself is ill understood, then a least-commitment algorithm can offer valuable information about the domain. If we are interested in the impact of certain design choices on an induction algorithm, then we need to know more than just the final solution obtained by an algorithm. We would also like some idea of the blind alleys explored, and those avoided. This ability to see more than just a narrow view is one advantage of a least-commitment algorithm.

There are many other design choices available. A grammar may be generalized in two different ways: by introducing new rules into the grammar, or by merging old non-terminals in existing rules. Each approach defines a partial order over the set of grammars consistent with a set of examples, and in both cases the partial order is a strict suborder of the partial order based on the predicate *more-specific-than*. The partial order defined by merging old non-terminals has been investigated elsewhere [29, 12, 20].

The parse completion algorithm provides an effective means to deal systematically with the different alternatives possible within the paradigm of generalization through the addition of rules. The approach taken is to classify rules added to a grammar in terms of the format of the right hand side (RHS). A natural classification scheme can be derived from the process of performing a top-down parse on a string. One can think of parsing a string as involving two steps. The first step partitions a string into several contiguous substrings. Each partition element is then labelled with some symbol from the grammar, either a terminal or non-terminal. The partitioning and labelling steps are repeated on each partition element labelled with a non-terminal, until all elements are labelled with terminals. At each stage the number of elements in the partition and the labels assigned to each element correspond to the RHS of some rule in the grammar. Conversely, a new rule RHS can be formed by taking a partition of a string and labelling its elements. In figure 1 the existing grammar is able to parse the first character in the string, l, and the last three characters, - l -. If we allow the non-terminal S to cover the last three characters, then the unparsed substring is d r a S. We can generate new rules to complete this parse by considering the partitions and labellings for this substring (Figure 2).

Partition	Labels	Rules
(d r a S)	-	S → d r a S
(d r a) (S)	A,-	S → A S A → d r a
(d r) (a S)	A,B	S → A B A → d r B → a S
(d) (r a) (S)	A,B,-	S → A B S A → d B → r a
(d r) (a) (S)	A,B,-	S → A B S A → d r B → a

Figure 2: Some partitions and labellings for the derivation tree in figure 1. "-" represents an unlabelled partition.

We can generate a variety of different algorithms from the parse completion framework by giving specific functions for generating partitions and labelling the elements. At one extreme, we could restrict the partitioning and labelling so that only rules already existing within our grammar could be generated. Under this restriction the parse completion algorithm becomes a simple top-down parser. With different restrictions new rules of varying power may be added to an existing grammar.

The RHS of rules may be classified according to whether they contain terminals, non-terminals that have previously appeared in the grammar, new non-terminals, and various combinations of the above. In addition, classification may be based on the length of the rules, or the order of the RHS constituents (*i.e.*, new rules may only be formed by adding to the right end of the RHS of an existing rule). Using these classification schemes, a partial order of RHS formats may be defined (see Section 4). This partial order of RHS formats is distinct from the partial order of grammars in the generalization hierarchy, although we show in Section 5 that the two are closely related. The algorithm is designed so that a particular point in this partial order may be selected, or the program may be permitted to move through the partial order itself. In this latter mode, :he most specific class of RHS format is tried first, and less specific RHS formats are tried only if the more specific ones fail to allow the parse to succeed. In this manner the program searches automatically for useful combinations of RHS formats.

Grammars may be classified by the syntactic structure of the rewrite rules that may appear in the grammar. Common classifications for grammars for regular and context free languages include:

- *Right Linear* - all productions are of the form $A \rightarrow \alpha B$ or $A \rightarrow \alpha$ where A,B are non-terminals and α a terminal string.
- *Left Linear* - all productions are of the form $A \rightarrow B\alpha$ or $A \rightarrow \alpha$ where A,B are non-terminals and α a terminal string.
- *Chomsky Normal Form* - all productions are of the form $A \rightarrow BC$ or $A \rightarrow a$, where A,B,C are non-terminals and a is a terminal.
- *Greibach Normal Form* - all productions are of the form $A \rightarrow a\beta$, where A is a non-terminal, a is a terminal, and β is a (possibly empty) string of non-terminals.

RHS format restrictions can be derived which will guarantee that all grammars generated belong to a particular class. Section 4 presents proof of these results for the classes of Right Linear and Chomsky Normal Form.

In section 4 and 5 we examine the RHS formats for Right Linear and Chomsky Normal Form grammars in detail. These two grammar classes were chosen as they can capture the classes of Regular and Context Free languages. The other grammar forms may be converted to one of these two forms by a simple mechanical transformation [11].

A simple syntactic distinction in the RHS formats was found to have a profound effect on the characteristics of the grammars generated by parse completion. For Right Linear and Chomsky Normal Form grammars the allowed RHS formats could be divided into those which introduced new non-terminals and those which reused existing non-terminals. In Section 5 it is shown that a restriction to new non-terminals can produce a grammar that has a finite language which is equal to the set of positive example strings presented to the algorithm. On the other hand, the use of existing non-terminals allows recursive rewrite rules to be introduced (i.e. a rule whose RHS may eventually be reduced to a string that contains an occurence of the non-terminals on its LHS). Recursive rewrite rules introduce the possibility of grammars which accept infinite languages. Since many interesting languages are recursive, RHS formats which allow reusing non-terminals are desirable. However, RHS formats which exclusively allow reusing old non-terminals can only use existing structure in the grammar in new combinations. Assume that we start with a situation in which we have a grammar that contains no recursive rules. In this grammar, for an arbitrary non-terminal A, there are a finite number of derivation trees which may be built with A as their root. When we create a new rule which uses this existing non-terminal A we are introducing a new situation in which the structures (i.e. trees) already associated with A can be introduced. This idea of using existing structure in novel situations is a very powerful generalization tool, but it cannot work in a vaccuum. There must be some initial structures to be manipulated, and these can only be introduced by the use of new non-terminals. So both types of RHS formats are necessary and the interesting question is whether we can always tell how much of each is required.

The RHS formats for Right or Left Linear grammars and Chomsky Normal Form grammars can both be shown to define a partial order over the set of grammars consistent with the examples. (Note that this is a partial order over the grammars themselves, not over the RHS formats.) Furthermore, these partial orders are well defined and finitely bounded,[4] and can therefore be used to define a version space-like structure for these grammar classes (see Section 5). This structure is useful in deriving a number of properties about induction algorithms for these grammar classes.

One important result that can be derived is that under certain conditions, and given only a set of positive example strings, a least-commitment induction algorithm will always converge to a grammar set containing at least one grammar for the target language in bounded time (for proofs see Section 5). The key idea is that it is not possible to arbitrarily introduce RHS formats which add structure and RHS formats which reuse existing structure. Unless a certain minimal amount of structure is present first, the grammars produced will be overly general and the partial order of grammars induced will fail to contain a grammar which captures only the target language.

The important question is how much structure is necessary to prevent over-generalization. One can show that if the examples are ordered so that the shortest ones come first, and if the learner

[4] The bounds however are quite large, even for simple grammars, hence computation of the entire partial order is often not practical.

starts out by adding only rules that introduce structure, then there is a well defined point at which no further structure need be added, and one can begin to introduce rules which use existing structure recursively.[5] Identifying the point at which no further structure need be added is equivalent to definin̦ a space complexity bound for the language being induced.

The derivation of the above results also suggests that certain *felicity conditions* [26] can be defined which will permit convergence. These conditions require the teacher to identify to the student whether a particular example is an example of a new concept in the domain, or merely a generalization of concepts the student has seen before (for details see Section 6).

3 The Parse Completion Algorithm

It is perhaps easiest to get an intuitive feel for the parse completion algorithm by considering a simple example. A simple grammar is defined in figure 3, part **a**, along with a new example string which is a member of the language. The first step in the algorithm is to attempt to parse the string from the top down. Applying the first rule in the grammar we get the partial derivation tree shown in part **b** of figure 3. It should be obvious to the reader at this point that the existing grammar cannot successfully complete this parse. We can extend the partial derivation tree by applying the second rule of the grammar to the non-terminal A, leaving the non-terminal **B** to *cover* the substring **abb**. When we consider a string to be parsed it is convenient to number the positions between each pair of terminals in the string, and before and after the string. Thus the substring **ab** in our new example string (figure 3 a) is found between positions 1 and 3.[6] A node in the derivation tree is said to *cover* a particular substring if the left-most leaf of the sub-tree rooted at that node is the first element of the substring and the right-most leaf of the sub-tree is the last element of the substring. Thus in figure 3 **f** the node labelled with the non-terminal **B** in the left tree covers the substring in positions 1 through 4 (i.e. **abb**).[7]

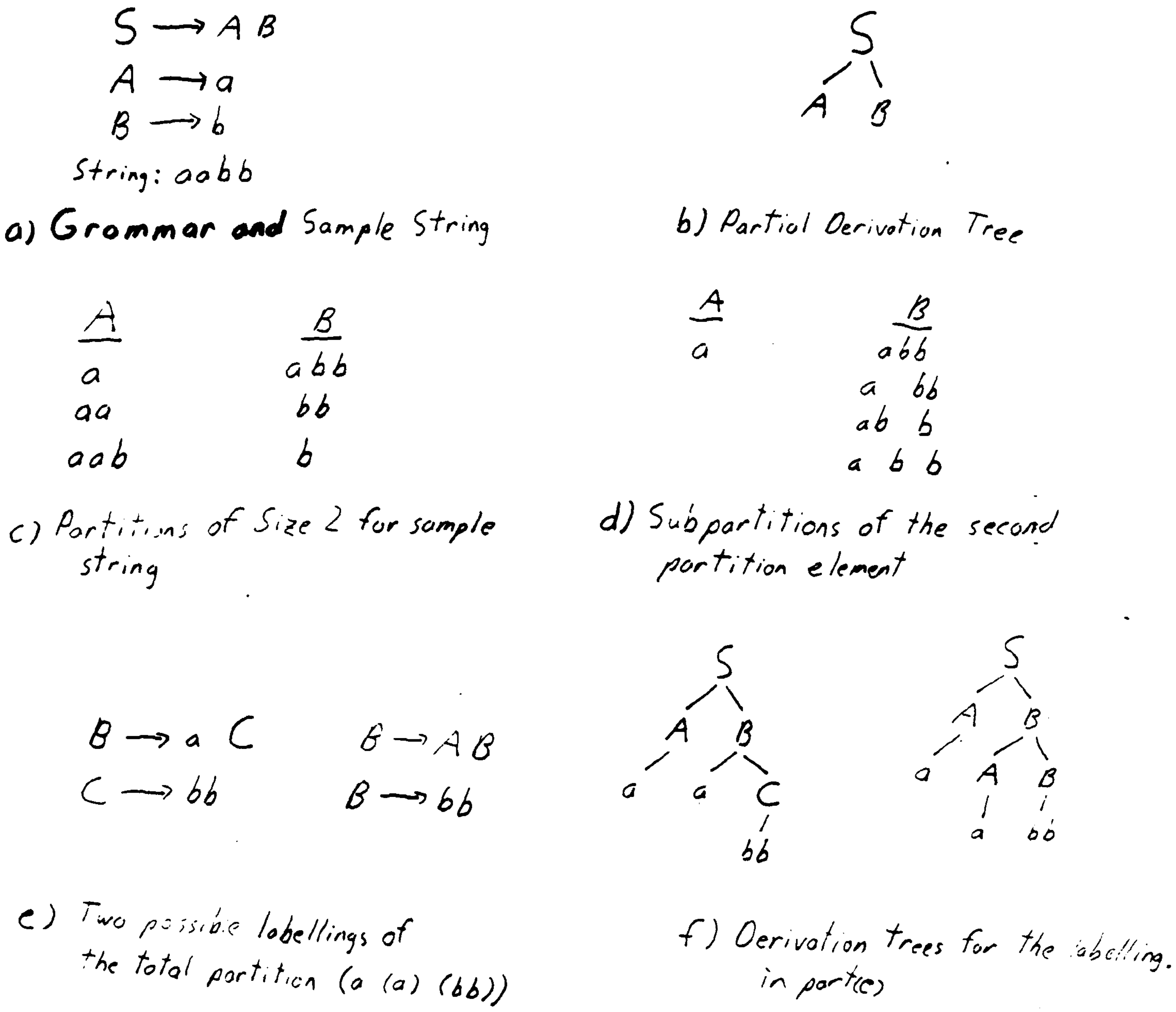
As we have noted it is possible to extend the derivation tree in figure 3 **b** by applying the

[5] These results are based on the Pumping Lemmas for regular and context free languages and are discussed in Section 5.

[6] The numbering of positions is from left to right, starting from 0 for the start of the string.

[7] Note that normally a given sequence of terminals may appear several times within a string, hence to avoid ambiguity we will usually refer to substrings via the position numbers. Similarly, when the labels on a parse tree are not unique, we will number the nodes in the tree in breadth first fashion starting at the root, and refer to the node by number, rather than by its label. The node labelled **B** in figure 3 could also be referred to as node 3.

Figure 3: A simple parse completion example.



a) Grammar and Sample String

b) Partial Derivation Tree

c) Partitions of Size 2 for sample string

d) Subpartitions of the second partition element

e) Two possible labellings of the total partition (a (a) (bb))

f) Derivation trees for the labelling in part(e) second rule in the grammar, but if we do this then we are restricting our derivation by forcing the third node in our derivation tree (the one labelled **B**) to cover the rest of the substring. It is not clear that we wish to introduce this bias into our algorithm. If the language we are attempting to describe can be defined by the regular expression $\mathbf{a}^{+}\mathbf{b}^{+}$, then one possible grammar for the language would be:

$S \rightarrow AB$
$A \rightarrow aA$
$A \rightarrow a$
$B \rightarrow bB$
$B \rightarrow b$

However, if we used the bias suggested then we could never discover this language. So it is not always best to attempt to extend the parse as far as possible before adding new rules to the grammar. This particular bias may also be undesirable because its effects are subtle, hence hard to specify in a non-procedural fashion.

One important issue to be resolved is how far one should attempt to push the parse with the existing grammar before considering additional rules. For the purposes of this example, assume that we stop with the partial derivation tree of figure 3 **b**. We have two non-leaf nodes beneath which we wish to build sub-trees to complete the parse. There will be one leaf node for each terminal of our string, and the problem is to decide how to allocate these leaves to the two sub-trees. We can regard this as a partitioning problem; in general we will wish to partition some string between positions k and l into m non-overlapping substrings such that the substrings when concatenated in left to right order form the original string and each substring is of length at least one. Each substring in a partition is called an *element* of the partition. The *size* of a partition is the number of elements in the partition. The *length* of a *partition element* is the length of the corresponding substring. For the example problem the possible partitions of size two are shown in figure 3 **c**. Each partition element, if it is of length greater than one, may be partitioned further. Consider the first of the three partitions in figure 3 **c**; the substring **a** will be covered by the node labelled **A** and the substring **abb** will be covered by the node labelled **B**. The first element of this partition cannot be partitioned further, but the second may be left as a single element of length three, or may be partitioned further into partitions of size two or three. (See figure 3 **d**.) In general each element of a partition may be partitioned further, and each partition for one element may be combined with any partition for the next element in forming a valid *total partition*. A total partition is a sequence of nested partitions: **(a (a) (bb))** and **(a (a) (b) (b))** are both total partitions for this example. Each total partition corresponds to a different topology for the sub-trees used to complete a parse.

For a fixed total partition, there are still a variety of sub-trees which represent different completions of the parse. Each of these is distinguished by a different set of labels for the interior nodes of the sub-tree.[8] Each different labelling of the set of children of some node in the derivation tree corresponds to a different right hand side (RHS) for a rule whose left hand side corresponds to the label of the parent node. The *length* of the RHS is defined to be the length of the corresponding partition. If we consider the total partition **(a (a) (bb))**, the new rules added by two possible labellings of this total partition are shown in figure 3 **e** and the corresponding derivation trees for the completed parses are shown in part **f** of the figure. These are only two of the many possible trees derivable by considering all possible partitions and all possible labellings of those partitions.

The careful reader will have noted that we introduced two restrictions into the types of grammars we will consider in the preceding example. The restriction that all partition elements had to be of length at least one means that we will not allow ε-rules in our grammars.[9] This restriction is of little consequence since it can be proven [11] that for any grammar containing ε-rules there is an equivalent grammar without any ε-rules.[10] The more significant restriction is that we assumed that the LHS of any rule in the grammar was simply the label on the parent node in the derivation tree. This assumption means that we are restricting ourselves to the class of Context Free grammars.[11] Although this class does not include all computable functions, it still contains a large and interesting class of algorithms, including those which can be computed with a simple stack.

The basic parse completion algorithm is presented in Figure 4. There are two steps at each

[8] The leaf nodes will always be labelled identically, since they correspond to the same substring in every case.

[9] An ε-rule is simply a rule whose RHS is empty.

[10] This is true if and only if the language defined by the grammar does not contain the empty string, an assumption we shall make henceforth.

[11] A Context Free grammar is one in which the RHS of rules may be any combination of terminals and non-terminals, but the LHS of a rule is restricted to being a single non-terminal. Procedurally, this restriction is equivalent to the invocation of a sub-goal being dependent only on the presence of its parent goal, and not on the presence of siblings of its parent.

stage of the parse: partitioning and substitution. Partitioning has already been discussed; substitution involves labelling each element of a partition with a non-terminal or a terminal string. A labelled partition corresponds to the RHS of a rule whose LHS will be the argument LHS passed in to the procedure. If a rule matching this LHS and RHS already exists in the grammar the grammar is unchanged, otherwise a new rule is added. A single old grammar can serve as parent to several new grammars, since a particular substring may be partitioned and labelled in several ways, each distinct way representing an *alternate* rule which may be added to the old grammar. The algorithm is then applied recursively to each new partition until all partitions are labelled with terminals, at which point we have a complete top-down derivation of the string. The algorithm is called initially with the start symbol, S, and with the left and right pointers set to the beginning and end of the string to be parsed.

```
parse-complete(left right LHS old-grammars)

  for each grammar in old-grammars do
   if LHS is a terminal symbol then
      if the terminal symbol matches the string between left and right
         parse succeeds and return old-grammar
      else
         parse fails and return fail(LHS left right)
   else
     if the string between left and right has length 1
        add a rule of form LHS --> terminal to grammar
        if necessary and return modified grammar
     else
       for all partitions of the string between left and right do
         for all substitutions for a partition do
           if the LHS, RHS pair are not already in the grammar add
             a rule of form LHS --> RHS to grammar to form mod-grammar
           for each partition element (left right) and element label
               parse-complete(left right label mod-grammar)
               if no successful parses were found, create a fail
                marker fail(LHS left right) and place it on list of
                new grammars
               else
                 add list of grammars returned to new grammars.
        Return list of new grammars.
```

Figure 4: Basic Parse Completion Algorithm

Several comments may be made about the basic algorithm. The test for partition size of one is a check for the case when a node in the derivation tree has only one child. In this case we force the child to be a terminal string[12], and hence a leaf node in the derivation tree. This restriction

[12] Which may be a string of length one.

eliminates the infinite class of redundant extensions to a grammar which have rules of the form $A_i \rightarrow A_{i+1}$, where A_i, A_{i+1} are arbitrary non-terminal symbols. The elimination of this class of grammars does not give up any representational power as any grammar that contains rules of this form can be reduced to a grammar that accepts the same language but contains no rules of this form [11]. The process of combining new rules and old grammars to create new grammars must also be treated with care. In general, there may be several ways to complete the parse for *each* element of a partition. Each completion for a particular element will have some set of new rules associated with it[13] and the set of new rules introduced by a particular parse is formed by unioning one of these sets from each partition element with any of the sets from the other partition elen:ents.

Failure of a parse can occur in two ways; either a mismatch occurs between a terminal introduced as a label in the derivation and the terminal in the corresponding position in the string, or the set of substitutions at some point in the parse is empty. When a failure occurs, a *fail marker* is generated which indicates the label of the node where the failure occurred, and the substring to be spanned by the node. These fail markers allow the algorithm's efficiency to be improved considerably, since if a parse fails to succeed, it is only necessary to reparse with a different class of partitions or substitutions from the fail markers, rather than restarting the parse from the top of the derivation tree.

There is one potential difficulty with the fail markers. Consider the two grammars in figure 5, both of which parse the single string **ba**. Assume that we have now given a new string **bb** to the algorithm. For the grammar in part **a** of the figure the fail marker generated would be **(1 2 B)**. Reparsing from this point and allowing new terminals for rule RHS would add the rule $B \rightarrow b$ to the grammar. However for the grammar in part **b** the fail marker created is **(1 2 a)**. We do not allow rules of the form $a \rightarrow b$ in our grammars, so any attempt to reparse from this fail marker is doomed to fail. In this case the rule we wish to modify is actually the parent of the node at which the failure occurred, so it is necessary to *promote* the fail marker up to this parent rule. (i.e. The desired fail marker is (**0 2** S)). When promoting fail markers in this way, one must be careful to remove any fail-markers associated with other children of the node the failure was promoted to.

The basic algorithm described thus far may be instantiated to a particular algorithm by

[13] Which may be empty.

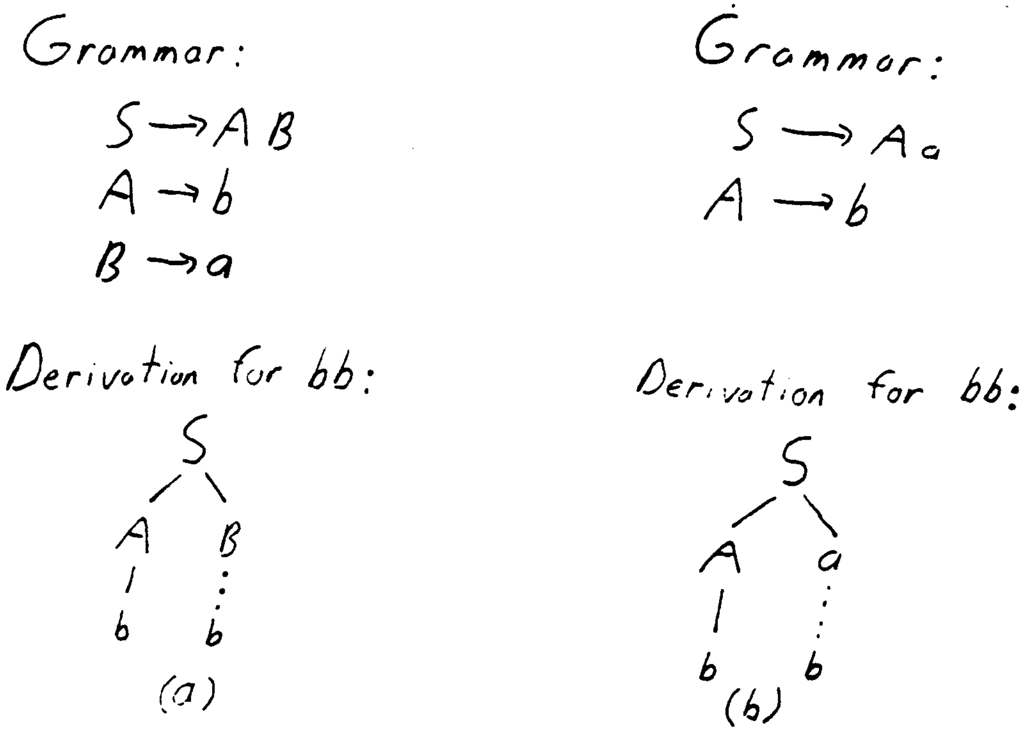


Figure 5: Two grammars for the string **ba**, and partial derivations for **bb**.

specifying functions for generating partitions and substitutions. For example, we may restrict the partitions and substitutions so that only partition and substitution pairs which correspond to existing grammar rules are generated. If our grammar contained just the rules:

$S \rightarrow a S$
$S \rightarrow a$

then we would only generate partitions of size one with label a assigned to the single partition element or partitions of size two, with the first element labelled **a** and the second element labelled S. With this pair of generators specified for the partitions and substitutions the parse completion algorithm becomes a simple top-down parser.

In section 4 a partial order of substitutions for the RHS of a rule will be described. It is possible for the parse completion algorithm to pick a particular point in this partial order and hold it fixed throughout a learning trial.[14] More interesting behaviour is generated however if the algorithm is allowed to move through this partial order on each example string. Initially, the most specific class of substitutions is tried and more general substitutions are used only if the

[14] A learning trial is defined as a set of positive examples and a (possibly empty) set of negative examples drawn from the language of a particular grammar.

more specific ones fail to allow a parse to succeed. If this second approach is taken, there is still a control issue to be resolved: One may either move through the partial order each time a failure in the parse occurs, or one can fix a point in the partial order at the start of the parse, and only move up a level in the partial order if all attempts at completing the parse at the current level fail. Fixing the substitution class once at the start of the parse would correspond to the algorithm illustrated in figure 6, while moving through the partial order at each failure would require a modification to the basic parse completion algorithm.(See figure 7.)

```
subst_level = 0
while no successful parse do
  parse-complete(start end S empty)
  subst_level = subst_level + 1
```

Figure 6: Algorithm for fixing substitution level at start of parse.

```
parse-complete(left right LHS old-grammars)

  for each grammar in old-grammars do
   if LHS is a terminal symbol then
      if the terminal symbol matches the string between left and right
         parse succeeds and return old-grammar
      else
         parse fails and return fail(LHS left right)
   else
     if the string between left and right has length 1
        add a rule of form LHS --> terminal to grammar
        if necessary and return modified grammar
     else
       for all partitions of the string between left and right do
         for all substitutions for a partition do
           if the LHS, RHS pair are not already in the grammar add
             a rule of form LHS --> RHS to grammar to form mod-grammar
           for each partition element (left right) and element label
               subst_level = 0
               while no successful parse and subst_level <= max do
                     parse-complete(left right label mod-grammar)
               if no successful parses were found create a fail
                marker fail(LHS left right) and place it on list of
                new grammars
               else
                 add list of grammars returned to new grammars.
        Return list of new grammars.
```

Figure 7: Parse Completion with movement through the substitution levels at each parse failure.

Both control strategies were tried. The approach in which one moved through the partial order each time the parse reached a failure point produces new grammars from old through hybrid substitutions which span multiple levels of the partial order. This makes it difficult to determine the characteristics of the grammars produced under a particular substitution strategy. For purposes of examining the properties of grammars under different classes of substitution, the approach in which an entire parse is attempted at one level before the next level of substitution is considered is preferred.

Before considering the classes of substitutions and partitions in more detail, we shall conclude this section with some comments on the complexity of this algorithm. The basic processes of partitioning and labelling in the algorithm can be equivalently regarded as constructing a rooted tree (*i.e.* a tree in which one node is distinguished as the root), and then labelling this tree according to the restrictions imposed by the current point in the substitution hierarchy. One can measure the complexity of the algorithm in terms of the number of possible trees that can be generated.

The trees we are interested in are rooted, and have *n* ordered leaves. In general when we partition, each node is allowed to have anywhere from two to *n* children. However we will first consider the simpler problem of the number of ordered binary trees with *n* leaves. We may assign the first *r* leaves to the left sub-tree and the remaining *(n - r)* to the right sub-tree of the root. If we let a_k be the number of rooted ordered binary trees with *k* leaves, then there are a_r distinct left sub-trees and a_{n-r} distinct right sub-trees when we assign *r* leaves to the left sub-tree. Thus the total number of distinct trees with *r* leaves in the left sub-tree is $a_r a_{n-r}$. Since we may assign anywhere from one to *n - 1* leaves to the left sub-tree we have the recurrence formula:

$$a_n = \sum_{k=1}^{n-1} a_k a_{n-k}$$

for the number of rooted ordered binary trees with *n* leaves. This recurrence formula corresponds to the Catalan series [6] and it can be shown that the number of rooted binary trees with n leaves is the *n-1st* Catalan number which is defined by $C_{n-1}=\frac{1}{n}\binom{2n-2}{n-1}$. It can be shown that $\binom{2k}{k}$ is bounded above by 2^{2k}.[15] Thus an upper bound on the number of rooted ordered binary trees with *n* leaves is $O(\frac{4^n}{n})$.

In the more general case, our trees are still rooted and ordered, but a node may have two or

[15] Intuitively this is obvious as 2^{2k} is the total number of subsets of *2k* items while $\binom{2k}{k}$ is the number of subsets containing exactly k items.

more children. The analysis in this case is greatly simplified by the fact that the rooted ordered trees with *n* vertices may be put in one to one correspondence with the number of rooted ordered binary trees with *n - 1* leaves. [6] From our previous results we can see that the number of rooted ordered trees on *n* vertices is the *n - 2nd* Catalan number. We are interested in the number of trees with *n* leaves rather than *n* vertices, but since every node but a leaf must have at least two children, with *n* leaves, every tree has at least *n + 1* vertices and no more than *2n - 1* vertices. We may simply sum over the number of trees for each number of vertices:

$$\sum_{k=n+1}^{2n-1} C_{k-2} = \sum_{k=n+1}^{2n-1} \frac{1}{k-1}\binom{2k-4}{k-2} \leq \frac{1}{2}\binom{4n-6}{2n-3}$$

Using the upper bound for $\binom{2k}{k}$ from before we have that the number of rooted ordered trees with *n* leaves is $O(16^n)$.

In most cases the more general partitioning algorithm is applied, but for certain cases (see section 4) we consider the more restrictive binary partitioning for the string. The important point is that the complexity bound in both cases is exponential. An exhaustive examination of all the possible structures is clearly infeasible for practical problems. However before one can understand the effects of various heuristics, one needs a map of the space of possible structures. The purpose of this algorithm was to provide a tool to help sketch out this space, and the rest of this paper is devoted to a description of some of the characteristics of this space that have been discovered.

4 A Space of RHS-formats

We can now begin to examine the types of rules that may be added to a grammar through the process of parse completion. The basic manner in which a new rule is formed is to first partition some substring of the current input. The length of this substring determines the length of the RHS of this new rule. However the composition of the RHS, and hence to a large extent the properties of the resulting grammar, is dependent on what sorts of labels are allowed for the RHS of the new rule. To give a trivial example, if we were to restrict our rules to allow only terminals to appear as partition labels then it is apparent that for any positive set of sample strings we would generate the trivial grammar that generates exactly that set of sample strings and no other strings. At the other extreme, if we restrict the RHS of new rules to be labelled only by the non-terminal S or a terminal, then, if Σ is the alphabet used in our sample strings, we will generate a grammar for Σ^* (*i.e.* the language of all possible finite length strings over Σ).

If we extend the arguments in the previous paragraph, we find that we can define a partial order over the RHS formats. This partial order is based on the generality of the grammars that can be produced by allowing only this type of RHS format to be used when performing parse completion. It is convenient to first characterize the RHS formats along two dimensions. One dimension of variation is the composition of the RHS, what sort of terms we allow to appear on a RHS. The three natural compositional categories are *terminals*, *new non-terminals* and *old non-terminals*. New non-terminals are simply those which have not appeared in any previous rule in the grammar, while old non-terminals have appeared previously. The second dimension of variation we have considered is the dimension of order. For example the class of regular grammars can be captured by left or right linear grammars, which have the restriction that all non-terminals either precede or follow all terminals in each RHS. Similarly center-embedded grammars can be characterized by imposing a restriction on the order of terminals and non-terminals in rule RHS's.

It is difficult to capture all of the order variation that is possible, so we have simplified the variability along this dimension by grouping the order restrictions into three broad classes. Admittedly these classes are somewhat arbitrary, but as a first pass they do capture some important distinctions. The three categories selected are *existing order*, *extension*, and *unrestricted*. Existing order limits RHS formats to those already existing in the grammar. Extension permits adding new components to the right of existing RHS formats only. This restriction allows one to capture the class of right linear grammars. Unrestricted allows the addition of new components to either end of an existing RHS format as well as arbitrary replacement of existing components. The three order restrictions may be applied to each type of RHS constituent independantly producing the two dimensional matrix of *RHS restrictions* shown in figure 8. For convenience, each cell in this matrix has been numbered and these numbers will be used to refer to the particular combination of constituent and order restriction represented by each cell. Note that the combination of new variables and existing order is not a legal combination since by definition a new variable cannot have a previously defined position in any rule.

A RHS format is defined as an ordered triple of restrictions, $<\alpha,\beta,\delta>$. The first element of the triple is the restriction that applies to terminal constituents in the RHS, the second element refers

	Terminals	New Non-terminals	Old Non-terminals
Existing Order	1	☒	2
Extension	3	4	5
Unrestricted	6	7	8

Figure 8: Matrix of RHS restrictions

to new non-terminal constituents, and the third to old non-terminal constituents. Each element is one of $\{\varnothing,\exists,E,U\}$, where $\varnothing$ means no constituents of this type are allowed, $\exists$ means existing order, and E and U refer to extension and unrestricted respectively. The format is the union of the sets represented by the three elements in the triple. The triple $<E,\exists,E>$ corresponds to the set of all RHS formats which can be formed by taking old non-terminals in their existing order and allowing extension with terminals or new non-terminals.

There is a total order over the restrictions. The set of RHS formats without a particular constituent is a strict subset of the set of RHS formats with that constituent in its existing order. The set of RHS formats with a particular constituent only in its existing order is a strict subset of the set of RHS formats which allow that constituent in its existing order and also as extensions to an existing format. Similarly, the set of RHS formats which allow extension with a particular constituent are a strict subset of the set of RHS formats which allow unrestricted use of that constituent. However, restrictions applied to distinct constituents (*i.e.* terminals and new non-terminals) are not directly comparable, meaning that we cannot define a total order over the RHS formats. We can define a partial order over the RHS formats:

$$
\begin{aligned}
&<a,b,c> \; > \; <a',b',c'> \textit{ iff } a \geq a' \;\&\; b \geq b' \;\&\; c \geq c' \\
&\qquad \textit{where } a,b,c,a',b',c' \in \{\varnothing,\exists,E,U\} \\
&\qquad \textit{and } U > E > \exists > \varnothing.
\end{aligned}
$$

This partial order has a unique upper and lower bound. The lower bound is defined by the triple $<\exists,\varnothing,\exists>$. This format allows only terminals and old variables in the same order as an existing rule in the grammar, so the lower bound corresponds to parsing a string with the existing grammar. The upper bound is defined by $<U, U, U>$ and allows unrestricted use of all three basic constituents (terminals, old and new non-terminals). This format contains the set of all RHS formats that contain terminals and non-terminals from the existing grammar plus up to n new non-terminals where n is the length of the partition. It is easy to show that this is the most general set of RHS formats allowed under the parse completion paradigm.

The partial order of RHS formats is related to the partial order of grammars developed in Section 5. The relationship arises because in parse completion the only mechanism to generalize a grammar (*i.e.* increase the set of strings accepted by the grammar) is to add additional rules to the grammar. One implication of this is that in parse completion the grammars always increase monotonically in size. Adding additional rules to a grammar may make a grammar more general than it was; however, the addition of rules to a grammar can never make a grammar less general than it currently is. Thus, once our induction process over-generalizes in this domain, we are stuck.[16] The second implication is that how much more general a grammar becomes when one additional rule is added is a function of the power of that rule. If the RHS format allows unrestricted use of old non-terminals then it becomes possible to create recursive rewrite rules and convert a grammar that accepts only a finite set of strings into one which accepts an infinite set of strings. On the other hand a RHS format which allows only the use of terminals or new non-terminals cannot convert a finite grammar into an infinite one. In general, if we consider a grammar G and some string s which cannot be parsed by G and two different RHS formats a and a', then if $a' > a$ and we let S be the set of candidate rules for completing the parse allowed by a, and S' the set of candidate rules allowed by a' then S is a subset of S', and there will be rules in S' that when added to G form a new grammar more general than any grammar that could be formed by adding rules from S to G. In this fashion the partial order of RHS formats determines how large a "step" we take in generalizing the grammar by adding one rule to it.

To illustrate some of the ideas discussed above we will now work through a few simple examples. Consider first the case of a simple regular language $(0+1)^+$. Assume the system has already been trained on some example strings and has generated the following initial grammar:

[16] This, of course, is true only if there is not some external backtracking mechanism capable of retracting a hypothesized grammar and returning the system to some previous state.

$S \rightarrow 0$
$S \rightarrow 0S$

This is a right linear grammar for 0^+. Now if we are given a new string 01 the partial derivation tree generated for this string will be as shown in figure 9. The parse will fail at the leaf labelled S and the fail marker returned will be (S 1 2). Now, since this is a partition of size 1, we will only consider the use of terminals for the RHS constituent. Also, since no existing rule begins with the string 1, we cannot extend an existing rule, hence our RHS format is an unrestricted terminal (*i.e.* $<U,\varnothing,\varnothing>$). This RHS format leads to the introduction of a new rule $S \rightarrow 1$ and our new grammar is:

$S \rightarrow 0$
$S \rightarrow 1$
$S \rightarrow 0S$

This grammar is still not general enough (it corresponds to the regular expression $0^*(0+1)$). Now consider adding another example string, 011. The partial derivation tree generated will be the same as that illustrated in figure 9, but in this case our fail marker will be (S 1 3), corresponding to the substring 11. In this case a variety of RHS format restrictions may be applied and it is instructive to consider the outcome under different RHS formats.

1. The RHS format $<U,\varnothing,\varnothing>$ which allows unrestricted terminals only.

   In this case the rule $S \rightarrow 11$ is added to the grammar and we still have a grammar which is not general enough.

2. The RHS format $<\varnothing,U,\varnothing>$ which allows unrestricted old non-terminals only.

   In this case the rule $S \rightarrow SS$ is added to the grammar, and the derivation is completed using existing rules in the grammar. This grammar is in fact a grammar for $(0+1)^+$. It should be noted however that this grammar is not right linear, and hence is actually more powerful than strictly necessary to capture this language.

3. The RHS format $<\varnothing,E,\varnothing>$ which allows extension with old non-terminals only.

   In this case we can start with the rule $S \rightarrow 1$ and extend it to yield the rule $S \rightarrow 1S$. The parse may be completed after the application of this rule by using rules already in the grammar. The new grammar produced is a right linear grammar for the regular language $(0+1)^+$, and is thus the most desirable grammar for this particular language.

The intent of this simple example was to illustrate how the choice of RHS format can affect the structure of the induced grammar (the second RHS format above does not preserve the right linearity of the grammar), and how well the induced grammar generalizes the given example strings.

Two interesting and well studied classes of language are regular languages and context free

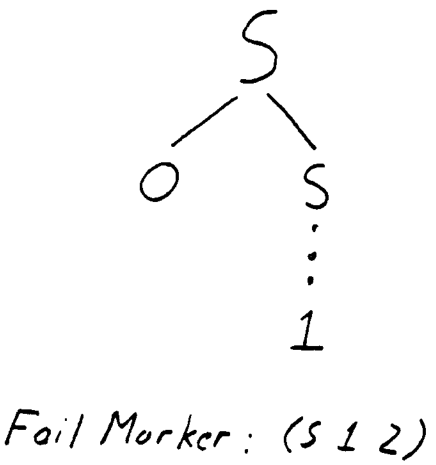


Figure 9: Partial derivation tree for the string 01

languages. Regular languages can be shown to be captured precisely by right linear grammars, while context free languages can be shown to be captured by Chomsky Normal Form (CNF) grammars [11]. The following two theorems show that these two grammar classes can be captured by an appropriate restriction of partitioning and RHS formats.

Theorem 1: Given a right linear grammar G, if we apply parse completion to it with the following restrictions, then the resulting grammar G' will always be right linear. The restrictions are that only partitions of size one or size two are allowed and that only the following two RHS formats are used in the indicated order[17]:

1. Extension with old or new non-terminals.$<\exists, E, E>$
2. Unrestricted terminals.$<U, \varnothing, \varnothing>$

Proof: Assume that the parse of an existing string fails, and we are left with a fail marker (N i j) where N is a non-terminal and i and j denote the start and end of the unparsed substring. There are two possible cases depending on whether we partition this substring into one piece or two pieces:

1. A partition of size one. Since all rules in G already have RHS of length at least one; we cannot extend an old rule to match a partition of size one. Thus we fall through to our second RHS format, which only permits unrestricted terminals. Our new RHS will be the entire unmatched substring, forming a new rule $N \rightarrow \beta$, where β is

[17] Recall that the general parse completion algorithm assumes that the set of RHS formats it uses is ordered, and will attempt to complete the parse with one RHS format before considering the next format in the order.

a string of terminals. Hence the new rule is a valid right linear rule.

2. A partition of size two. The second RHS format would allow us to substitute terminal strings for both parts of the partition, producing a rule of the form $N \rightarrow \beta_1\beta_2$, which is a valid right linear rule of the form $N \rightarrow \alpha$ where $\alpha = \beta_1\beta_2$, and is thus the same as case 1. The other choice is to extend an existing rule using the first RHS format. The only candidates for extension are rules with RHS length less than two. (Recall that we count a string of terminals not seperated by any non-terminals as one element when computing RHS length.) Since G is right linear, these rules must all be of the form $A \rightarrow \alpha$ where α is a terminal string. Extending a rule of this form with either an old or a new non-terminal produces a rule of the form $A \rightarrow \alpha B$, where α is a terminal string and B a non-terminal. This rule is a legal right linear rule.

In both cases the new rules added to the grammar will preserve the right linearity of the grammar. This completes the proof.

Theorem 2: Given a CNF grammar G, if we apply parse completion to it with the following restrictions, then the resulting grammar G' will always be CNF. The restrictions are that only partitions of size two are allowed except when the substring has length one, and that only the following two RHS formats are used in the indicated order:

1. Unrestricted old and new non-terminals.$<\varnothing, U, U>$
2. Substitution of *terminals only at leaves of derivation tree (i.e. at partitions of size* one).

The second RHS format specified is really just unrestricted substitution of terminals for partitions of size one (*i.e.* $<U,\varnothing,\varnothing>$).

Proof: Assume that the parse of an existing string fails, and we are left with a fail marker (N i j) where N is a non-terminal and i and j denote the start and end of the unparsed substring. Once again we must consider two cases, depending on the length of the substring.

1. The length of the substring is one. In this case our partition must be of size one, and we substitute the corresponding terminal in the substring for our RHS format yielding a rule of the form $N \rightarrow a$, where a is a terminal. This new rule is a valid CNF rule.

2. The length of the substring is greater than one. In this case we consider all possible partitions of size two. For each such partition we only allow the first RHS format which will produce a rule of the form $A \rightarrow BC$ where B and C are both non-terminals. This new rule will also be a valid CNF rule.

Thus at each point where we are unable to complete the parse we add a rule according to case 1 or case 2. In both cases the rule added will preserve the CNF. Finally, this process will always terminate since the substring corresponding to each element of the new partition is smaller than the original unparsed substring, and once we reach a substring of length one we must stop. This completes the proof.

There is one other common form of grammar which also encompasses the class of context free grammars, this is Greibach Normal Form. In a manner analogous to the above one can show that you will generate only Greibach Normal Form grammars if you restrict the parse completion algorithm in the following manner:

1. Allow only partitions where the length of the first element of the partition is one.
2. Use a RHS format which allows extension with new or old non-terminals for partitions of size greater than one.
3. Use a RHS format which allows unrestricted substitution of terminals for partitions of size one.

5 A Partial Order for Grammars

While developing the RHS formats for right linear grammars described in the previous section, biases which favoured using either old or new non-terminals first were also tried. Both of these biases turn out to be undesirable, but for different reasons. The bias in favour of new non-terminals will always produce a grammar for a finite language, since the grammar will never contain recursive rewrite rules. (A *rewrite rule is recursive* if the same non-terminal occurs in the LHS and RHS, or if a non-terminal in the RHS may eventually be rewritten as a string which contains the non-terminal on the LHS.) On the other hand, a bias in favour of old non-terminals will always produce a grammar for an infinite language, but the grammar rules will always contain only a single non-terminal. It is easy to show that any grammar of this form corresponds to a regular language[18] of the form $(\alpha_1+\alpha_2+ \ldots +\alpha_i)^*(\beta_1+\beta_2+ \ldots +\beta_j)$ or of the form $(\alpha_1+\alpha_2+ \ldots +\alpha_i)(\beta_1+\beta_2+ \ldots +\beta_j)^*$ where α_i and β_j are strings of terminals. The problem is that this language is usually much more general than the target language of the induction. The reason for this overgeneralization is that recursive rewrite rules were introduced into the grammar before the grammar contained sufficient structure to adequately capture the target language.

The effects of the bias in favour of either new non-terminals before old non-terminals or vice versa reveals a partial order of the grammars induced by parse completion for both the cases of

[18] The grammar is not necessarily Right Linear, it could be CNF or Greibach or several other forms. This is no paradox, the regular languages are a proper subset of the context free languages, so a CNF grammar could quite easily correspond to a regular language.

right linear grammars and CNF grammars.

Consider first the case of right linear grammars. The following theorem is used to show that a bias in favour of old non-terminals will always produce a grammar for the language Σ^+, where Σ is the alphabet of the sample strings, once a sufficient number of sample strings are given. This language is almost always more general than the target language, so the algorithm is always over-generalizing.

Definition: Let S be a set of strings. *Post(S)*, the set of postfix strings on S, is defined to be $\{ \beta \mid \exists\alpha\ \alpha\beta \in P \text{ and } length(\alpha) \geq 0 \}$. Note that S is a subset of Post(S).

Theorem: Given a regular language, L, a set of positive examples, P, and a known alphabet, Σ, if we have a bias in favour of old non-terminals as RHS constituents, then the RHS formats allowed for right linear grammars will produce only grammars of the form:

$$S \rightarrow \alpha_i S \quad \alpha_i \in Post(P)$$
$$S \rightarrow \beta_i \quad \beta_i \in Post(P)$$

Proof: This is easily proven by induction on the number of rules in the grammar.

Base Case: The first rule added to the grammar must be of the form $S \rightarrow \alpha$ *where* α *is the first* example string, hence α is an element of Post(P).

Inductive Case: Assume that all of the first *n - 1* rules added to the grammar are of the form indicated, and now consider the addition of the *n*th rule to the grammar. This rule will be introduced at a point where the parse of the string with the existing grammar failed. The fail marker returned (after promotion) must be of the from (S i j) as S is the only non-terminal currently in the grammar, thus S will be the LHS of the new rule. Now since our grammar is right linear any derivation can be organized so it is a leftmost derivation, hence our unparsed substring must extend from the point at which the parse failed to the end of the string. There are two cases to consider:

1. The unparsed substring contains no prefix that matches the RHS of an existing rule. In this case the second RHS format for right linear grammars must be applied and a new rule of the form $S \rightarrow \alpha$ will be created where α equals the unparsed substring. Since this substring is a postfix of the string currently being parsed, the new rule is of the correct form.

2. The unparsed substring contains some prefix that matches the RHS of an existing rule. Let β be the RHS of the existing rule, by the induction hypothesis $\beta \in$ Post(P). Since β exists the first RHS format may be applied in this case. Also since S is a

non-terminal already appearing in the grammar the bias in favour of old non-terminals will ensure that S is used in forming the new rule. Thus the new rule will have the form $S \rightarrow \beta S$ which is of the correct form.

This completes the proof of the inductive case and the theorem.

Assume now that rather than an arbitrary presentation of sample strings, the strings in our sample set P are presented in order of nondecreasing length. It is now possible to allow only terminal strings of length one in our rewrite rules. In this case the form of our grammar under the bias of old non-terminals becomes:

$$\begin{array}{ll} S \rightarrow a_i S & a_i \in \Sigma \\ S \rightarrow b_i & b_i \in \Sigma \end{array}$$

where Σ denotes the set of all terminals which have appeared in any string in P. If we let A denote the set of a_i appearing in the rules and similarly let B denote the set of b_i, then when sufficient examples have been presented we will have A=B=Σ. At this point a grammar of the above form defines a most general grammar G_G, where L(G) = Σ^+.

Similarly, we can show that a bias exclusively favouring new non-terminals can define a most specific grammar G_S. If we have a bias in favour of using new non-terminals then given a language that is regular and a set of positive examples P and a known alphabet Σ we will produce a grammar of the form:

$$\begin{array}{l} S \rightarrow b_1 \\ S \rightarrow a_1 B_1 \\ B_1 \rightarrow b_2 \\ B_1 \rightarrow a_2 B_2 \\ \cdot \\ \cdot \\ \cdot \\ B_n \rightarrow b_m \\ B_n \rightarrow a_m B_q \quad q > n \end{array}$$

where a_i, b_i are elements of Σ and B_i are non-terminals. Note that the condition q > n ensures that there are no recursive rewrite rules in this grammar. Also in the above analysis we have assumed that all terminal string substitutions are of length one. (It is easy to modify the algorithm to ensure that this condition is met.) The grammar just described, which we can denote as G_S is a finite grammar with L(G_S) = P. This is thus the most specific possible grammar which will generate the entire set of examples P.

We have just shown how a simple bias in favour of old or new non-terminals can generate a most general and a most specific grammar for a particular set of example strings P. The bias towards using old non-terminals if used exclusively will overgeneralize and produce a grammar for the universal language Σ^+, on the other hand using strictly new non-terminals yields the relatively uninteresting finite grammar for the set of strings given as example strings. The interesting cases arise when one uses a combination of biases, at different points in the presentation of the sample strings. In fact it is possible to define a partial order over the grammars generated from a set of sample strings. Assume that we initially use only the bias in favour of new non-terminals until we have some base grammar G_S. Parse completion now provides us with a principled way to generalize this grammar, by adding additional rules in which the constituents are either terminals or old non-terminals. These rules will convert G_S into a grammar for an infinite language by adding recursive rewrite rules. Further, each new sample string will produce a set of new grammars from each previous candidate grammar, and each of these new grammars will be strictly more general than at least one of the previous candidate grammars.

The following example will clarify this process. Assume that our target language is the regular language $L = 0(0 + 1)^*$. To generate our initial G_S we will consider all of our positive sample strings of length two or less. (We will see below why this is a good way to initialize G_S.) Thus the set of sample strings from which G_S is generated is $\{0, 00, 01\}$. Applying parse completion restricted to the RHS forms for right linear grammars and with a bias for new non-terminals, the following grammar, G_S, is generated:

$$
\begin{aligned}
&S \rightarrow 0\\
&S \rightarrow 0A_1\\
&S \rightarrow 0A_2\\
&A_1 \rightarrow 0\\
&A_2 \rightarrow 1
\end{aligned}
$$

Assume that we now start generalizing G_S by applying a bias in favour of old non-terminals. The next sample string is 000 which yields the partial derivation shown in figure 10 which returns the fail marker (A_1 1 3). The first RHS format for right linear grammars may be applied to this fail marker and allowing only extension with old non-terminals this RHS format yields 3 new candidate rules:

$$
\begin{aligned}
&A_1 \rightarrow 0A_1\\
&A_1 \rightarrow 0A_2\\
&A_1 \rightarrow 0S
\end{aligned}
$$

The first and third rule above permit the successful completion of the parse, so these two rules

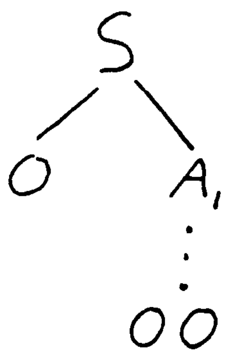


Figure 10: Partial derivation for the string 000

may be added to G_S producing two new more general grammars. Our next sample string is 011 which yields the partial derivation shown in figure 11. Applying parse completion with the same restrictions as before again yields three candidate rules:

$A_2 \rightarrow 1A_2$
$A_2 \rightarrow 1A_1$
$A_2 \rightarrow 1S$

In this case only the first of these rules will allow a successful completion of the parse, and this is the only candidate kept. Thus in this case each candidate grammar produces only one new more general grammar. The progression of grammars generated in this process is summarized by the tree structure presented in figure 12. This tree structure in fact represents the partial order of grammars induced by this set of sample strings. Any grammar in this tree is strictly more general than any ancestor in the tree. (This follows because of the monotonic increase in the number of rules in a grammar as you get further from the root and the fact that each rule is added *because* the parent grammar failed to parse a string.) One branch of the tree has been extended to show the effects of two additional sample strings 001 and 010. After these strings have been added a grammar is produced which exactly captures the target language. As with a version space, the only part of this upward growing tree which needs to be maintained is the current leaf set. The leaf set of this structure is analogous to the S set of a version space [17].

The important question is how to decide when to switch from rules that use new non-terminals to rules that use old non-terminals. More generally the question is at what point should we start

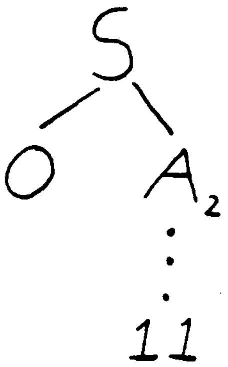


Figure 11: Partial derivation for the string 011

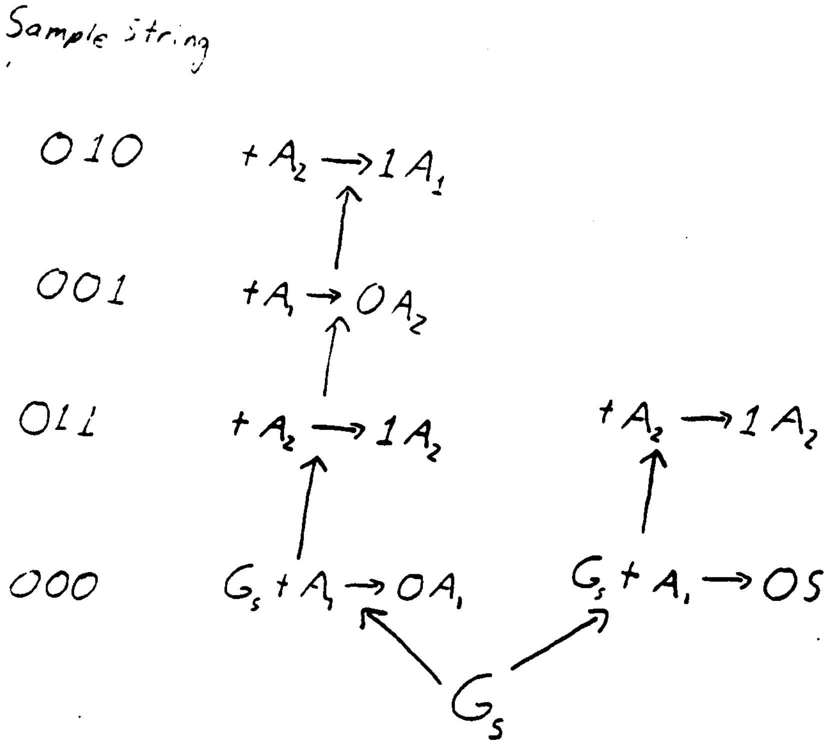


Figure 12: Tree of grammars derived for the language $0(0 + 1)^*$.

adding recursive rewrite rules to our grammar. The existence of G_G clearly illustrates that if we start adding recursive rewrite rules too early we will overgeneralize the target grammar. We have already noted that because parse completion only adds rules to existing grammars, the grammars produced increase monotonically in generality. This means that once over-generalization occurs the parse completion algorithm cannot recover from it. Thus we must ensure that initially G_S has enough "stuff" in it that our target grammar will appear somewhere in the partial order of grammars induced from G_S.

Can we build a suitable G_S from only a finite set of sample strings? For the case of regular languages and right linear grammars the answer is yes. This result follows from the Pumping Lemma for regular languages.

Pumping Lemma: Let L be a regular language. Then there is a constant n such that if z is any string in L and $|z| \geq n$, we may write $z = uvw$ in such a way that $|uv| \leq n$, $|v| \geq 1$, and for all $i \geq 0$, uv^iw is in L. Furthermore, n is no greater than the number of states of the smallest finite automaton (FA) accepting L. [11]

The important key is not the existence of the Lemma, but the ideas used in its proof. The proof relies on the fact that for any regular language there is a deterministic finite automaton (FA) accepting it. We let n be the number of states in the automaton and then show that in accepting a string of length greater than n the automaton must repeat a state. The path in the transition diagram for the automaton must therefore contain a loop, and this loop corresponds to the string v on which we pump. In fact, if we restrict the terminal strings in rules to length one, then we can create a correspondence between the transition diagram of our FA and our right linear grammar. We construct our FA so it has a unique start state α and a unique final state β.[19] Each rule of the form $S \rightarrow a_i$ corresponds to a transition from α to β with label a_i. Each rule of the form $A_j \rightarrow a_i$ corresponds to a transition from a state A_j to β with label a_i. Finally a rule of the form $A_j \rightarrow a_iA_k$ corresponds to a transition from a state A_j to a state A_k with label a_i. (If A_j = S then the transition is from α to state A_k.) For our example language $0(0 + 1)^*$ the induced grammar and corresponding FA are shown in figure 13. The important point about this correspondence is that for each non-terminal in the grammar there is a unique state in the FA. In fact if our FA has n states and we number these from 1 to n, with α numbered 1, β numbered n, and the other states numbered in the order in which the non-terminals which label the states were introduced to the grammar, and we remove all transitions in the FA which go from state i to some state $k < i$, then the resulting FA accepts precisely L(G_S). Thus we can now bound our G_S. If the minimum FA

[19] It is easy to show that any FA with multiple final states can be converted into an FA with a unique final state. [11]

that accepts language L has n states, then G_S must contain at least n non-terminals to be able to model any FA that accepts language L.

Grammar

$S \rightarrow 0$
$S \rightarrow 0A_1$
$A_1 \rightarrow 0$
$A_1 \rightarrow 0A_1$
$A_1 \rightarrow 0A_2$
$A_2 \rightarrow 1$
$A_2 \rightarrow 1A_2$
$A_2 \rightarrow 1A_1$

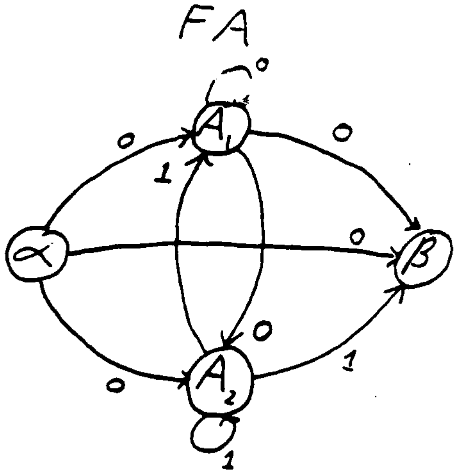


Figure 13: Grammar and FA for the language $0(0 + 1)^*$.

We shall now state and prove these results more formally. What we will prove is that there is a subset of the strings in L of length $\leq 2n-1$ from which we can define G_S using parse completion and a bias for new non-terminals. We can then guarantee that there is at least one grammar in the partial order generalized from G_S for the target language. The idea behind the proof is to show that given a minimal n-state FA for our target language, we can construct another machine which accepts exactly the same language and further contains all of the arcs and states that correspond to a grammar G_S built from some subset of the strings in the language of length $\leq 2n-1$. Finally we show that the second machine corresponds to some point in the partial order of grammars generalized from G_S.

Theorem: Given a regular language L there exists a finite subset of the strings in L which, if these strings are presented in increasing order of length, and parse completion for right linear grammars is applied with a bias in favour of new non-terminals, will generate a grammar, G_S, with the following property: The partial order of grammars generated from G_S by applying parse completion, with a bias for old non-terminals, contains at least one grammar for the language L.

Proof: The proof is by construction of appropriate FA's, and relies on the one to one correspondence between machine and right linear grammars already described.

Let M be a minimal FA for L and let n be the number of states in M. Let the start state of M be α, and the final state of M be β. Assume, without loss of generality, that M has one final state and does not have any ε transitions. Let G be the directed graph corresponding to the transition diagram for M.

We construct a new machine M_S from M. M_S is a machine that corresponds to our most specific grammar G_S. This implies that the transition diagram of M_S must be an acyclic directed graph and furthermore, that every node must lie on at least one path from α to β. The first property is required by the fact that the transitions in M_S that do not terminate at β correspond to productions of the form $A_i \rightarrow a_i A_k$ where $k > i$. Thus the nodes in the transition graph may be topologically ordered, hence the transition graph must be acyclic. The second property comes from the manner in which G_S is constructed. A node is added to M_S when a production of the form $A_i \rightarrow a_i A_k$, where A_k is a new non-terminal, is added to G_S. In parse comletion such a rule is added only if it is needed to complete the parse of the new string, so every non-terminal is used in the derivation of at least one string. So, the node in the transition graph corresponding to that non-terminal must lie on at least one path from α to β.

Initialize M_S to have start state α and final state β. Set i equal to one. Do a breadth first search of G starting at node α. For each arc out of α, if the node at the other end has not yet been labelled, we add that node to a queue of nodes to be scanned, label that node A_i, increment i, and add the newly labelled node and the arc just examined to M_S. If the arc scanned terminates at β, we also add this arc to M_S. If an arc out of α terminates in a labelled node, the arc is not added to M_S. When all the arcs out of α have been examined, the first node in the queue is removed and the arcs out of it are examined in the same manner. The process continues until the queue is empty. It is easy to show the resulting graph is acyclic. Each node is labelled just once, so the node labels can define a topological order on the nodes. The existence of a topological order on the nodes shows the graph is acyclic.

The machine we have constructed from this process is acyclic as desired, but not every node is guaranteed to be on a path from α to β. It is possible that some nodes will have no arcs leaving them. (Every node but α *however must have at least one* arc into it from the first time it is scanned.) For each node, A_i, with no arcs leaving it, find an acyclic path P from A_i to β in G. Let P be $A_i t_k A_k \cdots t_m A_m t_\beta \beta$, where A_j is a state label and t_j is a transition label. We add the path $A_i t_k A_l \cdots t_m A_p t_\beta \beta$ to M_S, where $A_l \cdots A_p$ are new states. Each set of states and transitions added in this fashion will not violate the acyclic nature of the trasition graph for M_S, and when this process is finished every node in M_S must lie on at least one path from α to β. Furthermore, the longest path from α to β in M_S is of length at most $2n-1$. Consider first all paths that only pass through nodes that came from M. Since all paths are acyclic, they can pass through each node at most once, so these paths are of length at most n. Now consider any path from α to β that passes through both nodes from M and new nodes added in step two of constructing M_S. Such a path must consist of two pieces, a prefix $\alpha \cdots A_k$ which contains only intermediate nodes from M, and a postfix $A_k \cdots \beta$ which contains only intermediate nodes that are not in M. The maximum length of the prefix is $n-1$, since it must be acyclic and cannot contain β. The postfix corresponds to some acyclic path in G from A_k to β, hence can have length at most n. So the total path length of any path from α to β in M_S is at most $2n-1$.

Let S be the set of strings accepted by M_S. S must be finite as M_S is acyclic, furthermore every string in S has length at most $2n-1$. Each of these strings must also be accepted by M, hence S is a subset of L. Using the mapping already described we can construct a G_S corresponding to M_S, and this G_S will be a grammar of the form:

$$\begin{array}{l}
S \rightarrow b_1 \\
S \rightarrow a_1 B_1 \\
B_1 \rightarrow b_2 \\
B_1 \rightarrow a_2 B_2 \\
\quad \cdot \\
\quad \cdot \\
\quad \cdot \\
B_n \rightarrow b_m \\
B_n \rightarrow a_m B_q \quad q > n
\end{array}$$

That is G_S has the from of a grammar built by parse completion for right linear grammars with a bias in favour of new non-terminals. The required presentation order of the strings in S to generate G_S can be derived mechanically from M_S. Start with the strings that correspond to all paths of length one from α to β. Then consider all paths of length two using as an intermediate node A_1, then each of the other nodes in topological order. Continue in this manner until you have enumerated every path from α to β in M_S. If you take the yield of each path in this order, the strings are enumerated in the desired presentation order.

So far we have shown that we can build a G_S, using parse completion for right linear grammars and a bias in favour of old non-terminals, from a finite subset of the strings in L. It now remains to show that the partial order of grammars generated from G_S by parse completion contains at least one grammar for the language L.

First we prove that from M_S we can construct a machine which accepts the language L. The required construction is simple; add all the arcs in M that are not in M_S to M_S. The new machine will be identical to M except for the extra paths added in step two of the construction of M_S. These additional paths cannot add any additional strings to L. The proof is by contradiction. Assume that the new machine, M', accepts some string l that is not accepted by M. There must be a path from α to β with yield l. Furthermore, this path must pass through some nodes not in M, otherwise the same path would exist in M. As noted previously, this path must consist of a prefix $\alpha \cdots A_k$ which contains only intermediate nodes from M, and a postfix $A_k \cdots \beta$ which contains only intermediate nodes not in M. However, the construction in step two for M_S will create a path $A_k t_f A_l \cdots t_m A_p t_\beta$ if and only if there is a path $A_k t_f A_j \cdots t_m A_m t_\beta \beta$ in M. Then the path $\alpha \cdots A_k t_f A_l \cdots t_m A_p t_\beta \beta$ and the path $\alpha \cdots A_k t_f A_j \cdots t_m A_m t_\beta \beta$ must have the same yield, but the path $\alpha \cdots A_k t_f A_j \cdots t_m A_m t_\beta \beta$ is contained entirely within M. Thus l must be accepted by M.

Now we must show that each arc in M - M_S (*i.e.* each arc in M but not in M_S) can be added by parse completion for right linear grammars restricted to using old non-terminals. This is sufficient since the partial order is searched exhaustively by parse completion and the number of arcs in M is finite. If each of the transitions actually in M - M_S will be added by applying parse completion to some string in L, then the set of transitions corresonds to some point in the partial order. (There may in general be many points in the partial order which correspond to this machine, each reached via a different permutation of the arc order.) An arc is added by parse completion if and only if the corresponding rule will allow the derivation of a string to be completed. So it is sufficient to show that for each arc in M - M_S there is a string in L whose derivation can be completed by adding this arc to the current machine. (Note that there may be other ways to complete the derivation which add different arcs to the machine, but as long as at least one complete derivation adds this arc there will be a path in the partial order leading to M'.) There is a simple construction to generate the required string l for each arc a. Let the tail of a be state A_i and the head of a be state A_j. A_i and A_j may be any states including α or β, and A_i may be the same as A_j. We have already proven that every state A_i in M_S lies on at least one path from α to β. Let P_i be an acyclic path from α to β passing through A_i, and P_j be an acyclic path from α to β passing through A_j. Construct l by taking the yield of the segment of the path from α to A_i, the label on the arc a, and the yield of the segment of the path from A_2 to β. We can guarantee that there is at least one derivation of this string that requires the addition of the arc *a*, and further this string *l* is in L.

Finally we note that only a finite number of strings are required to generalize M_S to a machine that accepts L. This follows immediately from the fact that M - M_S is finite, and that each string, *l*, defined above adds at least one arc in M - M_S to M_S. This completes the proof of the theorem.

Figure 14 illustrates the construction of M_S from M, and the generalization of M_S to M' for a particular machine M.

Thus for the case of linear grammars we have shown that there is a unique bound on the set of strings needed to build G_S, and that the partial order induced from this minimal grammar will contain at least one grammar for the target language, and this grammar may be found after a finite number of steps of parse completion. This process was illustrated in our example for the language $0(0 + 1)^*$, where *n* had the value two since the minimal FA for this language has two states.

The partial order induced from G_S provides a way to generate something equivalent to the S-set in a version space algorithm. This is only half of the version space algorithm. To complete the algorithm we need some manner to restrict G_G, our most general grammar. Parse completion yields no insights for this problem, however one way to create such a G-set for grammars has been suggested by [29].

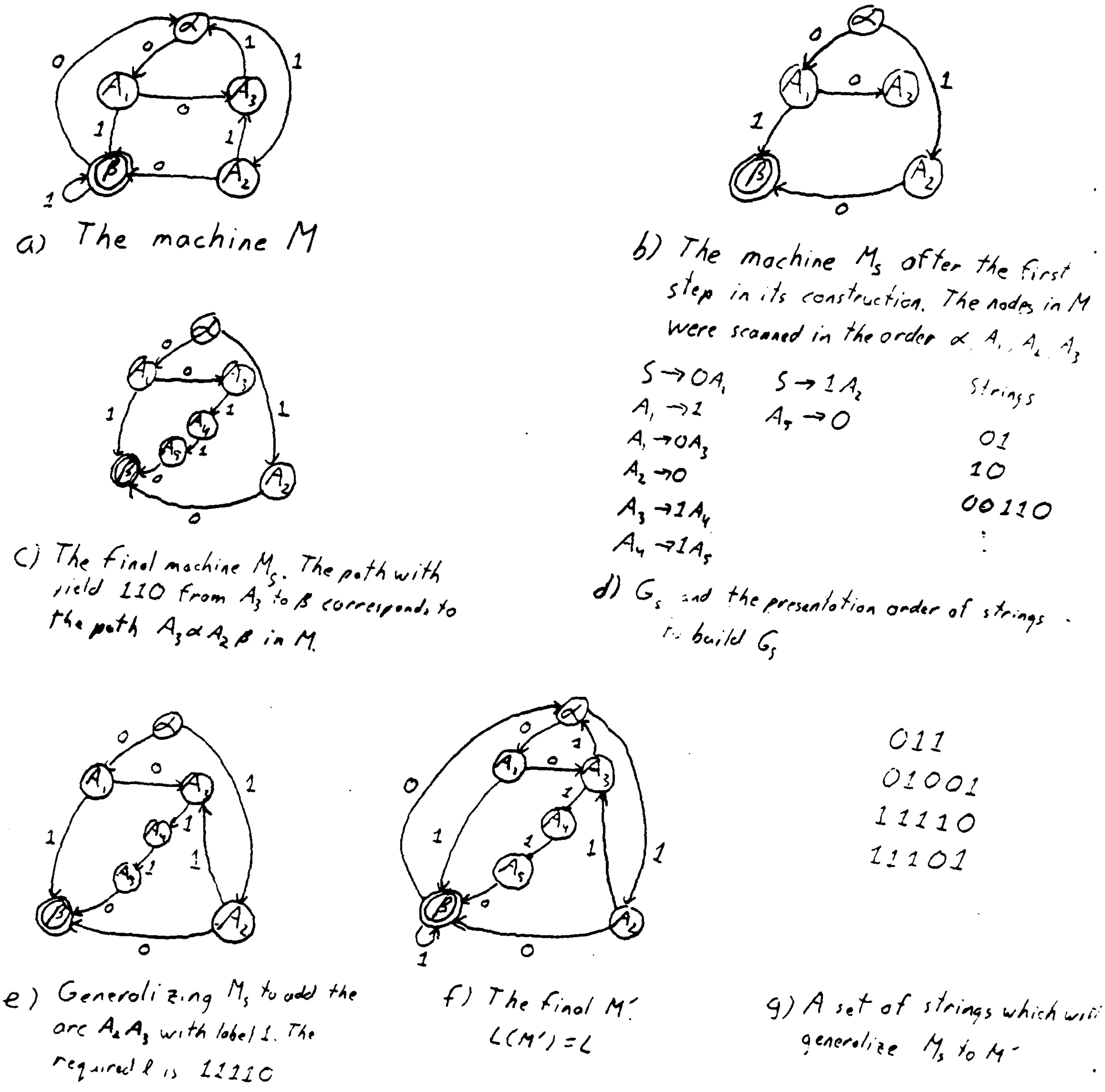
The results we have just presented for a partial order for right linear grammars can be generalized to provide a partial order for all context free languages. Assume that we are given a language that is context free, a set of positive examples P and a known alphabet Σ. We will again consider applying a bias in favour of old non-terminals and one in favour of new non-terminals to the RHS formats allowed for CNF grammars.[20]

If we favour old non-terminals in our RHS formats, then the resulting grammar will be of the form:

$$S \rightarrow SS$$
$$S \rightarrow a_i \quad \forall a_i \in \Sigma$$

[20] We can restrict ourselves to CNF grammars since any context free language may be described by a CNF grammar. [11]

Figure 14: Construction of M_S and M' for a particular M



a) The machine M

b) The machine M_s after the first step in its construction. The nodes in M were scanned in the order α, A_1, A_2, A_3

c) The final machine M_S. The path with yield 110 from A_3 to β corresponds to the path $A_3 \alpha A_2 \beta$ in M.

$S \rightarrow 0A_1$ $\quad$ $S \rightarrow 1A_2$

$A_1 \rightarrow 1$ $\quad$ $A_5 \rightarrow 0$

$A_1 \rightarrow 0A_3$

$A_2 \rightarrow 0$

$A_3 \rightarrow 1A_4$

$A_4 \rightarrow 1A_5$

Strings

01

10

00110

⋮

d) G_s and the presentation order of strings to build G_s

e) Generalizing M_s to add the arc A_2A_3 with label 1. The required ℓ is 11110

f) The final M'. $L(M') = L$

011

01001

11110

11101

g) A set of strings which will generalize M_s to M'

This can easily be proven by case analysis. Consider the first sample string in P of length greater than one. (Each sample string of length one can only be partitioned into one element of size one, and the RHS format for CNF grammars in this case can only produce a rule $S \rightarrow a_i$ where $a_i \in \Sigma$.) Since this string is of length greater than one the first RHS format for CNF grammars will apply, and a rule with two non-terminals on the RHS will be created. Further since the only non-terminal in the grammar is S this rule will have the form indicated. Now assume our sample string is of length n, then $n-1$ applications of the rule $S \rightarrow SS$ will partition the string into n partitions of size one. Each partition of size one will either already have a rule of the form $S \rightarrow a_i$ in the grammar, or application of the second RHS format will introduce a rule of this form. Since the rule $S \rightarrow SS$ is sufficient to partition any string into partitions of size one, once this rule is introduced a parse can never fail at a partition of size greater than one, so all other rules introduced into the grammar must be of the form $S \rightarrow a_i$. The process of adding new rules of this from must stop once we have a rule for each a_{i} in Σ, at which point we will have the grammar G_{G}. This grammar, G_{G}, generates the language Σ^+ and is clearly the most general grammar for the alphabet Σ.

Now consider the bias favouring new non-terminals. Once again we will assume that the strings in our sample set P are presented in order of non-decreasing length. With this assumption, we can show that this bias will generate a most specific grammar, $\mathrm{G_S}$:

$$\begin{array}{ll} A_0 \rightarrow A_i A_j & \\ A_i \rightarrow A_k A_l & k,l > i \\ A_i \rightarrow a_j & \end{array}$$

where $A_0 = S, A_i$ are non-terminals and a_i are terminals. The restriction that $k,l > i$ implies that there are no recursive rewrite rules, thus $\mathrm{G_S}$ is a finite grammar. It is easy to show that $\mathrm{G_S}$ must have this form. The RHS formats for CNF grammars ensure that all rules will be of one of the two forms in $\mathrm{G_S}$, further the restriction of allowing only new non-terminals in the RHS ensures that the condition $k,l > i$ holds each time a new rule of this form is introduced to the grammar. We will now prove that $\mathrm{L(G_S) = P}$, hence that $\mathrm{G_S}$ is our desired most specific grammar. The proof proceeds by induction on the number of sample strings shown to the system. Our inductive hypothesis is that there is a unique derivation for each sample string seen and that these are the only possible derivations in this grammar.

Base Case: If the first sample string α is of length one then this string will create a grammar with only one rule:

$S \rightarrow \alpha$

Clearly the language of this grammar is $\{\alpha\}$. If the first sample string is of length greater than one, then the grammar created will have rules of the form $A_i \rightarrow A_j A_k$ as well as rules of the form $A_i \rightarrow a_i$. The restriction that only new non-terminals may appear on the right hand side ensures that each non-terminal will appear as the left hand side of at most one rule in the new grammar. Further in parse completion rules are added to a grammar only when needed to complete the derivation of a string, so each rule in the grammar must be used in the derivation of the initial string. Thus there is a one to one correspondence between the internal nodes of the derivation tree and the non-terminals of the grammar. This implies that there is only one derivation tree that can be built with this grammar, and this tree corresponds to the derivation of the first sample string α. Thus our inductive hypothesis holds for the base case.

Inductive Case: Assume that after the first *n-1* sample strings we have a grammar with a unique derivation for each sample string, and that these *n-1* derivations are the only ones possible in this grammar. Now consider the derivation tree for the *nth* sample string. The parse completion algorithm will attempt to complete this parse as far as possible before adding new rules. Let T be the partial derivation tree for the new sample string. Since T contains only applications of rules in the existing grammar, by the induction hypothesis T must be a *unique* tree. Now since T is a partial derivation, it will contain some leaf nodes labelled with non-terminals. We first note that any such non-terminal will only appear in the existing grammar in rules of the form $A_i \rightarrow a_i$ since we know T is the most complete partial derivation possible. (A parse does not fail until we reach a point at which the terminals in any applicable rule and the terminals in the string do not match.) Let the non-terminals leaves in T be labelled T_1 to T_m, and the corresponding unparsed substrings of the sample string have labels β_1 to β_n. Now consider the derivation of β_i from T_i. It is easy to show using the argument of the base case that the rules added to complete this derivation alone, can generate only one derivation tree, that for the substring β_i. Further, since only new non-terminals are used to create these rules, the only non-terminal these new rules will have in common with the original grammar is T_i. Thus the new rules cannot interact with any of the existing rules to from any derivations other than the derivation of β_i. Finally since T is unique and each T_i will be unique, the entire derivation tree for the new string is unique, and further since none of the new rules can interact with any old rules except through T, the only additional derivation possible with this new grammar is that for the new sample string. This completes the proof of the inductive case.

As for the linear grammar case we now have a G_G and a G_S for CNF grammars. We can now define a partial order over the CNF grammars induced from a given set of sample strings P in the same manner we defined the partial order for right linear grammars. We start with a grammar G_S and then generalize it by biasing our substitutions in favour of old non-terminals. The following example illustrates this process.

Consider the context free language a^nb^n as our target language. To generate an initial G_S we will consider all positive sample strings of length 4 or less. Thus our sample set is {ab, aabb}. Applying parse completion restricted to CNF grammars and with a bias in favour of new non-terminals we find that our initial G_S set will consist of a pair of grammars:

$$
\begin{aligned}
G_{S_1} = \{ & S \rightarrow A_1A_2, A_1 \rightarrow a, A_2 \rightarrow b, \\
& A_2 \rightarrow A_3A_4, A_3 \rightarrow a, A_4 \rightarrow A_5A_6 \\
& A_5 \rightarrow b, A_6 \rightarrow b \}
\end{aligned}
$$

$$
\begin{aligned}
G_{S_2} = \{ & S \rightarrow A_1A_2, A_1 \rightarrow a, A_2 \rightarrow b, \\
& A_2 \rightarrow A_3A_4, A_3 \rightarrow A_5A_6, A_4 \rightarrow b, \\
& A_5 \rightarrow a, A_6 \rightarrow b \}
\end{aligned}
$$

Assume that our next sample string is aaabbb, and that we parse with G_{S_2}, which will yield the partial derivation shown in figure 15. The fail marker returned by this partial derivation is (A_6 2 5). Now we start generalizing G_S by considering CNF RHS forms and allowing only old non-terminals on the RHS. Since our current grammar has 7 non-terminals, there are 49 different RHS forms which may be tried to complete the parse. One of these creates the new rule $A_6 \rightarrow SA_6$ which will allow the parse to be completed using existing rules from the grammar. Thus G_S + $A_6 \rightarrow SA_6$ is one point in the partial order of generalizations of G_S, and in fact is a grammar for our target language a^nb^n. In fact the RHS formats tried at this stage yield several grammars which have L(G) = a^nb^n, which means that in this case there are several points one level above G_S in our partial order which correspond to our target language. As in the regular grammar case, repeated applications of parse completion produce an upward growing tree of grammars which corresponds to a partial order of the grammars based on generality. The leaf set of this tree at each stage of the algorithm is our current S set.

We now must attempt to answer the same question that faced us in the regular language case: Can we build a G_S from a finite set of sample strings such that our target language will appear somewhere in the partial order induced from G_S for any context free target language? We can derive a result very similar to our result for regular lanuages using the pumping lemma for

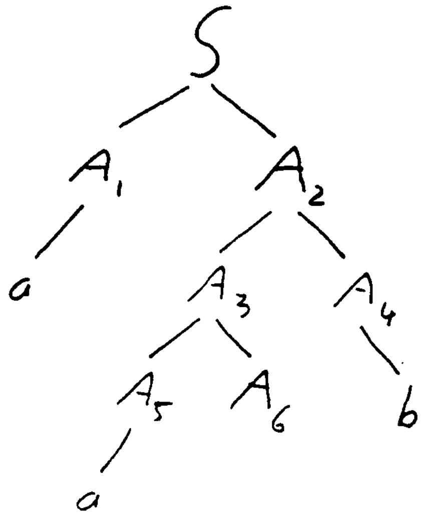


Figure 15: Partial derivation for the string **aaabbb.**

context free languages.

Pumping Lemma: Let L be any CFL. Then there is a constant n depending only on L, such that if z is in L and $|z| \geq n$, then we may write $z = uvwxy$ such that

1. $|vx| \geq 1$
2. $|vwx| \leq n$, and
3. for all $i \geq 0$ uv^iwx^iy is in L. [11]

As before, we are concerned with the proof of this Lemma rather than its existence. For the context free case, the proof relies on the fact that if there are k non-terminals in a minimal CNF grammar for L than any string of length > 2^k must have a repeated non-terminal somewhere in its derivation tree. This follows from the fact that if the parse tree of a string generated by a CNF grammar has no path of length greater than i, then the string is of length no greater than 2^{i-1}.(This can easily be proven by induction, see [11] for details.) Thus a string of length 2^k must have a longest path of length at least $k + 1$ in its derivation tree. This longest path must have $k + 2$ vertices in it of which $k + 1$ are labelled with non-terminals. Since there are only k distinct non-terminals, two vertices, v_1 and v_2 on this path have the same label. Then we can replace the subtree rooted at v_2 with the one rooted at v_1 producing a second copy of the *yield* of the subtree rooted at v_1.[21] We can repeat this process *i* times, and produce *i* copies of part of the string as illustrated in figure 16. Thus in this case the power to create infinite strings is produced by having rules which allow a non-terminal to be its own ancestor in the derivation tree.

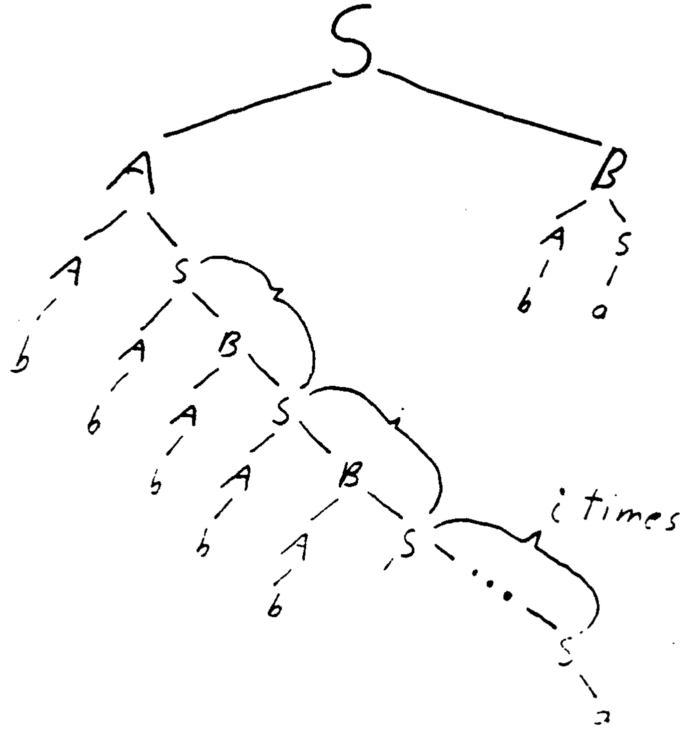


Figure 16: A derivation tree for uv^iwx^iy where u = a, v = bb, w = a, x = ε, y = ba.

We will now prove that if there is a CNF grammar for a context free language, L, with *k* non-terminals, then there is a finite subset of the strings in L from which we can define a G_S using the procedure described. We can then guarantee that there is at least one grammar in the partial order generalized from G_S for the target language. The idea behind the proof is to construct a subset of G that is a grammar for a finite language, and to modify this subset of G so that it contains no redundant non-terminals. We then show that the resulting G_S can be constructed from some subset of the strings in L of length $\leq 2^{2n-1}$ by parse completion. Finally we show that we can build a grammar G' from G_S such that L(G') (the language generated by G') equals L, and G' appears in the partial order of grammars generalized from G_S.

[21] The *yield* of a subtree is simply the substring that appears at the leaves of the tree.

Theorem: Given a context free language, L, there exists a finite subset of the strings in L which, if the strings are presented in increasing order of length, and parse completion for CNF grammars is applied with a bias for new non-terminals, will generate a grammar G_S with the following property: The partial order of grammars generated from G_S by applying parse completion with a bias for new non-terminals contains at least one grammar for the language L.

Proof: Let G be a non-redundant CNF grammar for L. (Non-redundant means all non-terminals appear in at least one derivation of a string in L.) Let n be the number of non-terminals in G.

Construct a graph T from G. The vertices of T are the non-terminals of G. There is a directed edge from A to B if and only if there is a production of the form $A \rightarrow BC$ or $A \rightarrow CB$.

A grammar G_S built by parse completion with a bias for new non-terminals has two characteristics: it generates a finite number of strings, and every non-terminal is used in the derivation of at least one string. We wish to restrict G to produce a grammar G_S with these two characteristics.

We construct a grammar G_{S_1} which is a restrictin of G by first finding the largest acyclic subgraph T' of T. We construct G_{S_1} from G by first including all rules of the form $A_k \rightarrow a_i$ that are in G. We then include each rule in G of the from $A_k \rightarrow A_jA_l$ if and only if there is an arc from A_k to A_l and an arc from A_k to A_j in T'. In [11] it is proven that if you construct a graph whose vertices are the non-terminals of a grammar and which includes an arc from A to B if and only if there is a production of the form $A \rightarrow BC$ or $A \rightarrow CB$, then if this graph is acyclic the corresponding grammar generates only a finite number of strings. In fact, if we define the *rank* of a non-terminal, A, as the length of the longest path in the graph beginning at A, the proof in [11] shows that if A has rank r no terminal string derived from A has length greater than 2^r. Now T' is the graph corresponding to G_{S_1} and T' is acyclic, so G_{S_1} is a grammar for a finite language.

G_{S_1} has one of the two properties required of G_S, but G_{S_1} may contain some non-terminals that cannot be reduced to terminals. Clearly these non-terminals will not appear in the derivation of any string in $L(G_{S_1})$. We must modify G_{S_1} so that all non-terminals are reducible to terminals. There are two ways in which a non-terminal can fail to be reducible to a terminal. The first occurs if the non-terminal does not appear as the LHS of any rule in the grammar. The second occurs if the non-terminal is part of an infinite derivational loop. (That is the non-terminal A_j can only be reduced to strings that contain one or more occurrences of A_j.) The second condition can only occur if T' were to contain a cycle, but T' is acyclic. So we need only be concerned with non-terminals, A_j, that do not appear on the LHS of any rule in G_{S_1}.

For each non-terminal A_j that does not appear on the LHS of any rule, find a shortest derivation $A_j \overset{*}{\underset{G}{\Rightarrow}} \alpha_j$, where α_j is a terminal string. Take the derivation tree for $A_j \overset{*}{\underset{G}{\Rightarrow}} \alpha_j$ and relabel all the nodes except A_j with non-terminals not yet occurring in the grammar. Add to G_{S_1} the productions derived from this relabelled derivation tree. The new productions will all have the form $A_i \rightarrow a_j$ or $A_i \rightarrow A_k A_l$, with $k, l > i$ if we number nodes from the root of the tree in breadth first order. So the new productions preserve the fact that the grammar generates a finite language. When this process is finished for all A_j that were not reducible to terminals in G_{S_1}, we will have a finite grammar in which every non-terminal is used in the derivation of at least one string. This grammar is the required G_S.

We next bound the length of any string generated by G_S. Recall that if A has rank r, no terminal string derived from A has length greater than 2^r. Now G_{S_1} contains only the non-terminals in G, hence T' contains at most n nodes. Thus the rank of S in G_{S_1} is at most n-1. Now consider the additional non-terminals added when converting G_{S_1} to G_S. The productions that these non-terminals appear in are derived directly from the derivation tree for a shortest derivation $A_j \overset{*}{\underset{G}{\Rightarrow}} \alpha$, so the rank of any of these non-terminals is simply the length of the longest path from that non-terminal to a leaf in the derivation tree. In a shortest derivation, in any path from the root to a leaf, each non-terminal can appear at most once. (The proof is by contradiction. If some non-terminal appears twice in the same path, call the appearance closest to the root the first occurrence, the appearance closest to a leaf the second occurrence. We can replace the sub-tree rooted at the first occurrence with the sub-tree rooted at the second and produce a shorter derivation.) Now since $A_j \overset{*}{\underset{G}{\Rightarrow}} \alpha$ is a shortest derivation, the longest path from the root to a leaf in the derivation tree is at most n. If we construct T'' for G_S as T' was constructed for G_{S_1}, we can increase any path in T' by at most the length of the longest path in any of the derivation trees used to convert G_{S_1} to G_S. Thus the rank of any node in T' will be increased by at most n in T'', implying that S will have rank at most $2n - 1$ in T''. From the theorem in [11] all strings generated by G_S have length at most 2^{2n-1}.

The bound on the length of any string produced by G_S shows that $L(G_S)$ is finite. We must also show $L(G_S) \subsetneq L$. G_{S_1} is a subset of G, so any complete derivation (*i.e.* any derivation whose final product is a string of terminals) in G_{S_1} must also be a derivation in G. So the string produced as the yield of any complete derivation using only rules in G_{S_1} is in L. Now we must consider derivations that use both rules in G_{S_1} and some of the rules that contain non-terminals that did not appear in G. The new non-terminals can only appear in a derivation of $A_j \overset{*}{\underset{G}{\Rightarrow}} \alpha_j$ for exactly one A_j in G. Thus we may split any derivation in G_S into two parts. The first part will be a derivation from S, using only non-terminals in G_{S_1}. Every rule used in this part of the derivation also appear in G, so this partial derivation tree may also be built in G. The leaves of this tree will either be terminals, or non-terminals which cannot be reduced any further using rules in G_{S_1}. The second part of the derivation will take each non-terminal, A_j, at a leaf and will attach the sub-tree corresponding to $A_j \overset{*}{\underset{G}{\Rightarrow}} \alpha_j$ to it to complete the derivation. Since $A_j \overset{*}{\underset{G}{\Rightarrow}} \alpha_j$ is also a valid derivation in G (although it will use different productions) for each A_j, the entire derivation tree could have been produced by G. Thus the string that is the yield of the derivation is in L. So $L(G_S)$ is a subset of L.

To be complete, we must also verify that the G_S we have defined can in fact be generated by parse completion, with a bias for new non-terminals, when the positive examples are the strings in $L(G_S)$ presented in increasing order of length. The proof is mechanical and the details are left to the reader.

Now it remains to show that the partial order of grammars generated from G_S by applying parse completion with a bias for old non-terminals, contains at least one grammar for the language L. First we will show that G_S can be generalized to a grammar G' such that L(G') equals L. The required construction is simply to add the productions in $G - G_{S_1}$ (*i.e.* the production in G that do not appear in G_{S_1}) to G_S. It is immediately clear that G' will generate at least every string in L, since $G \subset G'$, however, we must ensure that G' does not generate any string not in L. Assume that there is a string l, such that l is derivable from S in G' but l is not in L. There must be a derivation for l in G'. As noted before the derivation can be divided into two pieces, an initial partial derivation which uses only rules in G, and a second part where non-terminal leaves in the initial tree are reduced to terminal strings using rules not in G. Now consider any non-terminal A_j not reduced in the first part of the derivation. Assume in the second part of the derivation A_j is reduced to the terminal string β. If this derivation uses any rule not in G, it must use the rules which correspond to the derivation $A_j \overset{*}{\underset{G}{\Rightarrow}} \alpha_j$ (*i.e.* $\beta = \alpha_j$) since these are the only rules not in G which could refer to A_j. But in creating G_S from G_{S_1} we added rules to form the derivation $A_j \overset{*}{\underset{G}{\Rightarrow}} \alpha_j$ if and only if there already existed in G a derivation $A_j \overset{*}{\underset{G}{\Rightarrow}} \alpha_j$. So for any derivation using rules not in G, there must be a derivation using only rules in G. Hence l must have a derivation using only rules in G, thus l is in L.

As a final point, it is necessary to show that G' can be generated from G by parse completion. The proof is very similar to the proof in the regular grammar case. We show that for each production in G - G_{S_1} there is a string in L which requires the addition of this production to complete the derivation. Since G - G_{S_1} is finite, only a finite nrmber of strings are required to generalize G_S to G'. The details are left to the reader. This completes the proof of the theorem.

Figure 17 illustrates the constructin of T, T', G_{S_1}, T'', G_S, and G' for a particular language.

6 Felicity Conditions and Biases

The proofs in the previous section for the bound on the number of strings needed to define G_S were existence proofs only. They stated that a finite set of strings which could define G_S existed, but in fact the construction given to build this set of strings relied on knowing a great deal about the target language. In fact it was necessary to already have a minimal FA or CNF for the target language. For the simple example languages in the previous section it is easy to get this information by inspection, but for more realistic problems it is not likely that this information will be readily available. However, both proofs relied on making the same distinction between two types of grammar rules. On the one hand, G_S was originally created from rules which used terminals or new non-terminals. These rules may be regarded as adding structure to the grammar. In the regular language case, these rules correspond to adding states and initial transitions to these states in our FA. For the CNF grammars, these rules generated an initial set of subtrees to act as fundamental constituents in the grammar. Once G_S was established, we added recursive rules to the grammar, by allowing RHS formats which used old non-terminals. For regular languages, these rules corresponded to adding cycles into the FA, while in the CNF grammars these rules corresponded to creating paths in the derivation tree in which the same non-terminal could appear more than once.

This simple distinction between rules that add structure and those that recombine existing structure suggests a means by which to approximate the formal results associated with G_S. Instead of requiring the input strings in non-decreasing length, and using a finite subset of the

Figure 17: The construction of T, T', G_{S_1}, T'', G_S, and G' for the grammar shown in part (a).

$S \rightarrow C_b A$

$S \rightarrow C_a B$

$A \rightarrow C_a S$

$A \rightarrow C_b D_1$

$A \rightarrow a$

$C_a \rightarrow a$

$B \rightarrow C_b S$

$B \rightarrow C_a D_2$

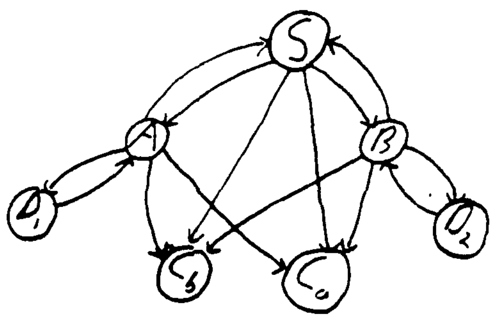
$B \rightarrow b$

$D_1 \rightarrow AA$

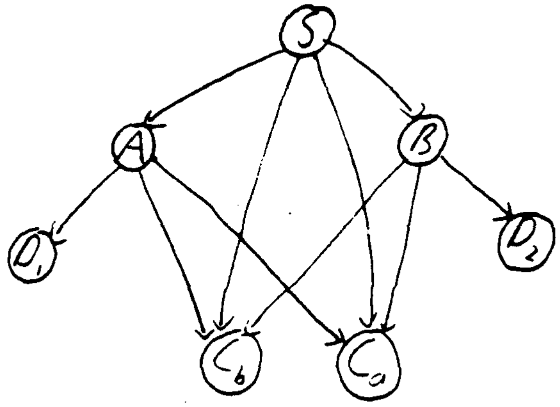
$D_2 \rightarrow BB$

$C_b \rightarrow b$

a) The grammor G



b) T corresponding to G



c) T', the largest acyclic subgraph of T

$S \rightarrow C_b A$

$S \rightarrow C_a B$

$A \rightarrow C_b D_1$

$A \rightarrow a$

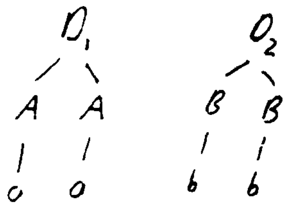
$B \rightarrow C_a D_2$

$B \rightarrow b$

$C_a \rightarrow a$

$C_b \rightarrow b$

d) G_{S_1}, the restriction of G to T'. Note that D_1 and D_2 do not appear as the LHS of any rule in grammar.



e) Derivation Trees for a shortest derivation for D_1 and D_2 in G.

$S \rightarrow C_b A$

$S \rightarrow C_a B$

$A \rightarrow C_b D_1$

$A \rightarrow a$

$C_a \rightarrow a$

$B \rightarrow C_a D_2$

$B \rightarrow b$

$C_b \rightarrow b$

$D_1 \rightarrow D_3 D_4$

$D_2 \rightarrow D_5 D_6$

$D_3 \rightarrow a$

$D_4 \rightarrow a$

$D_5 \rightarrow b$

$D_6 \rightarrow b$

f) The Grammar G_s

$S \rightarrow C_b A$

$S \rightarrow C_a B$

$A \rightarrow C_b D_1$

$A \rightarrow a$

$C_a \rightarrow a$

$B \rightarrow C_a D_2$

$B \rightarrow b$

$C_b \rightarrow b$

$D_1 \rightarrow AA$

$D_1 \rightarrow D_3 D_4$

$D_2 \rightarrow D_5 D_6$

$D_2 \rightarrow B B$

$D_3 \rightarrow a$

$D_4 \rightarrow a$

$D_5 \rightarrow b$

$D_6 \rightarrow b$

$A \rightarrow C_a S$

$B \rightarrow C_b S$

g) G' a generalization of G_s with $L(G') = L(G)$

strings less than a certain length[22] to create G_S, we simply require that the teacher provide additional information with each sample string, to indicate if this sample generalizes from previous sample strings, or is an instance of a new class of string in the language. For those strings which generalize previous sample strings we can apply the bias in favour of old non-terminal substitutions; strings which are instances of a new class of string will use the bias in favour of new non-terminals when completing the parse. The sort of additional information required about each string is an example of a felicity condition [27] for grammar induction. As an example, this heuristic was applied to the set of strings {ab, aabb, aaabbb} of which only the first was indicated as adding structure and the other two were examples of generalization. The parse completion algorithm produced the 9 candidate grammars shown in figure 18, of which grammars 2 and 6 are the interesting ones. These two grammars fail to capture exactly the target language a^nb^n, but they do capture the closely related language that consists of all strings of *a*'s and *b*'s that begin with an *a* and have an equal number of *a*'s and *b*'s. This language is only slightly more general than the target language, so the heuristic has done quite well. In fact it is easy to show that using only the string *ab* to create the structure you cannot possibly capture the target language exactly since the string *ab* introduces only 3 non-terminals into the grammar, while the smallest grammar for this language requires 4 non-terminals. However if we use the sample string *aabb* as a structural example, and the strings *ab* and *aaabbb* as generalizing examples, then the parse completion algorithm does produce a grammar for the target language a^nb^n.

The most important bias, the one towards new or old non-terminals in the RHS formats, has already been discussed in relation to the partial order of induced grammars. There is a second bias in this system, which we may regard as a bias in favour of parsimony. When a new sample string is introduced, if it can be parsed by any grammar in the existing set, these grammars are retained, and the other grammars are not considered further. This can be regarded as a bias in favour of grammars which generalize the sample strings better, or as a bias in favour of grammars with small numbers of rules.

The system at the moment contains no heuristic knowledge which allows it to prune the set of grammars in the partial order. Even though we can show that with a fixed G_S there are only a

[22] $2n - 1$ for regular languages, 2^{2n-1} for context free languages.

1. $S \to V_1 V_2$ $V_1 \to a$ $V_2 \to b$ $V_1 \to V_1 V_1$ $V_1 \to V_1 V_2$

*2. $S \to V_1 V_2$ $V_1 \to a$ $V_2 \to b$ $V_1 \to V_1 S$

3. $S \to V_1 V_2$ $V_1 \to a$ $V_2 \to b$ $V_1 \to V_1 V_2$ $V_2 \to V_1 V_2$

4. $S \to V_1 V_2$ $V_1 \to a$ $V_2 \to b$ $V_1 \to V_2 V_2$ $V_2 \to V_1 V_1$

5. $S \to V_1 V_2$ $V_1 \to a$ $V_2 \to b$ $V_1 \to S V_2$ $S \to V_1 V_1$

*6. $S \to V_1 V_2$ $V_1 \to a$ $V_2 \to b$ $V_2 \to S V_2$

7. $S \to V_1 V_2$ $V_1 \to a$ $V_2 \to b$ $V_2 \to V_1 V_2$ $V_2 \to V_2 V_2$

8. $S \to V_1 V_2$ $V_1 \to a$ $V_2 \to b$ $V_2 \to V_2 V_2$ $V_1 \to V_1 V_1$

9. $S \to V_1 V_2$ $V_1 \to a$ $V_2 \to b$ $V_2 \to V_1 S$ $S \to V_2 V_2$

Figure 18: The 9 grammars generated from the set {ab, aabb, aaabbb}.

finite number of grammars in the partial order, this finite number may be ve:y large. For the case of the CNF grammars if our G_S has k non-terminals we know there are at most $k^3 + k|\Sigma|$ rules that can appear in any grammar in the partial order[23], but as a grammar can contain any subset of these rules this still means there are on the order of $2^{k^3+k|\Sigma|}$ different grammars in the entire partial order. This makes the construction of the entire partial order infeasible for all but very small grammars. Experience has shown that generally a grammar for the target language can be found by exploring far less than the entire partial order. But the examples presented in this paper have been of very simple grammars precisely because an unrestricted exploration of the partial order is very expensive. To build an algorithm for practical problems parse completion will have to be augmented with additional heuristics to control its search. The advantage of having the basic algorithm as a base to work from is that the effects of particular heuristics can now be measured in terms of the complete partial order explored by the unrestricted algorithm.

One obvious and domain independant heuristic for CNF grammars has been suggested by the observations made in the proof that a bound for G_S exists. In this proof it was noted that the sub-trees contained in G_S serve as recursive building blocks for the derivations of longer strings. Currently these building blocks are tried in an arbitrary order. However, by examining the yields of these sub-trees, and selecting the sub-tree whose yield is closest to the substring that is being

[23] There are at most k^3 rules of the form $A_i \to A_j A_k$, and $k|\Sigma|$ of the form $A_i \to a_j$.

parsed, a form of best first search could be implemented. The next stage in the development of parse completion should be a systematic exploration of heuristics such as these to make the algorithm more efficient.

7 Conclusion

Our purpose was to explore the class of Parse Completion algorithms. In pursuing that purpose, we have produced a well defined design space for this class of algorithms. This design space is defined by a partial order over the RHS formats of new rules which may be added to complete a parse. One of the most interesting divisions based on RHS formats divided rules into those which added additional structure to the grammar, and those which generalized existing structure. This particular division led to the discovery of biases under which an induction algorithm can be designed which will always converge to a single, most specific, context free grammar from a finite set of positive example strings. Certain additional conditions must also be met to guarantee convergence. These conditions can be expressed as a complexity bound on the language, which uniquely identifies the point at which no additional structure needs to be added to the grammar. These conditions can also be expressed as felicity conditions, which require the teacher to distinguish examples that introduce a new concept from examples that only serve to generalize existing concepts. Perhaps the most important point to be learned from this study is that a systematic attempt to understand an induction *domain* can lead to useful insights for designing induction algorithms for that domain.

References

[1] Anderson, J. R.
The Architecture of Cognition.
Harvard University Press, Cambridge, MA., 1983.

[2] Anderson, J. R.
Skill Acquisition: Compilation of Weak-Method Problem Solutions.
Technical Report ONR-85-1, Office of Naval Research, 1985.

[3] Biermann, A. W.
On the Inference of Turing Machines from Sample Computations.
Artificial Intelligence (3):181-198, 1972.

[4] Biermann, A. W.
The Inference of Regular Lisp Programs from Examples.
IEEE Transactions on Systems, Man, and Cybernetics SMC-8(8), August, 1978.

[5] Biermann, A. W. and Feldman, J. A.
A survey of results in grammatical inference.
In S. Watanabe (editor), *Frontiers of pattern recognition.* Academic Press, New York, 1972.

[6] Cohen, Daniel I. A.
Basic Techniques of Combinatorial Theory.
John Wiley & Sons, New York, 1978.

[7] Fahlman, S.
A Planning System for Robot Construction Tasks.
Artificial Intelligence (5):1-49, 1974.

[8] Fikes, Hart, and Nilsson.
Learning and Executing Generalized Robot Plans.
Artificial Intelligence (3):251-288, 1972.

[9] Fu, K. and Booth, T.
Grammatical Inference: Introduction and survey.
IEEE Transactions on Systems, Man, and Cybernetics (5):95-111, 1975.

[10] Genesereth, M. R.
The role of plans in intelligent teaching systems.
In *Intelligent Tutoring Systems.* Academic Press, New York, 1982.

[11] Hopcroft, J. E. and Ullman, J. D.
Introduction to Automata Theory, Languages, and Computation.
Addison Wesley, Reading, MA., 1979.

[12] Horning, J. J.
A study of grammatical inference.
Technical Report CS-139, Stanford University, Computer Science Department, 1969.

[13] Knoblock, C. and Carbonell, J.
Learning by Completing Plans.
Technical Report, Computer Science Department, Carnegie-Mellon University, In Preparation.

[14] Laird, J., Rosenbloom, P. S. and Newell, A.
Chunking in Soar: The anatomy of a general learing machine.
Technical Report, Computer Science Department, Carnegie-Mellon University, 1985.

[15] Minsky, M.
A framework for representing knowledge.
In P. Winston (editor), *The psychology of computer vision*, pages 211-277. McGraw-Hill, New York, 1975.

[16] Mitchell, T. M.
The need for biases in learning generalizations.
Technical Report CBM-TR-117, Rutgers University Computer Science Department, 1980.

[17] Mitchell, T. M.
Generalization as search.
Artificial Intelligence (18):203-226, 1982.

[18] Newell, A., Shaw, J. C., and Simon, H. A.
Report on a general problem-solving program for a computer.
In *Information Processing: Proceedings of the International Conference on Information Processing*, pages 256-264. UNESCO, Paris, 1960.

[19] Osherson, D., Stob, M. and Weinstein, S.
Systems that Learn.
MIT Press, Cambridge, MA, 1985.

[20] Pao, T. W.
A solution of the syntactical induction-inference problem for a non-trivial subset of context-free languages.
Interim Report 69-19, Moore School of Electrical Engineering, University of Pennsylvania, 1969.

[21] Quillian, M. R.
Semantic Memory.
Semantic Information Processing.
MIT Press, Cambridge, Mass., 1968.

[22] Riesbeck, C. K.
Failure-driven reminding for incremental learning.
In *Proceedings of IJCAI 7*, pages 115-120. 1981.

[23] Sacerdoti, E.
Planning in a Hierarchy of Abstraction Spaces.
In *International Joint Conference on Artificial Intelligence 3*, pages 412-422. 1973.

[24] Schank, R. C., and Abelson, R. P.
Scripts, plans, goals and understanding.
Lawrence Erlbaum, Hillsdale, N.J., 1977.

[25] Schank, R.
Dynamic Memory: A Theory of Learning in Computers and People.
Cambridge University Press, Cambridge, 1982.

[26] VanLehn, K.
Human procedural skill acquisition: Theory, model and psychological validation.
In *Proceedings of AAAI-83.* Los Altos, CA, 1983.

[27] VanLehn, K.
Learning one subprocedure per lesson.
Artificial Intelligence 31(1):1-40, January, 1987.

[28] VanLehn, K.
Towards a Theory of Impasse Driven Learning.
In H. Mandel and A. Lesgold (editors), *Advances in Intelligent Teaching Systems Research.* Academic Press, New York, In Preparation.

[29] VanLehn, K. and Ball, W.
A Version Space Approach to Learning Context Free Grammars.
Machine Learning Journal , In Press.